# CIS*4650 Checkpoint 2 – Documentation

## Overview

This document outlines what was done for the second checkpoint in the C- programming language compiler implementation. In this checkpoint we build a runtime semantic analyzer using a symbol table and type checker. The program reads a C- file line by line and outputs the corresponding symbol table reporting any semantic errors that arise along the way.

## Assumptions

The most important assumption made for this implementation was that a *void* typed variable was a valid definition in C-. We assume that this type of variable can't be assigned any value even if the value of that assignment expression of type *void*. Additionally, we assume that a void variable or function call cannot be an operand in any expression.

## Limitations and Possible Improvements

This semantic analyzer can only handle semantic errors related to the C- language and those defined in the checkpoint two specification. Currently the analyzer can only validate integer expressions. All other types besides *void* and *int* remain unhandled and out of the scope of this assignment. For function return expression checks we check only the first expression found in the function, regardless if that expression was found in a subblock. Thus, flow control checking is out of the scope of this assignment.

Possible improvements include making more descriptive error messages and throwing/handling custom made exceptions instead of sending error messages statically.

## Design Process

Given the abstract syntax tree generated in checkpoint one we can navigate our program at runtime and check for semantic errors. We build this semantic analyzer by first constructing a symbol table to keep track of the program's declarations as they occur. Once we have this information we can begin to enforce semantic rules such as using only declared variables/functions, no name collisions in the same block, and using valid operands within expressions and function calls.

### Symbol Table

The global symbol table maintained when navigating the parse tree contains an updated account of all declarations and the scope they're in. As we move through the program and come across function or variable declarations we push them to the current block in the symbol table (most nested). If we enter any type of compound expression we create a new block in the symbol table. This becomes useful for semantic analysis when we attempt to access variables from higher scopes.

### Type Checker

The type checker verifies all types in the C- program. Given a function declaration it ensures that the return expression matches the expected return type. Given a function call it ensures that the argument list provided matches the parameter list both in length and type for each argument-parameter mapping. Finally, the type checker verifies that a given expression evaluates to a specific type (usually integer).

### Semantic Error

The semantic error class is simply a library of error messages that are printed when called during semantic analysis.

## Implementation Process

We start by constructing a **SemanticAnalyzer.java** class that implements the same *AbstractVisitor* interface as *ShowTreeVisitor* in checkpoint one. This will be used to navigate our program parse tree at runtime. The *Semantic Analyzer* is composed of a *Symbol Table* and *Type Checker.*

### Symbol Table

**SymbolTable.java** is a utility class with the sole purpose of manipulating an *ArrayList* of *HashMaps*. The list allows us to store different blocks of declarations in stack order. Each element in the list corresponds to a block in the program. We can add a new block by creating a new HashMap and appending it to the list. Similarly, we can remove a block in stack order by removing a HashMap from the back of the list. Each HashMap contains a mapping between identifiers (*String*) and their declarations (*Declaration*). The declaration is a custom type which contains the name of the type declared and a reference to the declaration object (*Dec*). Since the **Dec.java** class encapsulates any declaration we might need (variable and function) we can get declarations from the symbol table with a simple name lookup. To check if a symbol is defined, we can iterate over all blocks and perform a lookup on each one. We can fetch the closest symbol (in the nearest block) by iterating over the list in reverse order and performing a lookup in the table.

### Type Checker

**TypeChecker.java** is also a utility class that is composed of a symbol table. Given a general **Exp.java** to type check against and whether the type we are looking for is an integer (boolean), the type checker delegates type checking to one of its many overloaded methods. Each of these overloaded methods will perform the type checking for a concrete type such as *CallExp*, *IntExp*, *OpExp*, *ReturnExp*, and *VarExp* who each enforce their own specific type checking rules. Additionally, the type checker supports type checking for function declarations and parameter/argument lists. The main idea behind the type checker is to recursively compute type checking for expressions, evaluating types for sub expressions before returning the value of the root expression since these will determine a correct evaluation.

### Semantic Error

**SemanticError.java** is a class you can call from any program to report a semantic error. There are a variety of static errors defined such as errors for invalid array index expressions, function call argument lists, function return expressions, variable/function redeclarations, undeclared variable/function use, void expressions, and assignments to void types. Each method in the class constructs a String containing the specific error message and delegates printing the message to standard error with the line and column number to a private helper function **printErrorMessage**.

## Lessons Learned

Recursive programming is often desired when evaluating expressions that involve various subcomponents such as **Exp.java**. Exception handling with custom defined exceptions are usually preferred over manual checking as this can help you catch errors in order of appearance.