**GROUP 15 - CHECKPOINT 3 DOCUMENTATION**

Sam Engel, Affan Khan, Wisam Dayoub

**CHECKPOINT OVERVIEW**

For this checkpoint, we made modifications to the semantic analyzer and created an assembly code generator. To accomplish this, we added a new visitor class CodeGenerator.java, containing assembly-code emit functions that will be called as each node in the abstract syntax tree is visited. Moreover, SemanticAnalyzer.java does some additional preparation for code generation by linking variables to their declarations, function declarations to their prototypes, call expressions to function declarations and prototypes, etc. In total, our final product for this semester is a program that parses C- code, semantically analyses it, compiles it into assembly code, and reports errors along the way.

**DESIGN PROCESS**

For our design process, we followed the incremental Subtask list described in Lecture 11 – TM Simulator. But our design process was also informed by referencing the sample output in the TMSimulator package found on CourseLink. This helped fill in the blanks whenever the description offered by the lecture slides felt lacking. In particular, our attempt to follow the advice for assignments and operations from the lecture slides proved troublesome, and we found that reverse engineering the methodology from the sample assembly code output was easier to implement correctly and less convoluted. By contrast, the way that the sample assembly code handled While and If control structures seemed needlessly complicated, so we opted to design loops as described in the lecture slides instead – which was more straightforward.

Since we are much more accustomed to higher-level languages, trying to become reacquainted with the mechanics of assembly code proved to be a great source of friction when it came to completing the project. But eventually, one does get a sense for it during the testing/debugging process.

Over the course of the project, the steps we took were roughly as follows:

Sam Engel’s contributions

1. Completed Subtask 2: refactored the syntax trees and visitor interface.
2. Worked on Subtask 3: implemented emit routines, emit location and frame offsets, attempted to implement the assignment and operation methodology from the slides. Implemented standard prelude and finale.
3. Completed Subtask 4: implemented code generation for If and While control structures.

Affan Khan’s contributions

1. Worked on Subtask 3 & 7: implemented global offset incrementation. Set offsets for arrays. Finalized ReadMe, documentation and Test Files.
2. Tested and reconfigured CodeGenerator.java to match sample outputs. Added comment emits to resemble the sample output. Also made changes to the assignment and operation to match the sample output.
3. Fixed parameter error checking in SemanticAnalyzer.java, added support for other operations in CodeGenerator.java.

Further contributions from Sam Engel

1. Made further changes to assignment and operation methodology to mimic the sample output’s way of doing it.
2. Tested fac.cm, gcd.cm, and mutual.cm. Fixed bugs and added support for booleans and function prototypes.
3. Finished Subtask 6 & 7: implemented code generation for inner blocks and array references as well as runtime error checking of the array index. Tested and debugged sort.cm

Wisam Dayoub’s contributions

1. Created test files [1-3].cm that include no error as per assignment description. It shows the normal behavior of the program when there are no errors. They are intended to ensure that base test of our program produces valid assembly code for the TM
2. Created test files [4-8].cm. Those tests are intended to catch different types of errors such as different type mismatch errors, lexical errors, tests for undefined variables and/or functions, as well as redefinition errors.
3. Test files 9 and 0 tests for a combination of the errors above as well as return type mismatches. The 0.cm file has multiple errors to test if the program works with as many errors in the file. It also tests for invalid conditional statements.
4. Finalized readme

**ASSUMPTIONS**

* A function prototype will be treated as a global declaration; the globalOffset will give it a space in dMemory – which will hold the value of the address of the function proper. During code generation, if a function declaration has not been visited yet, any calls to the function will be instructed to load the address value held by the prototype’s space (the space in memory will be filled later with the address once the function declaration has been visited).
* Arrays are assumed to have a static size.
* Upon entering a nested block, the frame offset will change to accommodate for variables declared within. Upon leaving, the offset will revert, leaving the memory spaces reserved for those variables free to be overridden.
* It is assumed that non-special registers are to be used freely and are used in some operations such as retrieving the size of an array during runtime error checking and for UMINUS operations
* The UMINUS operator is used for operations of the following: NilExp – IntExp. In an assignment, this will look like: a = -4
* Assumed that input() functions will have no inputs

**LIMITATIONS**

* While the compiler will attempt to catch errors, this is by no means comprehensive and will only cover error conditions that seem most obvious.
* The bitnot operator ~ is filtered out at the grammar level and is not supported during the execution of any programs
* The “-a”, “-s”, and “-c” flags are mutually exclusive. When running the program, you have to choose one or the other. However, the “-c” flag runs both the semantic analyzer and compiles the program.
* Variables must be declared at the beginning of a function or else the parser will throw an error