

Lecture 02

Lexical Analysis

Part 1: specification of tokens

Hyosu Kim

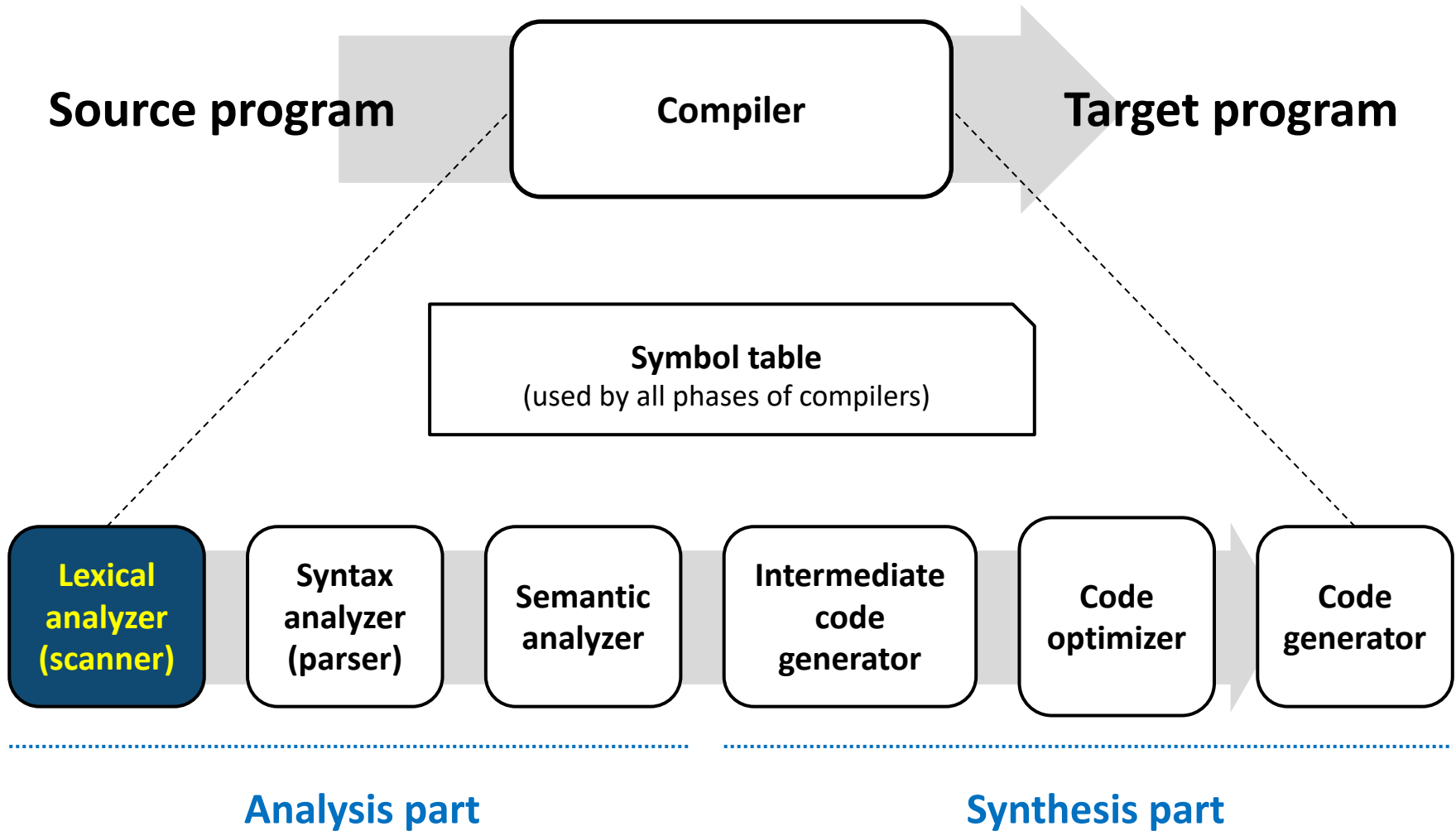
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

<https://hcslab.cau.ac.kr>

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

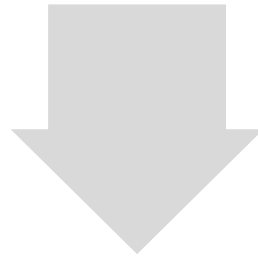
Overview



Overview

What does a lexical analyzer do?

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

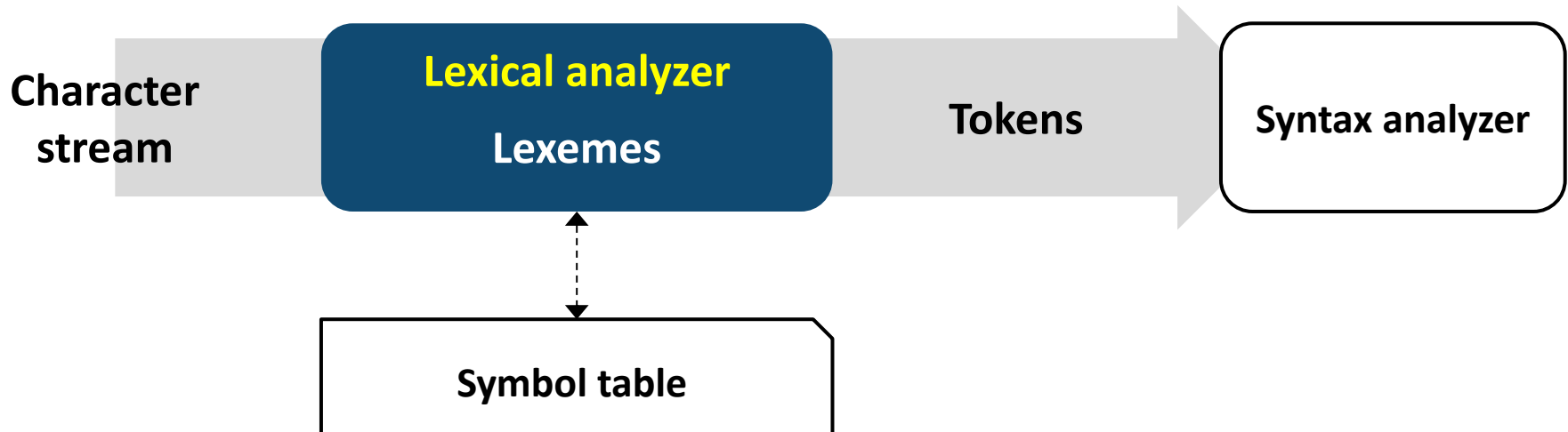


In / this / course / , / you / will / learn /
how / to / design / and / implement / compilers

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**

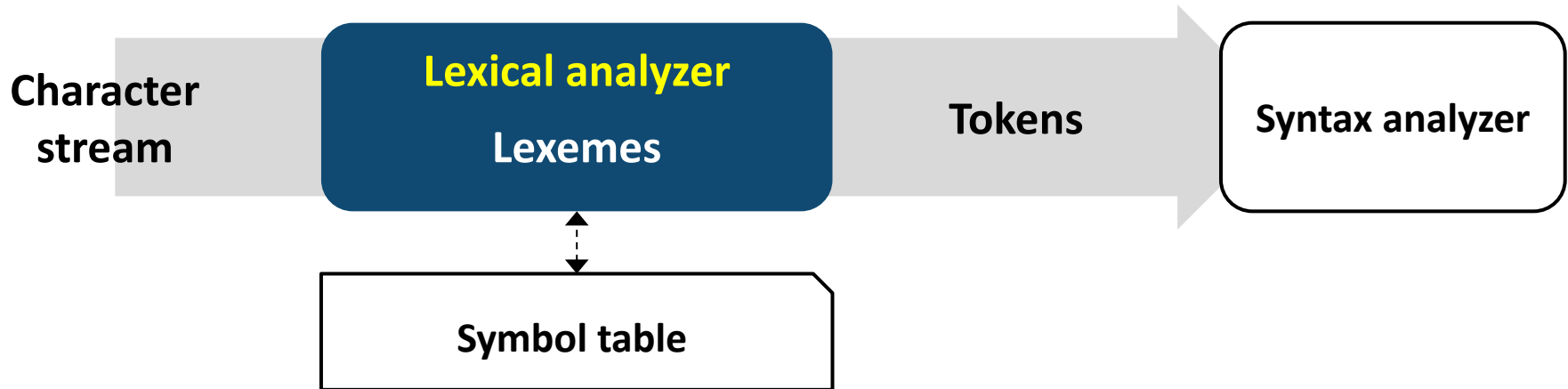
e.g., LPAREN for (, COMMA for ,

- **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?

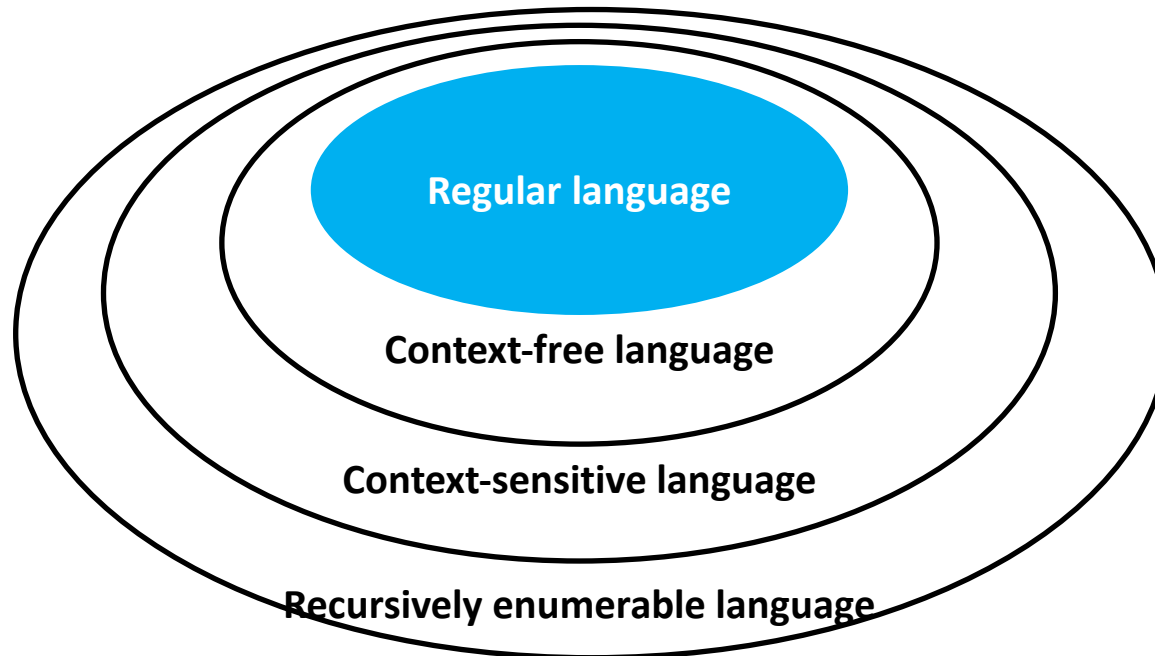
Finite automata

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** Σ 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 Σ is a finite set of symbols drawn from the alphabet

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

A **language** L is any set of strings over some fixed alphabet Σ

- If $\Sigma = \{a, b\}$, $L_1 = \{a, ab, ba, aba\}$ and $L_2 = \{a, b, aa, ab, ba, bb, aaa, \dots\}$
 - ❖ 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 (A finite set of symbols over alphabet Σ)	<ul style="list-style-type: none"> If $\Sigma = \{a, b\}$, $s = a, b, ab, ba, aa, bb, aaa, \text{or}, \dots$
$ s $	The length of s (The number of occurrences of symbols in s)	<ul style="list-style-type: none"> If $s = a_1a_2a_3 \dots a_k$, $s = k$ If $s = \text{compiler}$, $s = 8$
s_1s_2	Concatenation of s_1 and s_2	<ul style="list-style-type: none"> If $s_1 = \text{CSE}$, $s_2 = @$, and $s_3 = \text{CAU}$, $s_1s_2s_3 = \text{CSE@CAU}$ $s_1s_2 = s_1 + s_2$ $s_1s_2 \neq s_2s_1$, if $s_1 \neq s_2$

Operations on strings

Notation	Definition	Examples / properties
ϵ (epsilon)	An empty string	<ul style="list-style-type: none"> $\epsilon = 0$ $\epsilon s = s = s\epsilon$ $s_1\epsilon s_2 = s_1s_2$
s^i	Exponentiation of s (Concatenation of s i-times)	<ul style="list-style-type: none"> $s^0 = \epsilon$ $s^1 = s, s^2 = ss, s^3 = sss$ $s^i = s^{i-1}s, \text{ for all } i > 0$ e.g., $s^1 = s^0s = \epsilon s = s$

Operations on languages

Notation	Definition	Examples / properties
L	A language (A set of strings over alphabet Σ)	<ul style="list-style-type: none"> If $\Sigma = \{a, b\}$, $L = \{a, b, aa\}$
$L_1 \cup L_2$	Union of L_1 and L_2 $\{s \mid s \text{ is in } L_1 \text{ or } s \text{ is in } L_2\}$	<ul style="list-style-type: none"> If $L_1 = \{a, ab\}$ and $L_2 = \{b, aa\}$, $L_1 \cup L_2 = \{a, b, ab, aa\}$
$L_1 L_2$	Concatenation of L_1 and L_2 $\{s_1 s_2 \mid s_1 \text{ is in } L_1 \text{ and } s_2 \text{ is in } L_2\}$	<ul style="list-style-type: none"> If $L_1 = \{a, ab\}$ and $L_2 = \{b, aa\}$, $L_1 L_2 = \{ab, aaa, abb, abaa\}$
L^i	Concatenation of L i -times	<ul style="list-style-type: none"> If $L = \{a, ab\}$, $L^0 = \{\epsilon\}, L^1 = \{a, ab\}, L^2 = \{aa, aab, aba, abab\}$ $L^i = L^{i-1} L$

Operations on languages

Notation	Definition	Examples / properties
L^*	Kleene closure of L (Concatenation of L zero or more times)	<ul style="list-style-type: none"> $L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \dots$ If $L = \{0\}$, $L^* = \{\epsilon, 0, 00, 000, 0000, \dots\}$
L^+	Positive closure of L (Concatenation of L one or more times)	<ul style="list-style-type: none"> $L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \dots$ $L^+ = L^* - \{\epsilon\}$ If $L = \{0\}$, $L^+ = \{0, 00, 000, 0000, \dots\}$

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
ϵ	$L(\epsilon) = \{\epsilon\}$
a	$L(a) = \{a\}$, where a is a symbol in alphabet Σ
$r_1 r_2$	$L(r_1) \cup L(r_2)$, where r_1 and r_2 are regular expressions
r_1r_2	$L(r_1r_2) = L(r_1)L(r_2) = \{s_1s_2 s_1 \in L(r_1) \text{ and } s_2 \in L(r_2)\}$
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 $(^n)^n$ ($0 \leq n \leq \infty$) a regular expression over alphabet $\Sigma = \{(,)\}$?

Rules for regular expressions

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

Operations	Laws
$ $ (union)	<ul style="list-style-type: none"> • Commutative: $r_1 r_2 = r_2 r_1$ • Associative: $r_1 (r_2 r_3) = (r_1 r_2) r_3$
Concatenation	<ul style="list-style-type: none"> • Associative: $r_1(r_2r_3) = (r_1r_2)r_3$ • Concatenation distributes over $$: $r_1(r_2 r_3) = r_1r_2 r_1r_3$
ϵ	<ul style="list-style-type: none"> • The identity for concatenation: $r_1\epsilon = \epsilon r_1 = r_1$ • Always guaranteed in a closure: $r^* = (r \epsilon)^*$
a^*	<ul style="list-style-type: none"> • Idempotent: $a^{**} = a^*$

Examples of specifying tokens

Example 1. Keyword

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

Keyword = if|else|for| ...

Example 2. Comparison

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

Comparison = < |> | <= | >= ...

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:

$$Whitespace = |\backslash t \backslash n \backslash t \backslash t | \dots \rightarrow (\backslash t | \backslash n |)^+$$

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

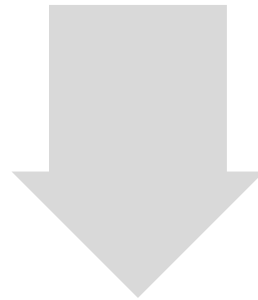
Whitespace

**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 $1 \leq i \leq n$

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 $a_1a_2a_3 \dots a_n$ is given,

$mIdx = 0;$	If $a_1a_2 \dots a_{mIdx} \in L(\textit{Keyword})$
<i>for</i> $1 \leq i \leq n$	and $a_1a_2 \dots a_{mIdx} \in L(\textit{identifier})????$
<i>if</i> $a_1a_2 \dots a_i \in L(\textit{Merged}), mIdx = i$	
<i>end</i>	Make a priority for each token!!!
<i>partition and classify</i> $a_1a_2 \dots a_{mIdx}$	

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,

This pseudo code also needs

$mIdx = 0;$

error handling routines

for $1 \leq i \leq n$

(e.g., what happens if $a_1 \notin L(Merged)$)

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 $a_1a_2a_3 \dots a_n$ is given,

$mIdx = 0;$

How to check this easily???

for $1 \leq i \leq n$

Finite automata

if $a_1a_2 \dots a_i \in L(\text{Merged}), mIdx = i$

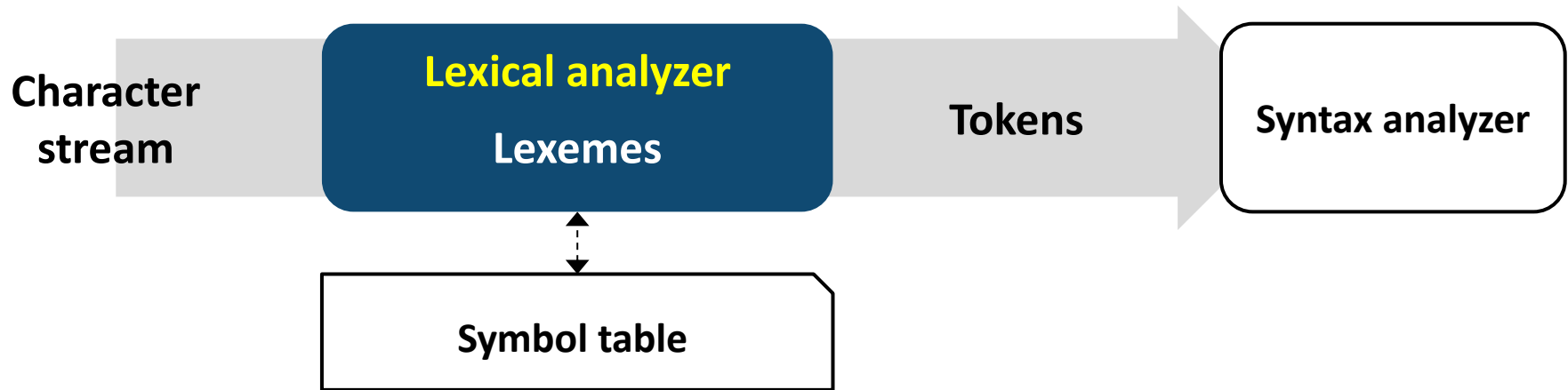
end

partition and classify $a_1a_2 \dots a_{mIdx}$

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**