Lab 3 Written Report

Written by:

Eric Elmoznino

Abhinav Rajaseshan

Pranit Wadkar

Documentation

All lexical analysis code resides in the scanner.l file. No additional code has been added to the starter version, other than parsing the ‘,’ token, which was not there beforehand.

Syntax parsing code all resides in parser.y. We used bison to define the grammar and parsing rules, and no additional c helper functions. The sections are divided as follows:

- token declarations
- precedence and associativity declarations
- grammar rules

When bison reduces a grammar rule, we print out the matching rule and initialize a node in the AST.

The AST structure definition and functions reside in ast.h and ast.c. Semantic analysis code resides in semantic.h and semantic.c, and the symbol table used for semantic analysis is defined in symbols.h and symbols.c

Breakdown of Work

Work was broken down as evenly as possible with the following responsibilities:

- Eric: Wrote the structure of the AST, its visitor functions, and the node allocation code in the parser.
- Abhinav: Worked on semantic analysis and created the symbol table used to assist during semantic analysis.
- Pranit: Worked on semantic analysis and wrote test cases.
- All: Everyone contributed to debugging and the writing of this document.

Overall Approach

The AST structure was defined recursively to mimic the grammar. We adopted the suggested structure for a C program and every node in the tree contained a discriminated union of the different types of language constructs they represented (e.g. statement, binary expression, etc.). In our visitor functions (print and free), we walked through the tree and recursively executed different code blocks for different node kinds using case statements. For printing, we walked through the tree in pre-order, whereas for printing we walked through the tree in post-order.

Semantic checking was also implemented recursively. The approach was to walk through the tree in pre-order and check all relevant errors based on the node kind using case statements. The code within the case statements would check for possible errors (e.g. illegal type conversion in an assignment statement), and would also check for errors in its children using recursive calls to the semantic check function.

The AST Structure

As mentioned above, the AST structure mimicked the recursive structure of the grammar and visitor functions executed different code on nodes based on their kind. One interesting optimization was that certain data members were not assigned during syntax parsing, but during semantic instead. This was simply out of convenience.

For instance, when printing an assignment node, we had to print the type of the variable being assigned to. Because this is not known at parsing time, we could not directly initialize the assignment node with the type. Instead of trying to traverse the tree and dynamically figure out its type when printing, we simply assigned the type during semantic traversal because we had access to the symbol table with all variable types.

Approach to Semantic Analysis

Our approach to semantic analysis was to recursively walk through the tree in pre-order and execute different error checks depending on the node kind. An interesting strategy that we had was to return an error integer code (-1) when something wrong happened, and a success code or type code otherwise. The advantage of this was that checking for errors caused by a node's dependencies on sub-nodes was very easy.

For instance, consider a declaration with initialization of the form “int a = b+c;”. In this case, we would need to make sure that the binary expression on the right hand side resolves to an int. Since the semantic check function returns an integer, the declaration would only have to call the semantic check function on its right hand side and then make sure that a) an error is not returned, and b) the integer value returned corresponds to the int type code.

Symbols Table

The way we implemented the symbol table is a linked list of a linked list. Each symbol table entry represents a variable. The child points to the first variable of next scope and the sibling points to the next variable in the same scope. Hence when a scope is entered, a pointer is made to the first element and every element in the scope is added as a sibling. This is a succinct way of defining symbol table with only two pointers. When the scope is exits, we free all the siblings.

We faced difficulty searching if the variable is defined in the parent scope. We solved most of this by freeing the scopes when the compiler left the scope.

The scope does not need to be tracked internally since if a scope is done, it can be freed. Any other variable still referenced in the table has already been declared and if it is a sibling in the current pointer of this stack, it has already been defined in this same scope.

Challenges Faced

Implementing all of the rules for predefined variables was time consuming, but the most difficult rule was that write-only variables could not be written to in an if statement block. At first, we tried to traverse the tree upwards in some way to solve this issue, but eventually we thought of a much simpler solution. We simply kept a global int called “isInIf” and initialized it to 0. Every time we entered an if statement, we would increment the variable. Every time we exited, we would decrement it. This way, we knew that we could only write to one of the predefined write-only variables when “isInIf” was equal to 0.

Another challenge was printing the AST nodes that required a type to be output, but the type was not included in the parsing rule. In other words, the type would have to be inferred based on the variable being assigned to or used. As detailed in the AST section above, we solved this

problem by assigning certain member variables of the AST nodes during semantic analysis, when we had the symbols table handy.

Vectors were also tricky because we had to convert their types when indexing into them. If we had something like “int a = b[2]”, we could not simply check that the type of b was int, because this would not be the case. Instead of trying to do arithmetic to get to the base type of the vector, we opted for a simpler solution. When a vector or variable was declared, we inserted not only its actual type into the symbol table, but also its base type. This way when we needed to see if “b[2]” was an int, for instance, we would just have to check its base type in the symbol table and not worry about if it was a vector or not.

A challenge when writing the AST was that we could not really interpret it based on what was being printed because it was very difficult to see the structure. The way we solved this was with an indentation scheme when printing. Every sub-expression was printed on the next line with an extra indent, which we carried and incremented recursively when calling our recursive print function.

Finally, we attempted the bonus where we had to make sure a variable had been initialized before being read. Instead of checking every possible control flow path to find the assignment of a given variable, we simply kept another data member in our symbols table called “is_init”, which indicated if the variable had been initialized. Whenever a declaration with assignment or just an assignment was made to a variable, we set its “is_init” flag to true. This way, whenever we had to read from the variable, we would only have to check its flag in the symbols table to see if it was legal.

Bonuses Attempted

- Ensure that every variable has been assigned a value before being read. This is challenging because it requires checking potentially all control flow paths that lead to a read of a variable.
- Report the line number on which the semantic error occurred.