Algorithms and Data Structures II

REPORT

Section 1

Description of the solution

The solution is a Postfix++ interpreter, it can do the following types of actions: perform simple arithmetical operations (+, -, *, /) assign variables and manage them in the Symbol Table (insert and search for variables). The key data structure utilized by Postfix++ is a stack. Programming stacks are based on the principle of last in first out (LIFO), a commonly used type of data abstract that consists of two major operations, push and pop.

Practical implementation

We utilize C++ language for this app.

The program starts by initializing `interpreter` object. It enters a loop, continuously reading lines of input from the console. Each line of input is treated as a postfix expression or an assignment statement. The `differentiate_input` method of the `PostfixInterpreter` object is called with the input string. This method parses the input string token by token and performs the required operations. For each token in the input:

If the token is a number, it is pushed onto the stack. If the token is a variable (a single letter), its value is fetched from the symbol table and pushed onto the stack. If the token is an operator (`+`, `-`, `*`, `/`), the required number of operands is popped from the stack, the operation is performed, and the result is pushed back onto the stack. If the token is an assignment operator (`=`), the top two values are popped from the stack, and the variable is assigned the value in the symbol table. If the token is invalid, an error is thrown.

Section 2

The interpreter processes postfix expressions by reading each token (numbers, variables, and operators) one by one. When a number or variable is encountered, it is pushed onto a stack. When an operator is encountered, the appropriate number of operands is popped from the stack, the operation is performed, and the result is pushed back onto the stack.

For variable assignments, the value is stored in a symbol table. This approach ensures that the interpreter can efficiently evaluate expressions and manage variable values.

Section 3

Pseudocode

We need to differentiate between operands (numbers and variables), operators, and assignment operations:

```
Function differentiate_input (input):

    initialize stack
```

```
for each token in input:
        if token is a number:
            stack.push(token)
        else if token is a variable:
            value = symbolTable[token]
            stack.push(value)
        else if token is an operator:
            stack_top2 = stack.pop()
            stack_top1 = stack.pop()
            result = apply operator to stack_top2  and stack_top1
            stack.push(result)
        else if token is "=":
            value = stack.pop()
            key = stack.pop()
            symbolTable[key] = value
        else:
            error
```

Also we need some operations with our symbolTable:

```
function insert(symbolTable, variable, value):
    if variable exists in symbolTable:
        symbolTable[variable] ← value
    else:
        symbolTable.add(variable, value)

function search(symbolTable, variable):
    if variable exists in symbolTable:
        return symbolTable[variable]
    else:
        return error "Variable not found"
```

Other

```
function getIndexForVariable(variable):
       return ASCII value of variable - ASCII value of 'A'
```

Section 4 Data structures

We have 2 main types of data structures: a stack and a symbol table.

A stack is used to store numbers, operators and to perform arithmetic operations. It is good for postfix++ expressions. We implement a stack with an array, we add tokens by pushing them on top of the stack and then pop them to perform arithmetic operations.

Symbol table is used to store values of variables , mapping the keys with values. We use a fixed size array (26 as the length of alphabet for letters A-Z), it is simple, efficient, especially if we want our interpreter to run on a small device with very limited memory, variable namespace is small, 'A'-'Z'.

Section 5

Link to the video:

https://github.com/CatPawsCoder/Postfix-_interpreter/blob/main/video_postfix.mp4

Source code:

We have two files a header file and the .cpp file

Postfix.h:

Here we announce the Postfix class and its private and public variables and functions

```
#include <iostream>
#include <string>
#include <stack>

// number of variables - letters of alphabet, fixed
const int NUM_VARS = 26;

class Postfix {
    private:
        std::stack<double> Stack;
        double symbolTable[NUM_VARS] = {0};

        int getIndex(char var);
         // Search for the value of a variable in the symbol table
        double search(char var);
        // Insert a value for a variable in the symbol table
        void insert(char var, double value);
       // Print Symboltable in console
        void printSymbolTable();
```

```
    public:
// here is the main algorythm
        double differentiate_input(const std::string& expression);
};
```

Postfix.cpp:

```
#include "Postfix.h"
#include <iostream>
#include <sstream>
#include <cctype>
#include <cmath>
#include <stdexcept>


// Convert a variable character to an index in the symbol table (A=0, B=1, ...,
Z=25)
int Postfix::getIndex(char var) {
    return var - 'A';
}

//we search for index of variable and return the value of the variable
double Postfix::search(char var) {
    int index = getIndex(var);
    return symbolTable[index];
}


//we  fill in the symbolTable with value at the right index and variable
void Postfix::insert(char var, double value) {
    int index = getIndex(var);
    symbolTable[index] = value;
    printSymbolTable();
}


//we print SymbolTable so that we can check that it is filled
void Postfix::printSymbolTable() {
    std::cout << "Symbol Table:" << std::endl;
    for (int i = 0; i < NUM_VARS; ++i) {
        if (symbolTable[i] != 0) {
            std::cout << static_cast<char>('A' + i) << ": " << symbolTable[i] <<
std::endl;
        }
    }
}


// help determine whether the expression is a variable or not
```

```cpp
bool isVariable(double value) {
    return value >= 'A' && value <= 'Z';
}



double Postfix::differentiate_input(const std::string& expression) {
    // read from input string,so that then we extract tokens
    std::istringstream tokens(expression);
    std::string token;


    //extract tokens separated by whitespaces, assign then to token
    while (tokens >> token) {
        //push token to stack , if digits in double format
        if (isdigit(token[0]) || (token.size() > 1)) {
            Stack.push(std::stod(token));
        //push token alphabetic character to stack , letters in ascii format
        }
        else if (isalpha(token[0]) && token.size() == 1) {
            Stack.push(static_cast<double>(token[0]));
        }
        //if token arithmetic operator (we add exponent and modulus %) perform
operation on top 2 elements
        else if (token == "+" || token == "-" || token == "*" || token == "/" ||
token == "^" || token == "%") {
            // added error detection
            if (Stack.size() < 2) {
                throw std::invalid_argument("Insufficient operands for
operation");
            }

            // pop the top 2 elements of stack
            double stack_top2 = Stack.top();
            Stack.pop();
            double stack_top1 = Stack.top();
            Stack.pop();


            // check if stack_top1 is within the ASCII range for uppercase
letters if yes convert to   char using static_cast<char>(stack_top1).
            if (isVariable(stack_top1)) {
                stack_top1 = search(static_cast<char>(stack_top1));
            }
            if (isVariable(stack_top2)) {
                stack_top2 = search(static_cast<char>(stack_top2));
            }
```

```
            // Compute the result based on the operator  and pushe the result
back onto the Stack
            double result = 0;
            if (token == "+") result = stack_top1 + stack_top2;
            else if (token == "-") result = stack_top1 - stack_top2;
            else if (token == "*") result = stack_top1 * stack_top2;
            else if (token == "/") {
                // division by 0 error
                if (stack_top2 == 0) throw std::invalid_argument("Division by
zero");
                result = stack_top1 / stack_top2;
            }
            else if (token == "^") result = std::pow(stack_top1, stack_top2);
            else if (token == "%") result = std::fmod(stack_top1, stack_top2);

            Stack.push(result);
        }
        // check if we have assignment operator, if yes we make assignment and
pop top 2 stack tokens
        else if (token == "=") {
            if (Stack.size() < 2) {
                throw std::invalid_argument("Insufficient operands for
assignment");
            }
            double value = Stack.top();
            Stack.pop();
            double var_as_double = Stack.top();
            Stack.pop();
            char var = static_cast<char>(var_as_double);

            // insert into symbolTable
            insert(var, value);
            // push the value to stack
            Stack.push(value);
        }

        // other case -error
        else {
            throw std::invalid_argument("Invalid token encountered");
        }
    }

    // return the top if stack is not empty
    if (Stack.empty()) {
        return 0;
    } else {
        return Stack.top();
    }
}

int main() {
```

```
    // we create object interpreter of class Postfix
    Postfix interpreter;
    std::string input;

    std::cout << "> ";
    // loop; function reads a line of text from the standard input and stores it
in the input string
    while (std::getline(std::cin, input)) {
        try {
            // process the input string as a postfix and return the result as
double
            double result = interpreter.differentiate_input(input);
            std::cout << result << std::endl;
        }
        //  catch block to handle any exceptions
        catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << std::endl;
        }
        std::cout << "> ";
    }

    return 0;
}
```