Benjamin Steenkamer
CPEG 621-010
5/20/19

Project Report: A Calculator Compiler with Optimizations

Abstract:

In this final project, the main goal is to create a calculator language compiler frontend, backend, data dependency analyzer, and implement two optimizations. The motivation for this lab is to learn how to put together all the main sections of the compiler I learned about through the semester, and to learn about the implementations, benefits, and tradeoffs of compiler optimizations. The key issue being addressed in this project is how to create a heuristic that allows common subexpression elimination (CSE) and copy-statement elimination to work together, improve code efficiency, and still reach a fix point.

Overall, my project was able to fully implement the calculator compiler language and has support of up to 3-deep nested if/else-statements in most cases. The compiler frontend and backend meet the exact requirements laid out in the lab instructions. The data dependency analysis is fully implemented and can correctly identify flow, anti, and write dependence between lines and works with n-deep nested if/else-statements. The CSE and copy-statement elimination optimizations are also fully implemented and work with n-deep if/else-statements. The cooperative heuristic that governs them ensures a fixed point is always reached, and that copy-statement elimination does not undo CSE's work. This was achieved mainly through special variable notation and look-ahead logic. The optimized program runtimes were compared to the original versions, and it was found that the optimizations speed up the programs by 36% on average. This speedup is due to my implementation focusing on the reduction of unnecessary variable reads to lower register pressure, the removal of redundant math operations, and the removal of dead temporary variables.

Detailed Strategy:

Compiler Flow:

To implement this calculator compiler, I split each main task of the project up into its own module so that it could work independently for easier testing. Each module is called from the main calc.y file to execute its task. The main strategies for this project were the use of module programming, consistent reuse of dependence tracking for easier implementation of tasks 2 and 3, and a focus on ensuring correctness of all cases over a more efficient implementation. The general flow of this compiler starts in the calc.l and calc.y files, where the input calculator program is tokenized, has grammar rules applied to it, and is then converted into a three address code (TAC) representation. As each TAC line is generated, they are sent to the data dependence module (data-dep.c), where the three types of dependencies (flow, anti, and write) are found between the current statement and previously recorded ones. After the full input program has been converted into TAC, the dependencies between each line are printed out to the console. Then the control is sent to the optimization module (cse.c and copy-stmt-elim.c) where the unoptimized TAC code is given as initial input. A loop is then run which iteratively calls each optimization, in a back and forth manner. Each module performs their optimizations and edits the TAC file that the previous one had just edited. This continues until the fix point condition is reached and no more changes can be done. The loop then exits, and the backend produces C code versions of the optimized and unoptimized TAC. After this, the calculator compiler has finished its process.

Running the calculator with a legal input program (./calc <input>) will produce 6 main files. tac-frontend.txt is the unoptimized TAC representation of the input program. backend.c is the corresponding C code version of the unoptimized TAC. backend-timing.c is this same C code, but with an added timing loop for conducting timing tests. tac-opt.txt, backend-opt.c, and backend-opt-timing.c are the corresponding files that instead contain the optimized TAC code. The C code files can be compiled and run as programs.

Flex and Bison Rules (Frontend):

The compiler frontend for this lab has changed very little between this project and the previous labs. The language tokens and syntax implemented by this frontend (variables, integer constants, and operators), variable constraints (allow undeclared initial use with 0 as the initial value and case-insensitive names), and usable operators (+, -, *, /, !, **, =, (), ?) are identical to the previous lab. To save time and space, I will not go into detail on the specific meanings of the Flex and Bison rules I used to implement the language. For more details on what each of these rules do, please refer to my previous lab submissions, as the rules are identical. The only significant change between the previous lab and this project is that the Bison rules have added calls to other modules, such as the data dependency and optimization modules.

The frontend's task is to take in a legal input program written in the calculator language and convert it into internal three address code (TAC). The Flex rules tokenize the input program and the Bison rules apply the grammar I created and checks for syntax errors. In the Bison rules, the TAC is generated for each statement, temporary variables in the form of _t# are introduced as needed, and the generating of if/else-statements is also handled. The depth of nested if/else-statements is artificially limited to 3-deep due to complexity issues with making the limit any higher. However, not all cases with 3-deep if/else-statements are covered in my implementation. The issue occurs when a variable is being assigned a value in the third nested if/else-statement, and the frontend incorrectly assumes this internal assignment is actually the variable being assigned to the result of the conditional expression. This is due to the recursive nature of the Bison rules mixed with the way I track the ending of if/else-statements. It will then incorrectly generate the closing of the if/else-statement structure using this internal assignment instead of waiting for the full expression to be resolved. This only occurs for 3-deep if/else-statements with the specific assignment sequence.

I found this issue very difficult to fix and dedicated over two days and applied several potential fixes attempting to solve this issue. In the end, I needed to spend my time on other parts of the project, and I was not able to solve the issue by the time of submission. If I had more time and resources, I would certainly go back and fix this issue. In an attempt to make up for this shortcoming, I made sure the following tasks of the project (data dependence and optimizations) would work with n-deep if/else-statements, and to the best of my knowledge, they do.

This issue is not present if the same sequence of operations is done with 1 or 2-deep if/else-statements. 1 and 2-deep if/else-statements work correctly in all cases. Also, for 3-deep if/else-statements, if no assignments occur in the third conditional statement, the 3-deep if/else will be properly formed without issue. This means my compiler can form 3-deep if/else-statements, but just of a limited kind. To demonstrate this, I have included several test programs in the Test folder that show even with this limited form, complicated 3-deep if/else-statements can still be created without issue. For the most part, the files with the number 3 in them will have examples of 3-deep if/else-statements.

Data Dependence:

As the frontend is producing the TAC lines, it is simultaneously sending the lines to the data dependence module for more processing and recording of data dependencies. For the current line, this module finds all three types of data dependencies for each variable in the line. The three types of dependencies are flow (read after write), anti (write after read), and write dependence (write after write). Also, only 1 "hop" of dependence is found for a variable. The general algorithm for this module is shown below:

1) Record variables in current TAC line and whether they are being read or written to (ignore constants); Record the if/else nest depth and whether the line is in if or else statement
2) Check dependencies for each variable in TAC line by looking backward into the previously recorded lines
    a) If variable written to in current TAC, check for write and anti-dependence
    b) If read from in current TAC, check for flow dependence
3) Record the dependences as they are encountered
    a) If a flow or write dependence in a guaranteed path is found, stop looking back. This occurs when:
        i) Flow or write dependence is found outside an if/else-statement (depth = 0)
        ii) Flow or write in 1-deep if <u>and</u> else statements (one of these paths is guaranteed to executed)
    b) If the start of the program is reached, stop
    c) For a potential dependency found in an if/else, look forward from the potential dependency to see if a future statement in the if/else will block it
        i) If it is not blocked, record it
        ii) Special case: If the current TAC line being processed is in the same if structure as this potential dependency, determine if that dependence will block other dependencies further up (in the straight-line code view) by comparing the statement depths

Once all the dependencies for the current line have been found, the control is returned to the compiler frontend to produce the next TAC line. When the entire input program has been processed, the dependencies between statements for each statement are printed out in the format specified by the lab instructions. An example of this output is:

```
S6:
Flow-dependence: S5
Anti-dependence: S0
Write-dependence: None
```

As seen above, the different types of dependencies between the current line and previous lines are given, but it does not specify which variables between the statements caused the dependencies. This was done to match the given implementation as specified by the lab instructions.

This data dependency tracking should work for any amount of nested if/else-statements, but it is artificially limited to 3-deep because the Bison rules only allow the production of 3-deep if/else-statements. I have tested and confirmed this n-deep compatibility is true by bypassing the frontend and providing the data dependency module directly with 4-deep if/else-statements. The module was able to process this nesting structure correctly and without issue. I believe this would be true for any sized if/else-statement.

For correctly determining dependencies for a given variable in the current TAC line, the module looks backwards at the already processed code, in a straight-line manner. Each line has a data structure which records the variables used in the current line, whether they are written or read from, and 3 arrays (one for each dependency type) that contain the line numbers of its dependencies. Looking backward, the module searches for previous uses of the given variable and whether the use was a read or write. Depending on how the variable is being used in the current line and the previous use type, it will record the line number of the dependency in the appropriate array for the current statement. The process of looking back stops when a previous write to the variable is found in a guaranteed execution path or when the start of the program is reached. This previous write acts as a "blocking point" and prevents any previous dependencies before this line from propagating onward. For example:

```
S1: y=5;
S2: y=a;
S3: x=y;   // Current line
```

S3 would only have flow dependence with S2 since the writing to y in S2 blocks the flow from S1's y to S3's y. This is also the case when looking for anti and write dependencies and a blocking write to the variable is found.

There are also a few other cases with this method that I accounted for an will now show:

```
S1: a=a;
S2: a=x;   // Current line
```

In this case, S2 would only have write dependence with S1 and not anti dependence since the writing to a is that last operation in S1 and would therefore block the read of a in S1.

```
S1: y=x;
S2: z=x;
S3: x=1;   // Current line
```

S3 would have anti dependence with S2 and S1 since there is no blocking write. This anti dependence would continue back until a blocking write was found. One final corner case is that I decided a line can't have anti-dependence with itself: a=a. For my definition of dependencies, it must be between the current statement and *previous* statements.

An important aspect of this data dependency tracking module is that it can correctly find dependencies within if/else-statement path flows. It does this by first going backward from the current line as normal and takes note of the if/else depth the previous lines are in. If a previous use of the variable is found and the depth of that variable is greater than zero, the module knows it is in an if/else-statement and should then start the second stage of the process. This second stage involves starting at the potential dependence and looking forward from this line until it has reached the original line that started the dependence search. If during that time it runs into another instance of the variable being written to at a depth equal to or higher (closer to 0) than itself, then the module learns that this future write will block this current one in the if/else path flow. In this case, this potential dependency that started the look forward should be ignored. Otherwise the dependency is added to the original line's list since we know no other variable use on the path between them will block the dependence. In short, my implementation can find these blocking statements with consideration of the if/else context that they appear in.

There are several other cases that this module also accounts for with regards to dependence in if/else-statements, but I will spare the details as all cases are heavily commented in my source code. A final note is that this system is also able to account for the case where the current TAC line is in an if/else-statement and can determine whether dependencies in the same or other if/else-statements will reach it.

This tracking scheme I used was a major design choice for this module. By keeping track of the if/else depth each variable is used in, it makes it possible for this module to work with n-deep if/else-statements and allows it to determine guaranteed paths of execution and blocking dependencies. I chose this specific way of looking backwards and then forwards as needed because I found it logically easy to implement, and it required minimal data structure management. The downside of this implementation is that it's not very efficient, as you only need to look backward once in a more efficient implementation. However, to do this I found I would have to construct several temporary data structures and a few other tracking measures, and I didn't want to deal with managing them. In short, the reason why I chose this implementation was that it guaranteed correctness for all data-flow paths and was logically simple for me to implement.

Optimization – Common Subexpression Elimination (CSE):

Once the frontend TAC has been generated and the dependencies have been found, the optimizations stage begins. The goal of CSE is to remove redundant arithmetic operations and reduce the number of variables that need to be in registers. Both aspects will improve performance of the original program while still maintaining the same functional correctness.

CSE achieves this by computing a common subexpression once and storing it into a new temporary variable (`_c# = a op b;`). It then substitutes every future use of the subexpression with this temporary variable until one of the original variables from the subexpression is written to. This writing invalidates the saved subexpression because the value of the original variable has changed, and the temporary variable now contains an outdated result of the operation. Subexpressions are recognized in the form of `b op c`, where b and c can be any combination of constants and variables. The idea is that by storing the result of the calculation and using the temporary variable in place of the expression, you will be increasing performance by removing the cost of recomputing the operations every time. Also, in the case where two variables are used in the expression, the use of a single temporary variable decreases the number of variables that need to be stored in registers. In the case where both b and c are constants, or one is a constant and the other is a variable, it is assumed that the time saved by removing the redundant arithmetic operations will outweigh the overhead of introducing a new variable.

Because of time constraints, I wasn't able to take into consideration the strength of the operations (e.g. the power operation takes more computation time than a plus operation) when deciding if a subexpression should be stored into a variable. This would certainly be a future modification I would make if I had more time. The tradeoff of my implementation is that it isn't "smart" when deciding whether to eliminate a subexpression. However, the benefit is that the simpler nature of this method made it easier for me to ensure all border cases were accounted for. Even with this simple implementation, the results I collected show it is effective at producing more performance efficient code.

The CSE optimization module works by using this general algorithm:

1) Read in a TAC line, look for a subexpression in the form: `a op b`
2) If this subexpression has never been seen before, look ahead from this line to see if there are any more future uses of it before it is invalidated
   a) An invalidation occurs if one of the variables in the subexpression is written to or the if/else context in which the subexpression exists is exited
   b) If there is at least one more use of this subexpression before it is invalidated, record it in the subexpression table:
      i) Record the if/else depth, the variables or constants, and the operator used in the subexpression
      ii) Generate and record a new temporary variable for this subexpression in the form `_c#` and assign the subexpression to this value; also assign the current TAC line this temporary variable if needed
         (1) If the current TAC line was a previous subexpression from the last round of optimization (i.e.: `_c# = a op b`), reuse the `_c#` temp instead of generating a new variable
2) If this subexpression was seen before in the program and a valid subexpression variable exists in the table, substitute the subexpression for this variable
3) Take the variable being assigned a value in this TAC line and go through the subexpression table and invalidate all subexpressions that this variable appears in
4) If the TAC line is the exiting of an if/else context, go through the common subexpression table and invalidate all subexpressions that were created in this if/else context.

With this algorithm, each new subexpression gets a unique number for its temporary variable (_c1, _c2, …) so that there are no conflicts between them. The module also goes through the TAC once before the start of the process to see if there are any previous subexpression variables from the last round of optimization. It makes sure to increase the starting number of new subexpression variables to one higher than the last one used. This algorithm only creates a record for a subexpression if it is found to be used at least twice in the program before being invalidated. This ensures the subexpression is actually "common" and prevents performance loss from inserting an extra temporary variable that will only ever be used once.

The purpose of reusing its own temporary variables when possible is to prevent a "stacking" of subexpression assignments that is wasteful to performance. Since this old temporary variable is already assigned to the subexpression that was found to be used more than once, CSE can reuse it without issue. This situation occurs when copy-statement elimination exposes more instances of a subexpression that was already eliminated in a previous round.

If any variable in a subexpression is written to after the expression is saved into the temporary variable, the expression becomes invalid. It does not matter what if/else path the assignment occurred on. The module must always assume this path is the one taken (even if it isn't) during runtime to prevent stale values from propagating in the case where the path is taken. If the subexpression is first introduced inside an if-statement, then when the context of the if-statement is left (the execution goes out of the statement), the subexpression becomes invalidated. This is because there is no guarantee that path will be taken during runtime and therefore no guarantee that the temporary variable assigned to the subexpression will be initialized for future use outside the if-statement.

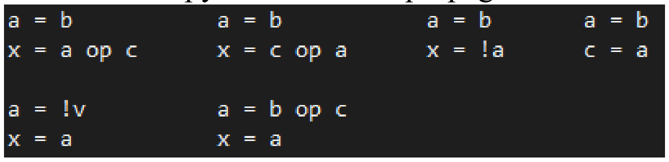
The CSE algorithm will also correctly identify and potentially eliminate common subexpressions that appear inside the if conditionals: `if(a op b)`. I point this out because it took extra care in ensuring the conditions for this border case were met, since the insertion syntax and look ahead logic were different from the normal case of `a = b op c;`. However, the same eligibility requirements for eliminating these types of subexpressions are the same as the normal case: they must be used at least one more time within the program before being invalidated.

My CSE implementation should work for any amount of nested if/else-statements, but it is artificially limited to 3-deep because of the Bison rules. For if/else context tracking, it uses a similar method to the data dependence module. CSE will also ignore `!var_name` and `!constant` subexpressions and not store them in temporary variables, as I assume this operation is fast enough to be done every time and wouldn't outweigh the overhead of introducing a temporary variable. CSE can also account for the commutative property of addition and multiplication, and it sees that `a + b == b + a` and `a * b == b * a`. It takes this into account when looking for common subexpressions and inserting temporary variables.

Optimization – Copy-statement elimination:

After CSE is run, the copy-statement elimination optimization is executed. Copy-statement elimination attempts to eliminate the need for `a = b;` statements by propagating the value of b in places where a is used. It also includes replacing the use of b with the contents previously assigned to b. The goal of this optimization is to reveal more common subexpressions so that CSE can find and eliminate them. There are also chances to reduce the number of variables that need to be in registers by replacing multiple equivalent variables with one variable. Copy-statement elimination will also remove dead temporary variables, which is when a temporary variable is assigned a value, but never used again. Removing these lines is a total performance gain because it can be removed without issue and execution of the line is a waste of resources if left in.

There are cases where copy-statement elimination, by itself, will not offer a performance improvement. However, by potentially revealing more common subexpressions in the program, the benefits of CSE can be increased. Copy-statement elimination recognizes the following situations as a chance to eliminate copy-statements or propagate values:



```
a = b           a = b           a = b       a = b
x = a op c      x = c op a      x = !a      c = a

a = !v          a = b op c
x = a           x = a
```

In this image, copy-statement elimination will be taking the right-hand side of the top expression and propagating it to the appropriate place in the right-hand side of the lower expression. The algorithm for copy-statement elimination is as follows:

1) Read in a TAC line; if the current line contains a variable being read from and it has a valid entry in the expression table, try to replace this variable with its previously assigned value
    a) Only do this replacement if the previous assignment and the current line match one of the forms shown in the image above (this is done to maintain TAC form)
2) Record the TAC line into the expression table, making note of what variable was assigned a value, the variables being read from (if any), the constants used (if any), and the if/else depth it is in
    a) **Do not record the line if the assigned variable is: _c# = …**
        i) This prevents an infinite back and forth of CSE and copy-statement elimination trying to undo each other
    b) Do not record self assignments: x = x;
3) Look at the variable being assigned in this line and go through the table of recorded statements and invalidate all previous entries that contain any instance of this variable
    a) This step won't interfere with _c# =… assignments
4) If the current line signals the exiting of an if/else context, go through the table of expressions and invalidate the ones that were first found in the context being left
5) If the line contains an assignment to a temporary variable (_t#), look ahead to see if it was ever used again on any path
    a) Delete the line if the variable was never used again

In this algorithm, invalidation of previously recorded statements occurs whenever a variable in those statements is assigned a new value. This is because with the assignment of a new value, the equality of the previous statements is no longer true, and they therefore can no longer be propagated as equivalent values. This invalidation will occur if the assignment is on any if/else-statement path for the same reasons as the assignment invalidation from the CSE module. Again, like the CSE module, invalidation occurs when the if/else context is left for the same reason: there is no guarantee the path will be taken, so the value can't be propagated outside the context. This invalidation works for n-deep nested if/else-statements.

Optimization – Heuristic:

The main purpose of this heuristic is to allow CSE and copy-statement elimination to run together iteratively and still be able to achieve a fix point (no more optimizations can be done). The heuristic must also allow the two optimizations to achieve a positive performance benefit to the program on top of achieving this fix point. The difficulty with creating this heuristic is that CSE and copy-statement elimination, in their raw forms, will undo each other's work and a fix point will never be reached. To solve these issues, I have implemented several features within each optimization that allow them to work together in a cooperative manner and prevent them from undoing each other's work. I also added in features so that they will not create redundant work for themselves. Many of the features that enable this have already been mentioned in the previous sections, but now I will give them context to the overall heuristic.

The main logic used in CSE is to only insert a temporary variable and eliminate the subexpression if it has been used at least twice in the program before becoming invalidated. This is to prevent the unnecessary overhead of adding a temporary variable for it to be used only once.

If this wasn't the case and CSE inserted a temporary variable for a subexpression used once, it would 1) only shift the location of where the variables are being accessed and introduce another variable that also needs to be stored in a register, and 2) not be gaining any arithmetic speedups, since the one and only calculation of the subexpression is still present. By making sure there is at least two instances of the subexpression, it is certain that the total number of arithmetic operations will be decreased with the elimination and that the total number of variables that need to be stored in registers will go down (if two variables make up the subexpression), reducing register pressure. If there are 1 or 0 variables in the subexpression, CSE assumes that eliminating the duplicate arithmetic operations will outweigh the introduction of the temporary variables in terms of computation time saved.

Related to this is the detail that CSE uses a special notation for its temporary variables, which is "_c#". This differs from the normal TAC temporary variables of the form "_t#". The purpose of doing this is to help CSE and copy-statement elimination differentiate between what statements have been optimized previously and which ones haven't. As mentioned before, if CSE sees the opportunity where it can eliminate a common subexpression, but the subexpression is assigned to a CSE temporary variable from a previous optimization cycle, it will record and reuse that temporary variable instead of introducing a new one. This prevents a stacking of subexpression elimination variables, which would look like:

```
_c1 = a op b;
_c0 = _c1;
```

By reusing its own temporaries, CSE prevents this case and the unneeded overhead of adding another temporary variable.

When CSE does insert one of these temporaries or substitutes a common subexpression with a recorded temporary variable, it will iterate a counter for the total number of changes it has made this optimization loop. After all TAC lines have been processed, it will return this number back to the main loop, where the value will be used to determine if a fixed point has been reached for the CSE module (a value greater than 0 means it has not been reached). With enough iteration cycles, CSE eventually finds there are no more subexpressions that can be eliminated. At this point, it will return a value of 0 changes to the main loop.

Overall, the heuristic with regard to CSE allows it to reduce the number of variables used within regions of the program, which in turn reduces register pressure and can increase performance since there is a lower chance of register spilling. By computing the value of an operation once and reusing the result as many times as possible, CSE also reduces the number of arithmetic operations which saves computation time. Finally, the reuse of its own temporary variables when possible prevents the introduction of unneeded copy-statements.

The heuristic also involves modifying the default behavior of copy-statement elimination to ensure a fixed point is reached. The main purpose of copy-statement elimination in this heuristic is to run right after CSE and reveal more common subexpressions through the propagation of values into or from copy-statements, but to not undo the benefits of CSE. To achieve this, copy-statement elimination will not propagate values that are assigned to the temporary variables introduced by CSE (_c#). In the case below, copy-statement elimination would not propagate a op b onward because of the assignment to _c0:

```
S1: _c0 = a op b;
```

```
S2: c = _c0;
S3: d = _c0;
```

If copy-statement elimination did not care about the _c# variable rule, it would see that S2 and S3 are copy-statements and are in the perfect form to propagate S1's right hand side down to S2 and S3. However, this would be counterproductive because 1) it would be reintroducing more redundant arithmetic operations that CSE just eliminated, and 2) if a and b are variables, it will have increased the register pressure by having more variables be accessed within the span. Ignoring this rule would also cause a back and forth effect where both optimizations continue to redo and undo each other's changes with each iteration, meaning a fix point would never be reached. By making copy-statement elimination not propagate these values, CSE takes precedence over copy-statement elimination and allows them to work together iteratively and still achieve a fix point.

In all other cases, if the copy-statement does not match the accepted forms (shown in the picture before), the module will not propagate the values onward. When it can propagate values, the hope is that it forms subexpressions that can be eliminated by CSE, thus increasing the benefits CSE offers. This propagation also has the potential to reduce register pressure, as the reuse of one equivalent variable instead of multiple equivalent variable means less variables need to be in registers overall. As mentioned before, copy-statement elimination will remove dead temporary variables. Besides removing the unneeded computation, this will also reduce the program code size. This positively counters a side-effect of CSE, since CSE adds more lines to a program with its temporary variable assignment. Finally, like CSE, copy-statement elimination will return the number of changes it made in one iteration cycle back to the main loop. When it finds it can't make any more changes, it will return a value of 0.

During this iteration loop of CSE and copy-statement elimination, if one of the optimizations makes at least one change to the TAC, another optimization loop will be done. When both have made zero changes in the current loop, a fix point has been reached and the loop will exit. The final optimized TAC will be in the Output/tac-opt.txt file. A second file Output/tac-opt-temp.txt will also be made, but this is used for internal processing and can be ignored. Because of this heuristic, the two optimizations can run back to back, in an iterative manner, without the issue of them undoing each other's work, and it guarantees a fix point will eventually be reached. It also allows them to benefit each other by revealing more optimization opportunities, which in turn increases the chances of reduced register pressure and removal of more redundant arithmetic operations. Since the focus of my optimization was to decrease the number of redundant arithmetic operations and decrease register pressure, I made the design choice to have CSE take precedence over copy-statement elimination. I determined that CSE can better achieve this goal when its changes to the TAC take priority over the ones made by copy-statement elimination. The tradeoff is that CSE will increase the code size with the addition of temporary variables, but the removal of other dead temporary variables helps counter this.

Compiler Backend:

In general, the compiler backend has changed very little from the previous labs. This module runs after the frontend and optimized TAC have been generated. As given in the lab requirements, this backend translates the internal TAC representation into a valid C program. It treats any user variables used without definition as user input and prints out the values of only the user variables at the end of the program. All variables that appear in the program are initialized with 0 at the start and labels are given to every statement in the program. This backend is called 4 times to create C program versions of the unoptimized and optimized TAC code and versions with timing functions. For timing, a for-loop with 500,000 iterations is placed around the code section to better see the timing differences between the optimized and unoptimized versions. Most programs would have too short of an execution time to measure the difference otherwise. The time taken to complete this loop is printed out at the end. After the backend is done generating the C files, the compiler has completed all steps and the program exits.

Results:

The overall results of this compiler implementation show that the calculator language is fully implemented and has support for up to 3-deep if/else-statements. 1 and 2-deep if/else statements are fully correct, but 3-deep if/else-statements must be limited to a specific format for correctness to be maintained. The data dependence module can find all three types of dependencies between each line for n-deep if/else-statement nesting and straight-line code. CSE and copy-statement elimination are fully implemented to the specifications I intended, and the heuristic that governs them properly ensures a fixed point is reached. These optimizations also work for n-deep if/else-statement nesting and straight-line code. All this functionality was verified using the many test cases I created, which can be viewed in the Tests directory. For every test, I verified each one produced a correct TAC form, recorded all the correct dependencies, and properly applied the optimizations. I also verified that the optimized code is functionally equivalent to the non-optimized version for each test case.

For the optimizations, performance tests were done to see if they provided the improvements I intended them to. I did this by compiling and running the timing versions of the backend output for the optimized and unoptimized TAC. Each version has a for-loop around the code sections with 500,000 iterations to help emphasize a timing difference. I compiled them with gcc -O0 to prevent any other optimizations from affecting the results. The results I collected were the run times and the size difference of each version in terms of line count. I ran this test on a handful of test programs and put the results in the table below.

In the presentation, I mentioned I originally saw only 1% to 3% speedup from the optimized output, which were the results of the simple and generalized tests I had at the time. Since then, I made some more improvements to border case coverage and was surprised to now see a significant speedup from the optimized program versions. Even with test programs where I didn't purposely insert common subexpressions and copy-statements, I now saw decent speedups. From the table below, the "opt" files were specifically created to show the potential benefits of my heuristic. They include many common subexpression and copy-statements that can be eliminated. However, the other files are just general tests.

Test File	Original Time	Optimized Time	Speedup	Lines removed
Tests/class_example.txt	0.001721	0.001345	1.28 = 28% faster	2 = 14% smaller
Tests/ifelse2a.txt	0.003480	0.002536	1.37 = 37% faster	4 = 10% smaller
Tests/test5.txt	0.003235	0.002931	1.10 = 10% faster	1 = 4.5% smaller
Tests/straightline2.txt	0.032369	0.027027	1.19 = 19% faster	8 = 30% smaller
Tests/opt1.txt	0.005577	0.003146	1.77 = 77% faster	4 = 31% smaller
Tests/opt2.txt	0.003038	0.002323	1.31 = 31% faster	5 = 13% smaller
Tests/opt3.txt	0.005810	0.003883	1.50 = 50% faster	12 = 29% smaller
Average	-	-	*36% faster*	*19% smaller*

As can be seen from these selected results, the heuristic I created is able to offer significant performance improvements over the baseline TAC generated by the frontend. The speedup is dependent on how many common subexpressions and copy-statements could be eliminated in the program, so there are potential cases where there could be no speedup at all over the original version. The speedup is also very dependent on the amount and type of redundant math operations that appear in the program. For example, if there were many multiplication, division, and exponential operations in the program, the speedup with this heuristic would be much higher compared to a program that only had addition and subtraction. The number of lines removed depends on the combination of how many dead temporary variables were removed versus how many CSE variables were inserted.

The speedup in these selected tests is due to two main reasons. The first is that interweaving variable reuse is minimized with copy-statement elimination and from the introduction of _c# variables with CSE. This in turn minimizes register pressure, since more of the same variables are being reused instead of a mixture of different ones. Since the same variable can stay in a register longer, there will be less register spilling and loading from memory, which is a time- expensive operation if it does occur. As mentioned before, the removal of redundant math operations also contributes to the speedup since the result is only calculated once instead of two or more times.

Another benefit of these optimized programs is that their code size is smaller in many cases. In all the tests I ran, the optimized version was in the worst case, the same length as the original program. However, in many more cases, the optimized program was at least a few lines shorter than the original program. This is due to the dead variable elimination performed by the copy-statement elimination module. Based on these two performance measures, my optimizations performed very well at the task they were intended to do. The heuristic allowed CSE and copy-statement elimination to work together, reach a fix point, and improve performance.

The only limitation of these results comes from the fact that the frontend can't produce all types of 3-deep if/else-statements. However, if it could, the way I programed the CSE and copy-statement modules would allow them to work without issue. As I show in several of my test cases in the Test folder, the heuristic can work correctly with all the 3-deep if/else-statements that the frontend is able to produce. There are no limitations present in the heuristic itself, in terms of correctness issues. Finally, the performance gains from the heuristic are entirely dependent on the input program, so there is no guarantee that the optimization will cause a significant improvement.

Conclusion:

Overall, I thought this project was difficult and that each task presented its own set of challenges. I essentially worked on this project every day since the time it was assigned up until the day it was due. A weakness of my project is that there are limited buffer sizes for my storage structures, which put a cap on the max size of the input program. However, I made these buffers large and added checks to prevent overflow from occurring. Another weakness is that it does not support all possible arrangements of 3-deep if/else-statements. Attempting to fix this issue gave me the most difficulty. Accounting for all the border cases in the data dependence task also gave me issues. Another weakness is that the CSE algorithm is not based on the strength of the operations, and it only considers whether the subexpression is used again. It is not as “smart” compared to other algorithms I saw presented in class, but it is functionally correct and able to have good performance gains in certain cases. If I had more time and resources, I would fix the 3-deep if/else issue and make the CSE algorithm smarter. The strengths of my project are that data dependence, CSE, and copy-statement elimination all work correctly and are compatible with n-deep if/else-statements. The heuristic I created is also successful in all the areas I intended it to be, and the resulting performance increase is significant for my test cases. Finally, the only error in functionality is when there is an assignment within a 3-deep if/else-statement. This will cause a malformed if/else nest to be generated. Other than that, I believe every other aspect of this compiler operates correctly and without error.