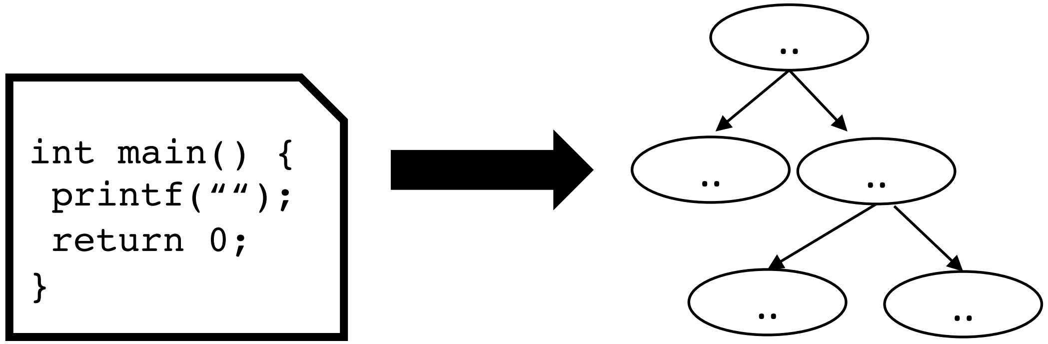
CSE211: Compiler Design

Oct. 6, 2020



- **Topic**: Parsing Overview

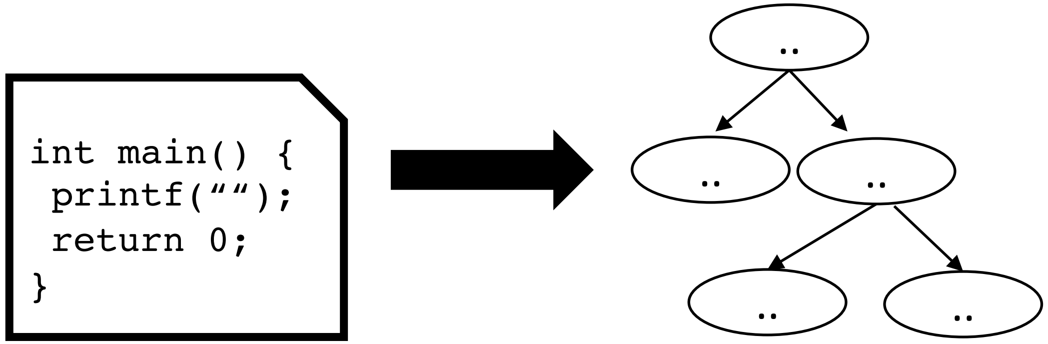
- **Questions**:
  - *What is parsing?*
  - *Have you used Regular Expressions before?*
  - *How do you parse Regular Expressions? What about Context-free Grammars?*

Announcements:

- Moving Homework due dates back one week (more time to work on homework after module is finished)
- Notes will include a reference to EAC
- Link to reserve is up
- How to watch YuJa recordings

CSE211: Compiler Design

Oct. 6, 2020



- **Topic**: Parsing Overview

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

Parsing is the first step in a compiler

- How do we parse language?

Parsing is the first step in a compiler

- How do we parse language?

The dog ran across the park

Parsing is the first step in a compiler

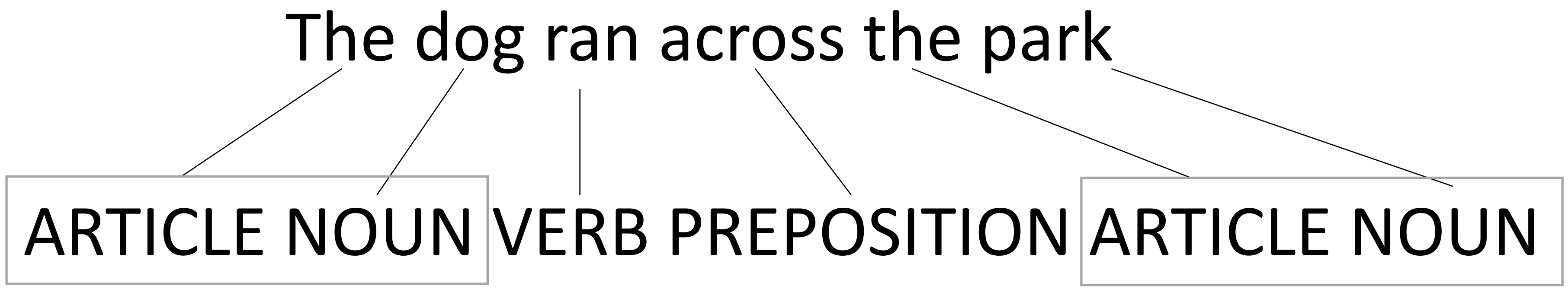
- How do we parse language?

The dog ran across the park

ARTICLE NOUN VERB PREPOSITION ARTICLE NOUN

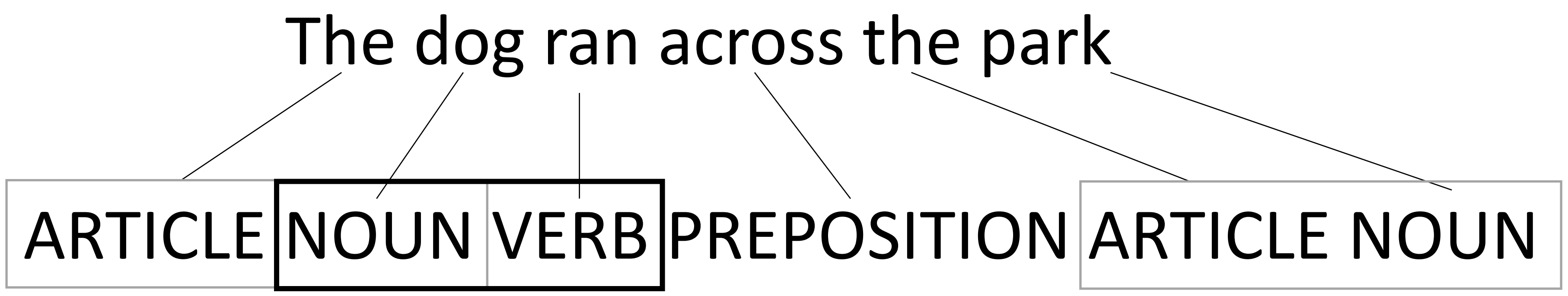
Parsing is the first step in a compiler

- How do we parse language?



Parsing is the first step in a compiler

- How do we parse language?



A Simple Language

- ARTICLE
- NOUN
- VERB
- ADJECTIVE

A Simple Language

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

A Simple Language

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

ARTICLE NOUN VERB

A Simple Language

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

ARTICLE ADJECTIVE? NOUN VERB

A Simple Language

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

ARTICLE ADJECTIVE? NOUN VERB

My Microsoft Computer Crashed

A Simple Language

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

ARTICLE ADJECTIVE? NOUN VERB

The Purple Dog Crashed

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 set precedence and associativity.
- Learn how to parse using a classic parser generator (Lex and Yacc) for a simple programming 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.

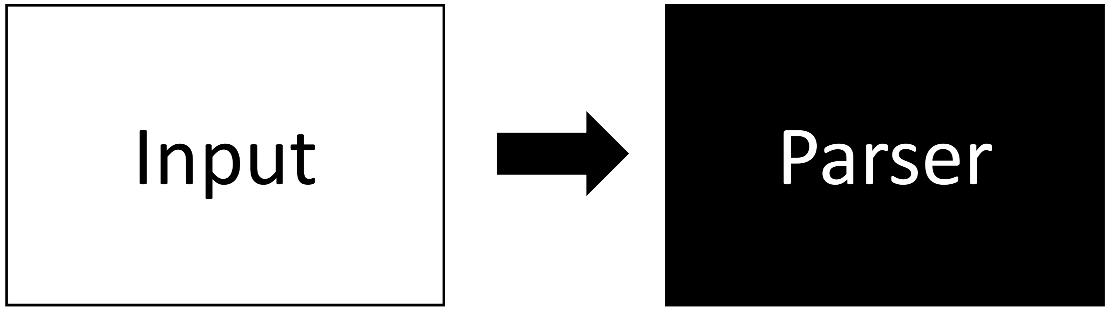
- This module should provide you with the background to implement simple compilers. It will make you very popular with future colleagues who are scared of compilers.

- These topics are typically covered in more depth in an undergrad course (e.g. formal properties of regular expressions, parsing algorithms).

High-level parser

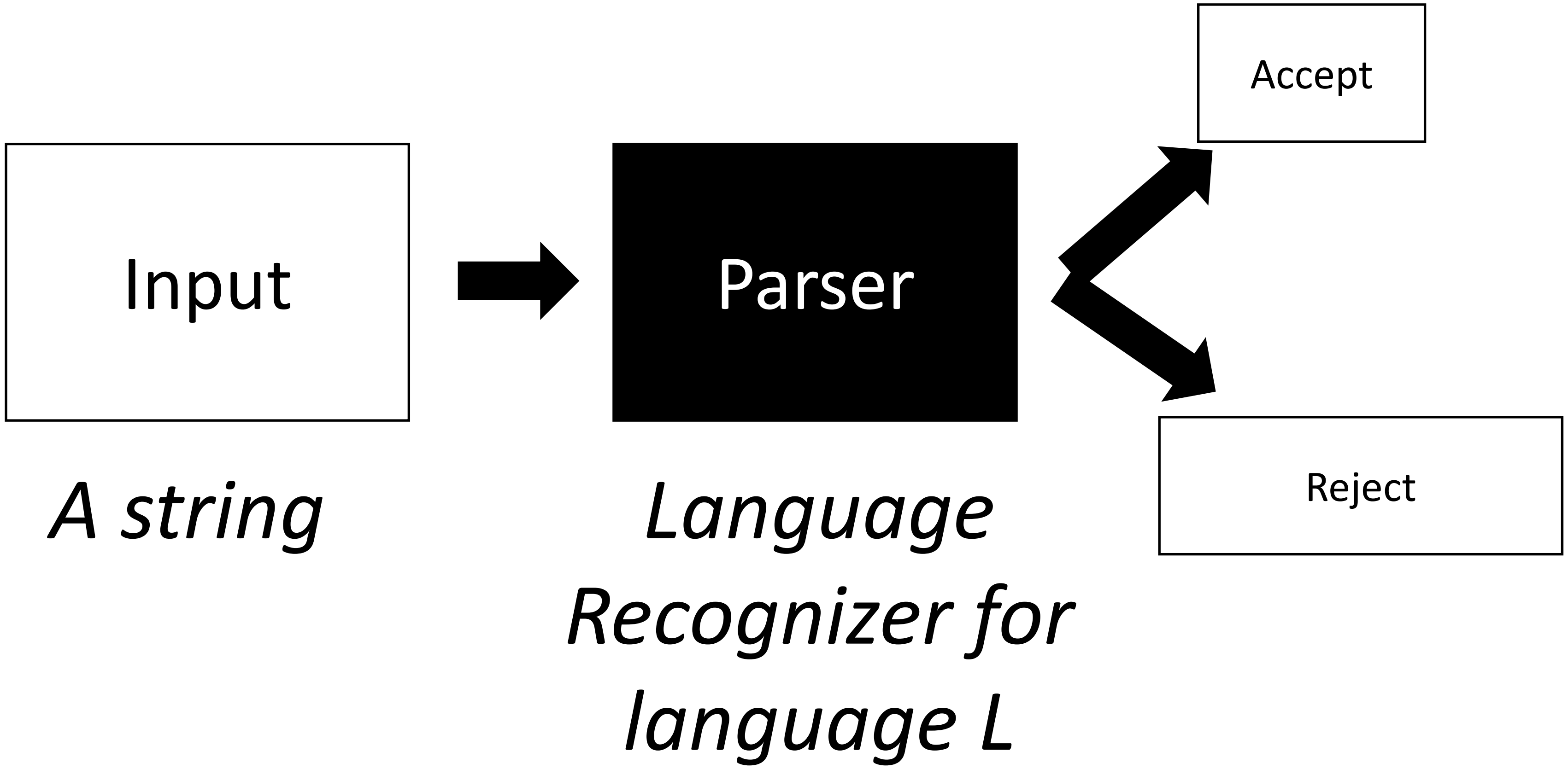
Parser

High-level parser



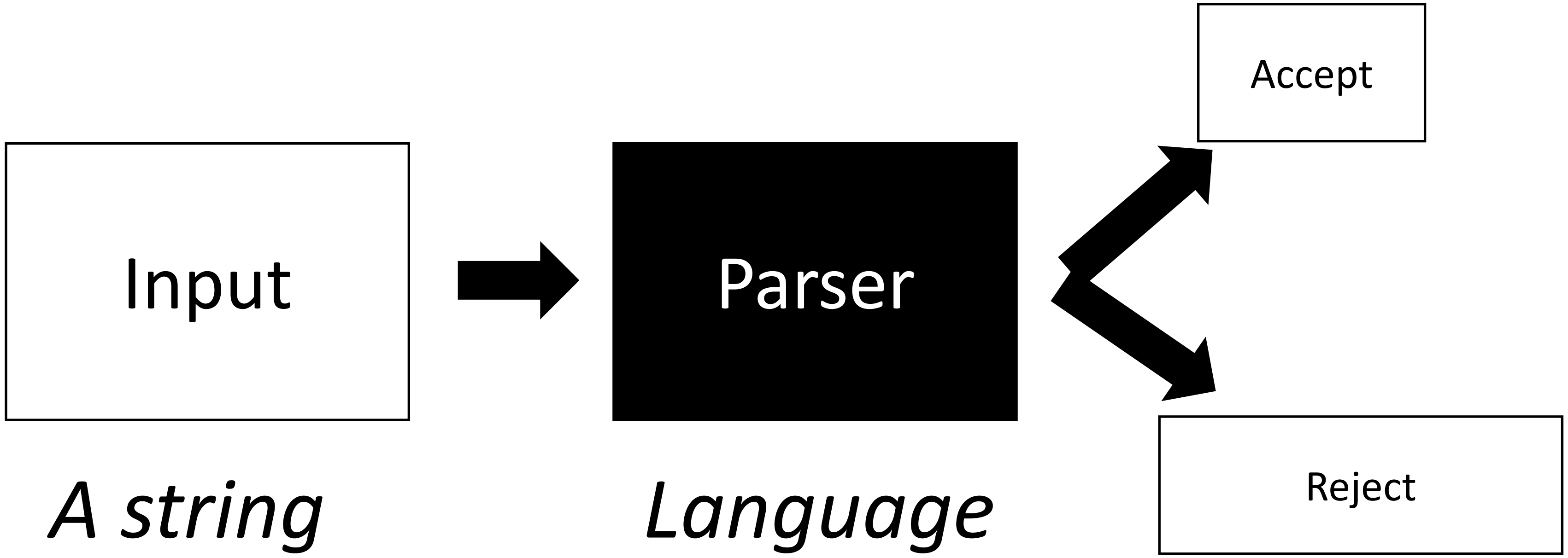
A string

High-level parser



High-level parser

The input string is in the language L

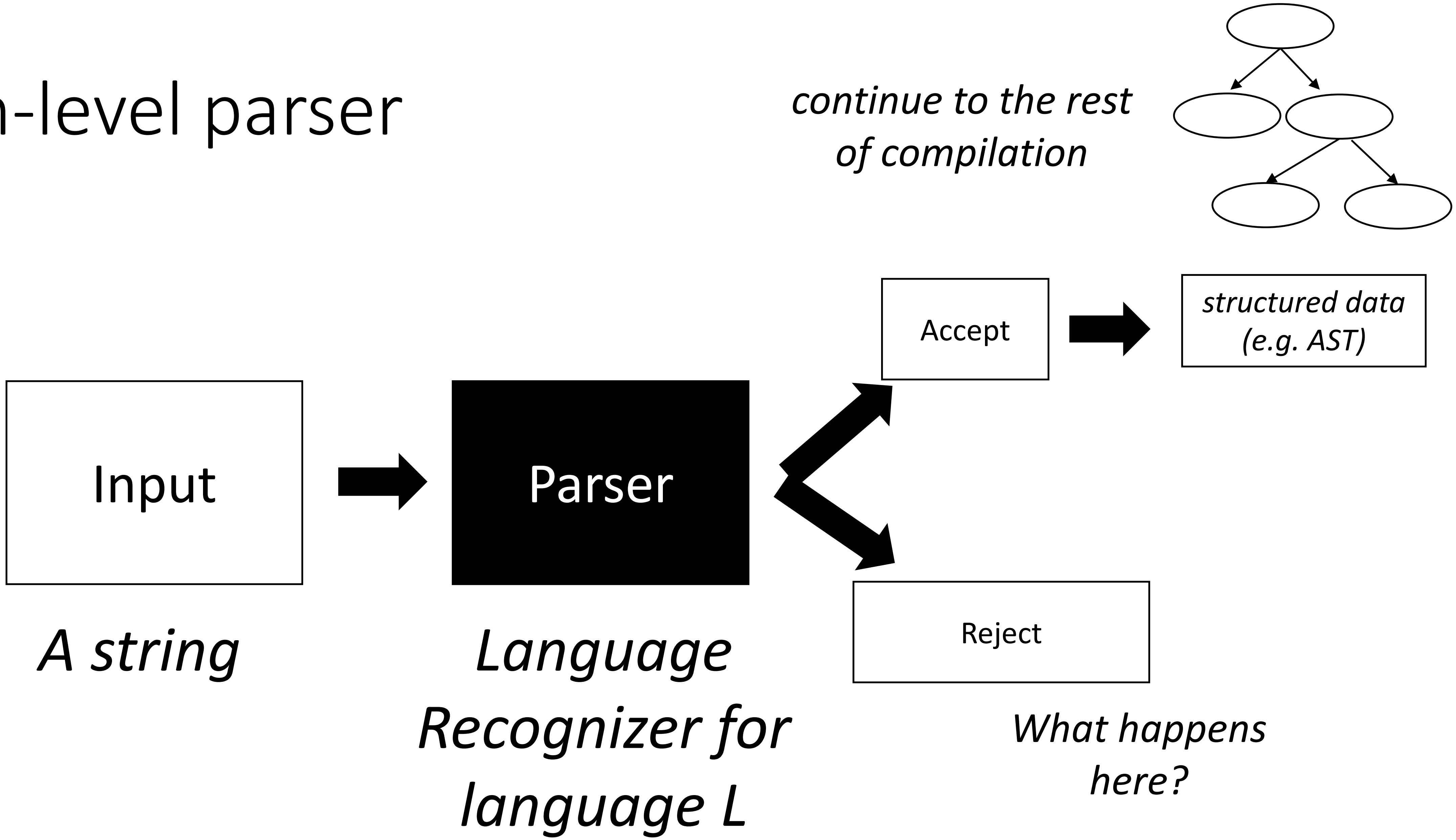


A string

Language Recognizer for language L

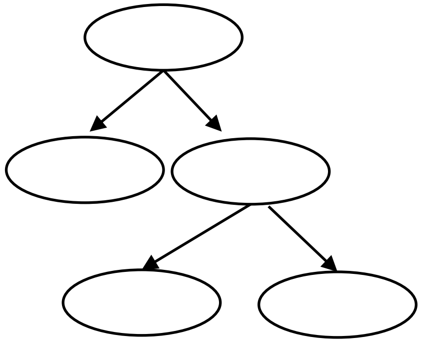
The input string is NOT in the language L

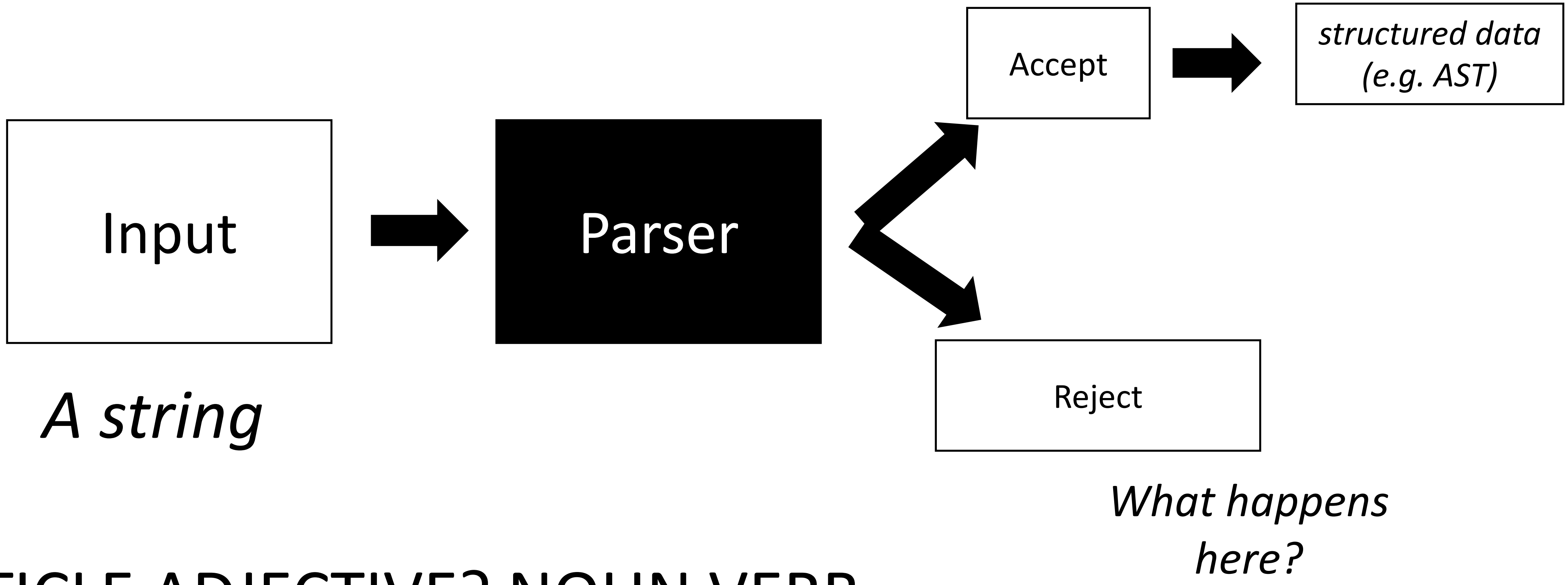
High-level parser



High-level parser

continue to the rest of compilation



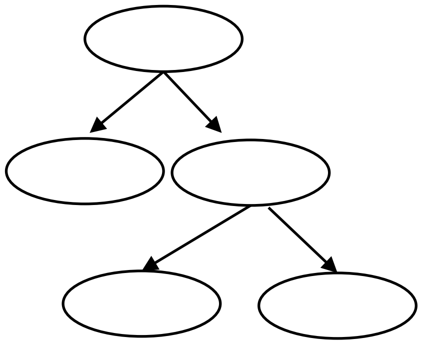


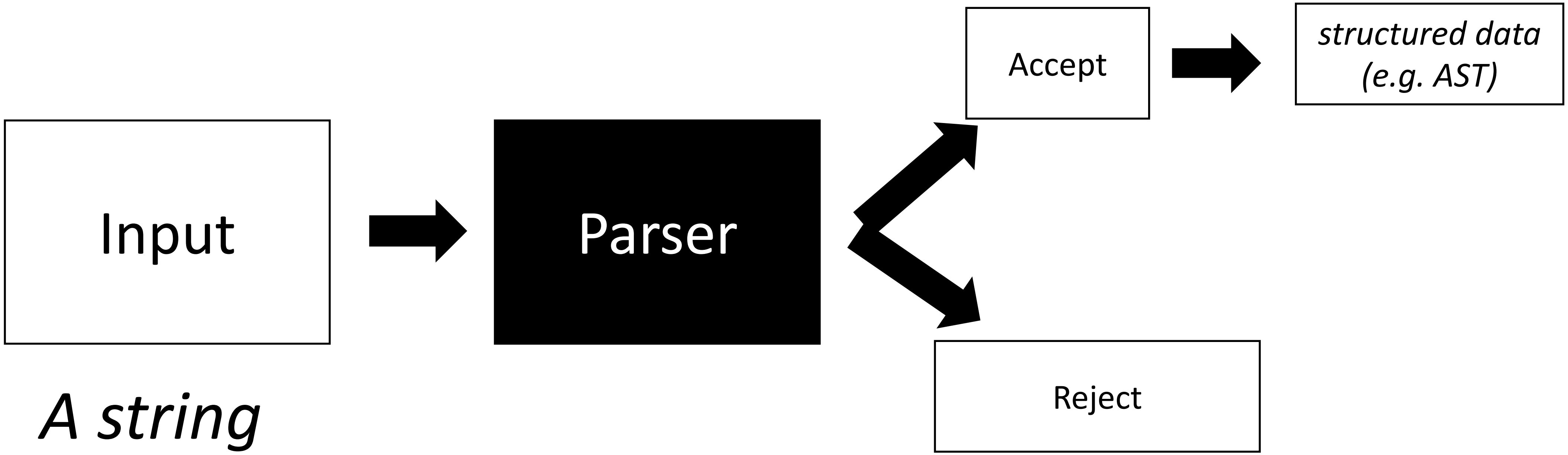
ARTICLE ADJECTIVE? NOUN VERB

My Microsoft Computer **Computer**

High-level parser

continue to the rest of compilation



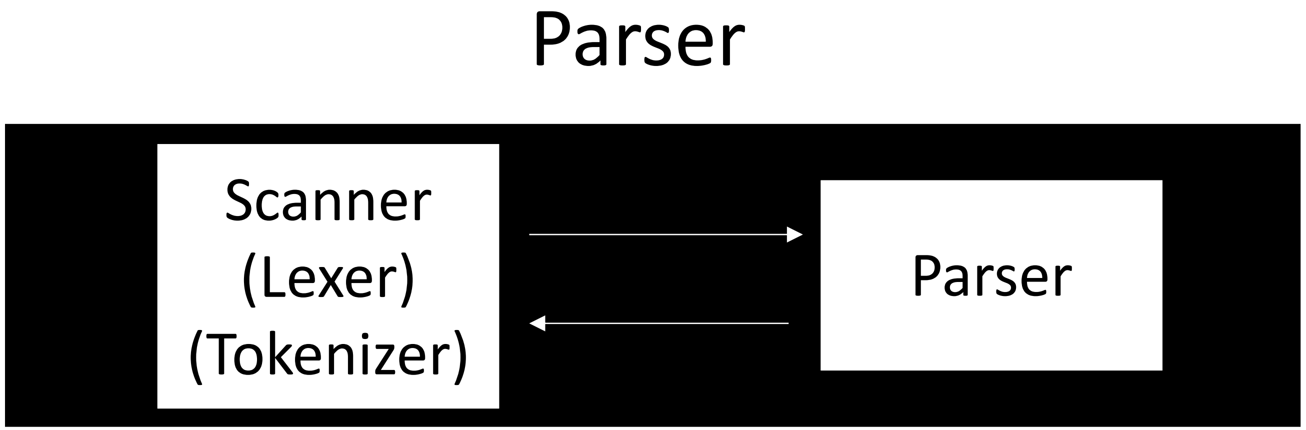


What happens here?

ARTICLE ADJECTIVE? NOUN VERB

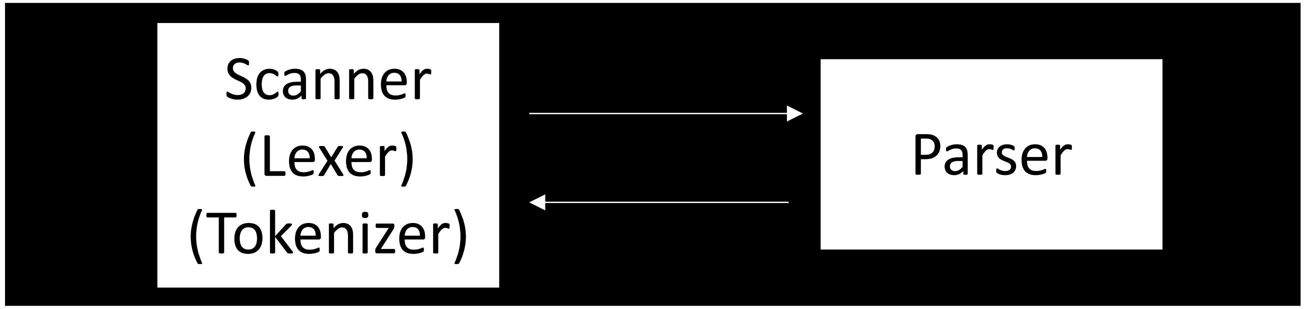
The Purple **Dog** Crashed

Parser architecture



Parser architecture

Parser

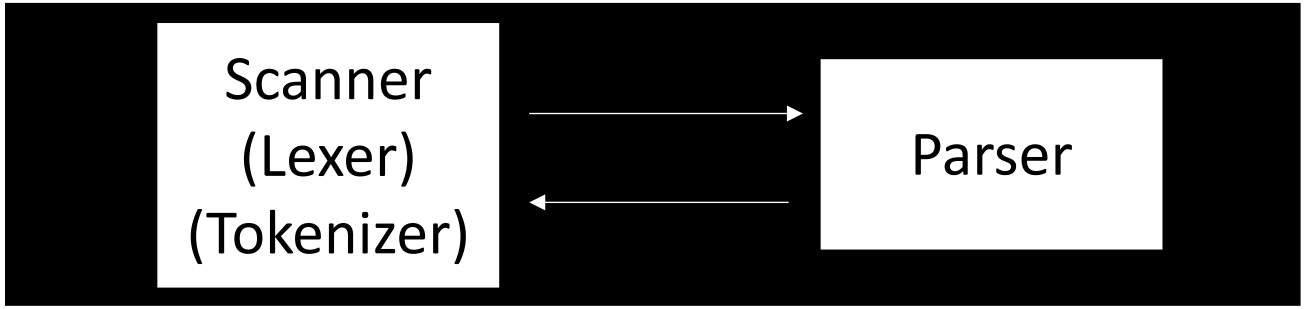


First level of abstraction. Transforms a string of characters into a string of tokens

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 Microsoft Computer Crashed

Scanner

My Microsoft Computer Crashed

↓ Scanner

(ARTICLE, my) (ADJECTIVE, Microsoft) (NOUN, Computer) (VERB, Crashed)

Scanner

My Microsoft Computer Crashed

Scanner

(ARTICLE, my) (ADJECTIVE, Microsoft) (NOUN, Computer) (VERB, Crashed)

Lexeme: (TOKEN, value)

Scanner

- Lets write tokens for arithmetic expression:

$$(5 + 4) * 3$$

Scanner

- Lets write tokens for arithmetic expression:

$$(5 + 4) * 3$$

(LPAREN, ‘(‘)
(NUMBER, 5)
(PLUS, +)
(NUMBER, 4)
(RPAREN, ‘)’)
(TIMES, *)
(NUMBER, 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” …}

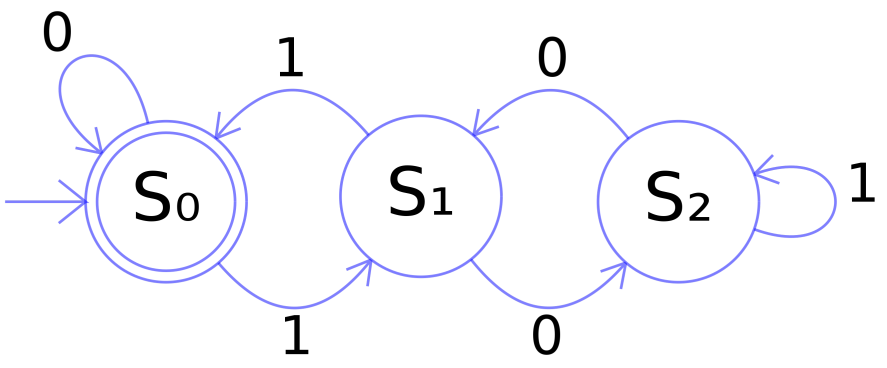
Defining tokens

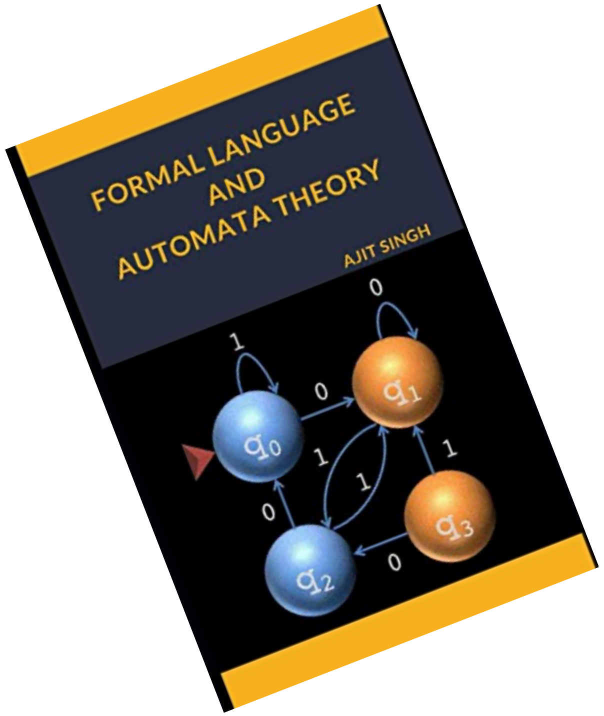
- ~~Literal – single character:~~
  - ~~PLUS = ‘+’, TIMES = ‘*’~~
  - –
- ~~Keyword – single string:~~
  - ~~IF = “if”, INT = “int”~~
  - –
- ~~Sets of words:~~
  - ~~NOUN = {“Cat”, “Dog”, “Car”}~~
  - –
- ~~Numbers~~
  - ~~NUM = {“0”, “1” …}~~

- 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)





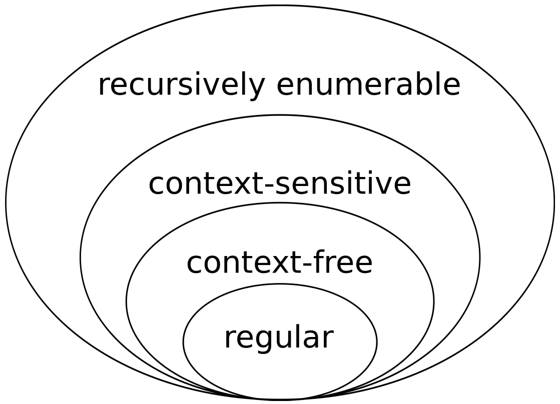


image source: wikipedia

Regular Expressions

We will define RE’s recursively:

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!

- Without ()’s, what is the precedence of concatenation, union, and star?
  - What are some experiments we can do?

Regular Expressions

- Use ()’s to force precedence!

- Without ()’s, what is the precedence of concatenation, union, and star?

- star > concatenation > union

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

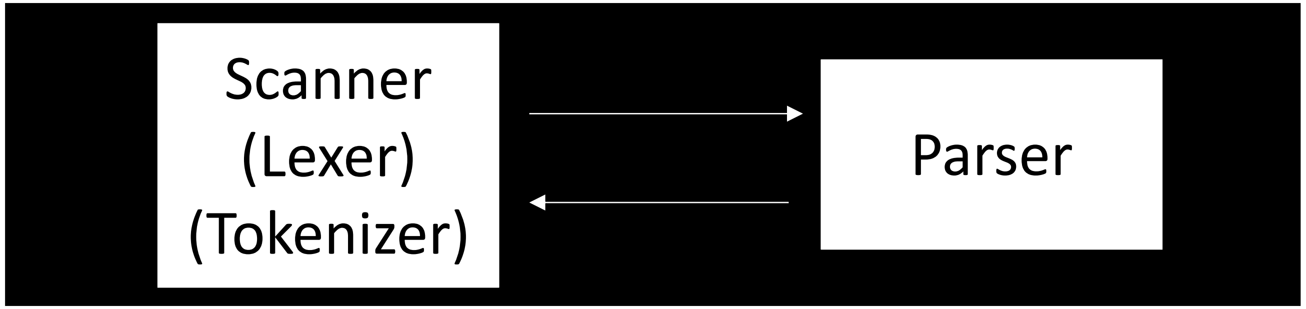
- Literal – single character:
  - PLUS = ‘+’, TIMES = ‘*’
- Keyword – single string:
  - IF = “if”, INT = “int”
- Sets of words:
  - NOUN = “(Cat)|(Dog)|(Car)”
- Numbers
  - SINGLE_NUM = [0-9]
  - NUM = (-|\+)?[0-9]+(\.[0-9]+)?

What about C-style IDs?

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



Parser

- Sentence:
  - ARTICLE ADJECTIVE? NOUN VERB

- What about a mathematical sentence (expression)?

Parser

- Sentence:
  - ARTICLE ADJECTIVE? NOUN VERB

- What about a mathematical sentence (expression)?

  - NUM

Parser

- Sentence:
  - ARTICLE ADJECTIVE? NOUN VERB

- What about a mathematical sentence (expression)?

  - NUM
  - NUM PLUS NUM

Parser

- Sentence:
  - ARTICLE ADJECTIVE? NOUN VERB

- What about a mathematical sentence (expression)?

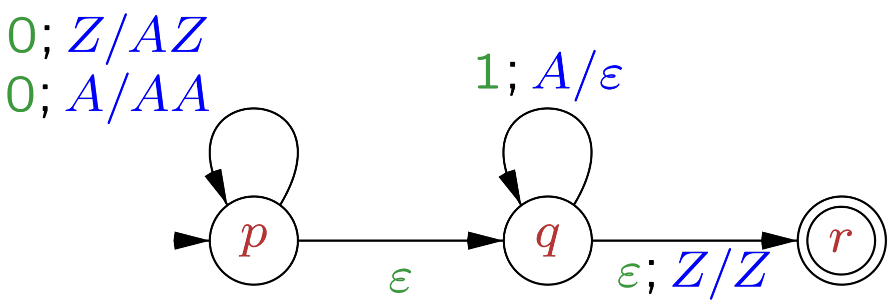
  - NUM
  - NUM BIN_OP NUM

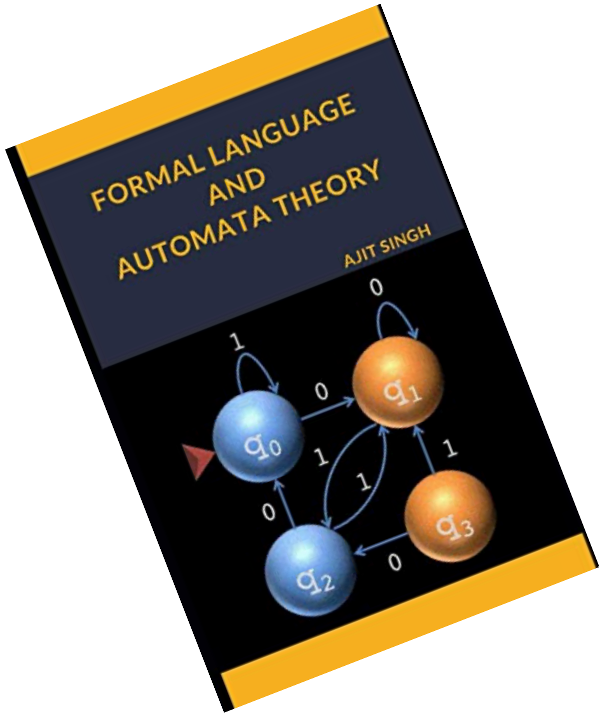
Parser

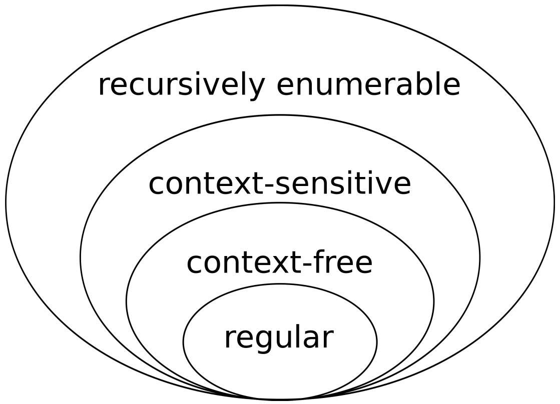
- Sentence:
  - ARTICLE ADJECTIVE? NOUN VERB
- What about a mathematical sentence (expression)?
  - NUM
  - NUM PLUS NUM
  - NUM TIMES NUM
  - NUM PLUS NUM TIMES NUM
  - NUM PLUS NUM TIMES NUM
  - NUM (BIN_OP NUM)*

Context Free Grammars

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







Parser

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

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

  sentence: ARTICLE ADJECTIVE NOUN VERB
  | ARTICLE NOUN VERB

Parser

- Production rules can reference other production rules

sentence: adjective_sentence
| non_adjective_sentence

adjective_sentence: ARTICLE ADJECTIVE NOUN VERB

non_adjective_sentence: ARTICLE NOUN VERB

Parser

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

sentence: ARTICLE adjective_list NOUN VERB

Parser

- 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>

Next week

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

- Homework is released next class:
  - Have a look, but we will cover PLY and parsing with derivatives next Tuesday

- See you on Thursday!