Assignment 5: Compiler for Simple Language of Expressions

COL106: Data Structures and Algorithms, Semester-I 2023–2024

Submission Deadline: 11th October 5:00 PM

Honor Code

Following is the Honor Code for this assignment.

- Copying programming code in whole or in part from another or allowing your own code (in whole or in part) to be used by another in an assignment meant to be done individually is a serious offence.
- The use of publicly available codes on the internet is strictly not allowed for this assignment.
- The use of ChatGPT and other AI tools is strictly forbidden for this assignment. If any student has been found guilty, it will result into disciplinary action.
- Collaboration with any other person or team would be construed as academic misconduct.
- Outsourcing the assignment to another person or "service" is a very serious academic offence and could result in disciplinary action.
- Sharing of passwords and login ids is explicitly disallowed in this Institute and any instance of it is academic misconduct. Such sharing only compromises your own privacy and the security of our Dept./Institute network.
- Please ensure that your assignment directories and files are well-protected against copying by others. Deliberate or inadvertent copying of code will result in penalty for all parties who have "highly similar" code. Note that all the files of an assignment will be screened manually or by a program for similarity before being evaluated. In case such similarities are found, the origin as well as destination of the code will be held equally responsible (as the software cannot distinguish between the two).

Introduction

A *compiler* is a program which takes an input program in a source language S and outputs an equivalent program in a target language `Targ`. The source language is typically a language like C or C++, and the target language is usually machine language, i.e. code which can be executed by the machine. For instance, you may have used G++ or Clang to compile the code you've written as a part of this course.

You have already created an expression evaluator for a simple language of arithmetic expressions, E++, in Assignment 4. In this part, you will improve your implementation of make it more efficient. Further, you have the additional task to create a compiler this language of expressions E++, which does not evaluate the program, but simply generates code for it which will run on a machine.

For evaluation, some test cases will be made available to you to help you debug your program but all the "evaluation test cases" will not be made available to you. You need to ensure that your program works correctly **on all the inputs**, not just on the test case inputs.

1 Things relevant to Assignment 4

1.1 Old Headers

Classes `UnlimitedInt` and `UnlimitedRational` are not needed in this assignment (we will be working with *ints* only). Class `Evaluator` is now just `Parser`, no *eval()* function. Also there is a support for two new statements in the syntax of E++ : **delete** and **return**.

1.2 Delete statement

In addition to normal E++ expressions, there is a new statement called **delete**.

1.2.1 Syntax

del := variable_name

In the form of tokens array: [”del”,”:=”,”variable_name”]
The *delete* statements will be semantically correct, that is the variable must have been assigned a value before.

1.2.2 Modifications to ExprTreeNode and Parser

The expression tree for the delete statement would be a simple tree consisting of three nodes. Root node as ”:=” with the left node of a new *type* **“DEL”** and right node of *type* **“VAR”**, storing the variable name.

1.3 Return statement

Every set of E++ expressions will have the last statement of type *return*. The semantics of return would be to stop the execution of the program, and return the value of the expression after return.

1.3.1 Syntax

ret := E++ expression

Eg : ret := (a x (b + c)), in tokenized form : `["ret", ":=", "(", "a", "x", "(", "b", "+", "c", ")", ")"]`
The *ret* node in the expression tree will have the type ”**RET**”

1.4 Class Parser (parser.h)

1.4.1 Parameters

- **int last_deleted**: An optional parameter to maintain the address of the last deleted symbol
- **vector<ExprTreeNode*> expr_trees**: Stores the roots of the parsed expressions.
- **SymbolTable* symtable**: New modified symbol table (See section 2 for more details).

1.4.2 Functions to implement (parser.cpp)

- **parse(vector<string> expression)**: Read an input vector of tokenized *expression*, parses it and converts it to a parse tree and push the root into *expr_trees*. Also makes necessary changes to the *symtable* depending upon the *expression* (if it’s a variable assignment then inserts the symbol, for a delete statement it deletes the symbol and does nothing to the symtable for the return statement).

Note

- You can assign any garbage value to the nodes of type "RET" or "DEL" or ":=", their values don't matter like other operator nodes. For nodes of type *VAR* and *VAL* we would only check parameters *id* and *num* respectively.
- Immutability no longer holds now, that is a variable can be reassigned a value, it's also not necessary a variable needs to be deleted before reassigning.

2 Improving the Expression Evaluator (60 Marks)

In Assignment 4, you implemented an expression evaluator for the language E++. Recall that for storing the values of variables, you had created a symbol table using Binary Search Trees. What if the BST used to implement the symbol table is unbalanced? The lookup could then be $O(n)$ in the worst case, where n is the number of variables in the program. In turn, n can be as large as the code - which can end up being a huge overhead.

Your task is to convert the BSTs you implemented for Assignment 4 into *AVL Trees*. Note that now that each node will no longer store a value, since we are not *evaluating*, but *compiling* the code. Instead, it will store an address, as described below. (This decouples it from the *Evaluator* you implemented in Assignment 4).

Note: Future assignments may involve the the use of AVL trees, thus it is strongly advised to do this part sincerely.

2.1 The AVL Tree Node class (symnode.h)

2.1.1 Parameters

- **string key**: Stores the key (variable_name).
- **int height**: Stores the height of the node in the SymbolTable.
- **int address**: Memory address (or index) allocated to the symbol while compiling by the compiler. See Section 3.4 for more details. Default value is set to -1, implying no mapping.
- **SymNode* par**: Pointer to the parent of the node in the symbol table. If the node is root then set it to NULL.
- **SymNode* left**: Pointer to the left child of the node in the symbol table. If it is leaf then set it to NULL.
- **SymNode* right**: Pointer to the right child of the node in the symbol table. If it is leaf then set it to NULL.

2.2 Functions (OPTIONAL) to implement (in symnode.cpp)

Any unbalanced tree can be balanced by using simple 4 rotations, see this for reference. You are free to implement and use these as your helper functions in the symbol table or not use them at all. If you are not going to implement any of the given optional functions, then do not delete their definitions, just write return *NULL* in those.

- **LeftLeftRotation()**: Performs the single left rotation on the node and returns pointer to the new root. The Case 1 in the class notes (Insert in AVL tree).
- **RightRightRotation()**: Performs the single right rotation on the node and returns pointer to the new root. The Case 2 in the class notes (Insert in AVL tree).

- **LeftRightRotation()**: Performs a left and then right rotation on the node and returns pointer to the new root. The Case 3 in the class notes (Insert in AVL tree).
- **RightLeftRotation()**: Performs a right and then left rotation on the node and returns pointer to the new root. The Case 4 in the class notes (Insert in AVL tree).

2.3 Symbol Table (symtable.h)

There is addition of a new function: **assign_address(string k,int idx)** which will assign a memory index to the input symbol while compiling (See Section 3.4 for more details). Other functions remain same, implement all of them, instructions similar to Assignment 4. Note that now *search*, *insert*, *remove* along with *assign_address* should take $\log(n)$ time.
Important: After each *insert()* and *delete()* function call, the symbol table must be an *AVL Tree*.

Things to note

- In search, if the key is not found then return NULL.
- For remove, if the key doesn't exist then it shouldn't modify the table.
- It can be assumed that insert is called for distinct set of keys only.

3 Code Generator (40 Marks)

In this part, you will actually generate code in the target language `Targ`. We will assume that `Targ` will execute on a *stack machine*, similar to the post-fix evaluator you created in Assignment 2.

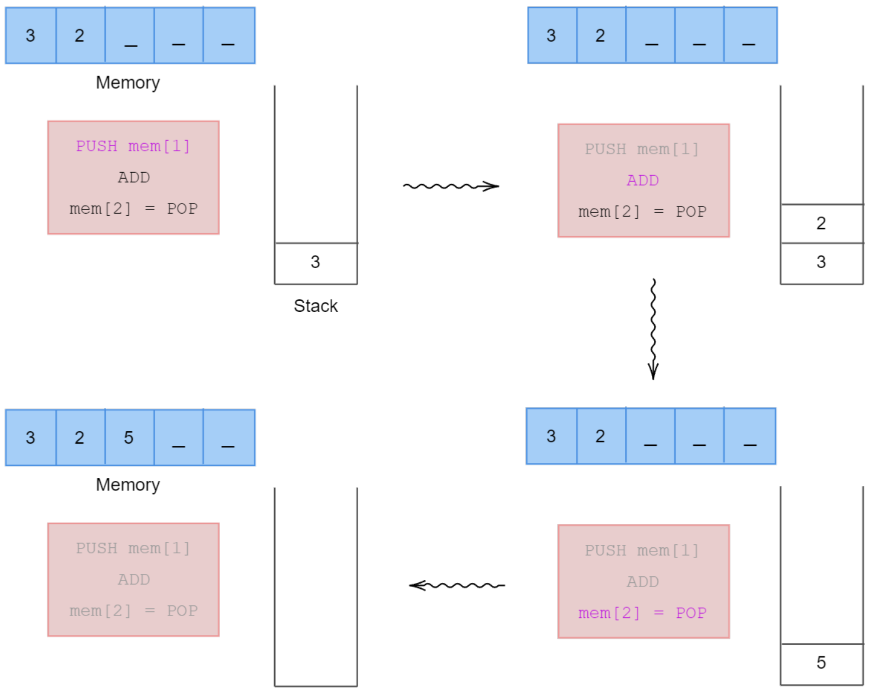

Figure 1: An example of running a sequence of instructions on the stack machine.

3.1 Stack Machine

The code in the target language you generate will assume the following architecture of the stack machine:

- **Stack:** An infinite memory, with operations providing access to only the top element(s). It is used to store temporary values in the evaluation of expressions.
- **Memory:** Access to a limited indexable (random-access) memory.

Essentially, the machine will go through the program in the target language `Targ`, perform some operations on the operands in the stack, and finally store the results in the memory.

Note: We are not asking you to implement this stack machine. You just need to generate code for your parse tree in the `Targ` language assuming it will execute on a stack machine as described.

3.2 Targ Syntax

The syntax of the language `Targ` is given in Table 3.2.

Syntax	
Command	Description
`PUSH x`	Push an integer x onto the stack
`PUSH mem[i]`	Push the value in the ith memory location onto the stack
`mem[i] = POP`	Pop the value at the top of the stack and store it in the ith memory location
`DEL = mem[i]`	Delete the value stored at ith memory location and free it for future use
`ADD`	Pop and add the top two elements of the stack and push the result back onto the stack
`SUB`	Pop the top two elements from the stack. Subtract the second from the top element from the top element of the stack and push the result back onto the stack
`MUL`	Pop two elements from the stack. Multiply them and push the result back onto the stack
`DIV`	Pop two elements from the stack. Divide the top element by the second from the top element. Push the *floor* of the result, i.e., $\lfloor res \rfloor$, back onto the stack. If division by zero is attempted, the result is set to $NULL$
`RET = POP`	Push the value on the top of the stack onto a special memory location named `ret`. The execution of the program ends after this operation.

Note: Any operation performed with `NULL` as one of the operands will result in `NULL` as the output.

3.3 Generating the Code

To generate the code in the target language, you will need to recursively iterate over the parse tree you created in Part 2. An example of this is given below:

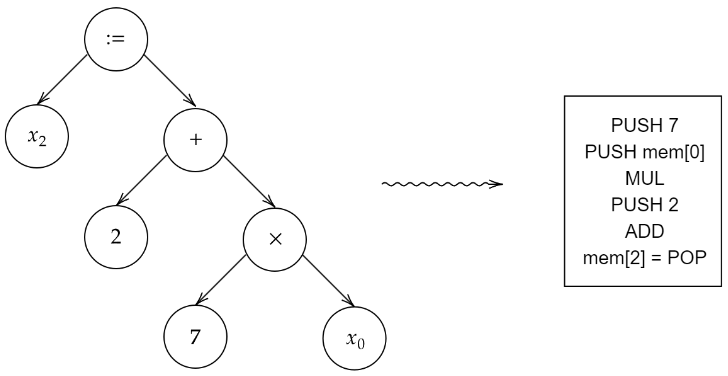

Figure 2: An example of converting a Parse Tree to code

3.4 Handling memory allocation of variables

When we do variable assignment in the E++ expressions, we don't assign the variables a specific memory address, instead this part is handled by the compiler itself. We will maintain a vector (in a stack fashion) of available free memory indexes, delete or insert the memory index from the end if a new variable is assigned a value or it's deleted respectively. We also assume **0-based indexing**.

Note: This will make the memory address allocation random. To check the correctness of the program, we will simulate it on a "stack machine". That is, we won't check the memory address and symbol mapping, but only the value returned. See Bonus section for more details.

3.5 Class EPPCompiler (eppcompiler.h)

3.5.1 Parameters

- ***Parser* targ**: The main object which will store the symbol table and parsed expressions tree nodes.
- **int memory_size**: Size of the *indexable memory* provided to map the symbols. Set during the construction and not changed afterwards. It can be assumed that the given size of the indexable memory is always sufficient for the input set of E++ expressions
- **string output_file**: The name of the output file in which generated `Targ` code needs to be dumped.
- **vector<int> mem_loc**: A vector to maintain the available memory locations.
- **MinHeap least_mem_loc**: A min-heap to maintain the least available memory indexes (See Bonus Section). You can ignore this parameter if you are not attempting the Bonus part.

3.5.2 Functions to Implement (in eppcompiler.cpp)

- **compile(vector<vector<string>> code)**: Parses each input tokenized expression, assigns a memory mapping if the expression is a variable assignment or frees up the memory address if it's a delete statement and finally compiles the *code* i.e generates and writes all the `targ` commands to the *output_file*.
- **generate_targ_commands()**: Compiles the last tree node in the *Parser* and generates `Targ` commands for it and returns them.
- **write_to_file(vector<string> commands)**: A helper function to write the given set of *commands* to the *output_file*.

We will call the *compile(code)* function once containing all the expressions in tokenized form. And then run the code in the *output_file* on the *stack_machine* to check the validity of the *E++Compiler*.

3.5.3 Writing to file

The file will be parsed line by line, strictly following the `Targ` syntax. All commands will be run in sequential manner on the stack machine. Empty lines will be ignored in the file. The library *fstream* is included in the header *eppcompiler.h* for your help in writing to the file. We are providing you with a mechanism to check if the file is in the accordance of `Targ` syntax or not. See the Section 5.3 for more details.

4 Bonus (20 Marks)

Rather having a random memory-variable mapping while compiling the code, we will now map the variable to the **least** available memory address (or index). For this we will use a min-heap.

4.1 Class HeapNode (heapnode.h)

4.1.1 Parameters

- **HeapNode* par**: Pointer to the parent of the node in the min-heap. Set to NULL if the node is root.
- **HeapNode* left**: Pointer to the left child of the node. Set to NULL if the child is a leaf.
- **HeapNode* right**: Pointer to the right child of the node. Set to NULL if the child is a leaf.

4.2 Class MinHeap (minheap.h)

4.2.1 Parameters

- **int size**: Stores the size of the min-heap.
- **HeapNode* root**: Pointer to the root of the min-heap.

4.2.2 Functions to implement (in minheap.cpp)

- **push_heap(int num)**: Pushes the element *num* into the heap.
- **get_min()**: Returns the minimum element in the heap.
- **pop()**: Deletes the minimum element (which is at root) from the heap.

We will check the memory-variable mapping in the stack machine to check the validity for the min-heap. If you are doing Bonus section then use the parameter **least_mem_loc** rather **mem_loc** in your E++Compiler for mapping.

5 Starter Code

The starter code can be downloaded from here, or in the zip format from here.

5.1 Directory Structure

- **bin**: Bin folder to store the compiled binaries.
- **include**: Maintains all the header files needed for the assignment.
- **samples**: Some samples of targ and e++ files provided for your reference.
- **src**: Contains the required .cpp files, in which you need to make all the implementations.
- **tests**: This is only for testing purposes, do not make changes to this folder.

5.2 Building the Compiler

From the main parent directory run the following command to build the compiler.

```
make
```

If the build process is successfull then a binary file named `e++` would have been generated in the *bin* folder. Note: For windows systems, it won't work, you need to install GNU `make` command first.

5.3 Running the compiler

You can test your compiler by running it on a set of E++ expressions. Write the set of **valid** E++ expressions in a file (suppose it's a.txt), in the tokenized form i.e between each token there should be an at least one space character separating the two tokens. Also note that the empty lines are ignored. See the *samples* folder for examples.
To compile the set of `E++` expressions in the file, place your built compiler in the same directory as the a.txt and run the command (eg is for MacOS/Linux):

```
./e++ a.txt
```

If there are no errors in the process, then a file named `targ.txt`, would be generated containing the targ commands for a.txt .

5.4 Testing the Targ Syntax

In the directory `tests/targ` there are binaries for each OS. Copy the respective binary file to the same folder in which your filename.txt is there, containing the `Targ` commands. Run the following command (eg is for MacOS/Linux):

```
./targ++ filename.txt
```

- If you get the permission denied error, then run "`sudo chmod +x targ++`" only once to make system trust the executable.
- For MacOS, if the issue persists still or you get unidentified developer error then open the binary by right-clicking on it and now you will be able to run the binary normally by `./targ++`".
- **Note**: The binaries only checks that the commands are syntactically correct or not. Neither their semantics nor their logical output is being checked.

5.5 Tester

Note: The tester is designed for Unix based systems only (i.e Linux and MacOS), it is strongly advised for Windows users to either install WSL, or dual-boot or remotely connect to their CSC machines over SSH in order to get hands on a Unix OS. If none of these options are feasible, students may wait for the autograder on Gradescope.

5.5.1 Building and Running the Tester

Run the following command from the parent directory:

```
make tester
```

If there are no errors then a binary file named *tester* would have been generated in the bin folder. Change your directory to the bin folder and run the command :

```
./tester
```

Follow rest of the instructions on your screen in order to run the tests.

5.5.2 Things to note

- The tester is dependent on your .cpp files, so you may need to build it again and again after every change in them.
- Do not enclose the filename in quotes when asked in the tester.
- The current tests are very basic in nature and do not check many of the edge cases and stress testing of your submissions.
- If you find any bugs in the tester, please feel free to report them on piazza.

6 Submission Instructions

Please read these instructions **very carefully**. You are required to submit all files present in the ***src*** folder on Gradescope, even if you are not attempting the Bonus section still submit the original minheap.cpp file too, and please **do not** submit anything else.

Please note the submission deadline is **11th October, 5 PM**, and not midnight.

7 Points to Note

1. Do not set any tree pointer parameter to nullptr, use NULL instead.
2. For each part, read the instructions carefully and use only the data structure specified.
3. Follow good memory management practices. We will stress test your submissions. There should be no memory leak, otherwise your code may fail on the evaluation test cases.