CIS*4650 - Compilers Milestone 2

Boris Skurikhin, Brayden Klemens

March 2022

Techniques:

- We used a hashmap for the symbol table. The “key” attribute of this map is the variable/function name and its value is a dynamic list of all instantiations of that variable/function and the scope they’re instantiated in. Because we are using post-order depth-first-search to traverse the syntax tree, we know that the last element in each list is instantiated in the most relevant (recent) scope. When exit a block, we remove all occurrences of declared variables in the previous scope.
- We used a hashset for tracking “return” statements inside int functions. We did this to ensure we were able to capture all “return” statement scopes inside the function. This design feature can be leveraged in the future to ignore dead code and report errors when a logical path does not yield in at least one return statement.
- We use global variables to keep track of certain global states. For example, when we’re type checking an expression, we want to ensure that there does not exist a call to a void function - void functions do not return int values, and thus cannot be used in expressions. We use a global boolean variable called “is_value_required” to keep track of when we’re allowed to call void functions and when we’re not. Another example of global state is the function_tracker variable - this is used to check return types of functions.

Design Process:

- Developing from the top - down. We started the design by working on elements that are highest-scoped and slowly progressing to things that are lower-scoped. In abstract, we went from function declarations → if conditions → expressions. This allowed us to quickly brainstorm and eliminate all big-picture edge cases before moving onto obvious edge-cases such as expression type checking.
- We kept a block comment that signified a TODO list at the top of our code, and often added potential ideas (edge cases) there, prioritizing them based on importance factors. This allowed us to quickly reference potential edge cases when writing code for various functions.
- Working on github (we have a private repository), we set up a few branches to play around with different configuration ideas for traversing the tree (including different global variables, and symbol_table definitions).

Lessons Gained in the implementation process:

- One of the key lessons we gained during M2 was the importance of simplifying code before moving onto more complex parts. Something we struggled with a little was keeping track of the global scope vs indentation level (for printing the tree). Initially, we had a hacky solution that worked, however, by the end of our coding process we ran into edge cases that were difficult to resolve and the hacky solution made our code extremely difficult to fix. This caused us a few extra hours of re-programming the entire logic of our parsing functions.

Assumptions:

- input() and output() functions may not be redefined

- We do not support function overloading (this is a feature in Java, but not in C)
- void variables are caught during stage 1 of compilation and not stage 2
- We print **INT[]** if the variable is an int array to make it extremely clear

Limitations:

- Our analyzer does not handle dead code after return statements, nor inside statements such as if, else, and while blocks.
- We report a warning if a return statement is not in the highest-level function scope, because we do not have code that checks all conditions to see if a return statement is guaranteed.
- We do not handle out of bounds indexes when using array variables.

Improvements:

- Adding support for more complex semantic errors such as:
    - Handling dead code -> For example, code that is placed after a return statement, code that cant be reached inside of if or while statements.
- Checking if an array index is out of bounds.
- Accessing elements in an array that do not have values stored.

Semantic Errors Handled:

- Redefining variables / array variables at the same scope
- Redefining a function a the top level scope
- Int return type functions that do not return a value
- Void function attempts to return a value
- Variables not being declared before being used
- Array variables not being declared before being used

- Functions not being declared before being called
- Calling void functions as part of integer expressions (ex: k = foo(); foo is void)
- Assigning functions to improper types (variable not declared or check type)
- Function arguments mismatches (‘array expected’ or ‘int expected’)
- Too many function arguments passed to a function call
- Insufficient function arguments passed to a function call

Contributions:

- As a team we worked over discord to tackle all of the problems in Milestone 2. VsCode live share was a very useful tool in our pair programming process, which allowed us to update the code base in real time. This prevented a lot of confusion when pushing changes. This type of teamwork allowed us to discuss trade offs and ideas in real time and be up-to speed with the latest code.

Boris:

- Semantic Analyzer helper functions, Semantic Analyzer visit functions, creating input and output predefined functions, Support -s in main.

Brayden:

- Semantic Analyzer helper functions, Semantic Analyzer visit functions.