1 AdvCalc

Todo : add examples , using print for (dictionary, lexer, parser) Todo : proof read Todo : comment out parser.c, and add header to doc

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 intalled 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. Comments 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 an 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 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 ouput is a stream of tokens.

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

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;
```

```c
  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.

1.3.4 executor.c

```c
/*
 * This function executes the statetement given as a input string, and writes the output to
 * 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 string.
 * 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 seperate 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);

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