

## **ASSIGNMENT B2**

**TITLE:** Lexical Analysis to generate tokens.

**PROBLEM STATEMENT:**

Write a program using LEX specifications to implement lexical analysis phase of compiler to generate tokens of subset of Java program.

**OBJECTIVE:**

- Analyze source code to solve the problem
- Identify tokens required in the lexical analysis process.

**S/W PACKAGES AND HARDWARE REQUIREMENTS:**

- 64-bit open source Linux (Fedora 20)
- Eclipse IDE, JAVA
- 64-bit architecture I3 or I5 machines
- LEX

**OUTCOME:**

We will be able to

- Tokenize any given input source code.

## THEORY:

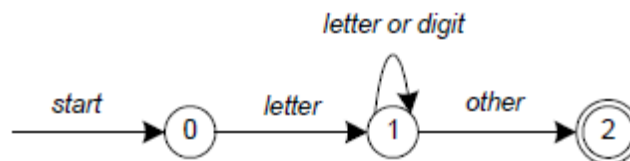
During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value. The following represents a simple pattern, composed of a regular expression that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

**letter (letter | digit)\***

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the “\*” operator
- alternation, expressed by the “|” operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.



Finite State Automate

In Figure, state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. *Any* FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

**start: goto state0**

**state0: read c**

**if c = letter goto state1**

**goto state0**

**state1: read c**

**if c = letter goto state1**

**if c = digit goto state1**

**goto state2**

**state2: accept string**

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next *input* character and *current state* the next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of lex's limitations. For example, lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "(" we push it on the stack. When a ")" is encountered we match it with the top of the stack and pop the stack. However lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures. Yacc augments an FSA with a stack and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

Pattern	Matches
.	any character except newline
\.	literal .
\n	newline
\t	tab
^	beginning of line
\$	end of line

**Table 1:** Special Characters

Pattern	Matches
?	zero or one copy of the preceding expression
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
a b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
abc	abc
abc*	ab abc abcc abccc ...
"abc*"	literal abc*
abc+	abc abcc abccc ...
a(bc)+	abc abcbcb abcbcbcb ...
a(bc)?	a abc

**Table 2:** Operators

Pattern	Matches
[abc]	one of: a b c
[a-z]	any letter a-z
[a\ -z]	one of: a - z
[-az]	one of: - a z
[A-Za-z0-9]+	one or more alphanumeric characters
[ \t\n]+	whitespace
[^ab]	anything except: a b
[a^b]	one of: a ^ b
[a b]	one of: a   b

**Table 3:** Character Class

Name	Function
<code>int yylex(void)</code>	call to invoke lexer, returns token
<code>char *yytext</code>	pointer to matched string
<code>yyleng</code>	length of matched string
<code>yylval</code>	value associated with token
<code>int yywrap(void)</code>	wrapup, return 1 if done, 0 if not done
<code>FILE *yyout</code>	output file
<code>FILE *yyin</code>	input file
<code>INITIAL</code>	initial start condition
<code>BEGIN</code>	condition switch start condition
<code>ECHO</code>	write matched string

**Table 4:** Lex Predefined Variables

Regular expressions are used for pattern matching. A character class defines a single character and normal operators lose their meaning. Two operators allowed in a character class are the hyphen (“-”) and circumflex (“^”). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

... definitions ...

%%

... rules ...

%%

... subroutines ...

Input to Lex is divided into three sections with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

%%

Input is copied to output one character at a time. The first %% is always required, as there must always be a rules section. However if we don’t specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are **stdin** and **stdout**, respectively. Here is the same example with defaults explicitly coded:

```
%%
/* match everything except newline */
. ECHO;
/* match newline */
\n ECHO;
%%
int yywrap(void) {
return 1;
}
int main(void) {
yylex();
return 0;
}
```

## CONCLUSION:

Thus, we successfully implemented Lexical Analysis to generate tokens for subset of java program.