PRACTICAL-6

AIM: Write a program to implement the functionalities of LALR Parser.

ABSTRACT:

- Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use and often are lax about checking their inputs for validity.
- YACC provides a general tool for describing the input to a computer program. The YACC user specifies the structures of his input, together with code to be invoked as each such structure is recognized. YACC turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.
- The input subroutine produced by YACC calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specifications.
- YACC is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.
- In addition to compilers for C, APL, Pascal, RATFOR, etc., YACC has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval systems and a Fortran debugging system.

INTRODUCTION:

- Each translation rule input to YACC has a string specification that resembles a production of a grammar-it has a non-terminal on the LHS and a few alternatives on RHS. For simplicity, we will refer to a string specification as a production.
- YACC generates an LALR(1) parser for language L from the productions, which is a bottom-up parser. The parser would operate as follows: For shift action it will invoke the scanner to obtain the next token and continue the parse by using token.
- While performing a reduce action in accordance with a production, it would perform the semantic action associated with that production. The semantic actions associated with productions achieve building of an intermediate representation or target code as follows: Every nonterminal symbol in the parser has an attribute. The semantic actions associated with production can access attributes of non-terminal symbols used in that production-a symbol '$n' in the semantic action, where n is an integer, designates the attribute of the nth nonterminal symbol in the RHS of the production and the symbol '$$' designates the attribute of the LHS nonterminal symbol of the production. The semantic action uses the values of these attributes for building abstract syntax trees.

YACC PROGRAM STRUCTURE:

- A YACC specification is structured along the same lines as a Lex specification.

```
 %{
        /* C declarations and includes */
  %}
      /* YACC token and type declarations */
 %%
 /* YACC Specification
   in the form of grammer rules like this:
*/
symbol   :   symbols tokens
{ $$ = my_c_code($1); }
 ;
%%
/* C language program (the rest) */
```

- The YACC Specification rules are the place where you "glue" the various tokens together that lex has conveniently provided to you.
- Each grammar rule defines a symbol in terms of:
  - other symbols
  - tokens (or terminal symbols) which come from the lexer.
- Each rule can have an associated action, which is executed after all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.

YACC GRAMMAR RULES:

- YACC rules define what is a legal sequence of tokens in our specification language. In our case, let’s look at the rule for a simple, executable menu-command:

```
menu_item    :        LABEL  EXEC
             ;
```

- This rule defines a non-terminal symbol, menu_item in terms of the two tokens LABEL and EXEC. Tokens are also known as "terminal symbols", because the parser does not need to expand them any further. Conversely, menu_item is a "non-terminal symbol" because it can be expanded into LABEL and EXEC.
- You may notice that People using UPPER CASE for terminal symbols (tokens), and lower-case for non-terminal symbols. This is not a strict requirement of YACC, but just a convention that has been established.

YACC ACTIONS:

- Actions within yacc rules take the form:

```
    menu_item       :       LABEL  EXEC  '\n'
        { C-program statements }
                      ;
```

- So far, we have only considered the tokens LABEL and EXEC as single-valued integers which are passed from the lexer to the parser. What we really need, is access to the text-strings associated with these tokens (i.e their semantic value).
- We could do this using a global variable (like token_txt in our spam-checking program), except that YACC executes the action after it has read all the tokens up to that point.
- Consider the MENU keyword, in our case. YACC has to check whether it is followed by another string or a newline, before it can decide whether it is being used to introduce a sub-menu within the same file, or an external menu-file.
- In any case, YACC provides a formal method for dealing with the semantic value of tokens. It begins with the lexer. Every time the lexer returns a value, it should also set the external variable yylval to the value of the token. YACC will then retain the association between the token and the corresponding value of yylval.
- In order to accommodate a variety of different token-types, yylval is declared as a union of different types.

Example: Write a YACC program to check whether given string is palindrome or not.

Lexical Analyzer Source Code:

```
% {

   /* Definition section */
   #include <stdio.h>
   #include <stdlib.h>
   #include "y.tab.h"
%}

/* %option noyywrap */

/* Rule Section */
%%

[a-zA-Z]+   {yylval.f = yytext; return STR;}
[-+()*/]    {return yytext[0];}
[ \t\n]     {;}

%%

 int yywrap()
 {
  return -1; }
```

Parser Source Code:

```
%{
   /* Definition section */
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
   extern int yylex();

   void yyerror(char *msg);
   int flag;

   int i;
   int k =0;
%}

%union {
   char* f;
 }

%token <f> STR
%type <f> E

/* Rule Section */
%%
S : E    {
         flag = 0;
         k = strlen($1) - 1;
         if(k%2==0){
         for (i = 0; i <= k/2; i++) {
           if ($1[i] == $1[k-i]) {
            } else {
               flag = 1;
              }
          }
         if (flag == 1) printf("Not palindrome\n");
         else printf("palindrome\n");
         printf("%s\n", $1);
        }else{

        for (i = 0; i < k/2; i++) {
          if ($1[i] == $1[k-i]) {
          } else {
              flag = 1;
             }
            }
        if (flag == 1) printf("Not palindrome\n");
        else printf("palindrome\n");
        printf("%s\n", $1);
```

```
          }
        }
  ;

E :  STR    {$$ = $1;}
  ;

%%

void yyerror(char *msg)
 {
    fprintf(stderr, "%s\n", msg);
    exit(1);
 }
//driver code
int main()
 {
    yyparse();
    return 0;
 }
```

For compiling YACC Program:

1. Write lex program in a file file.l and yacc in a file file.y

2. Open Terminal and Navigate to the Directory where you have saved the files.

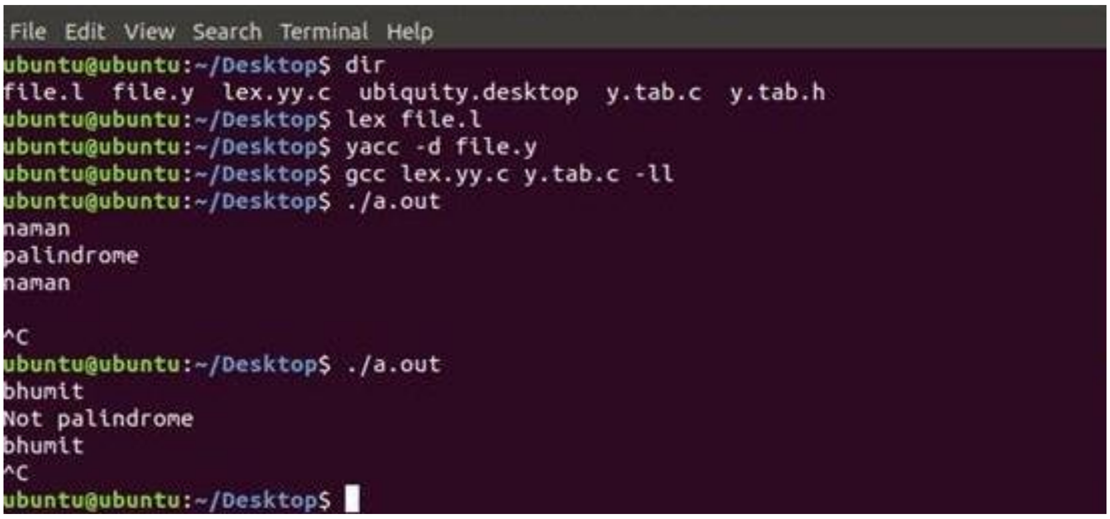
3. type lex file.l

4. type yacc file.y

5. type cc lex.yy.c y.tab.h -ll

6. type ./a.out

OUTPUT:



```
ubuntu@ubuntu:~/Desktop$ dir
file.l  file.y  lex.yy.c  ubiquity.desktop  y.tab.c  y.tab.h
ubuntu@ubuntu:~/Desktop$ lex file.l
ubuntu@ubuntu:~/Desktop$ yacc -d file.y
ubuntu@ubuntu:~/Desktop$ gcc lex.yy.c y.tab.c -ll
ubuntu@ubuntu:~/Desktop$ ./a.out
naman
palindrome
naman

^C
ubuntu@ubuntu:~/Desktop$ ./a.out
bhumit
Not palindrome
bhumit
^C
ubuntu@ubuntu:~/Desktop$
```