Benjamin Steenkamer
CPEG 621-010
4/15/19

Lab 2 Report: A Calculator Compiler Back End with Register Allocation

Abstract:

The goal of this lab is to create a calculator compiler backend for an extended version of the calculator language created in Lab 1. The main motivation for doing this is to learn how compilers take in a source code file, translate it to an intermediate form, perform transformations on it, and then produce a backend output in a new and efficient format. The key issue addressed in this lab involves the creation of the transformation stage of the compiler. Specifically, the option is given to create either a basic version of an instruction scheduling or register allocation algorithm.

Overall, I was able to create a compiler that reads in a calculator program file and translates it into C code with simulated register allocation. The result of this compiler is that it produces a functionally correct C code output that supports all required language features and has basic register allocation for the variables used in the program. Using Flex and Bison rules, the compiler frontend generates intermediate three-address code from the input. A register interference graph and heuristic optimistic algorithm are used to allocate registers to variables. Finally, this transformed code is outputted as C code that can be compiled and perform the same operations as the original input calculator program.

Detailed Strategy:

Overview of Calculator Compiler Operation:

Overall, the main strategy for implementing the calculator compiler was to choose methods that ensure correctness over efficiency. The overall flow of this compiler can be broken down into three stages: front end three-address code (TAC) generation, register allocation, and backend C code generation. The starting point is with the Flex and Bison rules used for scanning and parsing a program input file. This text file contains a program written in the calculator language defined for this class. The Flex rules (contained in calc.l) tokenize the input values which are then sent to the Bison rules (contained in calc.y). In the Bison rules, the code is parsed, checked for syntactical correctness, and user defined variables are tracked. As the calculator program is parsed, TAC is generated and written out to a file (Output/tac-frontend.txt). After the full input program is converted to TAC, the TAC is sent to the register allocation stage (reg_alloc.c and reg_alloc.h).

In the register allocation stage, a register interference graph is generated to assign variables to one of four registers (r1-r4). After this, the TAC is regenerated with variables being substituted with their register names. Loading and spilling of registers are also inserted into the TAC. Once complete, the new TAC file with register allocation is created (Output/tac-reg-alloc.txt). A simple optimization pass is then run on this new TAC to remove any redundant assignments introduced in the register allocation stage. The final, optimized TAC file is then produced (Output/opt-tac-reg-alloc.txt). This TAC file is then read in again to create the backend C code (C generation code is done in calc.y). This stage essentially copies and pastes the optimized TAC into a C file, adds variable declarations, inputs for uninitialized variables, code labels, and print statements for final user variable output (Output/c-backend.c and Output/c-reg-backend.c).

At this point, the compiler process is complete. To perform this operation, Makefile commands are used. Type `make calc` to generate the calculator compiler. Then type `./calc <input_program_file.txt>` to generate the TAC and C code files (there are sample program files in the Test directory). Finally, type `make ccode` to have gcc compile the backend C code into a runnable program (binaries will be placed in the Output directory).

Calculator Language:

The calculator language defined for this lab is very similar to the one defined for the previous lab. It only allows integer data types and can contain variables, constants, and operators. Variable names are case insensitive and are converted to lower case in the TAC and C code output. Variables don't need to be declared with an initial value, as they will have an initial value of 0. The allowed operators are `+`, `-`, `*`, `/`, `!` (bitwise not), `**` (power), `=`, and `()`. A new operator being introduced is `?`, the conditional operator. It is used in the form of (cond_expresion)?(expression) and behaves like the C ternary operator where (cond_expression)?(expression):(0). As a side note, input calculator program files must have both Unix line endings (LR) and a blank last line of the file to work correctly with the Flex and Bison rules.

Flex Rules:

Most of the Flex rules are very similar to the previous lab, so their design decisions will not be covered (see Lab 1 report for these details). Of the few rules that were changed, the variable names rule was edited to allow variables to start with an upper or lowercase letter because variables are now case insensitive ([**A-Z**a-z][A-Za-z0-9]*). Also, the `?` symbol was added to the operator rule to allow for the implementation of the conditional operator ([-+()=*/!**?**]). Also, constant values are now tokenized as strings instead of integers to allow for easier processing in the Bison rules. Beyond this, the design decisions of these Flex rules are the same as the last lab.

Bison Rules and Frontend TAC Generation:

The major changes and design decisions of the Bison rules were made to allow for easier generation of the three-address code. The first change was to allow input from a text file instead of stdin. Bison now sets yyin to the input file and calls yyprase() on this input file. Flex then tokenizes the values from this file and Bison interprets the grammar.

Going through the Bison grammar rules, when a variable token is encountered from the input program, it is "tracked" by adding it to a list of unique user variables. This list keeps track of every user variable and whether its first use was with an undefined value. This is saved for use in the C code generation in a later stage. When a variable is being assigned a value, TAC code is generated in the form of var_name = value;. This is written to the frontend TAC output file. Because variable names are case insensitive, they are all converted to lower case during the tracking and generation of the TAC output.

When expressions with operators are encountered (e.g. `expr + expr`), TAC code is also generated, but a new temporary value is introduced to hold the result of the expression. For example if the expression a + 1 is encountered, TAC code will be generated as `_t# = a + 1;`, where _t# is a temporary variable with a unique number. This new temporary variable is then added back on the Bison stack as the "result" of this operation. A downside to this method is that every operator expression will be assigned to a temporary variable, whether it needs to or not (e.g. a single line of `b = a + 1` doesn't need a temporary value). This creates potentially more temporary variables than are needed, but the tradeoff is that it simplifies the TAC generation process and guarantees every line has at most three operands. Lastly, a special case occurs when the unary `!` is used, which results in a TAC generation of `_t# = !expr;`.

The Bison rule for the conditional `?` operator is more complicated than the other rules. For TAC generation, the conditional operator is converted to an if-else statement, with the `cond_expression` placed inside the if-conditional, and the resulting `expression` placed inside the if-statement. When the first half of the expression (i.e.: `'(' expr ')' ?`) is encountered, the Bison rule will write out `if (expr) {` to the TAC file. In the second part of the rule `'(' expr ')'`, the TAC generation for the operations in `expr` will be done inside the if-statement.

In the last part of this Bison rule, it will push the result of the `expression` back to the Bison stack and increment a flag (`do_gen_else`) to state that a closing else-statement is needed. After this, at the end of every TAC variable assignment operation, this `do_gen_else` flag is checked. If it is set, it will decrement that value to zero to generate the matching number of closing else-statements in the TAC file (allowing for nested if/else-statements). By doing this at the *end* of a variable assignment operation, it will ensure that a variable being assigned to the result of `expression` will be inserted at the very end of the if-statement and before the start of the else-statement. A TAC line is then generated inside the else-statement to assign the triggering variable a value of zero, since in the case where `cond_expression` is false, the whole expression returns zero. If the result of the conditional statement is not being assigned to a value, then the Bison rule for variable assignment won't be called at the end of the if-statement. Instead, the `gen_tac_empty_else()` function call in the main `calc` Bison rule will be reached and close the if-statement with an empty else-statement with no variable being assigned a value of 0.

A final detail about the Bison grammar rules is that the `calc expr '\n'` grammar rule does not write out `expr` and instead throws it away. When the grammar rules reach this point, `expr` will be a single constant or variable. Writing this to the TAC file would result in a single value being written to a line with nothing else (e.g.: `var_name;`). While this is syntactically correct, it functionally does nothing, so it can be omitted without issue.

Register Allocation:

After the frontend TAC has been generated, the file will be passed on to the next stage, which involves allocating one of four registers (r1, r2, r3, r4) to each variable and ensuring they are properly loaded and spilled. When trying to ensure correctness of these operations in all cases, the register allocation code inside reg_alloc.c ended up becoming fairly complication. To be concise, I can't go over every border-case mechanism I created, but they are commented in the source code where it explains the purpose of each mechanism.

The register allocation process involves reading in the frontend TAC file, building a register interference graph (RIG), and then doing a forward and reverse pass to assign variables to registers that don't interfere with each other. The forward and reverse algorithm I used is the same heuristic optimistic algorithm from the class slides. To save room, I won't include the large flow chart in the report but know that my method is effectively the same as the one described in the slides. This includes the use of a RIG and stack, profitability to determine the may-spill nodes, and k=4 for the four available registers.

This algorithm was chosen for its initial simplicity of implementation. However, there are several downsides to my implementation. The first being that it treats the entire frontend TAC file as one base block, which means there can be an unnecessary amount of register interference between each variable. This leads to some or many variables not being assigned registers. The other issue is that, to enforce the local contexts of variables declared inside an if/else-statement, I had to add complex logic to track whether a variable had been assigned inside an if/else-statement. I also had to artificially limit the amount of nested if/else-statements to being two-deep to reduce the complexity.

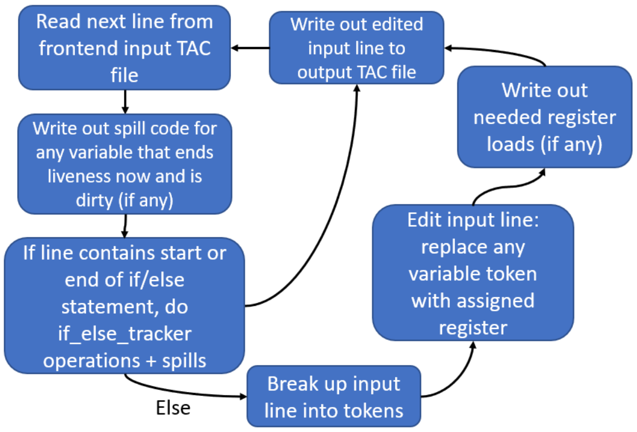
The first step in the algorithm above is to build the RIG. This is accomplished by running through the frontend TAC and determining the liveness periods of every variable. In general, a variable starts a new liveness period when it is assigned a value. Its liveness will start the line after the assignment takes place, since the register variable will not contain the value until after the line is executed. A variable's liveness period can also start because it is being read for the first time. The liveness period must start on the same line as the read so that the value is in the register at the time. A variable's liveness will end when it is read for the last time in the program, if it is assigned a new value, or in certain cases if the end of an if-else statement is reached. For the case where a variable is assigned a new value, a new liveness period will start following the line where the assignment occurred. During this process, the variable profitability (the total number of times the variable is referenced) is also calculated.

Once all the liveness periods are determined, each variable's liveness periods are compared to the other variables. If any one of the liveness periods overlap between two variables, they are said to interfere with each other and are added as neighbors in the RIG. This means that these two variables can't be assigned to the same register. The benefit of this interference detection is that its simplistic in its implementation. The tradeoff is that it causes a high number of interferences since it only cares if *any* liveness overlap occurs between variables, but not exactly *where* it occurs.

A caveat of these liveness periods is that if a variable is assigned a value inside an if/else-statement (i.e.: it starts a liveness period inside the conditional statement), that liveness period must end within the if/else statement and not be allowed to continue outside. I implemented this strategy because I saw that if a variable was allowed to continue its liveness period until the next variable assignment outside the loop, there could potentially be memory consistency issues. Say a variable is assigned a new value inside the if-statement and is then read soon after the if/else-statement. If liveness was not forced to end inside the if-statement, it would extend outside the if-statement and encompass the read. However, then say that if-statement isn't run (because the if-condition was zero) and so the expected value isn't placed into the variable's assigned register. When the line following the if/else-statement reads from the variable register, an old and unrelated value can exist inside the register from before the if-statement. In this scenario, the algorithm won't know to load the correct variable value into the register since it is still within a liveness period and therefore believes the register value is still valid. This will then cause an improper value to be read from the register and the results of the program would be invalid. The method I created to prevent this scenario uses several flags to check when an if/else-statement is entered or left and determines if a variable started its liveness before or during the current conditional statement.

TAC Generation with Registers and Load/Spills:

Once the register assignment is complete, the generation of the new TAC file with variables substituted for their assigned registers can begin. The below diagram shows a generalized view of the steps taken to convert the frontend TAC file and output TAC with the allocated registers.


Read next line from frontend input TAC file
Write out spill code for any variable that ends liveness now and is dirty (if any)
If line contains start or end of if/else statement, do if_else_tracker operations + spills
Else
Break up input line into tokens
Edit input line: replace any variable token with assigned register
Write out needed register loads (if any)
Write out edited input line to output TAC file

The first main design choice for this TAC generation algorithm is for the register loading and spilling. Loading a variable into its assigned register only needs to occur when a variable starts a liveness period by being *read* from. This is done by inserting a load instruction line (`_r# = var_name;`) right before the line where the register is used as the variable would. Every time a variable's liveness ends, and its register value is dirty, it will always be spilled back to the variable. See the example below where the left side is the frontend TAC and the right side is the TAC with variables substituted for their registers:

```
c = 1;              _r1 = 1;
…                   …
_t9 = c + 10;       c = _r1;         // Spill register
                    _r1 = _r1 + 10; // (c’s liveness ends here)
```

The spilling is achieved by inserting an assignment from the register to the variable right *before* the variable's liveness ends. Because of this, it allows for the overlapping of register assignment between variables within a single TAC line, which will prevent less spilling overall. Such is the case with _t9 and c both being assigned to register `_r1` without any interference. I understand that the lab assignment said to insert a storing instruction *after* the spilling of the register, but I found my method to be functionally equivalent while having the added benefit of overlapping register assignment. This method ensures the proper value is spilled back to the variable at the end of the liveness period, and the register will still contain the proper value for its final read use. This method also maintains correctness in the case where a register is assigned a value but is never read from before the end of its liveness. As mentioned earlier, a variable's liveness doesn't start until the line after it's been assigned a value. Since in this case the variable is also never used again, it's liveness will start and end on the same line right after the assignment. When the TAC line right after this assignment is encountered, the spill checker will see that the variable's liveness has ended and spill the register back to the variable right *after* the assignment has been completed on the previous line.

Overall, I consider this spilling strategy conservative because it always takes the safe route of spilling the register back to memory if the value is dirty. While this is functionally safe and guarantees correctness, there are cases where even if the register value is dirty at the end of its liveness period, it doesn't have to be written back to memory to maintain consistency. For example, the code `a = 1; a = 4;` and `a` is stored in `_r1`. My conservative algorithm would insert a spill right before the second assignment because a new assignment to `a` ends `a`'s previous liveness period. However, the value of 1 would immediately be overwritten by 4 anyway so the spill back to the variable was unnecessary. I ultimately chose to use this conservative method even though it is not efficient because it guarantees memory consistency.

Another important implementation detail about loading and spilling is that temporary variables (appear in the form of _t#) will never need to have their initial value loaded into a register because their first use will always be an assignment (initial value would just be overwritten). They also do not need to be spilled from a register at the end of their liveness period because their final values are no longer needed after their first and only liveness period in the program.

If-Else Special Cases for Spilling:

The last implementation details about the TAC generation with allocated registers are the features I added to ensure that proper spilling occurs for variables inside an if-statement. This implementation assumes there can only be two levels of if-else nesting. There are several issues that can occur when a variables liveness ends inside a conditional branch. For instance, if a variable starts its liveness period outside an if-statement with an assignment and is then used for the last time inside the if-statement, its value will have to be spilled inside. However, if the if-statement is not taken, then the spilling code would never be reached.

To solve this issue, I had to construct a spill tracker which determines the start and end of every if and else statement. It records all the spills inside an if-statement by variable registers defined before the current if-statement's context and then duplicates the spill code into the matching else-statement. This ensures that no matter whether the if- or else-statement is taken, the spilling of the register(s) will occur. Again, this works for up to two nested if/else-statements. So, if a variable's register was assigned a value outside a two-deep nested if-statement and then is spilled inside the inner if-statement, the spill tracker will insert spilling into both the inner else- and outer else-statements. This ensures that no matter which conditional route is taken, the spilling will occur after the variables last use in the straight-line view of the TAC. Also, if a variable was defined in the outer if-statement and then spilled in the inner if-statement, its spilling would be copied to the inner-else statement only. A variable defined and then spilled in the inner if-statement would not have its spill copied anywhere else because its liveness is contained entirely inside the inner if-statement.

Self-Assignment Removal Optimization:

After all these previous steps are completed, the TAC with register allocated variables will be produced (appears in Output/tac-reg-alloc.txt). However, because of the register overlapping mentioned earlier, there is now a chance to do an easy optimization on this code. For example, say that a line being converted in the register allocation TAC stage is `a = b;`, and both `a` and `b` can potentially be allocated to register `_r1` without issue. After the initial pass to convert this into TAC with register allocation, the line can now look like `_r1 = _r1;`. This is a redundant "self-assignment" and can be removed without issue. When this line is removed, variable `a` will still implicitly receive the value from variable `b` because `a`'s liveness starts right after `b`'s liveness ends. Since they were allocated to the same register, `a` will immediately take ownership of register `_r1` and thus gain the value inside the register. Because this optimization is safe and easy, a final pass is done on the register allocated TAC to remove any self assignments to produce the final, slightly more optimized register allocated TAC (appears in Output/opt-tac-reg-alloc.txt). This version of the TAC is then used to generate the backend C code.

Backend C Code Generation:

Once the register assignment and final optimization is completed, control will be sent back to the calc.y file where the code for the backend C code generation is. This section essentially copies the final TAC code into a newly created C file and adds a line label to every TAC line (excluding the non-executable lines of "`} else {`" and "`}`"). It also converts the `!` operator into the C `~` operator, and the `**` is replace with a call to `(int)pow(var1, var2)`. Before this TAC is inserted, it first writes out header file imports for needed system I/O and math libraries. It then declares integer variables for every user and temporary variable along with the four registers and initializes all of them to zero. It then goes through the array of variables that were recorded as undefined on their first use and inserts scanf() calls to ask the user to input initial values. The main TAC code is then inserted, which is followed by print statements that print the final value of every user variable. Final temporary variable and register values are not printed.

Results:

Overall, the frontend TAC, register allocated TAC, and backend C code all function correctly and equivalently. To test for this correctness, I created several test program files written in the calculator language (in Tests directory). Each test file exhibits certain behaviors and creates border cases that are meant to test every stage of the compiler. This includes, among other things, the use of every operator, long operation chains, interweaving variable assignments, and nested if/else statements. These were all done to test the correctness of the Bison rules, frontend TAC generation, RIG creation, register allocated TAC generation with spilling and loading, and the backend C output. All these tests were run, and the outputs of each stage were manually analyzed to verify the expected results occurred. Finally, the frontend TAC and register allocated TAC outputs for each test were converted into the backend C code, compiled, and run. It was found the outputs for both versions were correct and equivalent.

As mentioned before, performance of the TAC code was sacrificed for ensuring correctness in output. In this way, the TAC and backend C code can be considered inefficient due to the many loads and spills from the conservative approach I took in this area. The amount of tracking needed for each border case also makes the compiler somewhat slow. The compiler also has limits due to finite buffer sizes for things like variable names and numbers of liveness periods, and the number of nested if/elses is limited to two. However, I have many checks in place to warn when these limits are reached, and since the implementation passes all the correctness tests I created, I would say the loss in performance is worth it for the correctness.

Conclusion:

I thought this project was one of the most difficult projects I've ever had to do. My implementation quickly grew in lines of code as I realized how many potential issues could arise within if/else-statements. The weakness of the project is the complexity and inefficiency of the compiler source code and the inefficiencies introduced into the TAC code. However, I hope the strength of this project is that it accounts for all the border cases I could think of and that it produces functionally correct C code compiled from the input calculator program. Future modifications would be to improve the efficiency of the trackers and generation of the register allocated TAC and then introduce new methods like register coalescing and multiple base blocks to improve the TAC efficiency. The register allocation was by far the most difficult part of this project, but even the compiler frontend gave me difficulties with the conditional operator. As far as I know, there are no errors in functionality if the limits previously mentioned are not exceeded.