CMPE 230

İrem Nur Yıldırım

Başak Tepe

Advanced Calculator++ Transcompiler

Systems Programming Course Project

Submission: 01.05.2023

1. Introduction

In this project, we were expected to implement a Transcompiler for an advanced calculator that does basic arithmetic and bitwise operations using C language. During our previous project, we coded an Interpreter for a similar language using the Shunting Yard Algorithm. This time, the transcompiler was expected to take AdvCalc language as an input and produce code in LLVM IR to be executed. Our previous implementation of the Interpreter in Shunting Yard Algorithm enabled us to perform some changes in the way we evaluate the postfix expressions in the project, resulting in a considerably easy transformation from an Interpreter to a Transcompiler. Our initial approach to this problem was changing the structure of our previously coded postfix evaluator function: instead of performing actual arithmetic operations on a decreasing stack, we translated these arithmetic operations to LLVM syntax and directed the output to a file with **`fprintf`**.

2. Program Interface and Program Execution

For the activation of the program, the source code can be compiled with the “make” command in the terminal and executed afterwards with a command line argument to output an LLVM file. Executable name is ./advcalc2ir and it accepts the input file name *.adv as argument. If there are no errors encountered, the program will output a file in the format *.ll.

Later on, this .ll file can be transformed to assembly code with llc command on the terminal, and it can produce an executable with clang command. The final executable file can be run on the terminal as well.

3. Input and Output

The program takes inputs from a file and directs the output to a file as well. Input is in the format of mathematical expressions or/and assignments. The input of AdvCalc++ differs from AdvCalc in that it has 2 extra properties: division and remainder. Output is in the format of LLVM IR code. If errors are encountered during the compilation process, error messages are displayed in the terminal screen and no output file is created. The program terminates when an input file is evaluated completely: either resulting in an output with .ll file extension, or no output if the input file contains syntax errors or undefined variables in the AdvCalc++ language.

4. Program Structure

The structure followed in this project is quite similar to Project 1. It simply imitates the body of a transcompiler with the necessary subparts: the lexer and the translator.

5.1 Lexer / Tokenizer

We did not commit any changes in our previous lexer function. The lexer is responsible for cutting up meaningful pieces of information from the user's input. Later, these pieces of information, or terminals, are converted into a structure called "tokens". A token has a type and a value: **`NONE, NUM, OP,IDENT, FUNC_CALL,LPAR,RPAR,COMMA,ASSIGN,AND,OR`** make up the types of a token. On the other hand, values of a token are taken directly from the input string itself.

Token Examples

Type	Value
NUM	5
LPAR	(
COMMA	,

5.2. Parser (Postfix Generation)

We did not commit any changes in our previous parser function. Parser takes as an input the tokenized array, where every meaningful token is stored except whitespaces, and turns them into a postfix denotation. For achieving this it follows these steps: First, it creates a stack where we will store operators or left parentheses, later on we will check the top element of the stack to decide whether to append it to the postfix array or not. After creating the stack, the function starts to iterate through the token array, it directly adds each variable or number-operand- to the postfix array just because the calculation is from left to right and there is no precedence between them. If the current token is an operator or function call we check if this operator is precedent then the one we stored in the stack previously -if there is any-, if the stack has a more precedent operator waiting in the top then we add this operator to postfix. As an example for the input 3+4*5 function first adds 3 then adds + to the stack, since there is not any other one waiting and having more precedence than +. Then sees 4 and directly adds it, then sees * which also does not have a more precedent rival waiting in the stack (+ is waiting but is not more precedent) so it is also added to stack. Then the function sees 5 and directly adds it to postfix and leaves the for loop, so as we see in this example after exiting the for loop we have still some operators, in fact all the operators in this example, waiting in the stack, so we check if there is any operator remained and add all of them to postfix. Eventually, what makes this last appending from stack to postfix array operation ordered is that we add them to the stack such that they have decreasing level of precedence from top to bottom. In left and right parentheses handling this function applies two if blocks to catch left and right parentheses separately, whenever it sees a left parentheses it adds it to the stack but it is to be removed later on, when a right parentheses is encountered it adds all the operators or functions waiting in the stack - until a left parentheses is detected -to the postfix array because for loop came to the end of a term in parentheses and all the operations inside these parentheses must be added to postfix array before getting out of it.

5.4 Evaluating (Interpreter)

We did not commit any changes for our evaluation function. However, the evaluation function was our starting point for the translator function **`postfix_to_ir.`**

For evaluating the postfix array we again create a stack, but this time it consists not of tokens but integers only, we iterate through the postfix array and if we see an integer directly append it to the stack. Otherwise if the current element is an operator, then we extract the last two integers of the stack and do the current operation on them, using a postfix array to make the evaluation function easy and basic as much as possible .

5.5 **`postfix_to_ir`** (Translator)

This function is very similar to the evaluating process, except it accepts a stack containing strings instead of integers and uses pop, push methods to add new strings or discard previous strings as expected in the shunting-yard algorithm process.

At first we attempted to use the same integer stack of our previous implementation, a stack where we only manipulated indexation with an integer called **`top`**. We would increment and decrement the **`top`** to add or remove elements from the stack. As a consequence, our previous stack worked more like an array, where you override the integers.

Yet this approach was incompatible with the register number-integer value differentiation. Although our previous stack-like structure with index manipulation worked fine for integers, it did not perform well with strings.

In the new actual stack implementation, we used the string format to store elements in a stack after obtaining from the postfix array. Therefore, we needed strings that include the % sign within itself for registers not as a result of an external format specification. In other words, we wanted integers in a string to be represented as **`"int"`** and register numbers to have the format of **`"%int"`**. Another difference with the evaluating function is that it does not store the resulting value of the current block but instead **`stack[++top]`** stores the next registerNumber value that the next

operation will be held and uses the current register's value in the print statement. Print statement is for writing the current operation to the output.ll file with the according LLVM IR format.

5.6 Hash Table

We did not alter our Hash Table implementation for the previous project. However, we needed 2 extra hash tables differing from the first project: In addition to the table that holds variables with their corresponding values, we also needed a registerTable that held variables with their corresponding register numbers.
Also, for the loading process, we kept a varTable to track variables that have been marked as loaded so far. Shortly after the loading we discard these variables.
We use a quite simple hash table that includes insert and find methods. No delete method was required again for this project.

5.7 Stack

As mentioned before, even though our stack-like array structure performed well for integers, it was not a good fit for strings. So we resorted to implementing an actual stack. This stack structure was created newly for the current project and it was not present in the previous one. The stack is responsible for containing an array of strings. We operate on the stack with 3 methods: **`createStack()`**, **`pop()`** and **`push()`**. Although there are other methods like isEmpty() or isFull(), they are mainly used for error checking and are not in the scope of our concern since we did not commit any malicious use. Stack is used to evaluate, or transform postfix expressions to LLVM IR language code.

6. Improvements and Extensions

Second example in the description forced us to come up with innovative solutions for loading and using repeated variable names in an input line, such as *siu*siu-siu/siu.* These lines were previously handled by loading different register values to every *"siu"* as expected but it led us to obtain erroneous register numbers from hashtable - which stores a unique register value for a unique name. Thus when we were operating on a *siu*siu* expression, it was denoted as *" %18 = mul i32 %3, %3"* instead of *"%18 = mul i32 %3, %4"* in our output.ll file.

As a solution, we started to derive other key names for repeated variable names in the background, without explicitly noticing the user or the compiler. For example, while inserting a second *"siu"* to the hashtable, we are adding an distinguishing integer *"0"* and storing it as *siu0* in the Hashtable. Then, we increment that distinguishing factor by one integer right after, so that the 3rd possible *siu* would be named as *siu1*, and the next one would be named *siu2*... respectively. Thus, in the output.ll file it seems like every loaded siu is also named as *"siu"*, in fact it actually has another name.

Another great improvement to our project was to change the way we handle errors. To be able to print every line number with erroneous syntax or undefined variables, we resorted to a greater error indicator, an integer called **`AllError. AllError`** is set to 1 in every error encounter, meanwhile, a smaller error indicator called **`error`** is set to 0 before continuing the while loop to catch all other erroneous lines. In the end of the program, if **`AllError`** is 1, there was a problem in the input files and we discard the output file we have been forming.

7. Difficulties Encountered

Due to having employed a functional approach in building the first advanced calculator, at first AdvCal++ just seemed like a revision of the old functions. Yet of course we encountered quite cumbersome problems and memory issues during the project thanks to C language and its string handling. After we find it out that integer stack will not work for register number and integer differentiation, we hurried up to create a **`char* stack[]`** and use it as a Shunting-Yardish way which resulted in a semi-catastrophic string handling and did not perform well.

We could not succeed in overwriting the new result of the operation to **`stack[++top]`** which was occupied by the old operand that we used to calculate that result. So when we call for this register number with **`stack[top- -]`** afterwards, we kept getting the old operand instead of register number. The problem was quite complicated to detect at first, since pointers and this special way of string storing of C language is not the first that comes to mind.

Nevertheless, we solved the error by creating a stack struct instead of using a stack array with top index, where we can easily overwrite old places of the string with memory de-allocation and so.

Also in Windows we encountered an issue with opening and writing *.ll files, so one of the partners' computers could not compile the same code that was working well in another computer system.

8. Conclusion

After all, this was a satisfying project for us especially because updating and receiving a robust version of our previous advanced calculator gave us a maintenance experience which operates at interpreter level for simple line arguments. We had the chance to experiment with LLVM IR language, writing another makefile, creating a stack structure from scratch and printing errors line by line. Being familiar with our code from the start gave us a more intuitive sense of handling errors and coming up with more sustainable solutions which will work for most of the case (any case would be a little too assertive).

9. Appendix

Our source code and makefile can be found within the same zip file as this documentation.