CIS 4650: Checkpoint Three

Professor Fei Song

Andrew Trautrim, Andrew Truong

April 5th, 2023

Table of Contents

What was done?

Overview

In this checkpoint an assembly code generator for valid C- programs was implemented, in particular, the "-c" command was added. The assembly commands used for this assignment were designed to run on the Tiny Machine simulator given. The general approach was essentially the same as all previous checkpoints, traversing the syntax tree and creating output for each object. The first step was to understand how the simulator worked, which included the individual commands, the prelude, the finale, and the most challenging, backpatching. As mentioned before, the code generation is done as a traversal of the syntax tree. As the generator moves through the tree, assembly code corresponding to each node is generated and written to an output file. For example, an operation expression works by first evaluating the left hand side and storing the result in register r, and then evaluating the right hand side and storing the result in register r + 1, once this is done, we perform the associated operation and store the result in register r; if the operation is a boolean operation we add additional code like jumps. Furthermore, we also implemented error detection for out of bounds error when indexing an array. If an out of bounds error occurs, the program will simply halt.

Design Process

The majority of the project was relatively easy to implement. For declarations all we had to do was decrement the frame or global offset, variable access was just loading the value at a previously saved offset, and simple operations were just calling the associated assembly

command. However, the two major challenges of this assignment were function calls, and if and while statements. The first challenge was function calls, the code itself is quite small but conceptually difficult. We first had to evaluate the provided arguments and store them on the stack frame, then save the current frame pointer, create a new frame, save the return address plus an offset of 1, and finally jump to the function being called. Once the function was evaluated, the return value was saved in the ac register, the saved return address was loaded, and the latest stack frame was popped off.

Furthermore, the next challenge was if and while statements. Thanks to the flow charts given by the slides, the structure of each statement was simple enough to implement, i.e., the conditional and unconditional jumps, but the test blocks for either of these statements proved to be much more of a challenge. The reason for this difficulty was largely due to how complex these statements could be (e.g. if (A && ~(x <= y) || (B && (C || (y > z)))) {...}). The solution was to treat each binary operation as its own code block. Given a single operation, we would evaluate the left side, then the right side, and store the results in successive registers r and r + 1 and if the comparison was true we store 1 in register r, and 0 otherwise. This way we could construct increasingly complex conditional expressions. Moreover, since there are no inherent instructions for boolean operators, the AND/OR/NOT operators needed to be implemented using the simpler commands (e.g. add, subtract, multiply, divide). In order to circumvent this, AND was implemented using multiplication. Every non-zero value is considered true and zero is considered false, so multiplying the operands results in the correct evaluation of AND. The boolean OR operation, on the other hand, was much more difficult. In order to evaluate this, we instead used De Morgan's Law to instead evaluate an AND operation as before, i.e., A || B = ~(~A && ~B).

On top of the other challenges, there was another when declaring function prototypes. This was difficult because of how function prototypes only tell the program that a function will exist and not where it will exist in the future. This means that as the generator is creating assembly code that references such functions, it does not know if the function does exist or where it is located. Thus in order to get around this, the function prototypes are created to point to where their respective functions are located. This means that any code that references a function that has a prototype will instead actually reference its prototype, which will then point to the function itself. The prototype itself does not know where its respective function is actually located until the function is declared. Thus when the function itself is declared the generator must go back and tell the prototype where it is located.

Reflection

Limitations

Throughout this checkpoint there weren't as many limitations compared to previous checkpoints. However, the previous shortcomings propagated to this checkpoint and caused limitations in that way. Specifically, in C2 we were unsuccessful in checking whether or not the arguments in a function call matched the arguments in a function declaration. As a result, this is not caught as an error prior to code generation and can subsequently create logic errors within the generated code.

Potential Improvements

There are many improvements that the compiler could make use of. Most of those improvements will fall under the optimization category. Currently as it stands, short-circuiting of boolean operators is not supported. This in itself is purely just an optimization since the boolean operations will work normally, they just are not as efficient as they could be. Another large optimization that could be made is memory optimization. For the most part only the first two registers are used when calculating operations. This means that a lot of writing and storing must be done when doing operations. This especially is important if we need to constantly access a specific variable. Instead of constantly writing and storing it to the register, it could stay in a different register until it is no longer needed.

Contributions

Andrew Trautrim

- Operation Expressions.
- Declarations and variable references.
- Function calls.
- Prelude / IO / Finale.
- If and while statements.
- Out of bounds error.
- Documentation (README and report).

Andrew Truong

- Operation Expressions.
- Declarations and variable references.
- Function calls.
- Prelude / IO / Finale.
- If and while statements.
- Out of bounds error.
- Documentation (README and report).