April 6, 2016

Checkpoint 3 - Achievements

Brandon Tan
George Chapman-Brown

Achievements

At this point in time, we have most of the code generation completed. Aside from problems with variables inside local scopes and arrays, the code generates properly. Functions work as intended, variables are returned on the stack, and memory seems to be allocated and removed properly from the stack.

Work Division

For this assignment, Brandon was responsible for implementation and George was responsible for documentation. This division was based on time limitations and the difficulty in managing a project on git with multiple contributors on the same git project.

TMS Simulator

The language we generate is extremely simple assembly code, with only 17 instructions. Since it will not be passed through an interpreter, it does not have access to labels, which would make the code more manageable. This is only a small drawback because the language is not designed to be worked on by hand. This is only an issue because the lack of labels means that we require backpatching in our implementation to create jumps to the not-yet generated end of scopes. Luckily, we can hold off on generating the instructions, because the simulator runs them based on the given id of the lines, not based on the order, so we can store the line we need to put the jump on, and come back to it when we find out where the jump needs to end up. Another problem with this language is the lack of functions to manage the stack. Without these it becomes difficult to sort out the stack needed for recursive function calls, since registered need to be changed and updated manually to reflect the relative changes in the stack that would be done by pre-decrement and function-return instructions. Lastly, it’s support for only register to register math makes it extremely difficult to manage small calculations. Intermediary values have to get stored on the stack so that they can be used for later calculations, and then moved down into registers later. Despite all these problems, the the simple structure of the instructions makes them easy to generate from a machine, and the lack of any need to edit and read code by hand justifies the decision to use a simpler language for a simple compiler.

Implementation

In order to implement our Code generation, we created a new class called CodeGen.java. This file is responsible for printing and managing the code. It keeps track of the current line number, and is responsible for printing the lines of generated code. It contains key constants like which register is used for what. It also contains a stack for keeping track of code jumps for backpatching. This is necessary because the program will need to know where in the future it will need to jump to, and it does not know the size of the block it will need to jump passed. To generate the code, the show tree code in absyn is used to visit each of the nodes of the abstract syntax tree. Each node of the tree generates lines of code necessary to handle that node, and sends them to the code generator to add the line number and other necessary information.

Generated Code

In order to implement C-minus code in machine language, a stack is nessicary for function calls. Without stack-based function allocation, it becomes impossible to run a subroutine an unknown number of times. For this reason, we implemented a stack. Our stack grows up from 0. Each variable creation places that variable on the stack, and increments the stack pointer to point at the new head. A frame pointer is also placed on the stack in order to locate variables in the current function definition. Along with the normal stack, a stack of variable locations is placed at 1024, and grows up. This keeps pointers to variables on the stack, to make finding them easier. This grows and shrinks as scope changes, so it will match the variables pointed to in the current scope. We use the registers as follows to implement this language. Register 7 is the program counter, which increments for each instruction it goes through. Register 6 is used to point to the bottom of the stack, 5 is used for the frame pointer for getting current registers, and 4 points to a table of pointer locations.

Improvements from checkpoint 2

Along with code generation, we added type-checking improvements from our previous code. Array initialization now ensures that the size of the array is an integer, as it does not make sense to declare an array with any other type as the size parameter.

Next Steps

The next step for our code generation would be to optimize the generated code. By planning register use we could save a great deal of time in an implemented language. The common case for finished products is to “compile once, run many times”, so optimizing the code to run quickly is worth a great deal of time at compilation. Another good optimization is to replace tail recursion with loops, which would replace a function call with a simple go-to statement. This saves stack space, as well as a great deal of run time from recalling pointers and cleaning the stack.