ADS Mid Term Report

Xavier Neo

3 Jul 2024

1 Introduction

I have approached this challenge of creating a postfix interpreter by creating a REPL style interface, similar to the one used by Python.

REPL, which stands for Read, Evaluate, Print, Loop, is exactly what my program would do. It would take a user input, evaluate it as a postfix expression, print out the result, and repeat the process again.

To implement this, I chose to use the C++ language as language features such as references and pointers would give me control of how my program uses memory, which is vital for devices with limited memory, a characteristic of the targeted hardware. As such, my postfix interpreter uses memory intentionally, making sure not to create unnecessary copies of variables and using pointers to data objects.

Using the object oriented programming features of C++, I have also made the program modular, where by the different data structures used in the program can extended or used outside of this program.

This implementation also includes additional functionality, such as:

1. Support for floating point values
2. Additional mathematical functions (tan, sin , cos, log and $\surd$)
3. Memory manipulation functions (Clearing of Postfix stack)

2 Explaination of Algorithms

2.1 Postfix Interpreter Algorithms

2.1.1 Evaluate

The evaluation algorithm evaluates a postfix expression and executes the necessary operations to compute that expression. The steps of the evaluate algorithm are as such:

1. Tokenize the expression using space as the delimiter.
2. Evaluate each token from the start to the end of the expression.
3. For each token, they fall under a few categories, each with its own set of actions:
   (a) Operators: execute the arithmetic algorithm and carry out its specific operations on the postfix stack based on the operator.
   (b) Assignment operator, or =: execute the assignment algorithm to assign a variable with a value using a hash table.
   (c) Mathematical functions: execute the math function algorithm to carry out the specific mathematical function on the postfix stack.
   (d) Delete: execute the delete algorithm to remove a variable from the hash table.
   (e) Search: execute the search algorithm to search for a variable in the hash table.
   (f) Alphanumeric: push the token to the postfix stack.

   If the token does not fall into one of these categories, print a message that an invalid token is being evaluated.
4. Finally, print the postfix stack.

2.1.2 Arithmetic

This algorithm carries out arithmetic operations on a postfix stack based on the input operator. The steps of the algorithm are as such:

1. Pop the stack twice.
2. Carry out the arithmetic operation based on the operator given.
3. Push the result back to the postfix stack.

2.1.3 Assignment

This algorithm assigns a variable to a value using the top two elements of the stack. The steps of the algorithm are as such:

1. Pop the postfix stack twice.
2. Check that one element is a character or string, and the other is a number. If not, throw an error.
3. Insert a key-value pair into the hash table, using the character/string as the variable and the number as the value.

2.2 Stack Algorithms

For the stack algorithms, they are written in the context of using a C++ array for the implementation of the stack. Some characteristics to note about C++ arrays are:

1. Once an array is created, its size/capacity cannot be changed.
2. When an array is created, the program allocates memory to fit that size of the array regardless of whether the array is filled or empty. As such, the following algorithms take these characteristics into account.

Attributes of the stack are:

1. capacity: An integer value corresponding to the size of the stack.
2. topindex: An integer value corresponding to the position of the top of the stack.

2.2.1 Push

This algorithm adds an element to the top of the stack. The steps are as such:

1. Check if the stack is full by comparing the topindex and the capacity attributes.
2. If the stack is full, i.e. topindex=capacity-1:
    (a) Create a new stack twice the size of the previous stack.
    (b) Copy over elements of the previous stack.
    (c) Delete the previous stack.
3. If the stack is not full, skip this step.
4. Update the topindex attribute by incrementing it by 1.
5. Assign the value to be pushed to the stack at the position of topindex.

2.2.2 Pop

This algorithm emulates removing the topmost element from the stack. This is done by decrementing the value of topindex by 1. The value of the popped element is not deleted or removed as it will be reassigned when the stack is pushed. This removes the redundant step of deleting the variable so that it can be reassigned as the memory has already been allocated and thus won't reduce memory load.

2.2.3 Top

This algorithm retrieves the top of the stack. The steps are as such:

1. Check that the stack is not empty, i.e. no elements to retrieve from the top of the stack, by checking the value of topindex.
2. If the stack is empty, throw an error.
3. If the stack is not empty, retrieve the value of the array at the topindex position.

2.3 Linked List Algorithms

My implementation of a linked list varies slightly from a traditional singly linked list as it holds two data objects, a key and a value.

2.3.1 insertNode

This algorithm adds a new node at the end of the linked list. The steps are as such:

1. Create a new node with the given key and value.
2. Check if the linked list is empty, if it is, change the head pointer to point to the new node.
3. If the linked list is not empty, traverse the list until the end of the list is reached.
4. Point the next pointer of the last element of the list to the new node.
5. Point the next pointer of the new node to a null pointer, marking it as the end of the list.

2.3.2 deleteNode

This algorithm searches for a node based on a given key and removes it from the linked list. The steps are as such:

1. Check if the list is empty, i.e. no items left to be removed. If so, print a message saying that the list is empty and stop the algorithm.
2. Check the key value of the first node, if it matches:
   (a) Set the head pointer to point to the next element of the linked list.
   (b) Delete the first node.
   (c) Stop the algorithm.
3. Create another pointer called left, and temp, pointing to the first node.

4. Loop through each element in the list:
    (a) Check the key value of the node, and check that the node is not the last element of the list, if it is either, exit the loop.
    (b) The left pointer will point to the node that was just checked while the temp pointer will point to the next node.
5. If the loop was exited by reaching the end of the list, print a message saying that the key was not found and stop the algorithm.
6. At this step, this would mean that the temp pointer is pointing to the node with the found key and the left pointer is pointing to the previous node. Set the next pointer of the left node to the next node of the node temp is pointing to.
7. Delete the node that temp is pointing to.

2.4 HashTable Algorithms

My implementation of a hash table uses the djb2 hash function and separate chaining as its collision resolution method using the linked list above.

2.4.1 djb2 hash

This algorithm takes a string and returns a hash based on the input string. I chose to implement this hash function as it is a well established string hash function and has a good distribution thus lowering the chances of collisions. This function might be excessive for this use case, however I wanted to use this opportunity to learn and implement a hash function that is closer to a real-world implementation.

The steps are as such:

1. Initialize a hash variable of 5381, this number is intentional as it was shown to produce the least number of collisions.
2. Iterate over each character of the string, for each character:
    (a) Multiply the hash variable by 33.
    (b) Add the ASCII code to the hash variable.
    (c) Mod the hash by the table size.

2.4.2 insert

This algorithm inserts a key-value pair into the hash table.

The steps are as such:

1. Hash the key and get the corresponding linked list.

2. Check that the key does not exist in the list, if it does, replace the value and stop the algorithm.

3. If the key does not exist in the list, insert the new key-value pair at the end of the list.

2.4.3 search

This algorithm searches the hash table for a given key and retrieves the value.
The steps are as such:

1. Hash the key and get the corresponding linked list.

2. Search the linked list for the key, if found, retrieve the corresponding value.

3. If the key is not found, print a message that the key was not found.

2.4.4 remove

This algorithm removes a given key from the hash table.
The steps are as such:

1. Hash the key and get the corresponding linked list.

2. Search the linked list for the key, if found, delete the node from the linked list.

3. If the key is not found, print a message that the key was not found.

3 Psuedocode of Algorithms

3.1 Linked List Pseudocode

```
insertNode(key, value)
newNode = Node(key, value)
if HEAD = NIL
    HEAD = newNode
    return
last = HEAD
while last.next ≠ NIL
    last = last.next
last.next = newNode
newNode.next = NIL
return
```

```
DELETENODE(key)
if linkedlist is empty
    PRINT("List is empty")
    return
Initalise temp to HEAD
if temp.key = key
    HEAD = temp.next
    return
left = HEAD
while temp ≠ NIL and temp.key ≠ key
    left = temp
    temp = temp.next
if temp = NIL
    PRINT("Node not found")
    return
prev.next = curr.next
Delete node at temp
return
```

3.2 Hashtable Pseudocode

```
INSERT(key, value)
index = HASH(key)
list = table[index]
nodePtr = list.head
while nodePtr ≠ NIL
    if nodePtr.key = key
        nodePtr.value = value
        return
    nodePtr = nodePtr.next
Insert node(key, value) into list
return
```

```
SEARCH(key)
index = HASH(key)
list = table[index]
nodePtr = list.head
while nodePtr ≠ NIL
    if nodePtr.key = key
        return nodePtr.value
    nodePtr = nodePtr.next
return 0
```

```
REMOVE(key)
1  index = HASH(key)
2  list = table[index]
3  DELETE node with key from list
4  return
```

```
HASH(s) → INTEGER
1  hash = 5381
2  for each character In s
3      hash = hash × 33 + (ASCII value of character)
4  return hash % tableSize
```

3.3 REPL Pseudocode

```
EVAL(expression)
 1  Tokenize expression using space as delimiter
 2  for each token In expression
 3      if token is an arithmetic operator
 4          ARITHMETIC(token)
 5      elseif token is a mathematical function
 6          MATHFUNC()
 7      elseif token is "="
 8          ASSIGNMENT()
 9      elseif token is "DEL"
10          DEL()
11      elseif token is "SCH"
12          SEARCH()
13      elseif token is numeric or alphabetical
14          Push token to postfixStack
15      else PRINT("Invalid character")
```

```
ARITHMETIC(op)
 1  POP postfixStack twice and store values in num1 and num2
 2  if op = '+'
 3      result = num1 + num2
 4  if op = '-'
 5      result = num1 − num2
 6  if op = '*'
 7      result = num1 ∗ num2
 8  if op = '/'
 9      result = num1 ÷ num2
10      if num2 = 0
11          error
12  PUSH result to postfixStack
```

```
ASSIGNMENT()
1  POP postfixStack twice and store values in el1 and el2
2  if el1 and el2 are both alphabetical or both numeric
3      error
4  Insert el2 into symbolTable with key el1 or vice versa based on type
```

3.4 Stack Pseudocode

```
PUSH(value)
1  if $topIndex = capacity - 1$
2      Create newStack of size $capacity * 2$
3      for $i = 0$ to $capacity - 1$
4          newStack[i] = stack[i]
5      Delete array stack
6      stack = newStack
7      $capacity = capacity * 2$
8  $topIndex = topIndex + 1$
9  stack[$topIndex$] = value
```

```
POP()
1  if topIndex = -1
2      print “Stack is empty”
3      return
4  topIndex = topIndex − 1
```

```
TOP() → STRING
1  if topIndex = -1
2      error
3  return stack[topIndex]
```

4 Data Structures

4.1 Stack

The stack data structure was chosen to be used as the postfix stack. Due to the nature of the the postfix language, where operations are carried out on the last two element of the stack. A stack would be the most suitable choice for the postfix interpreter.

4.2 Linked List

The Linked List data structure was chosen to be used as part of the hash table for separate chaining as a method to resolve collisions. It is suitable for this task as it is unknown how many collisions will happen, therefore implementing another data structure such as an array would not be practical as the size of an array is fixed, and if the array if full, another array would need to be created and the original would need to be deleted. On the other hand, by using a linked list, the size of the list is not fixed and can increase as long as there is sufficient memory. This is because memory used for the linked list is not allocated in one contiguous chunk as compared to an array.

4.3 Hash Table

The Hash table data structure was chosen to implement the symbol table of the postfix interpreter. A hash table is suitable for this task as it is able to map key-value pairs while maintaining low memory consumption and search time through hashing and indexing of variables.

5 Implementation

The video demo of my implementation can be found at this link https://youtu.be/3qQ5HLUwAhU

5.1 stack.h

```
#include <string>

using namespace std;

class Stack {
public:
    Stack();
    Stack(int capacity);
    void push(string value);
    void pop();
    string& top();
    bool isEmpty();
    void clear();
    int size();
    void print();

private:
    string* stack;
    int capacity;
    int topIndex;
};
```

5.2 stack.cpp

```
#include "stack.h"
```

```
#include <iostream>

using namespace std;

// Default constructor
Stack::Stack()
{
    capacity = 10;
    stack = new string[capacity];
    topIndex = -1;
}

// Constructor if capacity is specified
Stack::Stack(int capacity)
{
    this->capacity = capacity;
    stack = new string[capacity];
    topIndex = -1;
}

// Adds an element to the top of the stack
void Stack::push(string value)
{
    // If the stack is full, double the capacity
    if (topIndex == capacity - 1) {
        string* newStack = new string[capacity * 2];
        for (int i = 0; i < capacity; i++) {
            newStack[i] = stack[i];
        }
        delete[] stack;
        stack = newStack;
        capacity *= 2;
    }

    // Increment topIndex and add value to stack
    stack[++topIndex] = value;
}

// "Removes" the top element of the stack
void Stack::pop()
{
    // If the stack is empty, print an error message
    if (topIndex == -1) {
        cout << "Stack is empty" << endl;
        return;
    }
    // Decrement topIndex
    topIndex--;
}

// Returns the top element of the stack
string& Stack::top()
{
    // If the stack is empty, throw an exception
    if (topIndex == -1) {
        throw runtime_error("Stack is empty");
    }

    return stack[topIndex];
}

// Returns true if the stack is empty
bool Stack::isEmpty()
{
    return topIndex == -1;
}

// Returns the number of elements in the stack
int Stack::size()
{
    return topIndex + 1;
}

// Prints the stack
void Stack::print()
{
    cout << "Stack: [ ";
    for (int i = 0; i <= topIndex; i++) {
        cout << stack[i] << " ";
    }
    cout << "]" << endl;
}
```

5.3 linkedlist.h

```
#include <string>

using namespace std;

class Node {
public:
    string key;
    int value;
    Node* next;

    Node(string key, double value)
        : key(key)
        , value(value)
        , next(nullptr)
    {
    }
};

class LinkedList {
public:
    Node* head;

    // Default constructor
    LinkedList();

    // Insert a node at the end of the linked list.
    void insertNode(string& key, double value);

    // Deletes a node with the given key.
    void deleteNode(string& key);

    // Prints the linked list.
    void printList();

    bool isEmpty() { return head == nullptr; };
};
```

5.4 linkedlist.cpp

```
#include "linkedlist.h"
#include <iostream>

LinkedList::LinkedList()
{
    head = nullptr;
}

/* Inserts node at end of list */
/*
 * Pseudocode

 insertNode(key, value)
```

```
    newNode = Node(key, value)
    if HEAD = NIL
        HEAD = newNode
        return
    last = HEAD
    while last.next != NIL
        last = last.next
    last.next = newNode
    newNode.next = NIL
    return

 */
void LinkedList::insertNode(string& key, double value)
{
    // Create new node to be inserted
    Node* newNode = new Node(key, value);

    // If LinkedList is empty
    if (head == nullptr) {
        head = newNode;
        return;
    }

    // Traverse to end of list
    Node* last = head;
    while (last->next != nullptr) {
        last = last->next;
    }

    // Set next pointer of last node to new node
    last->next = newNode;

    // Set next pointer of new node to nullptr
    newNode->next = nullptr;

    return;
}

void LinkedList::printList()
{
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->key << " " << temp->value << " -> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
```

```
}

/*
 * Pseudocode

 deleteNode(key)
    if linkedlist is empty
        print("List is empty")
        return
    Initalise temp to HEAD
    if temp.key = key
        HEAD = temp.next
        return
    left = HEAD
    while temp!= NIL and temp.key!= key
        left = temp
        temp = temp.next
    if temp = NIL
        print("Node not found")
        return
    prev.next = curr.next
    Delete node at temp
    return

 */
void LinkedList::deleteNode(string& key)
{
    // If LinkedList is empty
    if (isEmpty()) {
        cout << "List is empty" << endl;
        return;
    }

    // Initalize temp to HEAD
    Node* temp = head;

    // First node itself is the node to be deleted
    if (temp->key == key) {
        head = temp->next;
        delete temp;
        cout << "Key deleted" << endl;
        return;
    }

    // Search for node to be delete
    // Initalize left to HEAD
    Node* left = head;
    while (temp != nullptr && temp->key != key) {
        left = temp;
        temp = temp->next;
    }

    // If while loop was exited because temp is nullptr
    if (temp == nullptr) {
        cout << "Node not found" << endl;
        return;
    }

    // Set next pointer of left node to the next node
    left->next = temp->next;
    // Delete node
    delete temp;
    cout << "Key deleted" << endl;
    return;
}
```

5.5 hashtable.h

```
#include "linkedlist.h"
#include <string>
using namespace std;

class HashTable {
public:
    HashTable(int size);
    bool isEmpty();
    void insert(string key, double value);
    double search(string key);
    void remove(string key); // delete is a keyword
    void printTable();

private:
    int tableSize;
    int hash(std::string const& s);
    LinkedList* table;
};
```

5.6 hashtable.cpp

```
#include "hashtable.h"
#include <iostream>
```

```
using namespace std;

HashTable::HashTable(int size)
    : tableSize(size)
{
    table = new LinkedList[size];
}

bool HashTable::isEmpty()
{
    // Loop through all linked lists in the table
    for (int i = 0; i < tableSize; i++) {
        // Check that the linked list is empty, if not return false
        if (!table[i].isEmpty())
            return false;
    }
    return true;
}

// Inserts a key-value pair into the hash table
/*
 * Pseudocode

 INSERT (key, value)
    index = HASH(key)
    list = table|index]
    nodePtr = list.head
    while nodePtr # NIL
        if nodePtr.key = key
            nodePtr.value = value
            return
        nodePtr = nodePtr.next
    Insert node (key, value) into list
    return

 */
void HashTable::insert(string key, double value)
{
    // Get the hash of the key and access the linked list at that index
    int index = hash(key);
    auto& list = table[index];

    Node* nodePtr = list.head;

    // Search for key in linked list, if found, replace value
    while (nodePtr != nullptr) {
        if (nodePtr->key == key) {
            nodePtr->value = value;
            cout << "Key already exists, replacing " + key + " with stored value "\\
            << value << endl;
            return;
        }
        nodePtr = nodePtr->next;
    }

    // If key does not exist in the list, insert it
    list.insertNode(key, value);

    return;
}

// Searches for a key in the hash table and returns the value
/*
 * Pseudocode

 SEARCH (key)
    index = HASH(key)
    list = table[index]
    nodePtr = list.head
    while nodePtr # NIL
        if nodePtr.key = key
            return nodePtr.value
        nodePtr = nodePtr.next
    print "Key not found"
    return 0

 */
double HashTable::search(string key)
{
    // Get the hash of the key and access the linked list at that index
    int index = hash(key);
    auto& list = table[index];

    // Search for key in linked list, and return value if found
    Node* nodePtr = list.head;
    while (nodePtr != nullptr) {
        if (nodePtr->key == key) {
            return nodePtr->value;
        }
        nodePtr = nodePtr->next;
    }
    cout << "Key not found" << endl;
    return 0;
}

// Removes a key-value pair from the hash table
/*
 * Pseudocode

 DELETE (key)
    index = HASH(key)
    list = table[index]
    DELETE node with key from list
    return

 */
void HashTable::remove(string key)
{
    // Get the hash of the key and access the linked list at that index
    int index = hash(key);
    auto& list = table[index];

    // Delete node with key
    list.deleteNode(key);
    return;
}

// Prints table
void HashTable::printTable()
{
    cout << "Printing table" << endl;
    cout << "--------------" << endl;
    for (int i = 0; i < tableSize; i++) {
        if (table[i].isEmpty())
            continue;
        cout << "[ " << i << " ] ";
        table[i].printList();
    }
    cout << "--------------" << endl;
}

// djb2 hash function from
    https://stackoverflow.com/questions/19892609/djb2-by-dan-bernstein-for-c
/*
 * Pseudocode

    hash = 5381
    for each character In s
        hash = hash × 33 + (ASCII value of character)
    return hash % tableSize

 */
int HashTable::hash(std::string const& s)
{
    unsigned long hash = 5381;
    for (auto c : s) {
        hash = (hash << 5) + hash + c; /* hash * 33 + c */
    }
    return hash % tableSize;
}
```

5.7 repl.h

```
#include "stack.h"
#include <string>

#include "hashtable.h"

using namespace std;

class REPL {
public:
    REPL();

private:
    // REPL functions
    string prompt();
    void eval(string& expression);

    // Operations
    void arithmetic(string& op);
    void assignment();
    void mathFunc(string& func);
    void del();
    void search();

    // Data members
    Stack* postfixStack;
    HashTable* symbolTable;

    // Helper functions
    bool str_is_number(string& str);
    bool str_is_alpha(string& str);
    double top_to_num();
    bool exit = false;
};
```

5.8 repl.cpp

```
#include "repl.h"
#include <cmath>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>

REPL::REPL()
{
    postfixStack = new Stack(20);
    symbolTable = new HashTable(10);

    while (!exit) {
        string expression;
        expression = prompt();
        eval(expression);
    }
}

////////////////////
// REPL FUNCTIONS //
///////////////////

// Shows a prompt and gets the user input
string REPL::prompt()
{
    string input;
    cout << "> ";
    getline(cin, input);
    return input;
}

// Evaluates the expression
/*
 * Pseudocode

  EVAL(expression)
    Tokenize expression using space as delimiter
    for each token In expression
        if token is an arithmetic operator
```

```
            ARITHMETIC(token)
        elseif token is a mathematical function
            MATHFUNC()
        elseif token is " ="
            ASSIGNMENT()
        elseif token is " DEL"
            DEL()
        elseif token is "SCH"
            SEARCH()
        elseif token is numeric or alphabetical
            Push token to postfixStack
        else PRINT(" Invalid character")
*/
void REPL::eval(string& expression)
{
    // Print symbol table
    if (expression == "print") {
        symbolTable->printTable();
        return;
    }

    // Exit REPL
    if (expression == "exit") {
        exit = true;
        return;
    }

    // Clear stack
    if (expression == "clear") {
        postfixStack->clear();
        postfixStack->print();
        return;
    }

    istringstream iss(expression);
    string token;
    // Tokenize and loop through tokens
    while (getline(iss, token, ' ')) {

        // If token is an operator
        if (token == "+" || token == "-" || token == "*" || token == "/") {
            arithmetic(token);
        }

        // If token is a math function
        else if (token == "sqrt" || token == "sin" || \\
            token == "cos" || token == "tan" || token == "log") {
            mathFunc(token);
        }

        // Token is an assignment operator
        else if (token == "=") {
            assignment();
        }

        // If token is a delete operator
        else if (token == "DEL") {
            del();
        }

        // If token is a search operator
        else if (token == "?") {
            search();
        }

        // If token is a number or alphabet, push to postfixStack
        else if (str_is_number(token) || str_is_alpha(token)) {
            postfixStack->push(token);
        }
        // Unknown token
        else {
            cout << "Invalid character: " << token << endl;
        }
    }
    postfixStack->print();
}

////////////////
// OPERATIONS //
////////////////

// Performs arithmetic operations using the top 2 elements of the stack \\
    based on the input operator
/*
* Pseudocode

  ARITHMETIC(op)
    Pop postfixStack twice and store values in numi and num2
    if op ='+'
        result = num] + num2
    if op = '-'
        result = num1 - num2
    if op = '*>
        result = num] * num2
    if op ='/'
        result = num1 ÷ num2
        if num2 = 0
            error
    PUSH result to postfixStack

*/
void REPL::arithmetic(string& op)
{
    // First operand
    double num1 = top_to_num();
    postfixStack->pop();

    // Second operand
    double num2 = top_to_num();
    postfixStack->pop();

    double result;
    switch (op[0]) {
    case '+':
        result = num1 + num2;
        break;
    case '-':
        result = num1 - num2;
        break;
    case '*':
        result = num1 * num2;
        break;
    case '/':
        if (num2 == '0') {
            throw std::runtime_error("Division by zero");
        }
        result = num1 / num2;
        break;
    default:
        throw std::invalid_argument("Invalid operator");
    }

    cout << num1 << op << num2 << " = " << result << endl;
    postfixStack->push(to_string(result));
}

// Assigns a value to a variable using the top 2 elements of the stack
/*
```

```
* Pseudocode

  ASSIGNMENT()
    Insert (var, num) into symbolTable
    Pop postfixStack twice and store values in ell and el2
    if ell and el2 are both alphabetical or both numeric
        error
    Insert el2 into symbolTable with key ell or vice versa based on type

*/
void REPL::assignment()
{
    // Get top two elements
    string el1 = postfixStack->top();
    postfixStack->pop();
    string el2 = postfixStack->top();
    postfixStack->pop();

    // Check if one is a number and the other is a variable
    if ((str_is_alpha(el1) && str_is_alpha(el2)) || \\
            (str_is_number(el1) && str_is_number(el2))) {
        throw std::invalid_argument("Invalid assignment");
        return;
    }

    double num;
    string var;

    // Determine which is the number and which is the variable
    if (str_is_number(el1)) {
        num = stod(el1);
        var = el2;
    } else {
        num = stod(el2);
        var = el1;
    }

    // Insert into hashtable
    symbolTable->insert(var, num);
}

void REPL::mathFunc(string& func)
{
    // Get top of stack
    double num = top_to_num();
    postfixStack->pop();
    // Perform math function
    double result;
    if (func == "sqrt") {
        result = sqrt(num);
    } else if (func == "sin") {
        result = sin(num);
    } else if (func == "cos") {
        result = cos(num);
    } else if (func == "tan") {
        result = tan(num);
    } else if (func == "log") {
        result = log(num);
    } else {
        throw std::invalid_argument("Invalid math function");
    }

    cout << func << "(" << num << ") = " << result << endl;
    postfixStack->push(to_string(result));
}

// Deletes a symbol from the symbol table
void REPL::del()
{
    // Get the key from top of the stack
    string& key = postfixStack->top();
    postfixStack->pop();

    // Check if key is a valid variable
    if (!str_is_alpha(key)) {
        throw std::invalid_argument("Invalid variable");
    }

    // Remove the key from the symbol table
    symbolTable->remove(key);
}

// Searches and prints the value of a variable in the symbol table
void REPL::search()
{
    // Get the key from top of the stack
    string& key = postfixStack->top();

    // Check if key is a valid variable
    if (!str_is_alpha(key)) {
        throw std::invalid_argument("Invalid variable");
    }
    // Search for the key in the symbol table
    double value = symbolTable->search(key);
    cout << key << " = " << value << endl;
    postfixStack->pop();
}

//////////////////////
// Helper functions //
//////////////////////

// Checks if a string is a number
bool REPL::str_is_number(string& str)
{
    for (char& c : str) {
        if (!isdigit(c)) {
            if (c == '.' || c == '-')
                continue;
            return false;
        }
    }
    return true;
}

// Checks if a string is alphabetic
bool REPL::str_is_alpha(string& str)
{
    for (char& c : str) {
        if (!isalpha(c)) {
            return false;
        }
    }
    return true;
}

// Function to convert top of stack to a decimal number, as it can \\
        either be a number or a variable
double REPL::top_to_num()
{
    // Get top of stack
    string& postfixStackTop = postfixStack->top();

    // If top of stack is a number, return it
    if (str_is_number(postfixStackTop)) {
        return stod(postfixStackTop);
    }
    // If top of stack is a variable, return its value
    if (str_is_alpha(postfixStackTop)) {
        return symbolTable->search(postfixStackTop);
    }

    return 0;
}
```

6 Improvements

One notable flaw in my implementation is the absence of robust error handling. When encountering unexpected input, such as ”2 4 =”, the program throws an exception and terminates. To enhance this, try-catch blocks can be introduced in the caller functions to manage errors gracefully. This improvement would require accounting for all possible user inputs to ensure comprehensive error handling.

Another shortcoming of my implementation is the lack of features commonly found in CLI applications. Enhancements such as input history, auto-complete, and man pages would significantly improve the user experience. These features can be implemented using libraries such as ‘readline‘ for input history and auto-complete, and ‘libman‘ or ‘cppman‘ for creating manual pages.