CMPE230 – SYSTEMS PROGRAMMING

Name: Cem Güngör

Project title: advCalc

Project type: programming project

Submission date: 30/03/2023

Introduction

Problem: Implementing an interpreter for advanced calculator using C programming. Calculator will take expressions and assignments. Variables can be recalled during new assignments and expressions. Expressions include summation, multiplication, subtraction, bitwise and, or, xor, bitwise shift to both left and right, bitwise rotate to both left and right and bitwise not. Values will be 64-bit integer valued and none of the expressions or assignment will exceed that limit. Unary minus is not supported. Variable names consist of only alphabetical characters, and cannot be equal to function names. Inputs are at most 256 characters and “>” should be printed before each input, and undefined variables have 0 value. If assignment is given as input no printing, if expression is given as input interpreter should print the result. All errors are possible, interpreter should print “Error!” in such case. And for the last two rules, % denotes comments, and ctrl-d combination exites the code.

Solution: Using for loop checked each character, if there’s an error exited the loop and requested another input. While reading character by character, pushed every operator(functions, parenthesis and basic operators) and operands(variable values and integers) into two separate stacks and using operator priorities throught the application, calculated the expressions values. Precedence is checked while pushing the latest operator, if it has lower or equal priority than the operator on top, then calculated value until the top operator has a lower precedence.

Terminology:

Variable: a struct data type that has a name and value(long long int, to keep 64-bit numbers) to keep a define a variable that is input.

Stack: a struct data type that is used just as LIFO stack, it has three separate stacks and three separate ‘top’ named integers to keep track of top index. One stack for operands, one for operators and one for paranthesis check.

SyntaxCheck: a char array that keeps the track of each encountered meaningful parts. For example variables, operators, functions… Used v for variables, i for integers, o for operators, f for functions, and “,”,”(“,”)” for comma and opening and closing parenthesis. It will be used to check syntax errors.

Interface:

The advanced Calculator interpreter can be started using make command in root folder and then running the "advcalc.exe". The program will print ">" to indicate an input prompt, and while combination of ctrl-d is not pressed it will continue to prompt for input. Inputs can have two kinds: expression and assignment. The execution of the program doesn't include any parameters.

Starting the program: make command and advcalc.exe in the same folder.

Ending the program: pressing ctrl-d simultaneously.

Execution:

This section covers the the type of inputs, explanation of inputs, and what is an error for the program.

Input Types: Three types of inputs can be given to the program. First one is just an expression, second one is an assignment and the last one is to stop running**(ctrl-d).**

Expression: an expression is a mix of operations and values(integers,variables) that might or might not include functions(xor,ls,rs,lr,rr,not), basic operations (+,*,-) and bitwise operations (&,|) which are written syntatically correct and enough parameters are provided for each function used. Every expression must have a variable or integer, but it might not include any operators or functions.

Assignment: an assignment is basically a variable name equalized to an expression. Variable names can't be inside paranthesis, non-alphabetical characters cannot be used, and its case-sensitive. Moreover, function names can't be used as variable names, and variable names having a blank character in between cannot be used.

Errors: errors for the inputs can have various types. **Naming errors** include variable names being just only a function name, including a nonalphabetical character, trying to create a variable having a name which has more than one word. **Unary minus errors** include using unary minus in the program, for example: 3 + -5 or 3 + (-5). **Parenthesis errors** include having unmatched opening and closing parenthesis, a function not having parenthesis to take input parameters. **Function errors** include functions not having enough parameters, or having more than required parameters, trying to call shifting and rotating functions giving negative second parameter. **Operator errors** include having two operators repeated without anything inbetween…

Input and Output

Input format: as explained in prior sections there are mainly two types of inputs, one for expression, one for assignment.

Expression format: an input having a return value, must include a variable or integer, might include functions and operators, that cannot be greater than 256 characters, has matching parenthesis and has no unary minus, might have any number or spaces anywhere, calls functions using the right amount of parameters and obeys the main rules:

1. Can only start with a function, parenthesis, variable, or integer
2. An opening paranthesis must follow a function call
3. A closing parenthesis, operator or comma must follow an integer or variable
4. A closing parenthesis, operator or comma must follow closing parenthesis
5. Anything except operator, comma or closing parenthesis can follow an opening parenthesis
6. Both comma and operators must be followed by a variable or integer or function or '('

Functions xor, ls(left-shift), rs(right shift), rr(right rotate), and lr(left rotate) takes two input parameters and shifts and rotates second input cannot be negative. Not function takes only one parameter. These parameters can be expressions itself, as well as integers and variables also.

Assignment format: assignments have the format string(lhs) = string(rhs), rhs must be an expression and lhs must be a variable name. Lhs can't contain any non-alphabetical character, can't be just a function name, can only be one word. Lhs might have preceding or following blankspaces in it the program trims both of them.

Examples of inputs and outputs: input is indicated as i: and output is indicated as o: and explanation e:

1. **i:** 3 + 5 * (5 & 5) – 15 + x **o:** 13 **e:** program first adds 3 to operand stack and + to operator stack, then 5 to operand and * to operator stack, when it encounters a '(' pushes that also to operator stack, then pushes 5,&,5 to operator and operand stacks, and when it encounters ')' applies operators till (is on top of the stack, 5&5 = 5 in bit-wise operations. Then when – checks operator priority, since – has lower precedence than *, does * first: 5 * 5 = 25 then adds -,15 to operator and operand stack, at last when + is reached since it has equal priority with – and other + does those first: 25-15 = 10 10 + 3 = 13, to finish up adds 13 to x, which is still 13 because undefined variables have value 0.
2. **i:** x = xor(1,(not(-5)) + y + 5 **o: Error!** **e:** left hand side has no problem but while looping the right hand side, not function has parameter with unary minus which is forbidden so error!
3. **i:** x = xor(1,not(1-6)) + y + 5 **o:** **e:** no output since its just an assignment, and theres no error in the input
4. **i:** x **o:** 10 **e:** not(1-6) = not(-5) = 4, xor(1,4) = 5 and 5 + y + 5 is 10

Program Structure:

This program loops the input char by char, and creates stacks to evaluate expressions. While creating stacks it checks for all kinds of errors that can occur, and if an error occurs it prints out Error! The input it basically split into 7: operators, functions, variables, integers, parenthesis, commas, blanks and each of them is treated according to the rules.

Data Structures:

Variable: a struct data type to create a variable, has two parts, one: char array to define and keep the name of the variable, and two, long long int to keep 64 bit integers value.

Stack: a struct data type that is just a LIFO stack. Has three stacks in it, one for operands one for operators and one for parenthesis. Operands and operators are for the calculations whereas parenthesis is to check the parenthesis count in the given input.

syntaxCheck: A character array to keep track of syntax, while looping the input, the array is filled with operations, parenthesis, commas, variables and integers. Each element has its own char, the first letter of everyting except for comma:, and parenthesis:'(' and ')'. Later in the code it is used to check syntax.

unaryCheck: An integer that is used as a boolean, to check if two operators are used in a row.

errorCheck: An integer that is used as a boolean, changed to -1 when any error occurs so that for loop exits.

Functions:

Pop and push: All pop and push functions: (operator,parenthesis,operand) either returns the top element of the given stack and decrements index value by 1: **pop** or increments index value by 1 and puts new value on top of the stack.

getOperatorPriority: is a function that returns operator priority, higher the returned number higher the priority. Is used in the main logic of the evaluate

isOperator: a function to check if given char is an operator, just includes +,*,-,&,| since functions(xor,not…) are char arrays and should be checked whether they are functions or variables. This function returns 0 and changes errorCheck to -1 if unarycheck is -1, which means the last checked element was already an operator.

varNameCheck: a function that first trims the given input, then checks if there exist a space or a non-alphabetical character in the trimmed string. If not creates a varname and checks if it matches with one of the function names, if so prints an error, else evaluates the rhs of the expression and if no error is found in the rhs only then checks if that varname exists. If it exists then changes the value, if not creates new var.

findVar: a simple loop to check if given string is a variable name, if it is returns index

evaluate: function to evaluate the given expression, loops char by char, skips whitespace, other than whitespace checks the syntax each time an element occurs.

If inp[i] is , since , is used to separate a functions parameters, it calculates value till opening parenthesis is found, if inp[i] is) then first checks parenthesis stack, if not a problem checks for the syntax and then applies the operators and operands till (is reached.
If inp[i] is digit, first typecast from char to long long int, then while digit since we use base10, we equalize operand to operand * 10 (to shift the digit and add the last found digit.
If inp[i] is operator then if the last operator in the stack has higher or equal priority, to don't lose operator priority it applies operator and operand. And changes unaryCheck to -1 since an operator is found.

If inp[i] is alphabetical, extract till its not, then check the string if its variable or function, do the operations accordingly

If none of these is true then there's a mistake, errorcheck = -1

After the loop, check for parenthesis, if no problem do the remaining operations, if there's still no error then return the last operand which is the result.

applyOperator: applies the last operator in the operator stack. If operator is N, since not takes 1 argument, second operand is not popped, else its popped.

funcNameCheck: a function that checks if the given parameter: var is a function or variable, if function return – if variable return its value.

syntaxChecker: a function that checks the syntax using syntaxChecker array. It first checkes the first element since it cant start with comma or closing paranthesis etc. then checks elements two by two, for example if a function is preceded by a function changes error check to -1.

Examples

Input: xor(35,689) + 3 + 5 * 2 + (2 + (5 + not(3) + not(not(5-7))) + 2)

Output: 674→ xor's 35 and 689 then multiply 5 * 2 then starting from the most inner parenthesis not's -2, again not's 1 then adds first 5 then 2 then another 2. At last adds all together

Input: x = y * 5 + 1888 * not(1) + 5

Output: → equalizes x to y*5 which is 0 since y is undefined + 1888*-2 since not(1) is -2 + 5 therefore -3771

Input: x

Output: -3771

Input: y = x++

Output: Error! → since ++ is not supported

Input: y = x + not(x) + 1

Input: y

Output: 0

Improvements and Extensions

One improvement might be checking the function calls syntax. This code doesn't take into consideration if a function is called like xor((a,b)) which is an error. Maybe squareroot or powers can be added to the calculator since they are more basic functions compared to bitwise operations. Some parts of the code might be over used, those parts could be improved to use less. I first tried to implement the project using the grammar given, but I couldn't manage it this way.

Difficulties Encountered

I spent majority of my time to find an applicable solution since I couldn't manage to use the grammar. Then the next difficulty was to getting familiar with c programming language and its basics, such as char arrays, pointers etc. It was hard to even understand the indexing of the char arrays and finish element= '\0'. Moreover, while coding and trying the code I encountered new errors, and had to add unplanned features and variables to solve the problem.

Appendices:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define maxLen 256
typedef struct
{
    char name[maxLen];
    long long int value;
} Variable;
typedef struct
{
    int operandTop;
    int operationTop;
    int parenthesisTop;
    long long int operands[maxLen];
    char parenthesis[maxLen];
    char operators[maxLen];
} Stack;
```

```
Variable variableList[128];
int varCount = 0;
Stack operation;
static const Stack EmptyStack;
char syntaxCheck[128];
int syntaxLength;
int unaryCheck;
int errorCheck;
char popOperator(Stack *stack);
char popParenthesis(Stack *stack);
int getOperatorPriority(char op);
int isOperator(char op);
int varNameCheck(char *, char *);
int findVar(char *);
long long int evaluate(char *);
long long int applyOperator(Stack *stack);
long long int popOperand(Stack *stack);
long long int funcNameCheck(char *);
void pushOperand(Stack *stack, long long int operand);
void pushOperator(Stack *stack, char operator);
void pushParenthesis(Stack *stack);
void syntaxChecker();
int main()
{
    printf("> ");
    char inp[maxLen];
    while (fgets(inp, maxLen, stdin) != NULL)
    {
        operation = EmptyStack;
        operation.operandTop = -1;
        operation.operationTop = -1;
        operation.parenthesisTop = -1;
        unaryCheck = -1;
        errorCheck = 0;
        syntaxCheck[0] = '\0';
        syntaxLength = 0;
        inp[strcspn(inp, "\n")] = '\0';
        if (strchr(inp, '%') != NULL)
        {
            char temp[maxLen];
            char *pos = strchr(inp, '%');
            strncpy(temp, inp, pos - inp);
            temp[pos - inp] = '\0';
            strcpy(inp, temp);
        }
```

```
        if (strcmp(inp, "") == 0)
        {
            printf("> ");
            continue;
        }
        else if (strchr(inp, '=') == NULL)
        {
            long long int res = evaluate(inp);
            if (syntaxLength == 0 && errorCheck != -1)
            {
                printf("> ");
                continue;
            }
            else if (syntaxCheck[syntaxLength - 1] != 'v' && syntaxCheck[syntaxLength - 1] != 'i' && syntaxCheck[syntaxLength - 1] != ')')
            {
                errorCheck = -1;
            }
            if (errorCheck != -1)
            {
                printf("%lld\n", res);
            }
            else
            {
                printf("Error!\n");
            }
        }
        else
        {
            char lhs[maxLen];
            char rhs[maxLen];
            char *pos = strchr(inp, '=');
            strncpy(lhs, inp, pos - inp);
            lhs[pos - inp] = '\0';
            strcpy(rhs, pos + 1);
            rhs[strcspn(rhs, "\n")] = '\0';
            varNameCheck(lhs, rhs);
        }
        printf("> ");
    }
    return 0;
}
char popOperator(Stack *stack)
{
    return stack->operators[stack->operationTop--];
}
```

```
char popParenthesis(Stack *stack)
{
    return stack->parenthesis[stack->parenthesisTop--];
}
int getOperatorPriority(char op)
{
    if (op == '|')
    {
        return 1;
    }
    else if (op == '&')
    {
        return 2;
    }
    else if (op == '+' || op == '-')
    {
        return 3;
    }
    else if (op == '*')
    {
        return 4;
    }
    else if (
        op == '|' || op == '^' ||
        op == 'N' || op == 'R' ||           // not : N left shift: S right shift: s
        op == 'r' || op == 'S' || op == 's') // left rotate: R right rotate: r
    {
        return 5;
    }
    else
    {
        return 0;
    }
}
int isOperator(char op)
{
    if ((op == '+' || op == '*' || op == '-' || op == '&' || op == '|') && unaryCheck != -1)
    {
        return 1;
    }
    else if ((op == '+' || op == '*' || op == '-' || op == '&' || op == '|') && unaryCheck == -1)
    {
        errorCheck = -1;
        return 0;
    }
```

```
    else
    {
        return 0;
    }
}
int varNameCheck(char *var, char *rhs)
{
    int length = strlen(var);
    int startOfVar = 0;
    int endOfVar = length - 1;
    while (isspace(var[startOfVar]))
    {
        startOfVar++;
    }
    while (isspace(var[endOfVar]))
    {
        endOfVar--;
    }
    for (int indx = startOfVar; indx <= endOfVar; indx++)
    {
        if (!isalpha(var[indx]) || isspace(var[indx]))
        {
            printf("Error!\n");
            errorCheck = -1;
            return 0;
        }
    }
    char varName[endOfVar - startOfVar + 2];
    strncpy(varName, var + startOfVar, endOfVar - startOfVar + 1);
    varName[endOfVar - startOfVar + 1] = '\0';
    if (strcmp(varName, "xor") == 0 || strcmp(varName, "ls") == 0 ||
        strcmp(varName, "rs") == 0 || strcmp(varName, "lr") == 0 ||
        strcmp(varName, "rr") == 0 || strcmp(varName, "not") == 0)
    {
        errorCheck = -1;
        return 0;
    }
    long long int value = 0;
    value = evaluate(rhs);
    if (errorCheck != -1)
    {
        int index;
        index = findVar(varName);
        if (index == -1)
        {
```

```
            strcpy(variableList[varCount].name, varName);
            variableList[varCount].value = value;
            varCount++;
        }
        else
        {
            variableList[index].value = value;
        }
    }
    else
    {
        printf("Error!\n");
    }
}
Int findVar(char *name)
{
    for (int i = 0; i < varCount; i++)
    {
        if (strcmp(variableList[i].name, name) == 0)
        {
            return i;
        }
    }
    return -1;
}
long long int evaluate(char *inp)
{
    int length = strlen(inp);
    for (int i = 0; i < length; i++)
    {
        if (errorCheck == -1)
        {
            return 0;
        }
        else if (inp[i] == ' ' || inp[i] == '\t')
        {
            continue;
        }
        else if (inp[i] == ',')
        {
            syntaxCheck[syntaxLength++] = ',';
            syntaxChecker();
            unaryCheck = -1;
            while (operation.operators[operation.operationTop] != '(' && errorCheck != -1)
            {
```

```
            long long int result = applyOperator(&operation);
            pushOperand(&operation, result);
        }
    }
    else if (inp[i] == '(')
    {
        pushParenthesis(&operation);
        syntaxCheck[syntaxLength++] = '(';
        syntaxChecker();
        unaryCheck = -1;
        pushOperator(&operation, inp[i]);
    }
    else if (inp[i] == ')')
    {
        if (operation.parenthesisTop == -1)
        {
            errorCheck = -1;
            continue;
        }
        popParenthesis(&operation);
        syntaxCheck[syntaxLength++] = ')';
        syntaxChecker();
        while (operation.operators[operation.operationTop] != '(' && errorCheck != -1)
        {
            long long int result = applyOperator(&operation);
            pushOperand(&operation, result);
        }
        if (operation.operators[operation.operationTop] == '(')
        {
            popOperator(&operation);
        }
    }
    else if (isdigit(inp[i]))
    {
        syntaxCheck[syntaxLength++] = 'i';
        syntaxChecker();
        unaryCheck = 0;
        long long int operand = inp[i] - '0';
        while (isdigit(inp[i + 1]))
        {
            operand = operand * 10 + (inp[i + 1] - '0');
            i++;
        }
        pushOperand(&operation, operand);
    }
```

```
    else if (isOperator(inp[i]))
    {
        syntaxCheck[syntaxLength++] = 'o';
        syntaxChecker();
        while (operation.operationTop != -1 && errorCheck != -1 &&
               getOperatorPriority(inp[i]) <= getOperatorPriority(operation.operators[operation.operationTop]))
        {
            long long int result = applyOperator(&operation);
            pushOperand(&operation, result);
        }
        unaryCheck = -1;
        pushOperator(&operation, inp[i]);
    }
    else if (isalpha(inp[i]))
    {
        unaryCheck = 0;
        char varOrFunc[maxLen];
        int startI = i;
        int length = 0;
        while (isalpha(inp[i + 1]))
        {
            length++;
            i++;
        }
        strncpy(varOrFunc, inp + startI, length + 1);
        varOrFunc[length + 1] = '\0';
        long long int result = funcNameCheck(varOrFunc);
        syntaxChecker();
        if (result == -1)
        {
            ;
        }
        else
        {
            pushOperand(&operation, result);
        }
    }
    else
    {
        errorCheck = -1;
        return 0;
    }
}
if (operation.parenthesisTop != -1)
{
```

```
        errorCheck = -1;
    }
    while (operation.operationTop != -1 && errorCheck != -1)
    {
        long long int result = applyOperator(&operation);
        pushOperand(&operation, result);
    }
    if (errorCheck != -1)
    {
        if (operation.operandTop != 0 && operation.operandTop != -1)
        {
            errorCheck = -1;
        }
        else
        {
            return popOperand(&operation);
        }
    }
    else
    {
        return 0;
    }
}
long long int applyOperator(Stack *stack)
{
    char operator= popOperator(&operation);
    if (operation.operandTop == 0 && operator!= 'N')
    {
        errorCheck = -1;
    }
    long long int operand2 = popOperand(&operation);
    long long int operand1;
    if (operator!= 'N')
    {
        operand1 = popOperand(&operation);
    }
    else
    {
        ;
    }
    switch (operator)
    {
    case '+':
        return operand1 + operand2;
    case '*':
```

```
        return operand1 * operand2;
    case '-':
        return operand1 - operand2;
    case '&':
        return operand1 & operand2;
    case '|':
        return operand1 | operand2;
    case '^':
        return operand1 ^ operand2;
    case 'S':
        if (operand2 < 0)
        {
            errorCheck = -1;
            return -1;
        }
        else
        {
            return operand1 << operand2;
        }
    case 's':
        if (operand2 < 0)
        {
            errorCheck = -1;
            return -1;
        }
        else
        {
            return operand1 >> operand2;
        }
    case 'N':
        return ~operand2;
    case 'R':
        if (operand2 < 0)
        {
            errorCheck = -1;
            return -1;
        }
        else
        {
            return (operand1 << operand2) | (operand1 >> (64 - operand2));
        }
    case 'r':
        if (operand2 < 0)
        {
            errorCheck = -1;
```

```
            return -1;
        }
        else
        {
            return (operand1 >> operand2) | (operand1 << (64 - operand2));
        }
    }
}
long long int popOperand(Stack *stack)
{
    return stack->operands[stack->operandTop--];
}
long long int funcNameCheck(char *var)
{
    if (strcmp(var, "xor") == 0)
    {
        syntaxCheck[syntaxLength++] = 'f';
        pushOperator(&operation, '^');
        return -1;
    }
    else if (strcmp(var, "ls") == 0)
    {
        syntaxCheck[syntaxLength++] = 'f';
        pushOperator(&operation, 'S');
        return -1;
    }
    else if (strcmp(var, "rs") == 0)
    {
        syntaxCheck[syntaxLength++] = 'f';
        pushOperator(&operation, 's');
        return -1;
    }
    else if (strcmp(var, "lr") == 0)
    {
        syntaxCheck[syntaxLength++] = 'f';
        pushOperator(&operation, 'R');
        return -1;
    }
    else if (strcmp(var, "rr") == 0)
    {
        syntaxCheck[syntaxLength++] = 'f';
        pushOperator(&operation, 'r');
        return -1;
    }
    else if (strcmp(var, "not") == 0)
```

```
    {
        syntaxCheck[syntaxLength++] = 'f';
        pushOperator(&operation, 'N');
        return -1;
    }
    else
    {
        syntaxCheck[syntaxLength++] = 'v';
        int index = findVar(var);
        if (index != -1)
        {
            return variableList[index].value;
        }
        else
        {
            return 0;
        }
    }
}
void pushOperand(Stack *stack, long long int operand)
{
    stack->operands[++stack->operandTop] = operand;
}
void pushOperator(Stack *stack, char operator)
{
    stack->operators[++stack->operationTop] = operator;
}
void pushParenthesis(Stack *stack)
{
    stack->parenthesis[++stack->parenthesisTop] = '(';
}
void syntaxChecker()
{
    char checkChar = syntaxCheck[syntaxLength - 1];
    if (syntaxLength == 1 && syntaxCheck[0] != 'v' && syntaxCheck[0] != 'i' && syntaxCheck[0] != 'f' && syntaxCheck[0] != '(')
    {
        errorCheck = -1;
    }
    else
    {
        switch (syntaxCheck[syntaxLength - 2])
        {
        case 'v':
            if (checkChar != ')' && checkChar != 'o' && checkChar != ',')
            {
```

```
                errorCheck = -1;
            }
            break;
        case 'i':
            if (checkChar != ')' && checkChar != 'o' && checkChar != ',')
            {
                errorCheck = -1;
            }
            break;
        case 'f':
            if (checkChar != '(')
            {
                errorCheck = -1;
            }
            break;
        case '(':
            if (checkChar == 'o' || checkChar == ',' || checkChar == ')')
            {
                errorCheck = -1;
            }
            break;
        case ')':
            if (checkChar != ')' && checkChar != 'o' && checkChar != ',')
            {
                errorCheck = -1;
            }
            break;
        case 'o':
            if (checkChar != 'v' && checkChar != 'i' && checkChar != 'f' && checkChar != '(')
            {
                errorCheck = -1;
            }
            break;
        case ',':
            if (checkChar != 'v' && checkChar != 'i' && checkChar != 'f' && checkChar != '(')
            {
                errorCheck = -1;
            }
            break;
        default:
            break;
        }
    }
}
```