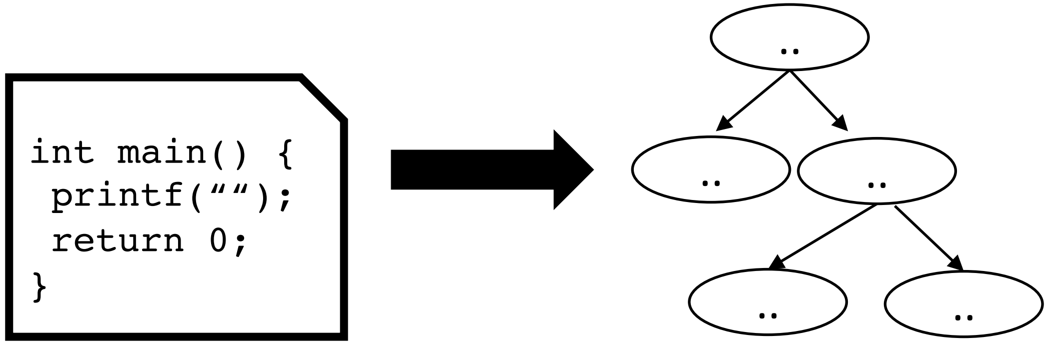
CSE211: Compiler Design

Sept. 27, 2022



```
int main() {
 printf("");
 return 0;
}
```

- **Topic**: Parsing
- **Questions**:
  - *What is parsing?*
  - *Have you used Regular Expressions before?*
  - *How do you parse Regular Expressions? What about Context-free Grammars?*

Enrollment

- Got most people in!
    - but class is closed now

- Class is up to 30 with an original cap of 19. This changes things!
    - We have some undergrad graders to help
    - Office hours will likely be fairly busy. There will not be additional tutoring available.
    - Assignments will done in pair programming.
        - This is not just to make assignments easier or grading easier, but for you to tutor each other!
        - Please pick a new partner for each assignment
        - Any programming work towards the assignment must be done together!! (zoom with shared screen is fine).

Attendance

- Please mark yourself present on the dates you attend
  - I’ll send out a google sheet later today
- Please don’t cheat.
- If you have questions, just ask!

Piazza

- I have a piazza set up, I will email out the link later

New students

- If you were not here the first day of class, please watch the recording.

Office hours

- Moved to Friday this week

Review

- What is a compiler?

Review

- What is a compiler?

- Besides translations, what else can a compiler do?

Review

- What is a compiler?

- Besides translations, what else can a compiler do?
  - analysis
  - optimizations

What can happen when the Input isn’t valid?

```
int my_var = 5;
my_var = my_car + 5;
```

```
int foo() {
  int *x = malloc(100*sizeof(int))
  return x[100];
}
```

What about this one?

Can the compiler make your code go faster?

```
int my_var = 0;
for (int i = 0; i < 128; i++) {
   my_var++;
}
```

Try running this on https://godbolt.org/
change the optimization level to -O3 and see what happens!

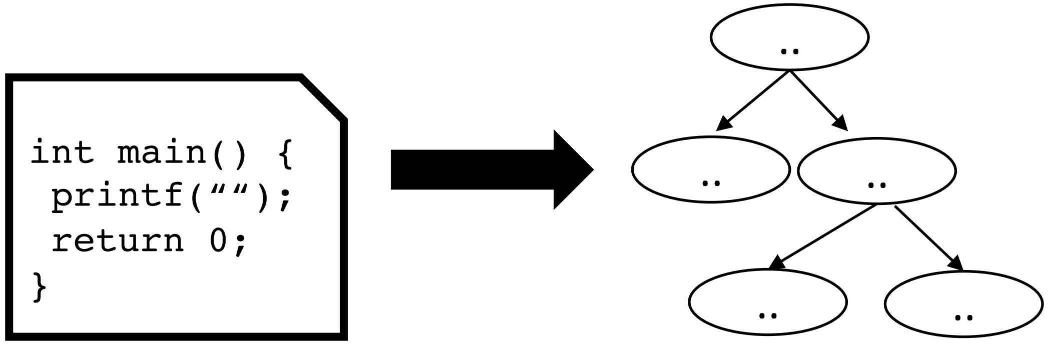
What is the compiler allowed to do?

```
void add_arrays(int *a, int *b)
for (int i = 0; i < 128; i++) {
   a[i] += b[i];
}
```

Try running this on https://godbolt.org/
change the optimization level to -O3 and see what happens!
Look for instructions like `paddd`. what does it do?

CSE211: Compiler Design

Sept. 27, 2022

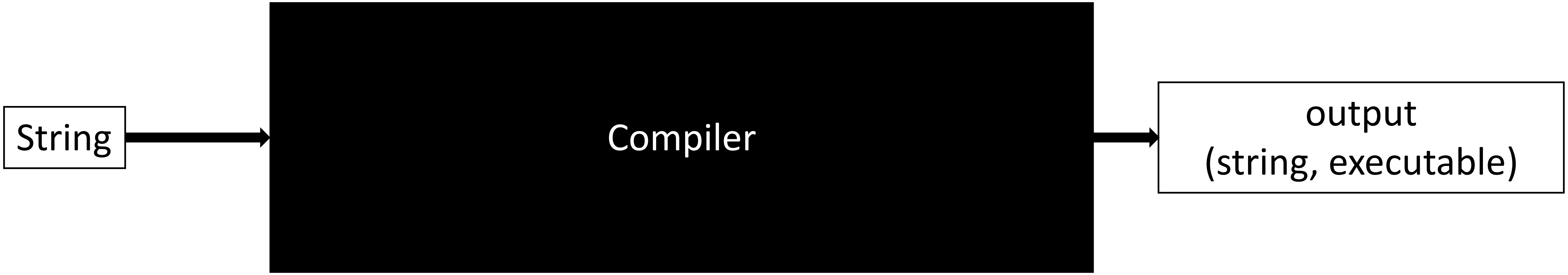


```
int main() {
 printf(“”);
 return 0;
}
```

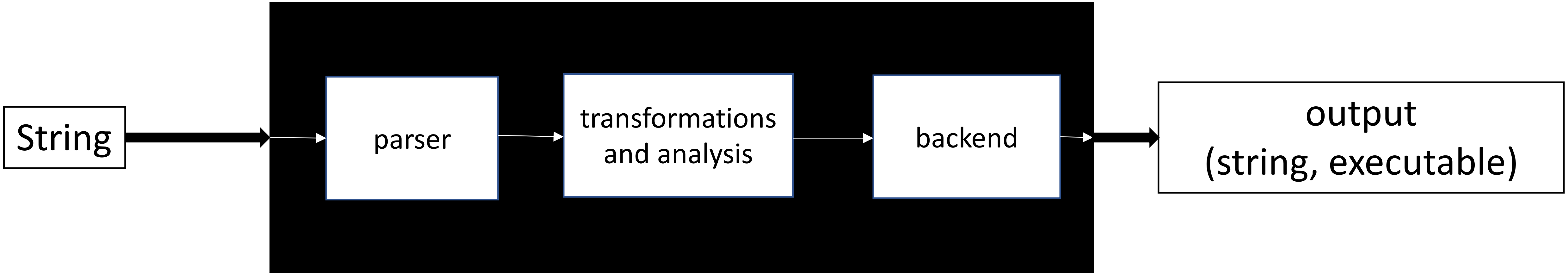
- **Topic**: Parsing

- **Questions**:
  - *What is parsing?*
  - *Have you used Regular Expressions before?*
  - *How do you parse Regular Expressions? What about Context-free Grammars?*

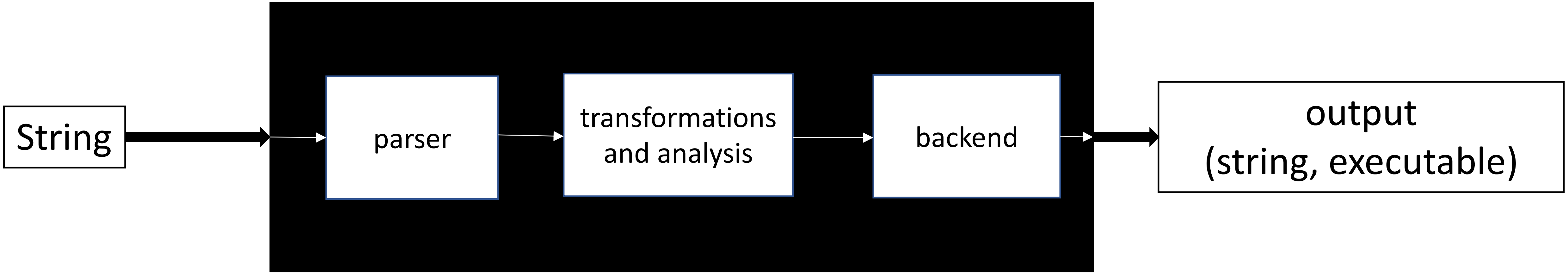
Compiler architecture overview



Compiler architecture overview

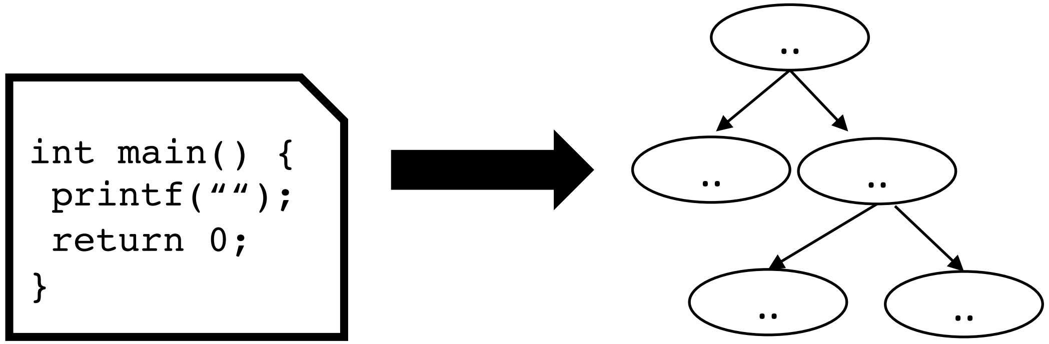


Compiler architecture overview



Parsing is the first step in the compiler

Creates structure



```
int main() {
 printf("");
 return 0;
}
```

Parsing is the first step in a compiler

- How do we parse a sentence in English?

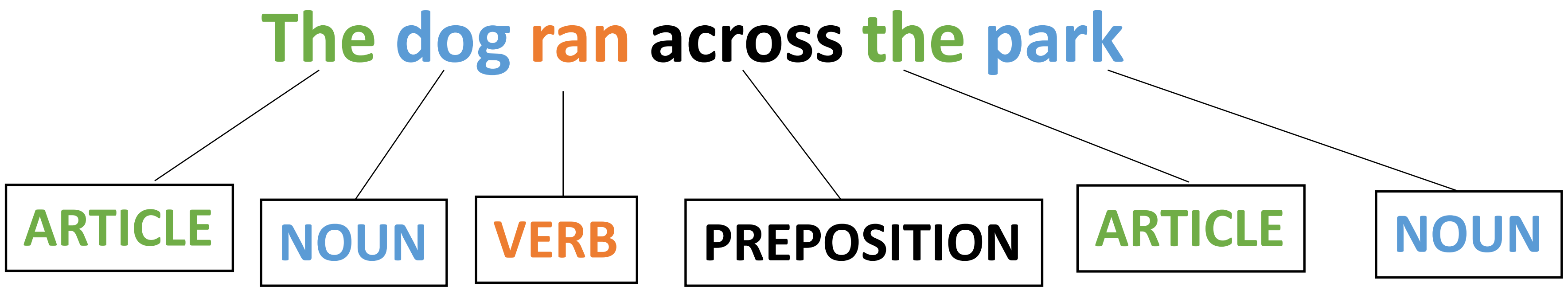
Parsing is the first step in a compiler

- How do we parse a sentence in English?

The dog ran across the park

Parsing is the first step in a compiler

- How do we parse a sentence in English?

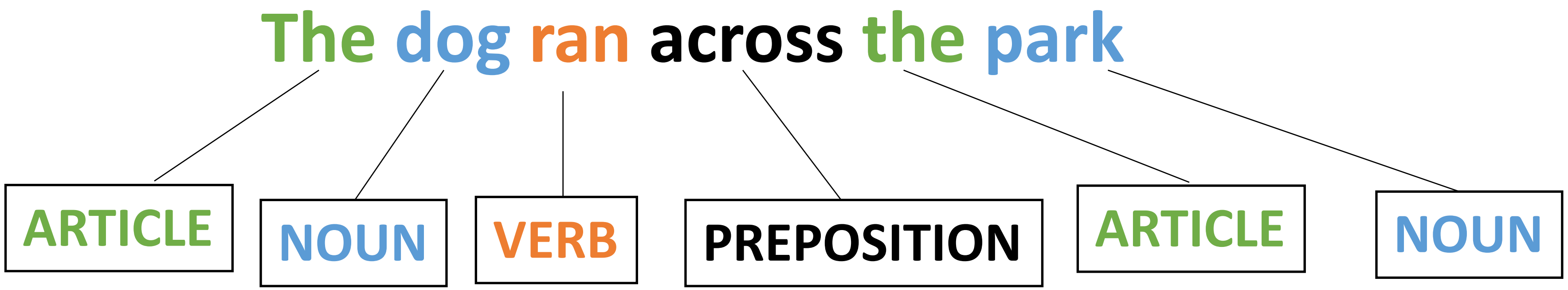


The dog ran across the park

ARTICLE NOUN VERB PREPOSITION ARTICLE NOUN

Parsing is the first step in a compiler

- How do we parse a sentence in English?

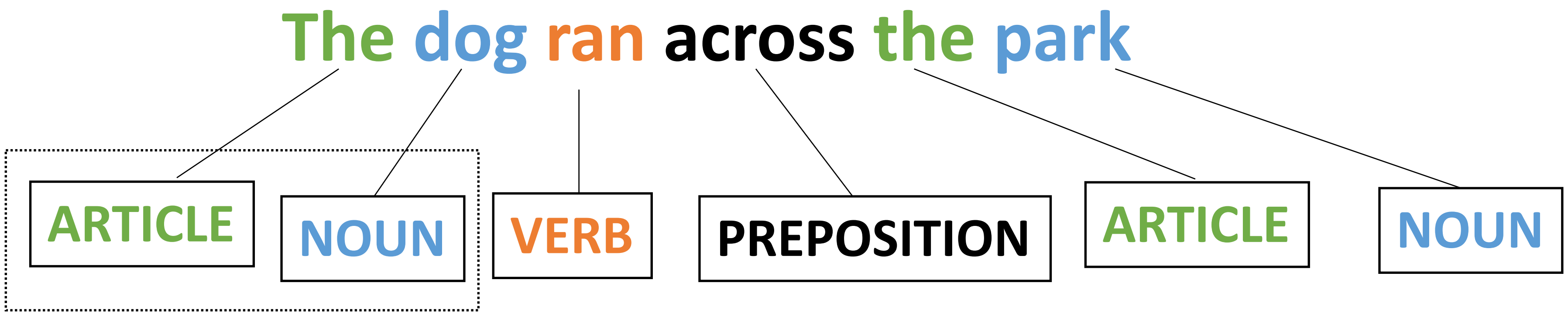


Grammar and Syntax

What about semantics?

Parsing is the first step in a compiler

- How do we parse a sentence in English?

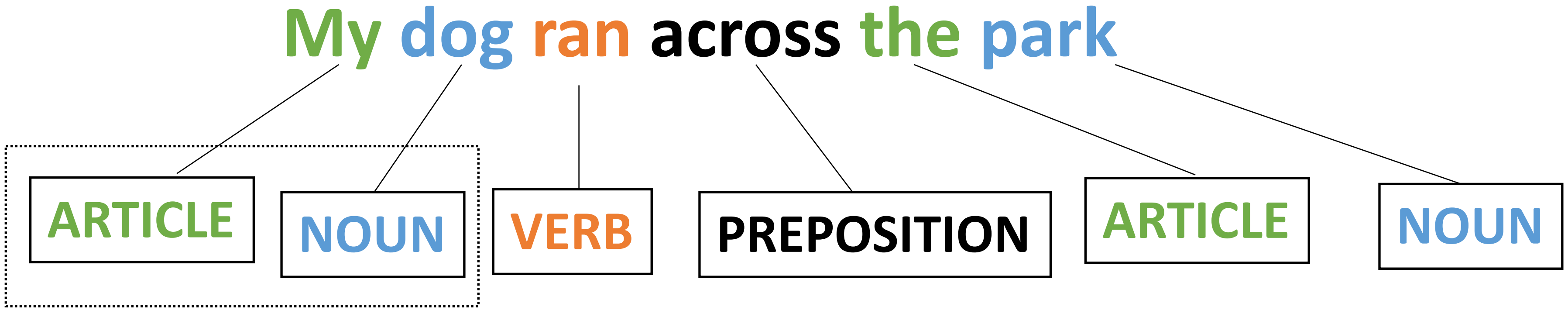


Grammar and Syntax

What about semantics?

Parsing is the first step in a compiler

- How do we parse a sentence in English?



Grammar and Syntax

What about semantics?

New Question

Can we define a simple language using these building blocks?

- ARTICLE
- NOUN
- VERB
- ADJECTIVE

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

ARTICLE NOUN VERB

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

ARTICLE ADJECTIVE NOUN VERB

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

Question mark means optional

ARTICLE ADJECTIVE? NOUN VERB

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

ARTICLE	ADJECTIVE?	NOUN	VERB
My	Old	Computer	Crashed

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

ARTICLE	ADJECTIVE?	NOUN	VERB
The	Purple	Dog	Crashed

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

Syntactically correct,
logically correct?

ARTICLE	ADJECTIVE?	NOUN	VERB
The	Purple	Dog	Crashed

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

What other sentences can you construct?

ARTICLE ADJECTIVE? NOUN VERB

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

What other languages can you specify?

ARTICLE ADJECTIVE* NOUN VERB

Goals in this module

- **Understand** the architecture of a modern parser (*tokenizing and parsing*)
- **Understand** the language of tokens (*regular expressions*) and parsers (*context-free grammars*)
- How to **design** CFG production rules so avoid **ambiguity** and encode *precedence and associativity*.
- **Utilize** a classic parser generator (*Lex and Yacc*) for a simple language

Goals in this module

- We will **NOT** discuss parsing algorithms for CFGs. If you are interested, you can do this for a paper assignment.

- This module should provide you with the background to implement parsers, which are **USEFUL** in many different projects.

- These topics are typically covered in more depth in an undergrad course.

High-level parser

Parser

High-level parser

A parser needs to know about the language:

- What forms can these take?

Parser

High-level parser

A parser needs to know about the language:

- 1800 page C++ specification,
  - English language
- Formal specification, mathematical
  - Mostly used in academics
  - X86, ARM, Functional languages

High-level parser

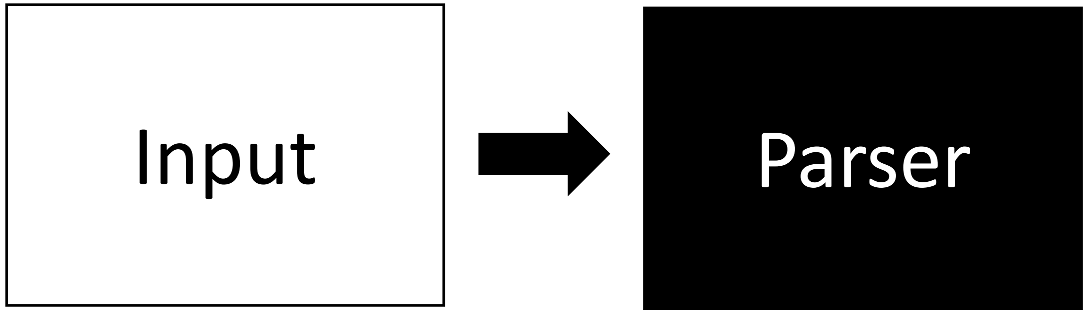
A parser needs to know about the language:

- 1800 page C++ specification,
  - English language
- Formal specification, mathematical
  - Mostly used in academics
  - X86, ARM, Functional languages

Parser

Parser needs only a small part of the specification!
*The **Grammar!***

High-level parser



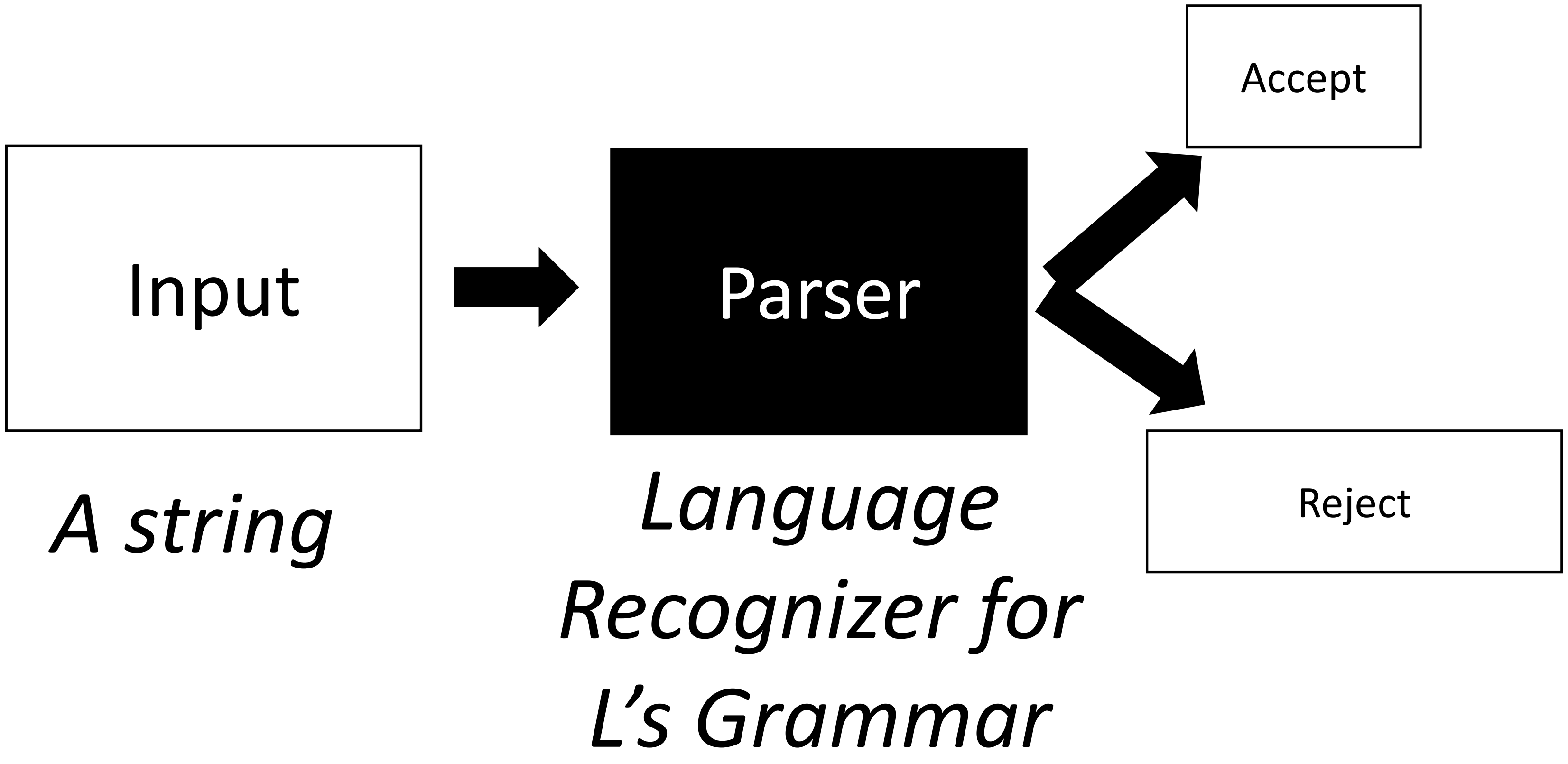
Input

Parser

A string

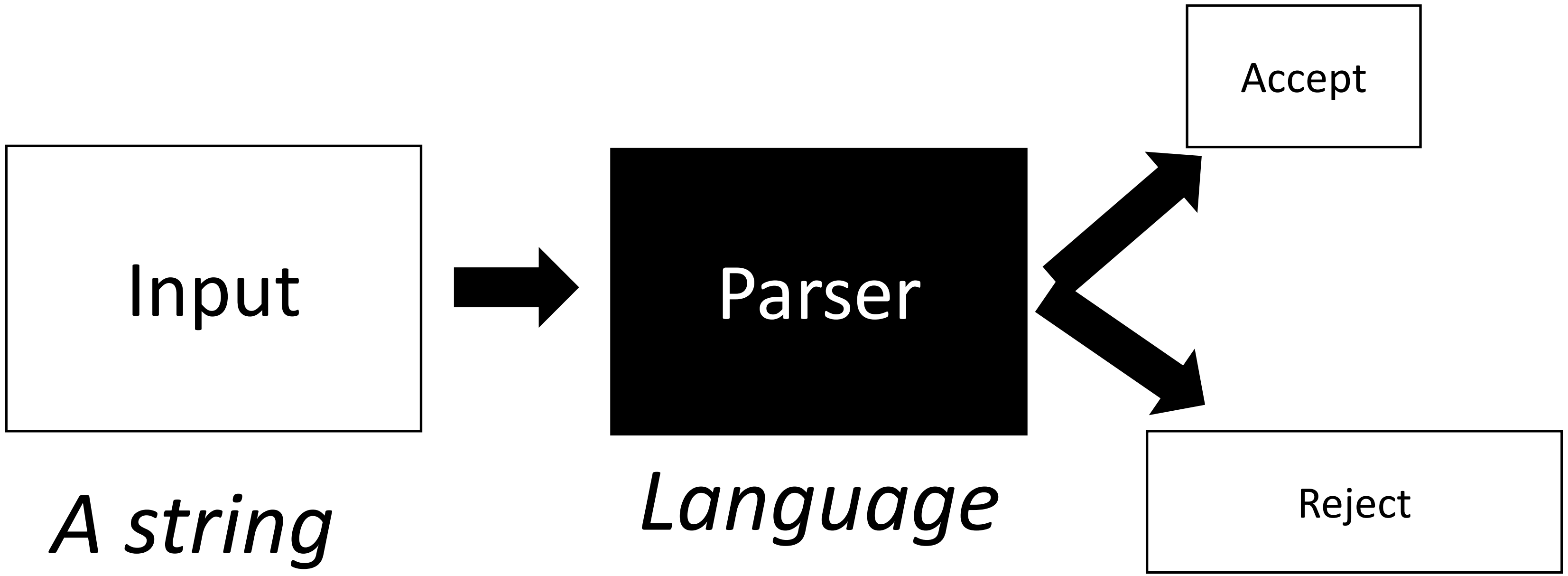
Language Recognizer for L’s Grammar

High-level parser



High-level parser

The input string
satisfies *L's grammar*



Input

Parser

Accept

Reject

A string

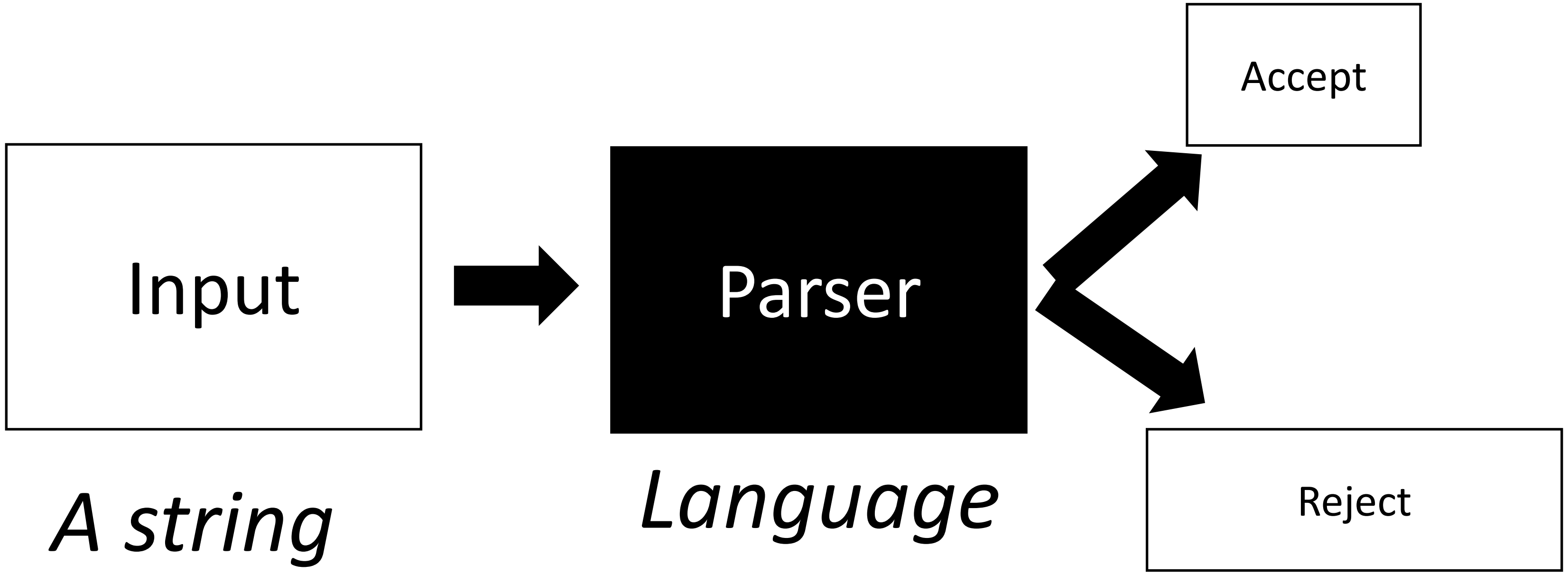
Language Recognizer for L's Grammar

The input string is ***NOT***
in the language L

High-level parser

what other types of errors might happen up here?

The input string ***satisfies*** *L's grammar*



A string

Language Recognizer for L's Grammar

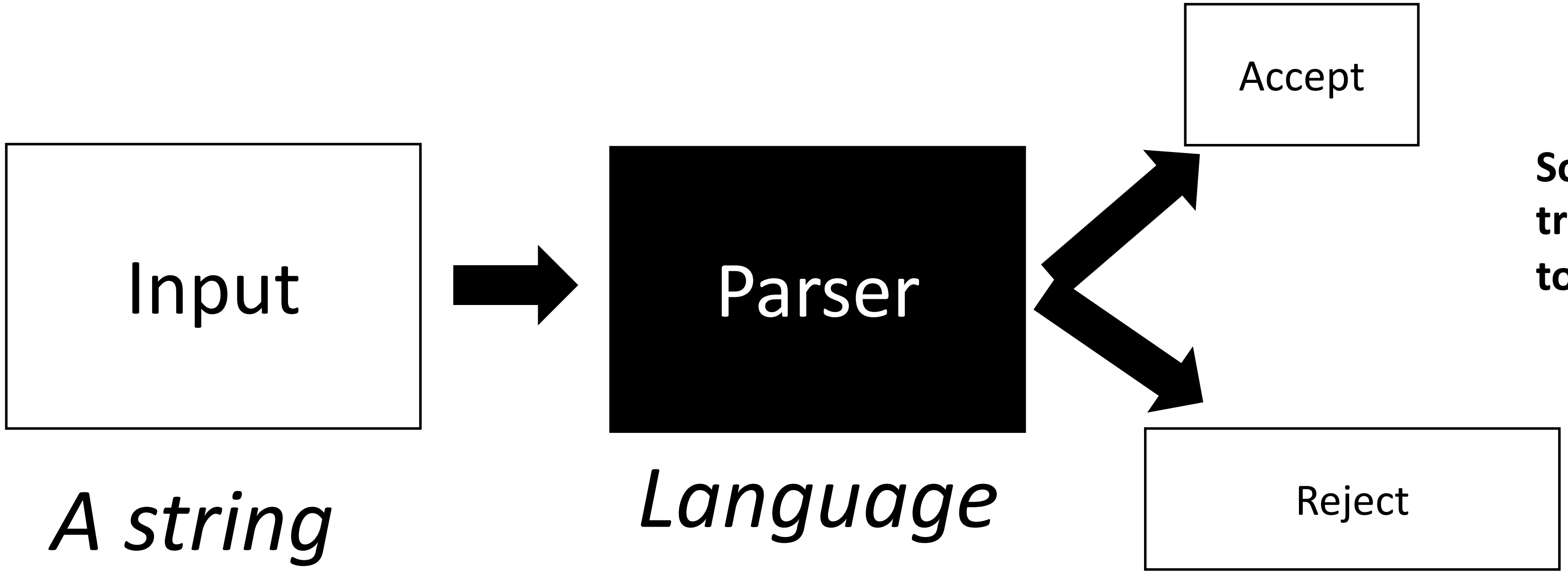
Syntax error

The input string is ***NOT*** *in the language L*

High-level parser

what other types of errors might happen up here?

The input string ***satisfies*** *L’s grammar*



Input

Parser

Accept

Reject

Some languages try to move logic errors to syntax errors!

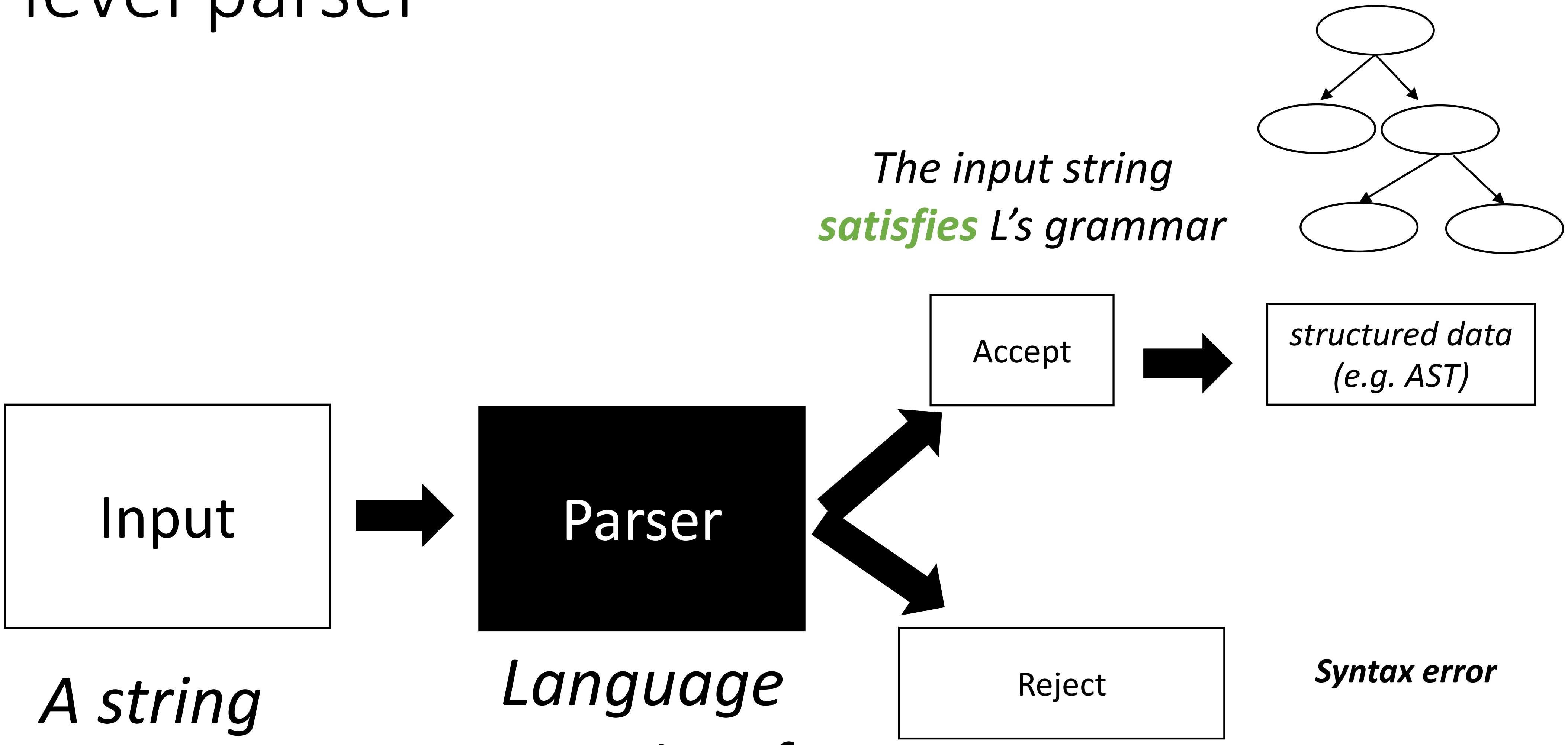
Syntax error

A string

Language Recognizer for L’s Grammar

The input string is ***NOT*** *in the language L*

High-level parser



continue to the rest of compilation

The input string ***satisfies*** *L’s grammar*

Accept

structured data (e.g. AST)

Input

Parser

Reject

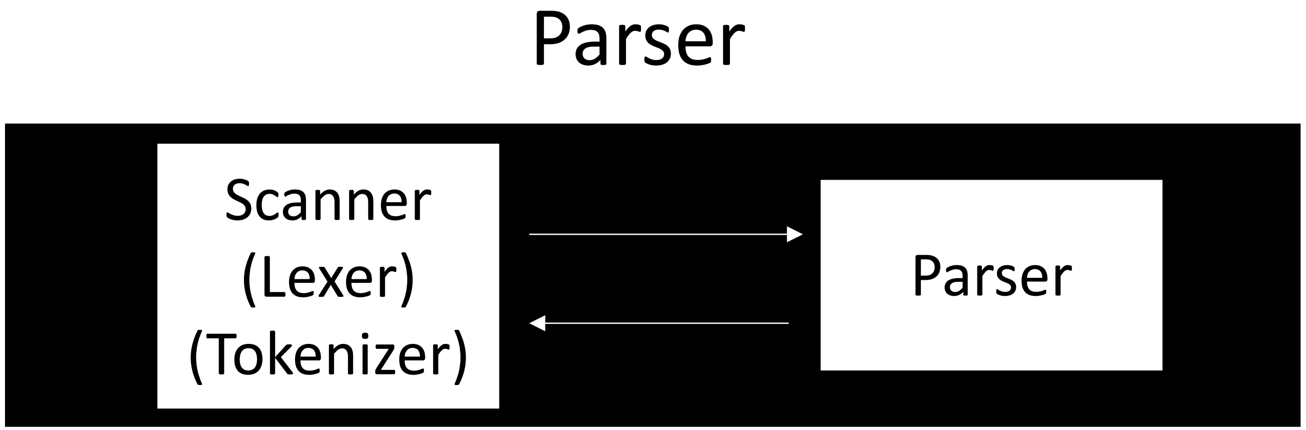
Syntax error

A string

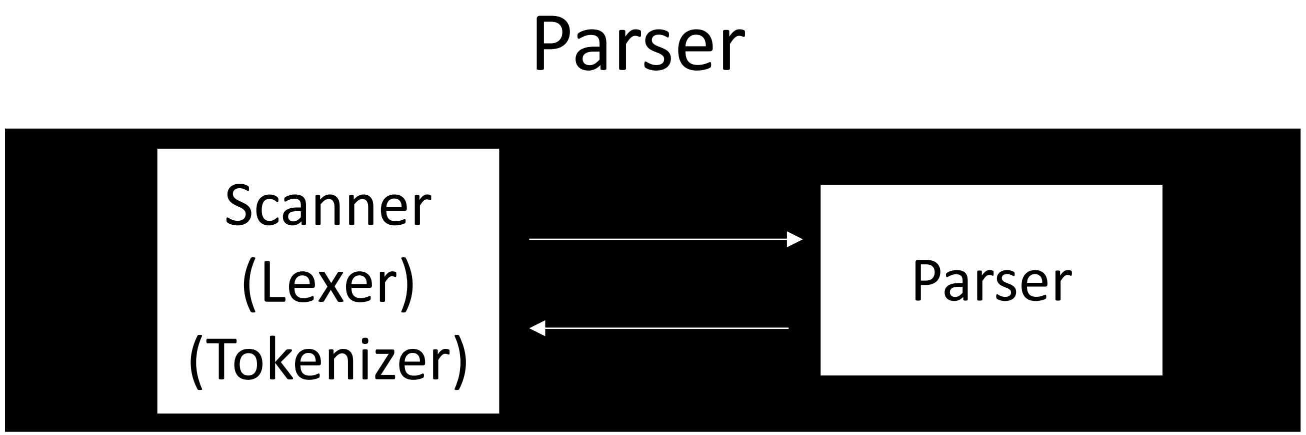
Language Recognizer for L’s Grammar

The input string is ***NOT*** *in the language L*

Parser architecture



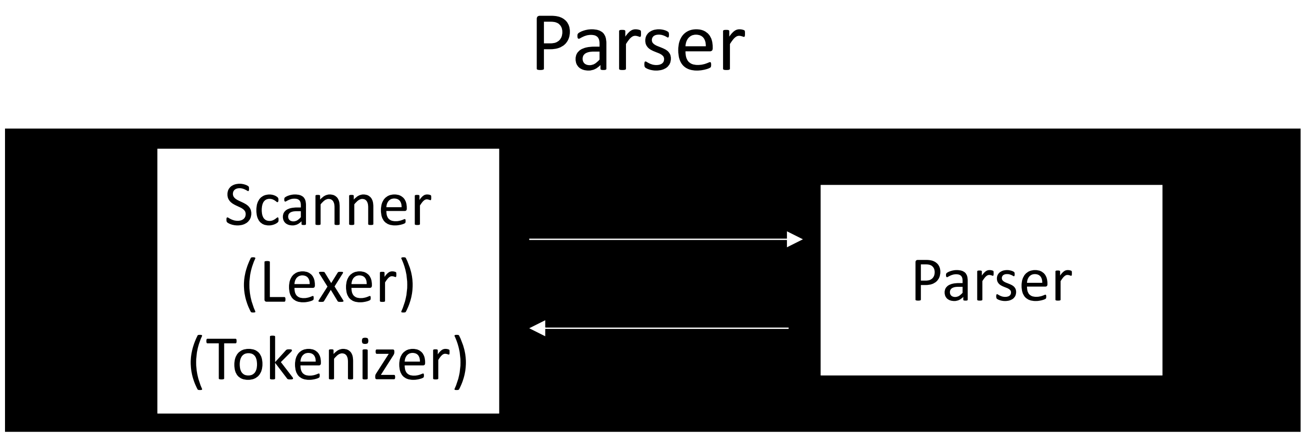
Parser architecture



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

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

Parser architecture



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

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

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

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

Scanner

- List of tokens:
- e.g. {NOUN, ARTICLE, ADJECTIVE, VERB}

Scanner

My Old Computer Crashed

Scanner

My Old Computer Crashed

Scanner

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

Scanner

My Old Computer Crashed

↓ Scanner

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

***Lexeme**: (TOKEN, value)*

Scanner

- Lets write tokens for arithmetic expression:

(5 + 4) * 3

ideas?

Scanner

- Lets write tokens for arithmetic expression:

LPAREN = ‘(‘
NUMBER = {‘5’,’4’,’3’, ..}
PLUS = ‘+’
RPAREN = ‘)’
TIMES = ‘*’

(5 + 4) * 3

Scanner

- Lets write tokens for arithmetic expression:

(5 + 4) * 3

LPAREN = ‘(‘
NUMBER = {‘5’,’4’,’3’, ..}
PLUS = ‘+’
RPAREN = ‘)’
TIMES = ‘*’

LPAREN = ‘(‘
NUMBER = {‘5’,’4’,’3’, ..}
OP = {‘+’, “*”}
RPAREN = ‘)’

You can generalize tokens

Scanner

- Lets write tokens for arithmetic expression:

LPAREN = ‘(‘
NUMBER = {‘5’,’4’,’3’, ..}
PLUS = ‘+’
RPAREN = ‘)’
TIMES = ‘*’

(5 + 4) * 3

LPAREN = ‘(‘
ONE = ‘1’
TWO = ‘2’
THREE = ‘3’
...
PLUS = ‘+’
RPAREN = ‘)’
TIMES = ‘*’

You can make tokens more specific

Scanner

- Lets write tokens for arithmetic expression:

LPAREN = ‘(‘
NUMBER = {‘5’,’4’,’3’, ..}
PLUS = ‘+’
RPAREN = ‘)’
TIMES = ‘*’

(5 + 4) * 3

PAREN = {‘(‘, ‘)’}
NUMBER = {‘5’,’4’,’3’, ..}
PLUS = ‘+’
TIMES = ‘*’

What about this one?

Defining tokens

Defining tokens

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

Defining tokens

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

- Keyword – single string:
    - IF = “if”, INT = “int”

Defining tokens

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

- Keyword – single string:
  - IF = “if”, INT = “int”

- Sets of words:
  - NOUN = {“Cat”, “Dog”, “Car”}

Defining tokens

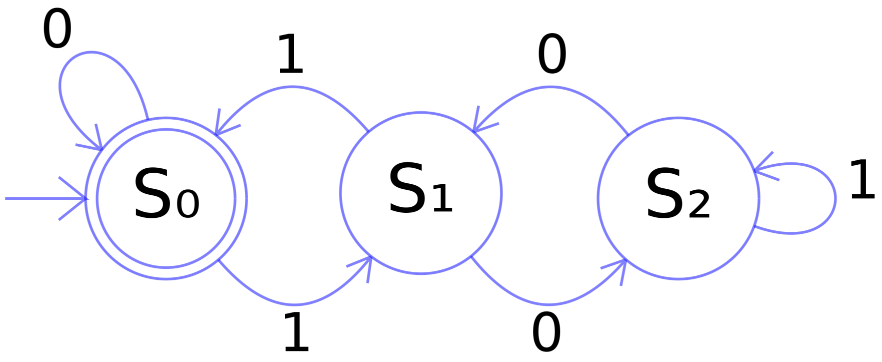
- Literal – single character:
  - PLUS = ‘+’, TIMES = ‘*’
- Keyword – single string:
  - IF = “if”, INT = “int”
- Sets of words:
  - NOUN = {“Cat”, “Dog”, “Car”}
- Numbers
  - NUM = {“0”, “1” …}

Defining tokens

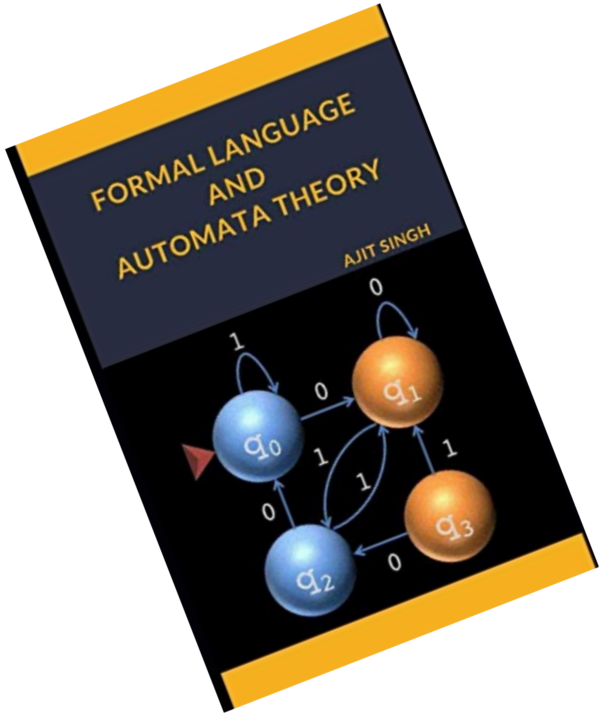
- ~~Literal – single character:~~
  - ~~PLUS = ‘+’, TIMES = ‘*’~~
  - ~~–~~
- ~~Keyword – single string:~~
  - ~~IF = “if”, INT = “int”~~
  - ~~–~~
- ~~Sets of words:~~
  - ~~NOUN = {“Cat”, “Dog”, “Car”}~~
  - ~~–~~
- ~~Numbers~~
  - ~~NUM = {“0”, “1” …}~~

- Regular expressions!

Regular Expressions

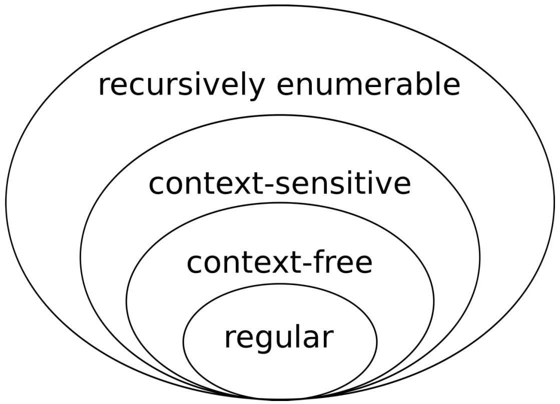


- Lots of literature!
  - Simplest grammar in the Chomsky language hierarchy
  - abstract machine definition (finite automata)
  - Many implementations (e.g. Python standard library)



FORMAL LANGUAGE AND AUTOMATA THEORY

AJIT SINGH



recursively enumerable

context-sensitive

context-free

regular

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’

Examples: (demo)

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

Examples (demo)

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

Examples (demo)

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

Examples (demo)

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 () liberally 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 = ‘*’

- Keyword – single string:
  - IF = “if”, INT = “int”

- Sets of words:
  - NOUN = “(Cat)|(Dog)|(Car)”

- Numbers
  - SINGLE_NUM = [0-9]
  - how to do INT = -?([1-9][0-9]*) | 0
  - 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

Scanner

LPARAN = ‘(‘
NUMBER = ‘[0-9]+’
PLUS = ‘+’
RPAREN = ‘)’
TIMES = ‘*’

“(5 + 4) * 3”

Scanner

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 Questions?

- 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 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?

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

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?
  - 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 lets 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 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
  - …

Define a full language using tokens?

- Where are we going to run into issues?

What about ()’s

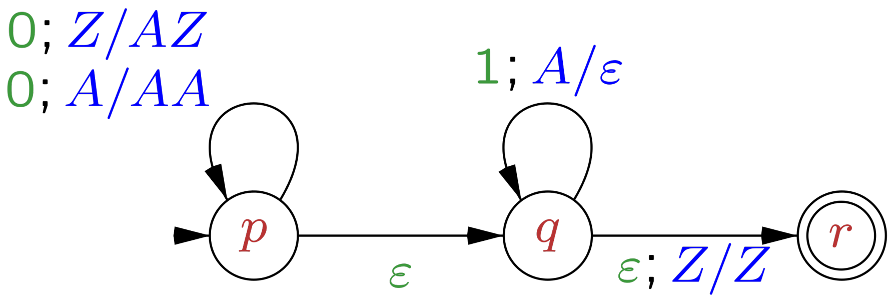
- there is a formal proof available that regex CANNOT match ()’s: pumping lemma

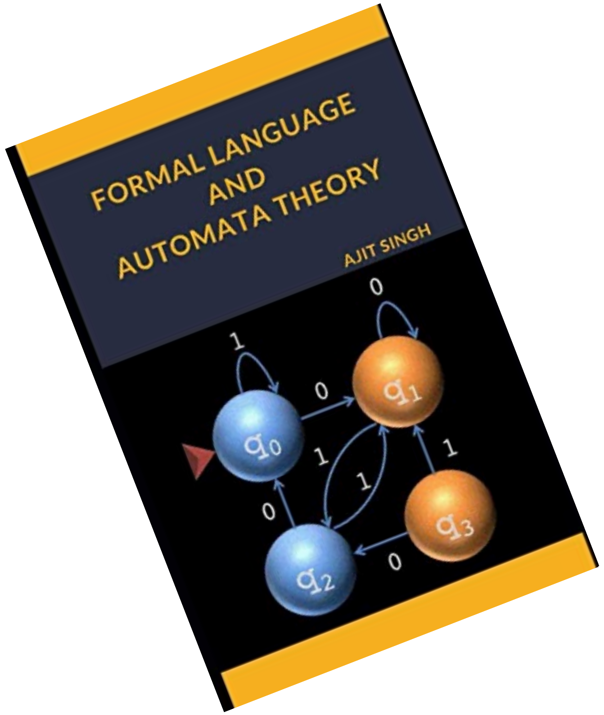
- Informal argument:
  - Try matching $(^n)^n$using Kleene star
  - Impossible!

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

Context Free Grammars

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





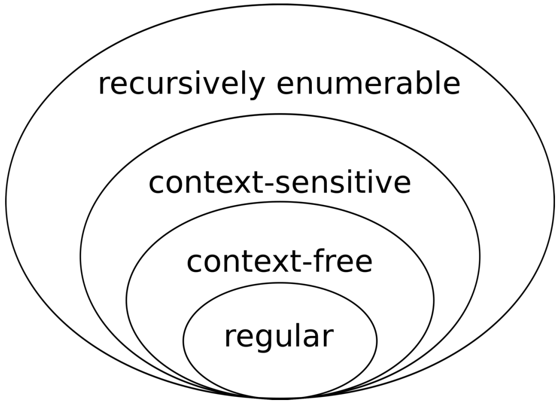


Image Credit:
By Jochgem - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=5036988

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
```

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

(previously) 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

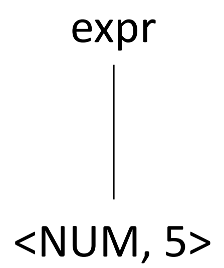
root of the tree is the entry production

Parse trees

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

```
input: 5
```

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



expr

<NUM, 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
```

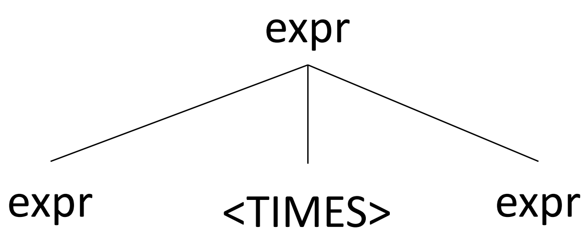
expr

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`

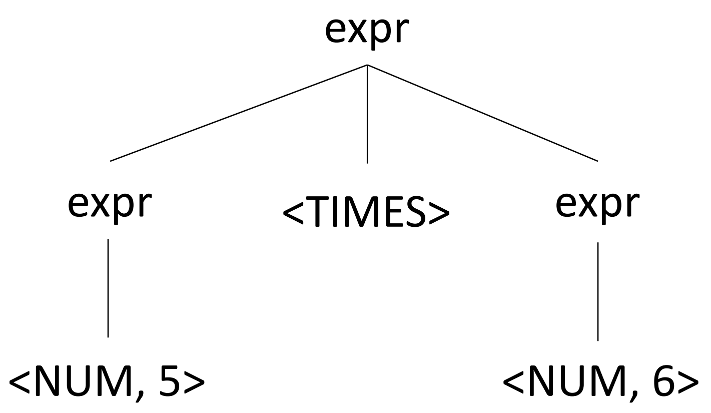


Parse trees

- 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`



expr

expr

<TIMES>

expr

<NUM, 5>

<NUM, 6>

Parse trees

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

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

What happens in an error?

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

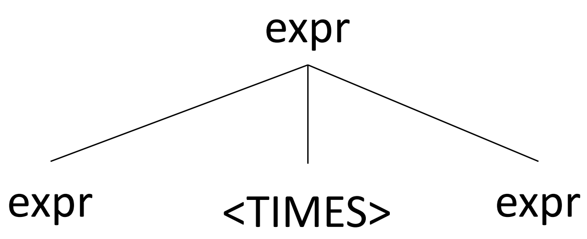
expr

Parse trees

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

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

`input: 5**6`



What happens in an error?

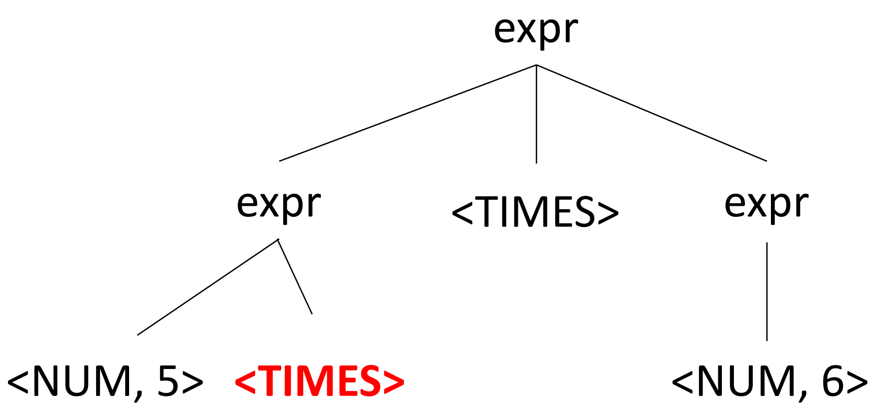
Parse trees

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

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

`input: 5**6`

What happens in an error?



Not possible!

Parse trees

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

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

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

Parse trees

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

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

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

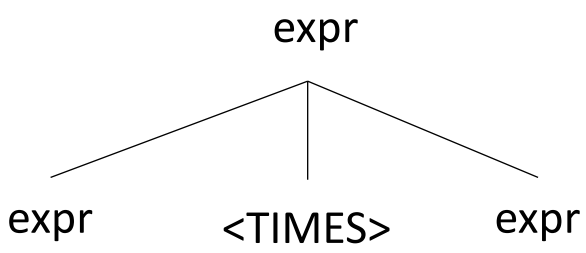
expr

Parse trees

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

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

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

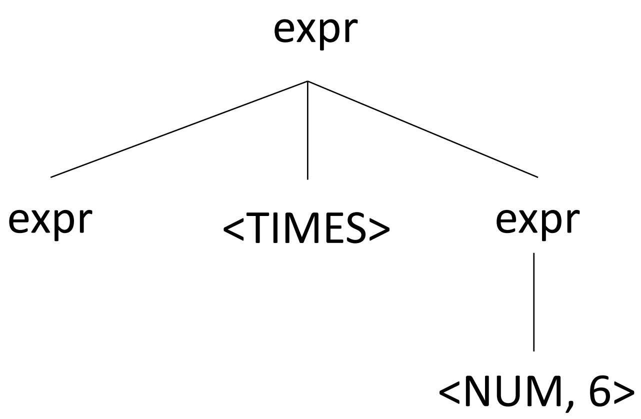


Parse trees

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

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

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

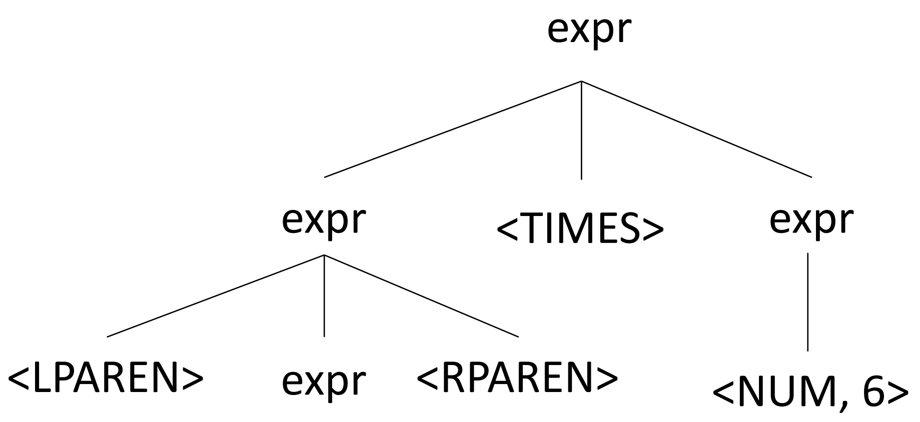


Parse trees

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

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

`input: (1+5)*6`

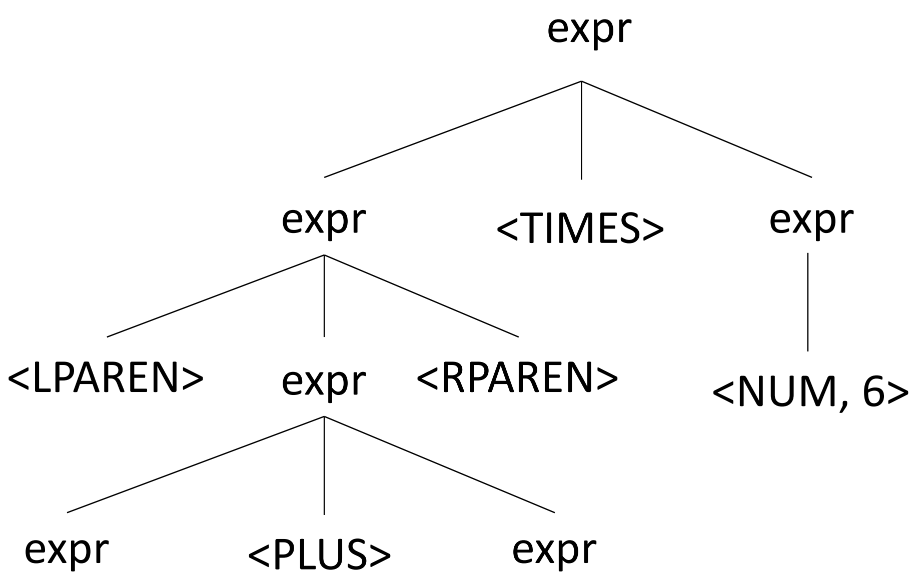


Parse trees

- 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.

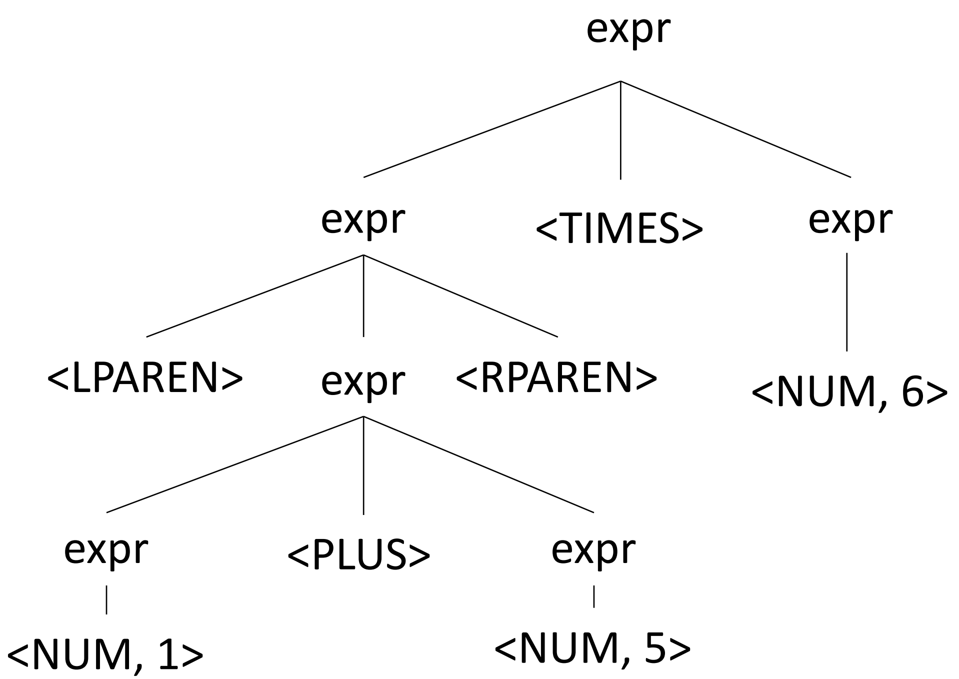
```
input: (1+5)*6
```

expr : NUM

| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN

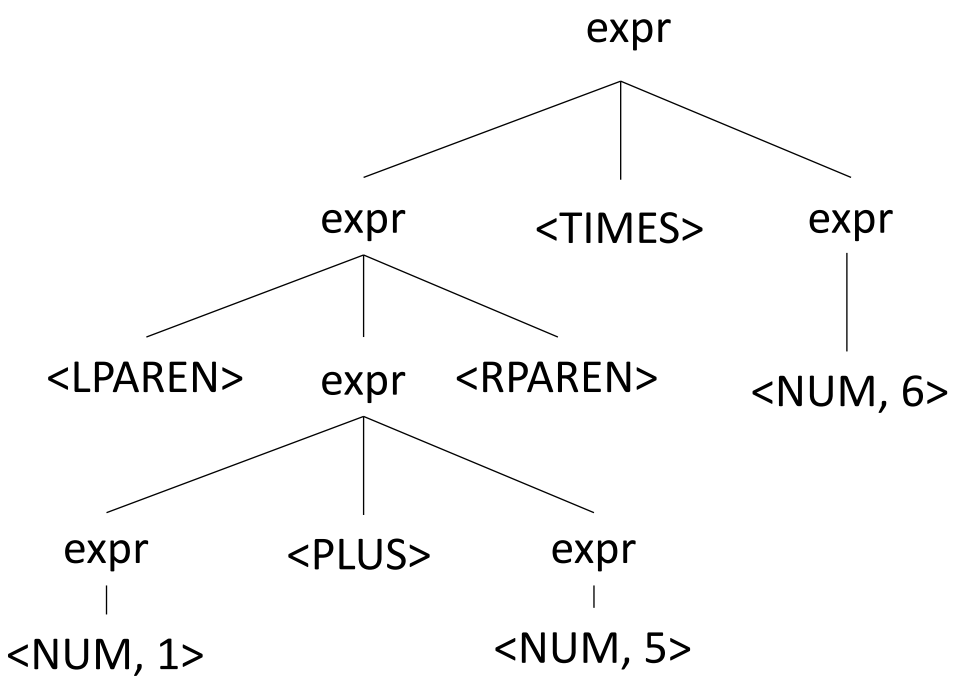


Parse trees

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

```
input: ?
```

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



expr

expr <TIMES> expr

<LPAREN> expr <RPAREN>

<NUM, 6>

expr <PLUS> expr

<NUM, 1>

<NUM, 5>

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

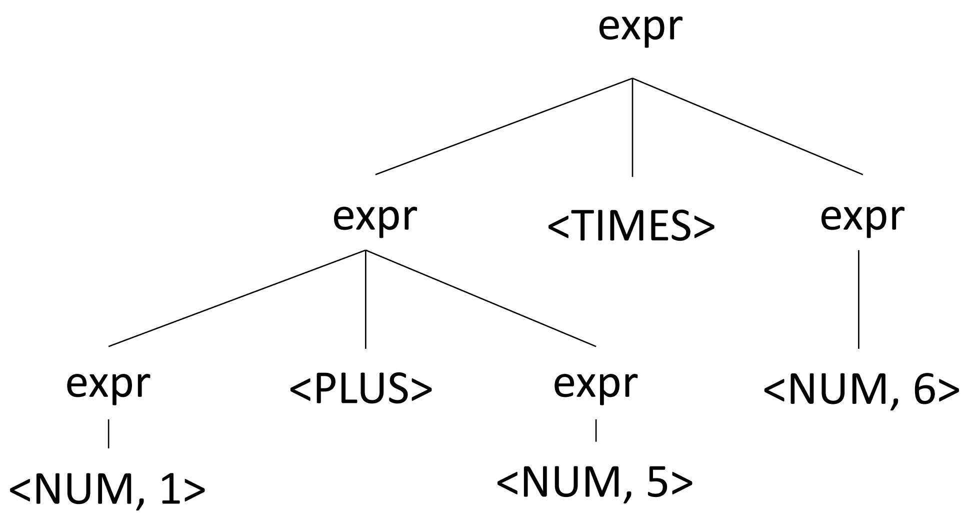
- Try making a parse tree from: `1 + 5 * 6`

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

Parse trees

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

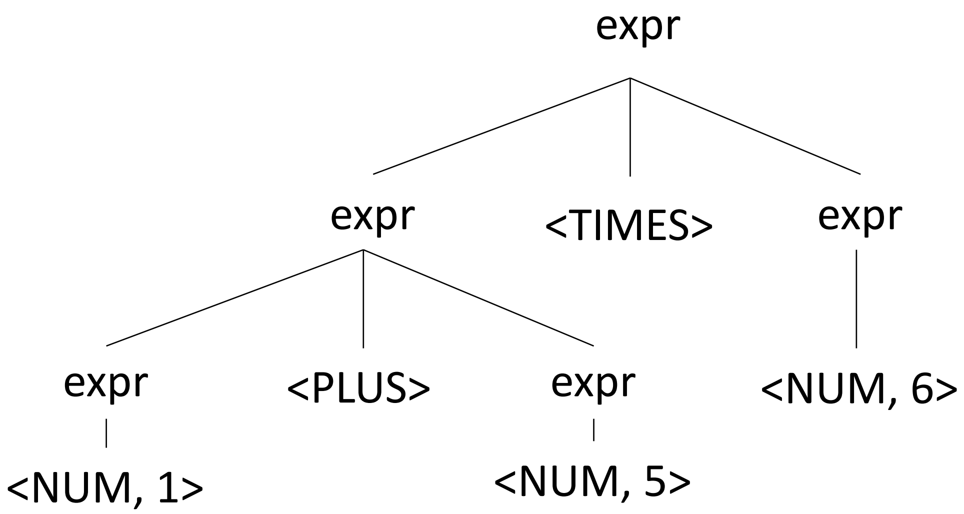
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



Parse trees

- `input: 1 + 5 * 6`

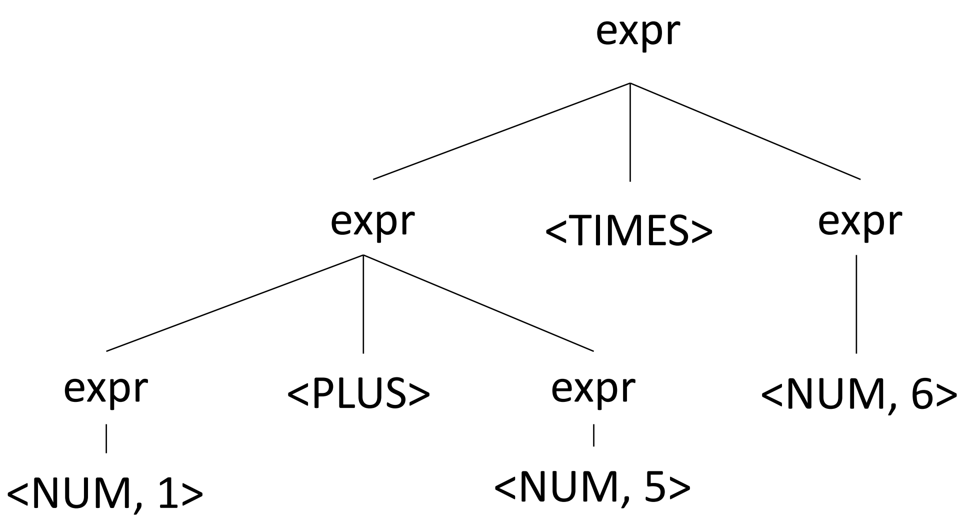
```
expr : NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

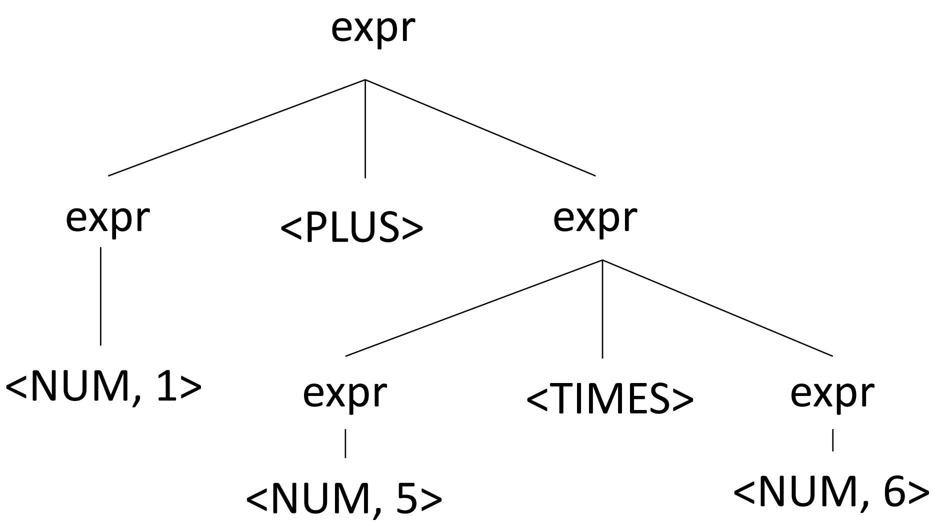


Parse trees

- `input: 1 + 5 * 6`

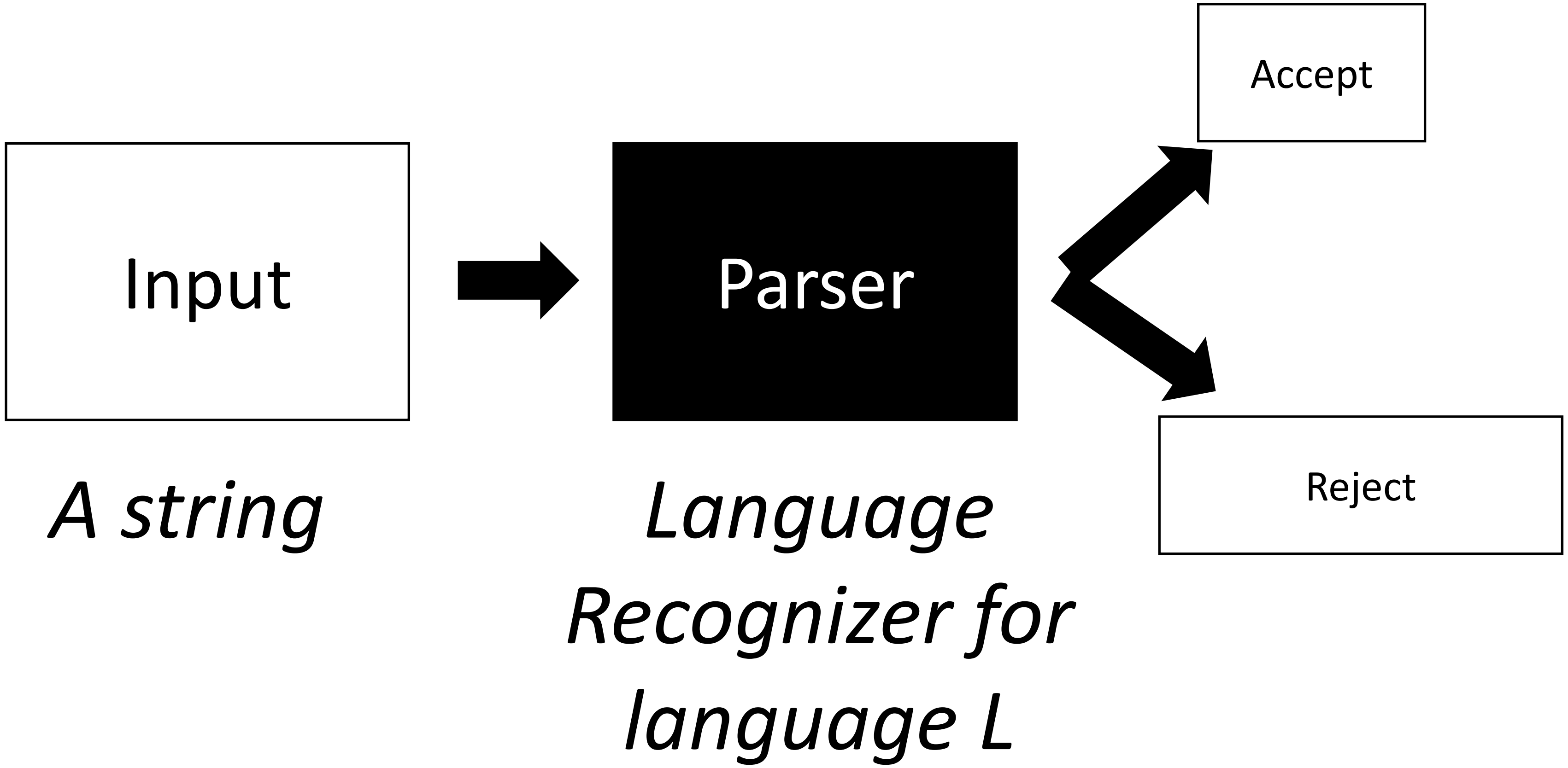
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```





Ambiguous grammars

- What's the issue?



Input

Parser

Accept

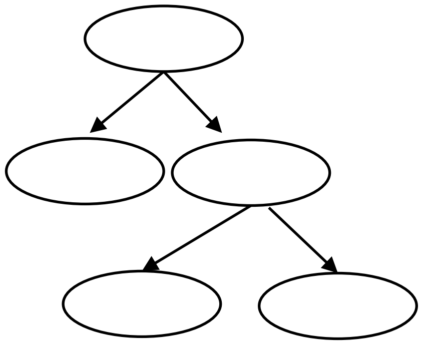
Reject

A string

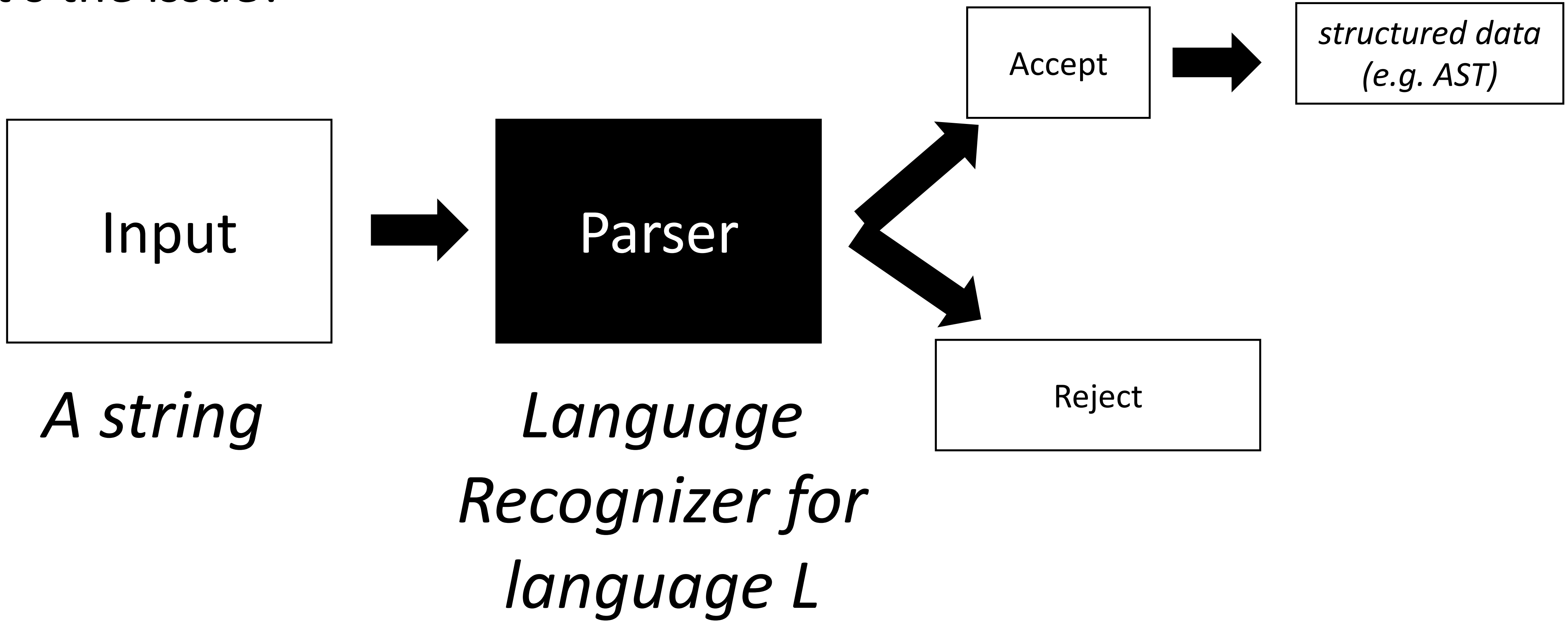
Language Recognizer for language L

Ambiguous grammars

continue to the rest of compilation



- What’s the issue?



Input

A string

Parser

Language Recognizer for language L

Accept

Reject

structured data (e.g. AST)

Meaning into structure

- Structural meaning defined to be a post-order traversal

Meaning into structure

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

Meaning into structure

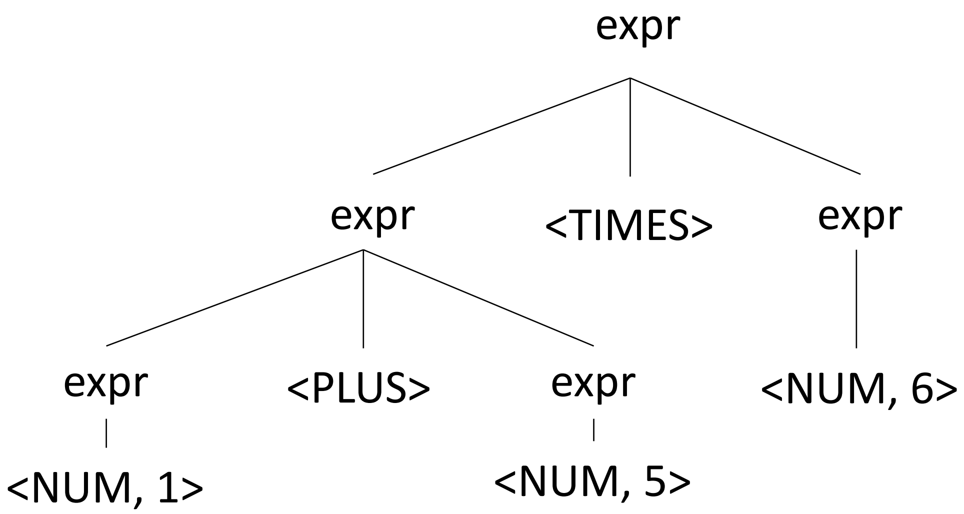
- Structural meaning defined to be a post-order traversal
    - Children return values to their parent
    - Nodes are only evaluated once all their children have been evaluated
    - Evaluated from left to right

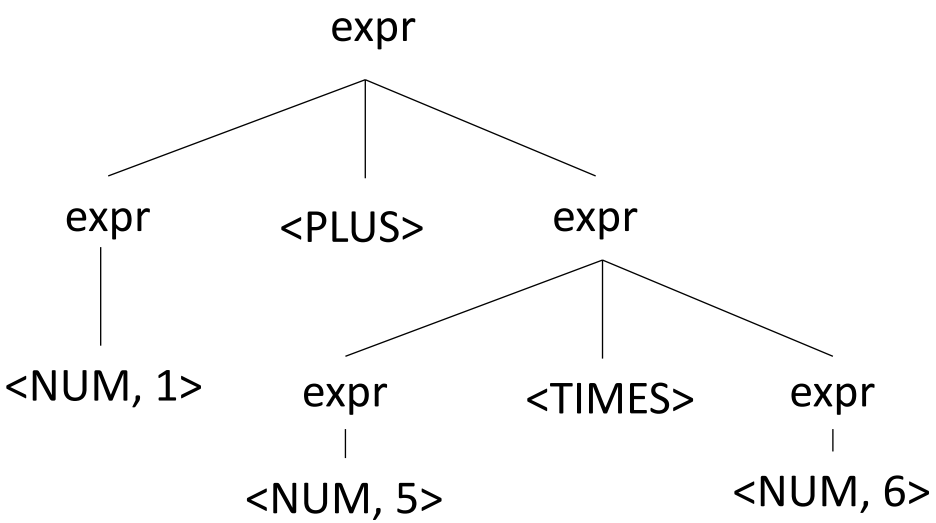
- Can also encode the order of operation

Ambiguous grammars

- `input: 1 + 5 * 6`

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





Avoiding Ambiguity

- How to avoid ambiguity related to precedence?

- Define precedence: ambiguity comes from conflicts. Explicitly define how to deal with conflicts, e.g. write* has higher precedence than +

- Some parser generators support this, e.g. Yacc

Avoiding Ambiguity

- How to avoid ambiguity related to precedence?

- **Second way**: new production rules
  - One rule for each level of precedence
  - lowest precedence at the top
  - highest precedence at the bottom

- Lets try with expressions and the following:
  - + * ()

Avoiding Ambiguity

- How to avoid ambiguity related to precedence?

- **Second way**: new production rules
  - One rule for each level of precedence
  - lowest precedence at the top
  - highest precedence at the bottom

- Lets try with expressions and the following:
  - + * ()

Precedence
increases going down

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

Now lets create a parse tree

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

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

Now lets create a parse tree

`input: 1+5*6`

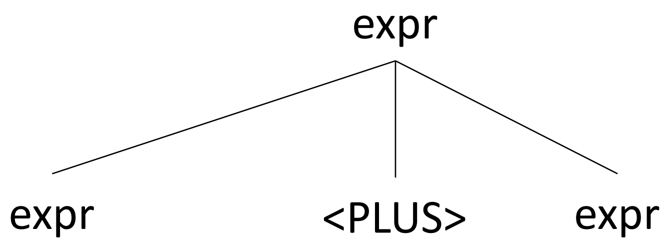
expr

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

Now lets create a parse tree

`input: 1+5*6`

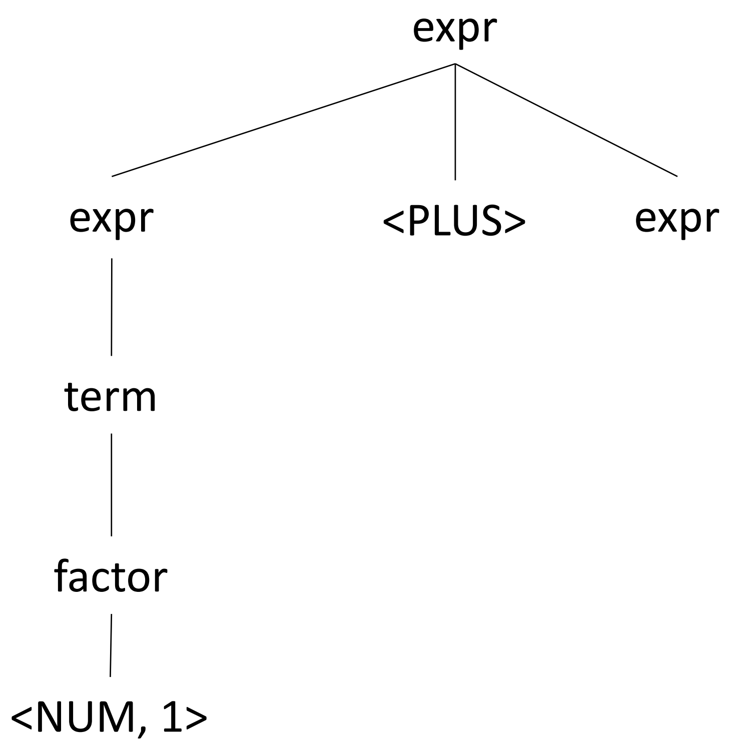
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

`input: 1+5*6`

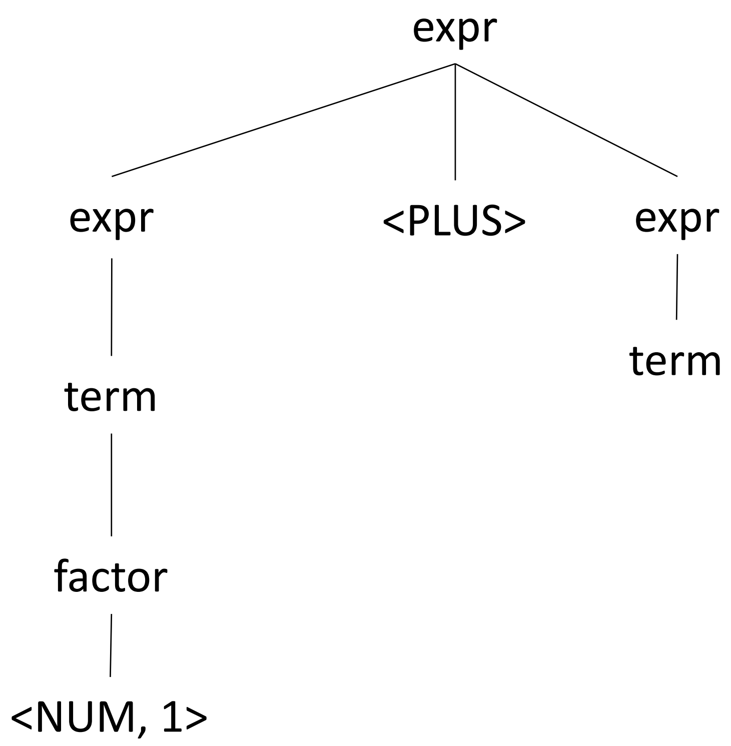
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

input: 1+5*6

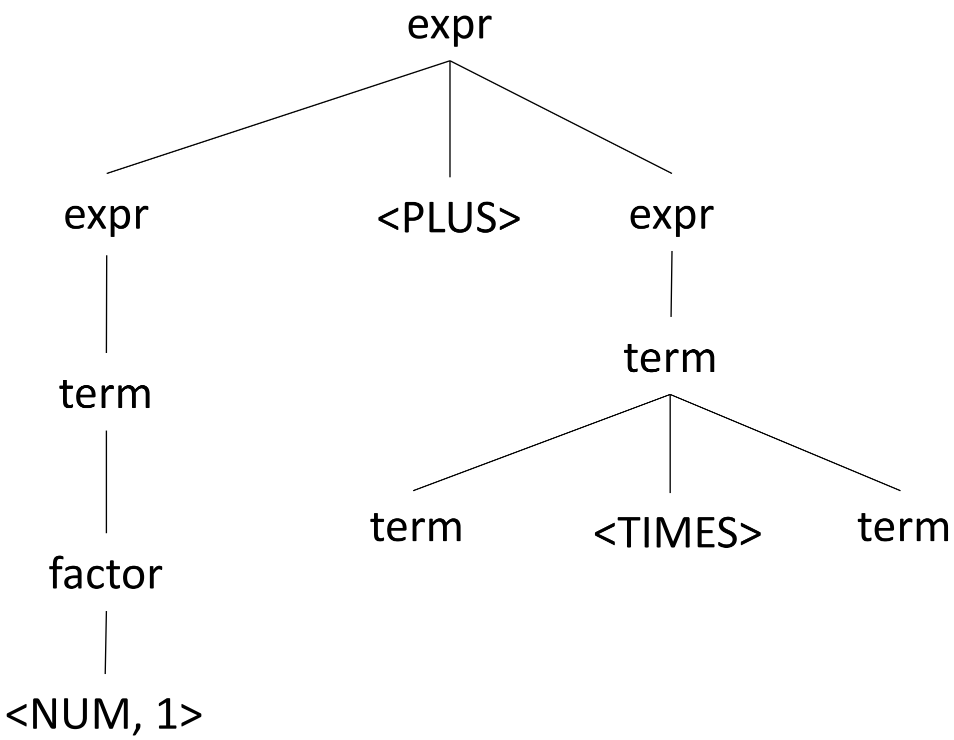
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

`input: 1+5*6`

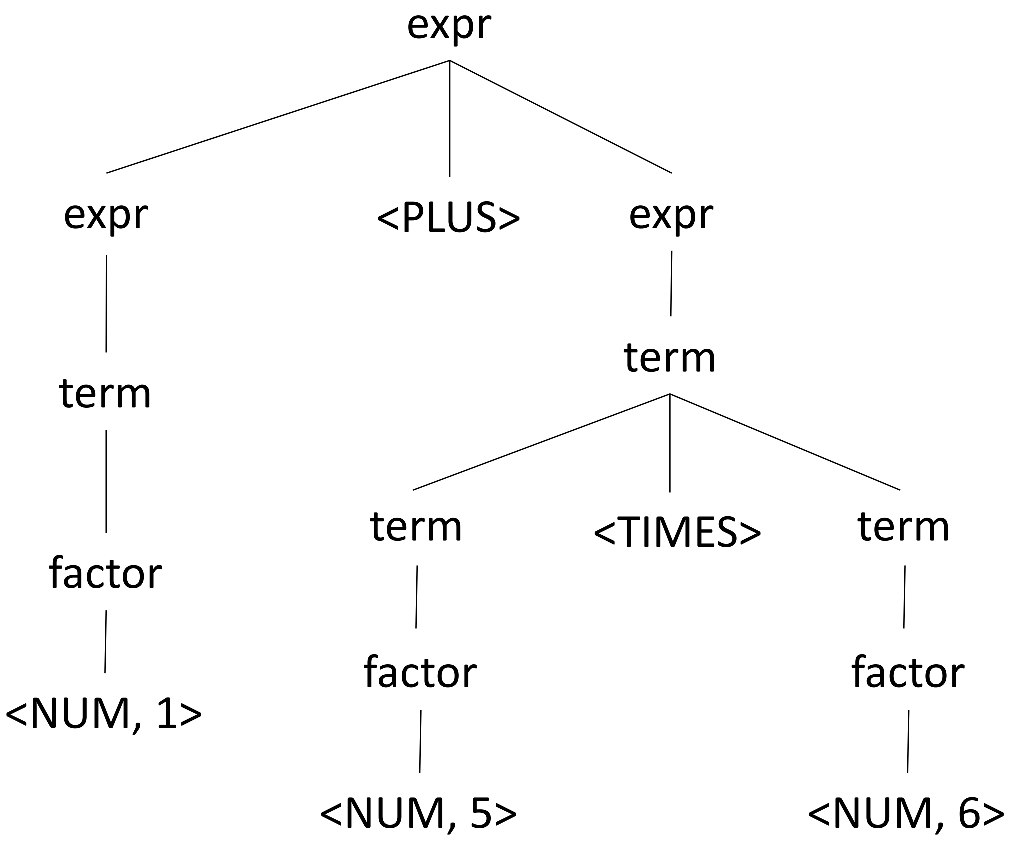
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

`input: 1+5*6`

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



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*
```

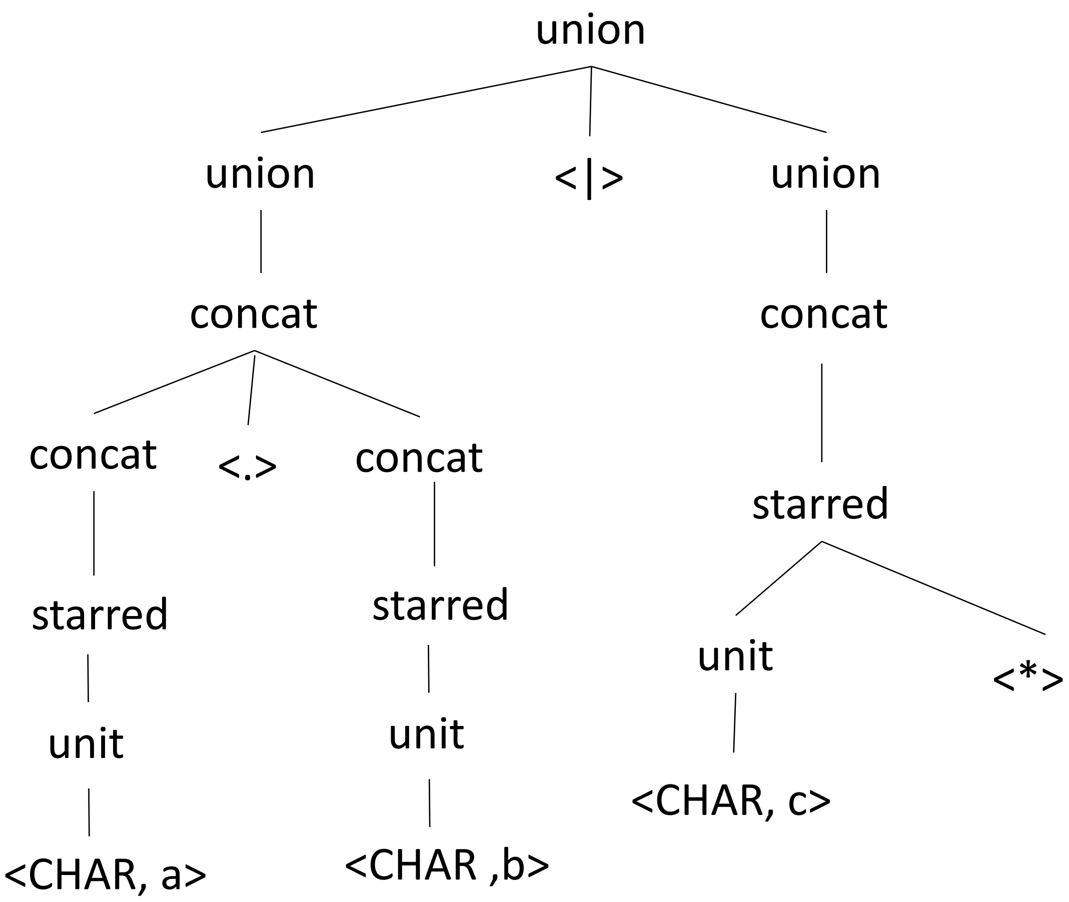
Operator	Name	Productions
|	union	: union PIPE union | concat
.	concat	: concat DOT concat | starred
*	starred	: starred STAR | unit
()	unit	: LPAREN union RPAREN | CHAR

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*`


union
union
<|>
union
concat
concat
concat
<.>
concat
starred
starred
starred
unit
<*>
unit
unit
<CHAR, c>
<CHAR, a>
<CHAR ,b>