AdvCalc

Yusuf AKIN & Atakan YASAR

2023-03-29

Contents

1 AdvCalc

AdvCalc is an CLI calculator implemented in **c**. The application works as a Read-Execute-Print-Loop (REPL).

1.1 Build and Execution

To build the program, ensure **gcc** is installed in your machine. Project has no dependencies other than the standard c library.

To compile the program from the source, run makefile from the top level directory:

```
make advcalc
```

The default build target is already advcalc, running make without any argument works as well:

```
make
```

The executable produced is named **"advcalc"**, to start the program execute the binary directly:

```
./advcalc
```

If build process has gone smoothly you should see the following prompt:

```
./advcalc
>
```

Now the calculator is ready to use. To exit the program, input **"EOF"** (i.e. **"Ctrl + D"**).

1.2 Program Interface

1.2.1 Variables and Constants

All the variables and constants are **64-bit** integers.

- Variable names are case-sensitive.
- Variable names can contain only uppercase and lowercase letters.
- Variables with no previous assigned value are treated as 0.
- Reserved words cannot be used as variable names (e.g. xor, not, ls, rs, rr, lr).

1.2.2 Operators

All the operations listed below work with **64-bit** integers.

1.2.2.1 Binary Arithmetic Operations AdvCalc supports the following binary infix arithmetic operations:

Operation	Use	Description
$+$	$a + b$	Sum of a and b
$-$	$a - b$	Difference of a and b
$*$	$a * b$	Product of a and b

1.2.2.2 Binary Bitwise Operations AdvCalc supports the following binary infix bitwise operations:

Operation	Use	Description
&	a & b	Bitwise AND of a and b
|	$a \mid b$	Bitwise OR of a and b

1.2.2.3 Unary Bitwise Operations The only unary bitwise operation supported is the bitwise NOT operation, it is used as a prefix operator (i.e. a function)):

Operation	Use	Description
$not(.)$	$not(a)$	Bitwise complement of a

1.2.2.4 Other Bitwise Operations Following functions are supported for bitwise operations:

Operation	Use	Description
$xor(.,.)$	$xor(a,b)$	Bitwise XOR of a and b
$ls(.,.)$	$ls(a,b)$	Bitwise left shift of a by b bits
$rs(.,.)$	$rs(a,b)$	Bitwise right shift of a by b bits
$rr(.,.)$	$rr(a,b)$	Bitwise right rotation of a by b bits
$rl(.,.)$	$rl(a,b)$	Bitwise left rotation of a by b bits

1.2.3 Expressions

Expressions are evaluated in the following order, from highest to lowest precedence:

1. Parenthesized expressions (i.e. expressions enclosed in "(" and ")"), and function calls.
2. Multiplication ("*")
3. Addition ("+") and subtraction ("-"). The operations are left associative (i.e. $a-b+c$ is evaluated as $(a-b)+c$, not $a-(b+c)$).
4. Bitwise and ("&"))
5. Bitwise or ("|")

1.2.4 Assignment Statement

Assignment is done using the "=" operator. The left-hand side of the assignment must be a variable name. The right-hand side can be any valid expression.

```
> a = 5
> b = 10
> a + b
15
```

1.2.5 Comments

Comments are supported in the calculator. They start with "%". Everything after the "%" is ignored by the calculator.

```
> % This is a comment
> a = 5 % This is also a comment
> a
5
```

1.3 Program Structure and Implementation

AdvCalc is implemented in **c**. The program is composed of the following modules:

- **main.c**: The main module of the program. It contains the main function and the REPL loop.
- **lexer.c**: The lexer module. It contains the functions that tokenize the input string.
- **parser.c**: The parser module. It contains the functions that parse the input string and convert it to a parse tree.
- **executer.c**: The executer module. It contains the functions that execute the parse tree and evaluate the expression.
- **dictionary.c**: The dictionary module. It contains a hash table implementation used to store the variables.

1.3.1 main.c

The main module contains the main function and the REPL loop. The main function initializes the dictionary and starts the REPL loop. The REPL loop reads the input from the user, tokenizes it, parses it, and executes it. The REPL loop continues until the user inputs "Ctrl-d".

1.3.2 lexer.c

"lexer.c" implements the lexer module. The output is a stream of tokens.

```c
#define MAX_TOKEN_LEN 257
/* Token types */
```

```c
typedef enum TokenType {

  TOKEN_UNKNOWN,    // 0
  TOKEN_LITERAL,    // 1
  TOKEN_IDENTIFIER, // 2

  TOKEN_STAR,  // 3
  TOKEN_PLUS,  // 4
  TOKEN_MINUS, // 5

  TOKEN_AND, // 6
  TOKEN_OR,  // 7
  TOKEN_NOT, // 8
  TOKEN_XOR, // 9

  TOKEN_LS, // 10
  TOKEN_RS, // 11
  TOKEN_LR, // 12
  TOKEN_RR, // 13

  TOKEN_EQ,      // 14
  TOKEN_COMMENT, // 15
  TOKEN_EOF,     // 16

  TOKEN_LEFT_PAREN,  // 17
  TOKEN_RIGHT_PAREN, // 18
  TOKEN_COMMA,       // 19
} TokenType;

/* Token struct
 * type: type of token
 * start: start location of token
 * length: length of token
 * value: value of token if it is a literal
 */
typedef struct Token Token;
struct Token {
  TokenType type;
  char *start;
  size_t length;
  long value;
};

/* Lexer struct
```

```
 * input: input string
 * cur_pos: current position in input string
 * input_len: length of input string
 * token_list: list of tokens
 * cur_token: current token
 */
typedef struct Lexer Lexer;
struct Lexer {
  char *input;
  size_t cur_pos;
  size_t input_len;

  Token token_list[MAX_TOKEN_LEN];
  size_t cur_token;
};


/* Create a new lexer
 * input: input string
 * input_len: length of input string
 */
Lexer *lexer_new(char *input, size_t input_len);


/* Free a lexer
 * lexer: lexer to free
 */
void lexer_free(Lexer *lexer);

/* Prints the tokinized input
 * lexer: lexer to print
 */
void print_lex(Lexer *lexer);

/* Get the next token
 * lexer: lexer to get token from
 */
void lexer_next(Lexer *lexer);
```

1.3.3 parser.c

”parser.c” implements the parser module. The output is a parse tree. Used grammar:

```
    <expr> = <term><eof> | <eof>
    <term> = <term><binop><term> | <primary> | (<term>)
```

```
    <primary> = <func> | <var>
    <func> = <funcname>(<term>,<term>) | <unop>(<term>)
    <var> = literal or identifier token
    <binop> = [*+-&|]
    <funcname> = (xor|ls|rs|lr|rr)
    <unop> = not
```

```c
typedef enum SyntaxNodeType { BINOP, UNOP, PAREN, TOKEN, ERROR } SyntaxNodeType;

typedef struct SyntaxNode SyntaxNode;

/*
 * SyntaxNode Struct
 * type: type of the node,
 *       BINOP: left child, mid child as binary operator, right child,
 *       UNOP: left child, mid child as unary operator,
 *       PAREN: left child as inside of parenthesis,
 *       TOKEN: token type for leave
 * left: left term of the tree
 * mid: contains operators
 * right: right term of the tree
 * token: leave of the tree, tokens are stored here
 */
struct SyntaxNode {
  SyntaxNodeType type;
  SyntaxNode *left;
  SyntaxNode *mid;
  SyntaxNode *right;
  Token *token;
};

typedef struct SyntaxTree SyntaxTree;

struct SyntaxTree {
  SyntaxNode *root;
};

/*
 * generates a parse tree
 * calls parse_expr
 * input: Token array
 */
SyntaxTree *parse(Token *tokens);
```

```c
/*
 * <expr> = <term><eof> | <eof>
 * calls parse_term then expects EOF token
 * input: current token
 */
SyntaxNode *parse_expr(Token **tokens);

/*
 * parses <term>
 * calls and returns parse_comma
 */
SyntaxNode *parse_term(Token **tokens);

/*
 * parses <term>,<term>
 * calls parse_or and assigns it to left child as left <term>
 * then if comma exists
 * assigns TOKEN_COMMA to mid child
 * calls parse_comma and assigns it to right child as right <term>
 */
SyntaxNode *parse_comma(Token **tokens);

/*
 * parses <term>|<term>
 * calls parse_and and assigns it to left child as left <term>
 * then if or exists
 * assigns TOKEN_OR to mid child
 * calls parse_and and assigns it to right child as right <term>
 */
SyntaxNode *parse_or(Token **tokens);

/*
 * parses <term>&<term>
 * calls parse_plus_minus and assigns it to left child as left <term>
 * then if and exists
 * assigns TOKEN_AND to mid child
 * calls parse_plus_minus and assigns it to right child as right <term>
 */
SyntaxNode *parse_and(Token **tokens);

/*
 * parses <term>(+|-)<term>
 * calls parse_mul and assigns it to left child as left <term>
 * then if plus or minus exists
```

```
 * assigns TOKEN_PLUS or TOKEN_MINUS to mid child
 * calls parse_mul and assigns it to right child as right <term>
 */
SyntaxNode *parse_plus_minus(Token **tokens);

/*
 * parses <term>*<term>
 * calls parse_primary and assigns it to left child as left <term>
 * then if star exists
 * assigns TOKEN_STAR to mid child
 * calls parse_primary and assigns it to right child as right <term>
 */
SyntaxNode *parse_mul(Token **tokens);

/*
 * parses <primary>
 * calls and return parse_func
 */
SyntaxNode *parse_primary(Token **tokens);

/*
 * parses <funcname>(<term>,<term>)
 * first token must be valid <funcname>
 * then calls parse_paren
 * then if comma exits in parenthesis
 * assigns left <term> to left child
 * assigns related token to mid child
 * assigns right <term> to right child
 */
SyntaxNode *parse_func(Token **tokens);

/*
 * parses <var>
 * expects TOKEN_IDENTIFIER or TOKEN_LITERAL
 */
SyntaxNode *parse_var(Token **tokens);

/*
 * parses (<term>)
 * expects TOKEN_LEFT_PAREN as first token
 * calls parse_term
 * at the end expects TOKEN_RIGHT_PAREN as last token
 */
SyntaxNode *parse_paren(Token **tokens);
```

```c
/*
 * Free all syntax nodes recursively
 */
void free_syntax_tree(SyntaxTree *tree);

/*
 * prints syntax nodes indented according to depths
 */
void print_syntax_tree(SyntaxTree *tree);
```

1.3.4 executor.c

```c
/*
 * This function executes the statetement given as a input string, and writes the outp
 * return value represents the status of the execution.
 * 0: expression
 * 1: assignment
 * 2: error
 * 3: empty line
 *
 * dict: dictionary to store or retrieve variables
 *
 * The function creates its own lexer and parser, and uses them to parse the input str
 * The output string is written to the output string.
 */
int exec(Dictionary *dict, char *input, char *output);
```

1.3.5 dictionary.c

This module implements a hash table to store the variables. The hash table handles collisions using separate chaining.

```c
#define HASHSIZE 31013 // a large prime number

/*
 * Chain struct
 * next: pointer to next entry in chain
 * name: name of variable
 * value: value of variable
 */
typedef struct Chain Chain;
typedef struct Chain { /* table entry: */
    Chain *next; /* next entry in chain */
    char *name; /* defined name */
```

```
    long value;
}Chain;

/*
 * Dictionary struct
 * hashtab: array of pointers to chains
 */
typedef struct Dictionary Dictionary;

struct Dictionary{
  Chain* hashtab[HASHSIZE];
};

/*
 * Create a new dictionary
 */
Dictionary* new_dictionary();

/*
 * Set a variable in the dictionary
 * dict: dictionary to set variable in
 * name: name of variable
 * value: value of variable
 */
void set_var(Dictionary *dict, char *name, long value);

/*
 * Get a variable from the dictionary
 * dict: dictionary to get variable from
 * name: name of variable
 */
long get_var(Dictionary *dict, char *name);

/*
 * Free a dictionary
 * dict: dictionary to free
 */
void free_dict(Dictionary* dict);

/*
 * Print a dictionary
 * dict: dictionary to print
 */
void print_dict(Dictionary* dict);
```

```c
/*
 * Hash function
 * s: string to hash
 */
unsigned hash(char *s);
```

1.3.6 Example Module Outputs

```
./advcalc
> rr(lr(ls(rs(xor(((x)), x) | z + y, 1), ((((1)))), 1), 1) - qqq * not(not(10))
0
```

1.3.6.1 lexer.c Generated using `print_lex` :

```
Lexer{
    input: `rr(lr(ls(rs(xor(((x)), x) | z + y, 1), ((((1)))), 1), 1) - qqq * not(not(10))
    cur_pos: 76
    input_len: 76
    cur_token: 51
    Token List:{
        Token{
            type: TOKEN_RR
            value: 0
            length: 2
            start: 'rr'
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_LR
            value: 0
            length: 2
            start: 'lr'
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
```

```
        },
        Token{
            type: TOKEN_LS
            value: 0
            length: 2
            start: 'ls'
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_RS
            value: 0
            length: 2
            start: 'rs'
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_XOR
            value: 0
            length: 3
            start: 'xor'
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
```

```
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_IDENTIFIER
            value: 0
            length: 1
            start: 'x'
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_COMMA
            value: 0
            length: 1
            start: ','
        },
        Token{
            type: TOKEN_IDENTIFIER
            value: 0
            length: 1
            start: 'x'
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_OR
            value: 0
```

```
        length: 1
        start: '|'
    },
    Token{
        type: TOKEN_IDENTIFIER
        value: 0
        length: 1
        start: 'z'
    },
    Token{
        type: TOKEN_PLUS
        value: 0
        length: 1
        start: '+'
    },
    Token{
        type: TOKEN_IDENTIFIER
        value: 0
        length: 1
        start: 'y'
    },
    Token{
        type: TOKEN_COMMA
        value: 0
        length: 1
        start: ','
    },
    Token{
        type: TOKEN_LITERAL
        value: 1
        length: 1
        start: '1'
    },
    Token{
        type: TOKEN_RIGHT_PAREN
        value: 0
        length: 1
        start: ')'
    },
    Token{
        type: TOKEN_COMMA
        value: 0
        length: 1
        start: ','
```

```
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_LITERAL
            value: 1
            length: 1
            start: '1'
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
```

```
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_COMMA
            value: 0
            length: 1
            start: ','
        },
        Token{
            type: TOKEN_LITERAL
            value: 1
            length: 1
            start: '1'
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_COMMA
            value: 0
            length: 1
            start: ','
        },
        Token{
            type: TOKEN_LITERAL
            value: 1
            length: 1
            start: '1'
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_MINUS
            value: 0
```

```
            length: 1
            start: '-'
        },
        Token{
            type: TOKEN_IDENTIFIER
            value: 0
            length: 3
            start: 'qqq'
        },
        Token{
            type: TOKEN_STAR
            value: 0
            length: 1
            start: '*'
        },
        Token{
            type: TOKEN_NOT
            value: 0
            length: 3
            start: 'not'
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_NOT
            value: 0
            length: 3
            start: 'not'
        },
        Token{
            type: TOKEN_LEFT_PAREN
            value: 0
            length: 1
            start: '('
        },
        Token{
            type: TOKEN_LITERAL
            value: 10
            length: 2
            start: '10'
```

```
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_RIGHT_PAREN
            value: 0
            length: 1
            start: ')'
        },
        Token{
            type: TOKEN_EOF
            value: 0
            length: 0
            start: ''
        },
    }
}
```

1.3.6.2 parser.c

Generated using `print_syntax_tree` :

```
|__ OPERATOR(-)
-  |__ OPERATOR(r)
-  -  |__ OPERATOR(l)
-  -  -  |__ OPERATOR(l)
-  -  -  -  |__ OPERATOR(r)
-  -  -  -  -  |__ OPERATOR(|)
-  -  -  -  -  -  |__ OPERATOR(x)
-  -  -  -  -  -  -  |__ PARENTHESIS
-  -  -  -  -  -  -  -  |__ PARENTHESIS
-  -  -  -  -  -  -  -  -  -  |__ TOKEN("x")
-  -  -  -  -  -  -  -  |__ TOKEN("x")
-  -  -  -  -  -  -  |__ OPERATOR(+)
-  -  -  -  -  -  -  -  |__ TOKEN("z")
-  -  -  -  -  -  -  -  |__ TOKEN("y")
-  -  -  -  -  -  |__ TOKEN("1")
-  -  -  -  -  |__ PARENTHESIS
-  -  -  -  -  -  |__ PARENTHESIS
-  -  -  -  -  -  -  |__ PARENTHESIS
-  -  -  -  -  -  -  -  |__ TOKEN("1")
-  -  -  -  |__ TOKEN("1")
```

```
-  -  |__ TOKEN("1")
-  |__ OPERATOR(*)
-  -  |__ TOKEN("qqq")
-  -  |__ UNARY(NOT)
-  -  -  |__ UNARY(NOT)
-  -  -  -  |__ TOKEN("10")
```

2 FAQ

Q: Any improvements or extensions?

- It should be possible to create a template in the source code to parse & lex arbitrary operations. For this, we should make some updates to make precedence and associativity handling generic.

Q: How does the parser work?

- See LL(1) Grammars, and Recursive Descent Parser. This is basically how the parser works, we didn't reinvent the wheel.

Q: What difficulties did you encounter?

- Not many, we spend a couple of hours to fix a memory leak problem in parser.c, we used address sanitizer stack trace. Other than this, we didn't encounter any major technical difficulty.
- For me, the most difficult part was to wait for Atakan to push his task. Most probably, I am just very impatient. (Yusuf)

Q: How to get a pdf version of the README.md ?

- Easy,

```
pandoc --pdf-engine=xelatex \
    --highlight-style tango \
    -V colorlinks \
    -V urlcolor=NavyBlue \
    -V toccolor=red \
    --toc -N \
    -V geometry:margin=1in \
    -V fontsize:12pt \
    -s -o Documentation.pdf README.md
```