

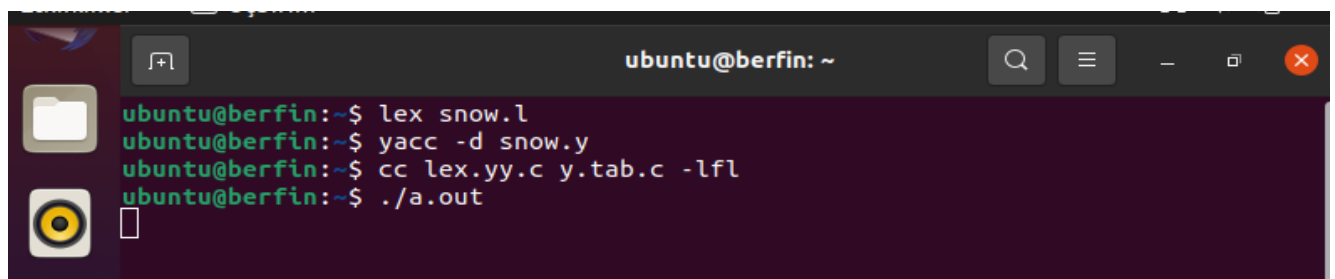
1) The steps to test my analyzer :

\$ lex snow.l

\$ yacc -d snow.y

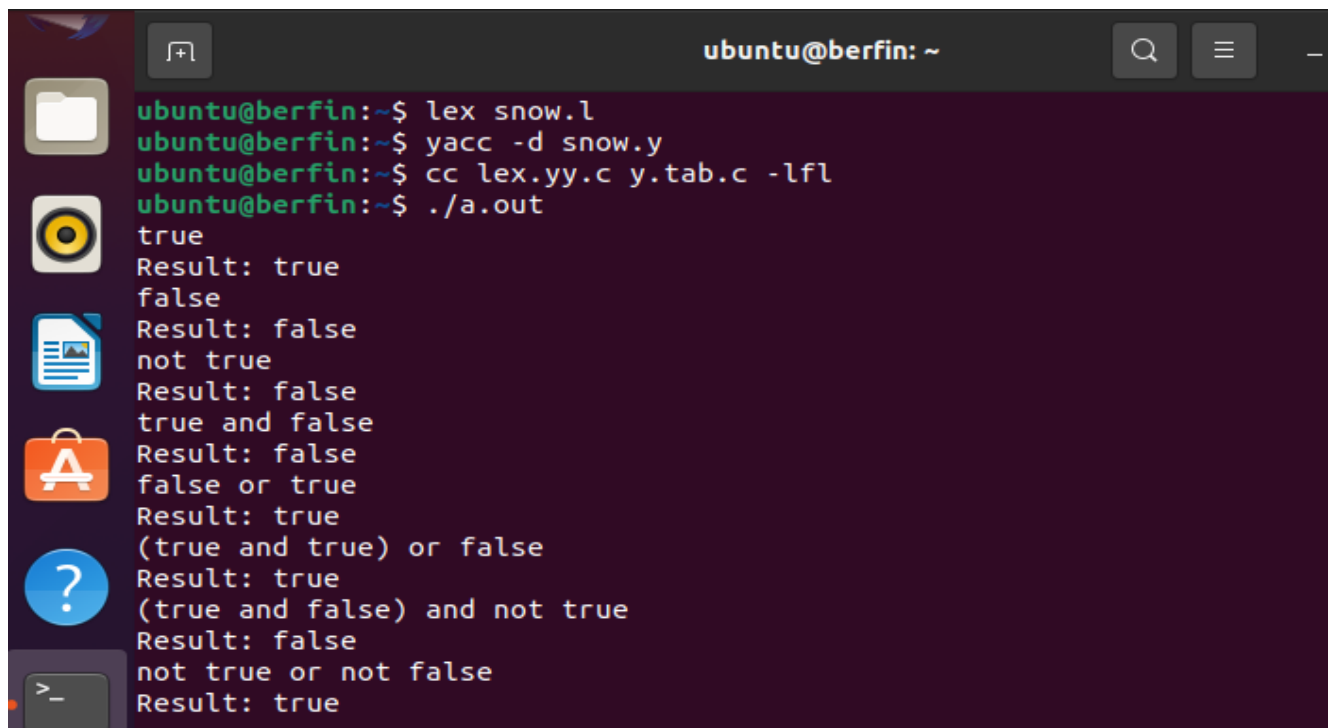
\$ cc lex.yy.c y.tab.c -lfl

\$./a.out



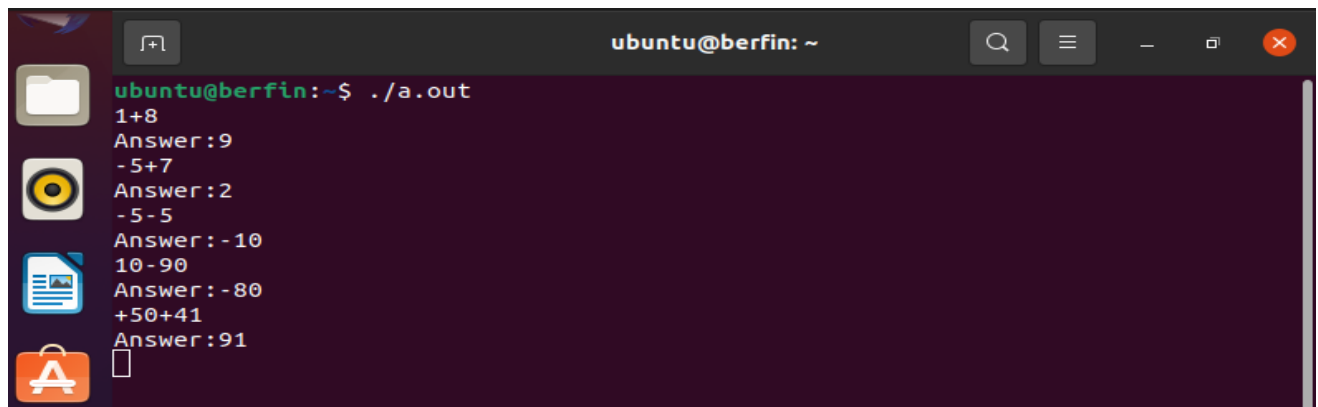
```
ubuntu@berfin: ~  
ubuntu@berfin:~$ lex snow.l  
ubuntu@berfin:~$ yacc -d snow.y  
ubuntu@berfin:~$ cc lex.yy.c y.tab.c -lfl  
ubuntu@berfin:~$ ./a.out
```

2) The program knows the concepts of “true” and “false” and can find the results of combinations of propositions. This program knows the precedence and the use of parentheses.



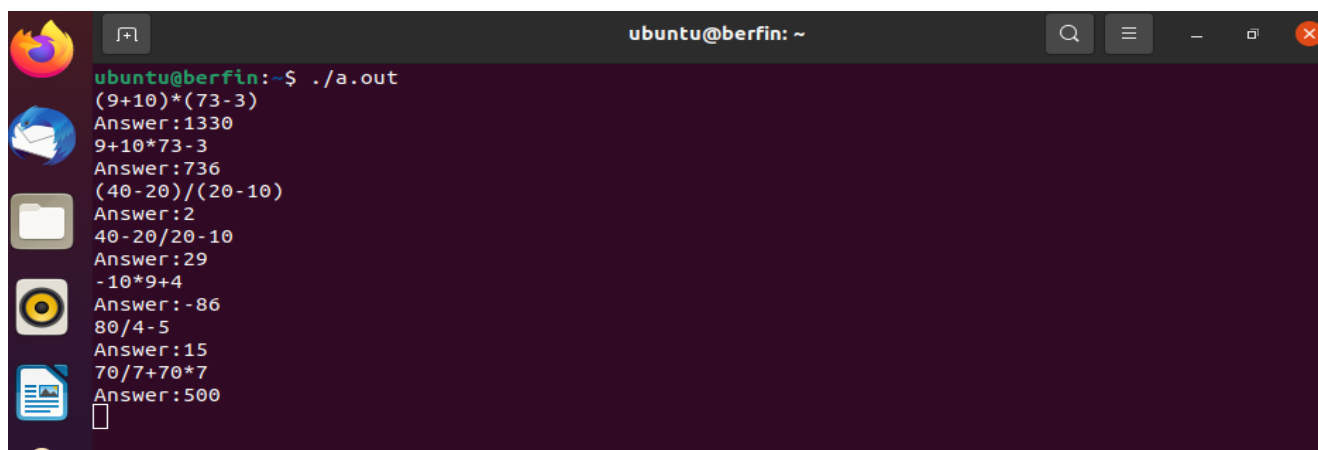
```
ubuntu@berfin: ~  
ubuntu@berfin:~$ lex snow.l  
ubuntu@berfin:~$ yacc -d snow.y  
ubuntu@berfin:~$ cc lex.yy.c y.tab.c -lfl  
ubuntu@berfin:~$ ./a.out  
true  
Result: true  
false  
Result: false  
not true  
Result: false  
true and false  
Result: false  
false or true  
Result: true  
(true and true) or false  
Result: true  
(true and false) and not true  
Result: false  
not true or not false  
Result: true
```

- 3) My program can distinguish negative and positive numbers and do addition and subtraction.



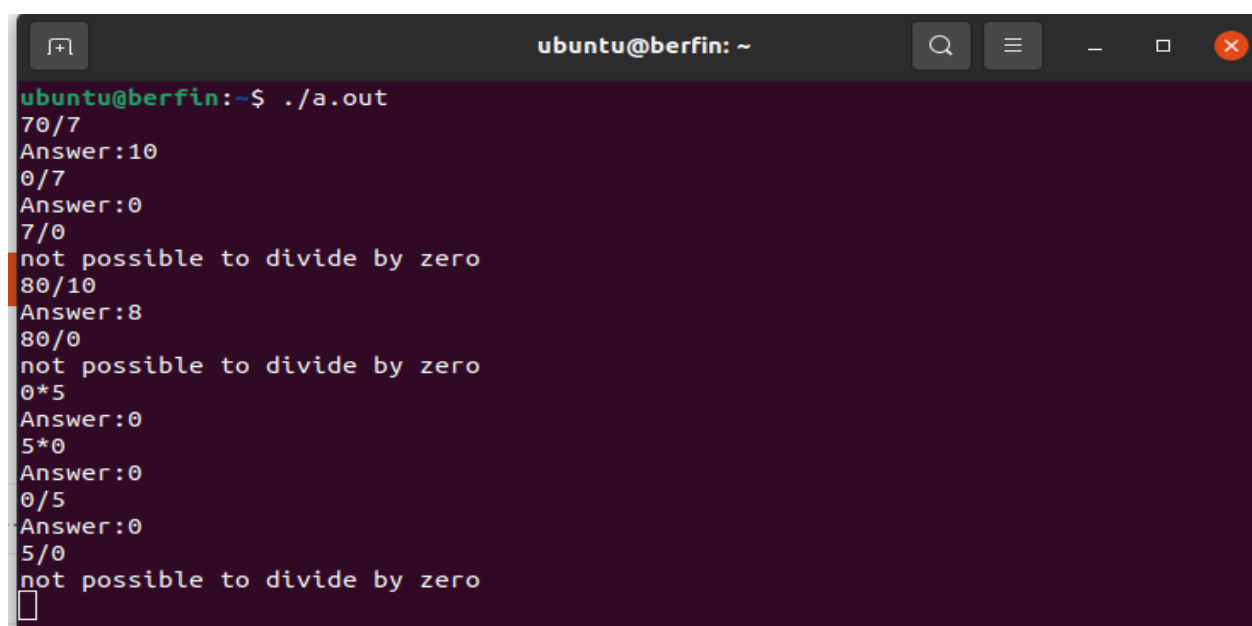
```
ubuntu@berfin: ~  
ubuntu@berfin:~$ ./a.out  
1+8  
Answer:9  
-5+7  
Answer:2  
-5-5  
Answer:-10  
10-90  
Answer:-80  
+50+41  
Answer:91  
□
```

- 4) The program pays attention to the precedence of multiplication and division operations. it also gives first priority to parentheses.



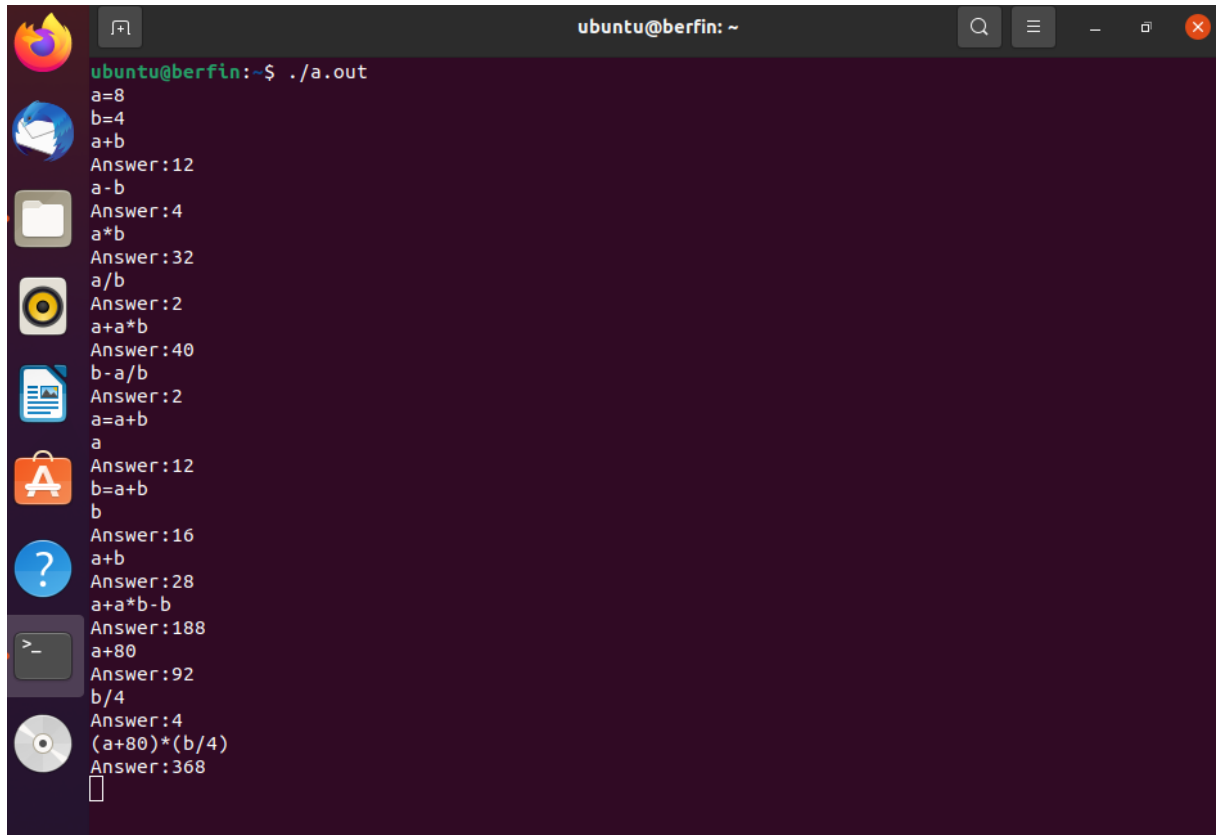
```
ubuntu@berfin: ~  
ubuntu@berfin:~$ ./a.out  
(9+10)*(73-3)  
Answer:1330  
9+10*73-3  
Answer:736  
(40-20)/(20-10)  
Answer:2  
40-20/20-10  
Answer:29  
-10*9+4  
Answer:-86  
80/4-5  
Answer:15  
70/7+70*7  
Answer:500  
□
```

- 5) Since a number cannot be divided by 0, it gives a warning when it encounters such an operation.



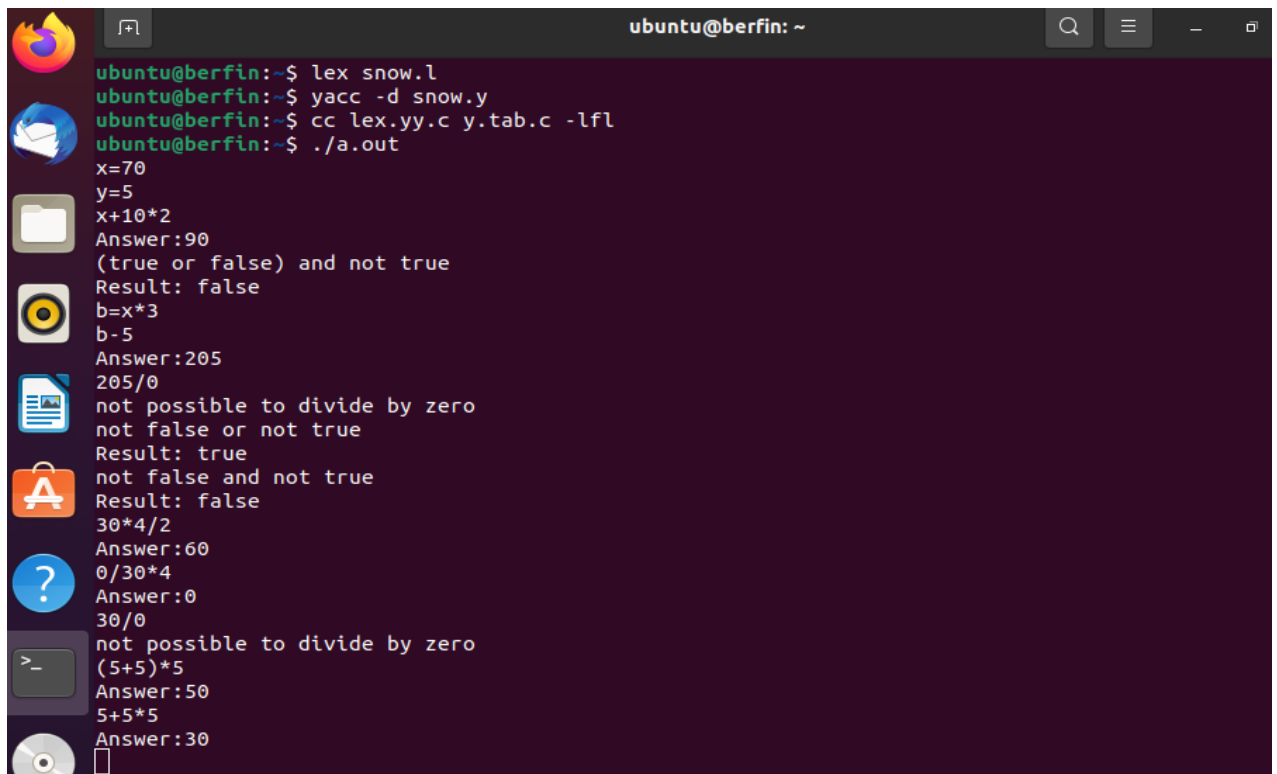
```
ubuntu@berfin: ~  
ubuntu@berfin:~$ ./a.out  
70/7  
Answer:10  
0/7  
Answer:0  
7/0  
not possible to divide by zero  
80/10  
Answer:8  
80/0  
not possible to divide by zero  
0*5  
Answer:0  
5*0  
Answer:0  
0/5  
Answer:0  
5/0  
not possible to divide by zero  
□
```

- 6) In this program, you can assign a value to a variable and use this variable in mathematical operations.



```
ubuntu@berfin: ~  
ubuntu@berfin:~$ ./a.out  
a=8  
b=4  
a+b  
Answer:12  
a-b  
Answer:4  
a*b  
Answer:32  
a/b  
Answer:2  
a+a*b  
Answer:40  
b-a/b  
Answer:2  
a=a+b  
a  
Answer:12  
b=a+b  
b  
Answer:16  
a+b  
Answer:28  
a+a*b-b  
Answer:188  
a+80  
Answer:92  
b/4  
Answer:4  
(a+80)*(b/4)  
Answer:368  
█
```

- 7) Example with all properties mixed :



```
ubuntu@berfin: ~  
ubuntu@berfin:~$ lex snow.l  
ubuntu@berfin:~$ yacc -d snow.y  
ubuntu@berfin:~$ cc lex.yy.c y.tab.c -lfl  
ubuntu@berfin:~$ ./a.out  
x=70  
y=5  
x+10*2  
Answer:90  
(true or false) and not true  
Result: false  
b=x*3  
b-5  
Answer:205  
205/0  
not possible to divide by zero  
not false or not true  
Result: true  
not false and not true  
Result: false  
30*4/2  
Answer:60  
0/30*4  
Answer:0  
30/0  
not possible to divide by zero  
(5+5)*5  
Answer:50  
5+5*5  
Answer:30  
█
```

USING OF LEX :

snow.l is an input file written in a language which describes the generation of lexical analyzer.

The lex compiler transforms **snow.l** to a C program known as **lex.yy.c**.

lex.yy.c is compiled by the C compiler to a file called **a.out**

input : stream of characters

output : stream of tokens

STRUCTURE :

```
% {    // The necessary functions and libraries are added to this section

#include <stdio.h>

#include <stdlib.h>

void yyerror(char *);

#include "y.tab.h"          // Generated by Yacc

    // yacc generates definitions for tokens and place them in file y.tab.h

% }

%%    // regular expressions - action referring code segments are added to this section

"true"  { yylval=1; return TRUE; }

"false" { yylval=0; return FALSE; }

"and"   { return AND; }

"or"    { return OR; }

"not"   { return NOT; }

[a-zA-Z][a-zA-Z0-9]* { yylval=*yytext - 'a'; return WORD; }    // VARIABLES

[0-9]+   { yylval=atoi(yytext); return NUMBER; }              // INTEGERS

[-+()=/*] { return *yytext; }                                    // OPERATORS

[ \t]    ;                                                       // IGNORING WHITESPACE

.\\n     { return yytext[0]; }

%%

int yywrap() { return 1; } // this function is called by lex when input is exhausted
```

```

1 %{
2 #include <stdio.h>
3 #include <stdlib.h>
4 void yyerror(char *);
5 #include "y.tab.h"
6 %}
7 %%
8 "true"    {yyval=1; return TRUE;}
9 "false"   {yyval=0; return FALSE;}
10 "and"     {return AND;}
11 "or"      {return OR;}
12 "not"     {return NOT;}
13 [a-zA-Z][a-zA-Z0-9]* {yyval=*yytext - 'a'; return WORD;}
14 [0-9]+    {yyval=atoi(yytext); return NUMBER;}
15 [-(+)=/*] {return *yytext;}
16
17 [ \t]     ;
18 .|\n     {return yytext[0];}
19 %%
20 int yywrap(){ return 1; }
21

```

DESCRIPTION OF THE RULES :

- *When choosing a variable name(word) , it is imperative to start with a lowercase or uppercase letter. other letters can contain numbers or upper or lower case letters.
- *If one or more digits (between 0 and 9 , also including 0 and 9) side by side, this is number.
- *These words are reserved : **not , or , and , false , true**

(It is important to put this description line first. Because the letter property rule we allow when defining variable name can cause confusion. For example, when the letters 'n' 'o' 't' are encountered next to each other, this is not a token 'WORD'. We have to return 'NOT'.) When these expressions are encountered(a match is found) , the associated action takes place to produce token ,(their meanings are returned immediately) and The token is then given to parser for further processing.used for grammar in the yacc file.)

- * ‘-’, ‘+’, ‘/’, ‘*’, ‘=’, ‘(’, ‘)’ program has these operators and it knows the priority of

parentheses. The presedence of operators :

1) ‘(’, ‘)’ 2) ‘*’ , ‘/’ 3) ‘+’, ‘-’

- *if it sees a gap (\t) she ignores it.

- *It is allowed to use as many operations , words and numbers (variables, constants ..) on the line as you want. This is unlimited .

USING AND STRUCTURE OF YACC :

/ this part contains the necessary functions and libraries and yacc definitions:*

*%start %token %left %right */*

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(char *s);

int yylex(void); *// lexical analyzer invoked by the parser*

int val[26];

int flag=0;

%}

//tokens = nonterminals

%token TRUE FALSE AND OR NOT WORD NUMBER

/ % left and % right are used to indicate the precedence levels association of these operators */*

%left AND

%left OR

%right NOT

%right '='

%left '+' '-'

%left '*' '/'

%left '(' ')'

%right UMINUS

%%

```
line :line expr '\n' { printf("Result: %s\n",$2?"true":"false");}
```

```
  |line stmtnt '\n'
```

```
  | '\n'
```

```
  |
```

```
  ;
```

```
stmtnt: phrase      {if(flag==0)printf("Answer:%d\n",$1);
```

```
                else {flag=0;}}
```

```
  | WORD '=' phrase { val[$1]=$3;}
```

```
  ;
```

```
phrase: NUMBER      // $2=attribute of token (stored in yylval)
```

```
  | '-'phrase      { $$=-$2;}
```

```
  | '+'phrase      { $$= $2;}
```

```
  | WORD           { $$=val[$1];
```

```
                if($$==0){flag=1;printf("error\n");}
```

```
  }
```

```
  | phrase '+' phrase { $$ = $1 + $3;flag=0; }
```

```
// $$ =attribute (parent)    // $1=attribute (child)
```

```
  | phrase '-' phrase { $$ = $1 - $3;flag=0; }
```

```
  | phrase '*' phrase { $$ = $1 * $3;flag=0; }
```

```
  | phrase '/' phrase { if($3==0){
```

```
                printf("not possible to divide by zero\n");
```

```
                flag=1;}
```

```
                else{ $$=$1/$3;flag=0;}
```

```
  }
```

```
  | '('phrase')'    { $$=$2;flag=0;}
```

```
  ;
```

```

expr : TRUE      {$$=$1;}
      | FALSE     {$$=$1;}
      | expr AND expr {$$=($1 & $3);}
      | expr OR expr  {$$=($1 | $3);}
      | NOT expr     {$$= !($2);}
      | ('expr')     {$$=$2;}
      ;

```

```
%%
```

```

void yyerror(char *s){ // Invoked by parser to report parse errors
    fprintf(stderr,"%s\n",s); }

```

```

int main(){
    yyparse(); // to run the parser
    return 0;
}

```

/ &1 ,&2,...,&n can be refer to the values associated with symbols
&& refer to the value of the left */*


```

snowy Kaydet
1 %{
2 #include <stdio.h>
3 #include <stdlib.h>
4 void yyerror(char *s);
5 int yylex(void);
6 int val[26];
7 int flag=0;
8 %}
9 %token TRUE FALSE AND OR NOT WORD NUMBER
10 %left AND
11 %left OR
12 %right NOT
13 %right '='
14 %left '+' '-'
15 %left '*' '/'
16 %left '(' ')'
17 %right UMINUS
18 %%
19 line :line expr '\n' { printf("Result: %s\n", $2?"true":"false"); }
20     |line stmt '\n'
21     | '\n'
22     ;
23
24 stmt: phrase {if(flag==0)printf("Answer:%d\n", $1);
25             else {flag=0;}}
26     | WORD '=' phrase {val[$1]=$3;}
27     ;
28 phrase: NUMBER
29        | '-' phrase  {$$=-$2;}
30        | '+' phrase  {$$= $2;}
31        | WORD        {$$=val[$1];
32                     if($$==0){flag=1;printf("error\n");}
33                     }
34        | phrase '+' phrase  {$$ = $1 + $3;flag=0; }
35        | phrase '-' phrase  {$$ = $1 - $3;flag=0; }
36        | phrase '*' phrase  {$$ = $1 * $3;flag=0; }
37        | phrase '/' phrase  {if($3==0){
38                             printf("not possible to divide by zero\n");
39                             flag=1;}
40                             else{$$=$1/$3;flag=0;}
41                             }
42        | '(' phrase ')'      {$$=$2;flag=0;}
43        ;

```

```

44
45 expr : TRUE          {$$=$1;}
46      | FALSE         {$$=$1;}
47      | expr AND expr  {$$=($1 & $3);}
48      | expr OR expr   {$$=($1 | $3);}
49      | NOT expr       {$$= !($2);}
50      | '(' expr ')'    {$$=$2;}
51      ;
52
53 %%
54 void yyerror(char *s){
55     fprintf(stderr, "%s\n", s); }
56 int main(){
57     yyparse();
58     return 0;
59 }
60

```

GRAMMAR :

line :

```
line expr '\n' { printf("Result: %s\n",$2?"true":"false");}  
| line stmnt '\n'  
| '\n'  
|  
;
```

(That is a line consists of zero or more expressions , or zero or more statements with a newline. If the line consists of zero or more expressions and newline is detected , the result of the expression (true or false) is printed on the screen.)

stmnt:

```
phrase      {if(flag==0)printf("Answer:%d\n",$1);  
              else {flag=0;}}  
| WORD '=' phrase {val[$1]=$3;}  
;
```

(That is a statement consists of phrase or assignment act where WORD is a token. We use flag to check status. Because of a number is not divided by 0 If we encounter such a situation, the value of the flag will change and we will not print a value on the screen. Otherwise, the answer of the statement (numerical value) is printed on the screen.)

phrase:

```
NUMBER  
| '-'phrase  {$$=- $2; }  
| '+'phrase  {$$= $2; }  
| WORD      {$$=val[$1];  
              if($$==0){flag=1;printf("error\n");}  
              }  
| phrase '+' phrase  {$$ = $1 + $3;flag=0; }  
| phrase '-' phrase  {$$ = $1 - $3;flag=0; }
```

```

| phrase '*' phrase  {$$ = $1 * $3;flag=0; }
| phrase '/' phrase  {if($3==0){
                        printf("not possible to divide by zero\n");
                        flag=1;}
                        else{$$=$1/$3;flag=0;}
                        }
| '('phrase')'    {$$=$2;flag=0;}
;

```

(When the divisor is 0 or when we encounter a letter that does not have a value, we print an error text on the screen and we set the value of flag to 1 so that no result is written on the screen. In this grammar we allow the phrase to be negative or positive and the use of parentheses.)

expr :

```

TRUE      {$$=$1;}
| FALSE    {$$=$1;}
| expr AND expr {$$=($1 & $3);}
| expr OR expr {$$=($1 | $3);}
| NOT expr   {$$= !($2);}
| '('expr')' {$$=$2;}
;

```

(it means that a expression can be true or false and that two or more expressions can be connected with "and" and "or" . Also parenthesis precedence has been added to the features.)