TITLE: Lexical analyzer for sample language using LEX

THEORY:

LEX

A tool widely used to specify lexical analyzers for a variety of languages Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed. The Lex compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to lex that allow it to scan and match strings in the input. Each pattern in lex has an associated action. Typically, an action returns a token, representing the matched string, for subsequent use by the parser. To begin with, however, we will simply print the matched string rather than return a token value. We may scan for identifiers using the regular expression

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general-purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex Specifications:

A Lex program (the .l file) consists of three parts:

Declarations

%%

Translation rules

%%

Auxiliary procedures

1. The declarations section includes declarations of variables, manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14), and regular definitions.

2. The translation rules of a Lex program are statements of the form

p1 {action 1}

p2 {action 2}

p3 {action 3}

where each p is a regular expression and each action is a program fragment describing what action, the lexical analyzer should take when a pattern p matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever auxiliary procedures are needed by the actions. Alternatively, these procedures can be compiled separately and loaded with the lexical analyzer.

How does this Lexical analyzer work?

The lexical analyzer created by Lex behaves in concert with a parser in the following manner. When activated by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions p. Then it executes the corresponding action. Typically, the action will return control to the parser. However, if it does not, then the lexical analyzer proeeds to find more lexemes, until an action causes control to return to the parser. The repeated search for lexemes until an explicit return allows the lexical analyzer to process white space and comments conveniently. The lexical analyzer returns a single quantity, the token, to the parser. To pass an attribute value with information about the lexeme, we can set the global variable yylval. e.g. Suppose the lexical analyzer returns a single token for all the relational operators, in which case the parser won't be able to distinguish between "≤" , "≥" , "<" , ">" , "==" etc. We can set yylval appropriately to specify the nature of the operator.

The two variables yytext and yyleng

Lex makes the lexeme available to the routines appearing in the third section through two variables yytext and yyleng

1. yytext is a variable that is a pointer to the first character of the lexeme.

2. yyleng is an integer telling how long the lexeme is.

A lexeme may match more than one patterns. How is this problem resolved Take for example the lexeme ‘if’. It matches the patterns for both keyword if and identifier. If the pattern for keyword if precedes the pattern for identifier in the declaration list of the lex program the conflict is resolved in favor of the keyword. In general, this ambiguity-resolving strategy makes it easy to reserve keywords by listing them ahead of the pattern for identifiers.

The Lexs strategy of selecting the longest prefix matched by a pattern makes it easy to resolve other conflicts like the one between

”<” and ”≤”.

In the lex program, a main() function is generally included as:

```
main(){
        yyin=fopen(filename,”r”);
        while(yylex());
}
```

Here filename corresponds to input file and the yylex routine is called which returns the tokens.

ALGORITHM:

1) Start
2) Input file name and valid sample C file
3) Open and Read file
4) Match the tokens
5) If token is variable/identifier, make entry in symbol table with corresponding datatype and default value
6) If token not matched display error message
7) Repeat step 4) 5) 6) till all tokens are matched
8) Display the lexemes, tokens and corresponding line number
9) Display symbol table
10) Stop

/**********OUTPUT**********

```
$ lex lexicalanalyzer.l
$ gcc lex.yy.c -ll -ly
$ ./a.out
```

Enter the file name: sample1.c

LEXEME	TOKENS	LINE NO.
#	Preprocessor	1
include	Include	1
<	less than	1
stdio.h	header file	1
>	greater than	1
int	Integer datatype	2
main()	Main function	2
(	open round bracket	2
)	close round bracket	2
{	Open curly bracket	3
double	Double datatype	4
x	Identifier	4
,	comma	4
y	Identifier	4
;	semicolon	4
float	Float datatype	5
s1	Identifier	5
,	comma	5
r	Identifier	5
=	Assignment Operator	5
2.5	Float number	5
,	comma	5
i	Identifier	5

, comma 5

**********ERROR**********

; semicolon 5

**********Comment**********

char Character datatype 7

c Identifier 7

; semicolon 7

int Integer datatype 8

p Identifier 8

= Assignment Operator 8

100 Integer 8

, comma 8

n Identifier 8

= Assignment Operator 8

3 Integer 8

; semicolon 8

c Identifier 9

= Assignment Operator 9

i Identifier 9

* Operator 9

5 Integer 9

; semicolon 9

**********Comment**********

**********ERROR**********

s1 Identifier 12

= Assignment Operator 12

p Identifier 12

*	Operator	12	
n	Identifier	12	
*	Operator	12	
r	Identifier	12	
/	Operator	12	
100	Integer	12	
;	semicolon	12	
printf	Keyword		13
(	open round bracket		13
"Simple Interest is %f"	String literal		13
,	comma	13	
s1	Identifier	13	
)	close round bracket		13
;	semicolon	13	
for	Keyword		14
(	open round bracket		14
i	Identifier	14	
=	Assignment Operator		14
0	Integer	14	
;	semicolon	14	
i	Identifier	14	
<	less than	14	
10	Integer	14	
;	semicolon	14	
i	Identifier	14	
+	Operator	14	
+	Operator	14	

)	close round bracket		14
{	Open curly bracket		15
if	Keyword		16
(	open round bracket		16
i	Identifier	16	
%	Operator	16	
2	Integer	16	
=	Assignment Operator		16
=	Assignment Operator		16
0	Integer	16	
)	close round bracket		16
{	Open curly bracket		17
printf	Keyword		18
(	open round bracket		18
"\n HELLO"	String literal		18
)	close round bracket		18
;	semicolon	18	
}	Close curly bracket		19
else	Keyword		20
{	Open curly bracket		21
printf	Keyword		22
(	open round bracket		22
"\t WORLD"	String literal		22
)	close round bracket		22
;	semicolon	22	
}	Close curly bracket		23
}	Close curly bracket		24

} Close curly bracket 25

Symbol Table:

VARIABLE	VALUE	DATATYPE
x	0.0	double
y	0.0	double
s1	0.0	float
r	0.0	float
i	0.0	float
c	null	char
p	0	int
n	0	int

**********/