

Full Compiler with 6502a Code Generation

Tyler DeLorey
tyler.delorey1@marist.edu

May 5th, 2025

Contents

1	Introduction	2
1.1	How I implemented Code Generation	2
1.2	Test Files for Code Generation	4
2	Special Comments for Code Generation	5
2.1	Starting Code with a While Loop	5
2.2	Boolean Hell	6
3	Test Cases	7
3.1	CodeGenNestedIfs1.txt	7
3.2	CodeGenWhileStatement.txt	8
3.3	CodeGenComplexVars2.txt	9
3.4	CodeGenHeapOverflow.txt	10
4	Conclusion	11

1 Introduction

1.1 How I implemented Code Generation

Before showing the test cases for Code Generator for this grammar, I want to briefly explain how I created the Code Generator in the first place. Below shows the creation of the CodeGen class.

```
124         // CODE GEN
125         CodeGen currentCodeGen = CodeGen(i + 1, currentAST,
currentSymbolTable);
126         currentCodeGen.generate();
127         currentCodeGen.print();
128
129         // PROPER MEMORY MANAGEMENT
130         currentParse.deleteCST();
131         currentAnalyzer.deleteAST();
132         currentAnalyzer.deleteSymbolTable();
```

Figure 1.1: Creating CodeGen Class (main.cpp)

After doing type/scope checking on the AST, the program goes straight into Code Generation. If there were any errors during Semantic Analysis, 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 semantic errors, the compiler will continue to CodeGen. The class is created with using the AST and the Symbol Table from Semantic Analysis (Line 125). These are two important data structure that need to be used for CodeGen. Next, the generate() class function is called, which completes code generation based on the 6502 Instruction Set (Line 126). Once it is completed, there is a call to the class function print(), which prints the result of the Code Generation if there weren't any errors (Line 127). For CodeGen, the only error that there could be is overflow, where there is too much code that it can't all fit in the available address space, which is only 256 bytes in this compiler.

I had to create a runtime environment to store all of the generated hexadecimal code using the 6502a Instruction Set. The beginning of the runtime environment, called the "code" section, has all of the opcodes and operands for the code. After this section, the "stack" section contains all of the locations of static variables. Static variables in this grammar are integers and booleans. Since this takes place right after the "code" section, when the "code" section is being generated, temporary values are used in place of static variables, and then once the "code" section is completed, the temporary locations become addresses in the "stack" section, and backpatching occurs where those temporary locations become their corresponding real addresses. The end of the runtime environment is the "heap" section, where all dynamic values are stored. For this language, this means only the strings. When strings are created, they are added in the heap, and a pointer to its first letter is saved.

To complete code generation, I decided to do a depth-first in-order traversal through the AST. Once I hit a certain branch, depending on what it was, I would call its children leaves, and depending on those values, I would add certain opcodes to the runtime environment. I'm not going to explain the nitty gritty of this entire class, but I will walk through the "Assignment Statement" code for Code Generation. Below is the section of the traverse function that deals with assignment statements.

```

182         // Assignment Statement
183         else if (name == "Assign")
184         {
185             // Get information about assignment
186             Node* locationValue = node->getChild(0);
187             Node* readValue = node->getChild(1);
188
189             // Get the temporary location for the variable (T0,
190             T1, etc)
191             string locationTemp = findVarIndex(locationValue->
192             getName());
193
194             // Assigning statement is normal if second child is
195             a leaf node
196             if (readValue->isLeaf())
197             {
198                 // Write the value to accumulator
199                 writeToRegister(readValue, "ACC");
200             }
201             // Do further traversing through the tree if there
202             are further branches (ADD/isEq/isNotEq)
203             else
204             {
205                 if (readValue->getName() == "ADD")
206                 {
207                     // Add a temporary value to the end of the
208                     Stack that holds the sum
209                     staticData.emplace_back("0", "0", name);
210                     lastStaticIndex = staticData.size() - 1;
211                     currentTempAddress = "T" + to_string(
212                     lastStaticIndex);
213                 }
214                 traverse(readValue);
215             }
216
217             // Write calculated value (ID or literal) from
218             accumulator into memory at locationTemp
219             write("8D");
220             write(locationTemp);
221             write("00");
222         }

```

Figure 1.2: CodeGen Assignment Statement (CodeGen.h)

When the traversal reaches an Assignment branch on the AST (Line 183), it will get the two children of this branch. Because the AST has no errors (since it made it to CodeGen in the first place), it will always have two direct children. The `locationValue` child is the variable that is being assigned (Line 186) and the `readValue` child is the value being used to assign that variable (Line 187). This value can either be a leaf node (ID or literal), or it could be another branch that leads to an operation like a boolean comparison or an addition. After, it gets the temporary location of the variable that is being assigned (Line 190). Temporary locations are used to backpatch the code once it is complete. After it gets its location, it checks if that second value is a leaf node (Line 193). For assignment statements, the assigned variable will always be a leaf node, so there is no need to check. If this second value is a leaf node, the value will be added to the accumulator (Line 196). The `writeToRegister()` class function will get the value of that variable at that scope if it was a variable. If it were a literal, it would be added directly to the accumulator. If it was a string, a pointer to its location in the heap will be added to the accumulator. If it wasn't a leaf node, that means it's an ADD branch or a boolean expression (Line 199). If it was an ADD, a new temporary location needs to be added that stores this addition in a location in the Stack (Lines 201-207). After a new location is added, or if it's just a boolean expression, this branch is traversed (Line 208). Now, whatever is traversed will be added to the accumulator. That means that when the recursion unwinds and ends up back here, since the stored value is already in the accumulator, it can be added to the temporary location of the variable, thus giving the read value to the location value, which is an assignment statement (Lines 212-214).

There's not much more to explain. There are most of the fundamentals of my CodeGen class. One important aspect I would like to note is that the current Symbol Table in the Symbol Table tree needs to be tracked when the blocks are changing. When a new block is encountered, the correct symbol table child needs to be tracked. When the block is complete, the pointer needs to move up the tree. This was something that took a lot of thinking to implement correctly, but it works. Another thing is that booleans in this language will be printed as 0 (false) or 1 (true). This made it easier to implement booleans since they are essentially the same as integers, except it's just 0 and 1.

1.2 Test Files for Code Generation

The repository has some test cases for Lex/Parse/SemanticAnalysis in the test-Files folder, but there are a TON of test cases for Code Generation. These test cases deal with scope changes, nested if statements, boolean hell, specific edge cases, and more. Section 3 will explore some of these test cases specifically.

2 Special Comments for Code Generation

2.1 Starting Code with a While Loop

This is a very weird situation, and I'm not sure why it happens, and I don't even know if it's my fault. For while statements, there is an unconditional branch after its block is completed at the bottom that branches up to before the condition, as shown below.

```
327         // If it was a While Statement, need to loop back
        to the beginning
328         if (name == "While")
329         {
330             // Do unconditional branch be comparing 1 to 0
331             write("A2");
332             write("01");
333
334             write("EC");
335             write("FF");
336             write("00");
337
338             // Branch
339             write("D0");
340
341             // Calculate branch amount
342             int branchAmt = 255 + posBeforeComp - this->pc;
343             write(toHex(branchAmt));
344         }
```

Figure 2.1: Unconditional Branch (CodeGen.h)

For some reason, if this branch lands on 0x00, the program fails. The operating system thinks that it lands on 0x100, when there are only 0xFF bytes, so it breaks. The branch only lands on 0x00 if the while statement is at the very beginning of the program. To fix this issue, I added "A9 00" to the beginning of the program, so that no matter what, it will never branch to 0x00.

```
23         // Starts generating code
24         void generate()
25         {
26             // Begin each program with loading accumulator with 0
27             // This fixes an issue where while loop at beginning
            will break in OS if pc jumps to 0x00
28             write("A9");
29             write("00");
30
31             // Traverse tree to generate code
32             traverse(myAST->getRoot());
```

Figure 2.2: Added Code In Beginning (CodeGen.h)

2.2 Boolean Hell

Boolean Hell isn't necessarily a special case because of the way I implemented boolean expressions in the first place. All of the recursion works out so the first half of the expression is put into the X register and gets compared to the second half of the expression. Then, depending on if was an equality or an inequality comparison, the correct value gets added to the accumulator (0 for false, 1 for true). Because of how I implemented this, it takes a lot of opcodes and operands to perform boolean expressions, especially those that are nested. However, it all works as intended, where complex expressions and test cases should work on my compiler.

Something that may be considered as a special case are string comparisons. To do string comparisons, I only compared pointer locations. To do this, when creating a string in the heap, check if it was already created, and if it was, don't create a duplicate and just get the pointer to that location. This makes it so duplicate strings are all referred to the same location in the heap. Thus, they all have the same pointers, making comparisons easy. I created a hashmap for strings in the heap to check for duplicates.

```
617 // Writes a string into the heap in the runtime environment
618 void createString(const string str)
619 {
620     // If this string doesn't already exists
621     if (existingStrings.find(str) == existingStrings.end())
622     {
623         // Update the heap pointer
624         heapVal = heapVal - str.length() - 1;
625
626         // Create temporary pointer
627         int ptr = heapVal + 1;
628
629         // Add to hash map of existing strings
630         existingStrings[str] = ptr;
631
632         // For each character in the string
633         for (char c : str)
634         {
635             // Convert to ASCII
636             int asciiVal = c;
637
638             // Write into heap
639             write(toHex(asciiVal), ptr);
640
641             ptr++;
642         }
643
644         // Write string pointer into code
645         write(toHex(heapVal + 1));
646     }
```

```

647         // If the string already exists
648     else
649     {
650         // Write its position in the runtime environment
651         write(toHex(existingStrings[str]));
652     }
653 }

```

Figure 2.3: createString (CodeGen.h)

The else statement in the createString() class function displays what happens when a string already exists. Its pointer location is written in the runtime environment in hex notation (Line 651).

3 Test Cases

3.1 CodeGenNestedIfs1.txt

The first test case I will be going over is CodeGenNestedIfs1.txt. This program tests if nested if-statements work. The output of this program should be success.

```

1
2  /* Boolean expression if statements and nesting if statements pt
3     .1. output: success*/
4  {
5      int a
6      a=5
7      if (a == 5) {
8          if (a == 1+4){
9              if (2+a==4+3){
10                 print("success")
11             }
12         }
13     }
14 }$

```

Figure 3.1: CodeGenNestedIfs1.txt

I won't actually be showing the output of code generation, instead I will be showing what the output is in the operating system to see if the opcodes and operands were all correct. Below is the output in the CyberCore operating system (should be the same in any other 6502a operating system).

```

>load
User input validated. Loading...
Done. PID: 0
>run 0
>success

```

Figure 3.2: CodeGenNestedIfs1 Output

Because all of the if statements are true, that means that the success statement will be in the output.

3.2 CodeGenWhileStatement.txt

The second test case I will be going over is CodeGenWhileStatement.txt. This is just testing if the While Statements works correctly, printing all values from 1-9 as the x variable increments each iteration.

```

1
2 /* While Statements output: 1 2 3 4 5 6 7 8 9*/
3 {
4 int x
5 while (x != 9){
6     x = 1+x
7     print(x)
8 }
9 }$

```

Figure 3.3: CodeGenWhileStatement.txt

Below is the output in the CyberCore operating system.

```

>load
User input validated. Loading...
Done. PID: 1
>run 1
>123456789

```

Figure 3.4: CodeGenWhileStatement Output

The output prints 1-9, so it works!

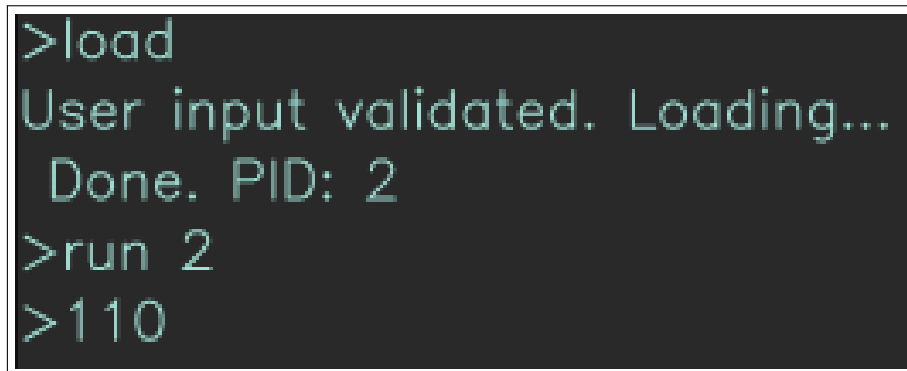
3.3 CodeGenComplexVars2.txt

The third test case is CodeGenComplexVars2.txt. This is a test case that shows how boolean expressions work when being used inside of print statements. The output should either be 0 or 1, 0 corresponding with false and 1 corresponding with true.

```
1
2 /* Variable comparisons (also in complex expressions) pt 2*/
3 {
4   boolean b
5   /*output: 1 (true) 1 (true) 0 (false) */
6   print((false == b))
7   print((true != b))
8   print((false != b))
9 }$
```

Figure 3.5: CodeGenComplexVars2.txt

Below is the output in the CyberCore operating system.



```
>load
User input validated. Loading...
Done. PID: 2
>run 2
>110
```

Figure 3.6: CodeGenComplexVars2 Output

- The first boolean expression results in true (1) since b is false.
- The second boolean expression results in true (1) since b is not true.
- The third boolean expression results in false (0) since b is equal to false, but it's checking for inequality

Since the output was 110, it worked!

3.4 CodeGenHeapOverflow.txt

The last test case I will be going over is CodeGenHeapOverflow.txt. This is one of the only errors that can occur in CodeGen (along with stack overflow and code overflow technically). This is when there are too many values in the heap, and it crashes into the stack and the code sections of the runtime environment. To do this, there is a very long string that gets created in the heap.

```
1  /* Heap overflow: This should fail code generation */
2  {
3      int a
4      a = 2
5      print("lorem ipsum dolor sit amet consectetur adipiscing elit
        sed do eiusmod tempor incididunt ut labore et dolore magna
        aliqua ut enim ad minim veniam quis nostrud exercitation
        ullamco laboris nisi ut aliquip ex ea commodo consequat dui
        aute irure dolor in reprehenderit in voluptate velit esse
        cillum dolore eu fugiat nulla pariatur excepteur sint occaecat
        cupidatat non proident sunt in culpa qui officia deserunt
        mollit anim id est laborum")
6      if (a == 2){
7          print("hi")
8      }
9      }$
```

Figure 3.7: CodeGenHeapOverflow.txt

Below is the output for the compiler.

```
INFO    Code Gen - Code Generation for Program #1
ERROR   Code Gen - Memory Overflow: Generated code exceeds available address space (256 bytes)
INFO    Code Gen - Code Generation completed with an error.
```

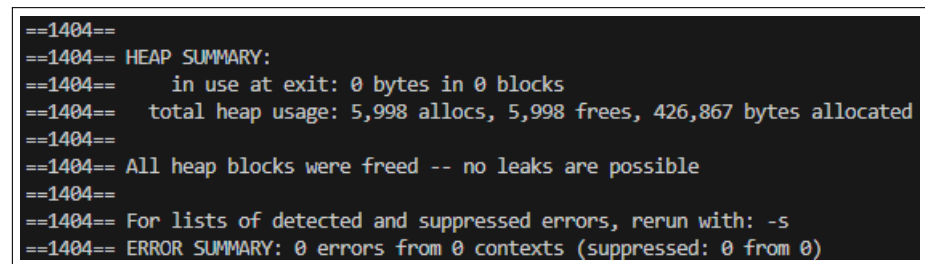
Figure 3.8: CodeGenHeapOverflow Output

Of course, no code will be displayed, as a heap overflow occurred.

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 Code Generation with harder than Lexing and Parsing, but in some ways, it was easier than Semantic Analysis. My first implementation of Semantic Analysis is probably why I think it was harder. I did it in a very questionable way, but I think that paved the way for me to make my code generation a lot better.

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 with a dark background and light-colored text. The text shows the output of a Valgrind memory check. It starts with a line indicating the output is from a file named '1404'. This is followed by a 'HEAP SUMMARY:' section. The summary states that at exit, there are 0 bytes in 0 blocks in use. It also reports a total heap usage of 5,998 allocations, 5,998 frees, and 426,867 bytes allocated. A key line states: 'All heap blocks were freed -- no leaks are possible'. Below this, it suggests running with '-s' for more details. The final line is an 'ERROR SUMMARY:' showing 0 errors from 0 contexts, with 0 suppressed.

```
==1404==  
==1404== HEAP SUMMARY:  
==1404==    in use at exit: 0 bytes in 0 blocks  
==1404==   total heap usage: 5,998 allocs, 5,998 frees, 426,867 bytes allocated  
==1404==  
==1404== All heap blocks were freed -- no leaks are possible  
==1404==  
==1404== For lists of detected and suppressed errors, rerun with: -s  
==1404== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 4.1: Valgrind Output