

XCS607 - COMPILER DESIGN LABORATORY

MINI PROJECT REPORT

ON

**COMMAND LINE CALCULATOR**

SUBMITTED BY

C.NATARAJ – 121012012756

M.SANTHOSH – 121012012762

M.MOHAMED YASAR – 121012012752

DEPARTMENT – B.TECH .CSE ( III YEAR )

# TABLE OF CONTENTS

|  |  |
| --- | --- |
| **S.No** | **Title** |
| **1.** | **Abstract** |
| **2.** | **Requirements** |
| **3.** | **Source Code** |
| **4..** | **Output** |
| **5.** | **Result** |
| **6..** | **References** |

## Aim:

The command line calculator is to be capable of parsing a human-readable mathematical expression with units, return the value if it can be evaluated and inform the user about the position of an error if not.

## Abstract:

A command line calculator which supports mathematical expressions with scientific functions is very useful for most developers. The calculator available with Windows does not support most scientific functions. Most of the time, I do not feel comfortable with the calculator available with Windows. I needed a calculator which will not restrict writing expressions. I use variables to store results. Every time I need a simple calculation, I have to face problems with the Windows calculator. To make such a calculator, I designed a complete Mathematics library with MFC. The most difficult part I found when designing such a calculator was the parsing logic. Later while working with .NET, the runtime source code compilation made the parsing logic easy and interesting. I read some articles on .NET CodeDOM compilation. And I decided to write a new command line calculator using CodeDOM. It uses runtime compilation and saves the variables by serializing in a file. Thus you can get the values of all the variables used in the previous calculation.

**Requirements to run the script: Software Requirements:** Windows/Ubuntu Operating System

C compiler (gcc, cc, egcs,..) **Hardware Requirements:**

CPU : Intel Core i5 Memory : 8GB for RAM

## Code:

#include <stdio.h> #include <stdlib.h> #include <assert.h> #include <math.h> #include <setjmp.h>

/\*Keep variables in a map. \*/

#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;

*}*

/\* General purpose structure used to represent things returned by the lexer and values as they are calculated up the parse tree\*/

#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;

/\*Print out a value\*/ 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);

}

}

/\*State variables for the lexer\*/

/\* number of command line arguments \*/ int argc;

/\* command line arguments array \*/ char\*\* argv;

/\* array parsed so far. Used for debugging and printing out error messages. \*/ static char buffer[1024];

/\* the token that was most recently scanned by the lexer \*/ val\_t next\_val;

/\* which argument we are currently scanning \*/ int arg = 0;

/\* the index into argv[arg] that we are currently scanning \*/ int argp = 0;

/\* the postion in buffer[] that we are storing characters. \*/ 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;

}

/\*Scanner. Scans tokens from the command line arguments.\*/ 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 ) {

/\* get the next character. Add to buffer. Do not increment the next \*/

/\* character to read. \*/ char ch;

if ( arg == argc ) {

val->type = TYPE\_EOF; val->d.fval = 0;

break;

}

ch = argv[arg][argp];

/\*printf("argv[%d][%d] = %c (state=%d)\n", \*/

/\* arg, argp, argv[arg][argp], state); \*/ 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;

}

}

/\* increment the character we are going to read. \*/ if ( ch == 0 ) {

argp = 0; arg++;

} else {

argp++; buffer[bpos++] = ch;

}

}

done:

next\_val = \*val; have\_next\_val = 1;

/\*printf("lex(): "); \*/

/\*print\_val( val ); \*/ return;

}

/\*If the next token is CH, then consume it and return 1. Otherwise, do not consume it and return 0.\*/

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;

}

/\*Return 1 if the next token is the end of file marker.\*/ int

match\_eof()

{

val\_t val; lex(&val, 0);

if ( val.type == TYPE\_EOF ) { return 1;

}

return 0;

}

/\*If the next token is a number, then consume it and return 1. Otherwise,do not consume it and 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 );

//#define DEBUG\_PRINT 1 #ifndef DEBUG\_PRINT #define dprintf(A) printf(A) #endif

int level = 0; void printtab() {

int i = 0;

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

}

}

/\* rest\_term := \* factor rest\_term

/ factor rest\_term

% factor rest\_term

<nil>\*/

void

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

{

printtab(); dprintf("parse\_rest\_term()\n"); 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;

}

/\* term := factor rest\_term\*/ void

parse\_term( val\_t\* val )

{

printtab(); dprintf("parse\_term()\n"); level++;

parse\_factor( val ); parse\_rest\_term( val ); level--;

return;

}

/\* rest\_num\_op := ^ num\_op rest\_num\_op <nil>

\*/ 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;

}

/\* num\_op := num rest\_num\_op ( expr ) rest\_num\_op \*/ void

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

{

printtab(); dprintf("parse\_num\_op()\n"); 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;

}

/\*factor := - factor

num\_op \*/

void

parse\_factor( val\_t\* val )

{

printtab(); dprintf("parse\_factor()\n"); level++;

if ( match\_char( '-' ) ) { parse\_factor( val );

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

} else {

parse\_num\_op( val );

}

level--; return;

}

/\* rest\_expr := + term rest\_expr

- term rest\_expr (nil) \*/

void

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

{

printtab(); dprintf("parse\_rest\_expr()\n"); 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;

}

/\* expr := term rest\_expr \*/ void

parse\_expr(val\_t\* val)

{

printtab(); dprintf("parse\_expr()\n");

level++;

if ( match\_variable( val ) ) { parse\_rest\_var( val );

} else {

parse\_term( val ); parse\_rest\_expr( val );

}

level--; return;

}

/\* rest\_var := '=' expr

rest\_num\_op \*/

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;

}

/\* Print usage information \*/ void

usage(void)

{

printf("Usage: calc [mathematical expression]\n"); exit(-1);

}

/\* main \*/ 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:

// print command line. 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);

//printf("%d\n", c); cmd[cmdlen++] = c; cmd[cmdlen] = 0;

}

}

reset( 1, &cmds );

/\* parse the expression. \*/ if ( parse( &val ) ) {

/\* print the value. \*/ print\_val( &val );

} else {

printf("Error.\n");

}

}

}

reset( pargc - 1, pargv + 1 );

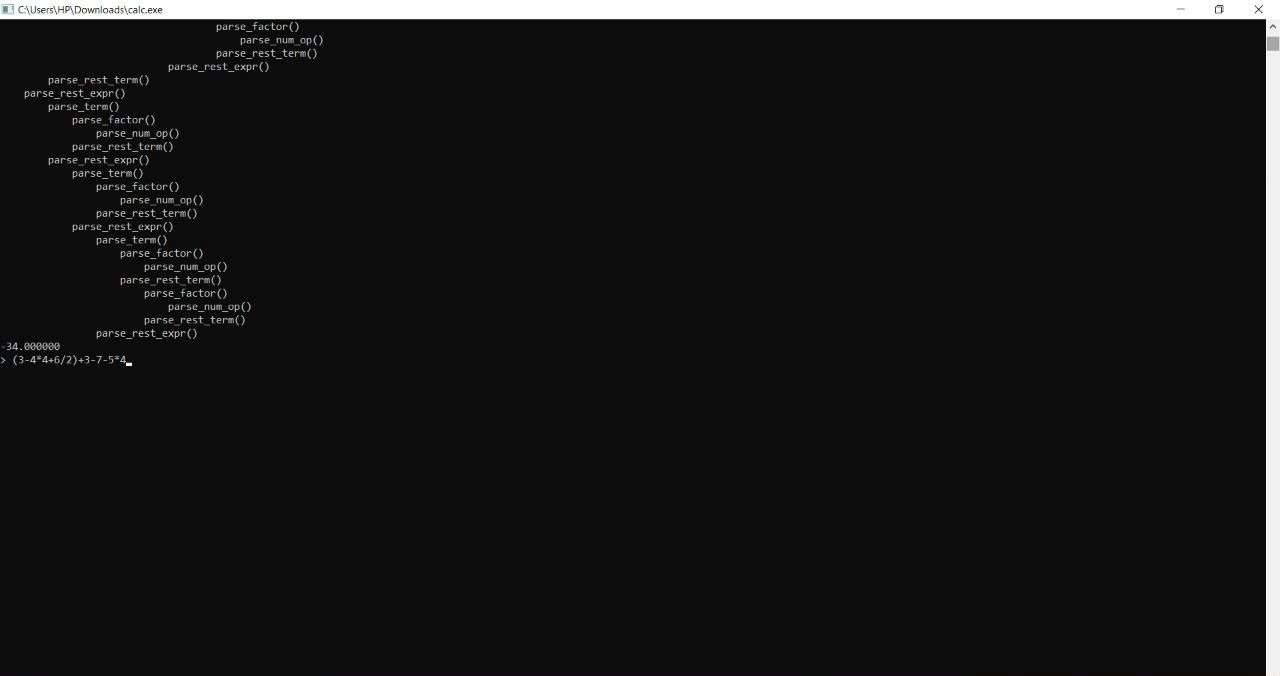
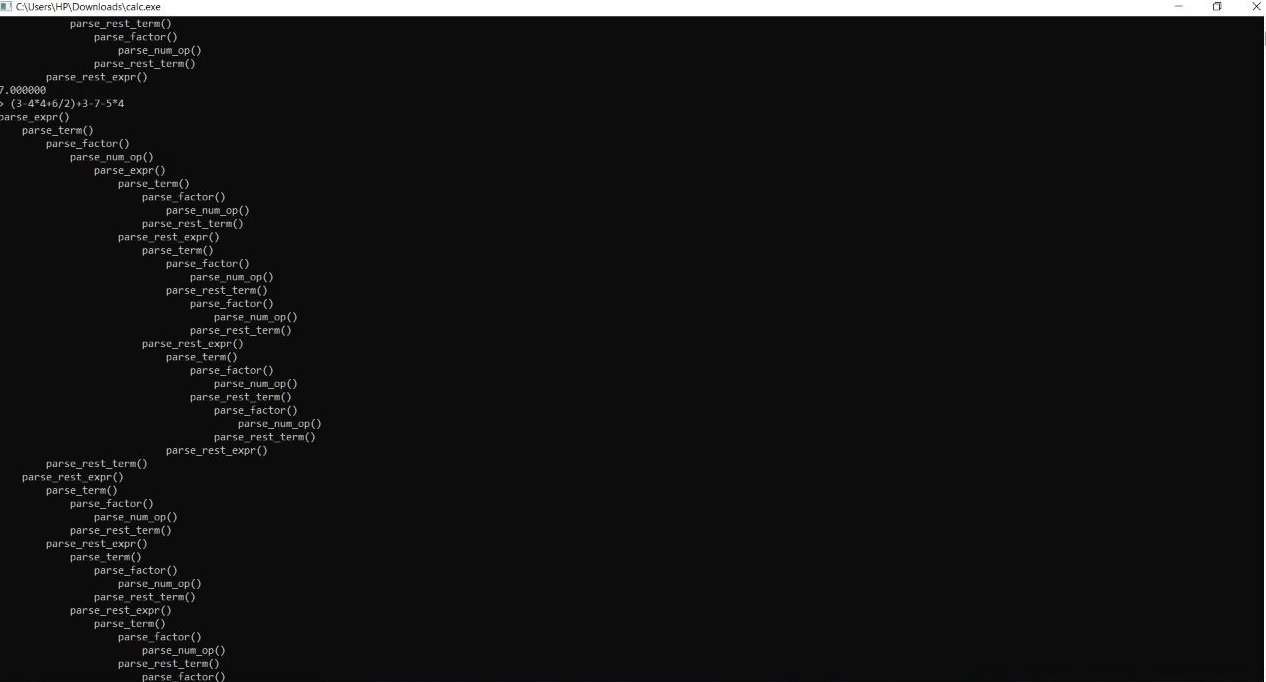
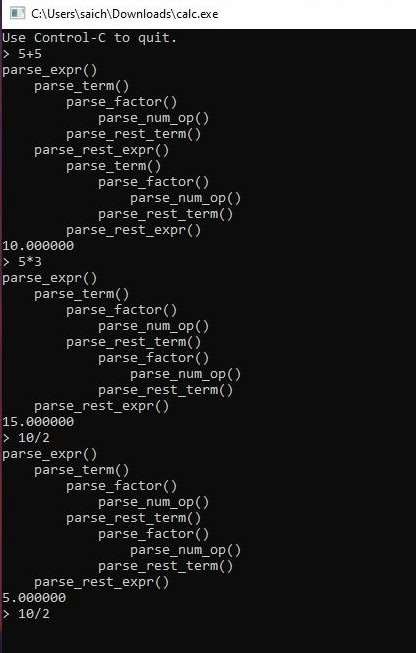
/\* parse the expression. \*/ parse\_expr( &val );

/\* print the value. \*/ print\_val( &val ); map\_clear();

return 0;

}

## Output:



## Result:

* The number of lines of the commands present is clearly available.
* The value is accurate and precise in comparison to manual calculation.
* This is a powerful and versatile command-line calculator that really lives up to your expectation.
* Preloaded on all modern Linux distributions, this can make your number crunching tasks much easier to handle without leaving your terminals.
* Besides, if your shell script requires floating point calculation, can easily be invoked by the script to get the job done.

## References :

* + https://[www.guru99.com/compiler-design-phases-of-compiler.html](http://www.guru99.com/compiler-design-phases-of-compiler.html)
  + https://unix.stackexchange.com /simple- command-line-calculator
  + <https://clcalc.net/>