

AST and Scope/Type Checking for Semantic Analysis

Tyler DeLorey
tyler.delorey1@marist.edu

April 14th, 2025

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 1.1 | How I implemented Semantic Analysis | 2 |
| 1.2 | Test Files for Semantic Analysis | 6 |
| 2 | Special Comments for Semantic Analysis | 7 |
| 2.1 | AST Special Cases | 7 |
| 2.2 | Scope/Type Checking Special Cases | 7 |
| 3 | Test Cases | 8 |
| 3.1 | semanticCompMismatch.txt | 8 |
| 3.2 | test1.txt | 9 |
| 3.3 | semanticDuplicate.txt | 10 |
| 3.4 | ASTgen2.txt | 11 |
| 4 | Conclusion | 12 |

1 Introduction

1.1 How I implemented Semantic Analysis

Before showing the test cases for Semantic Analysis for this grammar, I want to briefly explain how I created the Analyzer in the first place. Below shows the creation of the SemanticAnalyzer class.

```
96      // SEMANTIC ANALYSIS
97      SemanticAnalyzer currentAnalyzer = SemanticAnalyzer(i + 1,
currentCST);
98      currentAnalyzer.generate();
99      currentAnalyzer.traverseSymbolTable();
100
101      cout << endl;
102
103      currentAnalyzer.printAST();
104      errors = currentAnalyzer.getErrors();
105
106      if (errors > 0)
107      {
108          log("INFO", "Symbol Table for Program #" + to_string(i
+ 1) + " skipped due to Semantic Analysis error(s)");
109          currentParse.deleteCST();
110          currentAnalyzer.deleteAST();
111          currentAnalyzer.deleteSymbolTable();
112          continue;
113      }
114      currentAnalyzer.printSymbolTable();
```

Figure 1.1: Creating Semantic Analyzer Class (main.cpp)

After parsing and displaying the CST, the program goes straight into Semantic Analysis. If there were any errors during parse, that means it won't continue to this step and it will skip to the next program if there is any. If there were no parse errors, the compiler will continue to the Semantic Analysis step. After the class is created with the correct parameters (Line 97), the generate() class function is called, which generates the AST and also does scope/type checking for each branch of the AST once the branch is complete (Line 98). Once it is completed, there is a call to the class function traverseSymbolTable(), which traverses the symbol table (obviously) to check for extra warnings, like declared but never initialized variables, and initialized variables that were never used (Line 99). After this, the AST is printed to the terminal (Line 103). If there were any errors with the scope/type checking, the symbol table will not be printed and the program will not continue to Code Gen (Line 106-113). If this happens, all existing trees and hash tables for that program would be deleted to avoid any memory leaks. If there were no errors in Semantic Analysis, the symbol table will be printed (Line 114).

My approach for building the AST was to take the existing CST and only take the important branches and leaves. Generating the AST is done in the `inorder()` class function for Semantic Analysis. I'll only show a snippet of the code that takes place in the middle of the function since there is a lot, but I'll try to explain what the entire function is doing to the best of my ability.

```

218         else if (name == "Statement")
219         {
220             // Type of Statement
221             inorder(node->getChild(0));
222         }
223         else if (name == "Print Statement")
224         {
225             // Add print Node
226             myAST->addNode("branch", "Print");
227
228             // Expr
229             inorder(node->getChild(2));
230
231             // Scope/type checking for print statements
232             checkPrint();
233
234             // Move up AST
235             myAST->moveUp();
236         }
237         else if (name == "Assignment Statement")
238         {
239             // Add assignment Node
240             myAST->addNode("branch", "Assign");
241
242             // ID
243             inorder(node->getChild(0));
244
245             // EXPR
246             inorder(node->getChild(2));
247
248             // Scope/type checking for assignment
249             statements
250             checkAssignment();
251
252             // Move up AST
253             myAST->moveUp();
254         }

```

Figure 1.2: Generating the AST (SemanticAnalyzer.h)

The function is a recursive function that does a inorder traversal of the CST to generate the AST. It has a lot of similarities to the Recursive Decent Parser built for the Parsing step, except instead of looking at all of the Tokens or Nodes, this function only looks for the important ones. The one parameter for this recursive function is the current Node, which is either a branch or a leaf of the CST. The first node that is called is the Program branch, which then

leads to the Block node, which leads to "{ StatementList }" according to the grammar and the CST (since parsing was all correct, it will definitely lead to what was described). The program creates a new scope in the symbol table tree for this new Block and it will continue with the StatementList (since it is important), but that final closed brace "}" isn't important, so it will not be recursively called. Note that both StatementList and the CharList have epsilon productions, so I needed to be careful to deal with those since I didn't want to call the function with any nullptrs (The Block branch could've lead to {} if the StatementList was an epsilon production). In the above code snippet, when it reaches Statement, it will call the function for that type of statement (Lines 218-222). Then, depending on the type of statement, it will create an AST branch for it, call any existing children of that CST branch, and once all of the important children are called, scope/type checking will occur for the branch. An example of any non-important children would be like the "print", "(", and ")" in the Print Statement. Notice how only the expression inside the parenthesis is called, which will always be the second index of the children in the Print Statement branch of the CST (Line 229). Note that if a leaf node is called in the inorder function, it must be important, so it gets added regardless.

Before I came up with this approach for the AST generation, I had another approach which also involved using the CST to create the AST. I also did an inorder traversal of the CST, but instead of only traversing the important nodes, I would traverse every single Node, and decide which ones were important from there. This kind of worked, until I got into parts of the tree that were structurally different than how they were supposed to look in the AST, most notably the ADD statement and boolean expressions. Since these have different structures in the CST vs in the AST, finding the leafs to ADD and the boolean expressions, and reordering the tree, were extremely difficult. Nested boolean expressions in if/while statements absolutely broke my AST and my program. This may sound confusing since the approaches are very similar, but a majority of my commits on GitHub for this section dealt with my other approach (which is why there are so many commits, my bad), but I decided to toss it and rewrite the inorder function. The way the recursion works when only calling the important nodes makes it simpler to add children of the ADD and boolean expression branches.

Thankfully, I didn't have to change my approach much for actually scope/type checking the AST, since it mostly works well with both of my implementations for generating the AST. After any statement is completely generated in the AST, a function is called that will scope and type check to see if there are any Semantic errors or warnings (see Lines 232 and 249 in Figure 1.2). For this document, I will walk through the checkPrint() function, which is the scope/type checker for the print statement. Most other "check" functions work like this one. Below is the checkPrint function.

```

473     // SCOPE/TYPE CHECKING FOR PRINT STATEMENTS
474     void checkPrint()
475     {
476         // Get information about the current HashNode
477         HashNode* curHashNode = mySym->getCurrentHashNode();
478
479         // Get information about the current branch of AST for
scope/type checking
480         Node* currentBranch = myAST->getCurrentBranch();
481         Node* child = currentBranch->getChild(0);
482         string childName = child->getName();
483         Token* linkedToken = child->getToken();
484
485         // Checks to see if printing a variable
486         if (linkedToken->getType() == "ID")
487         {
488             // DEBUG log
489             log("DEBUG", "Print Statement: SCOPE checking for
identifier '" + childName + "'");
490
491             // Find this variable in the symbol table
492             HashNode* correctNode = findInSymbolTable(
curHashNode, childName);
493
494             // If variable exists, set it to used and link
token
495             if (correctNode)
496             {
497                 // Set it to used
498                 correctNode->setUsed(childName);
499
500                 // Throws warning if it wasn't initialized
501                 if (!correctNode->checkInitialized(childName))
502                 {
503                     log("WARNING", correctNode->getType(
childName) + " [" + childName + "] is used at (" + to_string(
linkedToken->getLine()) + ":" + to_string(linkedToken->
getColumn()) + "), but wasn't initialized");
504                     warningCount++;
505                 }
506             }
507             // If it was not found, throw 'use of undeclared
variable' error
508             else
509             {
510                 log("ERROR", "Use of undeclared variable '" +
childName + "' at (" + to_string(linkedToken->getLine()) + ":"
+ to_string(linkedToken->getColumn()) + ")");
511                 errorCount++;
512             }
513         }
514     }

```

Figure 1.3: checkPrint (SemanticAnalyzer.h)

The function, and all other "check" functions, begin with getting information

about the current branch of the AST and its children, as well as the current symbol table in the symbol table tree, which is determined by the scope that the program is currently in depending on Block branches (Lines 477-483). Next, the program determines if it is printing an identifier (Line 486). If it is printing any form of literal, there is no error/warning checking that needs to occur, as it will always be valid. If it is a variable, scope checking occurs (there is no type checking for print statements). It tries to find that variable in the symbol table tree (Line 492). If it doesn't exist, an undeclared variable error is thrown (Line 510). If it does, the variable is set to used in the correct symbol table (Line 498), and if it wasn't initialized a warning is thrown (Lines 501-505).

Even though there is no type checking in print statements, I feel the need to explain it a little bit anyway. I create my own function named `getType()`, which, well you guessed it, gets the type of whatever Node that is passed in. This is useful for assignment statements, boolean expressions, and addition statements, since type matters. If the types aren't equal, an error is usually thrown, since in this grammar, things like assignment statements won't work if types are different.

1.2 Test Files for Semantic Analysis

The repository has a ton of test cases in the `testFiles` folder, mostly from Lex and Parse, but there are also some test files specifically for testing Semantic Analysis. More tests will be added for code generation, but for now, the `testFiles` folder contains a lot of tests dealing with errors and warnings for Semantic Analysis. Section 3 will explore some of these test cases specifically.

2 Special Comments for Semantic Analysis

2.1 AST Special Cases

Dealing with the addition statements and boolean expressions are a lot easier with my implementation of generating the AST. However, one special cases for generating the AST is combining the CharLists to create a single string to add as a leaf to the AST. On the next page is how String Expr is dealt with with generating the AST in my inorder function.

```
330         // String Expr is a bit different
331         // Collect all chars from the CharList to create
the entire string
332         else if (name == "String Expr")
333         {
334             // Gets the entire string from the child
CharLists
335             string result = "";
336             Token* currentToken = nullptr;
337             collectCharNodes(node->getChild(1),
currentToken, result);
338
339             // Add the string as a leaf node
340             myAST->addNode("leaf", result);
341             myAST->getMostRecentNode()->linkToken(
currentToken);
342         }
```

Figure 2.1: StringExpr inorder function (SemanticAnalyzer.h)

A result string and a currentToken Token pointer is initialized (Lines 335-336) and a function called collectCharNodes() is called (Line 337), which will collect each character in the CharList branch, which will update the result string and currentToken pointer since they are passed by reference. Then, the result leaf is added as a node to the AST and a token is linked (Lines 340-341). The token that is linked will be token of the first character in the string, so if there were any error/warnings involving that string, it will display the line and column of that first character.

2.2 Scope/Type Checking Special Cases

There aren't many special cases dealing with scope and type checking. The only "special" things that occur in this step is when scope/type checking addition statements and boolean expressions. I'm not going to dive too deep into this, since it is mostly self-explanatory, but there are a couple of rules I kept in mind when implementing the checkADD and checkBool functions. For addition, only two integers can be added. If the variable is of type int (type is stored in the symbol table), or if its a literal digit (0-9), it is good. Anything else, like strings or booleans, cannot be added together. The checkADD function makes sure to

implement this, and it gives errors/warnings if something is off. For boolean expressions, if the types are equal, it is valid. If a string is being compared to a string, it's valid. An integer compared to an integer, it is valid. It doesn't matter if its a variable or a literal. These aren't necessarily special, but I had to create special functions for these two implementations, even if they are apart of an assignment statement or another type of statement.

3 Test Cases

3.1 semanticCompMismatch.txt

The first test case I will be going over is semanticCompMismatch.txt. This program does a lot of comparisons between two different types, resulting in type mismatch errors.

```
1 /* Types do not match in Boolean comparison*/
2 {
3   if(4 == false){
4     print("this no good")
5   }
6   if(4 == "hey"){
7     print("int to string")
8   }
9   if(false != "hey"){
10    print("bool to string")
11  }
12  if(4 != 3){
13    print("int to int")
14  }
15 }$
```

Figure 3.1: semanticCompMismatch.txt

Below is a portion of the Semantic Analysis output for this program.

```
INFO Analyzer - Semantic Analysis for Program #1
DEBUG Analyzer - Boolean Expression: TYPE checking for '4'
DEBUG Analyzer - Boolean Expression: TYPE checking for 'false'
ERROR Analyzer - Type mismatch: Comparing int to boolean in boolean expression at (3:4)
DEBUG Analyzer - Boolean Expression: TYPE checking for '4'
DEBUG Analyzer - Boolean Expression: TYPE checking for 'hey'
ERROR Analyzer - Type mismatch: Comparing int to string in boolean expression at (6:4)
DEBUG Analyzer - Boolean Expression: TYPE checking for 'false'
DEBUG Analyzer - Boolean Expression: TYPE checking for 'hey'
ERROR Analyzer - Type mismatch: Comparing boolean to string in boolean expression at (9:4)
DEBUG Analyzer - Boolean Expression: TYPE checking for '4'
DEBUG Analyzer - Boolean Expression: TYPE checking for '3'
DEBUG Analyzer - Finding extra warnings...
DEBUG Analyzer - None found.
INFO Analyzer - Semantic Analysis completed with 3 error(s) and 0 warning(s)
```

Figure 3.2: semanticCompMismatch Output

There is not much to explain here. Since there are semantic errors (two different variable types are being compared), it throws a type mismatch error. One thing that I think is important to note is even though it isn't shown in the above image, the AST will still fully print with these errors. The only thing that isn't printed is the symbol table.

3.2 test1.txt

The second test case I will be going over is test1.txt. This is just a normal program that should not and does not produce any errors.

```
1 {  
2     /* THIS IS SAMPLE CODE */  
3     int a  
4     a = 0 + 1  
5     boolean b  
6     b = true  
7     string c  
8     c = "hello world"  
9     print(c)  
10 }
```

Figure 3.3: test1.txt

Here I will show the output for the symbol table, since I haven't shown that before

| | | | | | | | | |
|------|--|------|---------|---------|---------|-------|------|--------|
| INFO | Analyzer - Symbol Table for Program #1 | | | | | | | |
| INFO | Analyzer - | NAME | TYPE | isINIT? | isUSED? | SCOPE | LINE | COLUMN |
| INFO | Analyzer - | c | string | true | true | 0 | 7 | 5 |
| INFO | Analyzer - | b | boolean | true | false | 0 | 5 | 5 |
| INFO | Analyzer - | a | int | true | false | 0 | 3 | 5 |

Figure 3.4: test1 Symbol Table

The output shows the name of every variable that was ever declared in the program. Since there are no extra blocks, they were all declared in Scope 0. It also shows its type, if it was initialized, if it was used, and where it was declared in the program (line, column). There is something else I want to explain about how Scope is displayed, but I'll save that for the final test case where it is actually shown.

3.3 semanticDuplicate.txt

The third test case is semanticDuplicate.txt. This is a test case that shows what happens when a variable is redeclared in the same scope.

```
1  /* Variables being declared again in same scope */
2  {
3  int a
4  {
5  string a
6  a = "this is fine"
7  }
8  boolean a /* this is not fine */
9  }$
```

Figure 3.5: semanticDuplicate.txt

Below is the Semantic Analysis output for this program.

```
INFO Analyzer - Semantic Analysis for Program #1
DEBUG Analyzer - Variable Declaration: Creating identifier 'a' at Scope 0
DEBUG Analyzer - Variable Declaration: Creating identifier 'a' at Scope 1a
DEBUG Analyzer - Assignment Statement: SCOPE checking for identifier 'a'
DEBUG Analyzer - Assignment Statement: TYPE checking for 'a'
DEBUG Analyzer - Assignment Statement: TYPE checking for 'this is fine'
DEBUG Analyzer - Variable Declaration: Creating identifier 'a' at Scope 0
ERROR Analyzer - Redeclared variable [a] at (8:1)
DEBUG Analyzer - Finding extra warnings...
WARNING Analyzer - int [a] is declared at (3:1), but never initialized
WARNING Analyzer - string [a] is declared at (5:1) and was also initialized, but never used
INFO Analyzer - Semantic Analysis completed with 1 error(s) and 2 warning(s)
```

Figure 3.6: semanticDuplicate Output

As you can see, an error is thrown saying that `a` was already declared in the same scope. Notice how it was completely fine with declaring `string a` on Line 5, since that was in another scope (1a), while `int a` (Line 3) and `boolean a` (Line 8) were declared in the same scope (0).

3.4 ASTgen2.txt

The last test case I will be going over is ASTgen2.txt. This is a test case that, again, tests that Semantic Analysis works as intended and the AST is built correctly, even with multiple scopes.

```
1 {  
2   int a  
3   {  
4     boolean b  
5     {  
6       string c  
7       {  
8         a = 5  
9         b = false  
10        c = "inta"  
11      }  
12      print(c)  
13    }  
14    print(b)  
15  }  
16  print(a)  
17 }$
```

Figure 3.7: ASTgen2.txt

I will again be showing the symbol table since the program was semantically correct (no errors).

| | | | | | | | |
|------|--|---------|---------|---------|-------|------|--------|
| INFO | Analyzer - Symbol Table for Program #1 | | | | | | |
| INFO | Analyzer - NAME | TYPE | isINIT? | isUSED? | SCOPE | LINE | COLUMN |
| INFO | Analyzer - a | int | true | true | 0 | 2 | 5 |
| INFO | Analyzer - b | boolean | true | true | 1a | 4 | 9 |
| INFO | Analyzer - c | string | true | true | 2a | 6 | 13 |

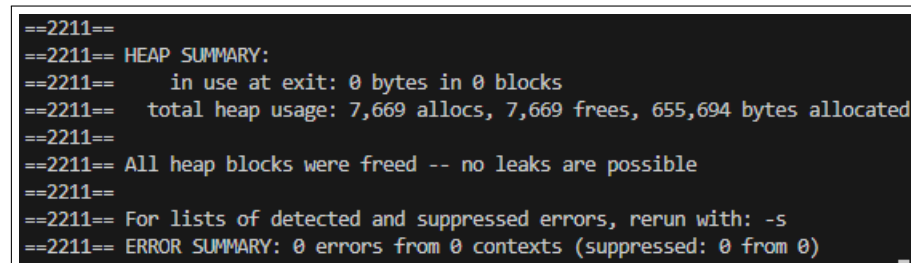
Figure 3.8: ASTgen2 Output

Again, the symbol table shows every variable that was declared in the program. However, this time, since we used different scopes, we can see the scope change between the variables. I implemented Scope as layers, where two symbol tables (scopes) in the symbol table tree that have the same parent are on the same layer, meaning they have the same number, just a different letter (1a vs 1b). Since, in this test case, the blocks go deep, they are on different layers, resulting in the number incrementing.

4 Conclusion

There are many more test cases that I have in the testFiles folder, but these are some I decided to chose to show. I think Semantic Analysis was a lot harder for me than Parsing, and it was still harder than Lex. My first implementation of Semantic Analysis when generating the AST was very rough, and I was demotivated at times to try to change it because I kept finding flaws. But, I decided to rewrite it completely and I'm glad it works now.

Some of my time was spent on finding memory errors in my program. Even though it would show that all heap blocks were freed and that there were no leaks, there still were errors in the error summary. Most of these errors involved the Token pointer being a nullptr and if there was a chance I would access the contents of that pointer, it would throw an error. However, I fixed all of the errors. Below shows some valgrind output that proves that the program has no memory leaks and all bytes are allocated correctly.

A screenshot of a terminal window displaying Valgrind's output. The text is as follows:

```
==2211==  
==2211== HEAP SUMMARY:  
==2211==    in use at exit: 0 bytes in 0 blocks  
==2211==   total heap usage: 7,669 allocs, 7,669 frees, 655,694 bytes allocated  
==2211==  
==2211== All heap blocks were freed -- no leaks are possible  
==2211==  
==2211== For lists of detected and suppressed errors, rerun with: -s  
==2211== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 4.1: Valgrind Output