Compilers Project Checkpoint 2

CIS*4650 - Compilers

Benjamin Bliss (1146484)

Uladzislau Kaparykha, (1096425)

Checkpoint 2 progress

In this project we are developing a compiler for the language C-, a simplified version of C. For this checkpoint, two major components were completed, implementing a symbol table to store all declarations with their scope and type checking which uses the symbol table to validate the type of an existing symbol when it gets called. These components lay the foundation for further stages of compilation, bringing us closer to a fully functional C- compiler.

The first step in working on this checkpoint was to build the symbol table. We decided that the best data structure for this class would be a stack of hashmaps. The hashmaps used strings as the key, holding the identifier and a new class Sym which held a string for the name, Dec for the AST declaration and an int for scope of the declaration. We chose a stack of hashmaps as it streamline the process of entering and exiting scopes in a source file. Such that when the program starts, the first hashmap gets pushed on representing global declarations and each time a local scope is entered a new hashmap is pushed to hold variables declared in that scope. Whenever a scope is left, the top hashmap is popped from the stack and its contents are printed to show all declarations in that scope. It should also be noted that function declarations are stored in a separate hashmap than the variables to allow functions and variables to share

names. This is also only a hashmap and not a stack as functions are restricted to being global in the C- grammar. The foundation of the SematicAnalyzer was also built here, which is structurally similar to ShowTreeVisitor of the last assignment, traversing the tree in post order. However, instead of printing out the contents of the tree, it inserts symbols into the symbol table and reports errors if the symbols are already contained in the current scope.

As for type checking, all of the individual checking was done pos-order within visit() nodes of SemanticAnalyzer. The contents of the node would be recursively visited, and then type checking would be performed. Type checking mainly checks if the provided contents is of ArrayDec type, if yes, raise error. Afterwards, where applicable, it checks if dtype (predefined in Exp class) is of a correct type, given the context (eg. must be either int or bool for a test condition of an if-statement, etc.) In terms of CallExp(), we do a hashmap lookup to see if it’s a valid function, do appropriate checking to verify provided ars match the datatype of the provided function; and then set dtype of CallExp to return value of predefined function s.t. callee can use dtype to do appropriate checks. In terms of ReturnExp, we keep track of the function name's current scope as a class-wide variable, and then use that to do a hashmap lookup to validate the return type.

What we learned

Through the process of implementing Checkpoint 2, we gained a deeper understanding of how compilers track variable and function declarations, ensuring that they are used correctly throughout a program. We learned about the possible structures

of a symbol table, in our case a stack of hashmaps, but also alternatives such as a hashmap<string, arraylist>. The second one may have better memory efficiency but we found our solution was more elegant and intuitive. This also improved my understanding on how to handle scoping rules, ensuring that variables are resolved correctly within nested blocks and function calls.

Through implementing Checkpoint 2, it became more apparent just how nitpicky the type checking has to be, and how many cases need to be considered while implementing it. There were a number of features (see Limitations) that had to be skipped because they were WAY outside the scope of C-minus, a simpler version of C. But if those features were implemented, a lot more care would have to go into making the SemanticAnalyzer more robust (eg. verifying all branches of a function lead to a return).

Work Breakdown

Symbol Table: Ben

Type Checking: Vlad

Assumptions/Limitations

With regards to the progress made in this checkpoint, there are many assumptions and limitations, however most are not required per the assignment description, however it is worth mentioning anyways. These include, called functions will not check if there is an implementation (only a prototype). We include the input() and output() functions in our symbol table from the start so no error is raised when they are

called as long as the arguments are correct. We implemented int to bool conversion similar to C. Our program does not check all branches of a function for a return. The compiler will only give errors and no warning. Our program checks if there is a main function available as it is required by the C- language. Finally, when running the compiler, one of the following flags (-a, -s, -c) must be provided to choose what the compiler is to do, -a outputs the AST and -s outputs the AST and Symbol table. -c is not implemented yet (content of Checkpoint 3), but the program supports it.

Possible Improvements

Possible improvements would be to resolve the previously mentioned function call limitation, adding a warning if prototypes do not have implementations and raising an error if function calls lack implementations. From the optimization side of it, our symbol table is less memory efficient when compared to a hashmap<string, arraylist> implementation, so changing it to that would provide a more optimal symbol table.