Assignment No.2

AIM:

Parser generator using YACC (Yet Another Compiler Compiler)

THEORY:

YACC

Each string specification in the input to YACC resembles a grammar production. The parser generated by YACC performs reductions according to this grammar. The actions associated with a string specification are executed when a reduction is made according to the specification. An attribute is associated with every nonterminal symbol. The value of this attribute can be manipulated during parsing. The attribute can be given any user-designed structure. A symbol '$n' in the action part of a translation rule refers to the attribute of the n^{th} symbol in the RHS of the string specification. '$$' represents the attribute of the LHS symbol of the string specification.

Example

Figure 3 shows sample input to YACC. The input consists of four components, of which only two are shown. The routine gendesc builds a descriptor containing the name and type of an id or constant. The routine gencode takes an operator and the attributes of two operands, generates code and returns with the attribute

```
%%
   E    :      E+T          {$$ = gencode('+', $1, $3);}
        |      T            {$$ = $1;}
   T    :      T*V          {$$ = gencode('*', $1, $3);}
        |      V            {$$ = $1;}
   V    :      id           {$$ = gendesc($1);}
%%
gencode (operator, operand_1, operand_2){  }
gendesc(symbol) {   }
```

Figure 3: A sample YACC specification

Parsing of the* string b+c*d where b, c and d are of type real, using the parser generated by YACC from the input of above Figure leads to following-calls on the C routines:

Gendesc (id#1);

Gendesc (id#2);

Gendesc (id#3);

Gencode (*, [c,real], [d, real]);

Gencode (+, [b, real], [t, real]);

where an attribute has the f*orm* <*name*>, <*type*> and t is the name of a location used to store the result of c*d in the code generated by the first call on gencode.

Compiling the Example Program

To create the desk calculator example program, do the following:

1. Process the **yacc** grammar file using the **-d** optional flag (which informs the **yacc** command to create a file that defines the tokens used in addition to the C language source code):

   yacc -d calc.yacc

2. Use the **ls** command to verify that the following files were created:

   y.tab.c

   The C language source file that the **yacc** command created for the parser

   y.tab.h

   A header file containing define statements for the tokens used by the parser

3. Process the **lex** specification file:

   lex calc.lex

4. Use the **ls** command to verify that the following file was created:

   lex.yy.c

   The C language source file that the **lex** command created for the lexical analyzer

5. Compile and link the two C language source files:

   cc y.tab.c lex.yy.c

6. Use the **ls** command to verify that the following files were created:

   y.tab.o

   The object file for the **y.tab.c** source file

   lex.yy.o

   The object file for the **lex.yy.c** source file

   a.out

   The executable program file

To run the program directly from the **a.out** file, type:

$ a.out

PROGRAM:

YACC program

The following example shows the contents of the **calc.yacc** file. This file has entries in all three sections of a **yacc** grammar file: declarations, rules, and programs.

```
%{
#include <stdio.h>
int regs[26];
int base;
%}

%start list
%union { int a; }
%type <a> expr number
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /*supplies precedence for unary minus */

%%                  /* beginning of rules section */

list:                   /*empty */
        |
        list stat '\n'
        |
        list error '\n'
```

```
        {
          yyerrok;
        } ;
stat:    expr
        {
          printf("%d\n",$1);
        }
        |
        LETTER '=' expr
        {
          regs[$1] = $3;
        } ;
expr:    '(' expr ')'
        {
          $$ = $2;
        }
        |
        expr '*' expr
        {

          $$ = $1 * $3;
        }
        |
        expr '/' expr
        {
          $$ = $1 / $3;
        }
```

```
|
expr '%' expr
{
  $$ = $1 % $3;
}
|
expr '+' expr
{
  $$ = $1 + $3;
}
|
expr '-' expr
{
  $$ = $1 - $3;
}
|
expr '&' expr
{
  $$ = $1 & $3;
}
|
expr '|' expr
{
  $$ = $1 | $3;
}
|
'-' expr %prec UMINUS
```

```
        {
          $$ = -$2;
        }
        |
        LETTER
        {
          $$ = regs[$1];
        }
        |
        number
        ;
number:  DIGIT
        {
          $$ = $1;
          base = ($1==0) ? 8 : 10;
        }       |
        number DIGIT
        {
          $$ = base * $1 + $2;
        }  ;


%%
main()
{
 return(yyparse());
}
```

```
yyerror(s)
char *s;
{
  fprintf(stderr, "%s\n",s);
}

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

The file contains the following sections:

- **Declarations Section.** This section contains entries that:
  - Include standard I/O header file
  - Define global variables
  - Define the list rule as the place to start processing
  - Define the tokens used by the parser
  - Define the operators and their precedence
- **Rules Section.** The rules section defines the rules that parse the input stream.
  - %start - Specifies that the whole input should match **stat**.
  - %union - By default, the values returned by actions and the lexical analyzer are integers. **yacc** can also support values of other types, including structures. In addition, **yacc** keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The **yacc** value stack is declared to be a union of the various types of values desired. The user declares the union, and associates union member names to each token and

nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, **yacc** will automatically insert the appropriate union name, so that no unwanted conversions will take place.

- %type - Makes use of the members of the **%union** declaration and gives an individual type for the values associated with each part of the grammar.
- %token - Lists the tokens which come from lex tool with their type.

- **Programs Section.** The programs section contains the following subroutines. Because these subroutines are included in this file, you do not need to use the **yacc** library when processing this file.

main	The required main program that calls the **yyparse** subroutine to start the program.
yyerror(s)	This error-handling subroutine only prints a syntax error message.
yywrap	The wrap-up subroutine that returns a value of 1 when the end of input occurs.

LEX program

This file contains include statements for standard input and output, as well as for the **y.tab.h** file. If you use the **-d** flag with the **yacc** command, the **yacc** program generates that file from the **yacc** grammar file information. The **y.tab.h** file contains definitions for the tokens that the parser program uses. In addition, the

calc.lex file contains the rules to generate these tokens from the input stream. The following are the contents of the **calc.lex** file.

```
%{
 #include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" "       ;
[a-z]     {
            c = yytext[0];
            yylval = c - 'a';
            return(LETTER);
          }
[0-9]     {
            c = yytext[0];
            yylval = c - '0';
            return(DIGIT);
          }
[^a-z0-9\b]    {
                    c = yytext[0];
                    return(c);
                }
```

INPUT:

2+3

OUTPUT:

5

CONCLUSION:

- ✓ YACC file has entries in three sections of a yacc grammar file: declarations, rules, and programs.
- ✓ The lexical analyzer file contains include statements for standard input and output, as well as for the **y.tab.h** file. If you use the **-d** flag with the **yacc** command, the **yacc** program generates that file from the **yacc** grammar file information.
- ✓ The **y.tab.h** file contains definitions for the tokens that the parser program uses. In addition, the **lexical analyzer** file contains the rules to generate these tokens from the input stream.
- ✓ Thus the parser is implemented using YACC

REFERENCES:

- Compilers - Principles, Techniques and Tools - A.V. Aho, R. Shethi and J. D. Ullman (Pearson Education)
- System Programming and operating systems – 2nd Edition D.M.Dhamdhere (TMGH)
- https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/com.ibm.aix.genprogc/ie_prog_4lex_yacc.htm