Lab 4 Written Report

Written by:

Eric Elmoznino

Abhinav Rajaseshan

Pranit Wadkar

Documentation

All lexical analysis code resides in the scanner.l file. No additional code has been added to the starter version, other than parsing the ‘,’ token, which was not there beforehand.

Syntax parsing code all resides in parser.y. We used bison to define the grammar and parsing rules, and no additional c helper functions. The sections are divided as follows:

- token declarations
- precedence and associativity declarations
- grammar rules

When bison reduces a grammar rule, we print out the matching rule and initialize a node in the AST.

The AST structure definition and functions reside in ast.h and ast.c. Semantic analysis code resides in semantic.h and semantic.c, and the symbol table used for semantic analysis is defined in symbols.h and symbols.c.

All code generation is done in the codegen.c and codegen.h files, which have a single public function to generate machine code from the root of an AST and several helper functions.

Breakdown of Work

Work was broken down as evenly as possible with the following responsibilities:

- Eric: Wrote helper functions for code generation and wrote test cases.
- Abhinav: Worked on the main code generation function.
- Pranit: Worked on the main code generation function.
- All: Everyone contributed to debugging and the writing of this document.

Overall Approach

Code generation was done recursively. Given an AST node, it would check its node kind (e.g. unary operation) and print code to a text file as well as call the function on its children nodes. For convenience, the main function returned an integer representing the temporary variable count corresponding to the result of the operation. For example, given the operation 2*4*6, we had to multiply 2 and 4, store it in a temporary register, then multiply the value in that temporary register with 6. The easiest way to do that was to recursively call the code generation function on the child binary expression, generate the code for that multiplication and store the result in a temporary register, then return the integer id of that temporary register so that the parent binary expression could complete the operation.

Obviously, not every node kind required a temporary register and it would have been an excessive use of resources to create one. For example, had the operation been 2*a instead, “a” would not require an additional temporary register because variables are already stored in registers when they are declared. So in this case, the recursive call on “a” would just return 0 instead of a temporary variable index and the parent call would use the register name corresponding to the variable name.

Non-Trivial Math Operations

Certain math operations had no corresponding assembly instructions. In these cases, we used multiple assembly instructions combined with temporary register to get the final result. For instance, to divide two numbers, we first used the REC instruction on one the second operand and stored it in a temporary register, then we used MUL to get the result.

Boolean Types

Boolean types were encoded with 1.0 for true and -1.0 for false. This symmetry simplified Boolean operations such as AND and OR because we were able to add up the operands getting possible results 2.0, 0.0, and -2.0, and then compare that result to what the thresholds for the operation were. For instance, AND would require the result to be 2.0.

If Statements

Implementing if statements was the most challenging part of the lab that we encountered. We eventually realized that the only implication of if statements is that we didn’t want to do

assignments to variables in clauses whose condition wasn't met. For instance, if we had the statement if(1) {a=1;} else (a=3;}, then we didn't want to do the assignment in the else clause. We thought that the easiest way to keep track of where we were in the code given the possibility of nested if statements was to use a stack. When entering or exiting parts of an if statement, we would push or pop states corresponding to the 'then' and 'else' clauses. We would also store the result of the 'if' statement's condition. Doing this, we were able to check if we were in an 'if' statement during a given assignment and what clause of that 'if' statement we were in, which allowed us to make assignments based on the condition of that 'if' statement.

Constants

We handled all semantic checking for constants within our previous lab, so there was fairly little to do differently for constants. The only difference was that we created registers for them using the PARAM keyword and made sure that they were assigned values in that same line of code.