Experiment-3

Study of LEX and YACC Tool

Aim: Study the LEX and YACC tool and Evaluate an arithmetic expression with parentheses, unary and binary operators using Flex and Yacc. [Need to write yylex() function and to be used with Lex and yacc.].

Description:

LEX-A Lexical analyzer generator:

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers").Lex is commonly used with the yacc parser generator.

Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language

1. A lexer or scanner is used to perform lexical analysis, or the breaking up of an input stream into meaningful units, or tokens.
2. For example, consider breaking a text file up into individual words.
3. Lex: a tool for automatically generating a lexer or scanner given a lex specification (.l file).

Structure of a Lex file

The structure of a Lex file is intentionally similar to that of a yacc file; files are divided up into three sections, separated by lines that contain only two percent signs, as follows:

Definition section:
%%
Rules section:
%%
C code section:
<statements>

- The **definition section** is the place to define macros and to import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules section** is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. This is the basis of how Lex operates.

- The **C code section** contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link it in at compile time.

Description:-

The lex command reads File or standard input, generates a C language program, and writes it to a file named lex.yy.c. This file, lex.yy.c, is a compilable C language program. A C++ compiler also can compile the output of the lex command. The -C flag renames the output file to lex.yy.C for the C++ compiler. The C++ program generated by the lex command can use either STDIO or
IOSTREAMS. If the cpp define _CPP_IOSTREAMS is true during a C++ compilation, the program uses IOSTREAMS for all I/O. Otherwise, STDIO is used.

The lex command uses rules and actions contained in File to generate a program, lex.yy.c,which can be compiled with the cc command. The compiled lex.yy.c can then receive input, break the input into the logical pieces defined by the rules in File, and run program fragments contained in the actions in File.

The generated program is a C language function called yylex. The lex command stores the yylex function in a file named lex.yy.c. You can use the yylex function alone to recognize simple one-word input, or you can use it with other C language programs to perform more difficult input analysis functions. For example, you can use the lex command to generate a program that simplifies an input stream before sending it to a parser program generated by the yacc command.
The yylex function analyzes the input stream using a program structure called a finite state machine. This structure allows the program to exist in only one state (or condition) at a time. There is a finite number of states allowed. The rules in File determine how the program moves from one state to another. If you do not specify a File, the lex command reads standard input. It treats multiple files as a single file.
Note: Since the lex command uses fixed names for intermediate and output files, you can have only one program generated by lex in a given directory.

Regular Expression Basics

. : matches any single character except \n
* : matches 0 or more instances of the preceding regular expression
+ : matches 1 or more instances of the preceding regular expression
? : matches 0 or 1 of the preceding regular expression
| : matches the preceding or following regular expression
[ ] : defines a character class
() : groups enclosed regular expression into a new regular expression
“…”: matches everything within the “ “ literally

Special Functions

• yytext
– where text matched most recently is stored
• yyleng
– number of characters in text most recently matched
• yylval
– associated value of current token
• yymore()
– append next string matched to current contents of yytext
• yyless(n)
– remove from yytext all but the first n characters
• unput(c)
– return character c to input stream
• yywrap()
– may be replaced by user
– The yywrap method is called by the lexical analyser whenever it inputs an EOF as the first character when trying to match a regular expression

Files

y.output--Contains a readable description of the parsing tables and a report on conflicts generated by grammar ambiguities.
y.tab.c---- Contains an output file.
y.tab.h----- Contains definitions for token names.
yacc.tmp-----Temporary file.
yacc.debug----Temporary file.
yacc.acts-----Temporary file.
/usr/ccs/lib/yaccpar---Contains parser prototype for C programs.
/usr/ccs/lib/liby.a----Contains a run-time library.

YACC: Yet Another Compiler-Compiler

Yacc is written in portable C. The class of specifications accepted is a very general one:LALR(1) grammars with disambiguating rules.

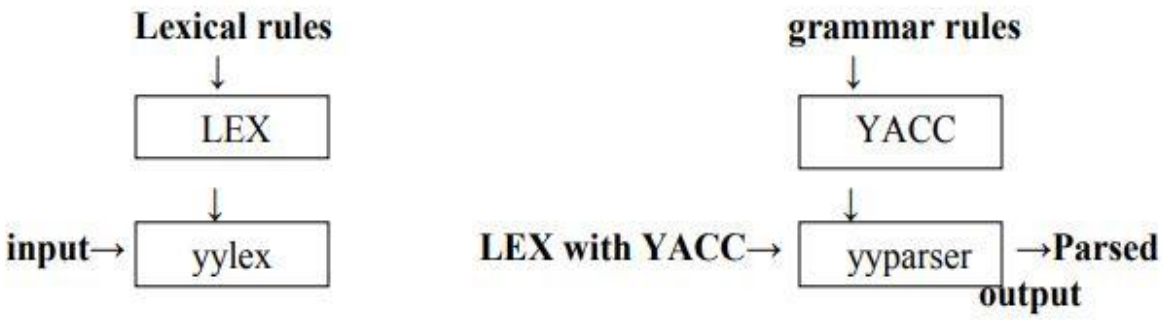
Basic specification

Names refer to either tokens or non-terminal symbols. Yacc requires tokens names to be declared as such. In addition, for reasons discussed in section 3, it is often desirable to include the lexical analyzer as part of the specification file, I may be useful to include other programs as well. Thus, the sections are separated by double percent "%%" marks. (the percent"%" is generally used in yacc specifications as an escape character). In other words, a full specification file looks like.
In other words a full specification file looks like
Declarations
%%

Rules
%%
Programs
The declaration section may be empty. More over if the programs section is omitted, the second %% mark may be omitted also thus the smallest legal yacc specification is
%%
Rules
Blanks, tabs and newlines are ignored except that they may not appear in
names or multi-character reserved symbols. Comments may appear wherever legal,
they are enclosed in /*….*/ as in C and PL/I
The rules section is made up of one or more grammar rule has the form:
A:BODY:

USING THE LEX PROGRAM WITH THE YACC PROGRAM

The Lex program recognizes only extended regular expressions and formats them into character packages called tokens, as specified by the input file. When using the Lex program to make a lexical analyzer for a parser, the lexical analyzer (created from the Lex command) partitions the input stream. The parser(from the yacc command) assigns structure to the resulting pieces. You can also use other programs along with programs generated by Lex or yacc commands.



A token is the smallest independent unit of meaning as defined by either the parser or the lexical analyzer. A token can contain data, a language keyword, an identifier or the parts of language syntax.

The yacc program looks for a lexical analyzer subroutine named yylex, which is generated by the lex command. Normally the default main program in the Lex library calls the yylex subroutines. However if the yacc command is loaded and its main program is used , yacc calls the yylex subroutines. In this case each Lex rule should end with:
return (token);

Where the appropriate token value is returned

The yacc command assigns an integer value to each token defined in the yacc grammar file through a # define preprocessor statement.

The lexical analyzer must have access to these macros to return the tokens to the parser. Use the yacc –d option to create a y.tab.h file and include the y.tab.h file in the Lex specification file by adding the following lines to the definition section of the Lex specification file:

```
%{
#include “y.tab.h”
%}
```

Alternatively you can include the lex.yy.c file the yacc output file by adding the following lines after the second %% (percent sign, percent sign) delimiter in the yacc grammar file:

```
#include”lex.yy.c”
```

The yacc library should be loaded before the Lex library to get a main program that invokes the yacc parser. You can generate Lex and yacc programs in either order.

Evaluating Arithmetic Expression using different Operators (Calculator)

Algorithm:

1) Get the input from the user and Parse it token by token.

2) First identify the valid inputs that can be given for a program.

3) The Inputs include numbers, functions like LOG, COS, SIN, TAN, etc. and operators.

4) Define the precedence and the associativity of various operators like +,-,/,* etc.

5) Write codes for saving the answer into memory and displaying the result on the screen.

6) Write codes for performing various arithmetic operations.

7) Display the possible Error message that can be associated with this calculation.

8) Display the output on the screen else display the error message on the screen.

Program:

CALC.L

```
%{
#include<stdio.h>
#include<stdlib.h>
void yyerror(char *);
```

```
#include "y.tab.h"
int yylval;
%}
%%
[a-z] {yylval=*yytext='&'; return VARIABLE;}
[0-9]+ {yylval=atoi(yytext); return INTEGER;}
CALC.Y
 [\t] ;
%%
int yywrap(void)
{
return 1;
}
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%{
int yylex(void);
void yyerror(char *);
int sym[26];
%}
%%
PROG:
PROG STMT '\n'
;
STMT: EXPR {printf("\n %d",$1);}
| VARIABLE '=' EXPR {sym[$1] = $3;}
;
EXPR: INTEGER
| VARIABLE {$$ = sym[$1];}
| EXPR '+' EXPR {$$ = $1 + $3;}
| '(' EXPR ')' {$$ = $2;}
%%
```

```
void yyerror(char *s)
{
printf("\n %s",s);
return;
}
int main(void)
{
printf("\n Enter the Expression:");
yyparse();
return 0;
}
```

Output:

```
$ lex calc.l
$ yacc -d calc.y
$ cc y.tab.c lex.yy.c -ll -ly -lm
$ . / a . out
Enter the Expression: ( 5 + 4 ) * 3
Answer: 27
```