# **Berg-Verhelst-Compiler Phase 4**

“Semantic Analyzer”

## Introduction

This submission is for the Semantic Analyzer of the compiler project. Included is the code for the semantic analyzer, parser and scanner, the executable jar and supporting files, test files and this document. The semantic analyzer takes the AST produced by the parser, traverses it, annotates it and validates that the semantic rules are not violated.

Participation

The new addition to this phase is SemAnalyzer.java which the member who created the method is included in the method comment as well if the other member made changes to that method. FFSet.java was also added for correcting our parser. The split of work on the methods for this phase was evenly split between the members.

## Project Status

For this phase we implemented and tested a Semantic Analyzer that traverses an AST, annotates it, and ensures that the semantic rules of the C\*13 language are not validated. The annotation of the AST involves storing references to variable and function declarations in variable nodes and call nodes. This ensures that each variable and call node is linked to its proper scope. The semantic analyzer also type checks the AST to ensure that there are no type mismatches. The semantic analyzer also applies the rules of the C\*13 language to the AST to ensure there are no violations. If the semantic analyzer finds an error it reports the error, attempts a recovery and continues its analysis.

In the previous phases our folder structure was split into many packages, we have removed the packages and consolidated all parts of the compiler into a single “Compiler” package. We removed classes that deal with unit testing as they are beyond the scope of the project. We have also added an FFSet.java to simplify how the parser handles first sets and how it generates synch sets.

**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

+ Needs Actual Basic Parser

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 (Current Submission: 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 traveral, full traversal) completed

-Scope analysis completed

-Type checking completed

-Semantic rule checks completed

+Error printing needs line numbers

+Some error messages need to be more helpful

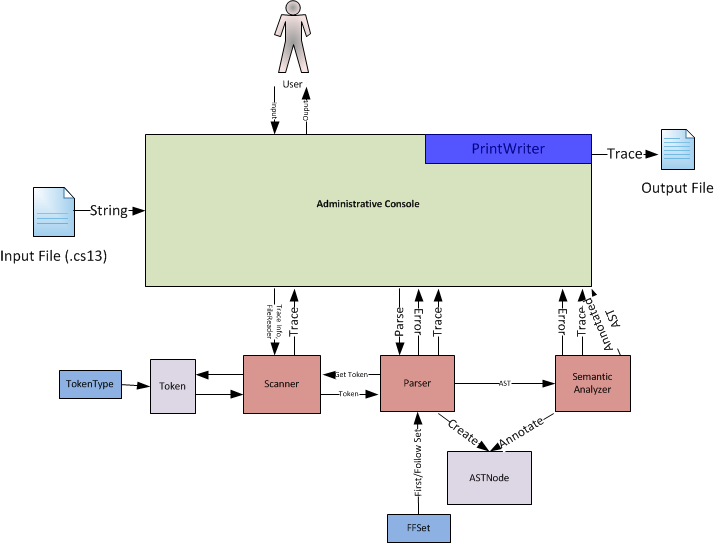
+Loop semantic rule (Exit cannot be in a compound statement) needs to be modified

Code Generator: Not Started

**Phase 5: Code Generator (To Be Submitted)**

-Not started

## Architecture and Design



This phase builds on the architecture of the previous phase. The Administrative Console operates in the same manner as it sets the environment based off of the supplied arguments, initiates a Scanner and a Parser and then executes the Parser’s parse() method. The parse() method returns the root node of the AST generated by the parser. This root node is then passed to an new instance of our Semantic Analyzer which recursively traverses the AST to annotate it and perform scope, type and semantic checks.

**Initial and Full Traversals**

The semantic analyzer runs two traversals of the AST tree. The first traversal is the *init* traversal which gathers all the global declarations in the program, that is, the global variables and the function names. It needs to do this because in C\*13 mutually recursive functions are valid and because C\*13 does not have forward declarations. The analyzer needs to know what functions are available when it checks for double declarations and when it checks if an identifier has been declared. C\*13 can get away with simply grabbing the global functions since nested functions (declaring a function in a function) is not allowed.

The full traversal uses the recursive structure of the AST to fully explore the AST using left-recursive descent. The full traversal uses a stack to represent the current scope at any point of the tree. The full traversal will annotate variable nodes and call nodes of the AST with references to the declarations of the variables and functions. The declarations these annotations point to are the most local declarations in the scope of the program at that point. The full traversal also does type checking and semantic rule validation as described below.

**Scope Checks**

The semantic analyzer represents the scope of a point of the program using a stack. The analyzer will push a new scope scope when it starts analyzing the AST (this is the global scope), when it finds a function declaration and when it finds a compound statement. Any declarations within a scope are pushed to that scope’s declaration list. Upon finding a variable or a function call the analyzer checks the scope to see if a declaration for the found identifier exists. If it exists it stores a reference in the respective ASTNode, if it doesn’t exist it displayed an”undeclared identifier” error, pushes a declaration of type UNI (universal type) of that identifier onto the local stack and continues. The analyzer pops a scope upon exiting a compound statement or exiting a function declaration.

Since the references from variables and calls to their respective declarations are stored in their respective ASTNodes, the scope stack does not persist beyond the semantic analysis phase.

**Type Checking**

When the semantic analyzer encounters an assignment, an array index or a binary operator, it validates the type. For an assignment it ensures the the type of the left hand side of the := operator is the same type as the right hand. For an array index it ensures that the index can be statically computed at compile time, that is, the index consists only of a number or of numbers and arithmetic operators. For a binary operator the semantic analyzer ensures that the left hand side of the operator has the same type as the right hand side of the operator.

When the semantic analyzer also type checks functions. It ensures the function returns an object of the type it declares it is returning. So for non-void functions it must return the type it specifies; void functions must have a return statement with no expression. Call statements are checked to ensure they are being used according to their function’s return type. Call statements are checked to ensure that they have the correct number and types of variables. If a function has a variable passed by reference the semantic analyzer ensures that all calls to that function are given a variable in that parameter field as only variables are valid since they reference a memory address.

**Semantic Rules**

The semantic analyzer checks the following semantic rules and prints an error if any are violated.

* Identifiers are not declared multiple times in the same scope
* Identifiers are declared before use
* Function calls are consistent with their definitions
* Assignments, calls, array indices and binary operators use types correctly
* The expression in an if statements is a boolean expression
* int main(void) **is** the last function declaration
* the four special functions are inserted by the analyzer and not redeclared
* *exit* and *continue* statements only exist within loops
* *exit* keyword must be present in a loop

**Test Programs List**

Correct: test.cs13

Incorrect: violation.txt

Implementation

For implementation we chose not to put the semantic analysis into the ASTNodes themselves. This is because we wanted the semantic analyzer to be its own separate entity so that any changes to the rules of the semantic analyzer do not affect the ASTNodes themselves.

One semantic rule has not been implemented fully. The semantic rule for the marker keyword ‘exit’ suggests that there should be a warning when ‘exit’ is found outside a loop, we made this an error. Also, the rule states that the ‘exit’ keyword must be in the immediate body of the loop, that is, it may not be contained in a nested scope (compound statement with variable declarations) in the loop. We do not have this check, instead we ensure that the ‘exit’ token exists in anywhere in the loop body, including the nested scopes.

## 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 runBasic” to run the basic parser on the test files. These files are: test/test.cs13 and test/program.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. These output files appear in the berg-verhelst-parser-basic\test\output folder when “sh runTest” 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: AdministrativeConsole

-compile Process all phases and compile (Default)

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

-help Displays Help Menu

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

-parse Process up to parser and print Lexical and Parser Phase and error

-q Only display error messages (Default)

-scan Process up to Scanner and print Lexical Phase and errors

-v Display all Trace Messages

-dev Prints out the “entering method:” and “leaving method:” parser trace

**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.

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.

**Observations**

All phases of the project have been tested using the files listed above and the ones previously received for the scanner phase and these test cases are as exhaustive as we could come up with for the semantic analyzer. There was a challenge in creating tests for the semantic analyzer as it required a firm grasp of the C\*13 language and knowledge of the semantics of the language. The violation test file is exhaustive.