Project 2 – Transcompiler

CmpE 230, Systems Programming, Spring 2023

Talha Ordukaya 2021400228 & Semih Yılmaz 2020400171

Submission Date: 02/05/2023

1. Introduction

- Purpose: In this project, we implemented a transpiler for an advanced calculator using the C programming language. It translates input in the form of assignment statements and expressions of the AdvCalc++ language into LLVM IR code that can compute and output those statements. 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 the quotient of the division.
a & b	Returns bitwise a and b.
a | b	Returns bitwise a or b.
a % b	Returns a modulo of 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> -> <expression> "|" <bitwiseAnd> | <bitwiseAnd>
- <bitwiseAnd> -> <bitwiseAnd> "&" <summation> | <summation>
- <summation> -> <summation> "+" <multiplication> | <summation> "-" <multiplication> | <multiplication>
- <multiplication> -> <multiplication> "*" <term> | <multiplication> "/" <term> | <multiplication> "%" <term> | <term>
- <term> -> "(" <expression> ")" | <factor>
- <factor> -> <function> | <integer> | <identifier>
- <function> -> "xor" "(" <expression> "," <expression> ")" | "not" "(" <expression> ")"
- <identifier> -> <alpha> <identifier> | <alpha>
- <alpha> -> [a-z,A-Z]
- <integer> -> <digit> <integer> | <digit>
- <digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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 call binary operation function and return the name of the register which holds the result of the operation.

Example of first type:

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

Example of second type:

```
char* evalExpression(char** tokens, int nofTokens){
    for(int i = nofTokens-1; i>=0 ;i--){
       if(!strcmp(tokens[i],"|")){
           if(isExpression(tokens,i)&&isBitwiseAnd(tokens+i+1,nofTokens-(i+1))){
              char* element1 = evalExpression(tokens, i);
              char* element2 = evalBitwiseAnd(tokens + i + 1, nofTokens - (i + 1));
              return binaryOperation("or",element1,element2);
          }
       }
    }
    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.

Binary Operation Function:

```
char* binaryOperation(char* operation, char* element1, char* element2){
    char regName[32];
    sprintf(regName, "%%reg%d", regCounter++);
    fprintf(output,"%s = %s i32 %s, %s\n", regName, operation, element1, element2);
    char* result = (char*) malloc(sizeof(char) * (strlen(regName) + 1));
    strcpy(result, regName);
    return result;
}
```

c) Identifier/Variable Handling

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

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

Identifier handling operations are done inside assignment function:

- long assignment(char**, long) => Parses the given tokens and assigns the result of expression to the register representing 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.

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. 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
                            true
                            isExpression: 1
                              isBitwiseAnd: 1
                                isSummation: 1
                                  isMultiplication: 1
                                    isTerm: 1
                                      isFactor: 1
                                        isInteger: 1
```

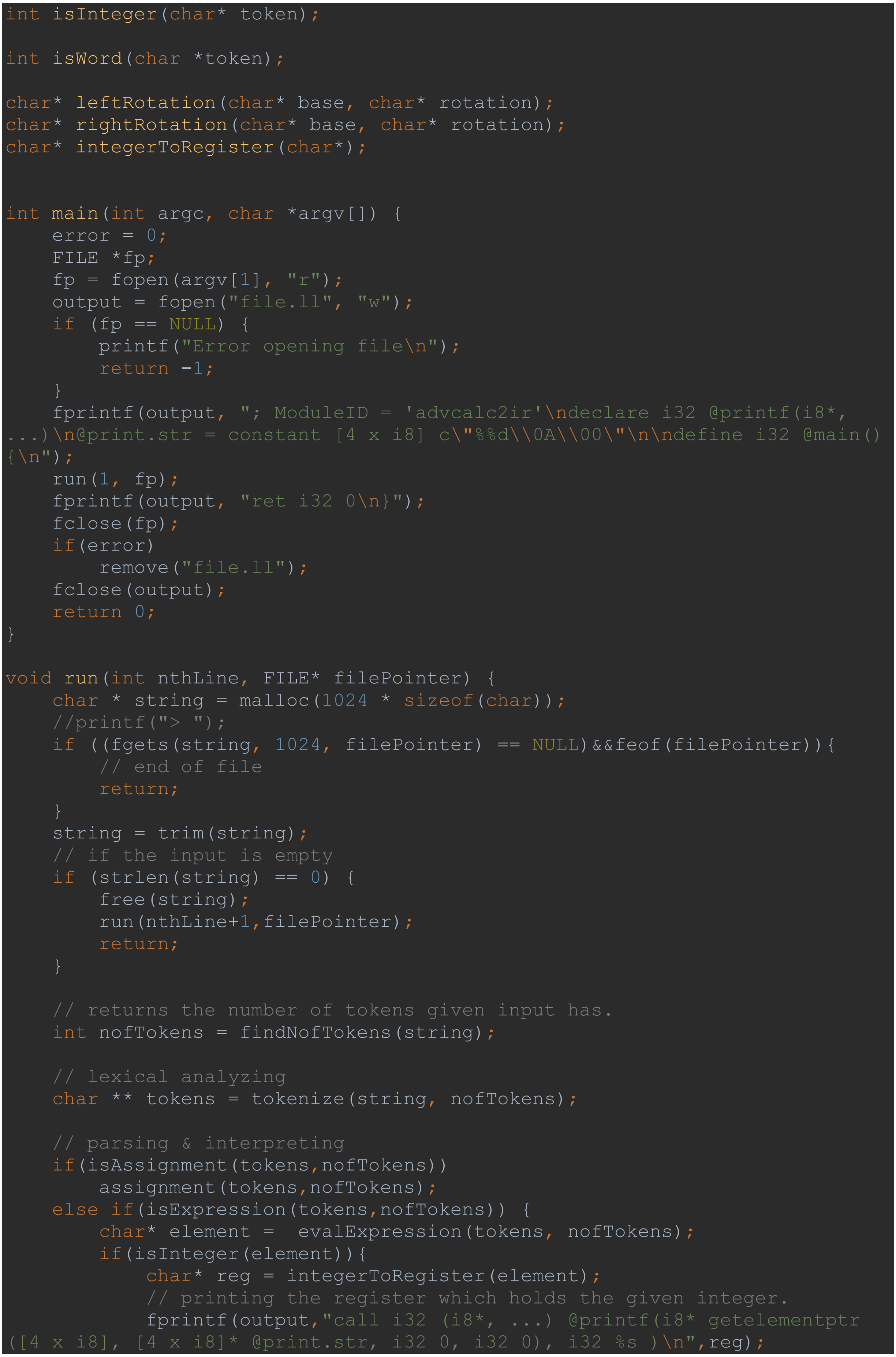
```
                                true
                              true
                            true
                          true
                        true
                      true
                    true
                  true
                true
              true
            true
          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
                    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 - 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
                      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;
    struct variables * next;
};
struct variables * VARIABLES_HEAD = NULL;

static int regCounter = 1;
static FILE* output;
static int error;

void run(int, FILE*);
char * trim(char * );
int findNofTokens(char *);
char ** tokenize(char *,int);

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

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

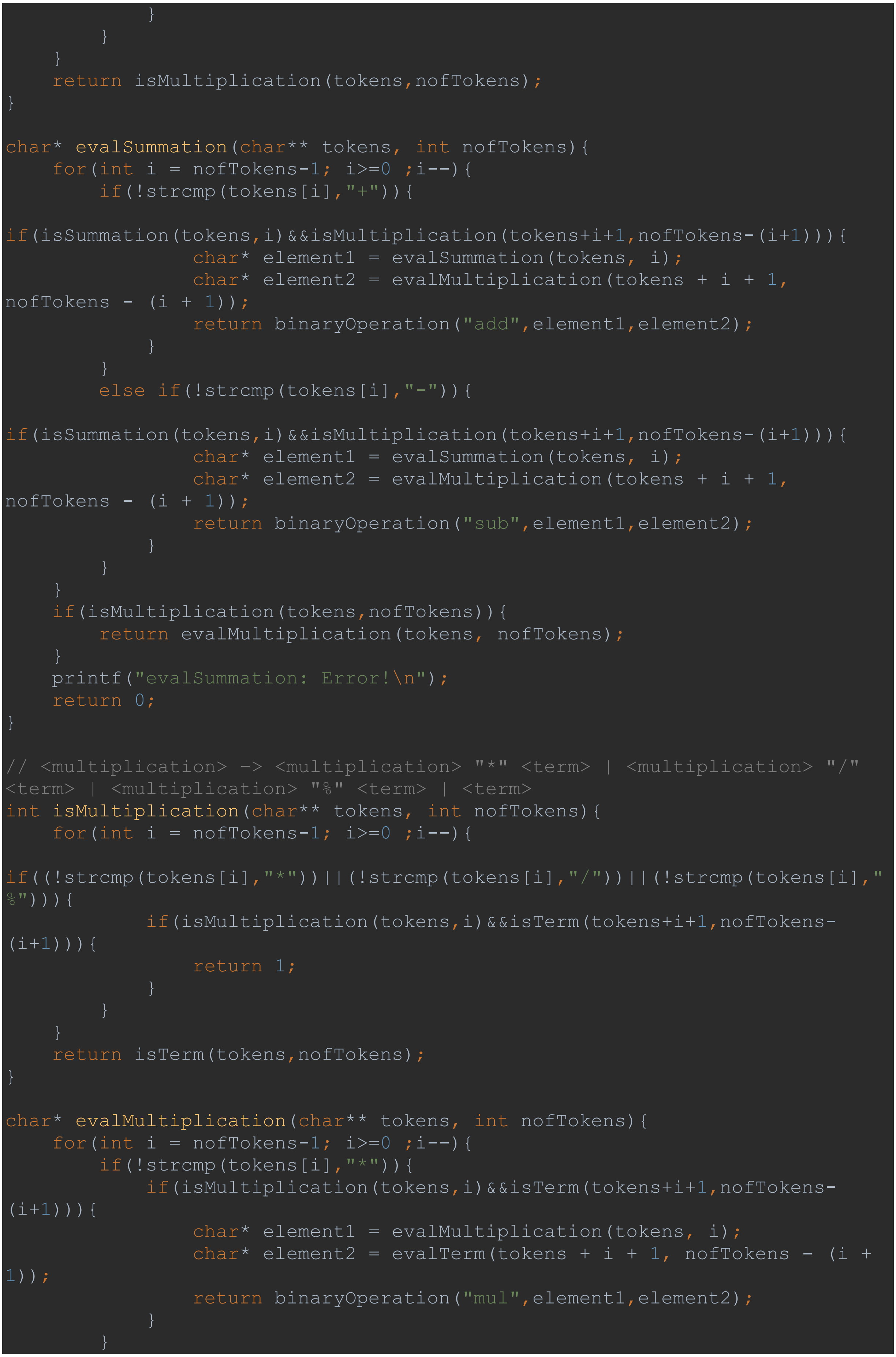


```
int isInteger(char* token);

int isWord(char *token);

char* leftRotation(char* base, char* rotation);
char* rightRotation(char* base, char* rotation);
char* integerToRegister(char*);


int main(int argc, char *argv[]) {
    error = 0;
    FILE *fp;
    fp = fopen(argv[1], "r");
    output = fopen("file.ll", "w");
    if (fp == NULL) {
        printf("Error opening file\n");
        return -1;
    }
    fprintf(output, "; ModuleID = 'advcalc2ir'\ndeclare i32 @printf(i8*,
...)\n@print.str = constant [4 x i8] c\"%%d\\0A\\00\"\n\ndefine i32 @main()
{\n");
    run(1, fp);
    fprintf(output, "ret i32 0\n}");
    fclose(fp);
    if(error)
        remove("file.ll");
    fclose(output);
    return 0;
}

void run(int nthLine, FILE* filePointer) {
    char * string = malloc(1024 * sizeof(char));
    //printf("> ");
    if ((fgets(string, 1024, filePointer) == NULL)&&feof(filePointer)){
        // end of file
        return;
    }
    string = trim(string);
    // if the input is empty
    if (strlen(string) == 0) {
        free(string);
        run(nthLine+1,filePointer);
        return;
    }

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

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

    // parsing & interpreting
    if(isAssignment(tokens,nofTokens))
        assignment(tokens,nofTokens);
    else if(isExpression(tokens,nofTokens)) {
        char* element =  evalExpression(tokens, nofTokens);
        if(isInteger(element)){
            char* reg = integerToRegister(element);
            // printing the register which holds the given integer.
            fprintf(output,"call i32 (i8*, ...) @printf(i8* getelementptr
([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %s )\n",reg);
```

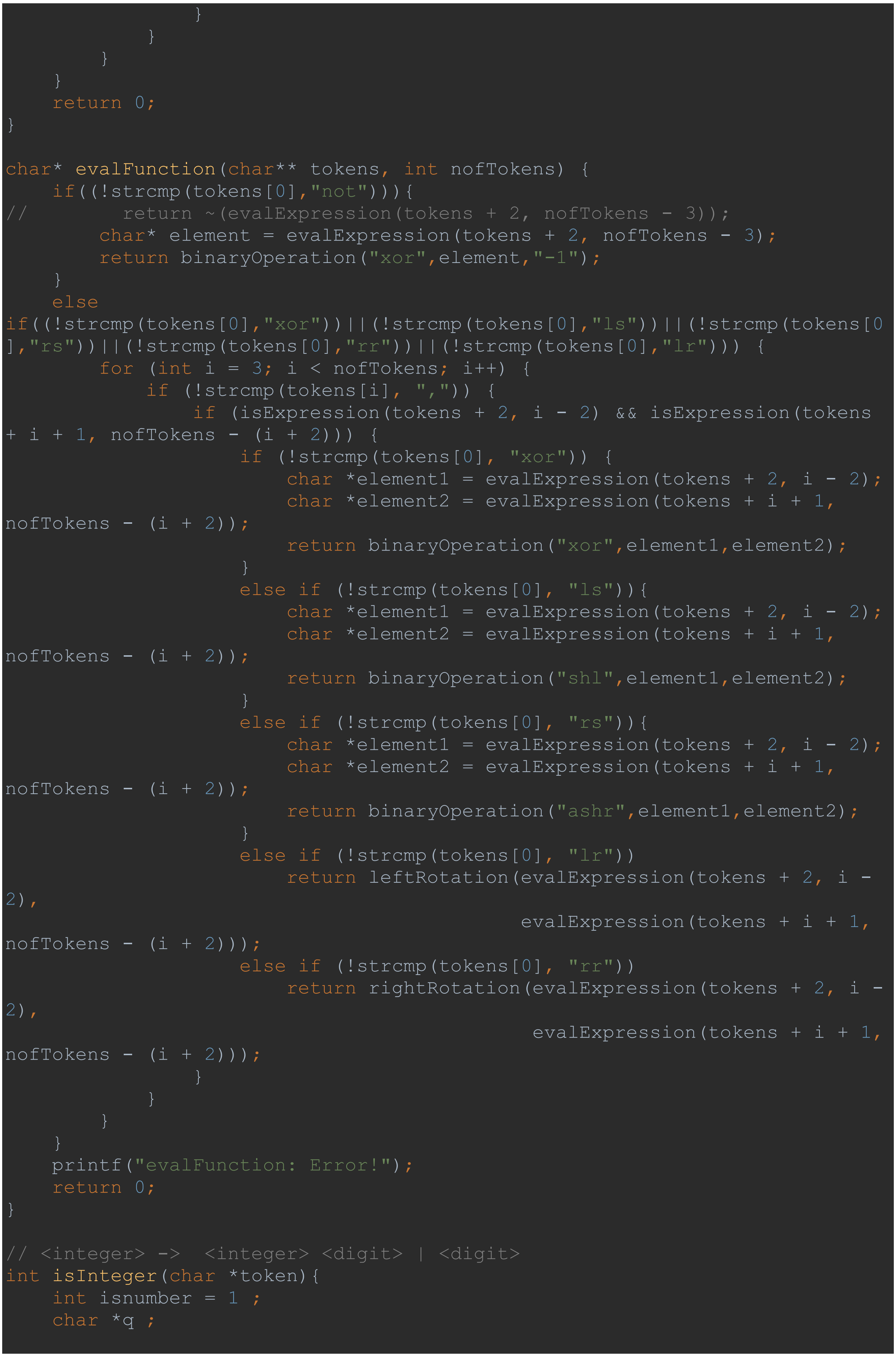
```
        }
        else{
            // printing the value of the register which holds the result of 
calculated expression.
            fprintf(output,"call i32 (i8*, ...) @printf(i8* getelementptr 
([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %s )\n",element);
        }
    }
    else{
        error = 1;
        printf("Error on line %d!\n",nthLine);
    }

    free(string);
    free(tokens);
    run(nthLine+1,filePointer);
}

// reduces adjacent spaces to single spaces and removes comments from the 
string.
char * trim(char * string) {
    int counter = 0, i = 0, flag = 0;
    // iterate over the string and count characters
    while (string[i] != '\0') {
        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') {
        // end of the string
        if(string[i] == '\n' ) {
            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 -, +, /, *, ",".
int findNofTokens(char * string) {
    int counter = 0;
    int 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 == ',' || ch == '(' || ch ==
')') {
            counter++;
            if (flag) {
                counter++;
            }
            flag = 0;
        }
            // whitespaces
        else if(ch == ' ' || ch == '\t' || ch == '\n'){
            if (flag) {
                counter++;
            }
            flag = 0;
        }
        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, int nofTokens) {
    int 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 == '=' || 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++;
        }
        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;
}

// creates a new register and assign the result of given binary operation
// then returns the name of the register.
char* binaryOperation(char* operation, char* element1, char* element2){
    char regName[32];
    sprintf(regName, "%%reg%d", regCounter++);
    fprintf(output,"%s = %s i32 %s, %s\n", regName, operation, element1, 
element2);
    char* result = (char*) malloc(sizeof(char) * (strlen(regName) + 1));
    strcpy(result, regName);
    return result;
}


// <assignment> -> <identifier> "=" <expression>
```

```
int isAssignment(char** tokens, int nofTokens){
    return ( (nofTokens>=3) && (isWord(tokens[0])) && (!strcmp(tokens[1],"=")) && (isExpression(tokens+2,nofTokens-2)) );
}

// parses the given tokens and assigns the result of expression to the identifier.
char* assignment(char** tokens, int nofTokens){
    char* 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->next = NULL;
        VARIABLES_HEAD = temp;
        // necessary IR operations for an assignment
        fprintf(output,"%%%s = alloca i32\n",tokens[0]);
        fprintf(output,"store i32 %s, i32* %%%s\n",res, tokens[0]);
    }
        // 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])) {
            // updating value in IR
            fprintf(output,"store i32 %s, i32* %%%s\n",res, tokens[0]);
        }
            // 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];
            // necessary IR operations for an assignment
            fprintf(output,"%%%s = alloca i32\n",tokens[0]);
            fprintf(output,"store i32 %s, i32* %%%s\n",res, tokens[0]);
            temp->next = NULL;
        }
    }
}

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

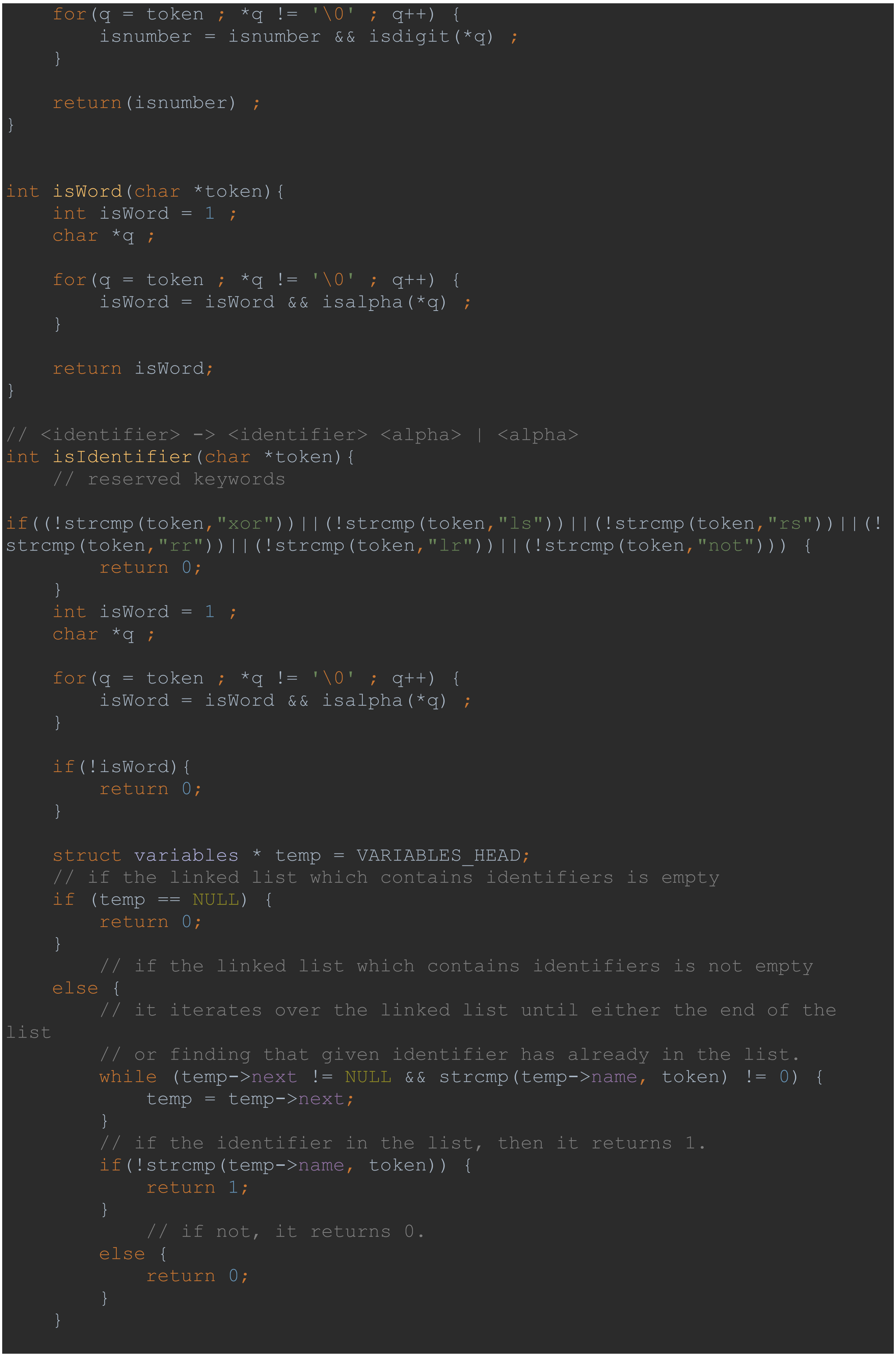
```
}

char* evalExpression(char** tokens, int nofTokens){
    for(int i = nofTokens-1; i>=0 ;i--){
        if(!strcmp(tokens[i],"|")){
            if(isExpression(tokens,i)&&isBitwiseAnd(tokens+i+1,nofTokens-
(i+1))){
                char* element1 = evalExpression(tokens, i);
                char* element2 = evalBitwiseAnd(tokens + i + 1, nofTokens -
(i + 1));
                return binaryOperation("or",element1,element2);
            }
        }
    }
    if(isBitwiseAnd(tokens,nofTokens)){
        return evalBitwiseAnd(tokens, nofTokens);
    }
    printf("evalExpression: Error!\n");
    return 0;
}

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

char* evalBitwiseAnd(char** tokens, int nofTokens){
    for(int i = nofTokens-1; i>=0 ;i--){
        if(!strcmp(tokens[i],"&")){
            if(isBitwiseAnd(tokens,i)&&isSummation(tokens+i+1,nofTokens-
(i+1))){
                char* element1 = evalBitwiseAnd(tokens, i);
                char* element2 = evalSummation(tokens + i + 1, nofTokens -
(i + 1));
                return binaryOperation("and",element1,element2);
            }
        }
    }
    if(isSummation(tokens,nofTokens)){
        return evalSummation(tokens, nofTokens);
    }
    printf("evalBitwiseAnd: Error!\n");
    return 0;
}

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

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



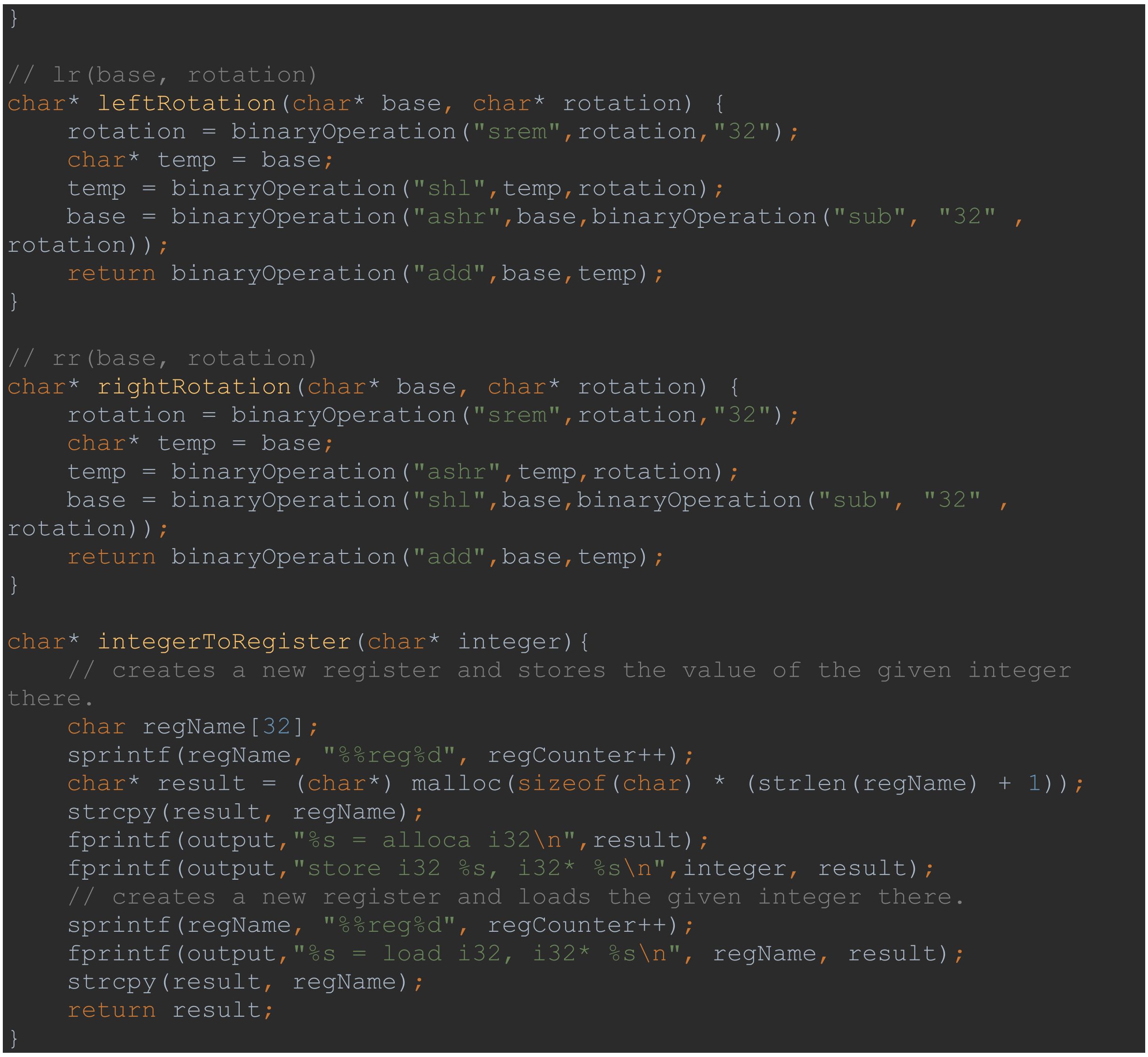
```
            }
        }
    }
    return isMultiplication(tokens,nofTokens);
}

char* evalSummation(char** tokens, int nofTokens){
    for(int i = nofTokens-1; i>=0 ;i--){
        if(!strcmp(tokens[i],"+")){

if(isSummation(tokens,i)&&isMultiplication(tokens+i+1,nofTokens-(i+1))){
                char* element1 = evalSummation(tokens, i);
                char* element2 = evalMultiplication(tokens + i + 1,
nofTokens - (i + 1));
                return binaryOperation("add",element1,element2);
            }
        }
        else if(!strcmp(tokens[i],"-")){

if(isSummation(tokens,i)&&isMultiplication(tokens+i+1,nofTokens-(i+1))){
                char* element1 = evalSummation(tokens, i);
                char* element2 = evalMultiplication(tokens + i + 1,
nofTokens - (i + 1));
                return binaryOperation("sub",element1,element2);
            }
        }
    }
    if(isMultiplication(tokens,nofTokens)){
        return evalMultiplication(tokens, nofTokens);
    }
    printf("evalSummation: Error!\n");
    return 0;
}

// <multiplication> -> <multiplication> "*" <term> | <multiplication> "/"
<term> | <multiplication> "%" <term> | <term>
int isMultiplication(char** tokens, int nofTokens){
    for(int i = nofTokens-1; i>=0 ;i--){

if((!strcmp(tokens[i],"*"))||(!strcmp(tokens[i],"/"))||(!strcmp(tokens[i],"
%"))){
            if(isMultiplication(tokens,i)&&isTerm(tokens+i+1,nofTokens-
(i+1))){
                return 1;
            }
        }
    }
    return isTerm(tokens,nofTokens);
}

char* evalMultiplication(char** tokens, int nofTokens){
    for(int i = nofTokens-1; i>=0 ;i--){
        if(!strcmp(tokens[i],"*")){
            if(isMultiplication(tokens,i)&&isTerm(tokens+i+1,nofTokens-
(i+1))){
                char* element1 = evalMultiplication(tokens, i);
                char* element2 = evalTerm(tokens + i + 1, nofTokens - (i +
1));
                return binaryOperation("mul",element1,element2);
            }
        }
        else if(!strcmp(tokens[i],"/")){
            if(isMultiplication(tokens,i)&&isTerm(tokens+i+1,nofTokens-
(i+1))){
                char* element1 = evalMultiplication(tokens, i);
                char* element2 = evalTerm(tokens + i + 1, nofTokens - (i +
1));
                return binaryOperation("sdiv",element1,element2);
            }
        }
        else if(!strcmp(tokens[i],"%")){
            if(isMultiplication(tokens,i)&&isTerm(tokens+i+1,nofTokens-
(i+1))){
                char* element1 = evalMultiplication(tokens, i);
                char* element2 = evalTerm(tokens + i + 1, nofTokens - (i +
1));
                return binaryOperation("srem",element1,element2);
            }
        }
    }
    if(isTerm(tokens,nofTokens)){
        return evalTerm(tokens, nofTokens);
    }
    printf("evalMultiplication: Error!\n");
    return 0;
}

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

char* evalTerm(char** tokens, int 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>
int isFactor(char** tokens, int nofTokens){
    if(nofTokens == 1){
        if(isInteger(tokens[0])){
            return 1;
        }
        if(isIdentifier(tokens[0])){
            return 1;
        }
        return 0;
    }
    return isFunction(tokens,nofTokens);
}

char* evalFactor(char** tokens, int nofTokens) {
    if (nofTokens == 1) {
        if (isInteger(tokens[0])) {
            char* result = (char*) malloc(sizeof(char) * (strlen(tokens[0])
+ 1));
            strcpy(result, tokens[0]);
            return result;
        }
        if (isIdentifier(tokens[0])) {
            char regName[32];
            sprintf(regName, "%%reg%d", regCounter++);
            fprintf(output,"%s = load i32, i32* %%%s\n", regName,
tokens[0]);
            char* result = (char*) malloc(sizeof(char) * (strlen(regName) +
1));
            strcpy(result, regName);
            return result;
        }
        printf("Error: Invalid factor token\n");
        return NULL;
    }
    else if (isFunction(tokens, nofTokens)) {
        return evalFunction(tokens, nofTokens);
    }
    printf("Error: Invalid factor expression\n");
    return NULL;
}


// <function> ->       "not" "("  <expression>  ")" | "xor"  "("
<expression>   ","  <expression>   ")" |
// "ls"  "("  <expression>   ","  <expression>   ")" | "rs"  "("
<expression>   ","  <expression>   ")" |
// "lr"  "("  <expression>   ","  <expression>   ")" | "rr"  "("
<expression>   ","  <expression>   ")"
int isFunction(char** tokens, int 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;
}

char* evalFunction(char** tokens, int nofTokens) {
    if((!strcmp(tokens[0],"not"))){
//        return ~(evalExpression(tokens + 2, nofTokens - 3));
        char* element = evalExpression(tokens + 2, nofTokens - 3);
        return binaryOperation("xor",element,"-1");
    }
    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")) {
                        char *element1 = evalExpression(tokens + 2, i - 2);
                        char *element2 = evalExpression(tokens + i + 1,
nofTokens - (i + 2));
                        return binaryOperation("xor",element1,element2);
                    }
                    else if (!strcmp(tokens[0], "ls")){
                        char *element1 = evalExpression(tokens + 2, i - 2);
                        char *element2 = evalExpression(tokens + i + 1,
nofTokens - (i + 2));
                        return binaryOperation("shl",element1,element2);
                    }
                    else if (!strcmp(tokens[0], "rs")){
                        char *element1 = evalExpression(tokens + 2, i - 2);
                        char *element2 = evalExpression(tokens + i + 1,
nofTokens - (i + 2));
                        return binaryOperation("ashr",element1,element2);
                    }
                    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>
int isInteger(char *token){
    int isnumber = 1 ;
    char *q ;
```



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

    return(isnumber) ;
}


int isWord(char *token){
    int isWord = 1 ;
    char *q ;

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

    return isWord;
}

// <identifier> -> <identifier> <alpha> | <alpha>
int 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) ;
    }

    if(!isWord){
        return 0;
    }

    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 1.
        if(!strcmp(temp->name, token)) {
            return 1;
        }
            // if not, it returns 0.
        else {
            return 0;
        }
    }
```



```
}

// lr(base, rotation)
char* leftRotation(char* base, char* rotation) {
    rotation = binaryOperation("srem",rotation,"32");
    char* temp = base;
    temp = binaryOperation("shl",temp,rotation);
    base = binaryOperation("ashr",base,binaryOperation("sub", "32" ,
rotation));
    return binaryOperation("add",base,temp);
}

// rr(base, rotation)
char* rightRotation(char* base, char* rotation) {
    rotation = binaryOperation("srem",rotation,"32");
    char* temp = base;
    temp = binaryOperation("ashr",temp,rotation);
    base = binaryOperation("shl",base,binaryOperation("sub", "32" ,
rotation));
    return binaryOperation("add",base,temp);
}

char* integerToRegister(char* integer){
    // creates a new register and stores the value of the given integer
there.
    char regName[32];
    sprintf(regName, "%%reg%d", regCounter++);
    char* result = (char*) malloc(sizeof(char) * (strlen(regName) + 1));
    strcpy(result, regName);
    fprintf(output,"%s = alloca i32\n",result);
    fprintf(output,"store i32 %s, i32* %s\n",integer, result);
    // creates a new register and loads the given integer there.
    sprintf(regName, "%%reg%d", regCounter++);
    fprintf(output,"%s = load i32, i32* %s\n", regName, result);
    strcpy(result, regName);
    return result;
}
```