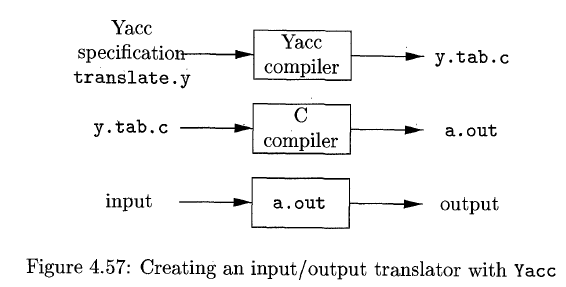
**Experiment No.: 5 Date: 09/11/2020**

**YACC**

**Aim:** To study YACC and write YACC programs to parse an expression for a given grammar.

**Theory:**

A parser generator can be used to facilitate the construction of the front end of a compiler. YACC is a LALR parser generator. It is widely available. YACC stands for “Yet Another Compiler-Compiler”. It is available as a command on the UNIX system. A translator can be constructed using YACC as shown below



1. A file, say translate.y, containing a YACC specification of the translator is prepared.
2. The UNIX system command yacc –d translate.y transforms the file translate.y into a C program called y.tab.c using the LALR method.
3. Compiling y.tab.c: gcc y.tab.c we obtain the desired object program a.out that performs the translation specified by the original YACC program.
4. run as ./a.out

The -d option will cause the header file y.tab.h to be written. The #define statements that associate the token numbers assigned by YACC with the user-declared token names. This allows source files other than y.tab.c to access the token codes.

The program y.tab.c is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. The LALR parser is compacted. All the grammars in the grammar specification will be automatically converted into C statements by the YACC tool and will be put under the function name of yyparse(). A YACC source program has three parts:

Declarations

%%

Translation rules

%%

Supporting C routines

“expt5\_DeskCalc.y” is a simple desk calculator. The calculator reads arithmetic expressions of the form 5+5, 4\*5, or (6); evaluates it, and then prints its numeric value. The desk calculator is built starting with the following grammar for arithmetic expressions:



The token digit is a single digit between 0 and 9. The lexical analyser reads input characters one at a time using the C-function getchar(). If the character is a digit, the value of the digit is stored in the variable yylval, and the token DIGIT is returned. Otherwise, the character itself is returned as the token.

**The Declarations Part** –

There are two optional sections in the declarations part of a YACC program.

In the first section, we put ordinary C declarations, delimited by %{ and %}. Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections. The program contains the include-statements that causes the C preprocessor to include the standard header files. (cytype.h for the predicate isdigit). Whatever is written in between “%{“ and “%}” are C codes, and they are copied to the beginning of the generated C files.

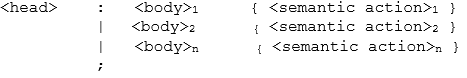
The second section of the definitions section, is outside the “%{“ and “%}” and just above the first “%%”. Here are placed declarations of grammar tokens. The statement %token DIGIT declares DIGIT to be a token. Tokens declared in this section can then be used in the second and third parts of the YACC specification. Much of the additional information required to resolve ambiguities in the CFG for the target language is provided here

**The Translation Rules Part** –

Translation rules are put after the first %% pair. Each rule consists of a grammar production and the associated semantic action. A production of the form:

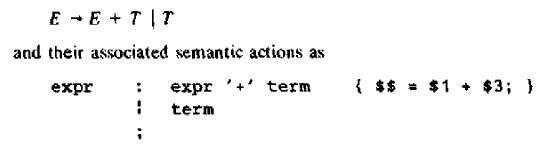


would be written in YACC as



In a YACC production, unquoted strings of letters and digits not declared to be tokens are taken to be non-terminals. A quoted single character, e.g. ‘c’ is taken to be the terminal symbol c, as well as the integer code for the token represented by that character.

**Alternative** bodies can be separated by a **vertical bar**, and a semicolon follows each head with its alternatives and their semantic actions. The first head is taken to be the **start** symbol. A YACC semantic action is a sequence of C statements. In a semantic action, the symbol **$$** refers to the **attribute value** associated with the non-terminal of the head. $i refers to the value associated with the ith grammar symbol (terminal or nonterminal) of the body. The semantic action is performed whenever we reduce by the associated production. So normally the semantic action computes a value for $$ in terms of the $i’s. *In general, $$ = $1 is the default semantic action.*



The NT term in the first production is the third grammar symbol on the right, while ‘+’ is the second. The semantic action associated with the first production adds the value of the expr and the term on the right and assigns the sum as the value of the NT expr on the left. The semantic action associated with the second production is omitted since copying the value is the default action for productions with a single grammar symbol on the right.

Note: the new starting production to the YACC specification



This production says that an input to the desk calculator is to be an expression followed by a newline character. The semantic action associated with this production prints the decimal value of the expression followed by a newline character.

**The supporting C-Routines Part** –

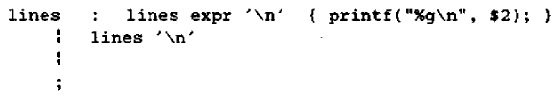
The third part of a YACC specification consists of supporting C-routines. A lexical analyzer by the name yylex() must be provided. Error recovery routines may be added, as necessary. The lexical analyzer yylex() produces tokens consisting of a token name and its associated attribute value. If a token name such as DIGIT is returned, the token name must be declared in the first section of the YACC specification. The attribute value associated with a token is communicated to the parser through a YACC-defined variable yylval.

**Using YACC with ambiguous grammars**

* Modifying the YACC specification so that the desk calculator can :

1) evaluate a sequence of expressions, one to a line; while allowing blank lines

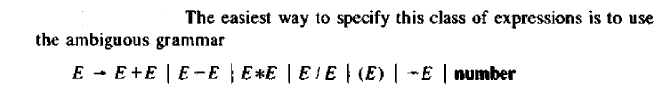
between expressions.



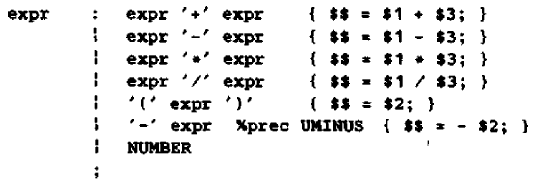
An empty alternative, as the third line is, denotes ε

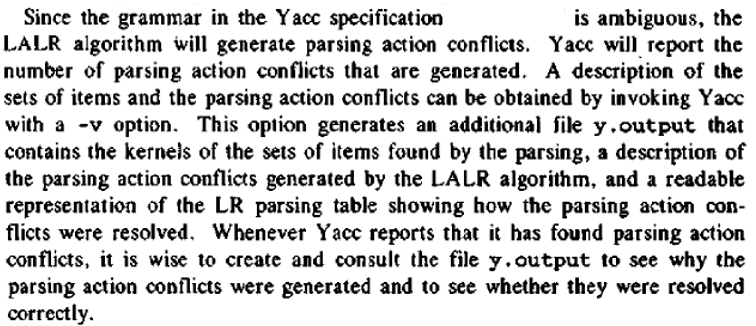
2) Evaluate numbers instead of single digits and include arithmetic operators +, - (both

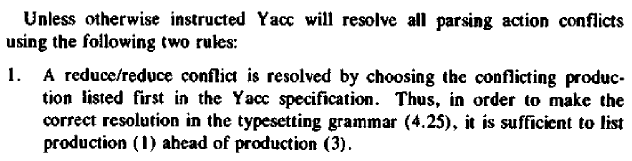
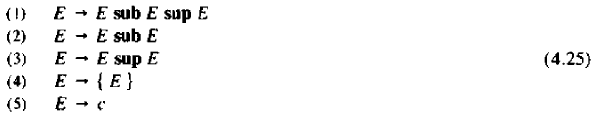
binary and unary), \*, and /



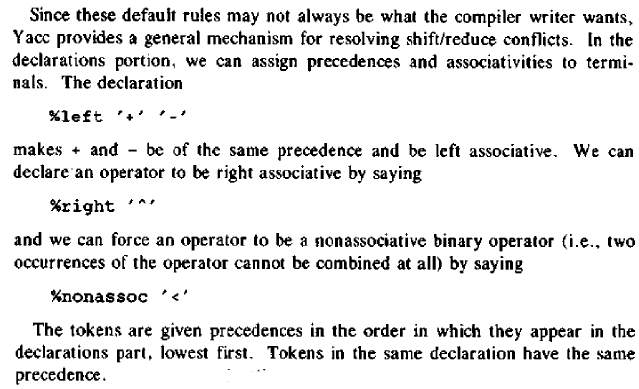
The YACC rule is

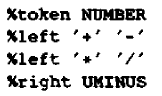


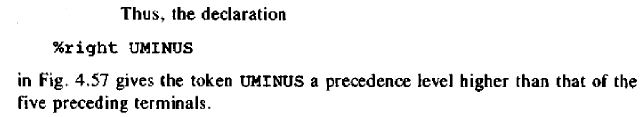


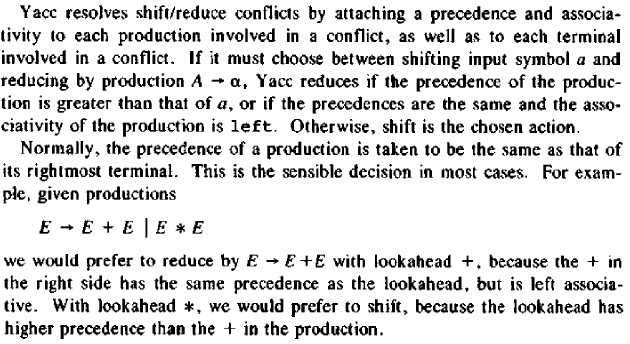
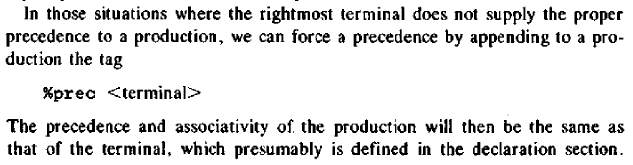


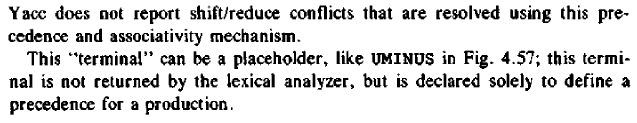


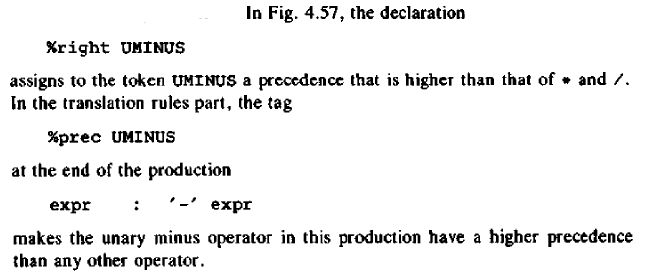
****

****

****

****

****



**Error Recovery in YACC**

Error recovery uses a form of error productions. The user decides what major NTs will have error recovery associated with them. The user then adds to the grammar error productions of the form



Where:

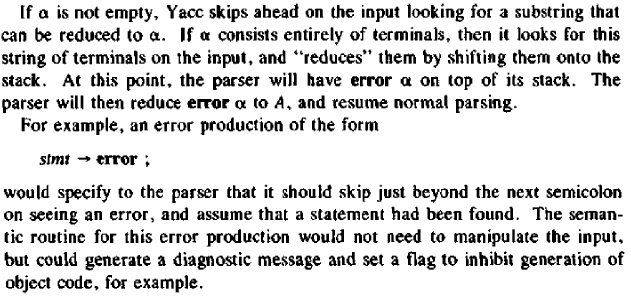
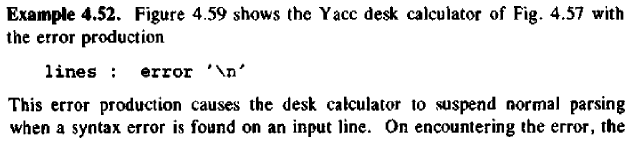
A is a major NT

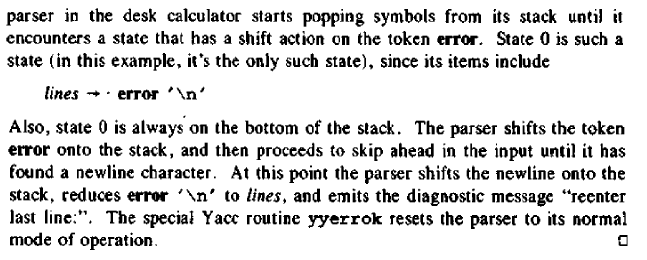
α is a string of grammar symbols, perhaps the empty string.

**error** is a YACC reserved word.

YACC will generate a parser from such a specification, treating the error productions as ordinary productions. However, when the parser generated by YACC encounters an error, it treats the states whose “sets of items” contain error productions in a special way. On encountering an error, YACC pops symbols from its stack until it finds the topmost state on its stack whose underlying set of items includes an item of the form 

The parser then “shifts” a fictitious token **error** onto the stack, as though it saw the token **error** on its input.





Make the stack hold double values by setting YYSTYPE. Yydebug=1 to enable debugging

**References:**

1. Vinu V. Das ; Compiler design with FLEX and YACC; PHI publication, ISBN:978- 81-203-3251-5
2. Aho, Ulman and Sethi; Compilers, Principles, techniques and tools; Pearson Education Asia, ISBN: 81-7808-046-X.

**PROGRAMS**

1. **DESKCALC**

//expt5\_prog1.y

%{

#include<ctype.h>

#include<stdio.h>

%}

%token DIGIT

%%

line : expr '\n' {printf("%d\n",$1);}

;

expr : expr '+' term { $$ = $1 + $3; }

| term

;

term : term '\*' factor { $$ = $1 \* $3; }

| factor

;

factor : '(' expr ')' { $$ = $2; }

| DIGIT

;

%%

int yylex(void)

{

int c;

c = getchar();

if(isdigit(c))

{

yylval = c-'0';

return DIGIT;

}

return c;

}

int yywrap(){return 1;}

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

int main(void)

{

return yyparse();

}

**OUTPUT:**

root@kali:~# vi exp5prgm1.y

root@kali:~# yacc -d exp5prgm1.y

root@kali:~# gcc y.tab.c

root@kali:~# ./a.out

2\*3

6

1. **ADVDESKCALC**

//expt5\_prog2.y

%{// AdvDeskCalc.y

#include<ctype.h>

#include<stdio.h>

#define YYSTYPE double /\* double type for YACC stack \*/

%}

%token NUMBER

%left '+' '-'

%left '\*' '/'

%right UMINUS

%%

lines : lines expr '\n' {printf("\n%g\n",$2);}

| lines '\n'

|/\*empty\*/

;

expr : expr '+' expr { $$ = $1 + $3; }

| expr '-' expr { $$ = $1 - $3; }

| expr '\*' expr { $$ = $1 \* $3; }

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

| '(' expr ')' { $$ = $2; }

| '-' expr %prec UMINUS { $$ = -$2; }

| NUMBER

;

%%

int yylex(void)

{

int c;

while( ( c = getchar() ) == ' ');

if( (c == '.') || (isdigit(c) ) ) {

ungetc(c, stdin);

scanf("%lf",&yylval);

return NUMBER;

}

return c;

}

int yywrap(){return 1;}

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

int main(void)

{

return yyparse();

}

**OUTPUT:**

root@kali:~# vi exp5prgm2.y

root@kali:~# yacc -d exp5prgm2.y

root@kali:~# gcc y.tab.c

root@kali:~# ./a.out

(2\*3)+6

12

(9/3)+(2\*4)-2

9

1. **ERRRECDESKCALC**

//expt5\_prog3.y

%{//expt5\_ErrRecDeskCalc.y

#include<ctype.h>

#include<stdio.h>

#define YYSTYPE double /\* double type for YACC stack \*/

%}

%token NUMBER

%left '+' '-'

%left '\*' '/'

%right UMINUS

%%

lines : lines expr '\n' { printf("%g\n",$2); }

| lines '\n'

|/\*empty\*/

|error '\n' { yyerror("Re-enter previous line: ");

yyerrok; }

;

expr : expr '+' expr { $$ = $1 + $3; }

| expr '-' expr { $$ = $1 - $3; }

| expr '\*' expr { $$ = $1 \* $3; }

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

| '(' expr ')' { $$ = $2; }

| '-' expr %prec UMINUS { $$ = -$2; }

| NUMBER

;

%%

int yylex(void)

{

int c;

while( ( c = getchar() ) == ' ');

if( (c == '.') || (isdigit(c) ) )

{

ungetc(c, stdin);

scanf("%lf",&yylval);

return NUMBER;

}

return c;

}

int yywrap(){return 1;}

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

int main(void)

{

return yyparse();

}

**OUTPUT:**

root@kali:~# vi exp5prgm3.y

root@kali:~# yacc -d exp5prgm3.y

root@kali:~# gcc y.tab.c

root@kali:~# ./a.out

2+3

5

4-

syntax error

Re-enter previous line:

4-2

2

3\*

syntax error

Re-enter previous line:

3\*2

6

**Conclusion:** YACC programs were successfully implemented