CMPE230 – SYSTEMS PROGRAMMING

Name: Cem Güngör

Project title: advcalc2ir

Project type: programming project

Submission date: 02/05/2023

Introduction

Problem: Implementing an translator for our prior project advanced calculator using C programming, and translating the input to LLVM IR. Calculator will take expressions and assignments. Variables can be recalled during new assignments and expressions. Expressions include 13 different operations some bitwise some integerwise. Values will be at most 32-bit integer valued and none of the expressions or assignment will exceed that limit. Unary is not supported. Variable names consist of only alphabetical characters, and cannot be equal to function names. Inputs are at most 256 characters and undefined variables create an error. If assignment is given as input then allocate,load etc to make LLVM IR work, if expression is given as input transpiler should make necessary writings to file so that when it's run it prints the result. All errors are possible, except for division by 0, interpreter should print "Error in line _!" in such case.

Solution: Using for loop checked each character in line, if there's an error exited the loop and read another line from input file. While reading character by character, pushed every operator(functions, parenthesis and basic operators) and operands(variable loaded line and integers as char arrays) into two separate stacks and using operator priorities throughout the application, calculated the expressions values(not as integers but as lines that point integers. 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, not a value this time, value is put on stack each time code reads the variable and loads it to memory.

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 **advcalc2ir** can be started using make command in root folder and then running the “advcalc.exe” with an input file as parameter, that has lines of expressions and assignments. The program will create a <parameter_name>.ll output file if no errors occur. Inputs can have two kinds: expression and assignment. The execution of the program has one parameter, file name of the input file.

Starting the program: make command and advcalc2ir.exe <filename> 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 input file. 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 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, **undefined variable errors** when an undefined variable is tried to called from an expression or assignment.

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: same as the advcalc, can be seen from the documentation page 4.

Assignment format: same as the advcalc, can be seen from the documentation page 4.

Examples of inputs and outputs: inputs are taken from a file and outputs are written into a file except for errors, examples might take a large space but small explanations:

%k = alloca i32 → alloca is used to allocate memory, i32 is for 32 bit integer and %k is for variable **k**

store i32 3, i32* %x → store is used to store a value into a variable, in this case store i32 3 to i32 x which is probably allocated before stored.

%5 = add i32 23,%4 → an addition operation between integer 23 and integer value of line %4, is equalized to line %5

%8 = load i32, i32* %x → loading a value of a variable from memory to variable %8, can't just call a variable as in C.

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 in line _! The input it basically split into 7: operators, functions, variables, integers, parenthesis, commas, blanks and each of them is treated according to the rules. If no errors occur, then all outputs are written to a file in correct syntax.

Data Structures:

All of them same as advCalc, can be read from page 4 of the documentation. Only the stack of operands changed to return and hold char arrays rather than integers since the values are not calculated anymore, they are add to the stack and written to file when used.

Functions:

Pop and push: same as advcalc, from documentation page 5

getOperatorPriority: same as advcalc, from documentation page 5, just added two new operators: % and /

isOperator: same as advcalc, from documentation page 5

varNameCheck: main idea same as advcalc, from documentation page 5, but this time rather than equalizing rhs to the variable name, or creating a new one, it allocates and then stored the rhs's value or just stores the value if variable exists.

findVar: same as advcalc, from documentation page 5

evaluate: main functioning same as advcalc, from documentation page 5, but this time prints error's line and the stack for digits and variables have char array rather than integers.

applyOperator: returns char pointer, pops last two operand and operator, if the operator is not "not", then writes to .ll file the operation it will make, then returns the line number of the file since they are the variables in the llvm

funcNameCheck: a function that checks if the given parameter: var is a function or variable, if function return – if variable then load the variable from memory if it exist,if not change error check.

syntaxChecker: same as advcalc, from documentation page 5

Improvements and Extensions

I write way too many fputs and sprintf, I would try to change them to improve dependency, or create a variable so that I don't have to change each time I make a syntactic change. Other than that I believe my approach was good.

Difficulties Encountered

Not knowing what to do at first while thinking between integers and llvm variables: %<int>, then trying to change the functions to char pointers or char arrays rather than integers.

Appendices:

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

#define maxLen 256

typedef struct
{
    char name[maxLen];
    int value;
} Variable;
typedef struct
{
    int operandTop;
    int operationTop;
    int parenthesisTop;
    // operands is a 2d array having maxlen to maxlen, first is the count, second
is the length
    char operands[maxLen][maxLen];
    char parenthesis[maxLen];
    char operators[maxLen];
} Stack;
Variable variableList[128];
int varCount = 0;
int lineCount = 0;
int fileLine = 1;
// files defined globally so that I can reach them inside functions.
char *inputFile;
char *outputFile;
FILE *outputF;

Stack operation;
static const Stack EmptyStack;
char syntaxCheck[128];
int syntaxLength;
int unaryCheck;
int errorCheck;
// main error to check if an error occured while evaluating the expr, if so
delete file.
int mainError;
```

```
char popOperator(Stack *stack);
char popParenthesis(Stack *stack);
int getOperatorPriority(char op);
int isOperator(char op);
int varNameCheck(char *, char *);
int findVar(char *);
char *evaluate(char *);
char *applyOperator(Stack *stack);
char *popOperand(Stack *stack);
int funcNameCheck(char *);
void pushOperand(Stack *stack, char operand[]);
void pushOperator(Stack *stack, char operator);
void pushParenthesis(Stack *stack);
void syntaxChecker();

// generally speaking most of the code is the same as advanced calculator, this
time rather than using integers
// used chars for operands and rather than actually applying the operators,
written it to a file, and used the line number
// in llvm file to add to the stack as the result.

int main(int argc, char *argv[])
{
    // this time since we are taking input file as parameter and output file as
the same name as input file
    // created input and output files using args, used fputs to put the header
lines to .ll file
    char inp[maxLen];
    mainError = 0;
    inputFile = argv[1];
    char *dot = strrchr(inputFile, '.');
    FILE *inputF;
    inputF = fopen(inputFile,"r");
    *dot = '\0';
    outputFile = malloc(strlen(inputFile) + 4);
    sprintf(outputFile, "%s.ll", inputFile);

    outputF = fopen(outputFile, "w");
    fputs("; ModuleID = 'advcalc2ir'\n",outputF);
    fputs("declare i32 @printf(i8*, ...)\n", outputF);
    fputs("@print.str = constant [4 x i8]  c\"%d\\0A\\00\"", outputF);
    fputs("\n\ndefine i32 @main() {\n", outputF);

    while (fgets(inp, sizeof(inp), inputF))
    {
```

```
        //line count is added to check which line we are currently on, so to
print error
        lineCount += 1;
        operation = EmptyStack;
        operation.operandTop = -1;
        operation.operationTop = -1;
        operation.parenthesisTop = -1;
        unaryCheck = -1;
        errorCheck = 0;
        syntaxCheck[0] = '\0';
        syntaxLength = 0;
        // changed both \n and \r to \0 since it created problems
        inp[strcspn(inp, "\n")] = '\0';
        inp[strcspn(inp, "\r")] = '\0';
        if (strcmp(inp, "") == 0)
        {
            continue;
        }

        else if (strchr(inp, '=') == NULL)
        {
            // since we write to a file, evaluated expression using char arrays
and pointes rather than integers
            // if there's a problem in the expression then didn't equalized it to
res
            char res[32];
            char *nullCheck = evaluate(inp);
            if (nullCheck != NULL)
            {
                strcpy(res, nullCheck);
            }
            else
            {
                strcpy(res, "False");
            }

            if (syntaxLength == 0 && errorCheck != -1)
            {
                continue;
            }

            else if (syntaxCheck[syntaxLength - 1] != 'v' &&
syntaxCheck[syntaxLength - 1] != 'i' && syntaxCheck[syntaxLength - 1] != ')')
            {
                errorCheck = -1;
```

```
            }

            if (errorCheck != -1)
            {
                char outputLine[256];
                sprintf(outputLine, "\tcall i32 (i8*, ...) @printf(i8*
getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %%%i)\n",
fileLine - 1);
                fputs(outputLine, outputF);
                fileLine++;
            }
            else
            {
                ;
            }
        }
        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';
            rhs[strcspn(rhs, "\r")] = '\0';
            varNameCheck(lhs, rhs);
        }
        if (errorCheck == -1){
            mainError = -1;
        }
    }
    // at last part closed inputfile, write a last line to make .ll file work and
closed it also, but even if one
    // line of the code had error, main error was set to -1, and if so removed
outputfile.
    fclose(inputF);
    fputs("ret i32 0\n}", outputF);
    fclose(outputF);
    if (mainError == -1){
        remove(outputFile);
    }
    return 0;
}
```

```
char popOperator(Stack *stack)
{
    return stack->operators[stack->operationTop--];
}

char popParenthesis(Stack *stack)
{
    return stack->parenthesis[stack->parenthesisTop--];
}

int getOperatorPriority(char op)
{
    // same get fct as the prior project, just added % and /
    if (op == '|')
    {
        return 1;
    }
    else if (op == '^')
    {
        return 2;
    }
    else if (op == '&')
    {
        return 3;
    }
    else if (op == '+' || op == '-')
    {
        return 4;
    }
    else if (op == '*' || op == '%' || op == '/')
    {
        return 5;
    }
    else if (
        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 6;
    }
    else
    {
        return 0;
    }
}
```

```
int isOperator(char op)
{

    if ((op == '+' || op == '*' || op == '-' || op == '&' || op == '|' || op ==
'%' || op == '/') && unaryCheck != -1)
    {
        return 1;
    }
    else if ((op == '+' || op == '*' || 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++)
    {
        // this time if variable name has space or something else other than
alphabetical, then return error
        if (!isalpha(var[indx]) || isspace(var[indx]))
        {
            printf("Error in line %i \n", lineCount);
            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)
    {
        printf("Error in line %i \n", lineCount);
        errorCheck = -1;
        return 0;
    }
    // also checked the evaluate function, if theres an error, return error, if
not then equalize value
    char value[32];
    char *nullCheck = evaluate(rhs);
    if (nullCheck == NULL)
    {
        errorCheck = -1;
        return 0;
    }

    strcpy(value, nullCheck);

    if (errorCheck != -1)
    {
        // if variable exist, then just store new value of the variable, if not
allocate first then store
        char outputLine[256];
        int index;
        index = findVar(varName);
        if (index == -1)
        {
            sprintf(outputLine, "\t%%%s = alloca i32\n", varName);
            fputs(outputLine, outputF);
            strcpy(variableList[varCount].name, varName);
            sprintf(outputLine, "\tstore i32 %s, i32* %%%s\n", value, varName);
            fputs(outputLine, outputF);
            varCount++;
        }
        else
        {
            sprintf(outputLine, "\tstore i32 %s, i32* %%%s\n", value, varName);
```

```
            fputs(outputLine, outputF);
        }
    }
    else
    {
        ;
    }
}

int findVar(char *name)
{

    for (int i = 0; i < varCount; i++)
    {
        if (strcmp(variableList[i].name, name) == 0)
        {
            return i;
        }
    }
    return -1;
}

char *evaluate(char *inp)
{
    // since I need to write to a file and use %(lineCount) rather than integers,
I changed evaluate to return char
    // other than that most of the function works the same, it checks index by
index, checks syntax,paranthesis, etc errors
    // if none occurs then evaluates using the stack, if error occurs prints
error in line and returns null.
    int length = strlen(inp);
    for (int i = 0; i < length; i++)
    {
        if (errorCheck == -1)
        {
            printf("Error in line %d!\n", lineCount);
            return NULL;
        }

        else if (inp[i] == ' ' || inp[i] == '\t')
        {
            continue;
        }

        else if (inp[i] == ',')
```

```
        {
            syntaxCheck[syntaxLength++] = ',';
            syntaxChecker();
            unaryCheck = -1;

            while (operation.operators[operation.operationTop] != '(' &&
errorCheck != -1)
            {
                char result[32];
                strcpy(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)
            {
                char result[32];
                strcpy(result, applyOperator(&operation));
                pushOperand(&operation, result);
            }

            if (operation.operators[operation.operationTop] == '(')
            {
                popOperator(&operation);
            }
        }
        else if (isdigit(inp[i]))
```

```
        {
            syntaxCheck[syntaxLength++] = 'i';
            syntaxChecker();
            unaryCheck = 0;
            int operand = inp[i] - '0';
            while (isdigit(inp[i + 1]))
            {
                operand = operand * 10 + (inp[i + 1] - '0');
                i++;
            }
            char number[32];
            sprintf(number, "%d", operand);
            pushOperand(&operation, number);
        }
        else if (isOperator(inp[i]))
        {
            syntaxCheck[syntaxLength++] = 'o';
            syntaxChecker();

            while (operation.operationTop != -1 && errorCheck != -1 &&
                   getOperatorPriority(inp[i]) <=
getOperatorPriority(operation.operators[operation.operationTop]))
            {
                char result[32];
                strcpy(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';

            int result = funcNameCheck(varOrFunc);
```

```
            syntaxChecker();
            if (result == -1 || result == 0)
            {
                ;
            }
            else
            {
                char res[32];
                sprintf(res, "%%%d", result);
                pushOperand(&operation, res);
            }
        }
        else
        {
            printf("Error in line %i!\n", lineCount);
            errorCheck = -1;
            return NULL;
        }
    }
    // since for loop finished, if any errors occur from the remaining stacks,
then print those
    // if not then do the operations till it finishes.
    if (operation.parenthesisTop != -1)
    {
        printf("Error in line %i!\n", lineCount);
        errorCheck = -1;
        return NULL;
    }
    while (operation.operationTop != -1 && errorCheck != -1)
    {
        char result[32];
        strcpy(result, applyOperator(&operation));
        pushOperand(&operation, result);
    }

    if (errorCheck != -1)
    {
        if (operation.operandTop != 0 && operation.operandTop != -1)
        {
            printf("Error in line %i!\n", lineCount);
            errorCheck = -1;
            return NULL;
        }
        else
        {
```

```
            return popOperand(&operation);
        }
    }
    else
    {
        printf("Error in line %i!\n", lineCount);
        return NULL;
    }
}

char *applyOperator(Stack *stack)
{
    // the function that plays a huge role in functioning of the code. Uses pop
operator and operands to look for the
    // operation, switch-case between operations and if match is found, writes
file the necessary output,
    // increments fileLine, which checks the %(fileline) elements in llvm and
returns the result line.
    static char result[32];
    char operator= popOperator(&operation);
    if (operation.operandTop == 0 && operator!= 'N')
    {
        errorCheck = -1;
    }
    char operand2[32];
    strcpy(operand2, popOperand(&operation));
    char operand1[32];
    char outputLine[256];
    if (operator!= 'N')
    {
        strcpy(operand1, popOperand(&operation));
    }
    else
    {
        ;
    }
    switch (operator)
    {
    case '+':
        sprintf(outputLine, "\t%%%i = add i32 %s, %s\n", fileLine, operand1,
operand2);
        fputs(outputLine, outputF);
        fileLine++;
        sprintf(result, "%%%d", fileLine - 1);
        return result;
```

```
    case '*':
        sprintf(outputLine, "\t%%%i = mul i32 %s, %s\n", fileLine, operand1,
operand2);
        fputs(outputLine, outputF);
        fileLine++;
        sprintf(result, "%%%d", fileLine - 1);
        return result;
    case '-':
        sprintf(outputLine, "\t%%%i = sub i32 %s, %s\n", fileLine, operand1,
operand2);
        fputs(outputLine, outputF);
        fileLine++;
        sprintf(result, "%%%d", fileLine - 1);
        return result;
    case '/':
        sprintf(outputLine, "\t%%%i = sdiv i32 %s, %s\n", fileLine, operand1,
operand2);
        fputs(outputLine, outputF);
        fileLine++;
        sprintf(result, "%%%d", fileLine - 1);
        return result;
    case '&':
        sprintf(outputLine, "\t%%%i = and i32 %s, %s\n", fileLine, operand1,
operand2);
        fputs(outputLine, outputF);
        fileLine++;
        sprintf(result, "%%%d", fileLine - 1);
        return result;
    case '|':
        sprintf(outputLine, "\t%%%i = or i32 %s, %s\n", fileLine, operand1,
operand2);
        fputs(outputLine, outputF);
        fileLine++;
        sprintf(result, "%%%d", fileLine - 1);
        return result;
    case '^':
        sprintf(outputLine, "\t%%%i = xor i32 %s, %s\n", fileLine, operand1,
operand2);
        fputs(outputLine, outputF);
        fileLine++;
        sprintf(result, "%%%d", fileLine - 1);
        return result;
    case '%':
        sprintf(outputLine, "\t%%%i = srem i32 %s, %s\n", fileLine, operand1,
operand2);
```

```
        fputs(outputLine, outputF);
        fileLine++;
        sprintf(result, "%%%d", fileLine - 1);
        return result;
    case 'S':
        if (operand2 < 0)
        {
            errorCheck = -1;
            sprintf(result, "-1");
            return result;
        }
        else
        {
            sprintf(outputLine, "\t%%%i = shl i32 %s, %s\n", fileLine, operand1,
operand2);
            fputs(outputLine, outputF);
            fileLine++;
            sprintf(result, "%%%d", fileLine - 1);
            return result;
        }
    case 's':
        if (operand2 < 0)
        {
            errorCheck = -1;
            sprintf(result, "-1");
            return result;
        }
        else
        {
            sprintf(outputLine, "\t%%%i = ashr i32 %s, %s\n", fileLine, operand1,
operand2);
            fputs(outputLine, outputF);
            fileLine++;
            sprintf(result, "%%%d", fileLine - 1);
            return result;
        }
    case 'N':
        sprintf(outputLine, "\t%%%i = xor i32 %s, -1\n", fileLine, operand2);
        fputs(outputLine, outputF);
        fileLine++;
        sprintf(result, "%%%d", fileLine - 1);
        return result;
    case 'R':
        if (operand2 < 0)
        {
```

```
            errorCheck = -1;
            sprintf(result, "-1");
            return result;
        }
        else
        {
            // in order to make rotates work, first right shift then left shift
then or line by line
            // separated the function operand1 << operand2 || operand1 >> (32-
operand2)
            sprintf(outputLine, "\t%%%i = shl i32 %s, %s\n", fileLine, operand1,
operand2);
            fputs(outputLine, outputF);
            fileLine++;

            sprintf(outputLine, "\t%%%i = ashr i32 %s, sub i32 32, %s\n",
fileLine, operand1, operand2);
            fputs(outputLine, outputF);
            fileLine++;

            sprintf(outputLine, "\t%%%i = or i32 %%%i, %%%i\n", fileLine,
fileLine - 1, fileLine - 2);
            fputs(outputLine, outputF);
            fileLine++;

            sprintf(result, "%%%d", fileLine - 1);
            return result;
        }
    case 'r':
        if (operand2 < 0)
        {
            errorCheck = -1;
            sprintf(result, "-1");
            return result;
        }
        else
        {
            sprintf(outputLine, "\t%%%i = ashr i32 %s, %s\n", fileLine, operand1,
operand2);
            fputs(outputLine, outputF);
            fileLine++;

            sprintf(outputLine, "\t%%%i = shl i32 %s, sub i32 32, %s\n",
fileLine, operand1, operand2);
            fputs(outputLine, outputF);
```

```
            fileLine++;

            sprintf(outputLine, "\t%%%i = or i32 %%%i, %%%i\n", fileLine,
fileLine - 1, fileLine - 2);
            fputs(outputLine, outputF);
            fileLine++;

            sprintf(result, "%%%d", fileLine - 1);
            return result;
        }
    }
}

char *popOperand(Stack *stack)
{
    // operands are no longer integers so returns pointer
    stack->operandTop--;
    return stack->operands[stack->operandTop];
}

int funcNameCheck(char *var)
{
    // if function is found then add function to syntaxcheck and push operator
    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
    {
        // if a variable is found then load variable to memory, and write to file
        char outputLine[256];
        syntaxCheck[syntaxLength++] = 'v';
        int index = findVar(var);
        if (index != -1)
        {
            sprintf(outputLine, "\t%%%i = load i32, i32* %%%s\n", fileLine,
variableList[index].name);
            fputs(outputLine, outputF);
            fileLine++;
            return fileLine - 1;
        }
        else
        {
            errorCheck = -1;
            return 0;
        }
    }
}

void pushOperand(Stack *stack, char operand[])
{
    // pushes the given operand to the given stack using strcpy.
    strcpy(stack->operands[stack->operandTop], operand);
    stack->operandTop++;
}

void pushOperator(Stack *stack, char operator)
{
```

```
    stack->operators[++stack->operationTop] = operator;
}

void pushParenthesis(Stack *stack)

{
    stack->parenthesis[++stack->parenthesisTop] = '(';
}

void syntaxChecker()
{
    // a syntax checker to check syntax of the given input, one by one, using the
conditions
    // such as line cant end with an operator etc.
    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;
        }
    }
}
```