# **Berg-Verhelst-Compiler Phase 5**

“Code Generation”

## Introduction

This submission is for the Code Generator phase of the compiler project. Included is the code for the code generator, semantic analyzer, parser, scanner, the executable and supporting files, test files and this document. The code generator takes the annotated AST as given by the semantic analyzer, and traverses it and generates quadruple code.

Participation

The new addition to this phase is CodeGen.java. In this class, the member who created the method is included in the method comment as well as if the other member made changes to that method. The split of the work of the methods was evenly split between members. Changes to the previous phases was also split evenly, as reflected in comments.

## Project Status

For this phase we implemented a code generator that takes an annotated AST and produces quadruple code. Unfortunately we were unable to fully run the interpreter due to issues when you enter nested compound statements and use returns. However our generated quadruples appear to be valid when manually examined. Since the program’s syntax and semantics are validated before this phase, the code generator will produce no error.

**Phase 1: Scanner (Submitted Sept 20, 2013)**

Scanner: Completed

- Removed Recursive Dependencies

- Added command line libraries to replace hand coded functions

- Separated TokenType from Token

Parser Basic: Consumes token in a while loop until end-of-file is found

Parser Full: Not Started

Semantic Analyzer: Not Started

Code Generator: Not Started

**Phase 2: Basic Parser (Submitted Oct 9, 2013)**

Scanner:

- Removed recursive dependency from the project

- Expanded the options available from the command line

- Trace now is contained by the scanner instead of admin console

- TokenType moved to its own class

Parser Basic:

- Creating the parse tree done

- Creating ASTNodes done

- Trace is available through the parser

Parser Full:

- Some basic code for panic mode (Currently commented out)

Semantic Analyzer: Not Started

Code Generator: Not Started

**Phase 3: Parser With Error Recovery (Submitted Oct 16, 2013)**

Scanner:

- Completed

Parser Basic:

- Left-most derivation completed

- Creating the parse tree completed

- Creating ASTNodes completed

- Trace is available through the parser

Parser Full:

- Error Recovery: Panic mode with Synch Sets completed

Semantic Analyzer: Not Started

Code Generator: Not Started

**Phase 4:Semantic Analyzer (Submitted Oct 30, 2013)**

Scanner:

- Completed

- Removed dependencies on external files

Parser Basic:

- Completed

Parser Full:

- Completed

- Changed from using Hashmaps for first and follow sets to using an external enum (FFSet.java)

- Fixed usages of first sets for the generation of synch sets

Semantic Analyzer:

- Added *Declaration* interface to ASTNode

- Traversing of AST from parser (init traversal, full traversal) completed

- Scope analysis completed

- Type checking completed

- Semantic rule checks completed

+ Error printing needs line numbers

Code Generator: Not Started

**Phase 5: Code Generator (Last Submission: Nov 20, 2013)**

Scanner:

- Completed

Parser Basic:

- Completed

Parser Full:

- Completed

Semantic Analyzer:

- Recovery of types fixed. If mismatch is found, UNI type is used

- Undeclared functions do not throw null pointer errors anymore

- Fixed bug for Array Indexes that would return a null pointer

- Fixed bug from redeclaration in same scope error handling

- Fixed errors caused by pre-generated functions (ex. writeint)

- Changed “-help” to “-h” and added “-ast” option as requested

+ Uninitialized variables (started, but Jernej said it was unnecessary as it is a runtime error)

Code Generator:

- Traverse the annotated AST produced by the Semantic Analyzer

- Input / Output functions completed

- Assignment/Copying functions completed

+ Unary Operations (lacks NOT function)

- Binary Operations completed

- Flow Control completed

- Structure completed

- Assignments involving array elements completed

+ (Level, Displacement) representation of variables is not implemented.

+ Temporary Variables are not reused

**Phase 5: Code Generator (Current Submission: Nov 28, 2013)**

Scanner:

- Completed

Parser Basic:

- Completed

Parser Full:

- Completed

Semantic Analyzer:

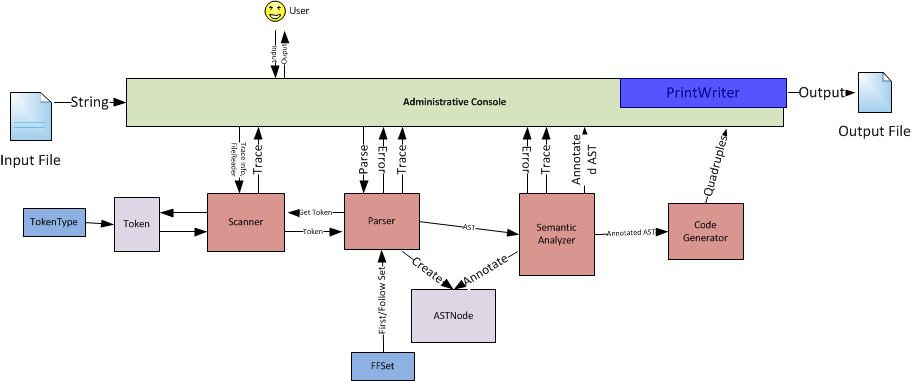
- Completed

Code Generator:

- Not Operator code generation added

- (Level, Displacement) code generation added

## Architecture and Design



This phase builds on the architecture of the previous phase. The Administrative Console operated in the same manner, although it has been internally optimized. It still sets the compiler environment based off of the supplied arguments. It initiates a Scanner, Parser, Semantic Analyzer and the Code Generator and executes each component in the order they have been mentioned. The AST is passed from the Parser to the Semantic Analyzer where it is annotated and finally is passed to the Code Generator where it is recursively traversed to produce quadruple code for the Quadruple Interpreter.

**Traversal Of Annotated AST**

The code generator traverses the annotated AST. Before it begins traversal it examines the program node to find the size requirement of the global variables. It stores this number in the (start, <global size>, -, -) quadruple. A single variable is given a size of one (1) and an array is given the size equivalent to the number of items it is declared to hold. The code generator then completes the first four quadruples that are necessary for every quadruple code program and begins traversing the children of the program node, namely the function declarations.

Each node of the tree is used to add quadruple codes or return items that are used within a quadruple, such as variable names and sizes. We use routing methods to decide what ASTNode type statements and expressions are and therefore add the correct quadruples. Expressions are split into quadruples and make use of temporary variables. Temporary variables are generated using the genTemp() method, which simply assigns a name ‘t{numTempVars++}’, where numTempVars starts at one (1). Temporary variables are not reused. For program flow Labels are used. Labels are generated by the genLabel() method which operates in the same manner as the genTemp method.

**Computation of Array Indexes**

Array indexes are a limited expression, that is, the expression denoting the index of the array needs to be able to be computed at compile time. Therefore the index may only contain literal numbers and operators. The semantic analyzer will reject any program that does not adhere to this requirement. The code generator will compute the index of the array during its execution. The expression is computed by a left recursive traversal that applies operators as each side of the operator is computed, the base case is a literal number.

**Finding Local Sizes**

The interpreter requires knowing the number the local variables for functions and compound statements. The number of local is computed for functions by counting the number of variable declarations in the compound statement that immediately follows the function stub. Compound statement’s local size is computed by counting the number of variable declarations in its immediate, non-nested, scope. Arrays count as a number of variables corresponding to the declared size of the array. The computed number of local variables is used in the quadruples corresponding for the function and compound statements.

**Finding Level and Displacement**

Another requirement for the interpreter is the level and displacement of variables used. These were set during the Semantic Analysis traversal as the levels and displacements can be easily calculated during this phase. When a Compound Node is entered the current level is increased and the displacement to be given to the Variable Declaration Nodes. These Variable Declaration Nodes are then entered and the level is set to the current level. Upon leaving the Compound node the current level is decreased. When a Call Node is entered the current level is stored and then sets 1. The Parameter Nodes are then given the negative displacements by negating the total number of parameters and adding to the number as they are checked. Upon leaving the call node the stored level is restored. Finally when a Function Declaration Nodes is entered the Compound node contained is given the displacement offset of 2 as required.

The final part was done during code generation to give temporary variables level and displacement. To accomplish this the level was tracked here also and when a temporary variable was generated this level was used, and the displacement used the local number of variables to offset the displacement.

**Test Programs List**

Correct: test.cs13

Implementation

For this implementation we separated the Code Generator into its own component. This is because we wanted the generator to be its own entity so that it changes to it would not affect other components.

Temporary variables are not reused for simplicity’s sake.

## Building and Use

For convenience we have included some shell files that build the jar, and run a series of tests. To use these shell files, go to your terminal, navigate to the “executables” directory of the project. Use “sh runScanner” to use the program to execute on the test/masterTest.cs13. Use “sh runParser” to run the parser with error recovery on correct and incorrect test files. These files are: test/program.cs13 and test/broken.cs13. Use “sh runSemantic” to run the semantic analyzer on correct and incorrect test files. These files are: test/test.cs13 and test/violation.txt. Use “sh runCode” to run the current phase, it takes in the test.cs13 file. These output files appear in the berg-verhelst-parser-basic\test\output folder when “sh runCode” is used, otherwise they will appear in the specified folder. Surrounding the file paths with quotation marks allows the use of spaces in the file path.

Also, you can run the program using: java -jar BergVerhelstCompiler.jar [options] <files>. Files in the output-test folder of the root directory can be specified using “../output-test/<testfile>”, as will the output file if the user desires to put the generated files in the output-test directory.

The project jar was compiled using Netbeans’s Clean and Build option, since the inclusion of a command line parsing library caused other methods of compiling to fail.

Command Line Arguments:

usage: Administrative Console

-ast Prints the astnode

-code Process up to the code generation phase

-dev Displays development messages

-err <arg> Print error (default (System.out))

-h Displays Help Menu

-o <arg> Print to file (default (System.out))

-parse Process up to the Parser Phase, prints AST

-q Only display error messages (Default)

-scan Process up to Lexical Phase

-sem Process up to the Semantic Phase

-v Display all Trace Messages

**Examples:**

java -jar BergVerhelstCompiler.jar ../output-test/output.cs13 -h

Will run the parser on the input file that is specified. The trace is disabled, only error messages are shown, the help will be displayed as well.

java -jar BergVerhelstCompiler.jar -v -dev “C:\User\Folder With Spaces\program.txt“ -o Out.txt

Will verbosely run the parser, printing all messages including the “Entering/Leaving method:” messages and save the output to out.txt.

## Code

Format: <packagename>/<filename.java>

**Compiler/Token.java**

-Token contains the functionality for the instantiation of tokens. A token consists of a lexeme (required), a token\_type (required) and an attribute value (optional).

**Compiler/TokenType.java**

-Token type contains the enumerations needed for the token types contained in the token class.

**Compiler/TNSet.java**

-TNSet is used to represent first and follow sets, which are sets of Tokens

**Compiler/Main.java**

-Main instantiates an administrative console and executes the compiler

**Compiler/AdministrativeConsole.java**

-The administrative console sets the compilers options based on arguments provided by the user to the compiler. It also has a UI component which provides the user a text based interface for entering options for the compiler.

**Compiler/Parser.java**

-The parser takes the tokens output by the scanner, checks them against production rules and classifies AST Nodes during the parse. Diagnostic information is printed out to the user if errors are found. Both the parse tree and AST Nodes can be shown using the command line options. These results can also be printed to file.

**Compiler/ASTNode.java**

-The ASTNode class has inner classes for the different Node types, as well as 3 interfaces to group the collect of nodes. This allows us to use related nodes easier without having to store multiple types in the related nodes.

Interfaces: Declaration, Statement, Expression

Classes: AssignmentNode, BinopNode, BranchNode, CallNode, CaseNode, CompoundNode, FuncDeclarationNode, IfNode, LiteralNode, LoopNode, MarkerNode, ParameterNode, ProgramNode, ReturnNode, UnopNode, VarDeclarationNode, VariableNode

**Compiler/Scanner.java**

- The scanner tokenizes the input file by using the maximum substring principle to classify lexemes into tokens and if necessary associate an attribute value. The scanner also houses the word table and the symbol table. The scanner skips over whitespace when appropriate. This class checks character types and will create error tokens for invalid characters, symbols or lexemes. This class contains the file(s) used to get the next character and print out traces.

**Compiler/FFSet.java**

-The FFSet class is an enumeration of first and follow sets for the various logical groupings of production statements. A user will retrieve a first or follow set using the syntactic format of the following example. Example: PROGRAM.firstSet();. The firstSet and the followSet are of the form TNSet.

**Compiler/SemAnalyzer.java**

- The Semantic Analyzer takes the root node of the AST produced by the scanner and executes two left recursive descent traversals of the AST. The first traversal retrieves the global declarations and the second traversal fully analyzes the AST: it does type checking and during the descent it also ensures no semantic rules are violated. The initial pass is instantiated by the constructor that takes a ProgramNode as a parameter. The full pass is done by calling the void ProgramNode(ProgramNode node) method.

**Compiler/CodeGen.java**

-The Code Generator receives the annotated AST as annotated by the Semantic Analyzer and uses left recursive descent traversals to generate quadruple code. The generator is initiated and executed on the generateCode(ProgramNode rootNode) method call. The ProgramNode is the root node of the annotated AST tree.

Tests and Observations

**Main Scanner Test Files**

These are the main files used to test the functionality of the scanner and program in general.

**mastertest.cs13**

- This was created to test extra edge cases which the other test did not cover.

**unit/scannerToken.cs13**

- This was created to test all possible scenarios we could think of for testing the generation of tokens. All characters are tested and keywords are used as well as any define specification we found.

**Main Parser Test Files**

**test/test.cs13**

-Test majority of the logical features, these include function calls, functions with parameters, nested if statements, loop and command expressions.

**test/program.cs13**

-Test basic feature more thoroughly and test the case statement

-Test error recovery on correct c\*13 code

**test/broken.cs13**

-Test error recovery on incorrect c\*13 code

**Main Semantic Analyzer Test Files**

**test/test.cs13**

-Test majority of the logical features, these include function calls, functions with parameters, nested if statements, loop and command expressions. Is a syntactically and semantically correct program.

**test/violation.txt**

-Semantically incorrect test file that is syntactically correct. Exhaustively violates semantic rules.

**Main Code Generation Test Files**

**test/test.cs13**

-Test majority of the logical features, these include function calls, functions with parameters, nested if statements, loop and command expressions. Is a syntactically and semantically correct program. This is the program that quadruple code will be generated for.

**Observations**

All phases of the project have been tested using the files listed above. We could not get the Quadruple Interpreter to execute our programs fully because it is unable to deal with nested compound statements with return calls. This made it more difficult to confirm the correctness of our generated code as we had to manually check it by hand.