TRANSPILER

Transpiler to convert SuperStack! into IITK Traveller language

Hall 3 (ARYANS)

Introduction

A transpiler (or transcompiler) is a source-to-source code translator, which converts a high-level programming language into another one, for example, converting C++ code into its equivalent in Python. An esoteric language is a programming language that tests the boundaries of computer programming. It is in a sense, a language written either for fun or to solve a very specific purpose. Some examples are the famous Brainf**ck, SuperStack! and our very own, IITK Traveller.

SuperStack! uses push, pop, traversal and other functions on a stack to solve problems. IITK traveler uses a variety of operations on a stack with 3 pointers as its backbone. Both languages are indirectly inspired by Brainf**k, which uses 8 operations on a stack and is Turing Complete. Hence, both of these are also Turing complete languages.

In this report, we present the transpiler made by the team of Hall3 (Aryans) which converts SuperStack! code into IITK traveller code.

1. The Problem Statement (PS)

Each team must create a program (transpiler) that converts any given line of SuperStack! esolang to IITK Traveller code. Additional functionalities such as bitwise operators like not, and, or, xor etc fetch additional points. IIT Traveller uses an infinite array of integers and 3 pointers (mem_1, mem_2, mem_3) for its working. mem[mem_1] denotes the value pointed to by mem_1. Input, output and certain functionalities are specific to mem_1, mem_2 while mem_3 can store the result of some operator on the values pointed by these 2 pointers. Basically, mem_3 acts as a temporary variable to help us carry out some operations to code the solution.

2. Our Construct

To use a stack the memory tape in IITK traveler has been modified, such that mem_3 points to somewhere near the beginning of the stack, followed by an **EOS (End of String)**. mem_2 points to the last element in the stack with a value, and mem_1 points to the element before it. mem_3 is located 2 spaces after the first element of the stack, the extra elements used to store more than one value, which might be temporarily required or a garbage value. mem_3 does not move forward and is kept stationary in most functionalities implemented.

While adding a new element, first mem_2 is moved forward, the input value is put in mem[mem_2] and then mem_1 is moved forward by 1. This construct greatly helps in implementation as the working space is bound between an EOS and mem[mem_2], while mem[mem_3] helps as a temporary variable and mem_1, mem_2 act as a prev, next pointer for a stack.

In the initial state of the stack, mem_1 and mem_3 point to the 3rd element of the stack, followed by an EOS. mem_2 points to the 4th element. This is done so because when adding an element to the stack, the first mem_2 is moved forward then an input is taken.

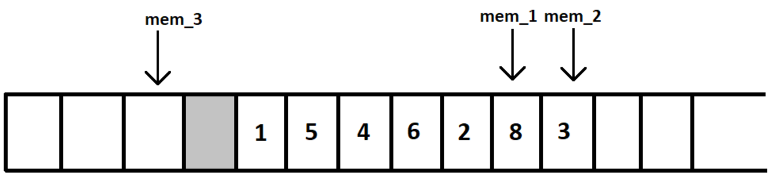


Figure 1 – An example of our construct storing the stack [1,5,4,6,2,8,3]

Implementation of Commands

In this section, we give a brief overview of how each command of SuperStack! was implemented in IITK Traveller. The commands were categorized as basic and advanced according to the PS. The advanced commands included an, or, xor, not, nand random, while the rest came under basic commands.

Initialising the stack construct

mem_1, mem_2 are moved forward by 2 spaces each. mem[mem_2] is assigned an EOS. Thus, we finally get mem[mem_3] as the 3rd element, followed by an mem[mem_1] (which is initially assigned to an EOS) as the 4th and mem[mem_2] as the 5th element of the stack.

1. I/O (Input/Output)

(i) Input/push operation:
We simply move mem_1 and mem_2 by 1 space, then take the input and put it in mem[mem_2].

(ii) Output/print operation:
The value of mem[mem_2], which is the topmost element of the stack, is printed. Then mem[mem_2] is set to 0 (default value). Finally mem_1 and mem_2 is moved back by 1 space.

(iii) Input ascii operation:
Here, we first take the string input through airstrip_land_2, then move the mem_2 pointer to its end and place an EOS (let be 1st EOS) after the string, then store the first element of the original stack(before input) at the next place (sfter above EOS). Then we move mem_1 and mem_2 ahead till mem_1 finds the above EOS, meanwhile transferring the contents stored by mem_1 in mem_2, hence placing a copy of the input string after the actual input string in the memory tape. Then we plant EOS (be 2nd EOS) at the end of this copy using pronite_2. Then we send mem_1 and mem_2 back till mem_2 reaches the 1st EOS, so mem_1 reaches the first element of the original input string. Then we move mem_2 ahead so it reaches the end of the copy of the inpt string. Then, we move mem_1 ahead and mem_2 back while transferring the contents of mem_2 to mem_1, resulting in the reversal of the input array(so as to store it in the format the Super-Stack stores it). Then we remove the EOS flags after the reversed strings using mem_2 to reach our initita contstruct again.

(iv) Output ascii operation:
The ASCII value corresponding to the value stored at mem[mem_1] is printed. Then, mem1, mem2 and mem3 are moved behind by 1 space, hence the last element is popped out of the stack.

(v) pop operation:
mem[mem_2] = 0 is done (as 0 is the default value for elements of the stack). Then mem_1, mem_2 are moved by 1 space behind.

(iv) debug (outputs entire stack):
mem_2 is moved forward, then an EOS is inserted at mem[mem_2]. This will help in knowing where to end printing the elements of the stack. mem_1 is continuously moved behind by 1 space till an EOS is occured. Then, it is moved forward by 1 space and elements are printed till an EOS is met. Finally, mem_1 is moved behind by 2 spaces.

2. Arithmetic Operators

(i) add/sub and mul/div operation:
mem[mem_3] is assigned the result of the arithmetic operation on mem[mem_1] and mem[mem_2], then mem_1 and mem_2 are moved behind by 1 space. This is equivalent to poping 2 numbers and pushing the result in the stack. The code for all 4 operations is same except where the operation is done. (example- hall2 is used for add while hall_3 is used for *mul*)

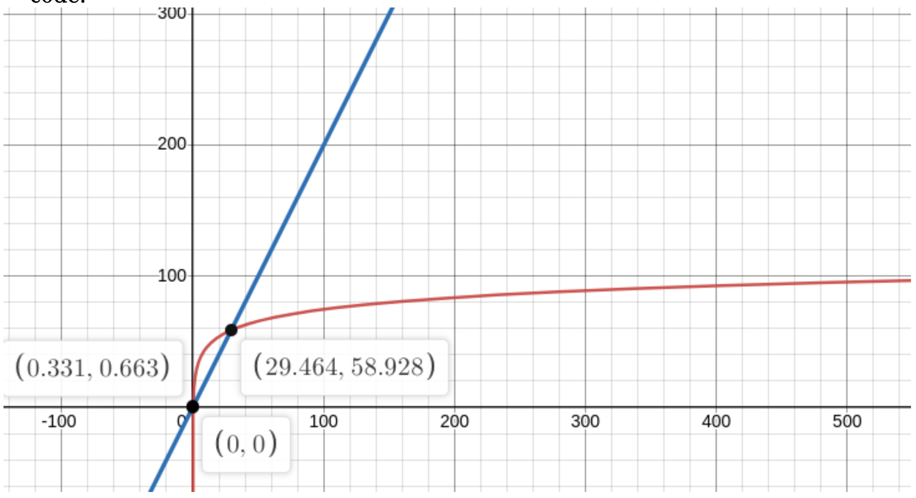
(ii) mod operation:
Here, we need to pop mem[mem_1] (say x), mem[mem_2] (say y) and push the result of x mod y into the stack. To do this, we first use hall12 to store the integer part of x/y in mem[mem_3]. mem_1, mem_2 are moved by 1 space and the new mem[mem_2] stores the value of mem[mem_3]. Then, mem[mem_3] is assigned mem[mem_1] * mem[mem_2] (= int(x/y) * y). mem[mem_2] = mem[mem_3] is done, mem_1 is moved back 1 spaces and made to store the value of mem[mem_1] - mem[mem_2], which is the desired result. Finally, mem_1 is moved back 1 space and mem_2 2 spaces.

(iii) 123 (pushing a number onto the stack)

Since IITK-Traveller doesn't provide any special operation to assign a number, hence to implement it, we would require to create the number from scratch (from 0! using the algebraic operations supported by IITK Traveller). Hence we used two different implementations for it :

BRUTE FORCE - We increase the created number by 1 until it reaches the original number, checking the above condition in the transpiler after every increase. The time complexity of this solution is $\mathcal{O}(n)$. BITWISE IMPLEMENTATION - Here, we extract the bits of the number(say 123) in the transpiler itself. Using mem_1, mem_2, and mem_3, we create the number by multiplying the created number by 2 whenever a bit is encountered and adding 1 to it whenever the encountered bit is 1. This time complexity of this solution is $\mathcal{O}(\log(n))$.

For a number n, while brute-force implementation takes $2{\cdot}n$ lines, the bitwise implementation takes roughly $9{\cdot}(\log_2 n){+}15$ lines. Hence comparing the above two functions in a graph plotter, we get for numbers between -30 to 30, brute force implementation gives the optimized number of lines in transpile code. In contrast, for the other range of numbers, the bitwise implementation is much better regarding the number of lines of code.



3. Stack/Array Operations

(i) swap operation:

mem[mem_3] is assigned the sum of mem[mem_1] (say x) and mem[mem_2] (say y), then mem[mem_2] stores the sum. Then, mem[mem_3] stores the new difference, i.e., x. mem[mem_2] then gets this value. Finally, mem[mem_1] gets the value of the new difference (y) by using mem[mem_3] as a temporary variable.

(ii) cycle operation:

Let the top element of the stack be x. An EOS is added next to mem[mem_2]. mem[mem_3] then stores x and mem_3 moves back by 1 space. A loop is run in which the condition is to check whether mem[mem_1] contains an EOS or not.

When the function is **false**, mem[mem_3] = mem[mem_1] is done, then mem[mem_2] = mem[mem_3] (= mem[mem_1]) is done. Then, mem_1 and mem_2 are pushed 1 space behind. When the **true** condition is hit, mem_3 is moved forward by 1 space and mem[mem_2] gets the value of x.

Then a 2nd loop starts, which checks whether mem_2 points to an EOS or not. While the condition is **false**, mem_1 and mem_2 are moved forward by 1 space each. When the **true** condition is hit, mem[mem_2] contains an EOS, so it is set back to the default value of 0 and mem_1, mem_2 are moved behind by 1 space.

(iii) rcycle operation:

It is quite similar to the 'cycle' operation. Let the bottom element be y. First, an EOS is added next to mem[mem_2]. A condition is kept to check whether mem_1 points to an EOS. When the condition is **false**, mem_1 and mem_2 are moving backwards. When the condition becomes **true**, mem[mem_3] gets the value of mem[mem_2] (which is y, as mem_1 points to the EOS next to mem_3 and mem_2 is 1 space after mem_1) and mem_3 moves back 1 space.

The 2nd loop's condition checks whether mem_2 points to an EOS. When it is **false**, mem_1, mem_2 are moved forward by 1 space. Then, mem[mem_3] = mem[mem_2] and then mem[mem_1] = mem[mem_3] is done (thus the elements of the stack keep shifting 1 space backwards). When the condition becomes **true**, mem[mem_3], mem[mem_2] is set back to the default value 0, mem_3 is moved forward by 1 space while mem_1, mem_2 are moved back 1 space. Finally mem[mem_2] gets the value of mem[mem_3] (which is y) and hence the bottom element reaches the top of the stack.

(iv) dup operation:
mem_1, mem_2 are moved forward by 1 space. Then, mem_1 value is copied to mem_2 using mem_3 as a temporary variable. Thus the last element of the stack is duplicated.

(v) rev operation:
Here, we move the mem_1 pointer to the base of the stack(where the first element is stored), then move the mem_1 and mem_2 pointers ahead and hence copy the whole stack ahead of the original stack. Then, we send the mem_1 pointer back to the first element of the original stack. Then, moving mem_1 ahead and mem_2 back correspondingly, we reverse the original stack by transferring every element encountered by mem[mem_2] to mem[mem_1]. All this moving of pointers is controlled by defining an EOS flag at the end of the original stack and the EOS flag at the start of the stack.

4. Bitwise Operations

We have followed a similar basic construct for almost all the bitwise operations. The operations **and, or, not** and **nand** have very similar implementations. The **not** operation has a slightly different implementation since it acts on a single operand. For **and, or, xor** and **nand** operations, we first generate 2^{30} using the locations *e_shop, oat_stairs and hall_3* by doing some basic manipulations. Then, we decrease the power of 2 by 1 in every loop iteration. Then, we check if the current power of 2 is smaller than or equal to any or both of the numbers. If the power of 2 is smaller than any of the numbers, it means that the corresponding bit in that number is 1.

For these comparisons, we have used the landmark *lecture_hall* for flow control. The condition for the outermost loop is the equality of two numbers, if the two numbers are equal, we do the corresponding operation (Ex. For **and**, we add that to our answer, but for **xor**, we don't add it to the answer) and then the process is finished. If, the two numbers are not equal, then we do the following operations correspondingly in each iteration.

(i) and operation:
We check if the current power of 2 is less than or equal to both the numbers meaning both the bits are set, then we add this power of 2 to the dummy variable at the location of mem_3 that is storing the final result. If either of the bit was unset we will not add to mem_3.

(ii) or operation:
We check if the current power of 2 is less than or equal to either of the numbers meaning one of the bits is set, then we add this power of 2 to the dummy variable at the location of mem_3 that is storing the final result. If none of the bits were set we will not add to mem_3.

(iii) xor operation:
We check if the current power of 2 is less than or equal to exactly one of the numbers, if it was then this means the corresponding bit of exactly one of those numbers was set which means xor operation will give one so we add this power of 2 to the dummy variable at the location of mem_3 that is storing the final result. Otherwise, we would not add.

(iv) nand operation:
We check if the current power of 2 is less than or equal to both the numbers meaning both the bits are set, then we do not do anything. But if either of the bit was unset we will add this power of 2 to the dummy variable at the location of mem_3 that is storing the final result.

Then the number that was just greater than or equal to the power of 2 will be subtracted by this 2's power to remove the corresponding bit from that number. We proceed to the lecture_hall_eq landmark of the final loop to check if we can exit it or still need to loop again.

(v) not operation:

In the implementation of not operation, we first generate $2^{31}-1$ using the above construct, by adding numbers from $2^0, 2^1, \cdots \text{to} 2^{30}$.

5. Conditional Operations

The simple conditionals can be easily implemented but we had to devise a way to handle nested conditionals. After checking the condition, we will go to either of that conditional's false or true landmarks. So we need to store the cond value corresponding to each conditional landmark to correctly go to the innermost landmark in the nest of ifs. To do this we maintain a stack of the conditions with the cond value corresponding to the innermost if at the top of the stack. This is done in if_stack() function whenever we encounter an if token. We print the corresponding false landmark in the if_stack() function. When the fi token is encountered before some other tokens are present within the loop which would have increased the current cond value. To jump back to the last if statement we take out the prev_cond value from the top of the stack and jump to it. We then print the true landmark of the conditional landmark used in the corresponding if statement. After the true landmark, the code needs to jump to the cond value corresponding to the landmark just after fi statement. We do this by maintaining a max_cond value which is the maximum cond value till the correct processing (cond value corresponding to the starting of the landmarks in the fi_stack function.) So we move one more from this maxCond value to reach the next landmarks and exit the conditionals.

6. Handling quit tokens

Whenever we encounter a quit token, instead of adding a new line where we move to finish from the current cond value, we add this finish statement at the end of the previous token.

Codebase

We chose to implement our construct in C++. Since the tokens in SuperStack! are more or less independent(except the if-fi statements), we start by parsing the SuperStack! code into its tokens. Next, we implemented functions to deal with each token, consisting mostly of print statements that print our logic of those operations in an output.txt file. To ensure that none of the token's codes interferes with each other, we track the value of cond and increment it by one after each token. This isolates every token from others as we will be mostly increasing the cond value avoiding any clashes.

First, tokenize() function runs to generate all the tokens from the file passed as the argument. Next initialize_stack() always runs to initialize the stack in the way described above. Next, we iterate over all other tokens sequentially to print out the implementation of other tokens.