Benjamin Steenkamer
CPEG 621-010
5/1/19

Lab 3 Report: A Calculator Compiler Middle End – Basic Blocks and SSA

Abstract

The main goal of this lab is to implement a calculator compiler middle end that produces basic block and static single assignment (SSA) representations of an input calculator program. The motivation for doing this is to gain deeper insight into how a compiler determines and represents dataflow of a program through different forms of intermediate representations. The key issue being addressed is how to implement the algorithms that generate basic block and SSA forms from a frontend input.

The outcome of this lab is that I was able to successfully implement the algorithms for basic block and SSA forms as a middle end for the compiler we've developed over the past two labs. This compiler middle end takes an input program and writes out the final basic block and SSA forms as text files. The algorithms work correctly with up to two-deep if/else-statements, and phi functions are inserted only where needed with only the necessary variables as the arguments. The basic block generation was achieved by using the "find the leader algorithm," and SSA form generation was implemented with variable assignment tracking and phi function insertion that take into consideration the if/else-statement scope a variable is assigned in.

Detailed Strategy

General Program Flow

To construct this compiler middle end, I divided the system into three main parts: frontend scanning and parsing, basic block generation, and conversion to SSA form. The general flow of operations starts at the input program file. This file contains a program written in the calculator language that was developed over the past two labs. The language syntax (variables, integer constants, and operators), variable constraints (allow undeclared initial use with 0 as the initial value and case-insensitive names), and available operators (+, -, *, /, !, **, =, (), ?) are the same as the previous lab. The compiler starts by reading in this program file, line by line, and passes each one through the flex rules for tokenization and then through the bison rules for grammar rules to be applied.

In the bison rules, each tokenized line is converted to three address code (TAC) in the same way as the last lab, with temporary variables inserted as needed. When a single TAC line is formed, it is written to the frontend TAC output file and then sent to the basic block module to be converted and written to the basic block output file. After the basic block line is formed, it is forwarded to the SSA module where it is converted to SSA form and then written out to the SSA output file. This final file then contains the basic block code in SSA form. After this, control is returned to the flex and bison modules to generate the next TAC line from the initial program file and the cycle repeats.

There are three main output files from this compiler middle end. The first is `tac-frontend.txt`, which contains the normal TAC code generated by the bison rules. `tac-basic-block.txt` is the result of the TAC lines being formed into basic blocks with `goto`s connecting

them. The basic blocks are decided by the flow paths of the if- and else-statements generated from the `?` operator. `tac-ssa.txt` is the basic block code that has had the variable names changed to be unique for each new assignment and includes inserted phi functions where needed. `tac-ssa.txt` is the final output of the compiler middle end.

Two other files are also generated after the previous process has completed: `c-frontend.c` and `c-basic-block.c`. These are C code versions of the `tac-frontend.txt` and `tac-basic-block.txt` files that can be compiled with `gcc` and run. They are generated with the same code used in the previous lab for the compiler backend output. These files were used for testing purposes and are not part of the main compiler middle end operations. A C code version of `tac-ssa.txt` is not generated since it wouldn't properly run with the inserted phi functions. For this project, the phi functions are just placeholders for the actual internal logic that would decide which previous variable assignment to select. Instructions to compile the calculator compiler middle end, run tests, and compile the generated C code are in the `Makefile`.

Flex and Bison Rules

In general, the flex (`calc.l`) and bison (`calc.y`) rules used in this lab are identical or nearly identical to the rules for lab 2. The comments in the current code files and the previous lab report describe what each rule does in detail. For the flex rules, they are unchanged from lab 2. For the bison rules, the actual grammar syntax is also unchanged from lab 2. However, the associated C code for a few rules has been changed. The changes are the addition of function calls to the basic block module in the TAC generation functions. When a grammar rule recognizes an input statement, it calls functions in the calc.y file to generate a TAC line for that statement. When the TAC line is formed, a function call to the basic block module is used to pass the TAC line to the next processing stage. This passing to the basic block module is also done for if- and else-statements generated when a `?` operator is encountered. Beyond this, nothing of note has been changed in these rules.

Basic Block Generation

Basic blocks are constructed from the TAC lines passed in from the `calc.y` bison file. This module (in `basic-block.c`) uses the "find the leader algorithm," which is the same one described in the class slides, to determine when a basic block needs to be created. The conditions are: the first statement in a program is a leader, a statement that is the target of a branch is a leader, and a statement that immediately follows a branch is a leader. When a leader is identified, a new basic block can be created. As stated in the lab instructions, a basic block starts with a label in the form `BB#:` and always ends with `goto` statement(s) to other basic blocks.

When the basic block module detects a leader statement, a new basic block label is created and printed into the output file (`Output/tac-basic-block.txt`). Then the leader statement and code following it are printed to the output file underneath this label. When an if-statement is passed into the basic block module, the module will insert an if/else-statement in the output file. Inside the if- and else-statements, `goto`s to the next basic block labels are inserted. For instance, if an if-statement is found in `BB1`, the generated if/else-statement will contain `goto`s to `BB2` and `BB3`. Since the TAC is processed as a straight line, the next line passed in will be the contents of the if-statement. The basic block module will start a new basic block by inserting the label `BB2:`,

insert all the TAC lines from inside the if-statement under this label, and then insert a `goto BB4;` at the end, since that will be the label for the basic block following the else-statement. The else-statement is generated in the same way, with label `BB3:` being used at the start and a `goto BB4;` at the end.

The case for basic block generation is more complicated when nested if/else-statements are introduced. The main idea is that the same process described before can be used to generate most of the block tags in the nest, and it can be done up to a max of a two-deep statement (by two-deep, I mean an if/else-statement nested inside another if-statement). Since the TAC lines are fed in as straight-line code, the basic block labels and ending `goto`s can still be incremented sequentially through the structure, up until the outer else-statement is reached. The label for this basic block needs to match the label used by the `goto` in the else-statement at the end of the initial basic block. However, you can't just use the next sequential block label, as that wouldn't correspond to the label used by this previous `goto`. This previous `goto` didn't "know" at the time there would be nested if/else-statements that would push the sequential label counter forward. To solve this, the basic block label is generated by offsetting the inner if/else statement labels and `goto`s by one and recording the label this specific `goto` used. The label is then recalled when the outer else-statement is reached in the TAC. In short, the system ensures every goto and basic block label matches up to mimic the same flow as the original if/else-statement nesting.

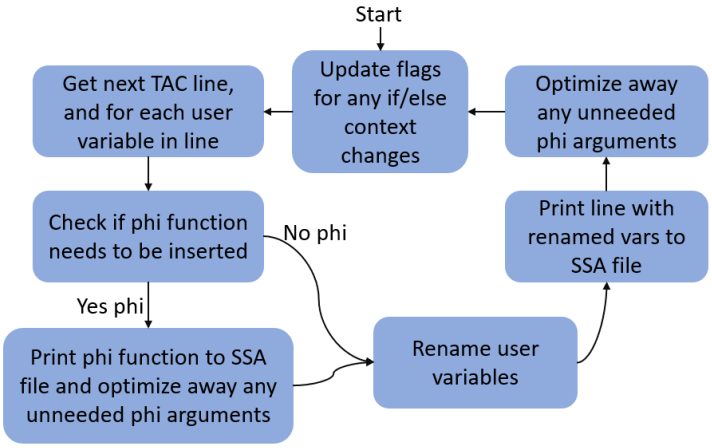
One behavior to note is that "empty" basic blocks can be created for else-statements in a certain case. This happens when nothing is assigned to the result of a conditional operator (e.g.: `(y)?(x+23);` vs. `z = (y)?(x+23);`). Because a variable isn't being assigned to the result, no variable can be assigned a value of zero in the else-statement. Thus, an empty else-statement is created, which is then translated into an empty basic block with just a label and a `goto` that immediately leaves the block. These empty blocks could be removed by doing a second sweep that deletes them and fixes any broken labels and `goto`s of the remaining blocks caused by the deletions. However, I decided to leave them in as it helped me see the form of the original program when testing the compiler and since the program is still functionally correct with the empty base blocks. The code will just jump into the block and then leave immediately with the `goto` to the next proper location.

A final point is that when the basic block module receives any TAC line, generates a tag, or generates an if/else-statement, it forwards the line(s) to the SSA module to be converted into SSA form. A special case is when the basic block module receives a TAC line that contains the opening or closing of an if- or else-statement. In this case, it forwards this change of if/else context information to the SSA module so that the SSA context tracker can update its flags. By doing this detection in the basic block module, it simplifies the code in the SSA module and allows the SSA module to have an accurate record of the current if/else context before it reads in the next TAC line.

SSA Form Generation

The below diagram gives the general flow of how a TAC line from the basic block module is processed and put into SSA form. In SSA form, each variable is defined exactly once. To achieve this from the original program, each variable is renamed to `originalName_#`. The number at the end is incremented by one each time the original variable is assigned a new value. Since there are

now unique names for every assignment, a special phi node or function is needed to join variables of the same base name together after branches occur. An example of the phi function format is `x_2 = phi(x_0, x_1);`. In the flow of the SSA module, any needed phi function(s) for the current line are inserted first, before the TAC line is processed. Then variables in the TAC line are renamed with the appended number. The phi insertion occurs first because it counts as an assignment to the base variable name, so this would affect the number to be appended to the renamed variable.



Phi Function Insertion

Phi functions only need to be inserted when a variable is being read from after it was written to inside an if- or else-statement (nested or not). Since it is possible that any given if/else branch was or wasn't taken, the phi function would in theory select the correct variable version from its arguments list that was on the actual execution path up to the current point. This allows the proper value to be propagated between each uniquely named version of the variable. This behind-the-scenes selection process for the phi function is assumed to happen, but the actual logic to do this was not implemented for this lab.

The general strategy for the phi function insertion logic is to track when and in what context (i.e.: outside or the depth inside and if/else-statement) a variable is assigned a new value. This was achieved by keeping an array of structs for every variable seen in the program. Each struct within the array contains the variable's current "ID" (the number appended at the end of the renamed variable) and an array (which acts like a stack) that contains all relevant, previous assignment IDs for the variable up to the current point. This entire system effectively mimics a control flow graph and dominance frontier for each variable. The SSA module tracks the final variable assignment when the input program leaves an if/else statement that the variable was assigned in. This variable instance's ID is recorded into the array with all other "phi arguments", which are then used in the argument list of the phi function when it is inserted: `phi(arg1, arg2, …)`.

When a variable is written to, either directly (e.g. `x_1 = 5;`) or through a phi function (e.g. `x_2 = phi(x_0, x_1);`), there is a chance to "optimize" or removed some or all recorded variable IDs in the phi arguments array. When a variable is written to, it acts as a join point for all previous definitions of the variable in the current context. Because of this, we can potentially remove all previously recorded assignments from the phi arguments array to minimize the number

of arguments that need to be included in the next phi function. For example, if a variable is written to entirely outside of an if/else-statement, that assignment acts as a join point for all other previous assignments to the variable. All combinations of previously assigned values to that variable can't propagate past this point, so the IDs stored in the phi arguments array for that variable can be removed and replaced with just the ID of this new assignment.

Another example is when a variable is assigned values in *both* an outer if- and else-statement (outer, as in, not nested if/else-statements). Since at least one of these two paths is guaranteed to be taken, this also acts like a join condition of all previous assignments. All recorded IDs in the variable's phi arguments array can be removed and replaced with the two IDs created from the if- and else-statement assignments. Then, the next time the variable is read from, a phi function can be inserted with only these two IDs as the arguments.

Several more optimizations like the ones mentioned above occur in other contexts throughout the SSA form generation. I will not list them all here, but you can view them in the `ssa.c` file within the functions of `_ssa_insert_phi_func` and `_ssa_assigned_in_guaranteed_path`, among a few other places. The code in these functions include optimizations and shortcuts used to clear out unneeded phi arguments and keep the number of phi insertions to the minimum needed for the correct propagation of values. A large amount of code in the `ssa.c` file in some way supports these types of optimizations, either with the tracking of contexts or testing for border cases.

Overall, this strategy of recording variable assignments as the program changes contexts and removing joined values was chosen because I felt like it was a simpler task than creating a full implementation of a control flow graph (CFG) with dominators. Since only if/else branches and no loops appear in the input calculator programs, the CFG method didn't seem necessary to me. The benefit of my method is that it was easier for me to conceptualize what needed to be done compared to the CFG method. The tradeoff of my method is that I had to make sure every border case was covered for every context. This was especially tricky for cases dealing with nested if/else-statements. Again, even though I didn't formally implement dominators and a CFG, my method achieves the same results for the given constraints of the input programs, with up to two-deep, nested if/else-statements.

SSA Variable Renaming

A core part of SSA form is that each variable is statically assigned a single time. Since the input programs can reassign values to the same variable more than once, a renaming of the variables needs to occur to enforce this rule of SSA. When reading in each TAC line, the SSA module will select out user variables and rename them as `originalname_#`. The default starting name of a variable is `originalname_0`. When the variable is assigned a new value, it will be renamed to `originalname_1`, and so on for each new assignment. When a variable is read, but needs a phi function inserted before it, this phi function counts as an assignment, so the variable will be renamed in the same manner.

Temporary variables are introduced in the frontend TAC generation stage before reaching the SSA module. Each temporary variable is uniquely named in the form `_t#`, with the number being incremented for each new temporary variable needed. Because of my implementation, temporary variables will only be assigned a value once and then read from zero or more times

within a single basic block. They will never be referenced again after their last use within the basic block. Therefore, it is unnecessary to rename temporary variables since they are already following the rule of a static, single assignment.

SSA Context Tracking

As mentioned before, context (or scope) tracking is an important feature that helps determine if a TAC line being processed is inside a one or two-deep if- or else-statement. Including this feature was a major design decision because it allows the recording of where user variables are assigned new values, and therefore makes it possible to know which arguments should be included in a phi function for the variable. For instance, if the context check determines that the program is leaving an outer if-statement, and variable `a_1` was written to inside that if-statement, it should save `a_1` to the variable `a`'s phi argument array for the next time a phi function needs to be inserted. However, the drawback of my context tracking is that it's not scalable if the compiler were to support more than two-deep if/else-statements. Even with only two-deep, it requires the use of six different states with specific logic for phi argument recording in most. Despite this complexity, I believe the system works properly for every border case that can occur.

Results

The overall results of this lab are that I was able to successfully implement a compiler middle end that takes an input calculator language program and produces basic block and SSA forms of the program as output. For the basic block generation, all lab requirements are met, including an algorithm that correctly finds and writes out each basic block. It also adds the necessary labels at the beginning and `goto`s at the end of each block. For the SSA form, my compiler can write out the basic blocks in proper SSA form. The variables are correctly renamed so that they are defined only once, and phi nodes/functions are explicitly inserted to the right locations. The phi functions are inserted the minimum amount of times needed, with no unnecessary phi function insertion exhibited. Also, only the relevant phi arguments appear in the phi function for each variable with the help of exhaustive border case checking.

To test the correctness of the compiler middle end, I created many test program files that exhibit a variety of variable and control flow behavior. They are in the `Tests/` directory. The file `TEST_INFO.txt` gives details on certain behaviors that are exhibited by variables in a selection of files. As I programmed each module, I thought of test cases that would trigger each feature I added. I created and ran the test files and made sure the results matched my expectations. I made sure the basic blocks had correct labels and `goto` statements that properly linked the program flow together. For SSA form, I made sure variables received a new name at every assignment and that phis were inserted in only the required join locations. Especially for the phi function arguments, I believe I had every possible combination of variable assignment propagation through nested if- and else-statements to make sure I had accounted for each edge case. After creating all these test programs, my compiler middle end successfully generated the correct basic block and SSA forms for each without any issues or unneeded phi insertions/arguments.

The limitations of these results are like the previous lab. They include a maximum of two-deep if/else-statements, a maximum variable name length (64 characters), a maximum combined number of user and temporary variables (128), and a maximum number of phi arguments recorded

at one time (256). These limitations are artificial and are due to either finite buffer sizes or in the case of nested if-else statements, an excessive increase in code complexity needed to increase the number of supported nests. I set all buffer sizes large enough so that a reasonable program would not reach them. I have checks in place so that the program will gracefully exit if a limit is reached instead of allowing buffer overflow to occur.

Conclusion

I thought this lab was difficult, but not as hard as lab 2. The strengths of my lab submission are that I believe it's functionally correct for every basic block and SSA form test case I could think of, and it is able to optimize away unneeded/joined phi arguments. It is also able to do this with up to two-deep if/else-statements, which I found difficult but satisfying to get working. Like lab 2, the main weakness of this lab submission is the complexity of my compiler logic. Specifically, the logic used to ensure correct functionality of all border cases for phi functions and nested if/else-statements took about 700 lines of code, and it was the part of the lab that gave me the most difficulty. I feel I could have made a simpler system to do the same task. Future modifications to this submission, if given more time and resources, would include cleaning up the previously mentioned logic and to create a system that would allow for nested if/else-statements deeper than two. As far as I know, there are no errors in functionality with this lab submission.