Compilers - Phase 2 Report

Naved Rizvi
Usman Ehtesham Gul

Symbol Table

To implement the symbol table, we followed the concepts provided in Engineering a Compiler Chapter 5.5 and did a “sheaf-of-tables” implementation. For each scope that we entered, an `initializeScope` method was called, which created a new symbol table and set the parent of the scope to the symbol table of the scope above it except for the global scope which has no parent symbol table. We also kept track of the level I am in, so that I know that I was pointing to the right parent symbol table. Whenever we exited a scope, we would get the current symbol table pointers to the parent symbol table, till we reached the symbol table of the global scope. To populate the symbol table, we created a listener that would walk through the parse tree and gather the variables, type definitions and functions declared. We used the listener methods as the listener methods are not responsible for explicitly calling methods to walk their children. As part of the tree walk, we add the standard library functions (printi, printf, not, and exit) to the global scope. We did some semantic checks as part of the symbol table generation; the remaining semantic checks were done as part of the semantic check listener. Some of the semantic checks we did as part of the symbol table generation included redefining a type, not defining variables in the global scope as static, not defining the variables in the let scope as var, defining an array without a user-defined type, referring to a variable that was not declared, and declaring functions and variables that were already defined in the current scope. We collect errors through the tree walk and at the end of the walk we check if errors were collected, then we print the errors found and exit with code 4 and not generate a symbol table. Format of the error message is defined in the “Semantic Checking” section. If no errors were found, then we have a symbol table generated which is then passed to the next tree walk that handles semantic checking.

In hindsight, one thing I would have done differently is that for variables who types were defined types, I would store the base type for those variables and that would have made it easy to retrieve the exact correct type during IR generation rather than always looking up the base type for a given variable whose type is user-defined.

Semantic Checking

Once the symbol table is generated, we do another tree walk to do semantic checks; we do not repeat the semantic checks done in the symbol table generation. For the semantic checking, we used the listener method. We followed the semantic rules defined in the Tiger specification and collected errors when we found invalid semantics. For a given statement, we would collect the

first error and return rather than collect multiple errors for a given statement. But over multiple statements, we would collect multiple errors. If at the end of the tree walk, we see that we have collected errors, we print the errors found and exit with code 4. The format of the error message is:
“line <line_number>:<column_number> <description>”
We used the listener methods as the listener methods are not responsible for explicitly calling methods to walk their children. This made it easier to focus on the semantic check logic without worrying about having to visit the right children nodes. We took advantage of the ParseTreeProperty class to annotate nodes so that it is easier to return values to the parent node after visiting the children nodes. For the semantic checks, we collected the return types of the expressions so that we can do correct type checks, type casting and type equivalence. We also used stacks to keep track of any control-flow context that we were in (if, if-else, while and for). This made it easier to check for a break statement outside of for or while loops.

IR Generation

If the semantic checks step was successful, then we create a new listener method to do a third tree walk and generate IR according to the IR specification. We used the listener methods as the listener methods are not responsible for explicitly calling methods to walk their children. This made it easier to focus on the IR generation logic without worrying about having to visit the right children nodes. To the listener method, we pass the symbol table generated in the first tree walk. Since IR generation deals a lot with temporary variables and name mangling of the variables, we took advantage of the ParseTreeProperty class again; this time we annotated notes with the variables name returned by the children nodes. These variables could be a mangled name of a variable or a temporary variable; the current node does not care as it relies on its descendants to do the right thing and return the correct variable generated if applicable. As we visit each node, we would collect IR statements that map to a given statement. For some nodes, we would emit IR on entering a node, for some on exiting a node and for some during both. For some IR generation output, we first created place holders as we needed to collect all variables generated including the temporary variables and then populate the placeholders with all the variables under a given function. The nested control flow for if-else was challenging because it was really difficult to tell when we were within an if context or an else contest. For the for loop, we implemented the IR following the IR format of the while loop.

In hindsight, I would have gone with the visitor pattern for the IR generation since that would have given us more control for each of the nodes visited and we would have more easily been able to tell whether we were in an if context or an else context for an if-else, for example.

Bugs/Features Not Fully Operational

We are currently failing one test: IR - demo_priority_queue.