

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203



18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

COMMAND LINE CALCULATOR

Gudied by:

M.Anand

Submitted By:

RA2011031010004 (Riddhisatwa Ghosh)

INDEX

- OBJECTIVE AND SCOPE
- ABSTRACT
- INTRODUCTION
- HARDWARE/SOFTWARE REQUIREMENTS
- ALGORITHM
- ARCHITECTURE DESIGN/BLOCK DIAGRAM
- SOURCE CODE
- RESULT/OUTPUT
- CONCLUSION
- REFERENCES

OBJECTIVE AND SCOPE

The objective of a command line calculator project is to create a program that allows users to perform arithmetic and mathematical operations via the command line interface. The scope of the project can vary depending on the specific requirements and goals, but it may include features such as:

- ▶ Accepting user input for mathematical expressions, including basic arithmetic operations (+, -, *, /), parentheses, and functions (e.g., sin, cos, sqrt).
- ▶ Evaluating the input expression and producing the corresponding output.
- ▶ Handling errors such as invalid input or division by zero.
- ▶ Allowing the user to store and recall previous calculations.
- ▶ Providing additional features such as support for complex numbers, units conversion, or plotting graphs.

The project can be implemented in various programming languages, including Python, C++, or Java. It may involve parsing input strings, implementing algorithms for arithmetic operations, and designing a user-friendly interface. Overall, the goal is to create a reliable and efficient calculator program that can be used via the command line interface.

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.

INTRODUCTION

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 a declaration of the variables. To Evaluate the given expressions. To perform basic calculations.

HARDWARE/SOFTWARE REQUIREMENTS

Software Requirements:

- C compiler (gcc, cc, egcs,...)

Hardware Requirements:

- CPU : Intel Core i5
- Memory : 8GB for RAM

ALGORITHM

Step 1 — START

Step 2 — input

Step 3 — parse_expr()

Step 3.1 parse_term()

Step 3.1.1 parse_factor()

Step 3.2 parse_num_op()

Step 3.2.1 parse_rest_term()

Step 3.3 parse_factor()

Step 3.3.1 parse_num_op()

Step 3.4 parse_rest_term()

Step 3.4.1 parse_rest_expr()

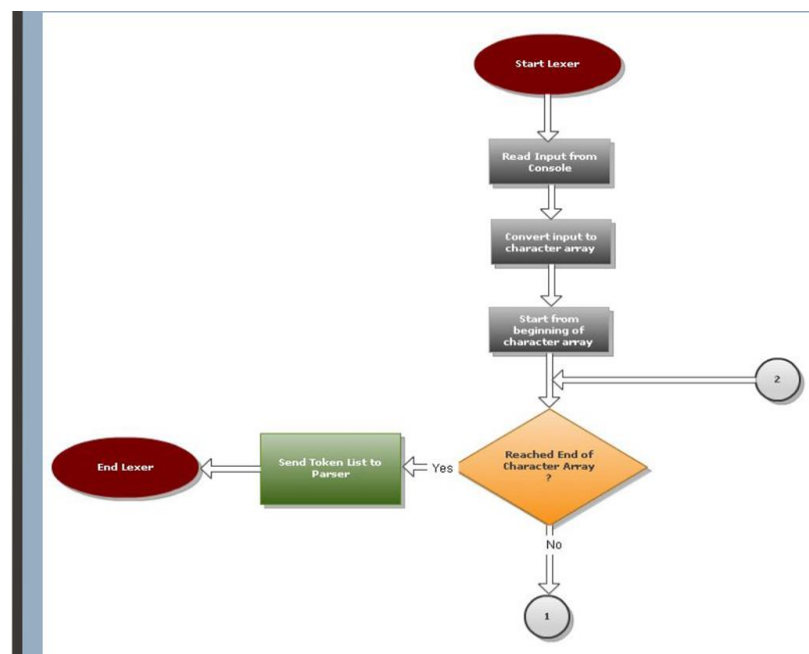
Step 4 — STOP

ARCHITECTURE DESIGN/BLOCK DIAGRAM

LEXER:

- A lexer is a software program that performs lexical analysis.
- Lexical analysis is the process of separating a stream of characters into different words, which in computer science we call 'tokens' .
- For Example -While reading we are performing the lexical operation of breaking the string of text at the space characters into multiple words.

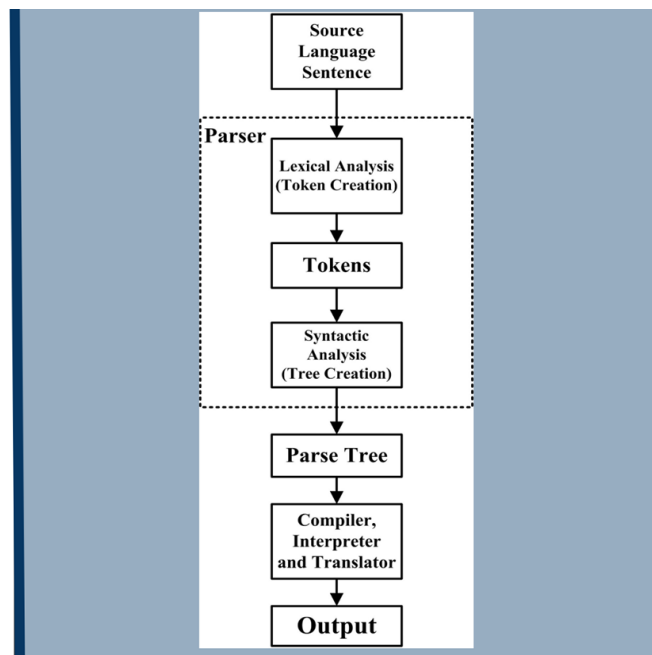
LEXER



PARSER:

- A parser goes one level further than the lexer and takes the tokens produced by the lexer and tries to determine if proper sentences have been formed.
- Parsers work at the grammatical level, lexers work at the word level.
- Generally yacc is used to parse language syntax.
- The name "yacc" stands for "Yet Another Compiler Compiler" .
- Yacc uses a parsing technique known as LR-parsing or shift-reduce parsing.

PARSER



SOURCE CODE

```
Users > prince > C o3.c > print_val(val_t *)
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <math.h>
5  #include <setjmp.h>
6
7  #define VAR_NAME_SIZE 31
8  typedef struct _MapEntry_t {
9      char name[VAR_NAME_SIZE+1];
10     double value;
11     struct _MapEntry_t* next;
12 } MapEntry_t;
13
14 MapEntry_t* varmap;
15
16 void
17 map_init(void)
18 {
19     varmap = 0;
20 }
21
22 void
23 map_clear(void)
24 {
25     MapEntry_t* cur = varmap;
26     while( cur ) {
27         MapEntry_t* next = cur->next;
28         free( cur );
29         cur = next;
30     }
31
32     varmap = 0;
33 }
34
35 MapEntry_t*
36 map_find( const char* var )
37 {
38     MapEntry_t* cur = varmap;
39     while( cur ) {
40         if ( strcmp( var, cur->name ) == 0 ) {
41             return cur;
42         }
43         cur = cur->next;
```



```

Users > prince > C o3.c > print_val(val_t *)
43     cur = cur->next;
44 }
45
46     return 0;
47 }
48
49 void
50 map_add( const char* var, double value )
51 {
52     MapEntry_t* entry = map_find( var );
53     if ( entry == 0 ) {
54         entry = (MapEntry_t*)malloc( sizeof(MapEntry_t) );
55         strncpy( entry->name, var, VAR_NAME_SIZE + 1 );
56         entry->name[VAR_NAME_SIZE] = 0;
57         entry->next = varmap;
58         varmap = entry;
59     }
60
61     entry->value = value;
62 }
63
64 int
65 map_lookup( const char* var, double* value )
66 {
67     MapEntry_t* entry = map_find( var );
68     if ( entry ) {
69         *value = entry->value;
70         return 1;
71     }
72
73     return 0;
74 }
75
76 #define TYPE_CHAR    0
77 #define TYPE_FLOAT   1
78 #define TYPE_EOF     2
79 #define TYPE_ERROR    3
80 #define TYPE_VARIABLE 4
81
82 typedef struct _val_t {
83     int type;
84     union {
85         double fval;

```

```

Users > prince > C o3.c > print_val(val_t *)
86     char cval;
87     char variable[255];
88     } d;
89 } val_t;
90 void
91 print_val( val_t* val )
92 {
93     if ( val->type == TYPE_FLOAT ) {
94         printf("%lf\n", val->d.fval );
95     } else if ( val->type == TYPE_CHAR ) {
96         printf("\'%c'\n", val->d.cval);
97     } else if ( val->type == TYPE_VARIABLE ) {
98         printf("Variable \'%s'\n", val->d.variable);
99     } else if ( val->type == TYPE_EOF ) {
100         printf("EOF\n");
101     } else if ( val->type == TYPE_ERROR ) {
102         printf("ERROR\n");
103     } else {
104         printf("Bad val type: %d\n", val->type);
105     }
106 }
107 int argc;
108
109 /* command line arguments array */
110 char** argv;
111
112 /* array parsed so far. Used for debugging and printing out error messages. */
113 static char buffer[1024];
114
115 /* the token that was most recently scanned by the lexer */
116 val_t next_val;
117
118 /* which argument we are currently scanning */
119 int arg = 0;
120
121 /* the index into argv[arg] that we are currently scanning */
122 int argp = 0;
123
124 /* the position in buffer[] that we are storing characters. */
125 int bpos = 0;
126
127 static int have_next_val = 0;
128

```

```

Users > prince > C o3.c > print_val(val_t *)
132 reset(int pargc, char** pargv)
133 {
134     argc = pargc;
135     argv = pargv;
136     buffer[0] = 0;
137     arg = 0;
138     argp = 0;
139     bpos = 0;
140     have_next_val = 0;
141 }
142
143 /*****
144  Scanner. Scans tokens from the command line arguments.
145  *****/
146 void
147 lex(val_t* val, int next)
148 {
149     char token[25];
150     int tpos = 0;
151     int done = 0;
152     int number = 0;
153     enum {
154         read_start,
155         read_int,
156         read_mantissa,
157         read_hex,
158         read_var
159     } state = read_start;
160
161     if ( next ) {
162         have_next_val = 0;
163         return;
164     } else if ( have_next_val ) {
165         *val = next_val;
166         return;
167     }
168
169     while( !done ) {
170         /* get the next character. Add to buffer. Do not increment the next */
171         /* character to read. */
172         char ch;
173
174         if ( arg == argc ) {

```

```

Users > prince > C o3.c > print_val(val_t *)
174     if ( arg == argc ) {
175         val->type = TYPE_EOF;
176         val->d.fval = 0;
177         break;
178     }
179
180     ch = argv[arg][argp];
181     /*printf("argv[%d][%d] = %c (state=%d)\n", */
182     /*      arg, argp, argv[arg][argp], state); */
183
184     switch ( state ) {
185     case read_start:
186         if ( ch >= '0' && ch <= '9' ) {
187             state = read_int;
188             tpos = 0;
189             token[tpos++] = ch;
190         } else if ( ch == '+' || ch == '-' ||
191                   ch == '/' || ch == '*' ||
192                   ch == '(' || ch == ')' ||
193                   ch == '%' || ch == '^' ||
194                   ch == '=' )
195         {
196             val->type = TYPE_CHAR;
197             val->d.cval = ch;
198             done = 1;
199         } else if ( ch == ' ' || ch == '\t' || ch == 0 ) {
200
201         } else if ( ch == '.' ) {
202             tpos = 0;
203             token[tpos++] = '0';
204             token[tpos++] = '.';
205             state = read_mantissa;
206         } else if ( isalpha( ch ) ) {
207             state = read_var;
208             tpos = 0;
209             token[tpos++] = ch;
210         } else {
211             buffer[bpos] = 0;
212             printf("Parse error after: %s\n", buffer);
213             longjmp( env, 1 );
214         }
215         break;

```

```

Users > prince > C 07.c > VAR_NAME_SIZE
220     state = read_mantissa;
221 } else if ( isalpha( ch ) ) {
222     state = read_var;
223     tpos = 0;
224     token[tpos++] = ch;
225 } else {
226     buffer[bpos] = 0;
227     printf("Parse error after: %s\n", buffer);
228     longjmp( env, 1 );
229 }
230 break;
231 case read_int:
232     if ( ch >= '0' && ch <= '9' ) {
233         if ( tpos < sizeof(token) ) {
234             token[tpos++] = ch;
235         } else {
236             token[tpos] = 0;
237             printf("Number too long: %s\n", token);
238         }
239     } else if ( ch == 'x' && tpos == 1 ) {
240         state = read_hex;
241     } else if ( ch == '.' ) {
242         if ( tpos < sizeof(token) ) {
243             token[ tpos++ ] = ch;
244         } else {
245             token[tpos] = 0;
246             printf("Number too long: %s\n", token);
247         }
248         state = read_mantissa;
249     } else {
250         token[tpos] = 0;
251         state = read_start;
252         val->type = TYPE_FLOAT;
253         val->d.fval = (double)atoi(token);
254         done = 1;
255         goto done;
256     }
257 break;
258 case read_mantissa:
259     if ( ch >= '0' && ch <= '9' ) {
260         if ( tpos < sizeof(token) ) {
261             token[tpos++] = ch;
262         } else {

```

```

C od.c > reset(int, char **)
248     val->d.fval = number;
249     done = 1;
250     goto done;
251
252 }
253 break;
254 case read_var:
255     if ( ch >= 'a' && ch <= 'z' ||
256         ch >= 'A' && ch <= 'Z' ||
257         ch >= '0' && ch <= '9' ||
258         ch == '_' )
259     {
260         if ( tpos < sizeof(token) ) {
261             token[tpos++] = ch;
262         } else {
263             token[tpos] = 0;
264             printf("Variable too long: %s", token);
265             longjmp( env, 1 );
266         }
267     } else {
268         token[tpos] = 0;
269         state = read_start;
270         val->type = TYPE_VARIABLE;
271         strcpy( val->d.variable, token );
272         done = 1;
273         goto done;
274     }
275 }
276
277 if ( ch == 0 ) {
278     argp = 0;
279     argp++;
280 } else {
281     argp++;
282     buffer[bpos++] = ch;
283 }
284
285 }
286
287 done:
288     next_val = *val;
289     have_next_val = 1;
290     return;

```

```

C od.c > resolve_variable(val_t*)
332     if ( val->type == TYPE_VARIABLE ) {
333         lex( val, 1 );
334         return 1;
335     }
336
337     return 0;
338 }
339
340 void resolve_variable( val_t* val )
341 {
342     double fval;
343     if ( val->type != TYPE_VARIABLE ) {
344         printf("Error: value is not a variable.\n");
345         longjmp( env, 1 );
346     }
347
348     if ( !map_lookup( val->d.variable, &fval ) ) {
349         printf("%s not defined.\n", val->d.variable);
350         longjmp( env, 1 );
351     }
352
353     val->type = TYPE_FLOAT;
354     val->d.fval = fval;
355 }
356
357 void parse_term(val_t* val);
358 void parse_expr(val_t* val);
359 void parse_factor( val_t* val );
360 void parse_num_op( val_t* val );
361 void parse_factor( val_t* val );
362 void parse_rest_num_op( val_t* val );
363 void parse_rest_var( val_t* val );
364
365 //define DEBUG_PRINT 1
366 #ifndef DEBUG_PRINT
367 #define dprintf(A) printf(A)
368 #endif
369
370 int level = 0;
371 void printtab() {
372     int i = 0;
373     for( i = 0; i < level; i++ ) {

```

```

C od.c > match_eof()
291
292 int match_char( char ch )
293 {
294     val_t val;
295     lex(&val, 0);
296
297     if ( val.type == TYPE_CHAR && val.d.cval == ch ) {
298         lex( &val, 1 );
299         return 1;
300     }
301
302     return 0;
303 }
304
305 int match_eof()
306 {
307     val_t val;
308     lex(&val, 0);
309
310     if ( val.type == TYPE_EOF ) {
311         return 1;
312     }
313
314     return 0;
315 }
316
317 int match_num( val_t* val )
318 {
319     lex( val, 0 );
320
321     if ( val->type == TYPE_FLOAT ) {
322         lex( val, 1 );
323         return 1;
324     }
325
326     return 0;
327 }
328
329 int match_variable( val_t* val )
330 {
331     lex( val, 0 );
332
333     if ( val->type == TYPE_VARIABLE ) {

```

```

C od.c > parse_rest_term(val_t*)
374 void printtab() {
375     int i = 0;
376     for( i = 0; i < level; i++ ) {
377         dprintf(" ");
378     }
379 }
380
381 void parse_rest_term( val_t* val )
382 {
383     printtab();
384     dprintf("parse_rest_term()\n");
385     level++;
386     if ( match_char( '*' ) ) {
387         val_t val2;
388         parse_factor( &val2 );
389         val->d.fval *= val2.d.fval;
390         parse_rest_term( val );
391     } else if ( match_char( '/' ) ) {
392         val_t val2;
393         parse_factor( &val2 );
394         if ( val2.d.fval != 0 ) {
395             val->d.fval /= val2.d.fval;
396         } else {
397             printf("Division by 0\n");
398             longjmp( env, 0 );
399         }
400         parse_rest_term( val );
401     } else if ( match_char( '%' ) ) {
402         val_t val2;
403         parse_factor( &val2 );
404         if ( val2.d.fval != 0 ) {
405             val->d.fval = fmod( val->d.fval, val2.d.fval );
406         } else {
407             printf("Division by 0\n");
408             longjmp( env, 0 );
409         }
410         parse_rest_term( val );
411     } else if ( match_eof() ) {
412         return;
413     }

```

```

C cd.c > parse_term(val_t*)
113     level--;
114     return;
115 }
116
117 void parse_term( val_t* val )
118 {
119     printtab();
120     dprintf("parse_term()\n");
121     level++;
122
123     parse_factor( val );
124     parse_rest_term( val );
125
126     level--;
127     return;
128 }
129
130
131 void parse_rest_num_op( val_t* val )
132 {
133     if ( match_char( '^' ) ) {
134         val_t val2;
135         parse_num_op( &val2 );
136         val->d.fval = pow( val->d.fval, val2.d.fval );
137         parse_rest_num_op( val );
138     }
139     return;
140 }
141
142 void parse_num_op( val_t* val )
143 {
144     printtab();
145     dprintf("parse_num_op()\n");
146     level++;
147
148     if ( match_num( val ) ) {
149         parse_rest_num_op( val );
150     } else if ( match_variable( val ) ) {
151         resolve_variable( val );
152         parse_rest_num_op( val );
153     } else if ( match_char( '(' ) ) {
154         parse_expr( val );
155         if ( !match_char( ')' ) ) {
156             printtab();
157             dprintf("parse_rest_expr()\n");
158             level++;
159
160             if ( match_char( '+' ) ) {
161                 val_t val2;
162                 parse_term( &val2 );
163                 val->d.fval += val2.d.fval;
164                 parse_rest_expr( val );
165             } else if ( match_char( '-' ) ) {
166                 val_t val2;
167                 parse_term( &val2 );
168                 val->d.fval -= val2.d.fval;
169                 parse_rest_expr( val );
170             } else if ( match_eof() ) {
171                 printtab();
172                 dprintf("parse_expr()\n");
173                 level++;
174
175                 if ( match_variable( val ) ) {
176                     parse_rest_var( val );
177                 } else {
178                     parse_term( val );
179                     parse_rest_expr( val );
180                 }
181
182                 level--;
183                 return;
184             }
185         }
186     }
187
188     level--;
189     return;
190 }
191
192 void parse_expr( val_t* val )
193 {
194     printtab();
195     dprintf("parse_expr()\n");
196     level++;
197
198     if ( match_variable( val ) ) {
199         parse_rest_var( val );
200     } else {
201         parse_term( val );
202         parse_rest_expr( val );
203     }
204
205     level--;
206     return;
207 }
208
209 void parse_rest_var( val_t* val )
210 {
211     printtab();
212     dprintf("parse_rest_var()\n");
213     level++;
214
215     if ( match_char( '=' ) ) {
216         val_t vexp;
217         parse_expr( &vexp );
218         if ( vexp.type != TYPE_FLOAT ) {
219             printtab();
220             dprintf("Error: Tried to assign non-number to %s.\n", val->d.variable);
221             longjmp( env, 1 );
222         }
223
224         printtab();
225         dprintf("Assigned to %s: ", val->d.variable);
226         map_add( val->d.variable, vexp.d.fval );
227         *val = vexp;
228     }
229     else {
230         parse_rest_num_op( val );
231     }
232 }
233
234 int
235 parse( val_t* val )
236 {
237     if ( setjmp( env ) ) {
238         return 0;
239     }
240
241     parse_expr( val );
242     if ( !match_eof() ) {
243         printtab();
244         dprintf("Trailing characters.\n");
245         longjmp( env, 1 );
246     }
247
248     return 1;
249 }
250
251 void
252 usage(void)
253 {
254     printf("Usage: calc [mathematical expression]\n");
255     exit(-1);
256 }
257
258 int
259 main( int argc, char* argv[] )
260 {
261     val_t val;
262     if ( !parse( &val ) ) {
263         usage();
264     }
265     printf("Result: %s\n", val.d.variable);
266 }

```



```

C od.c > parse_term(val_t*)
776 {
777     val_t val;
778     map_init();
779
780     if ( pargc == 1 ) {
781         char cmd[100];
782         char* cmds = cmd;
783         int cmdlen = 0;
784         cmd[0] = 0;
785
786         printf("Use Control-C to quit.\n");
787
788         for( ;; ) {
789             top:
790                 // print command line.
791                 printf( "\r> %s", cmd );
792
793                 cmdlen = strlen(cmd);
794
795                 for( ;; ) {
796                     char c = _getch();
797                     if ( c == '\b' ) {
798                         if ( cmdlen > 0 ) {
799                             cmd[--cmdlen] = 0;
800                             printf( "\r> %s \b", cmd );
801                         }
802                     } else if ( c == '\r' ) {
803                         putc('\n', stdout);
804                         break;
805                     } else if ( c == 3 ) {
806                         printf("QUIT\n");
807                         exit(0);
808                     } else if ( cmdlen < sizeof(cmd)-1 ) {
809                         putc(c, stdout);
810                         //printf("%d\n", c);
811                         cmd[cmdlen++] = c;
812                         cmd[cmdlen] = 0;
813                     }
814                 }
815
816                 reset( 1, &cmds );
817

```

```

C od.c > parse_rest_expr(val_t*)
618     /* parse the expression. */
619     if ( parse( &val ) ) {
620         /* print the value. */
621         print_val( &val );
622     } else {
623         printf("Error.\n");
624     }
625 }
626
627
628 reset( pargc - 1, pargv + 1 );
629 /* parse the expression. */
630 parse_expr( &val );
631
632 /* print the value. */
633 print_val( &val );
634
635 map_clear();
636
637 return 0;
638
639

```


RESULT/OUTPUT

```
C:\Users\saich\Downloads\calc.exe
Use Control-C to quit.
> 5+5
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()
10.000000
> 5*3
parse_expr()
  parse_term()
    parse_factor()
      parse_num_op()
    parse_rest_term()
      parse_factor()
        parse_num_op()
      parse_rest_term()
    parse_rest_expr()
15.000000
> 10/2
parse_expr()
  parse_term()
    parse_factor()
      parse_num_op()
    parse_rest_term()
      parse_factor()
        parse_num_op()
      parse_rest_term()
    parse_rest_expr()
5.000000
> 10/2
```

```
C:\Users\saiqh\Downloads\calc.exe
> (1)+(2+(3*2))
parse_expr()
  parse_term()
    parse_factor()
      parse_num_op()
        parse_expr()
          parse_term()
            parse_factor()
              parse_num_op()
                parse_rest_term()
                  parse_rest_expr()
                    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_rest_expr()
                                          parse_term()
                                            parse_factor()
                                              parse_num_op()
                                                parse_rest_term()
                                                  parse_rest_expr()
                                                    parse_rest_term()
                                                      parse_rest_expr()
                                                        parse_rest_term()
                                                          parse_rest_expr()
9.000000
>
```

C:\Users\HP\Downloads\calc.exe

```

    parse_factor()
    parse_num_op()
    parse_rest_term()
    parse_rest_expr()
    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_rest_term()
    parse_rest_expr()
    parse_term()
    parse_factor()
    parse_num_op()
    parse_rest_term()
    parse_factor()
    parse_num_op()
    parse_rest_term()
    parse_rest_expr()
    -34.000000
> (3-4*4+6/2)+3-7-5*4
```

```
    parse_factor()
        parse_num_op()
        parse_rest_term()
    parse_rest_expr()

    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_rest_term()
    parse_rest_expr()
    parse_term()
    parse_factor()
    parse_num_op()
    parse_rest_term()
    parse_factor()
    parse_num_op()
    parse_rest_term()
    parse_rest_expr()
-34.000000
> (3-4*4+6/2)+3-7-5*4
```

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.

REFERENCES

- [1] — <https://www.codeproject.com/Articles/12395/A-Command-Line-Calculator>
- [2] — <https://fedoramagazine.org/bc-command-line-calculator/>