SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

COMMAND LINE CALCULATOR

Gudied by:

Vidhya J V

Submitted By:

Varsha S(RA2011026010286)

Urvi Bhanu Hirani(RA2011026010293)

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this mini project report titled COMMAND LINE CALCULATOR is the bonafide work done by Varsha S(RA2011026010286) and Urvi Bhanu Hirani (RA2011026010293) who carried out the mini project work and Laboratory exercises under my supervision for 18CSC304J – COMPILER DESIGN. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

Vidhya J V
Assistant Professor
Department of Computing Intelligence

Dr. R.Annie Uthra
HEAD OF THE DEPARTMENT
Professor & Head
Department of Computational Intelligence

Aim :

To create a command line calculator in C program.

Abstract:

Command-Line Calculator provides the most fluid interface, especially if you are performing chained calculations i.e., multiple calculations that rely on the results of previous calculations. This will allow us to perform all kinds of operations such as scientific, financial, or even simple calculation. Also, it can be used in shell scripts for complex math. Basic calculator (bc command) is used for command line calculator. It is similar to basic calculator by using which we can do basic mathematical calculations. Arithmetic operations are the most basic in any kind of programming language. This provides the solution as, The calculate function calculates an expression. It uses the saved variables. I have generated code which has a declaration of the variables

●To Evaluate the given expressions.

●To perform basic calculations

Introduction:

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 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. 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.

Every modern Linux desktop distribution comes with a default GUI-based calculator app. On the other hand, if your workspace is full of terminal windows, and you would rather crunch some numbers within one of those terminals quickly, you are probably looking for a command-line calculator. In this category, GNU bc (short for "basic calculator") is a hard to beat one. While there are many command-line calculators available on Linux, I think GNU bc is hands-down the most powerful and useful.

In this command line calculator, the result is saved in a pre-defined variable called ans. The user can declare his/her own variables to store results and can use it later in different expressions. The validation of the variable name is the same as in C#. Similarly, expression support is the same as supported in C# .NET.

The Calculate function calculates an expression. It uses the saved variables. I have generated code which has declaration of the variables.

Objectives:

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.

Requirements:

software requirements: Windows/Ubuntu Operating System C programming Language

hardware requirements: Minimum of 4GB 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;
```

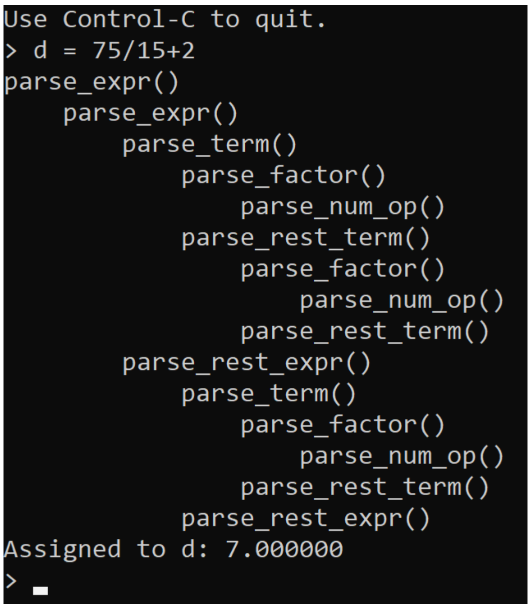
```
}

/*****************************************************************************
   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;
}
```

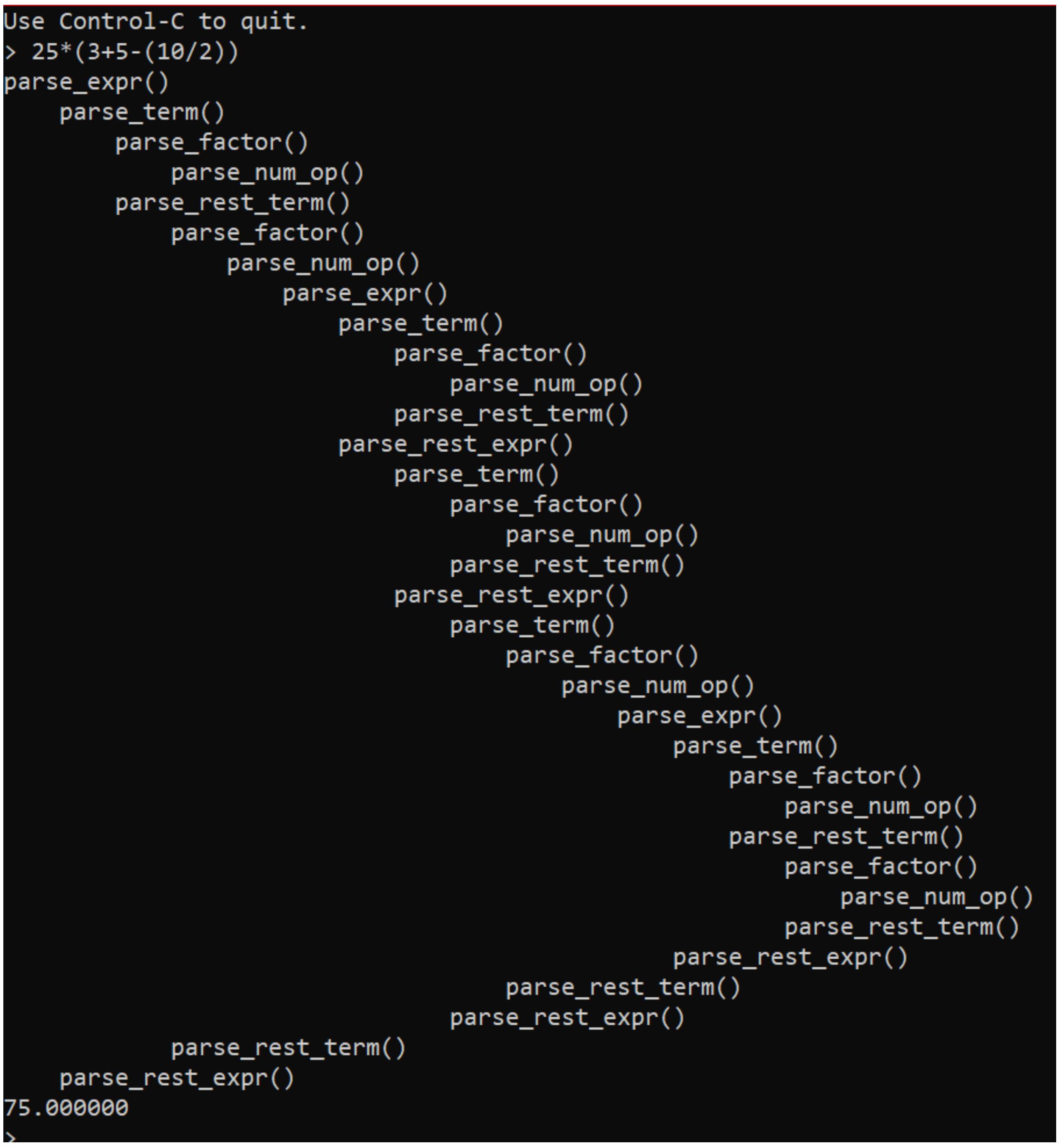
Sample Outputs:

Input: d = 75/15+2



```
Use Control-C to quit.
> d = 75/15+2
parse_expr()
    parse_expr()
        parse_term()
            parse_factor()
                parse_num_op()
            parse_rest_term()
                parse_factor()
                    parse_num_op()
                parse_rest_term()
        parse_rest_expr()
            parse_term()
                parse_factor()
                    parse_num_op()
                parse_rest_term()
            parse_rest_expr()
Assigned to d: 7.000000
> 
```

Input: 25*(3+5-(10/2))



```
Use Control-C to quit.
> 25*(3+5-(10/2))
parse_expr()
    parse_term()
        parse_factor()
            parse_num_op()
        parse_rest_term()
            parse_factor()
                parse_num_op()
                    parse_expr()
                        parse_term()
                            parse_factor()
                                parse_num_op()
                            parse_rest_term()
                        parse_rest_expr()
                            parse_term()
                                parse_factor()
                                    parse_num_op()
                                parse_rest_term()
                            parse_rest_expr()
                                parse_term()
                                    parse_factor()
                                        parse_num_op()
                                            parse_expr()
                                                parse_term()
                                                    parse_factor()
                                                        parse_num_op()
                                                    parse_rest_term()
                                                        parse_factor()
                                                            parse_num_op()
                                                        parse_rest_term()
                                                parse_rest_expr()
                                    parse_rest_term()
                                parse_rest_expr()
            parse_rest_term()
    parse_rest_expr()
75.000000
>
```

Conclusion:

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. All in all, CLC should definitely be in your productivity tool set.