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

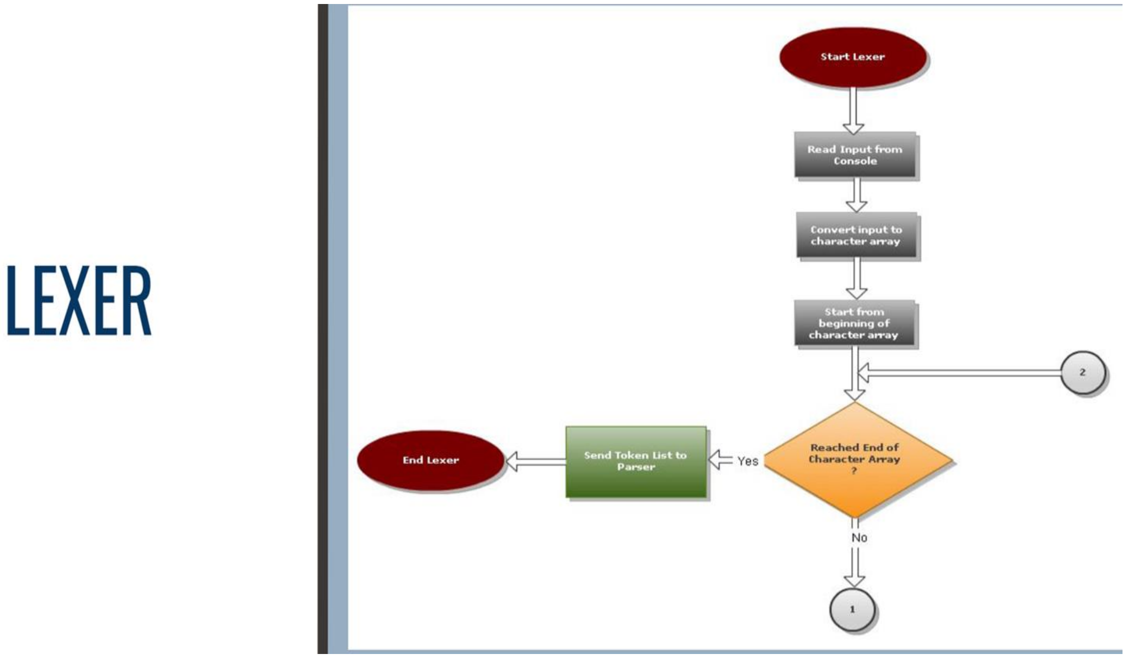
ALGORITHM

```
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

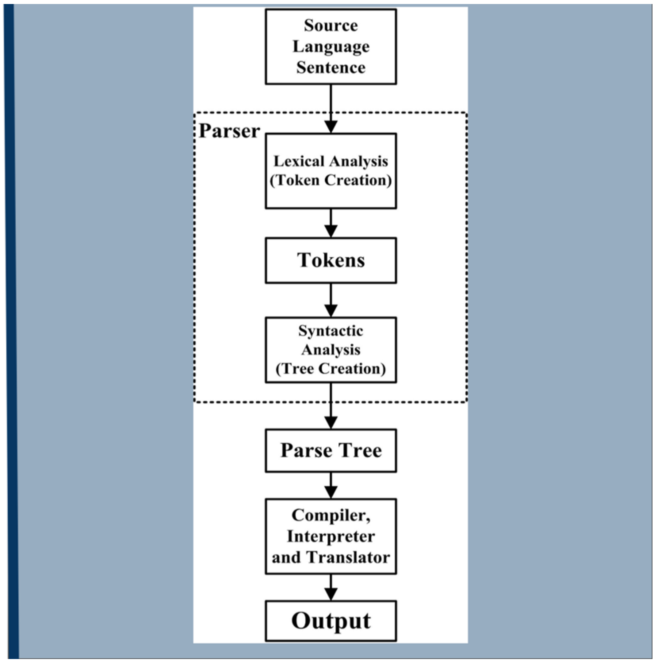
Start Lexer
Read Input from Console
Convert input to character array
Start from beginning of character array
2
Reached End of Character Array ?
Yes
Send Token List to Parser
End Lexer
No
1

PARSER:

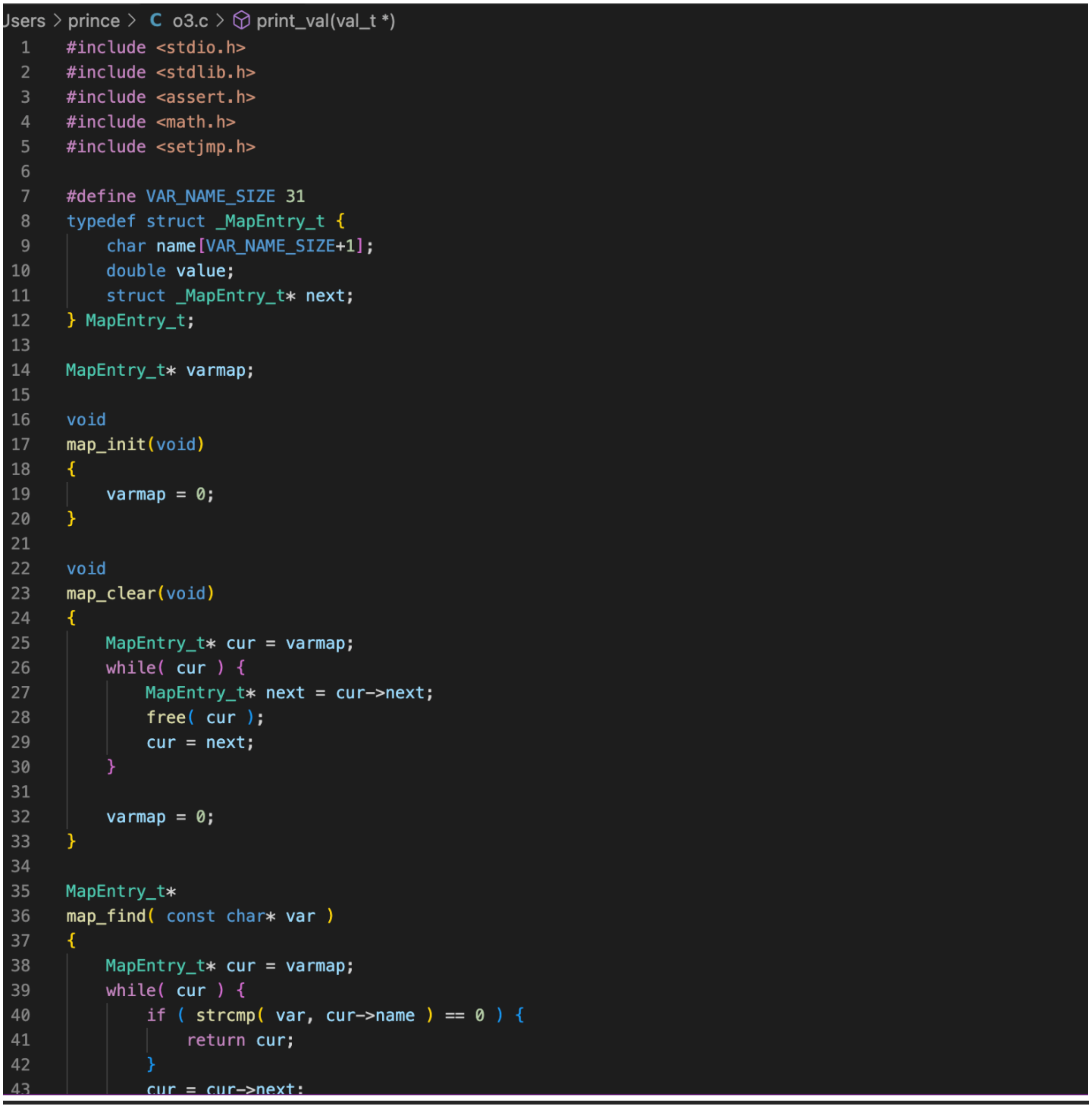
- A parser goes one level further than the lexer and takes the tokens produced by the lexer and tries to determine if proper sentences have been formed.

- 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

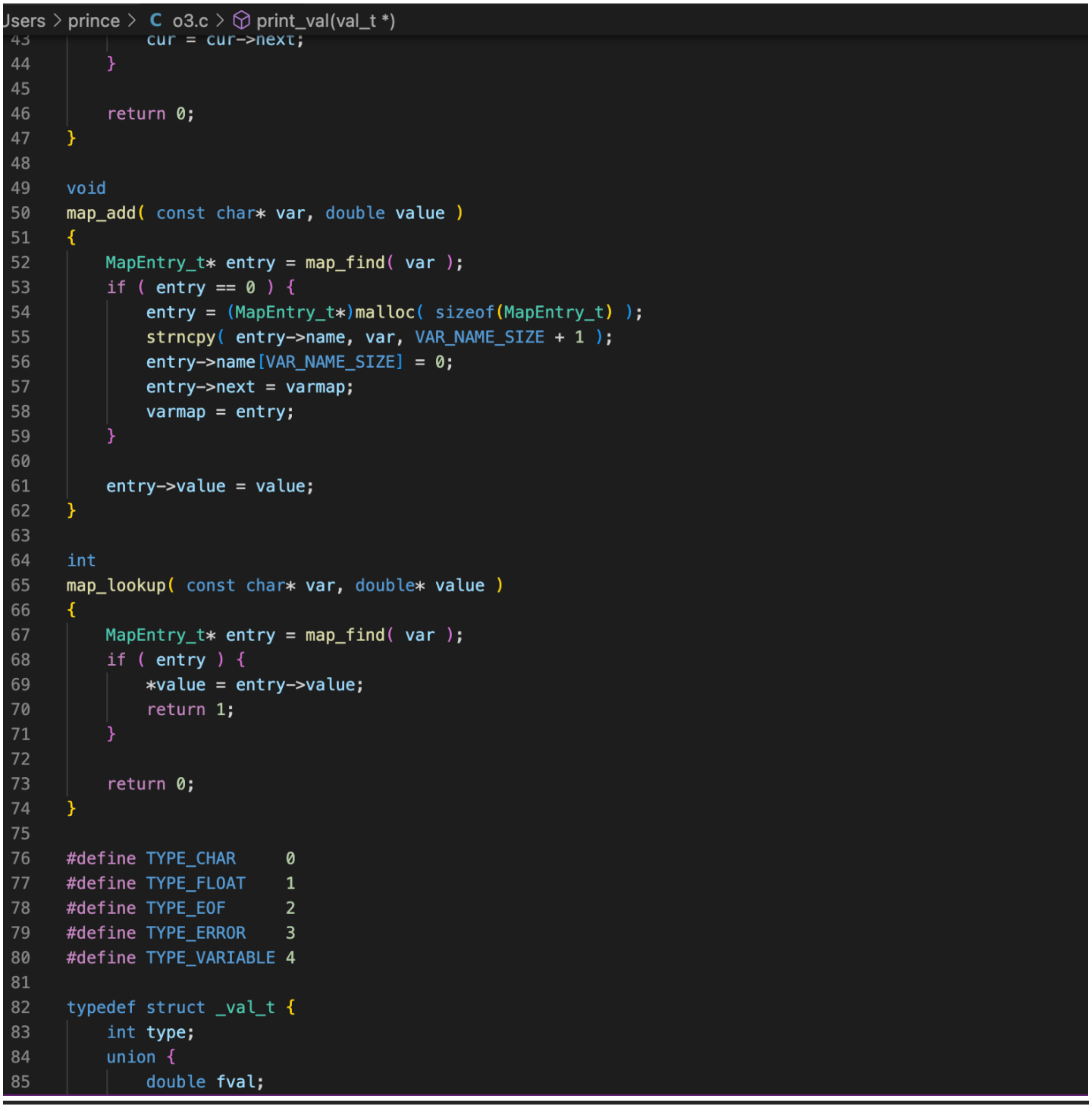

Source Language Sentence
Parser
Lexical Analysis (Token Creation)
Tokens
Syntactic Analysis (Tree Creation)
Parse Tree
Compiler, Interpreter and Translator
Output

SOURCE CODE

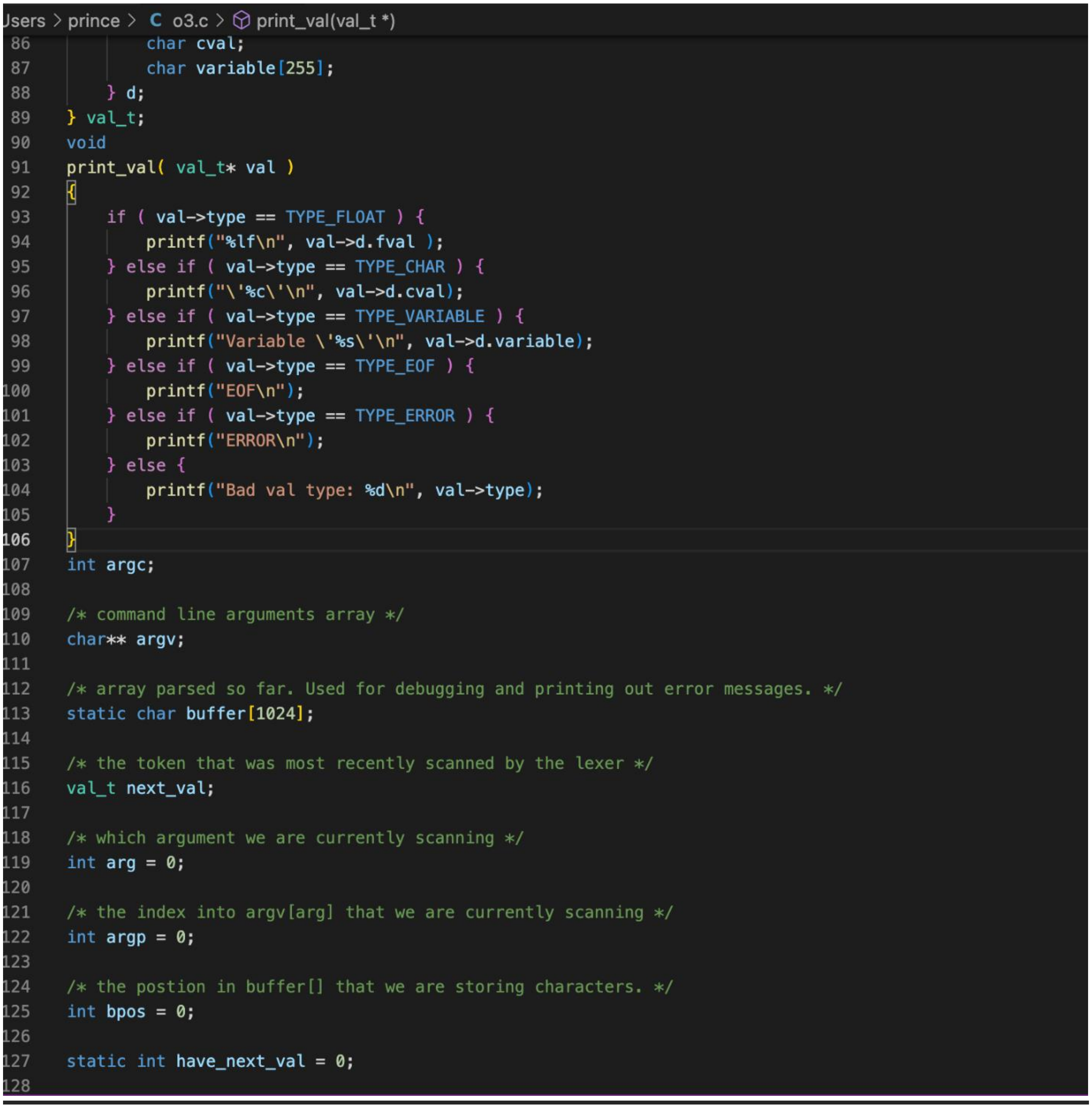


Users > prince > C o3.c > print_val(val_t *)

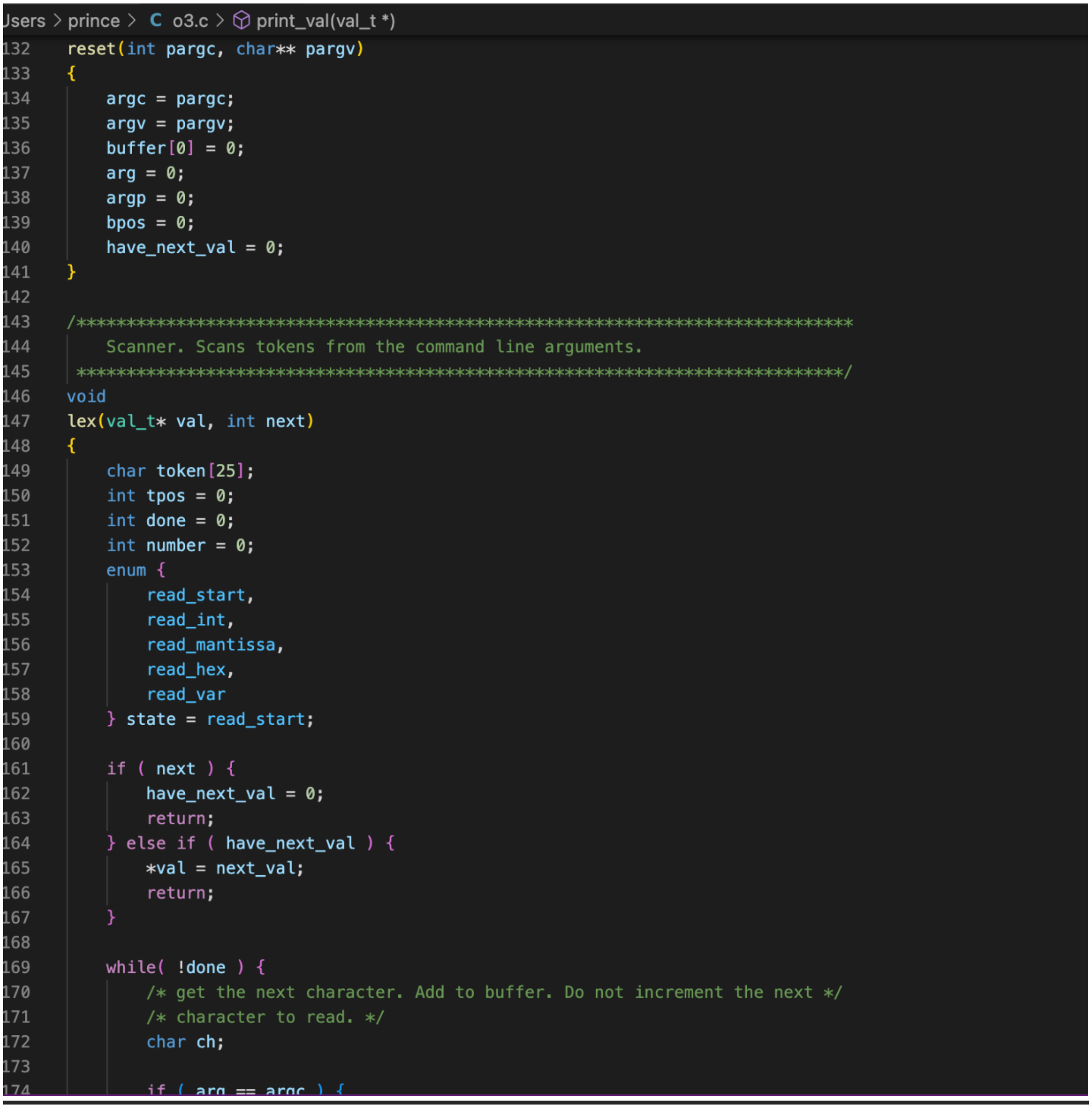
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <setjmp.h>

#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;
```



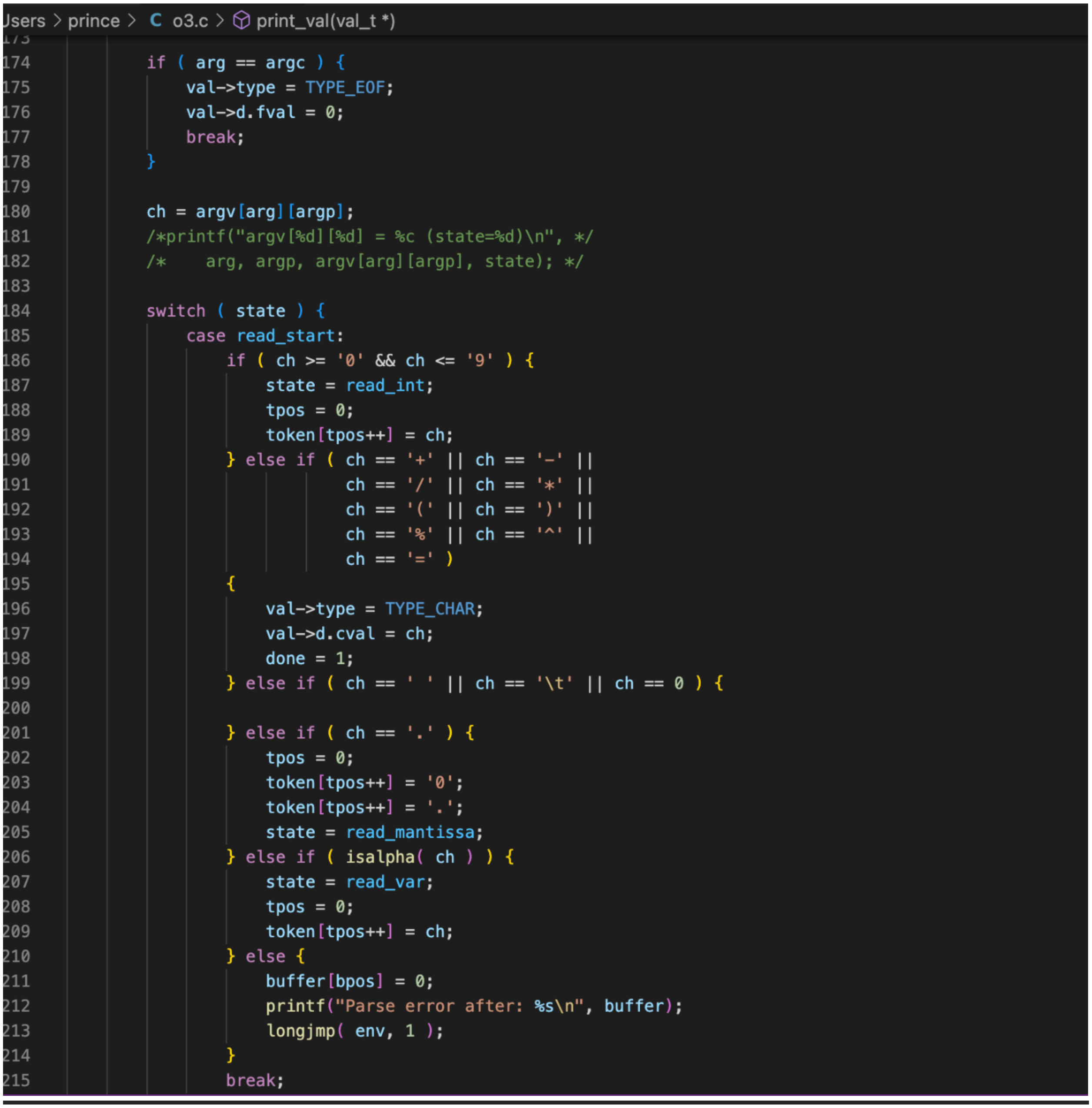
```
        cur = cur->next;
    }

    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;
```



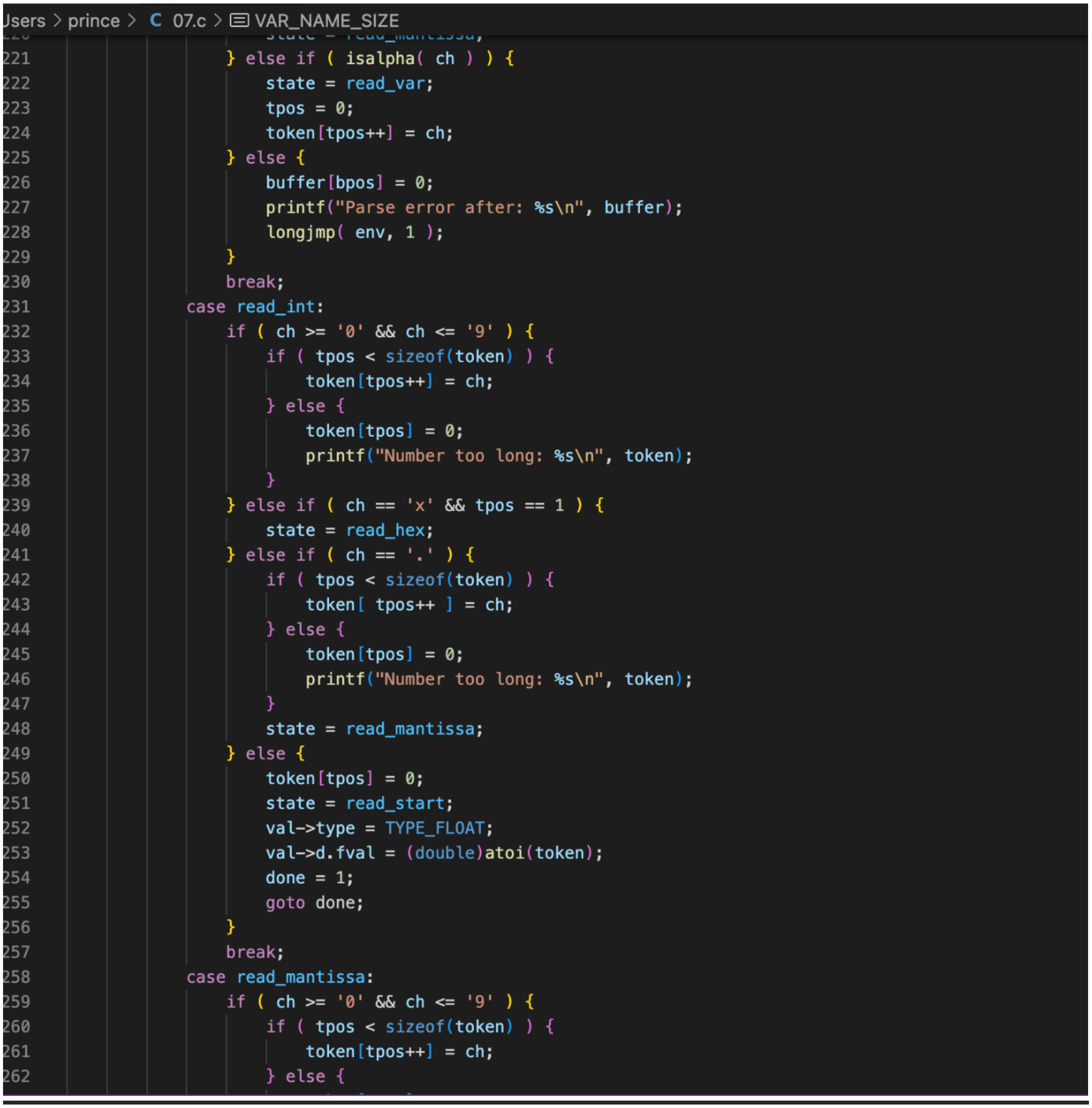
```
        char cval;
        char variable[255];
    } d;
} val_t;
void
print_val( val_t* val )
{
    if ( val->type == TYPE_FLOAT ) {
        printf("%lf\n", val->d.fval );
    } else if ( val->type == TYPE_CHAR ) {
        printf("\'%c\'\n", val->d.cval);
    } else if ( val->type == TYPE_VARIABLE ) {
        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;

/* 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;
```



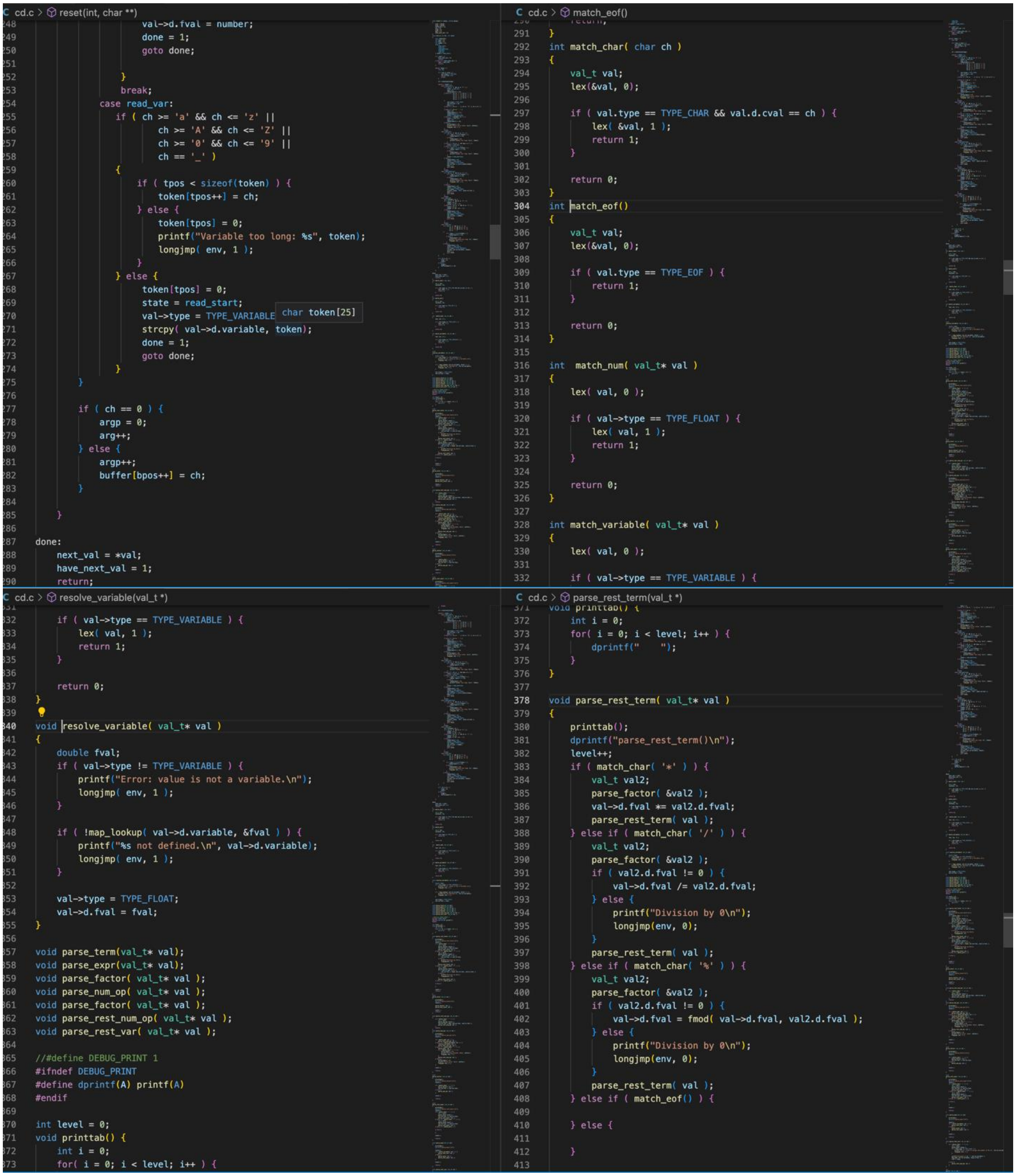
```
reset(int pargc, char** pargv)
{
    argc = pargc;
    argv = pargv;
    buffer[0] = 0;
    arg = 0;
    argp = 0;
    bpos = 0;
    have_next_val = 0;
}

/******************************************************************************
    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 ) {
```



```c
            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;
```



```
                        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 {
```

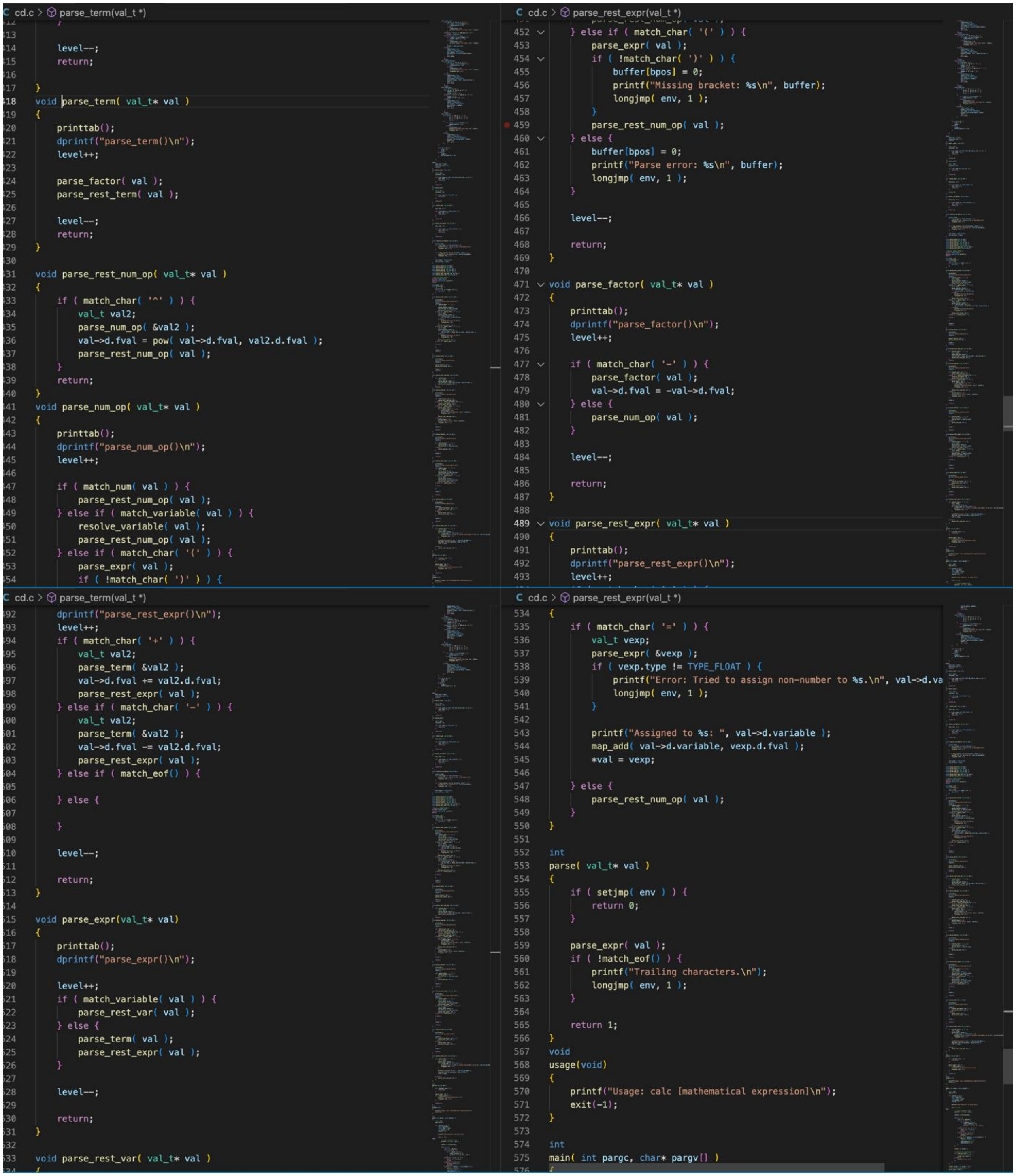


```c
                    val->d.fval = number;
                    done = 1;
                    goto done;

                }
                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;
```

```c
    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++ ) {
```

```c
    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 ) {
```

```c
void printtab() {
    int i = 0;
    for( i = 0; i < level; i++ ) {
        dprintf("    ");
    }
}

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 {

    }

```

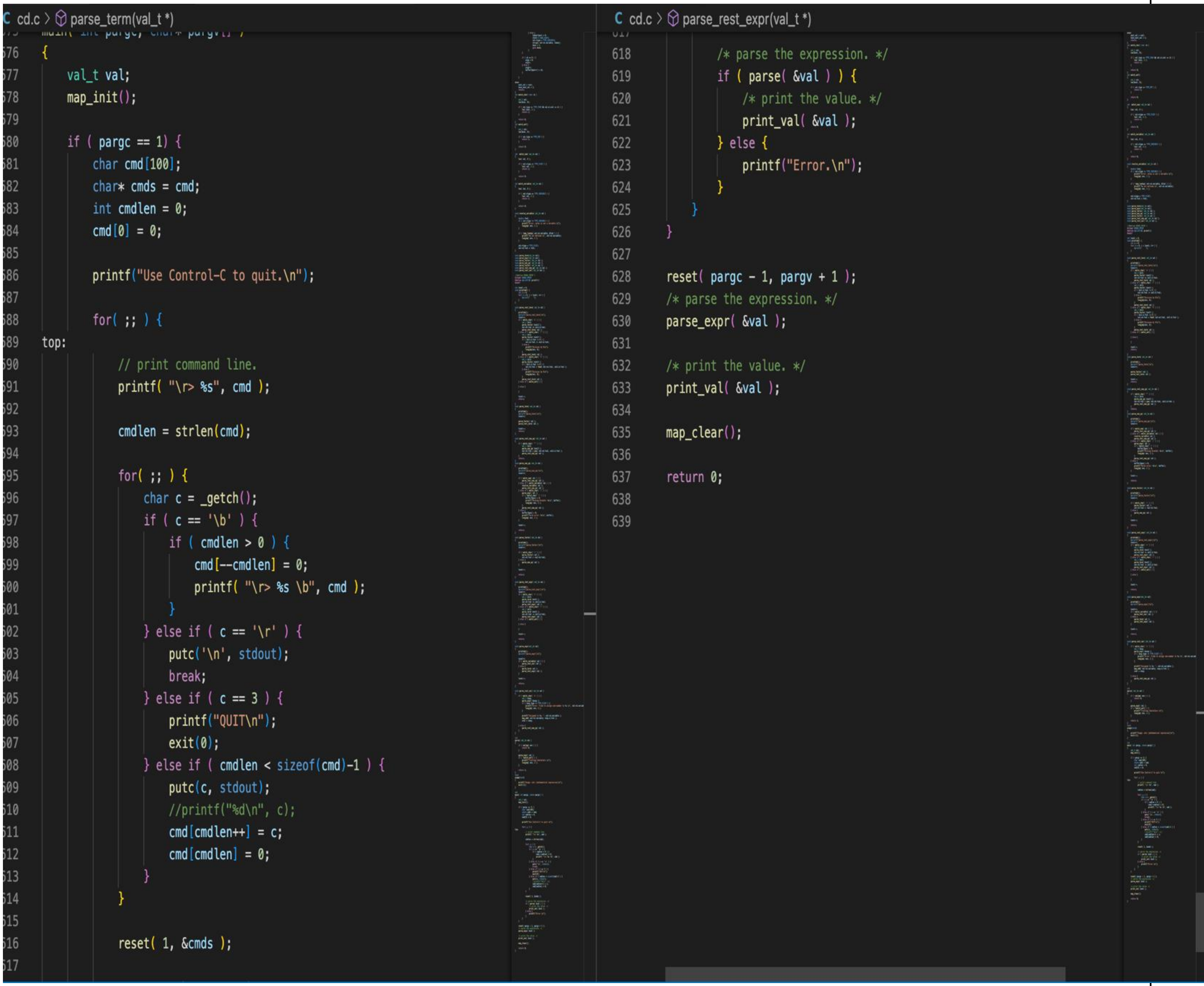

cd.c > parse_term(val_t *)
cd.c > parse_rest_expr(val_t *)
cd.c > parse_term(val_t *)
cd.c > parse_rest_expr(val_t *)

```c
    level--;
    return;

}
void parse_term( val_t* val )
{
    printtab();
    dprintf("parse_term()\n");
    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();
    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( ')' ) ) {
```

```c
    } 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();
    dprintf("parse_factor()\n");
    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();
    dprintf("parse_rest_expr()\n");
    level++;
```

```c
    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;
}

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;
}

void parse_rest_var( val_t* val )
```

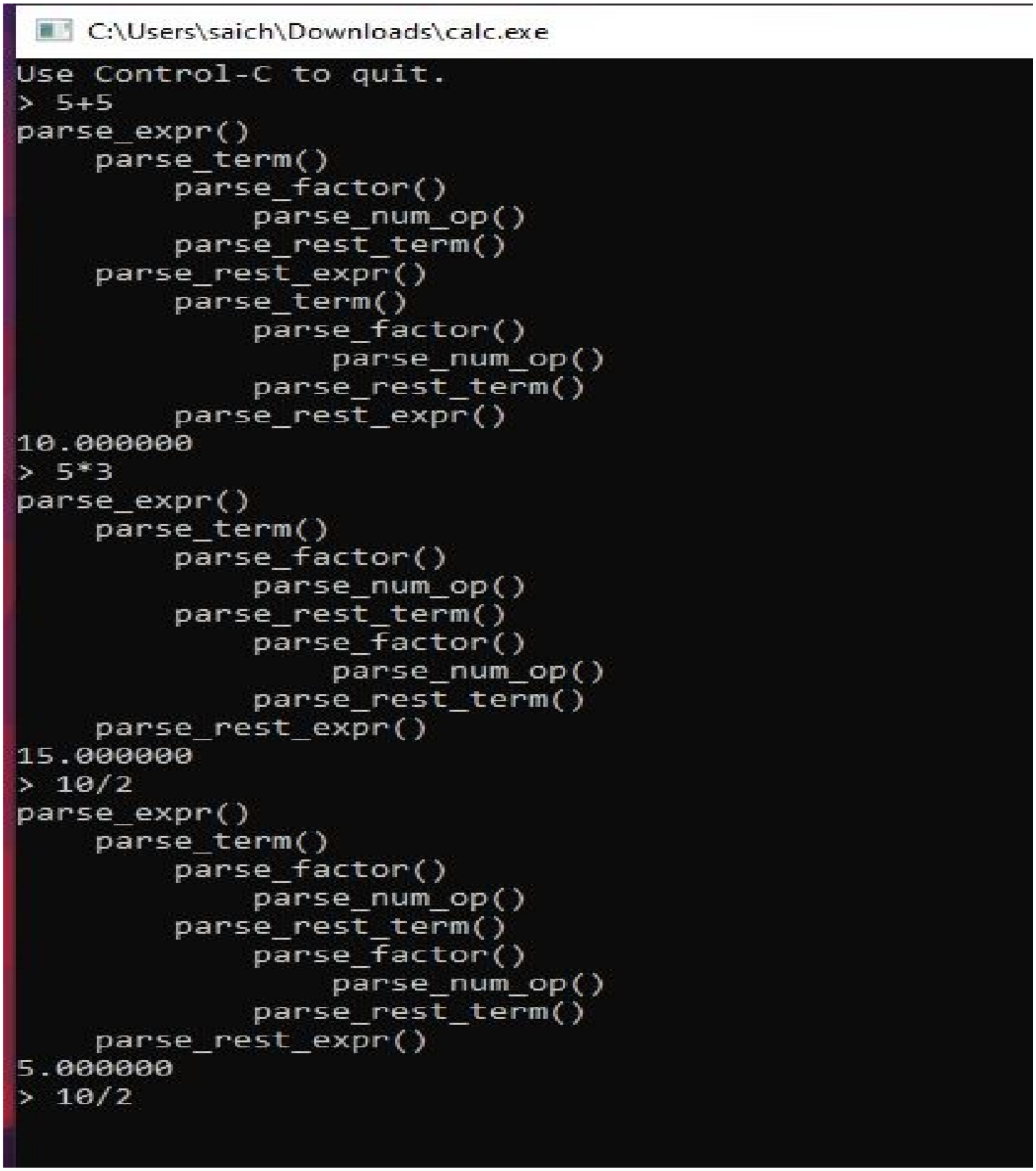
```c
{
    if ( match_char( '=' ) ) {
        val_t vexp;
        parse_expr( &vexp );
        if ( vexp.type != TYPE_FLOAT ) {
            printf("Error: Tried to assign non-number to %s.\n", val->d.va
            longjmp( env, 1 );
        }

        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[] )
```



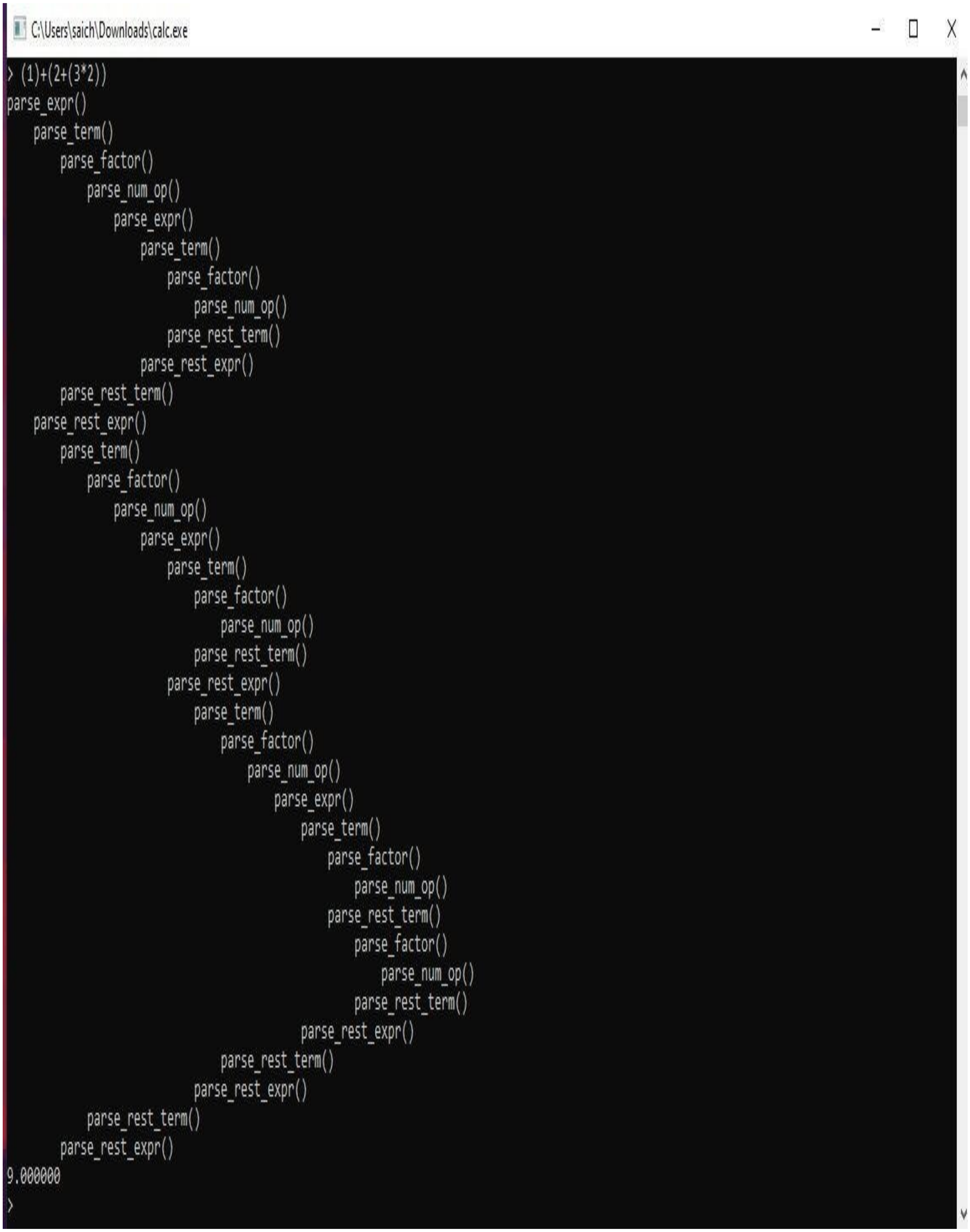
```c
main( int pargc, char* pargv[] )
{
    val_t val;
    map_init();

    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 );

```

```c
            /* 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;

```

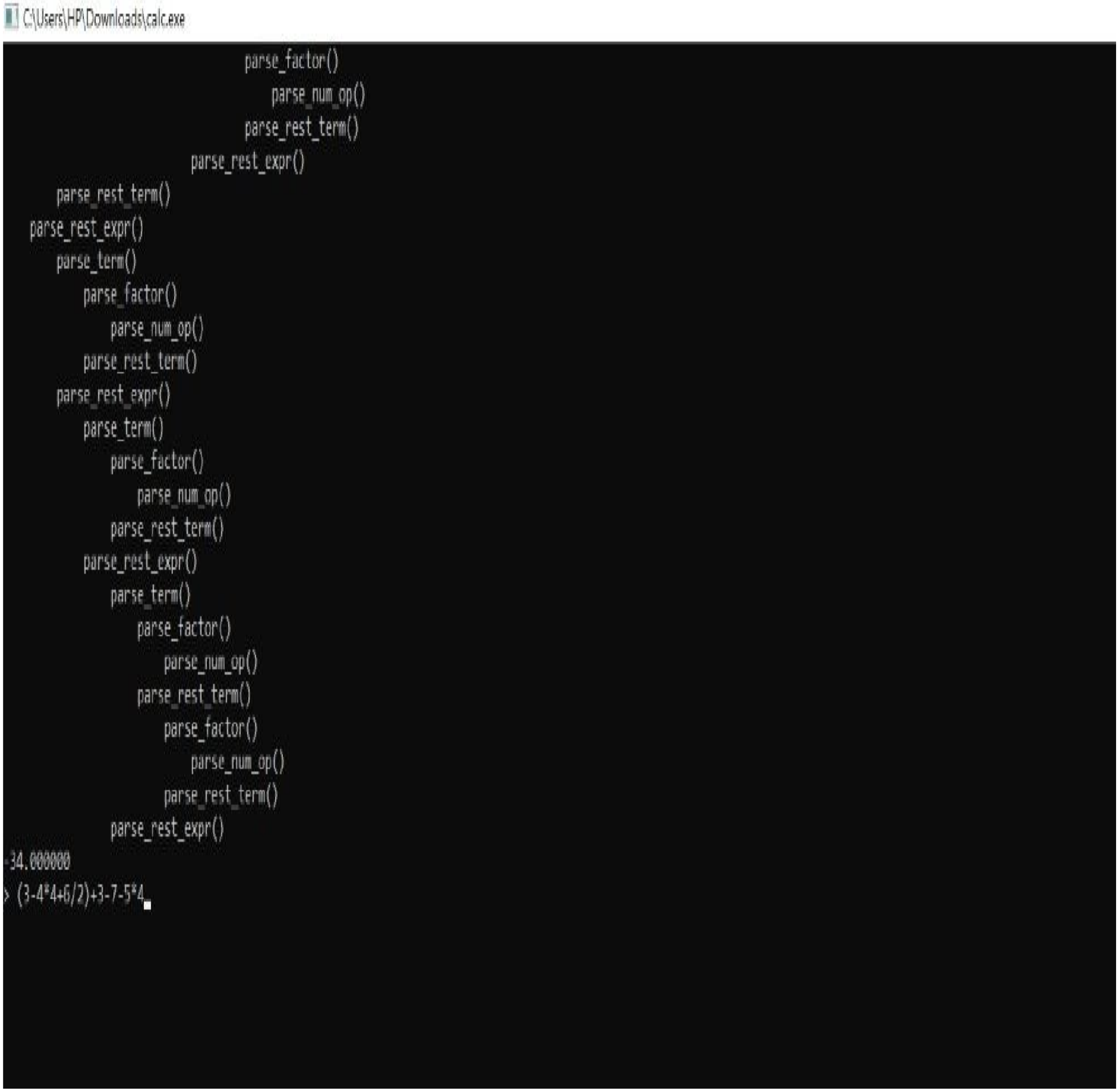
RESULT/OUTPUT



```
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
```

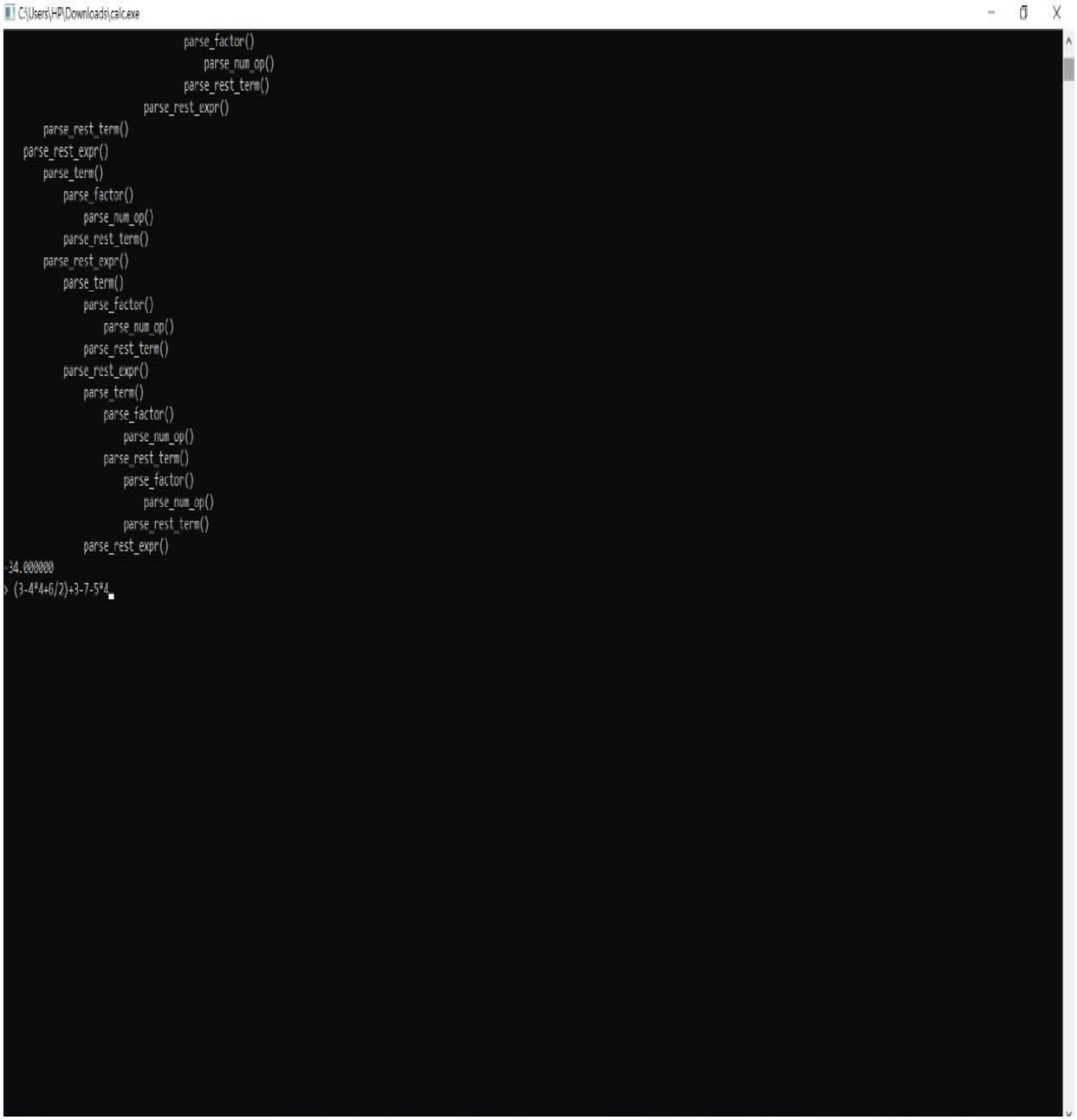


```
> (1)+(2+(3*2))
parse_expr()
    parse_term()
        parse_factor()
            parse_num_op()
                parse_expr()
                    parse_term()
                        parse_factor()
                            parse_num_op()
                        parse_rest_term()
                    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_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()
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/