**COMMAND LINE CALCULATOR**

by

D. Shashank - 13BCE1036

A. Kishor Babu - 13BCE1026

*A project report submitted to the*

**SCHOOL OF COMPUTING SCIENCES ENGINEERING**



**VIT UNIVERSITY, CHENNAI**

**Vandalur – Kelambakkam Road**

**Chennai – 600 127.**

**MAY 2016**

**ACKNOWLEDGEMENT**

**Course Code: CSE309**

This project report entitled “**COMMAND LINE CALCULATOR”** is a bonafide work of **SHASHANK. D (13BCE1036) and A. KISHOR BABU (13BCE1026)** carried out the Project work under the supervision and guidance of **Prof. ANASOOYA. G.** We wish to express our sincere thanks and deep sense of gratitude to our project guide, **Prof. ANASOOYA. G,** Associate Professor, School of Computing Sciences and Engineering, for her consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work

**TABLE OF CONTENTS**

1. Abstract
2. What will my code do?
3. Parsing
4. Code
5. Output Screenshot

**ABSTRACT**

It takes precious deci-seconds away from software development time to remove anyone’s hands from the keyboard and use the mouse for a calculator and moreover an on-screen calculator is not as efficient as a user-friendly calculator.

That's why we use a simple command line calculator (and it is also an example of recursive decent parsing). We use lexical analysis to get a stream of tokens from the input expression which on suitable recursive parsing gives us the desired result.

**WHAT WILL MY CODE DO?**

Whenever an expression is given, the expression will be considered as an argument passed and will be stored in an array. The array is passed to a lex function which outputs a set of tokens separating each and every character. Whenever it encounters an operator, the value preceding the operator and succeeding the operator will be pointed to separate buffers.

Whenever the parser encounters a left parenthesis, it checks for the right parenthesis and increases the priority to solve the inside parenthesis part first. As far as other operators are concerned, left to right, priority of multiply and divide rule over add and subtract whereas among the two, it is first come first serve.

The code will handle exceptions of Divide by zero, missing parenthesis etc.

Hence command line calculator is a perfect application of lexical analysis and syntax analysis.

**PARSING**

A perfect grammar must be used for performing calculations and parsing the right expressions and values.

The following grammar can do the job for the calculator.

Exp = Exp+Exp

Exp = Exp/Exp

Exp = Exp\*Exp

Exp = Exp-Exp

Exp = (Exp)

Exp = 0xExp

Exp = id

**CODE**

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <math.h>

#include <setjmp.h>

#include <string.h>

#define VAR\_NAME\_SIZE 31

typedef struct \_MapEntry\_t {

char name[VAR\_NAME\_SIZE+1];

double value;

struct \_MapEntry\_t\* next;

} MapEntry\_t;

MapEntry\_t\* varmap;

void map\_init(void)

{

varmap = 0;

}

void map\_clear(void)

{

MapEntry\_t\* cur = varmap;

while( cur ) {

MapEntry\_t\* next = cur->next;

free( cur );

cur = next;

}

varmap = 0;

}

MapEntry\_t\* map\_find( const char\* var )

{

MapEntry\_t\* cur = varmap;

while( cur ) {

if ( strcmp( var, cur->name ) == 0 ) {

return cur;

}

cur = cur->next;

}

return 0;

}

void map\_add( const char\* var, double value )

{

MapEntry\_t\* entry = map\_find( var );

if ( entry == 0 ) {

entry = (MapEntry\_t\*)malloc( sizeof(MapEntry\_t) );

strncpy( entry->name, var, VAR\_NAME\_SIZE + 1 );

entry->name[VAR\_NAME\_SIZE] = 0;

entry->next = varmap;

varmap = entry;

}

entry->value = value;

}

int map\_lookup( const char\* var, double\* value )

{

MapEntry\_t\* entry = map\_find( var );

if ( entry ) {

\*value = entry->value;

return 1;

}

return 0;

}

#define TYPE\_CHAR 0

#define TYPE\_FLOAT 1

#define TYPE\_EOF 2

#define TYPE\_ERROR 3

#define TYPE\_VARIABLE 4

typedef struct \_val\_t {

int type;

union {

double fval;

char cval;

char variable[255];

} d;

} val\_t;

void print\_val( val\_t\* val )

{

if ( val->type == TYPE\_FLOAT ) {

printf("%lf\n", val->d.fval );

} else if ( val->type == TYPE\_CHAR ) {

printf("\'%c\'\n", val->d.cval);

} else if ( val->type == TYPE\_VARIABLE ) {

printf("Variable \'%s\'\n", val->d.variable);

} else if ( val->type == TYPE\_EOF ) {

printf("EOF\n");

} else if ( val->type == TYPE\_ERROR ) {

printf("ERROR\n");

} else {

printf("Bad val type: %d\n", val->type);

}

}

int argc;

char\*\* argv;

static char buffer[1024];

val\_t next\_val;

int arg = 0;

int argp = 0;

int bpos = 0;

static int have\_next\_val = 0;

jmp\_buf env;

void reset(int pargc, char\*\* pargv)

{

argc = pargc;

argv = pargv;

buffer[0] = 0;

arg = 0;

argp = 0;

bpos = 0;

have\_next\_val = 0;

}

void lex(val\_t\* val, int next)

{

char token[25];

int tpos = 0;

int done = 0;

int number = 0;

enum {

read\_start,

read\_int,

read\_mantissa,

read\_hex,

read\_var

} state = read\_start;

if ( next ) {

have\_next\_val = 0;

return;

} else if ( have\_next\_val ) {

\*val = next\_val;

return;

}

while( !done ) {

char ch;

if ( arg == argc ) {

val->type = TYPE\_EOF;

val->d.fval = 0;

break;

}

ch = argv[arg][argp];

switch ( state ) {

case read\_start:

if ( ch >= '0' && ch <= '9' ) {

state = read\_int;

tpos = 0;

token[tpos++] = ch;

} else if ( ch == '+' || ch == '-' ||

ch == '/' || ch == '\*' ||

ch == '(' || ch == ')' ||

ch == '%' || ch == '^' ||

ch == '=' )

{

val->type = TYPE\_CHAR;

val->d.cval = ch;

done = 1;

} else if ( ch == ' ' || ch == '\t' || ch == 0 ) {

} else if ( ch == '.' ) {

tpos = 0;

token[tpos++] = '0';

token[tpos++] = '.';

state = read\_mantissa;

} else if ( isalpha( ch ) ) {

state = read\_var;

tpos = 0;

token[tpos++] = ch;

} else {

buffer[bpos] = 0;

printf("Parse error after: %s\n", buffer);

longjmp( env, 1 );

}

break;

case read\_int:

if ( ch >= '0' && ch <= '9' ) {

if ( tpos < sizeof(token) ) {

token[tpos++] = ch;

} else {

token[tpos] = 0;

printf("Number too long: %s\n", token);

}

} else if ( ch == 'x' && tpos == 1 ) {

state = read\_hex;

} else if ( ch == '.' ) {

if ( tpos < sizeof(token) ) {

token[ tpos++ ] = ch;

} else {

token[tpos] = 0;

printf("Number too long: %s\n", token);

}

state = read\_mantissa;

} else {

token[tpos] = 0;

state = read\_start;

val->type = TYPE\_FLOAT;

val->d.fval = (double)atoi(token);

done = 1;

goto done;

}

break;

case read\_mantissa:

if ( ch >= '0' && ch <= '9' ) {

if ( tpos < sizeof(token) ) {

token[tpos++] = ch;

} else {

token[tpos] = 0;

printf("Number too long: %s\n", token);

longjmp( env, 1 );

}

} else {

token[tpos] = 0;

state = read\_start;

val->type = TYPE\_FLOAT;

sscanf( token, "%lf", &val->d.fval );

done = 1;

goto done;

}

break;

case read\_hex:

ch = tolower( ch );

if ( ch >= '0' && ch <= '9' ) {

number <<= 4;

number += ch - '0';

} else if ( ch >= 'a' && ch <= 'f' ) {

number <<= 4;

number += 10 + ch - 'a';

} else {

token[tpos] = 0;

state = read\_start;

val->type = TYPE\_FLOAT;

val->d.fval = number;

done = 1;

goto done;

}

break;

case read\_var:

if ( ch >= 'a' && ch <= 'z' ||

ch >= 'A' && ch <= 'Z' ||

ch >= '0' && ch <= '9' ||

ch == '\_' )

{

if ( tpos < sizeof(token) ) {

token[tpos++] = ch;

} else {

token[tpos] = 0;

printf("Variable too long: %s", token);

longjmp( env, 1 );

}

} else {

token[tpos] = 0;

state = read\_start;

val->type = TYPE\_VARIABLE;

strcpy( val->d.variable, token);

done = 1;

goto done;

}

}

if ( ch == 0 ) {

argp = 0;

arg++;

} else {

argp++;

buffer[bpos++] = ch;

}

}

done:

next\_val = \*val;

have\_next\_val = 1;

return;

}

int match\_char( char ch )

{

val\_t val;

lex(&val, 0);

if ( val.type == TYPE\_CHAR && val.d.cval == ch ) {

lex( &val, 1 );

return 1;

}

return 0;

}

int match\_eof()

{

val\_t val;

lex(&val, 0);

if ( val.type == TYPE\_EOF ) {

return 1;

}

return 0;

}

int match\_num( val\_t\* val )

{

lex( val, 0 );

if ( val->type == TYPE\_FLOAT ) {

lex( val, 1 );

return 1;

}

return 0;

}

int match\_variable( val\_t\* val )

{

lex( val, 0 );

if ( val->type == TYPE\_VARIABLE ) {

lex( val, 1 );

return 1;

}

return 0;

}

void resolve\_variable( val\_t\* val )

{

double fval;

if ( val->type != TYPE\_VARIABLE ) {

printf("Error: value is not a variable.\n");

longjmp( env, 1 );

}

if ( !map\_lookup( val->d.variable, &fval ) ) {

printf("%s not defined.\n", val->d.variable);

longjmp( env, 1 );

}

val->type = TYPE\_FLOAT;

val->d.fval = fval;

}

void parse\_term(val\_t\* val);

void parse\_expr(val\_t\* val);

void parse\_factor( val\_t\* val );

void parse\_num\_op( val\_t\* val );

void parse\_factor( val\_t\* val );

void parse\_rest\_num\_op( val\_t\* val );

void parse\_rest\_var( val\_t\* val );

#ifndef DEBUG\_PRINT

#define dprintf(A) printf(A)

#endif

int level = 0;

void printtab() {

int i = 0;

for( i = 0; i < level; i++ ) {

dprintf(" ");

}

}

void parse\_rest\_term( val\_t\* val )

{

printtab();

level++;

if ( match\_char( '\*' ) ) {

val\_t val2;

parse\_factor( &val2 );

val->d.fval \*= val2.d.fval;

parse\_rest\_term( val );

} else if ( match\_char( '/' ) ) {

val\_t val2;

parse\_factor( &val2 );

if ( val2.d.fval != 0 ) {

val->d.fval /= val2.d.fval;

} else {

printf("Division by 0\n");

longjmp(env, 0);

}

parse\_rest\_term( val );

} else if ( match\_char( '%' ) ) {

val\_t val2;

parse\_factor( &val2 );

if ( val2.d.fval != 0 ) {

val->d.fval = fmod( val->d.fval, val2.d.fval );

} else {

printf("Division by 0\n");

longjmp(env, 0);

}

parse\_rest\_term( val );

} else if ( match\_eof() ) {

} else {

}

level--;

return;

}

void parse\_term( val\_t\* val )

{

printtab();

level++;

parse\_factor( val );

parse\_rest\_term( val );

level--;

return;

}

void parse\_rest\_num\_op( val\_t\* val )

{

if ( match\_char( '^' ) ) {

val\_t val2;

parse\_num\_op( &val2 );

val->d.fval = pow( val->d.fval, val2.d.fval );

parse\_rest\_num\_op( val );

}

return;

}

void parse\_num\_op( val\_t\* val )

{

printtab();

level++;

if ( match\_num( val ) ) {

parse\_rest\_num\_op( val );

} else if ( match\_variable( val ) ) {

resolve\_variable( val );

parse\_rest\_num\_op( val );

} else if ( match\_char( '(' ) ) {

parse\_expr( val );

if ( !match\_char( ')' ) ) {

buffer[bpos] = 0;

printf("Missing bracket: %s\n", buffer);

longjmp( env, 1 );

}

parse\_rest\_num\_op( val );

} else {

buffer[bpos] = 0;

printf("Parse error: %s\n", buffer);

longjmp( env, 1 );

}

level--;

return;

}

void parse\_factor( val\_t\* val )

{

printtab();

level++;

if ( match\_char( '-' ) ) {

parse\_factor( val );

val->d.fval = -val->d.fval;

} else {

parse\_num\_op( val );

}

level--;

return;

}

void parse\_rest\_expr( val\_t\* val )

{

printtab();

level++;

if ( match\_char( '+' ) ) {

val\_t val2;

parse\_term( &val2 );

val->d.fval += val2.d.fval;

parse\_rest\_expr( val );

} else if ( match\_char( '-' ) ) {

val\_t val2;

parse\_term( &val2 );

val->d.fval -= val2.d.fval;

parse\_rest\_expr( val );

} else if ( match\_eof() ) {

} else {

}

level--;

return;

}

void

parse\_expr(val\_t\* val)

{

printtab();

level++;

if ( match\_variable( val ) ) {

parse\_rest\_var( val );

} else {

parse\_term( val );

parse\_rest\_expr( val );

}

level--;

return;

}

void parse\_rest\_var( val\_t\* val )

{

if ( match\_char( '=' ) ) {

val\_t vexp;

parse\_expr( &vexp );

if ( vexp.type != TYPE\_FLOAT ) {

printf("Error: Tried to assign non-number to %s.\n", val->d.variable );

longjmp( env, 1 );

}

printf("Assigned to %s: ", val->d.variable );

map\_add( val->d.variable, vexp.d.fval );

\*val = vexp;

} else {

parse\_rest\_num\_op( val );

}

}

int parse( val\_t\* val )

{

if ( setjmp( env ) ) {

return 0;

}

parse\_expr( val );

if ( !match\_eof() ) {

printf("Trailing characters.\n");

longjmp( env, 1 );

}

return 1;

}

void usage(void)

{

printf("Usage: calc [mathematical expression]\n");

exit(-1);

}

int main( int pargc, char\* pargv[] )

{

val\_t val;

map\_init();

if ( pargc == 1) {

char cmd[100];

char\* cmds = cmd;

int cmdlen = 0;

cmd[0] = 0;

printf("Use Control-C to quit.\n");

for( ;; ) {

top:

printf( "\r> %s", cmd );

cmdlen = strlen(cmd);

for( ;; ) {

char c = \_getch();

if ( c == '\b' ) {

if ( cmdlen > 0 ) {

cmd[--cmdlen] = 0;

printf( "\r> %s \b", cmd );

}

} else if ( c == '\r' ) {

putc('\n', stdout);

break;

} else if ( c == 3 ) {

printf("QUIT\n");

exit(0);

} else if ( cmdlen < sizeof(cmd)-1 ) {

putc(c, stdout);

cmd[cmdlen++] = c;

cmd[cmdlen] = 0;

}

}

reset( 1, &cmds );

if ( parse( &val ) ) {

print\_val( &val );

} else {

printf("Error.\n");

}

}

}

reset( pargc - 1, pargv + 1 );

parse\_expr( &val );

print\_val( &val );

map\_clear();

return 0;

}

**Output Screenshots:**

