Java to C Compiler Project “CoopJa"

Group:

Nicholas Araklisianos

Miguel Cruz

Jacob Poersch

Carlos Sandoval

SPRING 2019 - May 16, 2019

COMP 430 – Language Design and Compilers

TABLE OF CONTENTS

Section 1. The Goal of the Project

Project Overview

The goal of our project is to create our own programming language based on the existing programming language "Java." Then, we will use this made-up language as input and create a Compiler that will translate the given program into valid C code.

Since we all are most familiar with the Java programming language, and our made-up language is based on Java, we decided to make Java our implementation language. After some discussion between the group members, we decided we could add some things that would be meaningful to the C programming language. We were also all interested in working with this language as well, so we made C our target language.

The name of our compiler "CoopJa" means and comes from the abbreviation: "C's Cooperative Object Oriented Programming from Java." This is because we plan bring Java's Object Oriented Programming nature to C with our compiler. Since Java supports OOP and C does not, our compiler will bridge the gap and implement this feature in C.

Definitions, Acronyms, and Abbreviations

This section outlines all definitions, acronyms, and abbreviations that may not be known to some / all of the readers of this documentation. These terms all pertain to our project specifically or relate to general terms we may use.

Term/Acronym	**Definition**
Implementation Language	**The programming language our Compiler is written in. In our case: Java**
Target Language	**The language our Compiler will compile to / its output. In our case: C**
OOP	**Object Oriented Programming**
"CoopJa"	**The name of our compiler. It stands for: C's Cooperative Object Oriented Programming from Java**

General Constraints

Due to the limited time of the semester, it is necessary to restrict CoopJa to have fewer features than the full Implementation Language. We will not be featuring any memory de-allocation, nor any garbage collection. We also will not be featuring any Generics in our language.

There are also several quirks with the language, due to Code Generation, that CoopJa contains. Having a "void" main method is not allowed. Giving an assignment to a variable declared at the top of the class is not allowed, but is allowed inside a method. Declaring an Auto type variable that the Typechecker cannot resolve the type of, that is to say it is never given an assignment, is not allowed. Auto variables are only allowed in the top of classes and methods. These restrictions can be seen in the Unit Tests.

Section 2. Language Specifications & Features (Syntax)

The following is our made-up language's syntax:

var is a variable
objectname is the name of a class
methodname is the name of a method
str is a string
i is an integer

```
type ::= int | char | boolean | string | auto |    [Built in types of variables]
         objectname                                 [Objects are also types]
op ::=   + | - | * | / |                            [Arithmetic operations]
         > | < | >= | <= | == | != | ==| |          [Comparison Operations]
         | | & | ^ | >> | << | ~                    [Bitwise Operators]
vardec ::= type var                                 [Variable declarations]

exp ::= var | str | i |                             [Basic expressions]
         exp op exp|                                [Arithmetic expression]
         this                                       [Refers to this instance]
         exp.Method(Var*)                  [Call Method]
         new objectName(exp*)                       [Declare a new instance of an object]

access ::= Public | Private | Protected              [access type for a method or var]
stmt  ::=        vardec; |                           [Variable Declarations]
                 var = exp; |                        [assignment to variable]
                 If (exp) Block_stmt else Block_stmt | [standard if/else statement]
                 while (exp) Block_stmt |            [loop statement with restriction]
                 for (vardec; exp; exp;) Block_stmt | [for loop statement]
                 break; |                                     [escape loop statement]
                 return exp;|                        [return an expression]
                 return; |                           [Empty return]
                 println(str)|                       [Prints to the terminal, string only]
                 printf(str, exp*)                            [C-Style printf statement]

Block_stmt ::= {stmt*}                               [block statement]

instancedec ::= [Access] vardec;
result_type ::= type | void                          [Return types]
methodef::=      [Access] result_type methodname (vardec*) Block_stmt [Method declarations]

objectdefheader ::= access class objectname | access class objectname extends objectname
objectdef::=     objectdefheader {
                          (vardec|methoddef)*                 [declarations]
                 }
entrypoint ::= [access] result_type main (vardec*)Block_stmt
```

```
objectdefmain :: = objectdefheader {
                        (vardec|methoddef)*              [declarations]
                        Entrypoint                       [main entry
                        (vardec|methoddef)*              [declarations]
                }

program ::= objectdefmain* | objectdef*       [Does not require entrypoint to compile]
```

Non-Trivial Features

- Computation Abstraction Non-Trivial Feature: Objects and methods with Class Based Inheritance. This means the user can define one class, define a second class that extends the first, and the second class will get all of the parent's declared variables and methods.
- Non-Trivial Feature #2: Access Modifiers (public & private) for Classes and variables. A class that is declared to be private cannot have any child classes. A public class can have child classes, but if this parent class has any private variables, its child class cannot use them.
- Non-Trivial Feature #3: Type Inference (Auto Type). A class can declare an "auto" type variable in either the top of a class or in the body of a method and the type will be resolved in the Typechecker.

Section 3. Implementation Order & Descriptions

This section details the order in which we implemented each portion of our compiler and some details about why this is the case. Each specific part of the compiler will be further discussed in its own section later in the document. Note that the technical aspects of our code and implementation are outlined in these later sections.

The first step in the creation of our compiler was working on the Tokenizer / Lexer. The purpose of the Tokenizer is to go through our input program and find recognized keywords and terms in a "reserved words" list. For example, in order to have classes in our program, we must reserve the word "class" to create a class. The reserved words list includes the types of variables (like int, string, boolean, etc.), access modifiers (like public & private), symbol names (like equals, less than, slash, etc.) and other general reserved words (like void, null, true, etc.). A complete list of our reserved words will be in the Tokenizer section of the documentation.

The Tokenizer then identifies those reserved words and replaces those words with a Token object corresponding to that word. However, this list is impossible to be exhaustive, since a user can create a variable object with any name, within the restrictions of our language. For this, when the Tokenizer finds a word that it does not recognize, it classifies this as an "Identifier," or the name of a variable, class, or method. The Tokenizer translates the entire input program into Tokens. This way, our compiler will be able to read and manipulate the given program to create meaningful output.

The next step in the process of writing our compiler was to write the Parser. The Parser takes the output from the Tokenizer and is able to decipher what is being asked from the original program and whether it is valid or not. For example, when declaring an “if” statement, the “if” keyword must immediately be followed by a left parenthesis, then an expression that resolves to a Boolean, than a right parenthesis and braces (Java & our language specific). If someone did not have the left parenthesis, the statement declaration would not be valid. The Parser uses the Tokens to determine if all that needs to be there in a declaration is in fact there. So the Parser will see an “if” token, and if it does not see a left parenthesis token immediately, it will call the statement invalid. However, the Parser is not able to tell if the statement within the parentheses actually resolves to a Boolean or not. This is because the Parser only cares about syntactic validity, and not content. The Parser is able to detect all that is defined within our language and checks the entire given program. When the Parser detects a proper-form “if” statement (for example), it puts this “if” statement into an object for ease of access later down the line. The Parser is able to define things such as variable declarations, method declarations, class declarations, and general statements. If the Parser passes, we can assume the program is syntactically valid.

After the Parser has checked the input program, the Typechecker will begin its check. The Typechecker’s job is to check the things the Parser does not. The Typechecker checked declared variables and makes sure they are not declared again or used incorrectly. For example, if the user declares an int variable but then tries to assign a string to it, this action would be caught as invalid by the Typechecker. The Typechecker would also be able to resolve the expression within an “if” statement (as mentioned previously), since it is keeping track of the names of all variables. This is to say the Typechecker is able to check the logic of the code. If there is no error in the Typechecker, the input program is valid.

The last step in the process of our compiler is the Code Generation. For this portion, we add some code to each object that the Parser creates. This code will be used to transform the given statement into C code. Code Generation is the process of checking each object, since we have determined that they are valid syntactically and logically, and outputting its C code. Once the entire program has been transferred to C code, our compiler can also output this program to a .c file and run it through a C compiler and check the output. This is used in testing to see if the expected output matches the actual output of the program we inputted.

Retrospective

Looking back on our project, our group is very pleased with the results. It is amazing to see the project come together, from learning what exactly a compiler does, and what each component entails, to actually building those components. It was a very interesting process through which we learned a lot.

If we were to do things differently, we might want to make an easier to parse language. We initially started writing the parser from scratch, but it was taking a lot of time. We found a library called "funcj.parser" for Java that made things both easier and, in some cases, more difficult. It was very easy to use but difficult to change. We also might use more libraries to help us create a more powerful compiler with more features built in.

Section 4. Tokenizer

Files Pertaining to the Tokenizer

- Token.java – The whole Tokenizer file is stored here, including a list of reserved words the Tokenizer uses.

- test\TokenizerUnitTests.java – The Tokenizer Unit Tests. Note "test\" refers to the fact that this file is not in the normal directory of .java files, and is in the test directory since it contains unit tests.

Other Files

- TokenizerExampleTest.java – This file explains the usage of the Tokenizer and gives some examples of it being used. This file could also be considered the Main() method for the Token file, since Token.java does not have one.

Reserved Words / Symbols

Our Tokenizer checks to see if the input program, word by word, is one of the listed reserved words. Complete list of reserved words here:

Keyword	Token Name
void	KEYWORD_VOID
int	KEYWORD_INT
double	KEYWORD_DOUBLE
char	KEYWORD_CHAR
boolean	KEYWORD_BOOLEAN
String	KEYWORD_STRING
auto	KEYWORD_AUTO
if	KEYWORD_IF
extends	KEYWORD_EXTENDS
this	KEYWORD_THIS
static	KEYWORD_STATIC
class	KEYWORD_CLASS
true	KEYWORD_TRUE
public	KEYWORD_PUBLIC
private	KEYWORD_PRIVATE
protected	KEYWORD_PROTECTED
break	KEYWORD_BREAK
return	KEYWORD_RETURN
while	KEYWORD_WHILE
for	KEYWORD_FOR
false	KEYWORD_FALSE
null	KEYWORD_NULL
println	KEYWORD_PRINTLN
else	KEYWORD_ELSE
new	KEYWORD_NEW

Symbol	Token Name
+	SYMBOL_PLUS
-	SYMBOL_MINUS
*	SYMBOL_ASTERISK
/	SYMBOL_SLASH
\	SYMBOL_BACKSLASH
>	SYMBOL_GREATERTHAN
<	SYMBOL_LESSTHAN
!	SYMBOL_EXCLAMATION
=	SYMBOL_EQUALS
|	SYMBOL_BAR
&	SYMBOL_AMPERSAND
^	SYMBOL_CARET
~	SYMBOL_TILDE
"	SYMBOL_QUOTE
;	SYMBOL_SEMICOLON
(	SYMBOL_LEFTPAREN
)	SYMBOL_RIGHTPAREN

Symbol	Token Name
{	SYMBOL_LEFTCURLY
}	SYMBOL_RIGHTCURLY
[	SYMBOL_LEFTBRACKET
]	SYMBOL_RIGHTBRACKET
,	SYMBOL_COMMA
.	SYMBOL_PERIOD
>=	SYMBOL_GREATERTHANEQUAL
<=	SYMBOL_LESSTHANEQUAL
++	SYMBOL_DOUBLEEQUALS
!=	SYMBOL_NOTEQUAL
||	SYMBOL_DOUBLEBAR
&&	SYMBOL_DOUBLEAMPERSAND
>>	SYMBOL_SHIFTRIGHT
<<	SYMBOL_SHIFTLEFT
++	SYMBOL_DOUBLEPLUS
--	SYMBOL_DOUBLEMINUS

Variable Naming Restrictions

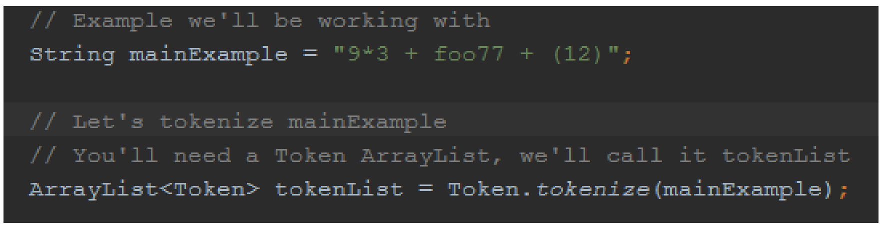
The Tokenizer is able to determine which inputted words are a variable / method / class name if it does not fall into one of the reserved words list. If the word does not appear in the list, it must be a user-defined variable name, also called an Identifier. After the Tokenizer checks the reserved words list, we use Regular Expressions to determine if this is the case. Restricting certain symbols from being part of the Identifier, and requiring the name begin with a letter or an underscore, but may contain numbers or more underscore characters after, our Regular Expression Pattern to determine this is:

```
Pattern p = Pattern.compile("\\\"(\\\\.|[^\"\\\\])*\\\"|(>=|<=|==|!=|\\|\\||&&|>>|<<|\\+\\+|--)|[a-zA-Z_]+[a-zA-Z0-9_]*|[0-9]+|\\S");
```

Tokenizer Output

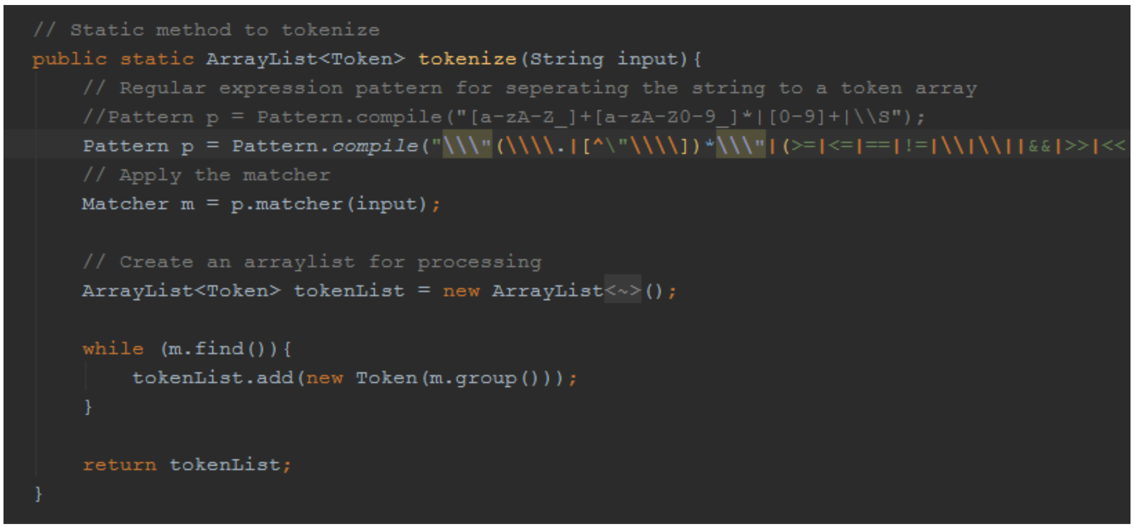
The Tokenizer converts string literal input into objects of type "Token." But the ultimate result of the Tokenizer is an ArrayList object filled with these Token Objects. Below are some examples of input and the Tokenizer's resulting output:

Example:
Here, let's use the example file we discussed earlier named "TokenizerExampleTest.java." In it, we see this:



```
// Example we'll be working with
String mainExample = "9*3 + foo77 + (12)";

// Let's tokenize mainExample
// You'll need a Token ArrayList, we'll call it tokenList
ArrayList<Token> tokenList = Token.tokenize(mainExample);
```

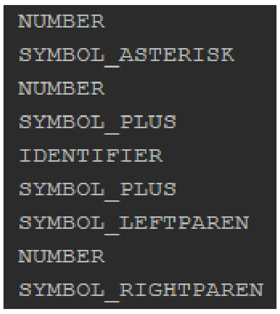
The first line "String myExample" can be thought of as our input program. We take this input program and resolve it to a string. In this case, although it is a string literal, in other instances it may be a text file. This string is then sent to the static method "tokenize()" in class Token as the parameter. The result of this function is an ArrayList<Token> (and in this example) called "tokenList." Here is the tokenize() function:



```
// Static method to tokenize
public static ArrayList<Token> tokenize(String input){
    // Regular expression pattern for seperating the string to a token array
    //Pattern p = Pattern.compile("[a-zA-Z_]+[a-zA-Z0-9_]*|[0-9]+|\\S");
    Pattern p = Pattern.compile("\\\"(\\\\.|[^\"\\\\])*\\\"|(>=|<=|==|!=|\\|\\||&&|>>|<<
    // Apply the matcher
    Matcher m = p.matcher(input);

    // Create an arraylist for processing
    ArrayList<Token> tokenList = new ArrayList<~>();

    while (m.find()){
        tokenList.add(new Token(m.group()));
    }

    return tokenList;
}
```

The cut off portion is already listed in the "Variable Naming Restrictions" section. We define a Regular Expression pattern to match against our input String. The result is a Matcher object, which we convert to an ArrayList using the while loop. The final output is an ArrayList<Token>, which is returned at the end of the method.

So, our original input string was "9*3 + foo77 + (12)" and our output from the Tokenizer is an ArrayList. When outputted, this is our result:



```
NUMBER
SYMBOL_ASTERISK
NUMBER
SYMBOL_PLUS
IDENTIFIER
SYMBOL_PLUS
SYMBOL_LEFTPAREN
NUMBER
SYMBOL_RIGHTPAREN
```

The Tokenizer was able to recognize all symbols and numbers, but the word "foo77," since it is not in our reserved words list, is found to be an Identifier. At this stage, the Tokenizer does not know, or care, whether "foo77" is a class name, variable name, or method name. This is handled within the Parser.

Section 5. Parser

Files Pertaining to the Parser

- MainParser.java – This file has some functionality to parse individual statements or tokens, as well as also functioning as a tester for the Parser.
- PClassDeclaration.java – This is one of the numerous Parser object classes used to hold parsed statements from the result of the Parser. This particular file is used to hold class declarations. It holds the name of the class (identifier), access modifier, and if the class extends another class. This object also holds an ArrayList of PDeclaration statements in the class, which essentially means all methods, variables, and any other lines of code present in the class.
- PDeclaration.java – This class is an interface for two other classes, both of which are declarations. The two classes which implement PDeclaration.java are PVariableDeclaration and PStatementFunctionDeclaration. The first of these are for variable declarations and the second is for method declarations.
- PExpression.java – This is another interface class to hold some types of expressions. This class is implemented by numerous other classes, all related to expressions or pieces of expressions.
- PExpressionAtom.java – This class is very similar to the previous entry, and actually extends it. The difference being that this class is only for single atoms of information, like values or calls to a function.
- PExpressionAtomBooleanLiteral.java – Used to store Boolean literal tokens.
- PExpressionAtomNullLiteral.java – Used to store null literal tokens.
- PExpressionAtomNumberLiteral.java – Used to store number literal tokens.

- PExpressionAtomStringLiteral.java – Used to store string literal tokens.
- PExpressionBinOp.java – This class is used to store a binary operator expression in its entirety, including both sides of the expression and the operator token.
- PExpressionOperator.java – Used to store operator tokens.
- PExpressionParserElement.java – Used to describe elements in an expression.
- PExpressionStub.java – Unused class used to document unknown objects.
- PIdentifierReference.java – This class is to store a specific type of PExpression, the reference of an internal component of an object. For example, if you want to call the "test()" method of the "temp" object, this would be stored as a PExpressionIdentifierReference object for our compiler.
- PStatement.java – Generic interface to classify certain classes as statements.
- PStatementForStatement.java – This class holds a "for" loop statement object and all parts of this statement. This includes the three parts in the header and all body statements.
- PStatementFunctionCall.java – Similar to PIdentifierReference, this class holds statements in the form of "object.method(parameters);" and includes as many parameters as necessary. Each of the parameters are stored as an ArrayList of PExpression objects.
- PStatementFunctionDeclaration.java – This class holds an entire function declaration. This includes the name of the method, the access modifier, the return type, parameters list, and entire statements list.
- PStatementIfStatement.java – This class holds an "if" statement object, including the expression in the header and all statements in the body. This includes the mandatory "else" keyword and body as well.
- PStatementPrintln.java – Used to hold any print statements.

- PStatementReturn.java – Used to hold the "return" keyword and the expression object that is to be returned.
- PStatementWhileStatement.java – This class hold a "while" loop object in its entirety. This includes the expression in the header expression and all statements in the body.
- PVariableAssignment.java – This class is used to hold the statement which gives an assignment to a certain variable name. Note this is not a variable declaration, but giving an assignment to an already declared variable.
- PVariableDeclaration.java – This class holds a variable declaration, including the name of the variable, the type, the access modifier, and an optional assignment.

Parser Library

Our parser makes use of a Java parser combinator framework called "funcj.parser." More information about this library can be found at this link:

https://github.com/typemeta/funcj/tree/master/parser

General Parsing Restrictions

The most common use of our parser, and in its final form, will only allow the user to parse entire programs. Although it contains the functionality to parse individual elements, we generally do not use this. So, given that we must parse an entire program, we must have at least one class to put our statements into. Also, because of the standard practices of the

Java language, which CoopJa is based upon, we cannot have any loops outside of a method. So this section will describe how to properly define classes and methods in order to pass the Parser. If a program does not pass the Parser, it has some improper syntax and will fail to compile.

In CoopJa, the proper way to define a class is:

<access modifier> class <identifier> [extends <class identifier>]*

* = optional

Our classes require an access modifier, which in most cases will be public, then must be followed by the "class" keyword," and then the class's identifier. If a class is to extend another class, then the keyword "extends" must be present, followed by the name of the class it is to extend. However, if a class tries to extend a class that has Private Access Type, this is not allowed. The extending feature is optional, so if a class does not extend anything, it would simply be, for example, "public class One."

In regards to the "static" keyword, CoopJa does not support it. This means there should not be use of the static keyword in the class definition, nor in a method declaration.

Variables can only be given an assignment inside of a method. This is due to the Code Generation needing to generate struct types for C and initial declarations would be ignored if allowed.

In order to properly define a method, it is not so different than standard Java definitions. The way to define a class in CoopJa is:

<access modifier>* <return type> <identifier> (<parameters list comma separated>*)

* = optional

So a typical method declaration might look like "public int main()." If no access modifier is given, it will be treated to be public.

Another restriction in CoopJa relates to "if" statements. We decided to restrict "if" statements to only occur with an "else" token as well. This means and "if" statement cannot exist without the "else" token, or it will fail at the Parser. The correct syntax is listed here: "if (<expression>) { <statements>* } else { <statements>* }"

<u>Parser Output</u>

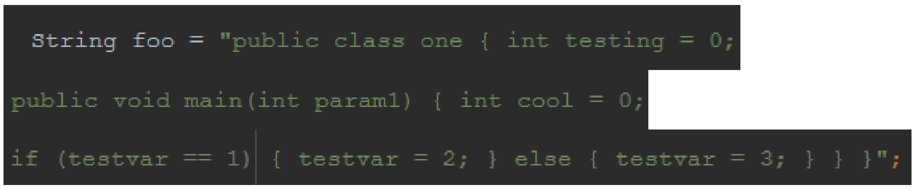
This section discusses what the output of our Parser is, but only within the context of an entire program, similar to the previous section.

We use a token list and MainParser object to assign the output to a PProgram object, which is the final output of the Parser. This PProgram object contains an ArrayList of PClassDeclaration objects, each with a class declaration and all variables/methods for the stated class. Each PClassDeclaration object has an ArrayList of PDeclaration objects, containing either a PVariableDeclaration object or a PStatementFunctionDeclaration object. This means that inside this class declaration, there are a list of variables and methods defined within the class. Although PVariableDeclaration objects are strictly for the declaration of a variable, inside the PStatementFunctionDeclaration object, there may be any type of statement in its statement list, including "if" statements, "for" loops, "while" loops, variable assignments, etc.

Example:

In the MainParser.java file, there is code to test out the Parser. This example is from this file.

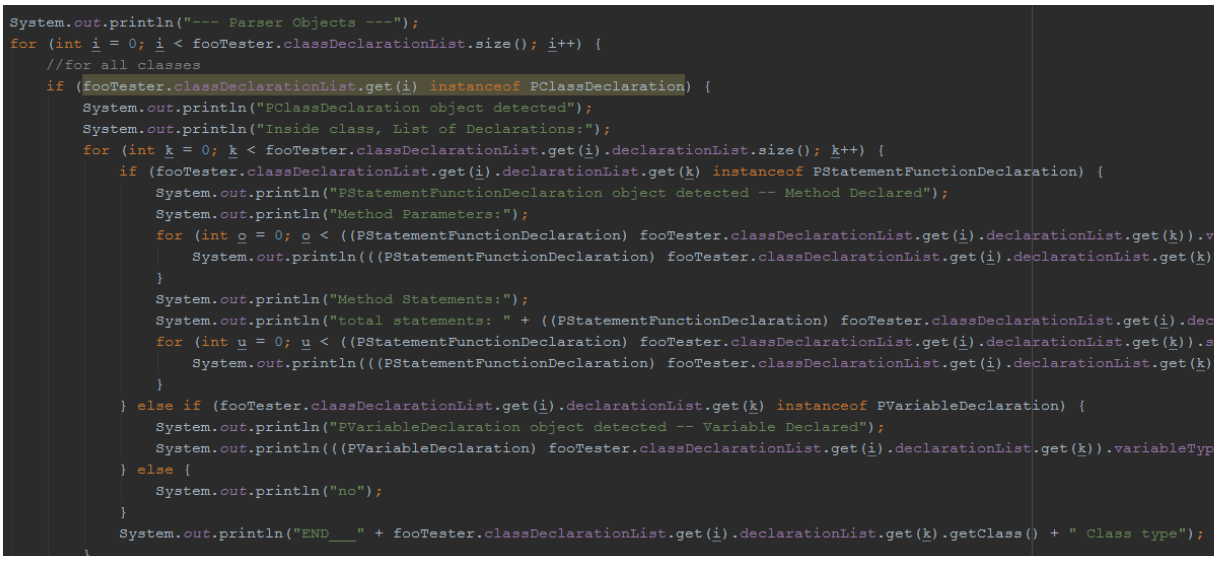
Here we have an example input string to act as our input program:



```
  String foo = "public class one { int testing = 0;
public void main(int param1) { int cool = 0;
if (testvar == 1) { testvar = 2; } else { testvar = 3; } } }";
```

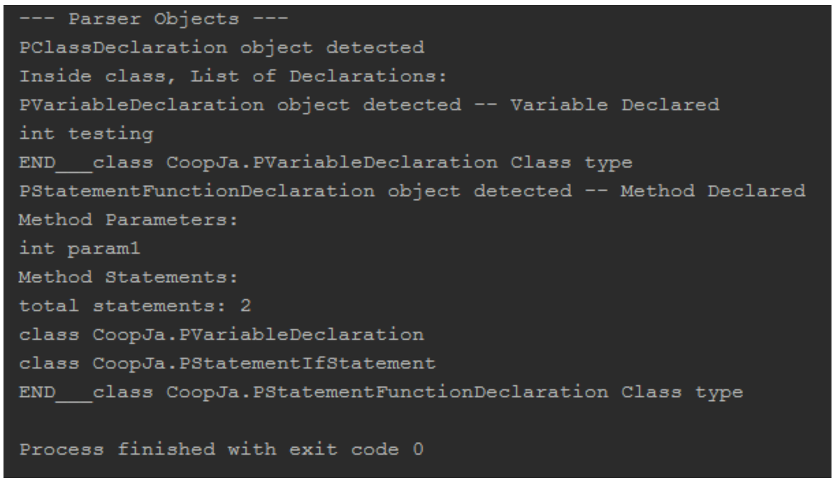
In the example, we have a main class with one variable declaration and one method declaration. Inside the method declaration, we have one parameter, one other variable declaration, and one "if" statement declaration.

Near the bottom of the class, there is some logic to somewhat peal apart the output parser object and discover which statements were resolved from the Parser:



```
System.out.println("--- Parser Objects ---");
for (int i = 0; i < fooTester.classDeclarationList.size(); i++) {
    //for all classes
    if (fooTester.classDeclarationList.get(i) instanceof PClassDeclaration) {
        System.out.println("PClassDeclaration object detected");
        System.out.println("Inside class, List of Declarations:");
        for (int k = 0; k < fooTester.classDeclarationList.get(i).declarationList.size(); k++) {
            if (fooTester.classDeclarationList.get(i).declarationList.get(k) instanceof PStatementFunctionDeclaration) {
                System.out.println("PStatementFunctionDeclaration object detected -- Method Declared");
                System.out.println("Method Parameters:");
                for (int o = 0; o < ((PStatementFunctionDeclaration) fooTester.classDeclarationList.get(i).declarationList.get(k)).v
                    System.out.println(((PStatementFunctionDeclaration) fooTester.classDeclarationList.get(i).declarationList.get(k)
                }
                System.out.println("Method Statements:");
                System.out.println("total statements: " + ((PStatementFunctionDeclaration) fooTester.classDeclarationList.get(i).dec
                for (int u = 0; u < ((PStatementFunctionDeclaration) fooTester.classDeclarationList.get(i).declarationList.get(k)).s
                    System.out.println(((PStatementFunctionDeclaration) fooTester.classDeclarationList.get(i).declarationList.get(k)
                }
            } else if (fooTester.classDeclarationList.get(i).declarationList.get(k) instanceof PVariableDeclaration) {
                System.out.println("PVariableDeclaration object detected -- Variable Declared");
                System.out.println(((PVariableDeclaration) fooTester.classDeclarationList.get(i).declarationList.get(k)).variableTyp
            } else {
                System.out.println("no");
            }
            System.out.println("END___" + fooTester.classDeclarationList.get(i).declarationList.get(k).getClass() + " Class type");
```

When we run the main() method of the MainParser class, we can see which objects the Parser resolved:



```
--- Parser Objects ---
PClassDeclaration object detected
Inside class, List of Declarations:
PVariableDeclaration object detected -- Variable Declared
int testing
END___class CoopJa.PVariableDeclaration Class type
PStatementFunctionDeclaration object detected -- Method Declared
Method Parameters:
int param1
Method Statements:
total statements: 2
class CoopJa.PVariableDeclaration
class CoopJa.PStatementIfStatement
END___class CoopJa.PStatementFunctionDeclaration Class type

Process finished with exit code 0
```

We can see the result is one class, inside of which we have one variable declaration "int testing" and a method declaration. Within the method, there is one parameter "int param1" and two total statements, a variable declaration and an "if" statement object. We can also see that the program finished with no errors, which is the result of a successful parsing.

Section 6. Typechecker

Files Pertaining to the Typechecker

- Typechecker.java – This is the entire Typechecker file. It also contains several sub-classes that will be discussed in the next section.
- TypeCheckerException.java – This is the class for all Typechecker exceptions. It will print a message about what was wrong from the Typechecker.
- test\Typechecker_UnitTests.java – This class contains all of the Unit Tests for the Typechecker.

Sub-Classes of Typechecker.java

The Typechecker has numerous sub-classes that are integral to its functionality. Here each one will be explained.

- VarStor class – A VarStor object, short for "variable storage," holds information about a particular variable, including its type, its access modifier, and the name of their identifier class (if they are of type "classname"). Notice that the identifier of this specific variable is information a VarStor object is distinctly missing. This is because these VarStor objects are stored in a HashMap as values. The key of this VarStor object is its identifier. More info about this will be in further class descriptions.

- FunctStor class – A FunctStor object, short for "function storage," similar to a VarStor object, holds everything the Typechecker needs to know about a specific method, including its return type, parameters list, and most importantly the variables declared inside it. The idea behind the variable storage is to use the previously explained VarStor objects in a HashMap to hold these declared variables' information. This class also does not contain its own name, for the same reason as VarStor. These two classes are used in tandem to create the Storage class.
- Storage class – A Storage object holds all information the Typechecker needs to know about a specific class, a list of its Variables and Methods and their types. It holds all of its variables in a HashMap. The way this works is predicated on the existence of the VarStor class. The HashMap which holds the variables is of type HashMap<String, VarStor>, where String is the key and VarStor is the value. The thinking behind this involves the Typechecker knowing the name it needs to discover the type of only, like in the case of a variable assignment. For example, at the top of a class, a variable "int foo;" is declared. Later on, the statement "foo = 9;" is given. At this statement, the Typechecker only knows the name of "foo" and must resolve its type. So it uses this name to pull from its HashMap the VarStor object that contains its type. The same logic is involved with the storing of methods, except the HashMap is of type HashMap<String, FunctStor>. Inside the Storage object, if this class extends another, is the entire Storage object of its parent class. This is so the Typechecker can know the parent scope of any variable in the child to, for example, make sure there are no naming conflicts with the parent.

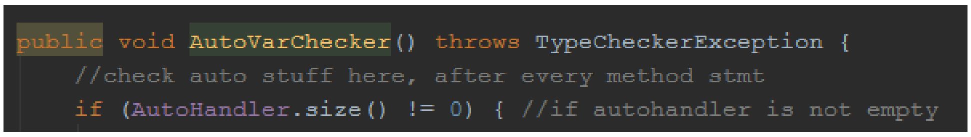
- AutoTicket class – An AutoTicket object is a "ticket" that is generated when the Typechecker encounters a variable of type "auto" for later evaluation. Lots of information is gathered about exactly where the auto variable was encountered in the class declaration list (for later replacement), although not all of it is used. Some of the key pieces of information are: a Boolean to determine if this ticket was generated at the declaration statement of the variable, target name of the variable to change the type of (auto var), and the new type to change the auto variable to. An AutoTicket is also generated when the Typechecker knows the new type of this variable. More information about how exactly the Typechecker uses these AutoTickets to resolve a type will be in a later section.
- ChildOverride class – This is an old and unused class pertaining to the altering of the parser object. More info on this will be in a later section.

Typechecker Specific Jobs

Besides the basic functionality of a typechecker described in an earlier section, our Typechecker has a few important jobs to do before the code hits Code Generation. Those jobs and why they must be done will be described in this section.

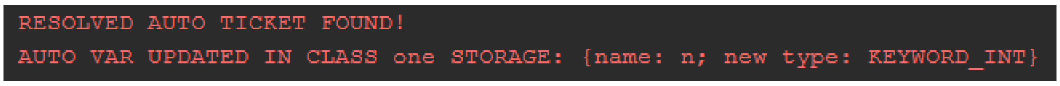
Auto Type Fixing

The Typechecker, when checking a class, will find any variable declared as type auto. When an auto type variable has been detected, it uses its sub class "AutoTicket" to generate a "ticket" for this specific variable to handle later. A ticket is generated each time this variable is encountered to try to resolve its type. One ticket is generated at the point of declaration, and one is generated when its type has been resolved, but this is only the instance of not immediately giving the variable a type. So, for any successful auto variable detection and new type resolution, there will be a minimum of one AutoTicket generated. After a method statement has been checked, the Typechecker will run a method called AutoVarChecker() to try to resolve any pending AutoTickets.



```
public void AutoVarChecker() throws TypeCheckerException {
    //check auto stuff here, after every method stmt
    if (AutoHandler.size() != 0) { //if autohandler is not empty
```

The first thing this method does is check to see if there are any AutoTickets in the ArrayList "AutoHandler" to resolve, or try to. It then goes through this list, grouping AutoTickets of the same variable name, in hopes to find the two AutoTickets needed to resolve the variable type: its declaration position and new type. It then attempts to resolve this variable by seeing if the NewType field of the AutoTicket is not null. This means that the Typechecker was able to resolve the type of this variable and updates the type in its Storage variable.



```
RESOLVED AUTO TICKET FOUND!
AUTO VAR UPDATED IN CLASS one STORAGE: {name: n; new type: KEYWORD_INT}
```

The reason that the Typechecker checks for AutoTickets after each method statement is in the hopes of resolving the auto variable early. For example, if an auto variable was declared and instantly given a value, it makes more sense for there to be only one AutoTicket generated and resolve the variable faster. After a single variable has been resolved, all AutoTickets corresponding to that variable are removed from the list. At the end of the entire program, the Typechecker checks the "AutoHandler" variable to check if it is empty. This signals the Typechecker that all auto variables were given a type and were able to be resolved. If this list is not empty, the Typechecker will throw an exception.

```
CoopJa.TypeCheckerException: Auto Typecheck Error: Could not resolve some Auto Variable Types
AutoHandler Size: 2
VarName: k in Class: one, Awaited Type: null
```

Parser Output Reordering

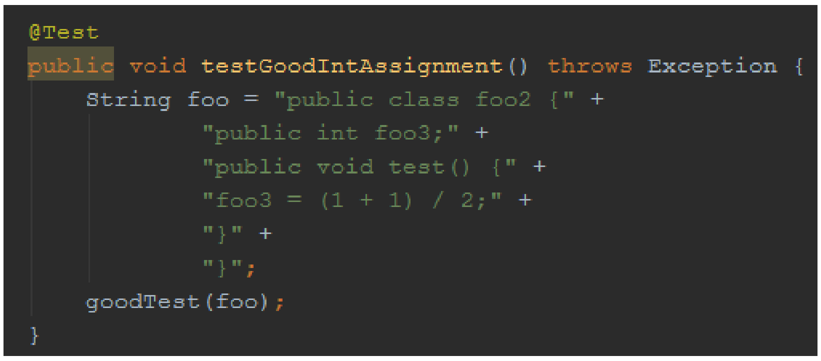
After the parser runs, it generates an output file of type PProgram, which corresponds to the entire program that is parsed into various parser objects, depending on their type. The Typechecker then takes this PProgram object in as input and typechecks it. But after the Typechecker runs, it returns a new updated PProgram object. There are two situations in which the Typechecker needs to update the original input program: Auto Type and Inheritance.

Both of these situations are due to the way Code Generation works for CoopJa. Although it will be explained more fully in a later section, briefly: each P-Type object that the parser generates has code to "print itself" into valid C code. In the case of Auto type, it must be removed before Code Generation runs. In the case of inheritance and how it was implemented, a child must have the same "header" definitions in its struct as its parent. For example, if the parent has declared a variable "int x" and a method "public int run()", no matter the order the child declared them in, and also considering its own unique declarations, the first two declarations it has must be "int x" and "public int run()." So, if the child declared, in the following order, "int a", "int b", "public int main()", the final order that the Typechecker will change it to is: "int x", "public int run()","int a", "int b", and "public int main()."

Typechecker Output

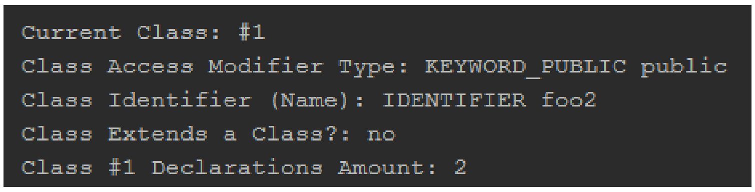
This section takes a unit test and breaks it apart to show exactly what the Typechecker outputs.

Take the following as an example of an input program:



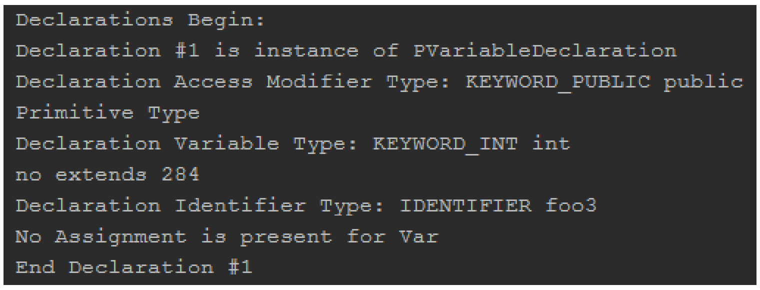
```java
@Test
public void testGoodIntAssignment() throws Exception {
    String foo = "public class foo2 {" +
            "public int foo3;" +
            "public void test() {" +
            "foo3 = (1 + 1) / 2;" +
            "}" +
            "}";
    goodTest(foo);
}
```

When the Typechecker is run, it will begin by showing the (first) current class information:



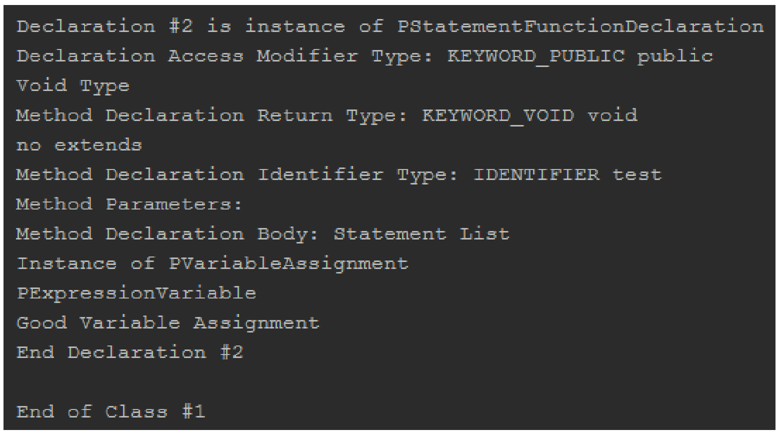
```
Current Class: #1
Class Access Modifier Type: KEYWORD_PUBLIC public
Class Identifier (Name): IDENTIFIER foo2
Class Extends a Class?: no
Class #1 Declarations Amount: 2
```

Here we see it has found the first class to be called “foo2,” which has a “public” access modifier and does not extend another class. It also shows the total declaration amount.



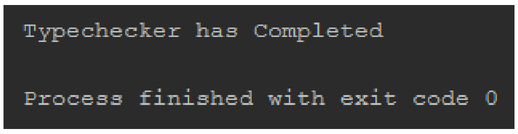
```
Declarations Begin:
Declaration #1 is instance of PVariableDeclaration
Declaration Access Modifier Type: KEYWORD_PUBLIC public
Primitive Type
Declaration Variable Type: KEYWORD_INT int
no extends 284
Declaration Identifier Type: IDENTIFIER foo3
No Assignment is present for Var
End Declaration #1
```

Next, it shows the declarations the class has in order. The first is of type PVariableDeclaration, which is the parser object for a variable declaration. We can also see the information about this variable declaration, like its name, access modifier, and type.



```
Declaration #2 is instance of PStatementFunctionDeclaration
Declaration Access Modifier Type: KEYWORD_PUBLIC public
Void Type
Method Declaration Return Type: KEYWORD_VOID void
no extends
Method Declaration Identifier Type: IDENTIFIER test
Method Parameters:
Method Declaration Body: Statement List
Instance of PVariableAssignment
PExpressionVariable
Good Variable Assignment
End Declaration #2

End of Class #1
```

After that, we move on to the next declaration, which is of type PStatementFunctionDeclaration. This is referring to the class declaring a function, and we can see here the information about this function. We can also see the function's statement list, which in this case is a variable assignment. After all declarations for the class are checked, the Typechecker outputs that it is at the end of the class. In this case, that is the end of the Typechecker output.



```
Typechecker has Completed

Process finished with exit code 0
```

It should be noted that in some instances, red text will also be displayed in the output. As long as no exception is thrown, this is normal.

Section 7. Code Generation

Files Pertaining to Code Generation

- All P-Type Files – Since each component generates its own C code.
- CodeGenException.java – Code Generation's exception class
- J_CodeGen_ExpressionTest.java – Used to run the C compiler in our project, "tcc," at the end of unit tests to match output.
- test\CodeGen_UnitTests.java – The unit tests for Code Generation.
- (old file) C_CodeGenTest.java – Old file used for testing.
- (old file) N_CodeGenAdd.java – Old file used for testing.

Object Oriented Code Generation

The job of the Code Generator (CG) is to convert the input program in the Source Language (SL) into valid code in the Target Language (TL) so that it is executable. The CG must do its work after the parser and type-checker in order for the code to be valid in the TL. The CG takes as input the input program file. It goes through the code and, if necessary, modifies it to work in C. It turned out that the way we did code generation is through an internal method, rather than an external method, meaning it is the PObject's responsibility to generate its own code string. In turn, PObjects have a method named `generateObjectNameString( )` which returns its object's code as a String, and this PObject calls any nested PObjects' `generateObjectNameString( )` method.

Because C is not an Object Oriented programming language, our team had to figure out ways of making Object Oriented programming viable in C. It was decided that classes would be made into structs. Class members in the SL were made into variables of the struct in the TL. Method members of a class in SL become function pointer members of their pertaining struct in TL to carry out class inheritance. The definition of these functions (pointers) lie outside the struct because they are not allowed to be defined in a C struct.

Variables that are struct members are given an identifier: `structname_function`.

Each struct function member has an initializing function that initializes the function with its definition.

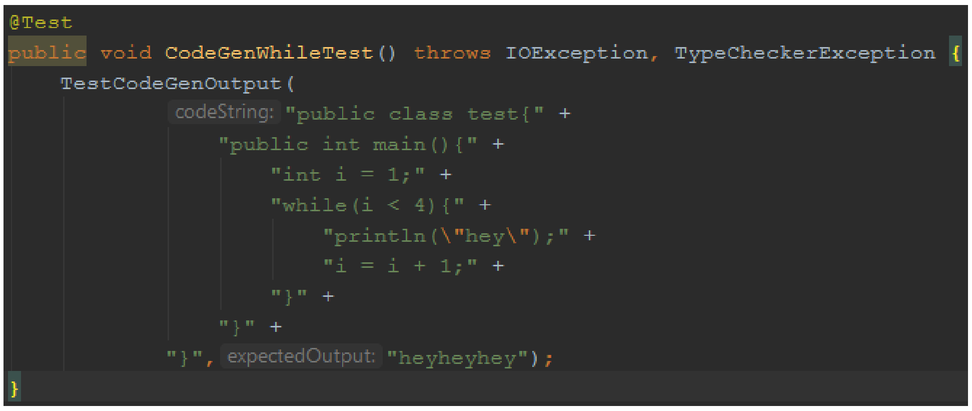
While processing the class function declarations in the SL, we search for a main function, and we treat it differently to the other functions. We define the main function with the parameters it needs in the TL. To implement class-based inheritance, we did a lookup technique, where superclasses are looked up and its code is put in the subclasses' struct, and any overrides of functions in the subclass are able to be done through our use of function pointers in the TL.

After the code generator process, we save this as a C file.

Code Generation Output

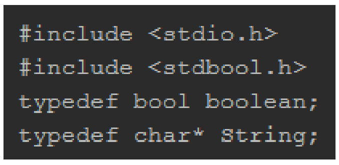
This section takes a unit test and breaks it apart to show exactly what is outputted as a result of Code Generation.

Take the following as an example of an input program:



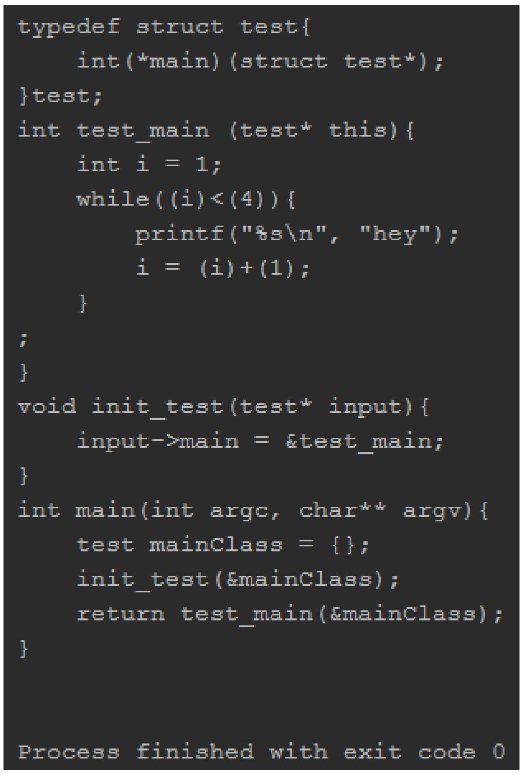
```
@Test
public void CodeGenWhileTest() throws IOException, TypeCheckerException {
    TestCodeGenOutput(
            codeString: "public class test{" +
                "public int main(){" +
                    "int i = 1;" +
                    "while(i < 4){" +
                        "println(\"hey\");" +
                        "i = i + 1;" +
                    "}" +
                "}" +
            "}", expectedOutput: "heyheyhey");
}
```

We see an example class that loops through and prints the word “hey” three times. Here is the output C code that is produced:



```
#include <stdio.h>
#include <stdbool.h>
typedef bool boolean;
typedef char* String;
```

First there are the headers that are on every program. There are two #include files and two renamed types “boolean” and “String” to correspond with the token types in CoopJa. The rest of the output is:



```
typedef struct test{
    int(*main)(struct test*);
}test;
int test_main (test* this){
    int i = 1;
    while((i)<(4)){
        printf("%s\n", "hey");
        i = (i)+(1);
    }
;
}
void init_test(test* input){
    input->main = &test_main;
}
int main(int argc, char** argv){
    test mainClass = {};
    init_test(&mainClass);
    return test_main(&mainClass);
}


Process finished with exit code 0
```

We can see the class is converted into a struct that holds its member. The main method creates a test instance and initializes it, which initializes a function pointer. We also see that the class test's function called "main" has now been changed to fit the form of

[class name]_[function name], in this case "test_main"

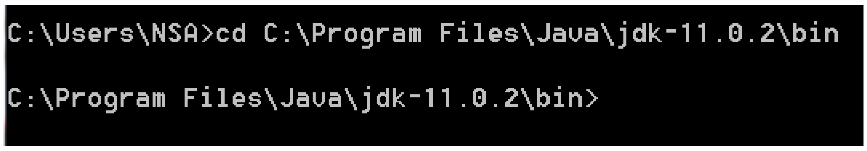
Because this is in a unit test, it has also been run through the C compiler in our project "tcc" and the output of which is compared with expected output. The "Process finished with exit code 0" indicates that was successful.

<u>Section 8. Compiling and Running CoopJa</u>

<u>*Running CoopJa from the JAR File*</u>

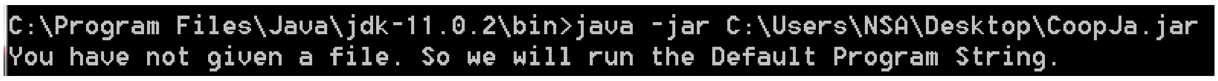
Included in the root of the project file is a file "CoopJa.jar," which is the compiled version of our project. The following is a Windows Operating System based way to run this jar file and the outputs that will occur.

1. Run the Command Prompt (cmd.exe) [Note: it is recommended to "Run as Administrator", which will be explained in a later step]
2. Move to the directory of "java.exe", where Java is installed in the system ("cd [dir]"). Our version was "jdk-11.0.2", so it is recommended to be around there.



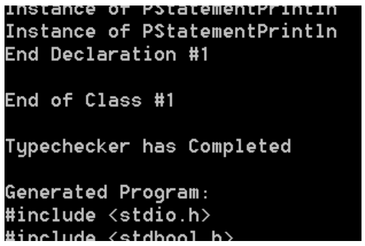
```
C:\Users\NSA>cd C:\Program Files\Java\jdk-11.0.2\bin

C:\Program Files\Java\jdk-11.0.2\bin>
```

3. Type the command "java –jar [location]\CoopJa.jar" to see the default output of CoopJa. [location] refers to the directory of the "CoopJa.jar" file. In this case, it is located on the desktop.



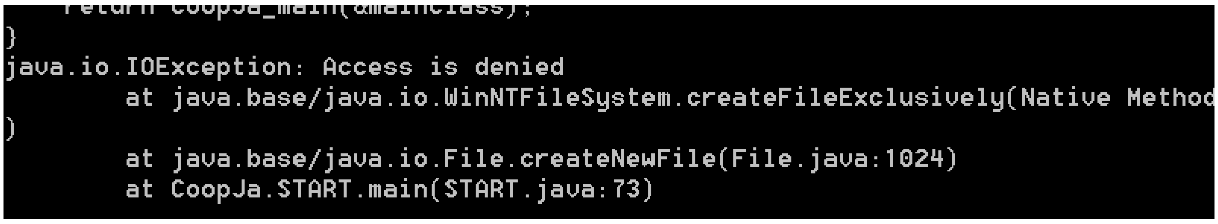
```
C:\Program Files\Java\jdk-11.0.2\bin>java -jar C:\Users\NSA\Desktop\CoopJa.jar
You have not given a file. So we will run the Default Program String.
```

4. Below this line, you will see the Typechecker output, followed by the Code Generation output ("Generated Program:").



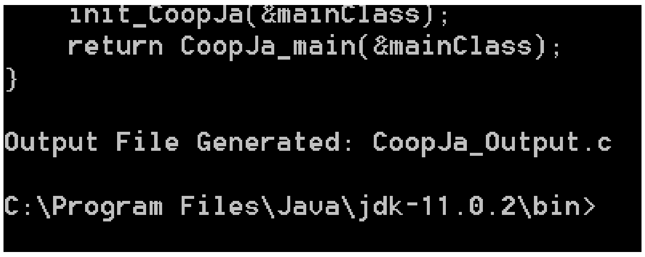
```
Instance of PStatementPrintln
Instance of PStatementPrintln
End Declaration #1

End of Class #1

Typechecker has Completed

Generated Program:
#include <stdio.h>
#include <stdbool.h>
```

5. After the C code is printed to the console, it will attempt to write this output to a file called "CoopJa_Output.c". If the Command Prompt was not run with Admin privileges, you will see this message:



```
   return CoopJa_main(&mainClass);
}
java.io.IOException: Access is denied
        at java.base/java.io.WinNTFileSystem.createFileExclusively(Native Method
)
        at java.base/java.io.File.createNewFile(File.java:1024)
        at CoopJa.START.main(START.java:73)
```

6. If the Command Prompt was run with Admin privileges, you will see this message:

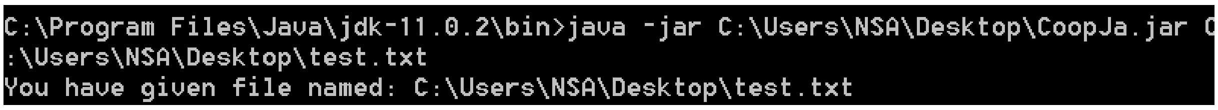


```
    init_CoopJa(&mainClass);
    return CoopJa_main(&mainClass);
}

Output File Generated: CoopJa_Output.c

C:\Program Files\Java\jdk-11.0.2\bin>
```

- Note: This "CoopJa_Output.c" output file will be put in the directory of "java.exe", not the directory of the CoopJa jar file.
- Note: If no input file is given, output will still be generated, as there is a default input program built in for this case.

- If you want to run the file with a custom input program, here is an example of that:
  - Navigate to the "java.exe" directory and run the revised command

    "java –jar [location]\CoopJa.jar [input file dir]\[input file name with extension]"

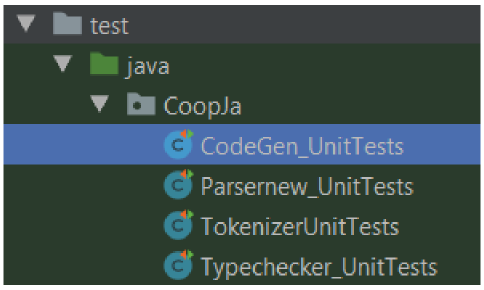


```
C:\Program Files\Java\jdk-11.0.2\bin>java -jar C:\Users\NSA\Desktop\CoopJa.jar C
:\Users\NSA\Desktop\test.txt
You have given file named: C:\Users\NSA\Desktop\test.txt
```

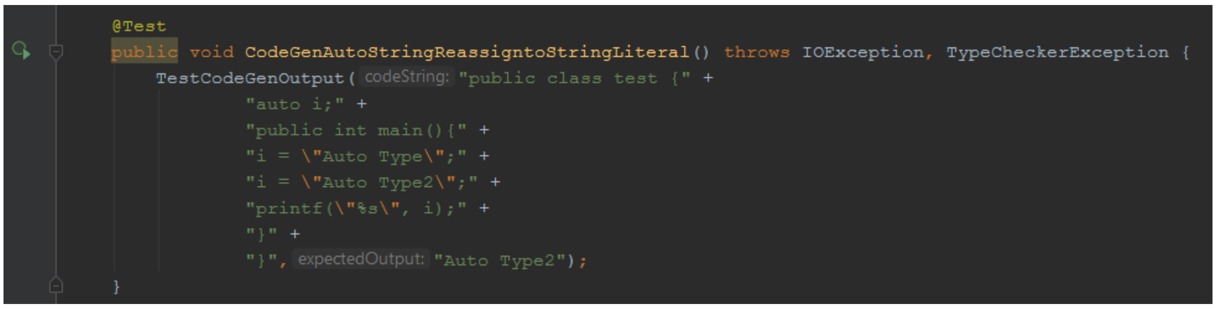
Running CoopJa from the IDE

Our project was created with IntelliJ IDEA and can be ran from this as well. We can specifically use our unit case file to do so cleanly.

1. Open the project in IntelliJ and open the "src" folder. Then open the "test" folder and open the "CodeGen_UnitTests.java" file



2. Go to any one of the methods with the yellow header "@Test"



```
@Test
public void CodeGenAutoStringReassigntoStringLiteral() throws IOException, TypeCheckerException {
    TestCodeGenOutput( codeString: "public class test {" +
            "auto i;" +
            "public int main(){" +
            "i = \"Auto Type\";" +
            "i = \"Auto Type2\";" +
            "printf(\"%s\", i);" +
            "}" +
            "}", expectedOutput: "Auto Type2");
}
```

3. Replace the "codeString" with a different string and write an "expectedOutput" string to match the output the compiler will give.
4. Click the green circle / play button to the left of that specific method to run that single test.

- Note: This generates an ".c" that is then run through the "tcc" compiler at the directory of our project "...\CoopJa\tcc\tcc.exe"