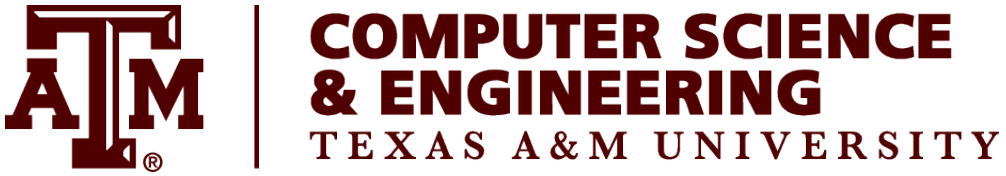

CSCE 434/605: Compiler Design

Spring 2024

Compiler for CO_2

April 26, 2025

Author	*UIN*
Aum Palande	132001644
Parth Kumar	835000337

Summary

Our Compiler converts and executes the code, following the language of CO_2. This language allows for features such as if statements, for loops, while loops, functions, and more. Our compiler takes in an input file and pipelines the inputs to a scanner, parser, type checker, optimizer, and a code generator. The compiler can optimize the code by running constant folding, constant propagation, copy propagation, orphan function elimination, common subexpression elimination, and dead code elimination. Our code compiles and generates machine code for the DLX architecture, which contains a maximum of 24 general use registers, 7 special registers, and memory.

Feature Check List

- ✗ SSA
- ✓ Type Checker
- ✓ Constant Folding
- ✓ Constant Propagation
- ✓ Copy Propagation
- ✓ Orphan Function Elimination
- ✓ Uninitialized Variables
- ✓ Common Subexpression Elimination
- ✓ Dead Code Elimination
- ✓ Graph-coloring Register Allocation
- ✗ Linear Scan Register Allocation
- ✗ Register Allocation - Coalescing

KLOC

Code: 11.6
Comment: 0.75
Total: 12.35

Implementation

Scanner

To implement our scanner, we used simplified maximal munch in order to generate our token stream. While simplified maximal munch may have some issues with certain inputs, we can assume that for the most part, this algorithm will work. There were some issues with generating extra error tokens, but our error tokens were only generated extras if there was already an error token. So this does not create any errors.

Parser

Once we are given the token stream from the scanner. To solve this, we used recursive descent technique, which broke this down into methods for each grammar rule, and made sure that the token stream correctly follows the grammar. A lot of these methods were straightforward, but some of the declarations created issues. We output any errors we might get, which means that our input code is nonconforming to our grammar.

Typechecker

Our type checker is self explanatory and checks for the correct type expressions (i.e. adding ints to ints, comparing booleans compared to ints). We used specific checkings for each type in our AST. For example, our checking for if statements first looked at the relation to make sure it has the correct typing, and then looked at the following blocks. There was also a need for function checking, to make sure that all the arguments that get passed are in the correct type, as well as the return type being the same as well. Similarly, we also implemented a spaghetti stack for our symbol table, which allowed us to confirm that all variables were also correctly initialized and in scope.

Optimizer and IR generation

We converted our abstract syntax tree into immediate representation. This immediate representation of three address code allowed us to have our code represented in assembly adjacent language, where each instruction has a destination, and some expression. The expression can either be an addition, subtraction, function call, or similar. We chose not to use SSA, as generating our code in SSA was rather difficult, and we could not get it working well. We figured the benefits outweigh the possible errors that could have been created from our conversion algorithm.

Our optimization process was simple, and followed a very similar algorithm for all of our different features. All of our optimizations had a kill operator, a gen operator, and apply operator. For example, our common subexpression elimination held a set of subexpressions that can be replaced. If we find a new subexpression, we can add it to the set. If we use a reassign a variable, x, we kill all subexpressions that contain the value x. Finally, if we find an expression that is already in our set, we can replace that expression with the variable that already contains that same value. We ran these optimizations on each basic block, and carried over the set if needed.

Register Allocation

Our register allocation runs a scan of our variables, and determines what registers should hold each variable. To do this, we construct a Register Interference Graph that determines which registers will interfere others. We say that an interference occurs when a register's live variable analysis overlaps with a different register's live variable. After we have this graph constructed, we can run our graph coloring algorithm to see if any registers are spilled. We have specified Register 25 for the destination, and Register 26 for the left value and Register 27 for the right value. In other words, if our instruction was t0 = t1 + t2, t0 would be in Register 25, t1 would be Register 26, t2 would be Register 27. While Register 25 is also used for the return value, we expect that there will be no overlap, as we save the value of the destination back into memory before marking it as "free" for the next instruction (which might be a return).

Code Generation

We use a similar "visitor" pattern that is used in the previous steps. Each node in the CFG has a specific method, which generates the instruction, or instructions that is necessary for a specific instruction. If a variable is spilled, we need to add specific instructions that load the value into the designated register, and specific instructions to load the destination back into memory if its also spilled. To handle branching, we mark the starting position of each block, and add the specific branch instruction to that starting position. We do a similar algorithm for function call, and always return to the address held in Register 31.

Design Features

Unitialized Variable Detection

We check for uninitialized variables. This is checked, and if a variable is used before its initialized, we first set it to 0.

IR instruction representation

We represent each instruction in three address code, or TAC. These TACs will at maximum contain a destination, left side, operator, and right side, i.e., t0 = t1 + t2 or t0 = t1. This helps ensure for our graph coloring that we will only need a maximum of 3 registers for spillage at all time.

Epilogues

Each function may have multiple return points but all of these return points consolidate into one return point. The returns for each CFG will return the same previous location saved in Register 31. All the function points will bring the program counter to the return block, which will return from the code.

Return values

Similar to x86 architecture, we will mark a specific register as a return value. We can use this register as a normal register, but ensuring that the function call will have the value in that register at the time of returning. Register 25 is implicitly marked as a return value. Any function call must

make sure that this register is free before calling a function, and this will always be true, as 25 is reserved for destination spilling, which will always be free after an instruction is handled.

Heuristics used for graph coloring

For graph coloring, we use a simple algorithm similar to shown in lecture. We pop each node that has edges $< k$, where k is the number of registers, or pop a random register and mark it as spilled if no proper node exists. When adding back, we use a heuristic of lowest register first, i.e, where we first check if we can assign register 1, then register 2, etc. If we have a tie between two registers, where they both have a valid number of edges, we can randomly select either one, and our algorithm is still valid. This may cause issues in debugging, but not in theory.

Spilled variables

We reserve three registers for spilled variables. We will only need a maximum of three registers, as we are using three address code. After we load and use our spilled variable, we will then immediately mark it as free, and get ready for the next instruction.

Optimizations

Test cases evaluate critical compiler optimizations that improve execution efficiency and reduce runtime. Specifically, the focus is on constant propagation, constant folding, dead code elimination, common subexpression elimination, elimination of uncalled functions, and analysis of unreachable call trees. These techniques are pivotal for optimizing compiled code by reducing its complexity and enhancing performance. All optimizations run in a specific compilation is noted in the file: "date-time-optimization-output.txt", where the date and time is the time when the command is given (more specifically formatted: yyyy.MM.dd.HH.mm.ss). Below are the pre and post optimized dot graphs.

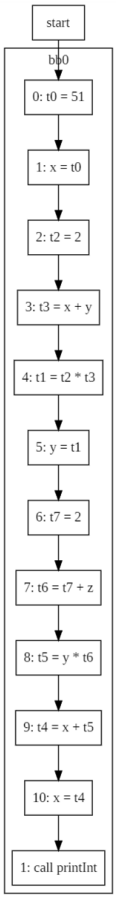

Figure 1: Pre-Optimization for Test 1

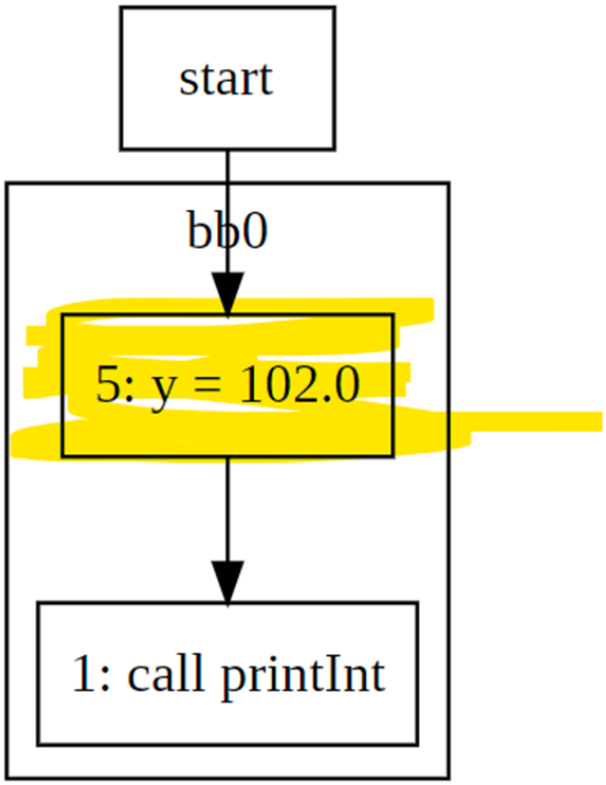

Figure 2: Post-Optimization for Test 1

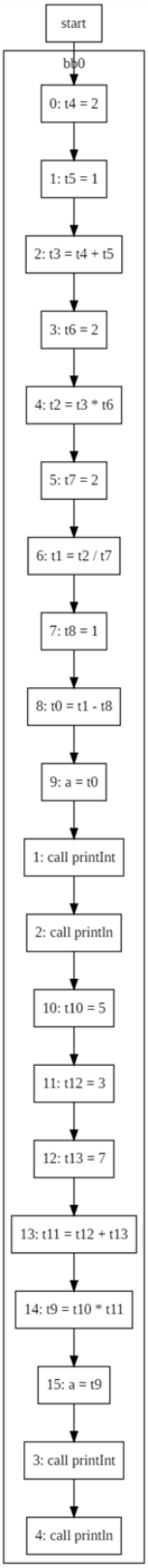

Figure 3: Pre-Optimization for Test 2

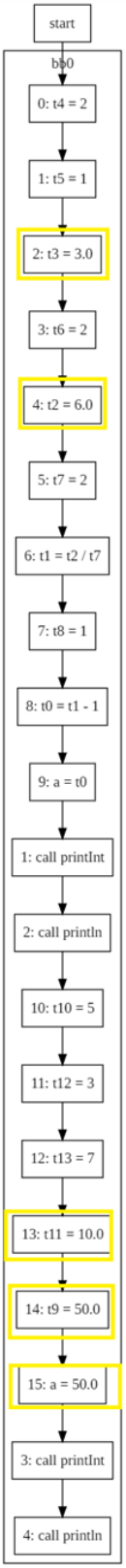

Figure 4: Post-Optimization for Test 2

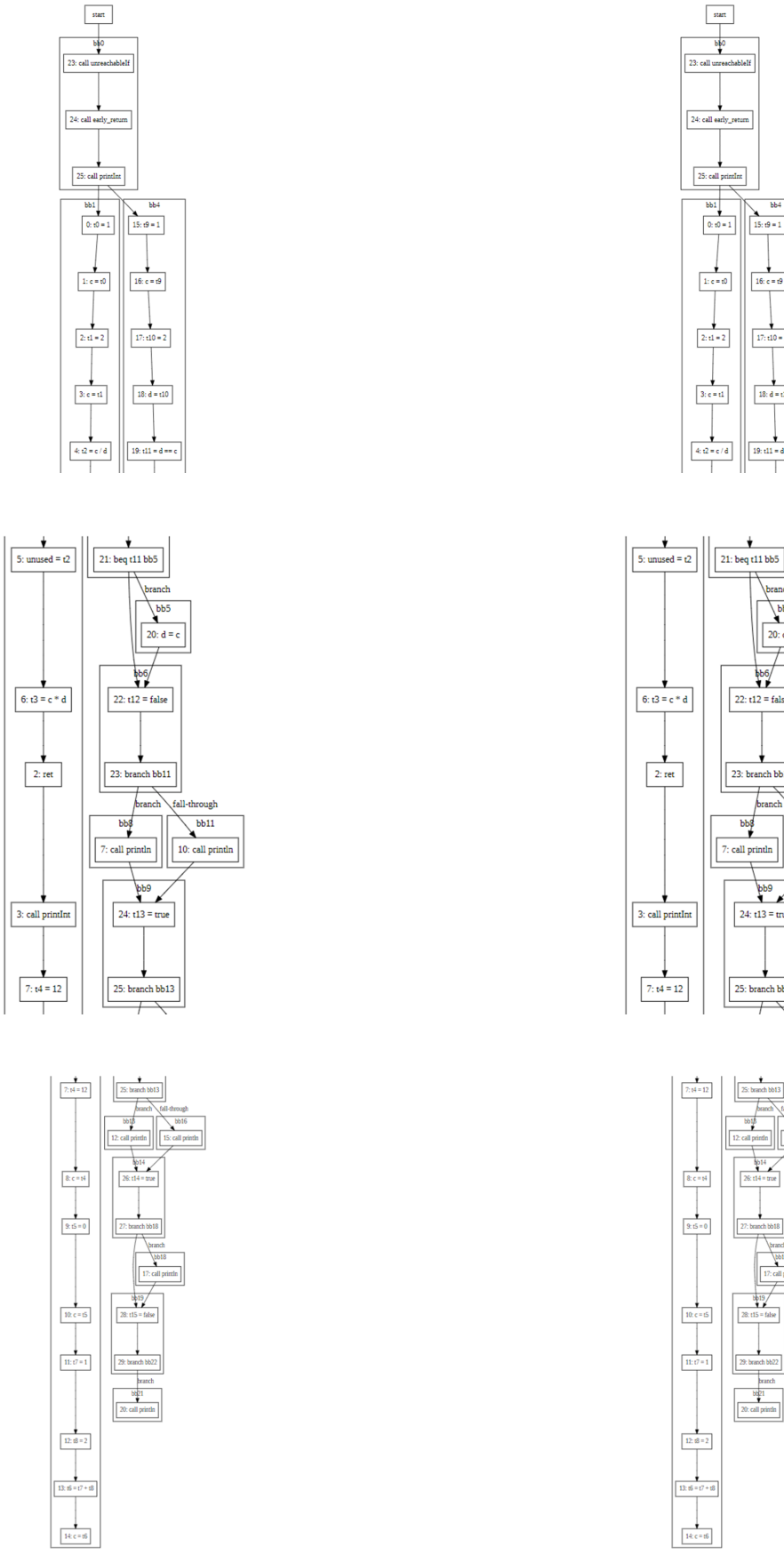

Figure 5: Pre-Optimization for Test 3

Figure 6: Post-Optimization for Test 3

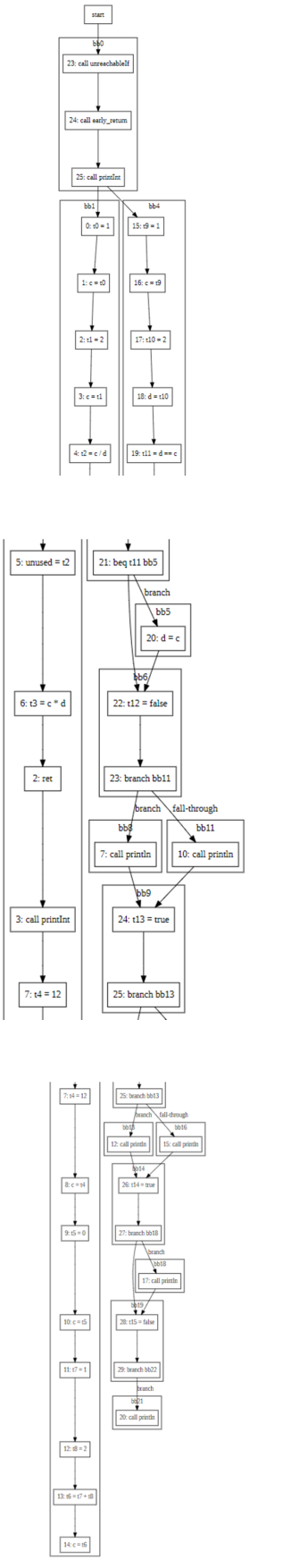

Figure 7: Pre-Optimization Phases for Test 4

Figure 8: Post-Optimization Phases for Test 4

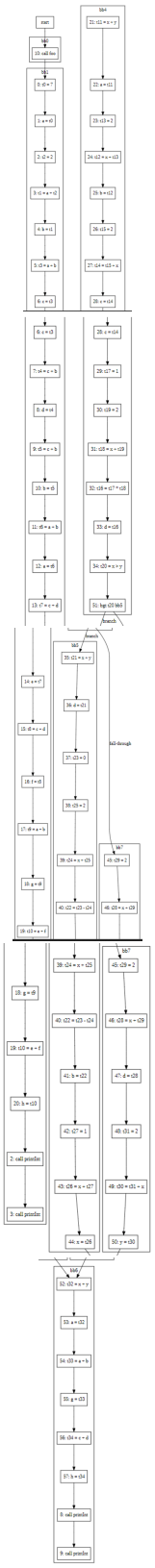

Figure 9: Pre-Optimization for Test 5

Figure 10: Post-Optimization for Test 5

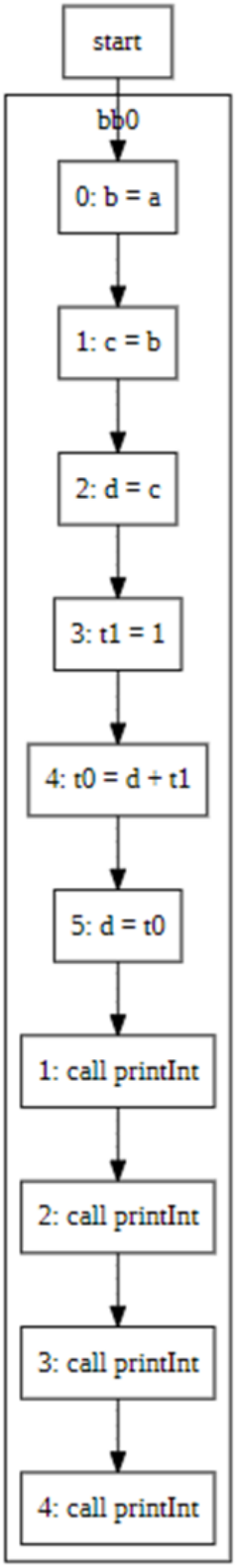

Figure 11: Pre-Optimization for Test 6

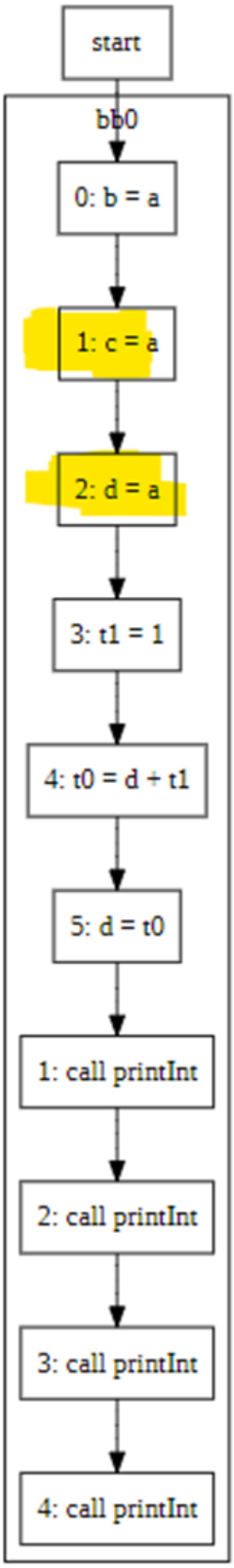

Figure 12: Post-Optimization for Test 6

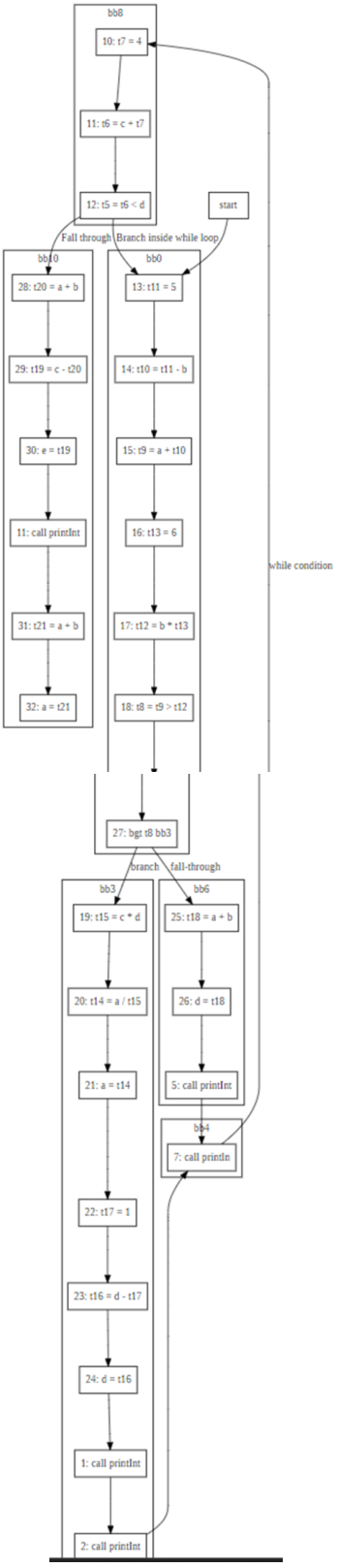

Figure 13: Pre-Optimization for Test 7

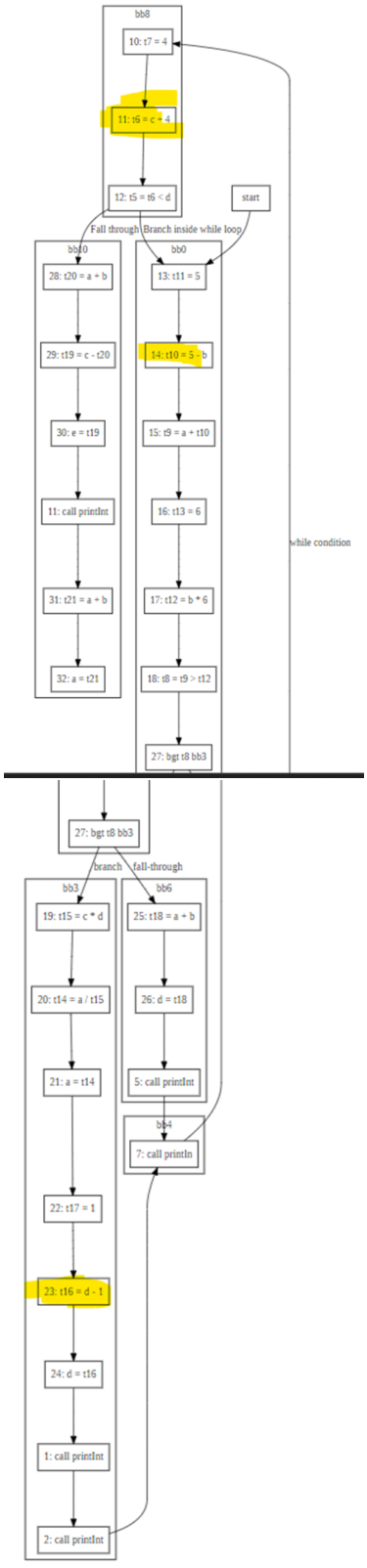

Figure 14: Post-Optimization for Test 7

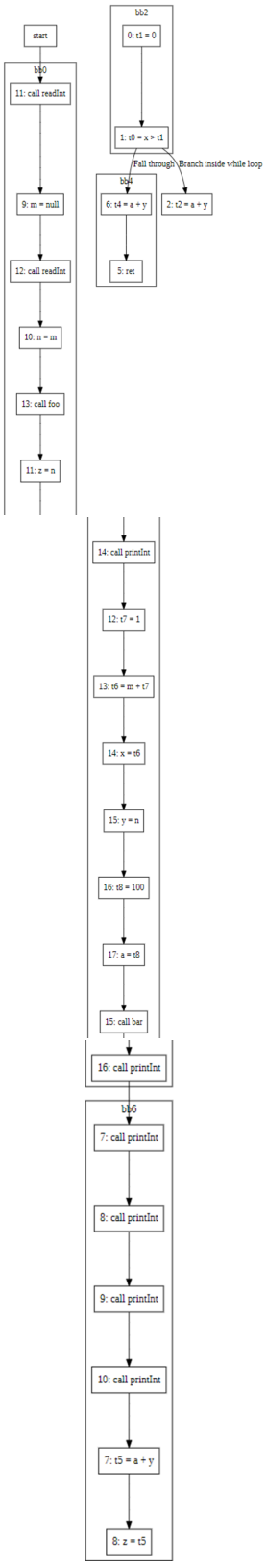

Figure 15: Pre-Optimization for Test 8

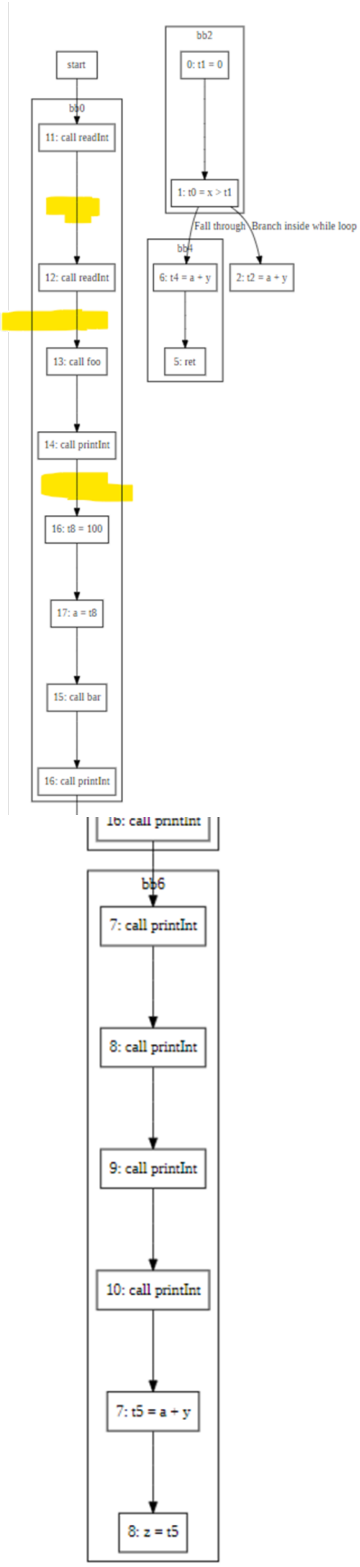

Figure 16: Post-Optimization for Test 8

Updates after PAs 5-6 Evaluation

Most other test cases that were not working before the evaluation are working now. We fixed lots of bugs in register allocation, graph coloring, code generation, and spilled variables. Before the evaluation, we did not have a way to handle spilled variables, and still had lots of errors. Now, these errors are handled and should not be causing issues. For register allocation, we fixed an issue where our graph was not correctly being created, and had less edges than we should have. Similarly, we were also allowing values to get the register value 0, but this has been fixed. For code generation, we added the handle for spilled variables and fixed more issues with handling floats and immediates. We also fixed Load and Store issues which caused a problem in the way spilled variables were being stored in memory, and removed code which was redundant and adding more instructions. We had a few issues which stemmed from the AST, so we went ahead and fixed those as well.

Reflection

What went well

Our code generation, and the early portions of the project went very well. We knew that these early portions need to be near-perfect in order to setup for the next stages of the code. Our scanner, parser, type checker, AST generation, IR Generation, and most of our code generation also went well, and was clear. Technically, we understood the bulk of what was understood of us, and it helped when the explicit algorithm was understandable as well. We were able to co-ordinate well after the overhead of joining forces, and split tasks to agreement, analyzing and debugging parts of the code together.

What did not go well

Branching, function calls, and arrays did not go well. We had trouble with a lot of the execution of these, especially in the type checker and the code generation. We worked on branching and function calls for a long time, and tried multiple different algorithms, which kept giving us infinite loops. We thought of the algorithm of writing jumps to the beginning position of each block, which is what we ended up on, but we also had to problem of accounting for extra instructions that we are adding. Similarly, arrays have been a problem since type checking as well, but we were able to catch up and still create a semi-functioning system.

Advice

I think that starting early is always a good thing to do. People always say it, but this is especially true for the later stages in the project. The project often seems a lot smaller than it should be, but the longest portion of each PA is the debugging. Make sure to find a partner, and split the work well so that you both understand the inner workings of the whole system. Joining another team late is much harder to do, as you have to relearn and understand a large code base and can results in lots of catch up and wasted time. Some mistakes include adding silly error and edge case handlers rather than specifically thinking about why that edge case is passing through your algorithm. Also, making sure that you understand what you should be doing before you start is a big deal. Often times, I found myself going back and reformatting large amounts of code that was wrong, or solved the wrong problem. The project should also definitely be carefully planned out, especially the backend since PA4, errors can quickly snowball and be hard and excruciating to debug and having to run through stack traces can be tedious, time-consuming and not feel rewarding, make sure to debug well throughout and not leave major errors, never skip testing. Don't jump straight to the code, always have discussions with your partner and make work scopes clear.

Work Arrangement and Contribution

Work Arrangement

Aum completed PA1, PA2, PA3, and PA4 alone, Parth had completed PA1, PA2 and PA3 before joining Aum. Due to unforseen circumstances, Parth had no partner, and joined Aum to work together on PA5 and PA6, moving forward with Aums code due to having higher autograder scores. For PA5 and PA6, we split the work evenly. We worked evenly, and had Aum focus on stuff that contained some larger portions of the older code, as he knew his code better.

Code Contribution Graph

The graph are down below (for some reason, LaTex kept moving below the certification). Some of Aum's code was because he accidentally committed a two folders that he was going to zip for submission. This most likely contributed for about 20,000 lines of code and 7,000 deletions, leaving his contributions to be 4,077 additions and 2,753 deletions. The graph only marks from March 25 onwards, as thats when Aum pushed his code to GitHub. The graph also doesn't include Parths PA1-3 lines of code since both Aum and Parth had worked on those alone, and moved forward with Aums code.

We affirm that each of us had fully read this report, and we reached a consensus. This report describes the correct status of our submission as of April 26, 2025.

"On our honor, as Aggies, we have neither given nor received unauthorized aid on this academic work."

Signed, Aum Palande and Parth Kumar

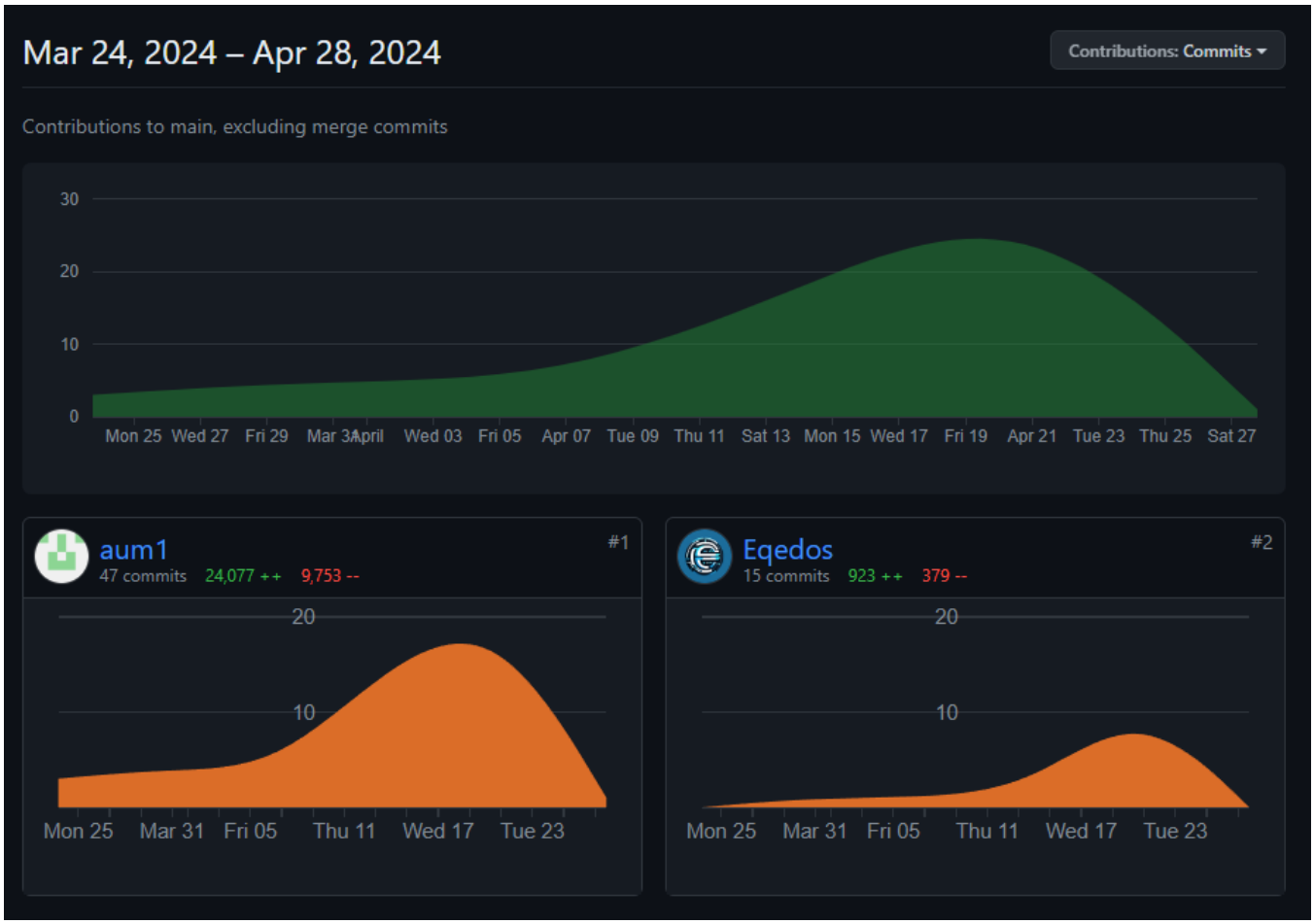

Figure 17: Code Commit Graph

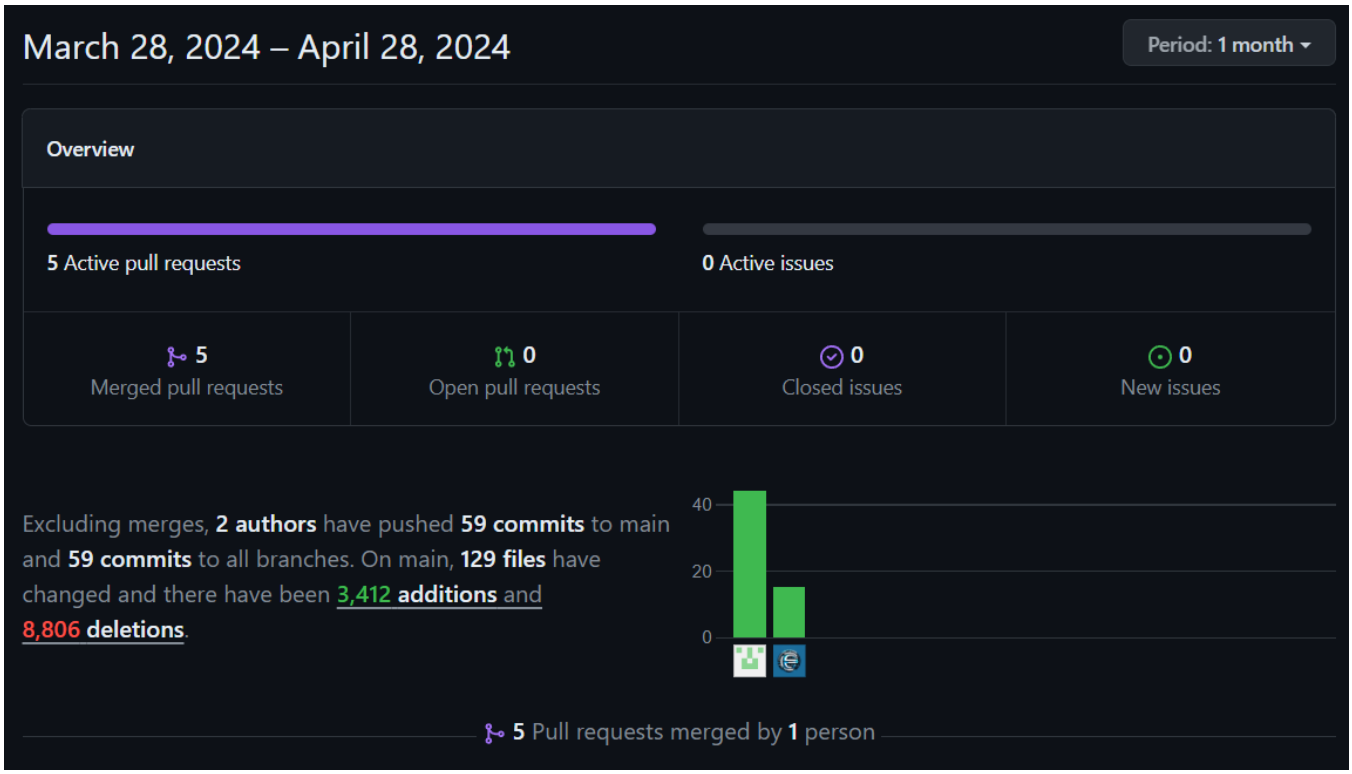

Figure 18: Pulse Commit Graph