My C Compiler

Spring 2020 (Jan – May)

Caleb Seely

The task of building my compiler was broken into seven parts, **a scanner**, **a parser**, **an abstract syntax tree, semantic analysis**, **error handling**, **memory management**, and **code generation**. Each section builds off the previous to create functional machine code.

The Scanner

I used Flex and Bison to create a scanner for different patterns that are found in the grammar. The image below looks at some examples of pattern rules. I chose these two because they were two of the more complicated patterns I was matching.

```
171   \'.{2,}[^\n]\' return Set_Value(LINE_NUMBER, CHAR_ERR ,yytext);
172   [\'][\\].[\']  return Set_Value(LINE_NUMBER, CHARCONST_ESC, yytext);
```

Line 171 tries to identify any pattern of two or more characters between single quotes. Line 172 looks for any character following a backslash that is between single quotes. This code would later be adjusted. Not only should the rules be flipped to give precedence to the rule on 172 but I would break these down further for '\0' and '\n' because they are special to the language.

Here is a link to the code!

The Parser

This part of the project defines the grammar and lays the foundation for the abstract syntax tree.

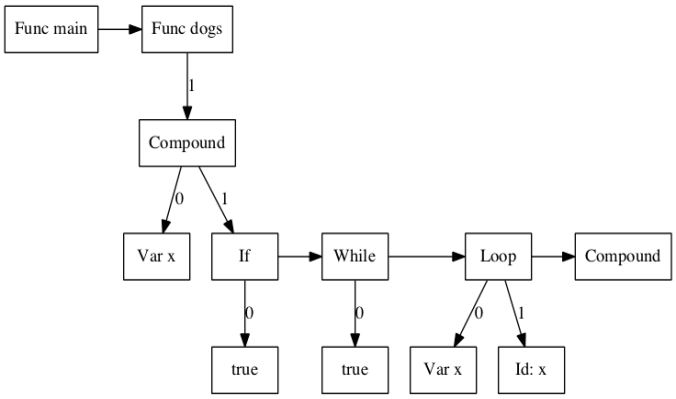
```
66   declarationList : declarationList declaration
67                   {
68                      if(cdbug) printf("<-declarationList declaration\n");
69                      $$ = addSibling($1,$2);
70                   }
```

Here is just a look at one of the first rules of the language, declarationList, which can go to itself followed by a declaration or it can just go to declaration (not pictured). These rules build off each other before leading to terminals.

The code for the grammar rules can be found here!

The Abstract Syntax Tree

To gain a better understanding of the innerworkings I added a visual tree. The diagram below represents a whole AST. I chose to only include a segment of a program's AST because they are not compact.




```
Var a of type bool [line: 3]
Sibling: 0 Var b is array of type int [line: 4]
Sibling: 1 Func max returns type int [line: 7]
!   Child: 0 Param x of type int [line: 7]
!   Sibling: 0 Param y of type int [line: 7]
!   Child: 1 Compound [line: 8]
!   !   Child: 0 Var z of type int [line: 9]
!   !   Sibling: 0 Var zz of type int [line: 9]
!   !   Sibling: 1 Var c of type char [line: 10]
!   !   Child: 1 If [line: 12]
!   !   !   Child: 0 Op: > [line: 12]
!   !   !   !   Child: 0 Id: x [line: 12]
!   !   !   !   Child: 1 Id: y [line: 12]
!   !   !   Child: 1 Assign: = [line: 12]
```

The tree can also print debugging information depending on flags. Data types and memory information can be displayed to assist in debugging.

Take a look at the recursive printing function here!

Semantic Analysis & Error Handling

The main idea here is to start protecting the user from themselves and any mistakes that could occur. Here is just one example of problems I was trying to code against.

```
543                 if( tree->child[0]->type != tree->type){
544                    printf("ERROR(%d): Variable '%s' is of ", tree->lineno, tree->attr.name);
545                    exprTypePrint(tree->type);
546                    printf(" but is being initialized with an expression of ");
547                    exprTypePrint(tree->child[0]->type);
548                    printf(".\n");
549                    numErrs++;
550                 }
```

This if statement is doing some type checking and gives a helpful error message if the types do not match. Another interesting problem presented in this section is maintaining break statements are only inside of loops.

Have a look at the bulk of my defense here!

Memory Management

This part of the project was just about accurately calculating stack frames, their location and size. Here is an example of static variables being declared.

```
588                 if(tree->isStatic){
589                    tree->memType = LocalStatic;
590                    tree->offset = Goffset--;
591                    Loffset++;  // static is technically global
```

The static variable is held in global memory space but used in the local frame. I set the memory type for debugging. I decrement the global offset to set it and increment the local offset because the default is to set it.

Code Generation

The final step of my project was to produce machine code with the appropriate commands and offsets.

```
255                 emitRM((char *)"LD", 3, tree->offset , 1, (char *)"Load var ", tree->attr.name);
```

This code loads a declared variable value into the proper offset in memory. Below is just a section of machine code I produce for a program.

```
65    * FUNCTION  main
66     42:     ST  3,-1(1) Store return address
67    * Compound
68    * Call output
69     43:     ST  1,-2(1) Store fp in ghost frame for  output
70    * Param  output
71     44:    LDC  3,987(6)   Load int const
72     45:     ST  3,-4(1) Store param
```

My machine code is not perfect, I have a problem with arrays and some smaller issues with loops and while statements also. I never anticipated getting it perfect, this project was always going to be a learning experience. There is a lot I would do differently next time, plenty of mistakes made and lessons learned all of which I am grateful happened

Checkout all the machine code generation here!