Project 1 - AdvCalc

CmpE 230, Systems Programming, Spring 2023

Fatih Furkan Bilsel 2021400024 & Semih Yılmaz 2020400171

Submission Date: 01/04/2023

1. Introduction

- Purpose: In this project, we implemented an interpreter for an advanced calculator using the C programming language. The advanced calculator (AdvCalc) accepts expressions and assignment statements. Possible expressions in the AdvCalc language are given in the table below.

a + b	Returns summation of a and b.
a * b	Returns multiplication of a and b.
a - b	Returns the subtraction of b from a.
a & b	Returns bitwise a and b.
a | b	Returns bitwise a or b.
xor(a, b)	Returns bitwise a xor b.
ls(a, i)	Returns the result of a shifted i bits to the left.
rs(a, i)	Returns the result of a shifted i bits to the right.
lr(a, i)	Returns the result of a rotated i times to the left.
rr(a, i)	Returns the result of a rotated i times to the right.
not(a)	Returns bitwise complement of a.

- Overview of solution: After taking input in string form, the program divides the input into lexemes. After that step, the program checks whether lexemes represent an assignment or not. If an assignment exists, then the program performs the assignment operation. If not, the program checks whether lexemes represent a valid expression. If lexemes stand for an expression, then it evaluates the result of the expression and prints to the console. In case given lexemes represent neither an assignment nor an expression, error message will be printed to the console.

2. Program Structure

a) Lexical Analysis

```
string = trim(string);
long nofTokens = findNofTokens(string);
char ** tokens = tokenize(string, nofTokens);
```

- char * trim(char *) => The function reduces adjacent spaces to single spaces and removes comments from the string.
- long findNofTokens(char *) => The function counts the number of tokens that will be created when the string is split to ensure allocating the necessary number of pointers. It works as it counts the number of delimiters such as -, +, /, *, ",".
- char ** tokenize(char *, long) => The function creates an array of strings using the number of tokens given from arguments, then separates the input string into individual tokens, and assign each token to its position.

b) Parsing & Interpreting

The Grammar rules used in the program can be written in BNF notation as below:

- <assignment> -> <identifier> "=" <expression>
- <expression> -> <bitwiseAnd> "|" <expression> | <bitwiseAnd>
- <bitwiseAnd> -> <summation> "&" <bitwiseAnd> | <summation>
- <summation> -> <multiplication> "+" <summation> | <multiplication> "-" <summation> | <multiplication>
- <multiplication> -> <term> "*" <multiplication> | <term>
- <term> -> "(" <expression> ")" | <factor>
- <factor> -> <function> | <integer> | <identifier>
- <function> -> "not" "(" <expression> ")" | "xor" "(" <expression> "," <expression> ")" | "ls" "(" <expression> "," <expression> ")" | "rs" "(" <expression> "," <expression> ")" | "lr" "(" <expression> "," <expression> ")" | "rr" "(" <expression> "," <expression> ")"
- <identifier> -> <alpha> <identifier> | <alpha>
- <integer> -> <digit> <integer> | <digit>
- <digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- <alpha> -> [a-z,A-Z]

There are two kinds of function written in the program for each rule of BNF. First type of functions return 1 if the given lexemes satisfy the corresponding BNF rule and otherwise return 0. Second type of functions are called if first type of functions return 1. (i.e., when the program assures corresponding BNF rule holds.) Their task is to evaluate expressions and return the results.

Example of first type:

```
long long isExpression(char** tokens, long long nofTokens){
    for(int i = 0; i<nofTokens;i++){
        if(!strcmp(tokens[i],"|")){
            // <expression> -> <bitwiseAnd> "|" <expression>
            if(isBitwiseAnd(tokens,i)&&isExpression(tokens+i+1,nofTokens-(i+1))){
                return 1;
            }
        }
    }
    // <expression> -> <bitwiseAnd>
    return isBitwiseAnd(tokens,nofTokens);
}
```

Example of second type:

```
long long evalExpression(char** tokens, long long nofTokens){
   for(int i = 0; i<nofTokens;i++){
     if(!strcmp(tokens[i],"|")){
        // <expression> -> <bitwiseAnd> "|" <expression>
        if(isBitwiseAnd(tokens,i)&&isExpression(tokens+i+1,nofTokens-(i+1))){
          return evalBitwiseAnd(tokensi|evalExpression(tokens+i+1,nofTokens-(i+1));
        }
     }
   }
   // <expression> -> <bitwiseAnd>
   if(isBitwiseAnd(tokens,nofTokens)){
       return evalBitwiseAnd(tokens, nofTokens);
   }
   printf("evalExpression: Error!\n");
   return 0;
}
```

The first argument of the functions is a pointer of array of strings. It points to the first lexeme which will be processed by the function. The second argument of the functions is an integer. It tells the functions how many lexemes they should process. The mechanism of the functions explained above is shown on the third section (Parser Mechanism) by tracing over examples.

c) Identifier/Variable Handling

We implemented a linked list data structure as shown below in order to keep variables.

```
struct variables {
    char * name;
    long value;
    struct variables * next;
};
struct variables * VARIABLES_HEAD = NULL;
```

Identifier handling operations are done inside assignment and identifier functions:

- long assignment(char**, long) => Parses the given tokens and assigns the result of expression to the identifier. To assign the result, first it checks whether the linked list is empty or not. If it is empty, it sets given identifier to the head of the list. Otherwise, it iterates over the linked list until either the end of the list or finding that given identifier has already in the list. If the identifier in the list, then it updates the value of the identifier. If not, the identifier is added to the end of the list.

- long identifier(char*) => The function checks whether the linked list is empty or not. If it is empty, the function returns 0. Otherwise, it iterates over the linked list until either the end of the list or finding that given identifier has already in the list. If the identifier in the list, then it returns the value of the identifier. If not, it returns 0.

3. Parser Mechanism

In the following examples, how the parser traces over functions is shown.

- 3+5

```
isAssignment: 3 + 5
        isIdentifier: 3
        false
false
isExpression: 3 + 5
        isBitwiseAnd: 3 + 5
                isSummation: 3 + 5
                        isMultiplication: 3
                                isTerm: 3
                                        isFactor: 3
                                                isInteger: 3
                                                true
                                        true
                                true
                        true
                        isSummation: 5
                                isMultiplication: 5
                                        isTerm: 5
```

isFactor: 5
isInteger: 5
true
true
true
true
true
true
true
true

- h = (rs(11, 2) | ls(4, 1)) * (6 - not(3))

(some middle steps are deleted here. To see the original version, please look at Appendices A.)

isAssignment: h = (rs (11 , 2) | ls (4 , 1)) * (6 - not (3))
isIdentifier: h
true
isExpression: (rs (11 , 2) | ls (4 , 1)) * (6 - not (3))
isBitwiseAnd: (rs (11 , 2)
isSummation: (rs (11 , 2)
isMultiplication: (rs (11 , 2)
isTerm: (rs (11 , 2)
isExpression: rs (11 , 2
(...)
isFunction: rs (11 , 2
false
(...)
false
isBitwiseAnd: (rs (11 , 2) | ls (4 , 1)) * (6 - not (3))
isSummation: (rs (11 , 2) | ls (4 , 1)) * (6 - not (3))
isMultiplication: (rs (11 , 2) | ls (4 , 1)) * (6
isTerm: (rs (11 , 2) | ls (4 , 1))
isExpression: rs (11 , 2) | ls (4 , 1)
isBitwiseAnd: rs (11 , 2)
(...)
isFunction: rs (11 , 2)
isExpression: 11
(...)
isInteger: 11
true
(...)
true
isExpression: 2
(...)
isInteger: 2
true
(...)
true
isExpression: ls (4 , 1)
(...)

```
                        isFunction: ls ( 4 , 1 )
                          isExpression: 4
                                  (...)
                                      isInteger: 4
                                      true
                                  (...)
                          true
                          isExpression: 1
                                  (...)
                                      isInteger: 1
                                      true
                        (...)
    true
    isMultiplication: ( 6
      isTerm: ( 6
        isFactor: ( 6
          isFunction: ( 6
          false
        false
      false
    false
    isTerm: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6
      isFactor: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6
        isFunction: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6
        false
      false
    false
false
isMultiplication: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6 - not ( 3 ) )
  isTerm: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) )
    isExpression: rs ( 11 , 2 ) | ls ( 4 , 1 )
      isBitwiseAnd: rs ( 11 , 2 )
                  (...)
                  isFunction: rs ( 11 , 2 )
                    isExpression: 11
                            (...)
                                isInteger: 11
                                true
                            (...)
                    true
                    isExpression: 2
                            (...)
                                isInteger: 2
                                true
                    (...)
      true
      isExpression: ls ( 4 , 1 )
                (...)
                    isFunction: ls ( 4 , 1 )
                      isExpression: 4
                        isBitwiseAnd: 4
                          isSummation: 4
```

```
                                        isMultiplication: 4
                                           isTerm: 4
                                              isFactor: 4
                                                 isInteger: 4
                                                 true
                                        (...)
                                  true
                                  isExpression: 1
                                     isBitwiseAnd: 1
                                        isSummation: 1
                                           isMultiplication: 1
                                              isTerm: 1
                                                 isFactor: 1
                                                    isInteger: 1
                                                    true
                                           (...)
                        true
                        isMultiplication: ( 6 - not ( 3 ) )
                           isTerm: ( 6 - not ( 3 ) )
                              isExpression: 6 - not ( 3 )
                                 isBitwiseAnd: 6 - not ( 3 )
                                    isSummation: 6 - not ( 3 )
                                       isMultiplication: 6
                                          isTerm: 6
                                             isFactor: 6
                                                isInteger: 6
                                                true
                                             true
                                          true
                                       true
                                       isSummation: not ( 3 )
                                          isMultiplication: not ( 3 )
                                             isTerm: not ( 3 )
                                                isFactor: not ( 3 )
                                                   isFunction: not ( 3 )
                                                      isExpression: 3
                                                         (...)
                                                                  isInteger: 3
                                                                  true
                  (....)
true
```

4. Input / Output Examples

```
% ./advcalc
> x = 5+8 & 55 | 74*3       % 223
> y = not(54) - rr(3,2)
> x
223
> y
4611686018427387849
> z = y - lr(5,3)
> z
4611686018427387809
> x y
Error!
> x + unknown     % Please note that undefined variables are 0.
223
> 45 * (21+9)
1350
> 55          55
Error!
>                          8888
8888
> lr(ls(rs(xor(((1)), 1) | 64 + 4, 1), (((1)))), 1)
136
> <Ctrl-D>
%
```

5. Difficulties Encountered

At the beginning we didn't think about the precedence of operators. Our code was not able to handle precedence correctly. To overcome the issue, we wrote down grammar rules in BNF notation.

Appendices A

```
> h = (rs(11, 2) | ls(4, 1)) * (6 - not(3))


isAssignment: h = ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6 - not ( 3 ) )
  isIdentifier: h
  true
  isExpression: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6 - not ( 3 ) )
    isBitwiseAnd: ( rs ( 11 , 2 )
      isSummation: ( rs ( 11 , 2 )
        isMultiplication: ( rs ( 11 , 2 )
          isTerm: ( rs ( 11 , 2 )
            isExpression: rs ( 11 , 2
              isBitwiseAnd: rs ( 11 , 2
                isSummation: rs ( 11 , 2
                  isMultiplication: rs ( 11 , 2
                    isTerm: rs ( 11 , 2
                      isFactor: rs ( 11 , 2
                        isFunction: rs ( 11 , 2
                        false
                      false
                    false
                  false
                false
              false
            false
          false
        false
      false
    false
    isBitwiseAnd: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6 - not ( 3 ) )
      isSummation: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6 - not ( 3 ) )
        isMultiplication: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6
          isTerm: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) )
            isExpression: rs ( 11 , 2 ) | ls ( 4 , 1 )
              isBitwiseAnd: rs ( 11 , 2 )
                isSummation: rs ( 11 , 2 )
                  isMultiplication: rs ( 11 , 2 )
                    isTerm: rs ( 11 , 2 )
                      isFactor: rs ( 11 , 2 )
                        isFunction: rs ( 11 , 2 )
                          isExpression: 11
                            isBitwiseAnd: 11
                              isSummation: 11
                                isMultiplication: 11
                                  isTerm: 11
                                    isFactor: 11
                                      isInteger: 11
                                      true
                                    true
```

```
                    true
                  true
                true
              true
            true
            isExpression: 2
              isBitwiseAnd: 2
                isSummation: 2
                  isMultiplication: 2
                    isTerm: 2
                      isFactor: 2
                        isInteger: 2
                        true
                      true
                    true
                  true
                true
              true
            true
          true
        true
      true
    true
  true
true
isExpression: ls ( 4 , 1 )
  isBitwiseAnd: ls ( 4 , 1 )
    isSummation: ls ( 4 , 1 )
      isMultiplication: ls ( 4 , 1 )
        isTerm: ls ( 4 , 1 )
          isFactor: ls ( 4 , 1 )
            isFunction: ls ( 4 , 1 )
              isExpression: 4
                isBitwiseAnd: 4
                  isSummation: 4
                    isMultiplication: 4
                      isTerm: 4
                        isFactor: 4
                          isInteger: 4
                          true
                        true
                      true
                    true
                  true
                true
              isExpression: 1
                isBitwiseAnd: 1
                  isSummation: 1
                    isMultiplication: 1
                      isTerm: 1
                        isFactor: 1
```

```
                                isInteger: 1
                                true
                              true
                            true
                          true
                        true
                      true
                    true
                  true
                true
              true
            true
          true
        true
      true
    true
  true
  isMultiplication: ( 6
    isTerm: ( 6
      isFactor: ( 6
        isFunction: ( 6
        false
      false
    false
  false
  isTerm: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6
    isFactor: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6
      isFunction: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6
      false
    false
  false
false
isMultiplication: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) ) * ( 6 - not ( 3 ) )
  isTerm: ( rs ( 11 , 2 ) | ls ( 4 , 1 ) )
    isExpression: rs ( 11 , 2 ) | ls ( 4 , 1 )
      isBitwiseAnd: rs ( 11 , 2 )
        isSummation: rs ( 11 , 2 )
          isMultiplication: rs ( 11 , 2 )
            isTerm: rs ( 11 , 2 )
              isFactor: rs ( 11 , 2 )
                isFunction: rs ( 11 , 2 )
                  isExpression: 11
                    isBitwiseAnd: 11
                      isSummation: 11
                        isMultiplication: 11
                          isTerm: 11
                            isFactor: 11
                              isInteger: 11
                              true
                            true
                          true
                        true
```

```
                true
              true
            true
            isExpression: 2
              isBitwiseAnd: 2
                isSummation: 2
                  isMultiplication: 2
                    isTerm: 2
                      isFactor: 2
                        isInteger: 2
                        true
                      true
                    true
                  true
                true
              true
            true
          true
        true
      true
    true
  true
true
isExpression: ls ( 4 , 1 )
  isBitwiseAnd: ls ( 4 , 1 )
    isSummation: ls ( 4 , 1 )
      isMultiplication: ls ( 4 , 1 )
        isTerm: ls ( 4 , 1 )
          isFactor: ls ( 4 , 1 )
            isFunction: ls ( 4 , 1 )
              isExpression: 4
                isBitwiseAnd: 4
                  isSummation: 4
                    isMultiplication: 4
                      isTerm: 4
                        isFactor: 4
                          isInteger: 4
                          true
                        true
                      true
                    true
                  true
                true
              isExpression: 1
                isBitwiseAnd: 1
                  isSummation: 1
                    isMultiplication: 1
                      isTerm: 1
                        isFactor: 1
                          isInteger: 1
                          true
```

```
                                true
                              true
                            true
                          true
                        true
                      true
                    true
                  true
                true
              true
            true
          true
        true
      true
    true
true
isMultiplication: ( 6 - not ( 3 ) )
  isTerm: ( 6 - not ( 3 ) )
    isExpression: 6 - not ( 3 )
      isBitwiseAnd: 6 - not ( 3 )
        isSummation: 6 - not ( 3 )
          isMultiplication: 6
            isTerm: 6
              isFactor: 6
                isInteger: 6
                true
              true
            true
          isSummation: not ( 3 )
            isMultiplication: not ( 3 )
              isTerm: not ( 3 )
                isFactor: not ( 3 )
                  isFunction: not ( 3 )
                    isExpression: 3
                      isBitwiseAnd: 3
                        isSummation: 3
                          isMultiplication: 3
                            isTerm: 3
                              isFactor: 3
                                isInteger: 3
                                true
                              true
                            true
                          true
                        true
                      true
                    true
                  true
                true
              true
            true
          true
```

true
true
true
true
true
true
true
true
true
true

Appendices B – Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// linked list data structure
struct variables {
    char * name;
    long long value;
    struct variables * next;
};
struct variables * VARIABLES_HEAD = NULL;

void run();
char * trim(char * );
long long findNofTokens(char *);
char ** tokenize(char *,long long);

// Grammar of the AdvCalc language is explained below in BNF notation

// <assignment> -> <identifier> "=" <expression>
long long isAssignment(char** tokens, long long nofTokens);
long long assignment(char** tokens, long long nofTokens);
// <expression> -> <bitwiseAnd> "|" <expression> | <bitwiseAnd>
long long isExpression(char** tokens, long long nofTokens);
long long evalExpression(char** tokens, long long nofTokens);
// <bitwiseAnd> -> <summation> "&" <bitwiseAnd> | <summation>
long long evalBitwiseAnd(char** tokens, long long nofTokens);
long long isBitwiseAnd(char** tokens, long long nofTokens);
// <summation> -> <multiplication> "+" <summation> | <multiplication> "-"
<summation> | <multiplication>
long long isSummation(char** tokens, long long nofTokens);
long long evalSummation(char** tokens, long long nofTokens);
// <multiplication> -> <term> "*" <evalMultiplication> | <term>
long long isMultiplication(char** tokens, long long nofTokens);
long long evalMultiplication(char** tokens, long long nofTokens);
// <term> -> "(" <expression> ")" | <factor>
long long isTerm(char** tokens, long long nofTokens);
long long evalTerm(char** tokens, long long nofTokens);
// <factor> -> <function> | <integer> | <identifier>
long long isFactor(char** tokens, long long nofTokens);
long long evalFactor(char** tokens, long long nofTokens);
// <function> ->       "not" "("  <expression>  ")" | "xor"  "("
<expression>   ","  <expression>   ")" |
// "ls"  "("  <expression>   ","  <expression>   ")" | "rs"  "("
<expression>   ","  <expression>   ")" |
// "lr"  "("  <expression>   ","  <expression>   ")" | "rr"  "("
<expression>   ","  <expression>   ")"
long long isFunction(char** tokens, long long nofTokens);
long long evalFunction(char** tokens, long long nofTokens);
// <identifier> -> <identifier> <alpha> | <alpha>
// <alpha> -> [a-z,A-Z]
long long isIdentifier(char* token);
long long identifier(char* token);
// <integer> ->  <integer> <digit> | <digit>
// <digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
long long isInteger(char* token);
long long integer(char* token);
```

```c
long long leftRotation(long long base, long long rotation);
long long rightRotation(long long base, long long rotation);


int main() {
    run();
    return 0;
}

void run() {
    char * string = malloc(1024 * sizeof(char));
    printf("> ");
    if ((fgets(string, 1024, stdin) == NULL)&&feof(stdin)){
        // ctrl-d executed
        printf("\n");
        return;
    }
    string = trim(string);
    // if the input is empty
    if (strlen(string) == 0) {
        free(string);
        run();
        return;
    }

    // returns the number of tokens given input has.
    long long nofTokens = findNofTokens(string);

    // lexical analyzing
    char ** tokens = tokenize(string, nofTokens);

    // parsing & interpreting
    if(isAssignment(tokens,nofTokens))
        assignment(tokens,nofTokens);
    else if(isExpression(tokens,nofTokens))
        printf("%lld\n", evalExpression(tokens, nofTokens));
    else
        printf("Error!\n");

    free(string);
    free(tokens);
    run();
}

// reduces adjacent spaces to single spaces and removes comments from the
string.
char * trim(char * string) {
    long long counter = 0, i = 0, flag = 0;
    // iterate over the string and count characters
    while (string[i] != '\0' && string[i] != '%') {
        if(string[i] == '\n') {
            break;
        }
        if (string[i] != ' ') {
            flag = 1;
            counter++;
        }
        else {
            // if previous character was non-whitespace
            if (flag) {
                counter++;
```

```
            }
            flag = 0;
        }
        i++;
    }
    char * newString = malloc(++counter * sizeof(char));
    counter = 0;
    i = 0;
    flag = 0;
    // iterates over the string
    while (string[i] != '\0' && string[i] != '\n' && string[i] != '%') {
        // end of the string
        if(string[i] == '\n' || string[i] == '%') {
            i++;
            break;
        }
        if (string[i] != ' ') {
            newString[counter++] = string[i];
            flag = 1;
        }
        else {
            // if previous character was non-whitespace
            if (flag) {
                newString[counter++] = string[i];
            }
            flag = 0;
        }
        i++;
    }
    newString[counter] = '\0';
    free(string);
    return newString;
}

// returns number of tokens given string has
// It works as it counts the number of delimiters such as -, +, /, *, ",".
long long findNofTokens(char * string) {
    long long counter = 0;
    long long i;
    int flag = 0;
    for (i = 0; i < strlen(string); i++) {
        char ch = string[i];
        // delimiters
        if (ch == '+' || ch == '*' || ch == '-' || ch == '&' || ch == '|'
|| ch == '=' ||  ch == ',' || ch == '(' || ch == ')') {
            counter++;
            if (flag) {
                counter++;
            }
            flag = 0;
        }
        // whitespaces
        else if(ch == ' ' || ch == '\t' || ch == '\n'){
            if (flag) {
                counter++;
            }
            flag = 0;
        }
        // comment indicator
        else if(ch == '%'){
            break;
```

```
        }
        else {
            flag = 1;
        }
    }
    if (flag) {
        counter++;
    }
    return counter;
}

// creates an array of strings using the number of tokens given from
arguments
// separates the input string into individual tokens, and assign each token
to its position in the array.
// then returns the array
char ** tokenize(char * string, long long nofTokens) {
    long long i, j, lastStart = 0, flag = 0, counter = 0;
    // allocating memory
    char ** tokens = malloc(nofTokens * sizeof(char *));
    for (i = 0; i < strlen(string); i++) {
        char ch = string[i];
        // delimiters
        if (ch == '+' || ch == '*' || ch == '-' || ch == '&' || ch == '|'
|| ch == ',' || ch == '=' || ch == '(' || ch == ')') {
            // if we were previously building up a token, add it to the
array
            if (flag) {
                flag = 0;
                tokens[counter] = malloc((i - lastStart + 1) *
sizeof(char));
                for (j = lastStart; j < i; j++) {
                    tokens[counter][j - lastStart] = string[j];
                }
                tokens[counter][i - lastStart] = '\0';
                counter++;
            }
            // add the delimiter character as a separate token
            tokens[counter] = malloc((2) * sizeof(char));
            tokens[counter][0] = ch;
            tokens[counter][1] = '\0';
            counter++;
        }
        // after '%' is comment
        else if(ch == '%'){
            // if we were previously building up a token, add it to the
array
            if (flag) {
                flag = 0;
                tokens[counter] = malloc((i - lastStart + 1) *
sizeof(char));
                for (j = lastStart; j < i; j++) {
                    tokens[counter][j - lastStart] = string[j];
                }
                tokens[counter][i - lastStart] = '\0';
                counter++;
            }
            return tokens;
        }
        else if(ch == ' ' || ch == '\t' || ch == '\n'){
            // if we were previously building up a token, add it to the
```

```
array
            if (flag) {
                flag = 0;
                tokens[counter] = malloc((i - lastStart + 1) *
sizeof(char));
                for (j = lastStart; j < i; j++) {
                    tokens[counter][(j - lastStart)] = string[j];
                }
                tokens[counter][i - lastStart] = '\0';
                counter++;
            }
        }
        else {
            if (!flag) {
                lastStart = i;
            }
            flag = 1;
        }
    }
    // if we were previously building up a token, add it to the array
    if (flag) {
        tokens[counter] = malloc((i - lastStart) * sizeof(char));
        for (j = lastStart; j < i; j++) {
            tokens[counter][j - lastStart] = string[j];
        }
        tokens[counter][i - lastStart] = '\0';
    }
    return tokens;
}

// <assignment> -> <identifier> "=" <expression>
long long isAssignment(char** tokens, long long nofTokens){
    return ( (nofTokens>=3) && (isIdentifier(tokens[0])) &&
(!strcmp(tokens[1],"=")) && (isExpression(tokens+2,nofTokens-2)) );
}

// parses the given tokens and assigns the result of expression to the
identifier.
long long assignment(char** tokens, long long nofTokens){
    long long res = evalExpression(tokens + 2, nofTokens - 2);
    struct variables * temp = VARIABLES_HEAD;
    // if linked list is empty, it sets given identifier to head of the
list
    if (temp == NULL) {
        temp = (struct variables *)malloc(sizeof(struct variables *));
        temp->name = (char *)malloc(sizeof(char));
        temp->name = tokens[0];
        temp->value = res;
        temp->next = NULL;
        VARIABLES_HEAD = temp;
    }
    // if linked list is not empty,
    else {
        // it iterates over the linked list until either the end of the
list
        // or finding that given identifier has already in the list.
        while (temp->next != NULL && strcmp(temp->name, tokens[0]) != 0) {
            temp = temp->next;
        }
        // if the identifier in the list, then it updates the value.
        if(!strcmp(temp->name, tokens[0])) {
```

```
            temp->value = res;
        }
        // if not, the identifier is added to the end of the list.
        else {
            temp->next = (struct variables *) malloc(sizeof(struct
variables *));
            temp = temp->next;
            temp->name = (char *)malloc(sizeof(char));
            temp->name = tokens[0];
            temp->value = res;
            temp->next = NULL;
        }
    }
}

// <expression> -> <bitwiseAnd> "|" <expression> | <bitwiseAnd>
long long isExpression(char** tokens, long long nofTokens){
    for(int i = 0; i<nofTokens;i++){
        if(!strcmp(tokens[i],"|")){
            if(isBitwiseAnd(tokens,i)&&isExpression(tokens+i+1,nofTokens-
(i+1))){
                return 1;
            }
        }
    }
    return isBitwiseAnd(tokens,nofTokens);
}

long long evalExpression(char** tokens, long long nofTokens){
    for(int i = 0; i<nofTokens;i++){
        if(!strcmp(tokens[i],"|")){
            if(isBitwiseAnd(tokens,i)&&isExpression(tokens+i+1,nofTokens-
(i+1))){
                return evalBitwiseAnd(tokens, i) | evalExpression(tokens +
i + 1, nofTokens - (i + 1));
            }
        }
    }
    if(isBitwiseAnd(tokens,nofTokens)){
        return evalBitwiseAnd(tokens, nofTokens);
    }
    printf("evalExpression: Error!\n");
    return 0;
}

// <bitwiseAnd> -> <summation> "&" <bitwiseAnd> | <summation>
long long isBitwiseAnd(char** tokens, long long nofTokens){
    for(int i = 0; i<nofTokens;i++){
        if(!strcmp(tokens[i],"&")){
            if(isSummation(tokens,i)&&isBitwiseAnd(tokens+i+1,nofTokens-
(i+1))){
                return 1;
            }
        }
    }
    return isSummation(tokens,nofTokens);
}

long long evalBitwiseAnd(char** tokens, long long nofTokens){
    for(int i = 0; i<nofTokens;i++){
        if(!strcmp(tokens[i],"&")){
```

```
            if(isSummation(tokens,i)&&isBitwiseAnd(tokens+i+1,nofTokens-
(i+1))){
                return evalSummation(tokens, i) & evalBitwiseAnd(tokens + i
+ 1, nofTokens - (i + 1));
            }
        }
    }
    if(isSummation(tokens,nofTokens)){
        return evalSummation(tokens, nofTokens);
    }
    printf("evalBitwiseAnd: Error!\n");
    return 0;
}

// <summation> -> <multiplication> "+" <summation> | <multiplication> "-"
<summation> | <multiplication>
long long isSummation(char** tokens, long long nofTokens){
    for(int i = 0; i<nofTokens;i++){
        if((!strcmp(tokens[i],"+"))||(!strcmp(tokens[i],"-"))){

if(isMultiplication(tokens,i)&&isSummation(tokens+i+1,nofTokens-(i+1))){
                return 1;
            }
        }
    }
    return isMultiplication(tokens,nofTokens);
}

long long evalSummation(char** tokens, long long nofTokens){
    for(int i = 0; i<nofTokens;i++){
        if(!strcmp(tokens[i],"+")){

if(isMultiplication(tokens,i)&&isSummation(tokens+i+1,nofTokens-(i+1))){
                return evalMultiplication(tokens, i) + evalSummation(tokens
+ i + 1, nofTokens - (i + 1));
            }
        }
        else if(!strcmp(tokens[i],"-")){

if(isMultiplication(tokens,i)&&isSummation(tokens+i+1,nofTokens-(i+1))){
                return evalMultiplication(tokens, i) - evalSummation(tokens
+ i + 1, nofTokens - (i + 1));
            }
        }
    }
    if(isMultiplication(tokens,nofTokens)){
        return evalMultiplication(tokens, nofTokens);
    }
    printf("evalSummation: Error!\n");
    return 0;
}

// <multiplication> -> <term> "*" <evalMultiplication> | <term>
long long isMultiplication(char** tokens, long long nofTokens){
    for(int i = 0; i<nofTokens;i++){
        if(!strcmp(tokens[i],"*")){
            if(isTerm(tokens,i)&&isMultiplication(tokens+i+1,nofTokens-
(i+1))){
                return 1;
            }
        }
```

```
    }
    return isTerm(tokens,nofTokens);
}

long long evalMultiplication(char** tokens, long long nofTokens){
    for(int i = 0; i<nofTokens;i++){
        if(!strcmp(tokens[i],"*")){
            if(isTerm(tokens,i)&&isMultiplication(tokens+i+1,nofTokens-
(i+1))){
                return evalTerm(tokens, i) * evalMultiplication(tokens + i
+ 1, nofTokens - (i + 1));
            }
        }
    }
    if(isTerm(tokens,nofTokens)){
        return evalTerm(tokens, nofTokens);
    }
    printf("evalMultiplication: Error!\n");
    return 0;
}

// <term> -> "(" <expression> ")" | <factor>
long long isTerm(char** tokens, long long nofTokens){
    if((!strcmp(tokens[0],"("))&&(!strcmp(tokens[nofTokens-1],")"))){
        if(isExpression(tokens+1,nofTokens-2)){
            return 1;
        }
    }
    else{
        return isFactor(tokens, nofTokens);
    }
}

long long evalTerm(char** tokens, long long nofTokens){
    if((!strcmp(tokens[0],"("))&&(!strcmp(tokens[nofTokens-1],")"))){
        if(isExpression(tokens+1,nofTokens-2)){
            return evalExpression(tokens + 1, nofTokens - 2);
        }
    }
    if (isFactor(tokens, nofTokens)){
        return evalFactor(tokens, nofTokens);
    }
    printf("evalTerm: Error!");
    return 0;
}

// <factor> -> <function> | <integer> | <identifier>
long long isFactor(char** tokens, long long nofTokens){
    if(nofTokens == 1){
        if(isInteger(tokens[0])){
            return 1;
        }
        if(isIdentifier(tokens[0])){
            return 1;
        }
        return 0;
    }
    return isFunction(tokens,nofTokens);
}

long long evalFactor(char** tokens, long long nofTokens){
```

```
    if(nofTokens == 1){
        if(isInteger(tokens[0])){
            return integer(tokens[0]);
        }
        if(isIdentifier(tokens[0])){
            return identifier(tokens[0]);
        }
        printf("evalFactor: Error!");
        return 0;
    }
    else if(isFunction(tokens, nofTokens)){
        return evalFunction(tokens, nofTokens);
    }
    printf("evalFactor: Error!");
    return 0;
}

// <function> ->       "not" "("  <expression>  ")" | "xor"  "("
<expression>   ","  <expression>   ")" |
// "ls"  "("  <expression>   ","  <expression>   ")" | "rs"  "("
<expression>   ","  <expression>   ")" |
// "lr"  "("  <expression>   ","  <expression>   ")" | "rr"  "("
<expression>   ","  <expression>   ")"
long long isFunction(char** tokens, long long nofTokens){
    // one argument function type (not)

if((!strcmp(tokens[0],"not"))&&(!strcmp(tokens[1],"("))&&(!strcmp(tokens[no
fTokens-1],")"))){
        if(isExpression(tokens+2,nofTokens-3)){
            return 1;
        }
    }
    // two argument function types (xor, lr, ls, rs, rr)
    else
if((!strcmp(tokens[0],"xor"))||(!strcmp(tokens[0],"ls"))||(!strcmp(tokens[0
],"rs"))||(!strcmp(tokens[0],"rr"))||(!strcmp(tokens[0],"lr"))){

if((nofTokens>=6)&&(!strcmp(tokens[1],"("))&&(!strcmp(tokens[nofTokens-
1],")"))) {
            for (int i = 3; i < nofTokens; i++) {
                if (!strcmp(tokens[i], ",")) {
                    if (isExpression(tokens + 2, i - 2) &&
isExpression(tokens + i + 1, nofTokens - (i + 2))) {
                        return 1;
                    }
                }
            }
        }
    }
    return 0;
}

long long evalFunction(char** tokens, long long nofTokens) {
    if((!strcmp(tokens[0],"not"))){
        return ~(evalExpression(tokens + 2, nofTokens - 3));
    }
    else
if((!strcmp(tokens[0],"xor"))||(!strcmp(tokens[0],"ls"))||(!strcmp(tokens[0
],"rs"))||(!strcmp(tokens[0],"rr"))||(!strcmp(tokens[0],"lr"))) {
        for (int i = 3; i < nofTokens; i++) {
            if (!strcmp(tokens[i], ",")) {
```

```
                if (isExpression(tokens + 2, i - 2) && isExpression(tokens
+ i + 1, nofTokens - (i + 2))) {
                    if (!strcmp(tokens[0], "xor"))
                        return evalExpression(tokens + 2, i - 2) ^
evalExpression(tokens + i + 1, nofTokens - (i + 2));
                    else if (!strcmp(tokens[0], "ls"))
                        return evalExpression(tokens + 2, i - 2) <<
evalExpression(tokens + i + 1, nofTokens - (i + 2));
                    else if (!strcmp(tokens[0], "rs"))
                        return evalExpression(tokens + 2, i - 2) >>
evalExpression(tokens + i + 1, nofTokens - (i + 2));
                    else if (!strcmp(tokens[0], "lr"))
                        return leftRotation(evalExpression(tokens + 2, i -
2),
                                                evalExpression(tokens + i + 1,
nofTokens - (i + 2)));
                    else if (!strcmp(tokens[0], "rr"))
                        return rightRotation(evalExpression(tokens + 2, i -
2),
                                                 evalExpression(tokens + i + 1,
nofTokens - (i + 2)));
                }
            }
        }
    }
    printf("evalFunction: Error!");
    return 0;
}

// <integer> ->  <integer> <digit> | <digit>
long long isInteger(char *token){
    int isnumber = 1 ;
    char *q ;

    for(q = token ; *q != '\0' ; q++) {
        isnumber = isnumber && isdigit(*q) ;
    }

    return(isnumber) ;
}

// converts given string to long long
long long integer(char* token){
    return atoll(token);
}

// <identifier> -> <identifier> <alpha> | <alpha>
long long isIdentifier(char *token){
    // reserved keywords

if((!strcmp(token,"xor"))||(!strcmp(token,"ls"))||(!strcmp(token,"rs"))||(!
strcmp(token,"rr"))||(!strcmp(token,"lr"))||(!strcmp(token,"not"))) {
        return 0;
    }
    int isWord = 1 ;
    char *q ;

    for(q = token ; *q != '\0' ; q++) {
        isWord = isWord && isalpha(*q) ;
    }
```

```
    return(isWord) ;
}

// returns the value of the given identifier if it is set previously.
// returns 0 if the given identifier is not set by any value.
long long identifier(char* token){
    struct variables * temp = VARIABLES_HEAD;
    // if the linked list which contains identifiers is empty
    if (temp == NULL) {
        return 0;
    }
    // if the linked list which contains identifiers is not empty
    else {
        // it iterates over the linked list until either the end of the
list
        // or finding that given identifier has already in the list.
        while (temp->next != NULL && strcmp(temp->name, token) != 0) {
            temp = temp->next;
        }
        // if the identifier in the list, then it returns the value of the
identifier.
        if(!strcmp(temp->name, token)) {
            return temp->value;
        }
        // if not, it returns 0.
        else {
            return 0;
        }
    }
}

// lr(base, rotation)
long long leftRotation(long long base, long long rotation) {
    rotation %= 64;
    long long temp = base;
    temp = temp << rotation;
    base = base >> (64 - rotation);
    return base + temp;
}

// rr(base, rotation)
long long rightRotation(long long base, long long rotation) {
    rotation %= 64;
    long long temp = base;
    temp = temp << (64 - rotation);
    base = base >> rotation;
    return base + temp;
}
```