Lecture 02

# Lexical Analysis

## Part 1: specification of tokens

**Hyosu Kim**

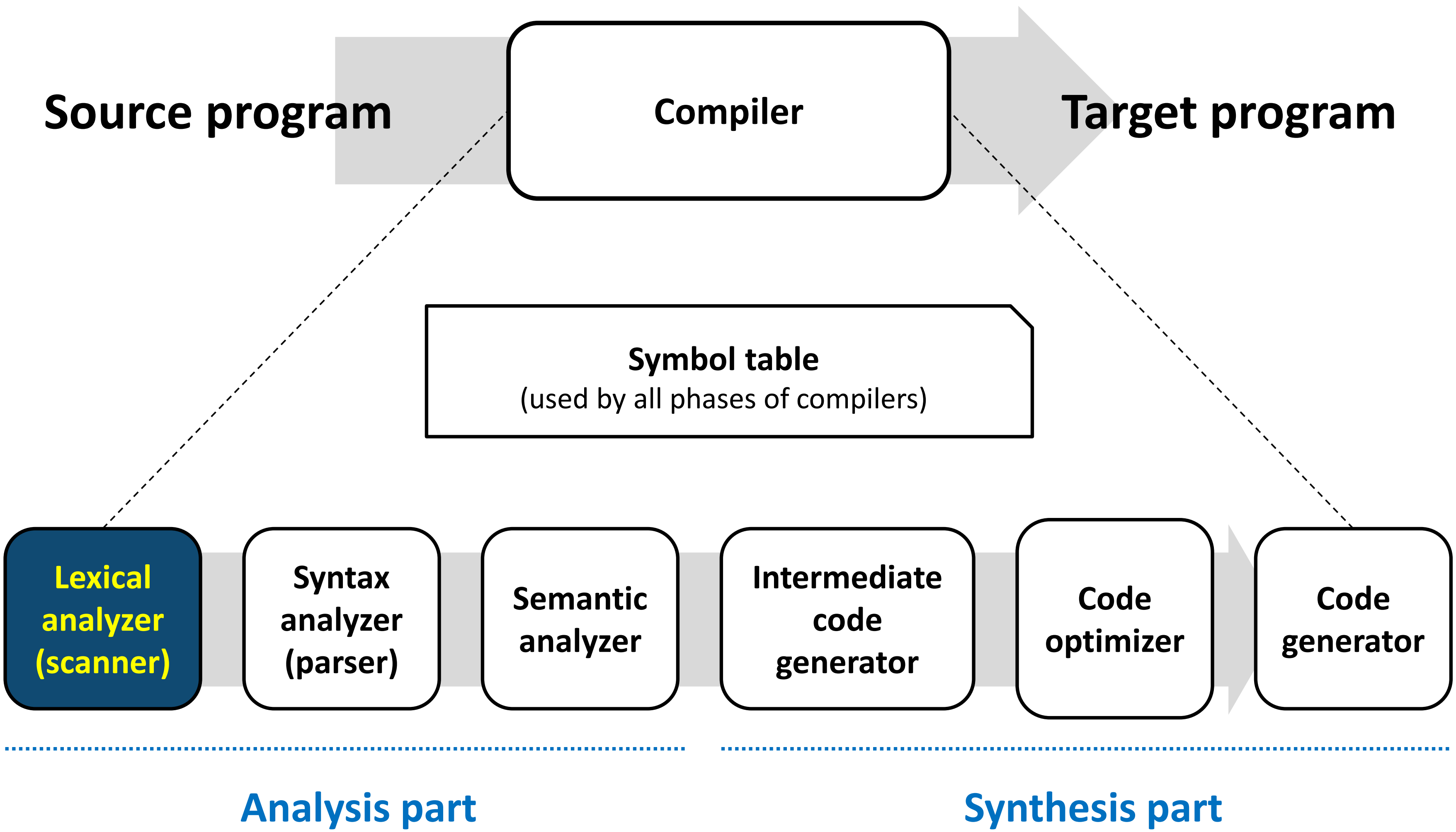
**School of Computer Science and Engineering**

**Chung-Ang University, Seoul, Korea**

https://sites.google.com/view/hyosukim

hskimhello@cau.ac.kr, hskim.hello@gmail.com

# Overview



Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part

# Overview

**What does a lexical analyzer do?**

"In this course, you will learn how to design and implement compilers"

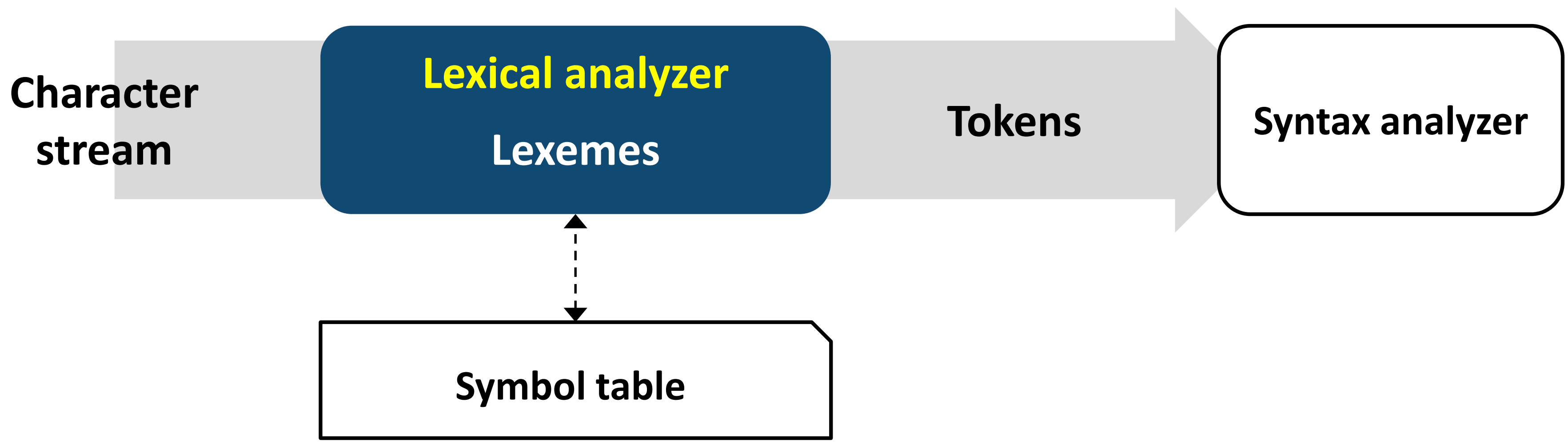
In / this / course / , / you / will / learn /

how / to / design / and / implement / compilers

# Overview

### What does a lexical analyzer do?

1. Reading the input characters of a source program
2. Grouping the characters into meaningful sequences, called **lexemes**
3. Producing a sequence of **tokens**
4. Storing the token information into a symbol table
5. Sending the tokens to a syntax analyzer



# Definition: Tokens

**A token is a syntactic category**

- Examples
  - In English: **noun, verb, adjective, …**
  - In a programming language: **identifier, number, operator, …**

- Tokens are structured as a pair consisting of a token name and an optional token value
  - e.g., for an identifier A,

    its token name is “identifier” and its token value is “A”

# Definition: Lexemes

**A lexeme is a sequence of characters that matches the pattern for a token**

- Pattern: a set of rules that defines a token

- Examples

| Token (token name) | Lexeme |
|---|---|
| IDENTIFIER (or simply ID) | pi, score, i, j, k |
| NUMBER | 0, 3.14, … |
| IF | if |
| COMMA | , |
| LPAREN | ( |
| LITERAL | “Hello world” |

# Class of tokens

- **Keyword**

  e.g., IF for if, ELSE for else, FLOAT for float, CHAR for char

- **Operators**

  e.g., ADD for +, COMPARISON for <, >, ==, and, …

- **Identifiers**

  e.g., ID for all kinds of identifiers

- **Constants**

  e.g., NUMBER for any numeric constant, INTEGER, REAL, LITERAL

- **Punctuation symbols**

  e.g., LPAREN for (, COMMA for ,

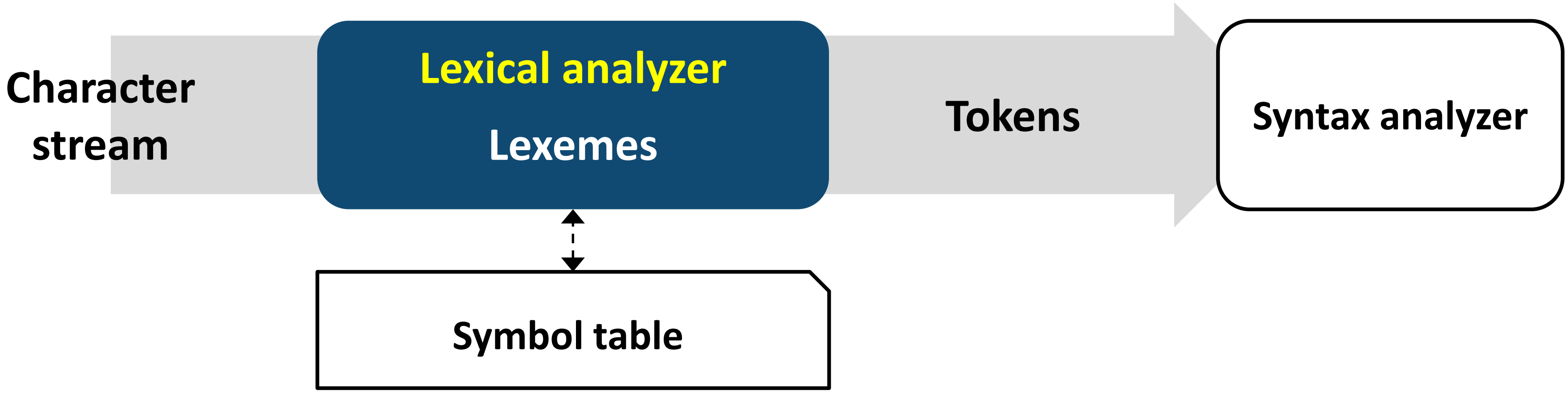
- **Whitespace**

  e.g., a non-empty sequence of blanks, newlines, and tabs

  - ❖ Lexical analyzers usually discard **uninteresting** tokens that don't contribute to parsing

    e.g., whitespace, comment

# Lexical analysis does




1. **Partitioning input strings into substrings (lexemes)!!**
2. **Identifying the token of each lexeme**

# Examples of lexical analysis

| | | | | | |
|---|---|---|---|---|---|
| **Input** | A | = | B | + | C |
| **Token name** | ID | ASSIGN | ID | ADD | ID |
| **Token value** | A or pointer to symbol-table entry for A | | B | | C |
| **Output** | <ID, A> | <ASSIGN> | <ID, B> | <ADD> | <ID, C> |

# Examples of lexical analysis

**Input**

```
bool compare(int a, int b) {
  /* compare a and b */
  return a >= b;
}
```

**Output**

# Remaining questions

**How to specify the patterns for tokens?**

**Regular languages**

**How to recognize the tokens from input streams?**

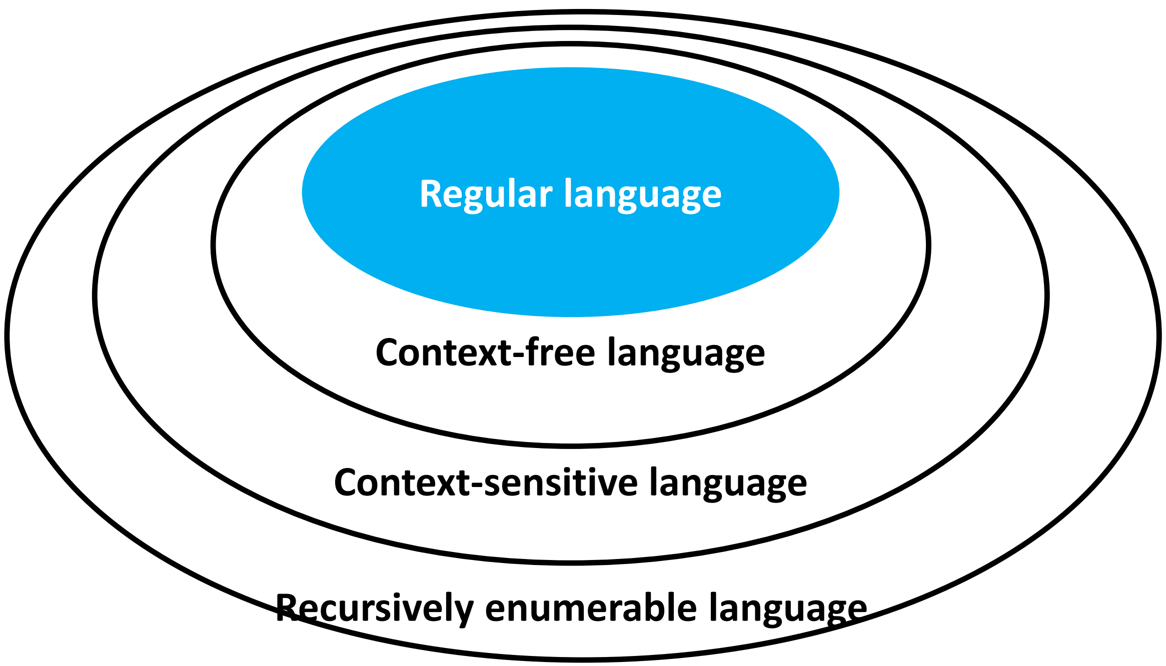
**Finite automata**

# For the specification of tokens

**Why do we use "regular languages"??**

**Simple, but powerful enough to describe the pattern of tokens**

- **The coverage of formal languages**



# Definition: Alphabet, string, and language

***An alphabet* $\Sigma$ is any finite set of symbols**

- $Letter = \Sigma^L = \{A, B, C, \dots, Z, a, b, c, \dots, z\}$
- $Digit = \Sigma^D = \{0, 1, 2, \dots, 9\}$

***A string s over alphabet* $\Sigma$ is a finite set of symbols drawn from the alphabet**

- $If\ \Sigma = \{0\},\ s = 0, 00, 000, or, \dots$
- $If\ \Sigma = \{a, b\},\ s = a, b, aa, ab, ba, bb, aaa, or, \dots$

***A language L* is any set of strings over some fixed alphabet $\Sigma$**

- $If\ \Sigma = \{a, b\},\ L_1 = \{a, ab, ba, aba\}\ and\ L2 = \{a, b, aa, ab, ba, bb, aaa, \dots\}$
  - ❖ $L_1$ is a finite language (the number of strings in the language is finite)
  - ❖ $L_2$ is an infinite language (the number of strings in the language is infinite)

# Operations on strings

| Notation | Definition | Examples / properties |
|---|---|---|
| $s$ | A string<br>(A finite set of symbols<br>over alphabet $\Sigma$) | • $If\ \Sigma = \{a, b\},\ \ s = a, b, ab, ba, aa, bb, aaa, or, \ldots$ |
| $\|s\|$ | The length of $s$<br>(The number of occurrences<br>of symbols in $s$) | • $If\ \ s = a_1 a_2 a_3 \ldots a_k,\ \ \|s\| = k$<br>• $If\ \ s = compiler,\ \ \|s\| = 8$ |
| $s_1 s_2$ | Concatenation of $s_1$ and $s_2$ | • $If\ s_1 = CSE,\ \ s_2 = @,\ \ \ and\ s_3 = CAU,$<br>$s_1 s_2 s_3 = CSE@CAU$<br>• $\|s_1 s_2\| = \|s_1\| + \|s_2\|$<br>• $s_1 s_2 \neq s_2 s_1,\ \ if\ s_1 \neq s_2$ |

# Operations on strings

| Notation | Definition | Examples / properties |
|---|---|---|
| $\epsilon$ (epsilon) | An empty string | • $\|\epsilon\| = 0$ <br> • $\epsilon s = s = s\epsilon$ <br> • $s_1 \epsilon s_2 = s_1 s_2$ |
| $s^i$ | Exponentiation of $s$ <br> (Concatenation of $s$ i-times) | • $s^0 = \epsilon$ <br> • $s^1 = s,\ s^2 = ss,\ s^3 = sss$ <br> • $s^i = s^{i-1}s,\ for\ all\ i > 0$ <br> e.g.,$s^1 = s^0 s = \epsilon s = s$ |

# Operations on languages

| Notation | Definition | Examples / properties |
|---|---|---|
| $L$ | A language<br>(A set of strings over alphabet $\Sigma$) | • $If\ \Sigma = \{a, b\},\ \ L = \{a, b, aa\}$ |
| $L_1 \cup L_2$ | Union of $L_1$ and $L_2$<br>$\{s \mid s\ is\ in\ L_1\ or\ s\ is\ in\ L_2\}$ | • $If\, L_1 = \{a, ab\}\ and\ L_2 = \{b, aa\},$<br>$L_1 \cup L_2 = \{a, b, ab, aa\}$ |
| $L_1L_2$ | Concatenation of $L_1$ and $L_2$<br>$\{s_1s_2 \mid s_1\ is\ in\ L_1\ and\ s_2\ is\ in\ L_2\}$ | • $If\ L_1 = \{a, ab\}\ and\ L_2 = \{b, aa\},$<br>$L_1L_2 = \{ab, aaa, abb, abaa\}$ |
| $L^i$ | Concatenation of $L$ $i$-times | • $If\ L = \{a, ab\},$<br>$L^0 = \{\epsilon\}, L^1 = \{a, ab\}, L^2 = \{aa, aab, aba, abab\}$<br>• $L^i = L^{i-1}L$ |

# Operations on languages

| Notation | Definition | Examples / properties |
| --- | --- | --- |
| $L^*$ | Kleene closure of L<br>(Concatenation of L<br>zero or more times) | • $L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \cdots$<br>• $If\ L = \{0\},$<br>$L^* = \{\epsilon, 0, 00, 000, 0000, \dots\}$ |
| $L^+$ | Positive closure of L<br>(Concatenation of L<br>one or more times) | • $L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \cdots$<br>• $L^+ = L^* - \{\epsilon\}$<br>• $If\ L = \{0\},$<br>$L^+ = \{0, 00, 000, 0000, \dots\}$ |

# Regular expressions

## A notation for describing regular languages

Each regular expression $r$ describes a regular language $L(r)$

### Basic regular expressions

| Regular expression | Expressed regular language |
|---|---|
| $\boldsymbol{\epsilon}$ | $L(\epsilon) = \{\epsilon\}$ |
| $\boldsymbol{a}$ | $L(a) = \{a\}$, *where $a$ is a symbol in alphabet* $\Sigma$ |
| $\boldsymbol{r_1 \mid r_2}$ | $L(r_1) \cup L(r_2)$, *where $r_1$ and $r_2$ are regular expressions* |
| $\boldsymbol{r_1 r_2}$ | $L(r_1 r_2) = L(r_1)L(r_2) = \{s_1 s_2 \mid s_1 \in L(r_1) \text{ and } s_2 \in L(r_2)\}$ |
| $\boldsymbol{r^*}$ | $L(r^*) = \bigcup_{i \geq 0} L(r^i)$ |

# Regular expressions

**An expression is a regular expression**

**If and only if it can be described by using the basic regular expressions only**

- **Q1. Is $a^+$ a regular expression over alphabet $\Sigma = \{a\}$?**
  - **Yes, $a^+ = aa^*$**
- **Q2. Is $(^n)^n$ $(0 \le n \le \infty)$ a regular expression over alphabet $\Sigma = \{(,)\}$?**

# Rules for regular expressions

- **Precedence:** exponentiation ($^*$, $^+$) > concatenation > union (|)
  - $(r_1)|\big((r_2)^*(r_3)\big) = r_1|r_2^*r_3$
- **Equivalence:** $r_1 = r_2$, if $L(r_1) = L(r_2)$
- **Algebraic laws**

| Operations | Laws |
|---|---|
| \| (union) | • Commutative: $r_1\|r_2 = r_2\|r_1$<br>• Associative: $r_1\|(r_2\|r_3) = (r_1\|r_2)\|r_3$ |
| Concatenation | • Associative: $r_1(r_2r_3) = (r_1r_2)r_3$<br>• Concatenation distributes over \|: $r_1(r_2\|r_3) = r_1r_2\|r_1r_3$ |
| $\epsilon$ | • The identity for concatenation: $r_1\epsilon = \epsilon r_1 = r_1$<br>• Always guaranteed in a closure: $r^* = (r\|\epsilon)^*$ |
| $a^*$ | • Idempotent: $a^{**} = a^*$ |

# Examples of specifying tokens

## Example 1. Keyword

- Keyword is “if”, “else”, “for”, or …
- Regular expression:

$$Keyword = if|else|for|\ldots$$

## Example 2. Comparison

- Comparison is all the operators related with comparison (e.g., <, >, <=, >=)
- Regular expression:

$$Comparison = < | > | <= | >= \cdots$$

# Examples of specifying tokens

Example 3. Whitespace

- Whitespace is a non-empty sequence of blanks, newlines, and tabs (e.g., \t, \n, \t\t, \n  )
- Regular expression:

$$Whitespace = \ |\backslash\mathrm{t}|\backslash\mathrm{n}|\backslash\mathrm{t}\backslash\mathrm{t}|\ \ldots \rightarrow (\ |\backslash\mathrm{t}|\backslash\mathrm{n}|\ \ )^{+}$$

Example 4. Integer

- Integer is a non-empty string of digits (e.g., 0, 11, 1530, … )
- Regular expression:

# Examples of specifying tokens

Example 5. Identifier

- Identifier is a non-empty string of letters or digits or ‘_’, not starting with digits

  (e.g., aaa, i, funcAtoB, _var, main, variable123, …)

- Regular expression:

# Examples of specifying tokens

Example 6. Float

- 0.5, 3.14, -6.111, 1.0E+3, 100.0, -10.53E-1, 9.123E3
- Regular expression:

# We can specify tokens

Keyword

Identifier

Comparison

Float

Whitespace

How can we recognize

these tokens from input streams?

# To recognize tokens

1. Merge the regular expression of tokens

Keyword Identifier Comparison Float Whitespace

**Merged = Keyword | Identifier | Comparison | Float | Whitespace | …**

# To recognize tokens

2. When an input stream $a_1 a_2 a_3 \dots a_n$ is given,

$$mIdx = 0;$$

$$for\ 1 \le i \le n$$

$$\quad if\ a_1 a_2 \dots a_i \in L(Merged), mIdx = i$$

$$end$$

$$partition\ and\ classify\ a_1 a_2 \dots a_{mIdx}$$

3. Do the step 2 for the remaining input stream

# To recognize tokens

2. When an input stream $a_1 a_2 a_3 \dots a_n$ is given,

$$mIdx = 0;$$

$$for\ 1 \le i \le n$$

$$\quad if\ a_1 a_2 \dots a_i \in L(Merged), mIdx = i$$

$$end$$

$$partition\ and\ classify\ a_1 a_2 \dots a_{mIdx}$$

**If $\boldsymbol{a_1 a_2 \dots a_{mIdx} \in L(Keyword)}$**

**and $\boldsymbol{a_1 a_2 \dots a_{mIdx} \in L(identifier)}$????**

**Make a priority for each token!!!**

3. Do the step 2 for the remaining input stream

# To recognize tokens

2. When an input stream $a_1 a_2 a_3 \dots a_n$ is given,

**This pseudo code also needs error handling routines (e.g., what happens if $\boldsymbol{a_1 \notin \mathrm{L}(Merged)}$)**

$$mIdx = 0;$$

$$for\ 1 \le i \le n$$

$$\quad if\ a_1 a_2 \dots a_i \in L(Merged), mIdx = i$$

$$end$$

$$partition\ and\ classify\ a_1 a_2 \dots a_{mIdx}$$

3. Do the step 2 for the remaining input stream

# To recognize tokens

2. When an input stream $a_1 a_2 a_3 \dots a_n$ is given,

$$mIdx = 0;$$

$$for\ 1 \leq i \leq n$$

$$\quad if\ \boldsymbol{a_1 a_2 \dots a_i} \in \boldsymbol{L(Merged)}, mIdx = i$$

$$end$$

$$partition\ and\ classify\ a_1 a_2 \dots a_{mIdx}$$

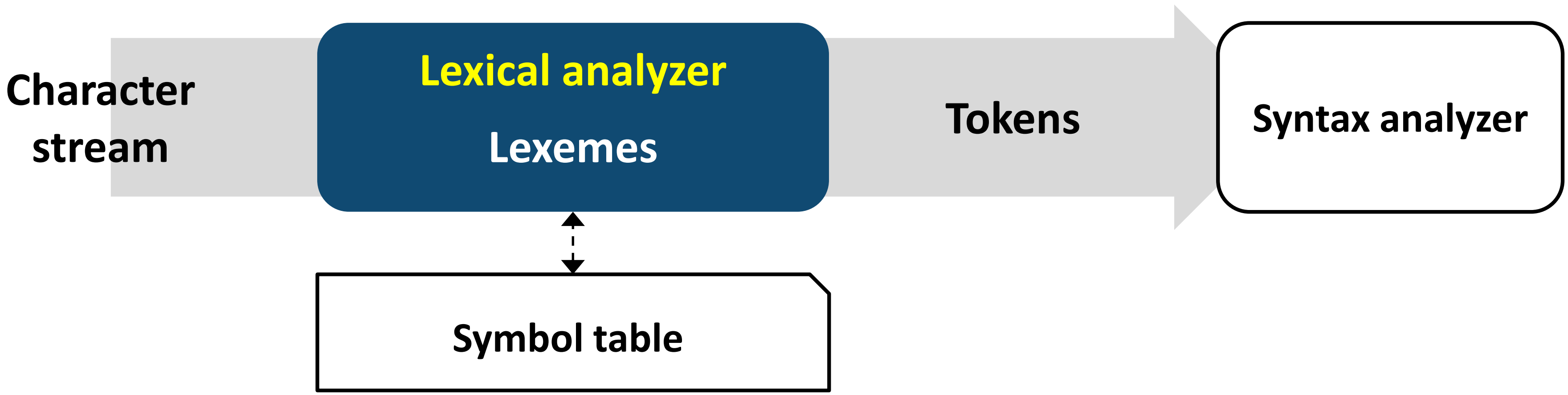
**How to check this easily???**

**Finite automata**

3. Do the step 2 for the remaining input stream

# Summary

**What does a lexical analyzer do?**




**More questions in designing lexical analyzers**

1. How to specify the patterns for tokens? **Regular languages**
2. How to recognize the tokens from input streams? **Finite automata**