Compiler Design

Dr. Sahar kamal

Lexical Analysis

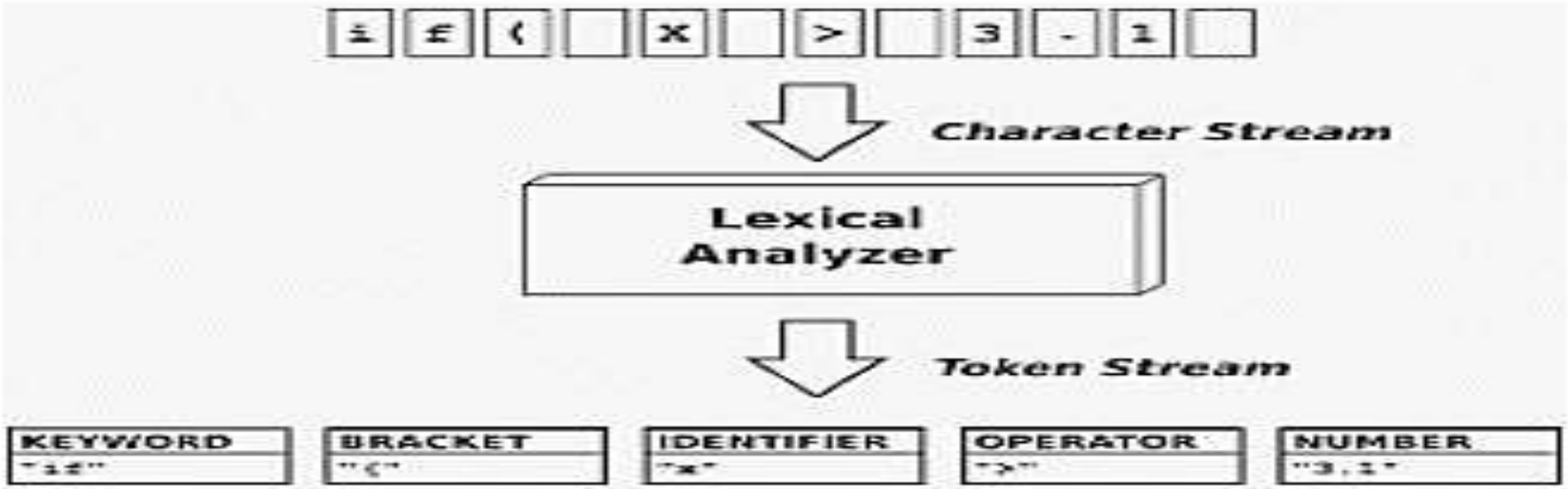
Lecture 2

Compilers *Principles, Techniques, & Tools*

Second Edition

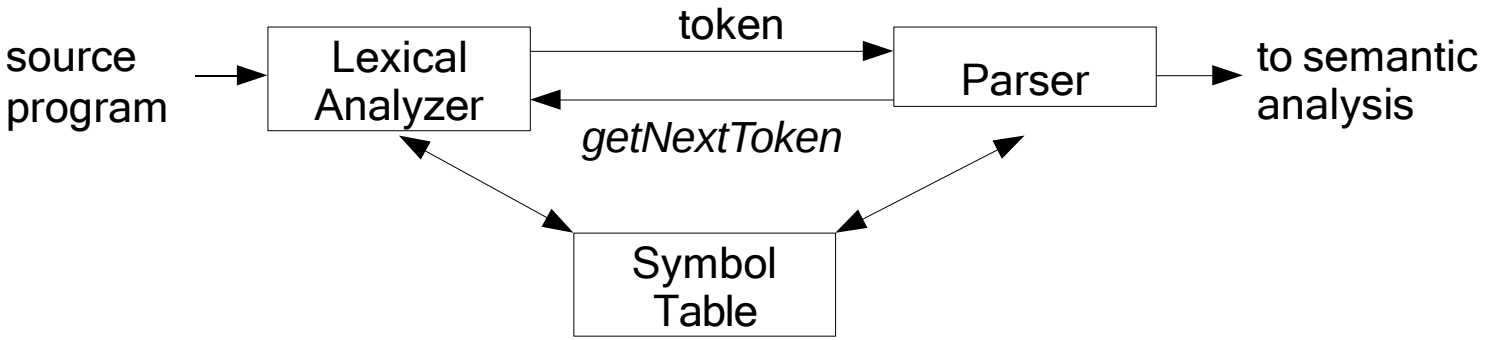
The Role of Lexical Analyzer

- It is the first phase of a compiler
- the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output tokens for each lexeme in the source program.



The Role of the Lexical Analyzer

- Read input characters, group into lexemes, produce a sequence of tokens
- May insert into symbol table (identifier)



- Strip out comments and whitespace
- Correlating error messages with source (line No.)

Scanning + lexical analysis

The Role of Lexical Analyzer

- Sometimes, lexical analyzers are divided into a cascade of two processes:

1. **Scanning** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
2. **Lexical analysis** proper is the more complex portion, which produces tokens from the output of the scanner.

Tokens, Patterns and Lexemes

- **A pattern** is a description of the form that the lexemes of a token may take [or match]. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.
- **A lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.
- **A token** is a pair consisting of a token name and an optional attribute value. Token is a sequence of characters in the input that form a meaningful word. In most languages, the tokens fall into these categories: **Keywords, Operators, Identifiers, Constants, Literal strings and Punctuation.**
- For simplicity, a token may have a single attribute which holds the required information for that token. For identifiers, this attribute a pointer to the symbol table, and the symbol table holds the actual attributes for that token. EX: < id, attr>
- Example: In the following C language statement , printf (“Total = %d\n”, score) ;
- both **printf** and **score** are lexemes matching the **pattern** for token **id**, and **"Total = %d\n‖** is a lexeme matching **literal [or string]**.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

In many languages Token is

- In many languages:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token comparison mentioned.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon

Attributes for Tokens

- We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token id, where we need to associate with the token a great deal of information. Normally, information about an identifier | e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) | is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.
- **Some attributes:**
- **<id, attr> where attr is pointer to the symbol table**
- **<assg-op,_> No attribute is needed (if there is only one assignment operator)**
- **<num, val>where val is the actual value of the number.**

Attributes for Tokens

- **Example 3.2 :** The token names and associated attribute values for the statement
- **E = M * C ** 2 ;**
- are written below as a sequence of pairs.
    1. <id, pointer to symbol-table entry for E>
    2. <assign op>
    3. <id, pointer to symbol-table entry for M>
    4. <mult op>
    5. <id, pointer to symbol-table entry for C>
    6. <exp op>
    7. <number, integer value 2>
    8. <separator(punctuation),;>

Example E=M*C**2;

Lexeme	<token, token attribute>
E	<id, pointer to symbol table entry for E>
=	<assign-op,>
M	<id, pointer to symbol table entry for M>
*	<mult-op,>
C	<id, pointer to symbol table entry for C>
**	<exp-op,>
2	<number, integer value 2>
;	<separator, >

Lexical Analysis

- **Example of Lexical Analysis, Tokens, Non-Tokens**

```
#include <stdio.h>
    int maximum(int x, int y) {
        // This will compare 2 numbers
        if (x > y)
            return x;
        else {
            return y;
        }
    }
```

Examples of Tokens created

Lexeme	Token
int	Keyword
maximum	Identifier
(	Punctation
int	Keyword
x	Identifier
,	Punctation
int	Keyword
Y	Identifier
)	Punctation
{	Punctation
If	Keyword

Lexeme	Token
(	Punctation
x	Identifier
>	Operator
y	Identifier
)	Punctation
return	Keyword
x	Identifier
;	Punctation
else	Keyword
{	Punctation
return	Keyword
y	Identifier
;	Punctation
}	punctation
}	punctation

Examples of Nontokens

Type	Examples
Comment	// This will compare 2 numbers
Pre-processor directive	#include <stdio.h>
Pre-processor directive	#define NUMS 8,9
Macro	NUMS
Whitespace	/n /b /t

Lexical Errors & **Error Recovery in Lexical Analyzer**

- **Lexical Errors**
- A Character Sequence Which Is Not Possible To Scan Into Any Valid Token Is A Lexical Error. Important Facts About The Lexical Error:

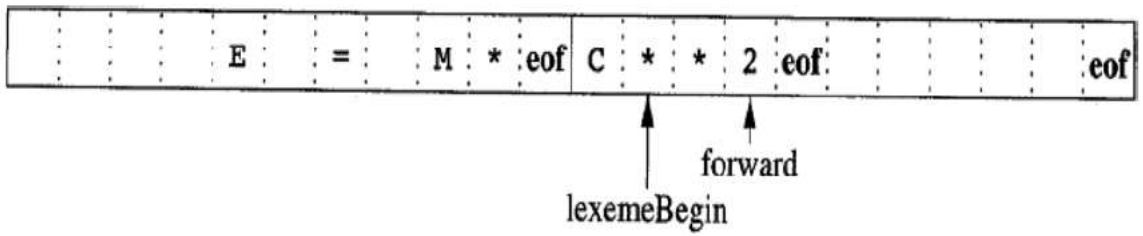
1. Lexical Errors Are Not Very Common, But It Should Be Managed By A Scanner
2. Misspelling Of Identifiers, Operators, Keyword Are Considered As Lexical Errors
3. Generally, A Lexical Error Is Caused By The Appearance Of Some Illegal Character, Mostly At The Beginning Of A Token

- **Error Recovery in Lexical Analyzer**
- **Here, are a few most common error recovery techniques:**

1. **Removes one character from the remaining input**
2. **In the panic mode, the successive characters are always ignored until we reach a well-formed token**
3. **By inserting the missing character into the remaining input**
4. **Replace a character with another character**
5. **Transpose two serial characters**

Buffer Pairs

- Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof (end of file), marks the end of the source and is different from any possible character of the source program.
- Two pointers to the input are maintained:
  1. Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  2. Pointer forward scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter



```
switch ( *forward++ ) {
        case eof:
                if (forward is at end of first buffer ) {
                        reload second buffer;
                        forward = beginning of second buffer;
                }
                else if (forward is at end of second buffer ) {
                        reload first buffer;
                        forward = beginning of first buffer;
                }
                else /* eof within a buffer marks the end of input */
                        terminate lexical analysis;
                break;
        Cases for the other characters
}
```

Strings and Languages

- **An alphabet** is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set {0,1}is the binary alphabet. ASCII is an important example of an alphabet; it is used in many software systems. Uni-code, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.
- **A string** over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms sentence" and word" are often used as synonyms for string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, **banana** is a string of length six. The empty string, denoted , is the string of length zero.
- **A language** is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like ;, the empty set ε, or the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed C programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly.

Operations on Languages

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure : Definitions of operations on languages

Example : Let L be the set of letters $\{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ and let D be the set of digits $\{\mathtt{0}, \mathtt{1}, \ldots \mathtt{9}\}$. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of Fig.

Operations on Languages

1. L U D is the set of letters and digits | strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L4 is the set of all 4-letter strings.
4. L* is the set of all strings of letters, including ε the empty string.
5. L(L U D)*is the set of all strings of letters and digits beginning with a letter.
6. D+ is the set of all strings of one or more digits.

Regular Definitions

- **Example 3.5 :** C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$
\begin{aligned}
\mathit{letter\_} &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\mathit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{id} &\rightarrow \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^*
\end{aligned}
$$

Regular Definitions

- Example 3.6 : Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits & \rightarrow & digit\ digit^{*} \\
optionalFraction & \rightarrow & \texttt{.}\ digits \mid \epsilon \\
optionalExponent & \rightarrow & (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ digits\ ) \mid \epsilon \\
number & \rightarrow & digits\ optionalFraction\ optionalExponent
\end{array}
$$

The regular definition of Example 3.6 can also be simplified:

$$
\begin{array}{rcl}
digit & \rightarrow & [\texttt{0}\texttt{-}\texttt{9}] \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\ \texttt{.}\ digits)?\ (\ \texttt{E}\ [\texttt{+}\texttt{-}]?\ digits\ )?
\end{array}
$$

Representation Of Tokens Using Regular Expression

- **Regular expression** is an important notation for specifying patterns. The term alphabet or character class denotes any finite set of symbols. Symbols are letters and characters. The set {0,1} is the binary alphabet. ASCII examples of computer alphabets.
- Regular expressions are a declarative way to describe the tokens. Describes *what* is a token, but not *how* to recognize the token
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**
- Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular Expressions (Rules)

Reg. Expr	Language it denotes
ε	$L(\varepsilon) = \{\varepsilon\}$
$a \in \Sigma$	$L(a) = \{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\,(r_2)$	$L(r_1)\,L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

Extension

$(r)^+ = (r)(r)^*$	$(L(r))^+$ Positive closure
$(r)? = (r) \mid \varepsilon$	$L(r) \cup \{\varepsilon\}$ zero or one instance
$[a_1\text{-}a_n]$	$L(a_1 \mid a_2 \mid \ldots \mid a_n)$ character class

Regular Expressions

- **We may remove parentheses by using precedence rules.**
    1. * ---> highest
    2. concatenation ---> next
    3. | --->lowest
- ab*|c means (a(b)*)|(c)

Ex:

- E= {0,1}
- 0|1 => {0,1}
- (0|1)(0|1) => {00,01,10,11}
- 0*=> {ε,0,00,000,0000,....}
- (0|1)*=> all strings with 0 and 1, including the empty string.{ε,0,1,00,000,00…0,11,111,11….1}

Regular Expressions

- Σ = {a, b}
- 1) a|b denotes …….
- 2) (a|b)(a|b) denotes …….
- 3) a* denotes …….
- 4) (a|b)* denotes
- ……..
- 5) a|a*b denotes …….

Regular Expressions

- Σ = {a, b}

1) a|b denotes {a, b}
2) (a|b)(a|b) denotes {aa, ab, ba, bb} = aa|ab|ba|bb
3) a* denotes zero or more a = {ε, a, aa, …}
4) (a|b)* denotes all strings of zero or more a or b ={ε,a, b, aa, ab, bb, ba, aaa,.}
5) a|a*b denotes {a, b, ab, aab, aaab, ...}