March 23, 2016

Checkpoint 2 - Achievements

Brandon Tan
George Chapman-Brown

Achievements

For Checkpoint Two, we have implemented a simple symbol table. We have implemented this as a stack of hashmaps, with each hashmap representing a scope. As we enter a new scope, a new hashmap is created, and new variables will be placed in it. This happens because we use a stack structure to implement nested scopes, and we place the new variables in the top hashmap. When we leave the scope, the top hashmap is popped off the stack, losing access to the variables stored within it. By ensuring we have a pop for every push operation, the stack will contain only the global scope when the scanning is completed.

Implementation

We have added three classes to our Checkpoint One submission to implement the semantic map. The first is the Symbol Table itself, called SemanticHashmap.java. This contains the stack of hashmaps, as well as functions to add and remove scopes by popping or pushing the stack. It also includes codes for adding variables and functions to a region, and type checking functions. The next class is Identifier.java, which holds a variable identifier, which is simply a variable name and a type. The last class is FunctionIdentifier.java, which inherits from identifier. It contains an arraylist to store the function arguments in addition to the Identifier code. By adding simple identifiers and

function identifiers to the hashmap, we construct a table that can be used for code generation and type checking.

Building the Semantic Map

We leverage the syntax tree code from the last assignment to build our symbol table. Since this already visits the entire tree, it's a good starting place to start building the table. We added code to error check and add variables and change scope to different nodes on the ast.

Work Division

Brandon coded the Semantic hashtable, which included adding scopes, adding variables and functions, and type checking code. George completed most of the calling code to add variables and functions into the table, with some debugging from Brandon. Brandon implemented the type checking code, and George did most of the documentation.

Type Checking

Our code checks many different variable errors. The simplest type is checking whether the variable or function exists in its current scope when it is used. If it does not, an error is generated. Another error we catch is that you can't do any math operation on a void, or any comparison operation. We check each expression up the comparison tree to make sure each operation is the same type. When functions are called, we check

both whether they exist and that the correct number of variables are provided as parameters to the function. We also check the that only integers can be used to dereference arrays, as void won't make any sense inside the square brackets. The assignment operator is type-checked as well, assigning values up the expression tree until it gets to the assignment itself, then checks both sides of the operation. This works for both function calls and mathematical operators.

Code Demonstration:

We have five C-minus programs with various levels of errors. The first one, 1.cm, has no errors to demonstrate proper function. 2.cm contains erros related to void type matching. It attempts to use a void variable to dereference an indexed variable using a void variable, which fails as expected. Then a void is assigned to an int variable, and an indexed variable is used with no index. Next is 3.cm, which checks mathematical operations using void variables and int variables, which fails due to type checking. 4.cm checks assignment of a variable from a function call which returns a different type, and a function with the wrong number of arguments. 5.cm tests function calls with wrong number and type of variables nested together, which causes many errors to occur.

Improvements from Checkpoint 1

We have also fixed errors that were present in checkpoint one. This includes merging operations in the grammar to simplify it. This includes merging relop and operations. We also fixed the order of operations to match with the rules of mathematics.

Going Forward:

The next step will be to implement code generation. This will use the generate map to create machine language code to implement the instructions. This will often use 3-point code, which uses a temporary variable to store the result of the given math for later use. The map we generate will be used to create local storage, as our language is statically allocated, so all the memory allocation will have to be pre-allocated. We will generate variables on the program stack, and the programs will use these locations to refer to the variables involved.