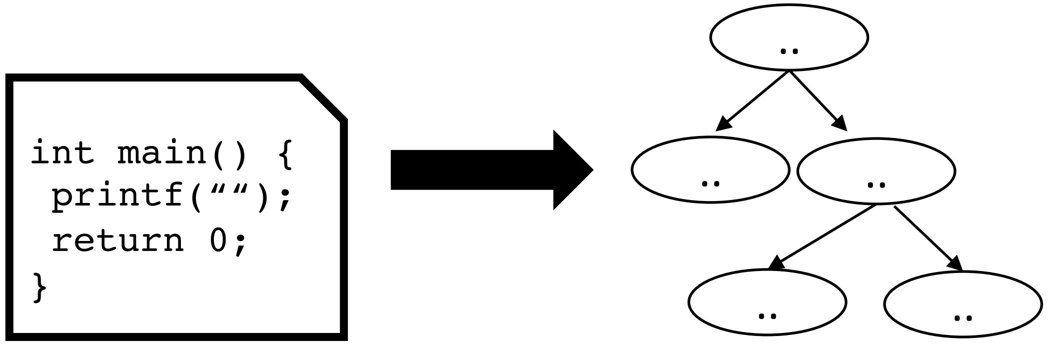
CSE211: Compiler Design

Sept. 27, 2021



- **Topic**: Parsing overview 1 (tokenizing)

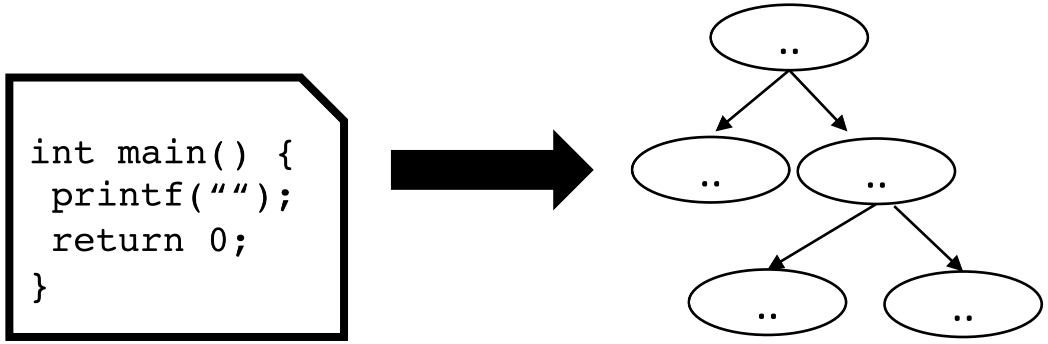
- **Questions**:
  - *What is parsing?*
  - *Have you used Regular Expressions before?*
  - *How do you parse Regular Expressions? What about Context-free Grammars?*

Announcements:

- Tutorial for Docker on website
- Any issues so far?
  - Accessing text book
  - Slides
- Vote for slack, discord, piazza on the website, closes at the end of tomorrow!
- 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

CSE211: Compiler Design

Sept. 27, 2021

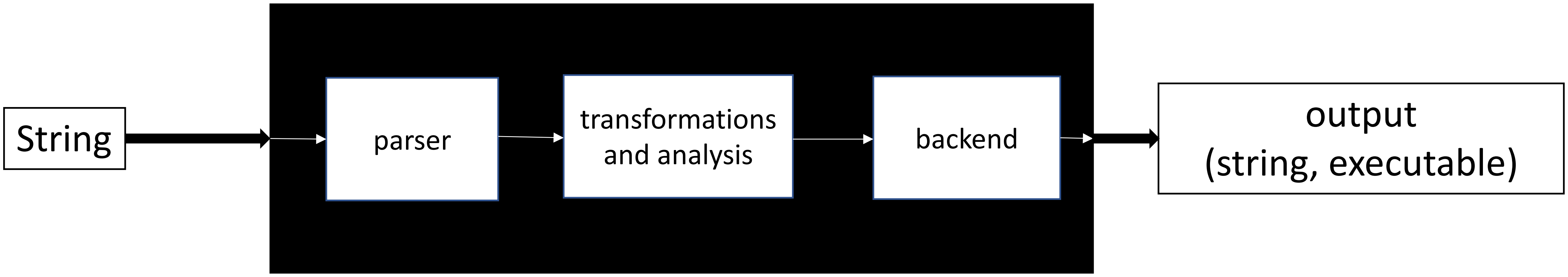


- **Topic**: Parsing overview 1 (tokenizing)

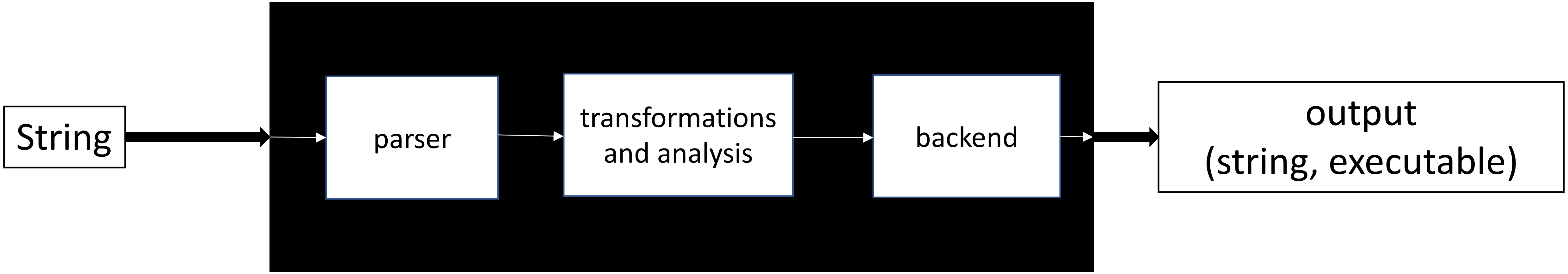
- **Questions**:
  - *What is parsing?*
  - *Have you used Regular Expressions before?*
  - *How do you parse Regular Expressions? What about Context-free Grammars?*

Compiler architecture overview

Compiler architecture overview

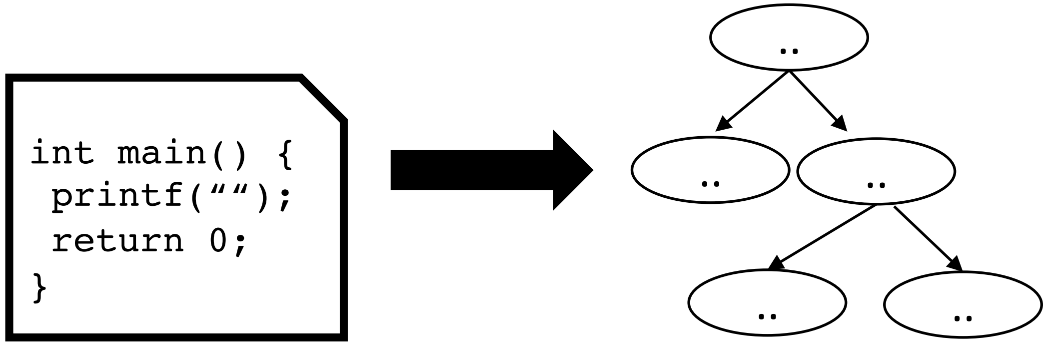


Compiler architecture overview



Parsing is the first step in the compiler

Creates structure



Parsing is the first step in a compiler

- How do we parse a sentence in English?

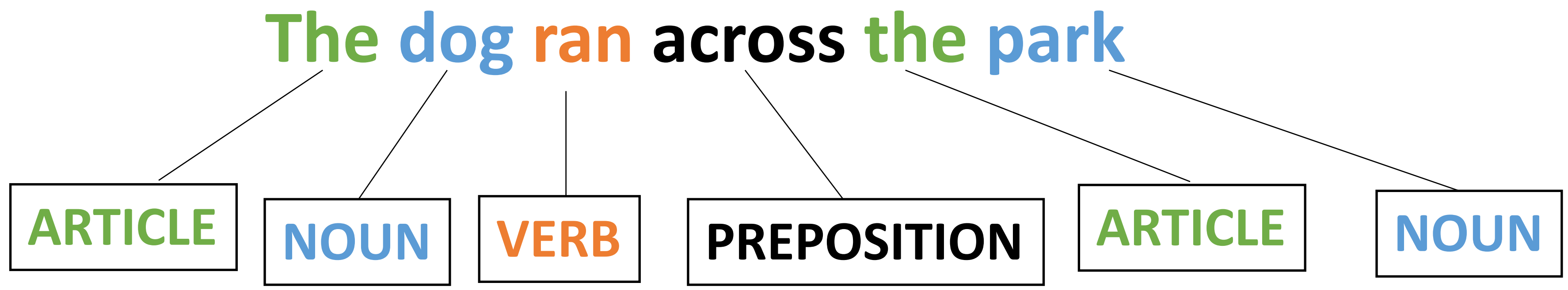
Parsing is the first step in a compiler

- How do we parse a sentence in English?

The dog ran across the park

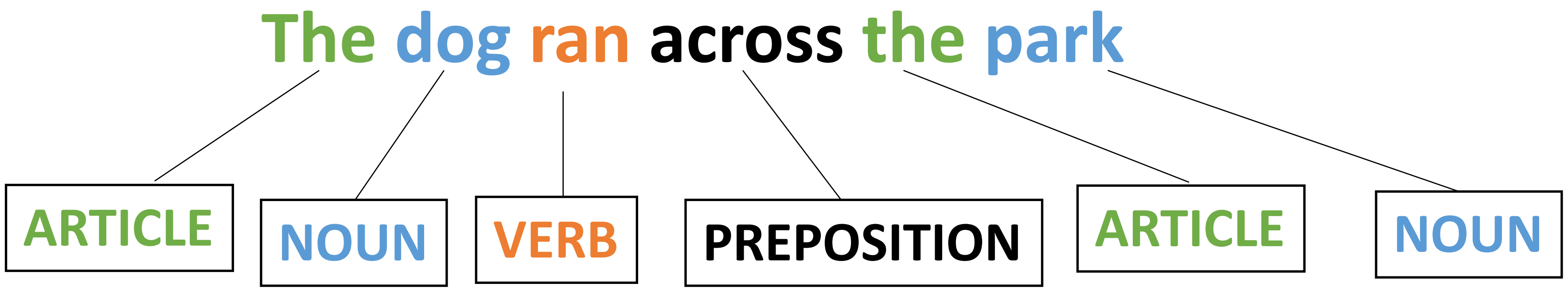
Parsing is the first step in a compiler

- How do we parse a sentence in English?



Parsing is the first step in a compiler

- How do we parse a sentence in English?

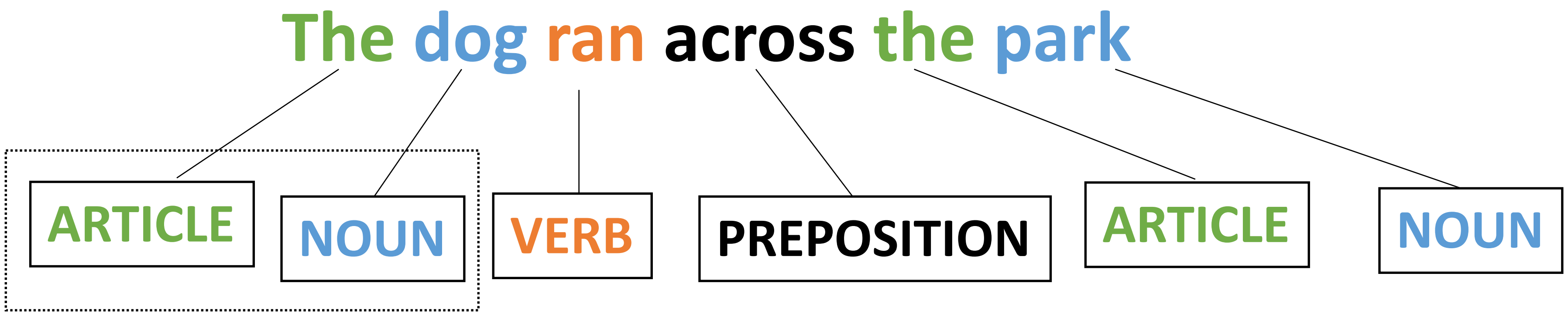


Grammar and Syntax

What about semantics?

Parsing is the first step in a compiler

- How do we parse a sentence in English?

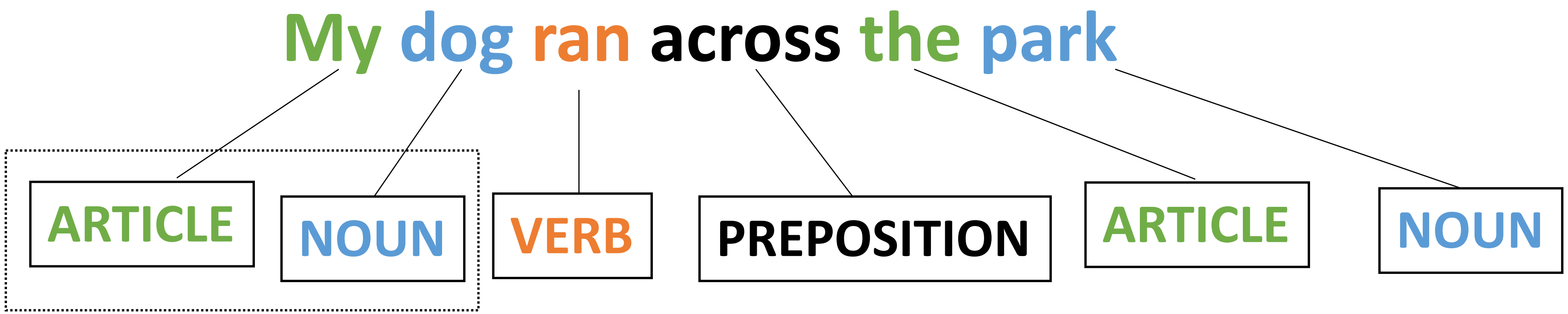


Grammar and Syntax

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}

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. It is a deep dark hole. 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 (e.g. formal properties of regular expressions, parsing algorithms).

High-level parser

Parser

High-level parser

A parser needs a language specification:

- What forms can these take?

Parser

High-level parser

A parser needs a language specification:

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

High-level parser

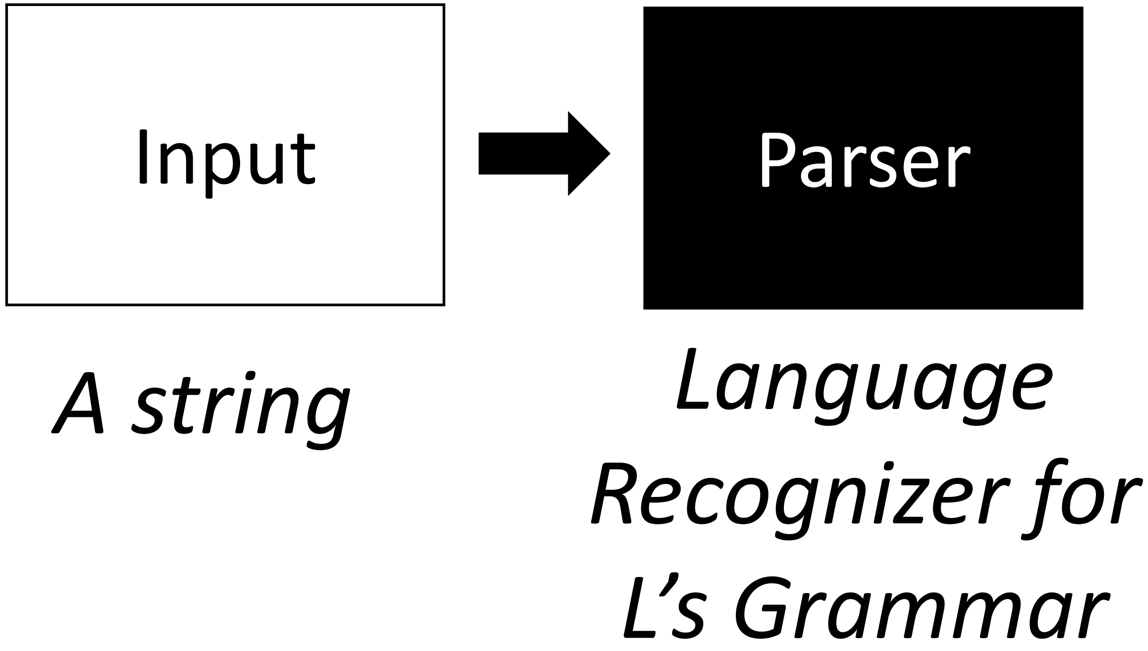
A parser needs a language specification:

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

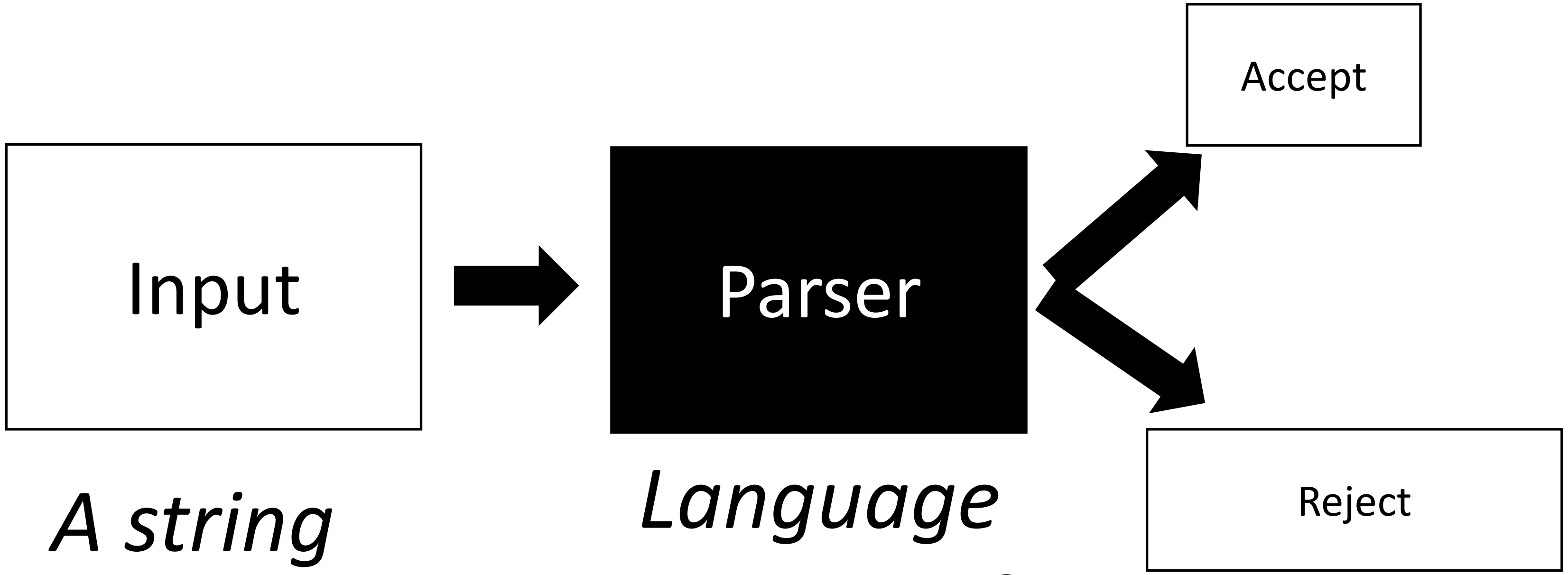
Parser

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

High-level parser



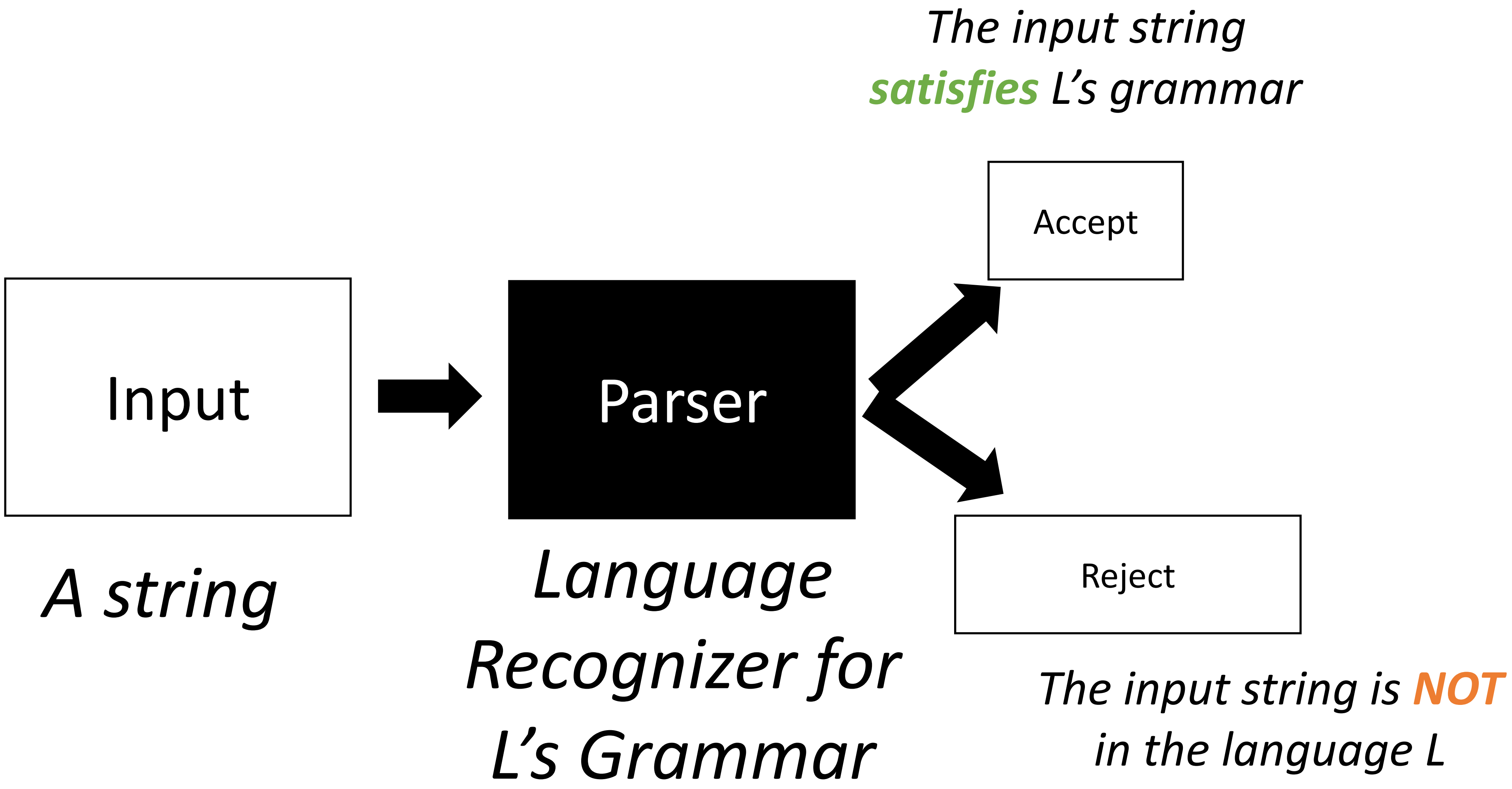
High-level parser



A string

Language Recognizer for L's Grammar

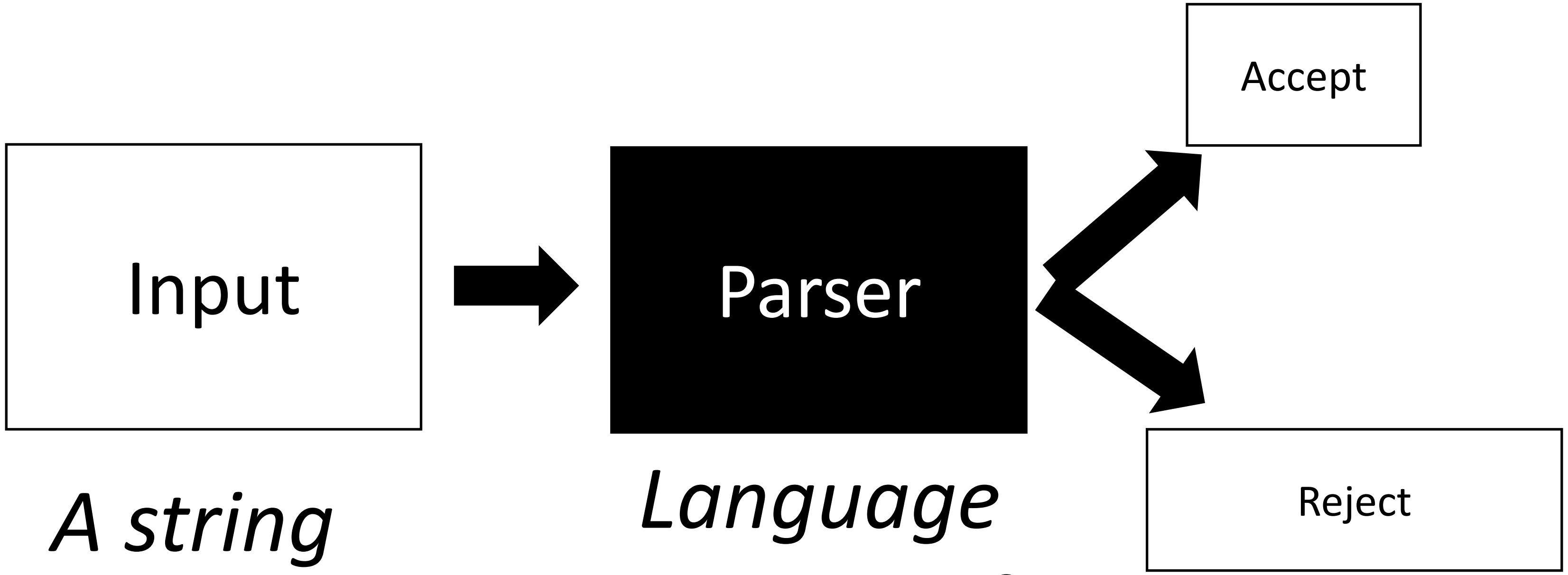
High-level parser



High-level parser

what other types of errors might happen up here?

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



Syntax error

A string

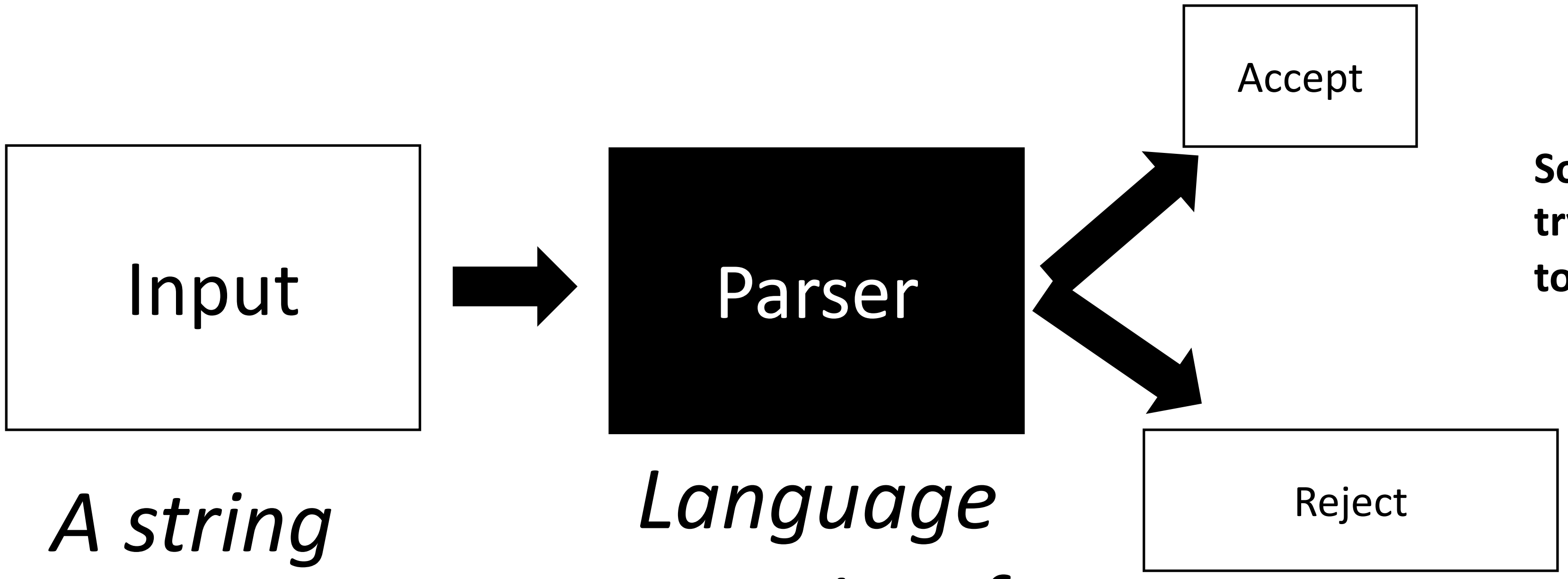
Language Recognizer for L's Grammar

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*



Some languages try to move logic errors to syntax errors!

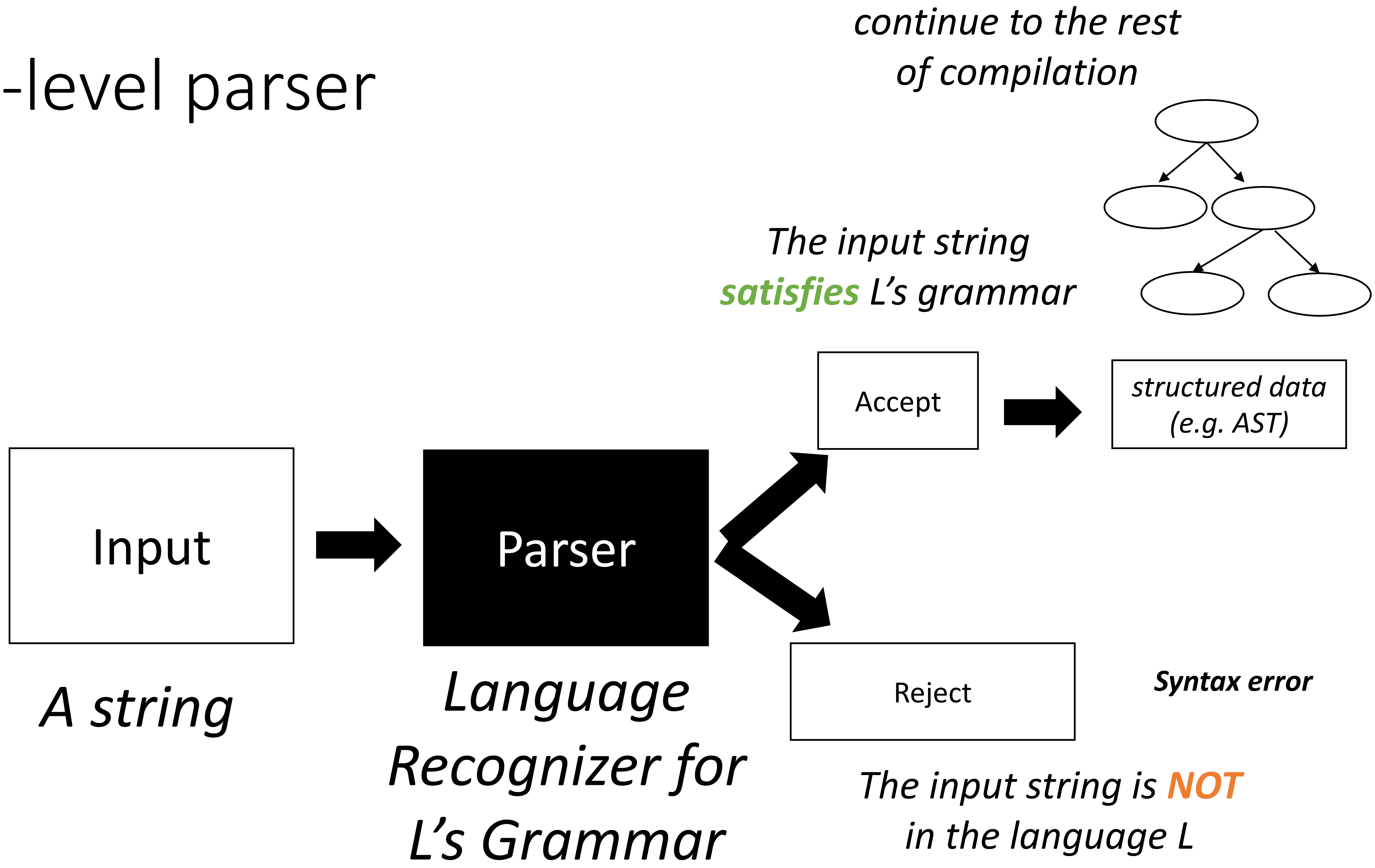
Syntax error

A string

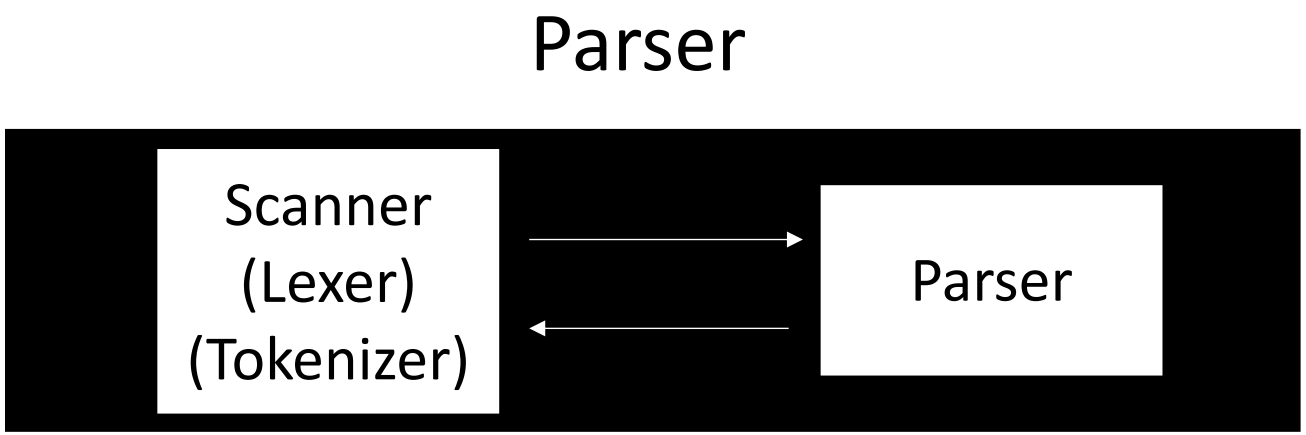
Language Recognizer for L's Grammar

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

High-level parser

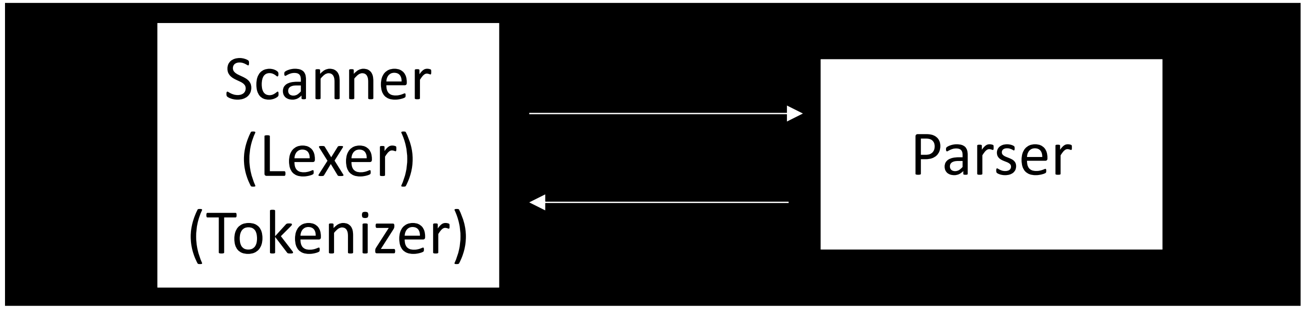


Parser architecture



Parser architecture

Parser

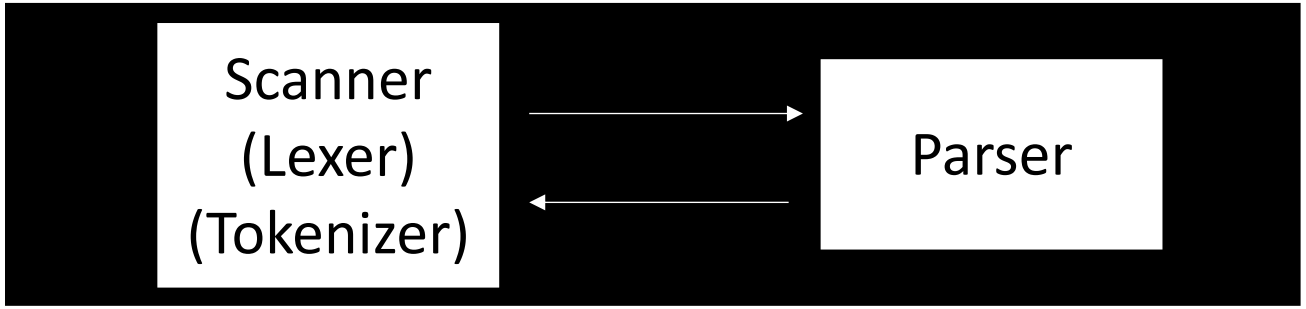


First level of abstraction. Transforms a string of characters into a string of tokens

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

Parser architecture

Parser



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?
numbers
operators
parenthesis
whitespace

Scanner

- Lets write tokens for arithmetic expression:

(LPAREN, ‘(‘)
(NUMBER, 5)
(PLUS, +)
(NUMBER, 4)
(RPAREN, ‘)’)
(TIMES, *)
(NUMBER, 3)

(5 + 4) * 3

Scanner

- Lets write tokens for arithmetic expression:

(5 + 4) * 3

(LPAREN, ‘(‘)
(NUMBER, 5)
(PLUS, +)
(NUMBER, 4)
(RPAREN, ‘)’)
(TIMES, *)
(NUMBER, 3)

(LPAREN, ‘(‘)
(NUMBER, 5)
(OP, +)
(NUMBER, 4)
(RPAREN, ‘)’)
(OP, *)
(NUMBER, 3)

You can generalize tokens

Scanner

- Lets write tokens for arithmetic expression:

(5 + 4) * 3

(LPAREN, ‘(‘)
(NUMBER, 5)
(PLUS, +)
(NUMBER, 4)
(RPAREN, ‘)’)
(TIMES, *)
(NUMBER, 3)

(LPAREN, ‘(‘)
(FIVE, 5)
(PLUS, +)
(FOUR, 4)
(RPAREN, ‘)’)
(TIMES, *)
(THREE, 3)

You can make tokens more specific

Scanner

- Lets write tokens for arithmetic expression:

(5 + 4) * 3

(LPAREN, ‘(‘)
(NUMBER, 5)
(PLUS, +)
(NUMBER, 4)
(RPAREN, ‘)’)
(TIMES, *)
(NUMBER, 3)

(**PAREN, ‘(‘**)
(NUMBER, 5)
(PLUS, +)
(NUMBER, 4)
(**PAREN, ‘)’**)
(TIMES, *)
(NUMBER, 3)

Some choices are more obvious!

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

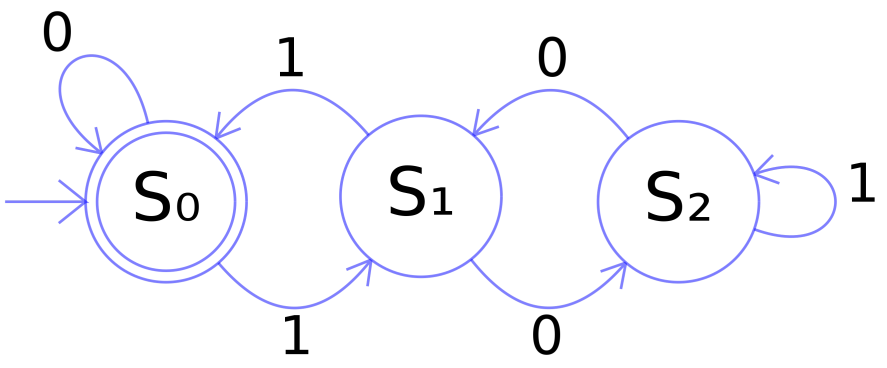
- Literal – single character:
  - PLUS = ‘+’, TIMES = ‘*’
- Keyword – single string:
  - IF = “if”, INT = “int”
- Sets of words:
  - NOUN = {“Cat”, “Dog”, “Car”}
- Numbers
  - NUM = {“0”, “1” …}

Defining tokens

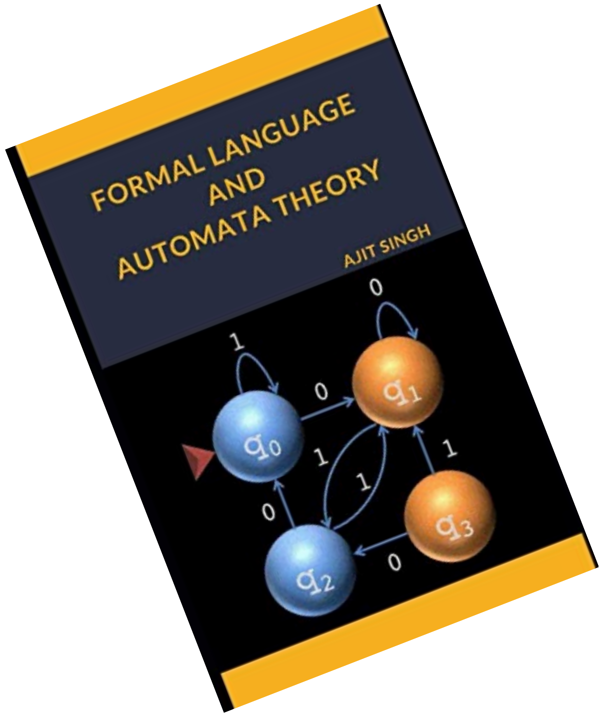
- ~~Literal – single character:~~
  - ~~PLUS = ‘+’, TIMES = ‘*’~~
  - –
- ~~Keyword – single string:~~
  - ~~IF = “if”, INT = “int”~~
  - –
- ~~Sets of words:~~
  - ~~NOUN = {“Cat”, “Dog”, “Car”}~~
  - –
- ~~Numbers~~
  - ~~NUM = {“0”, “1” …}~~

- 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)



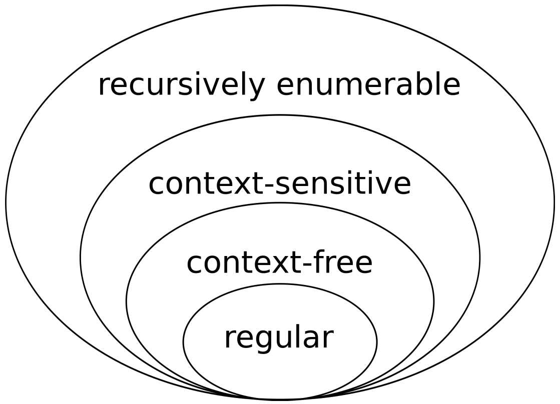


image source: wikipedia

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?
  - “cat | dog”
  - (xy)*

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?
  - 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 = (-|\+)?[0-9]+
  - FLOAT = (-|\+)?[0-9]+(\.[0-9]+)?

Longest possible match

- Consider the re:
- CLASS_TOKEN = {“cse” | “211” | “cse211”}
- What would the lexeme be for: “cse211”
- (CLASS_TOKEN, ?)

Longest possible match

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

how would we parse “x++;”

```
(ID, “x”) (ADD, “+”) (ADD, “+”) (SEMI, “;”)
```

```
(ID, “x”) (INCREMENT, “++”) (SEMI, “;”)
```

Longest possible match

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

how would we parse “x++;”

```
(ID, “x”) (ADD, “+”) (ADD, “+”) (SEMI, “;”)
```

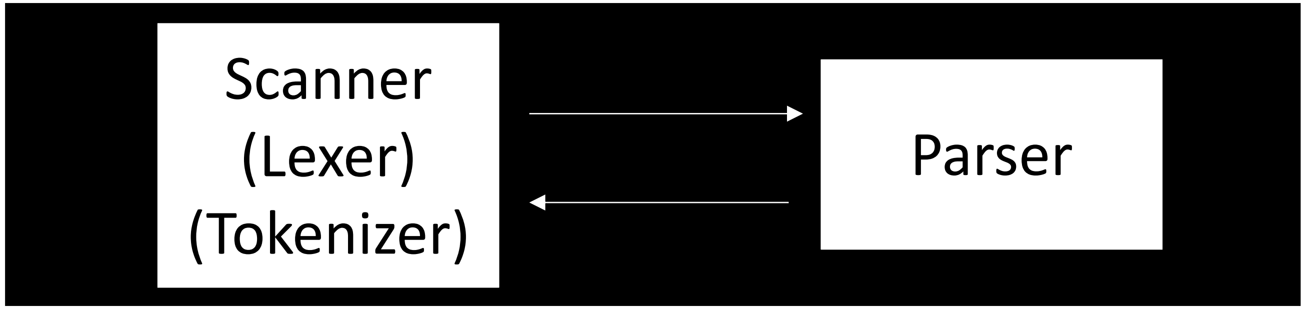
```
(ID, “x”) (INCREMENT, “++”) (SEMI, “;”)
```

We match the longest possible substring

Scanner Questions?

- A scanner splits a string into lexemes
- Tokens are defined using regular expressions
- Regular expressions are good for matching operators, parenthesis, variable names, numbers, key words etc.

Parser



Next class

- Chapter 2 in EAC goes into detail on regular expression parsing
  - Finite automata etc.

- Production rules for expressions
  - parse trees
  - associativity
  - ambiguous grammars

- For you:
  - Try out docker instructions
  - Vote for Canvas alternatives!
  - Homework is released in 1 week!

- See you on Wednesday!