Justin Canton, 1000017910
Samprit Raihan, 998138830

Compilers and Interpreters CSC467 Lab 4
Code Generation Documentation

The implementation of this lab required us to generate ARB Assembly code at each node in our AST. Each node was handled differently.

The SCOPE_NODE was implemented by just propagating the generateAssembly() call through both sides, the declarations and the statements.

The BINARY_EXPRESSION_NODE had to be handled differently for each type of expression as well as what was on the either side of the binary operation. If there was only a variable of a literal value on both sides of the equation, then these values were just placed right into the operation call, such as ADD a, b, 4;. But if there were other binary expressions or function calls on one or both sides of the operand, those needed to be resolved and placed into a temporary variable before the original binary expression could be evaluated.

The UNARY_EXPRESSION_NODE was handled mush the same way that the BINARY_EXPRESSION_NODE was. Based on what was on the side of the operator as well as what the operator was, the action taken changed.

The INT_NODE just printed out the integer value whenever it was called, allowing us to place this value directly into assembly code calls.

The FLOAT_NODE also printed out the value directly.

The BOOL_NODE changed what was printed depending on whether the input was true or false. If the boolean was true, it would print out 1, which we made to represent true. If the boolean was false, it printed out -1, which we made to represent false.

The VAR_NODE printed out the name of the variable, but it also made sure that if one of the predefined variables was called, it would change the name to what was required.

The DECLARATION_NODE had multiple different functions. As with the BINARY_EXPRESSION_NODE, if the expression on the right hand side was either a literal or a variable, it just directly equated this the new variable that was created in the declaration. If there was a binary expression or a function or a constructor, it resolved these first, before equation the results from that to the new variable create. Lastly, if the variable is a const variable, it would declare that variable with the PARAM function in the ARB assembly code which represents constants.

The IF_STATEMENT_NODE declared a condVar variable that store whether the condition was true or false. This condVar was then checked against anything inside the if statement. If the

condVar was > 1, meaning true, then using CMP, the new value would be stored into the variable, otherwise the old value would be restored.

The STATEMENT_NODE functions like the SCOPE_NODE, just calling generateAssembly() on both sides of the equation to pass along function.

The ASSIGNMENT_NODE had to take into account what was on the right side of the equation as well as if the line of code was within an if statement. If the right side of the equation was not a variable or literal, it needed to resolve the expression before trying to assign the new value. If the expression was within an if statement, it needed to check all the condVar variables before assigning the new value to the variable.

The FUNCTION_NODE calls the required function and then depending on what the input is.

The CONSTRUCTOR_NODE creates a variable and assigns the values that were input into the correct variable component.

The NESTED_SCOPE also functions like the SCOPE_NODE, just calling generateAssembly() to pass along function.

The DECLARATION_NODE functions like the SCOPE_NODE, calling generateAssembly() on both sides of the equation to pass along function.

The ARGUMENTS_NODE resolves what the value is, as well as keeping track of the component that the resolved value should be placed in.

The ARGUMENTS_OPTS_NODE passes along the generateAssembly() function to the ARGUMENTS_NODE for that to do the work.

The EXPRESSION_NODE passes along the generateAssembly() function since there is nothing for it to do at this node.

The VARIABLE_NODE also passes along the generateAssembly() function since there is nothing to do at this node.

Each math operation had to be implemented differently.

For addition, the ADD function was called, and similarly for subtraction, the SUB function was called.

For multiplication, the MUL function was utilized, but for division, both sides of the operation were placed in temp variables, the RCP was called on the right side to determine the reciprocal. These two values were the multiplied together.

When computing a power, the POW function was called.

When an AND function was used, the two booleans were added together, then 1 was subtracted. Then using the CMP function, unless the value was > 0, the return was false.

When making the OR function, we took the maximum of the two values using the MAX function, and then, using the CMP function, if the value was 1, then true was returned, otherwise false was returned.

The LEQ and GEQ functions were implemented using the SLT and SGE assembly functions respectively.

The greater than and less than functions were implemented by subtraction the right side from the left then using CMP to determine if the value was less than, or greater than.

Lastly, the equals and not equals functions calls SGE on the right side and left side, then calls SGE on the left side and right side. Then multiplies these two values together then multiplies this value by -1. Then using CMP, it return true or false.