# **Berg-Verhelst-Compiler Phase 3**

“Parser Phase with Error Recovery”

## Introduction

This submission is for the Parser Phase with Error Recovery of the compiler project. Included is the code for the parser and scanner, the executable jar and supporting files, test files and this document. The Parser checks syntax by following the derivation productions and generates an ASTNode. If an error is encountered the error recovery method ‘Panic Mode with Synchronized Sets’ is used.

Participation

The new additions to this phase came in the form of Parser.java and ASTNode.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. To add error recovery we divided the work on the methods to the one who created them, this involved minor changes and adding the synch sets. 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 Parser and allowed the generation and printing of an Abstract Syntax Tree (AST). The parser takes tokens and checks them against productions rules to ensures the Syntax of the program is correct. As the parser checks it generates an AST which can be printed. This tree contains nodes which groups the tokens and productions rules into a more compact form, as originally there were 37 productions rules, which have been replace by 17 AST Nodes with 2 groupings (Interfaces).

The Parser now uses the ‘Panic Mode with Synchronized (stop) Sets’ error recovery method. When an error is found, the error recovery method will discard all tokens until it finds a token that exists in the synch set of the calling method. The synch set is passed through all methods and each method adds its own synch set to the set being passed in.

The previous parser phase has been modified to add panic mode using synchronizing sets. Additionally the development messages have been turned off by default, these displayed when entering and leaving a method.

Scanner:

-Completed

Parser Basic:

-Left-most derivation completed

-Creating the parse tree completed

-Creating ASTNodes completed

-Trace is available through the parser

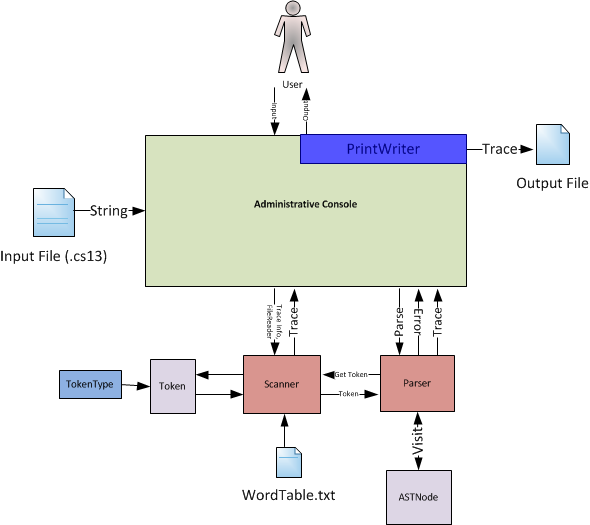
Parser Full:

-Error Recovery: Panic mode with Synch Sets completed

Semantic Analyzer: Not Started

Code Generator: Not Started

## 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 Parser’s parse() has changed since the last phase. To parse a program it grabs the first token of the source files and begins to check syntax through recursive productions. This syntax checking all boils down to matching the the current node that has been returned from the Scanner to the expected node defined in the first sets of the productions. If there is an unexpected token the parser reports an error and the compiler will print ‘FAIL’ once the parser completes its operation.

The Parser uses a the recursive structure of the productions to call the visit() method to generate Abstract Syntax Tree (AST). The visit() method simplifies the trace of the Parser. The AST nodes are used to represent the constructs of the program. There are 19 types of ASTNodes which are categorically grouped into ASTNode (all nodes included), Statements and Expressions. ASTNodes contain information about the construct they represent, for example, the IfStmtNode (which is a Statement and an ASTnode) will contain an Expression, a Statement that represents the ‘then’ component and an optional Statement representing an ‘else’ component. Since the nodes are linked together, the AST can be traversed using simple recursion.

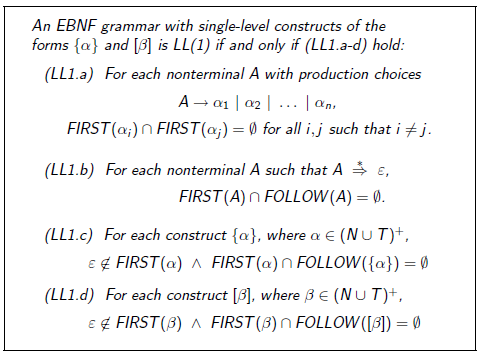
**Test Programs List**

Correct: test.cs13, program.cs13

Incorrect: masterTest.cs13, identifiers.cs13

**Production Rules And Comments**

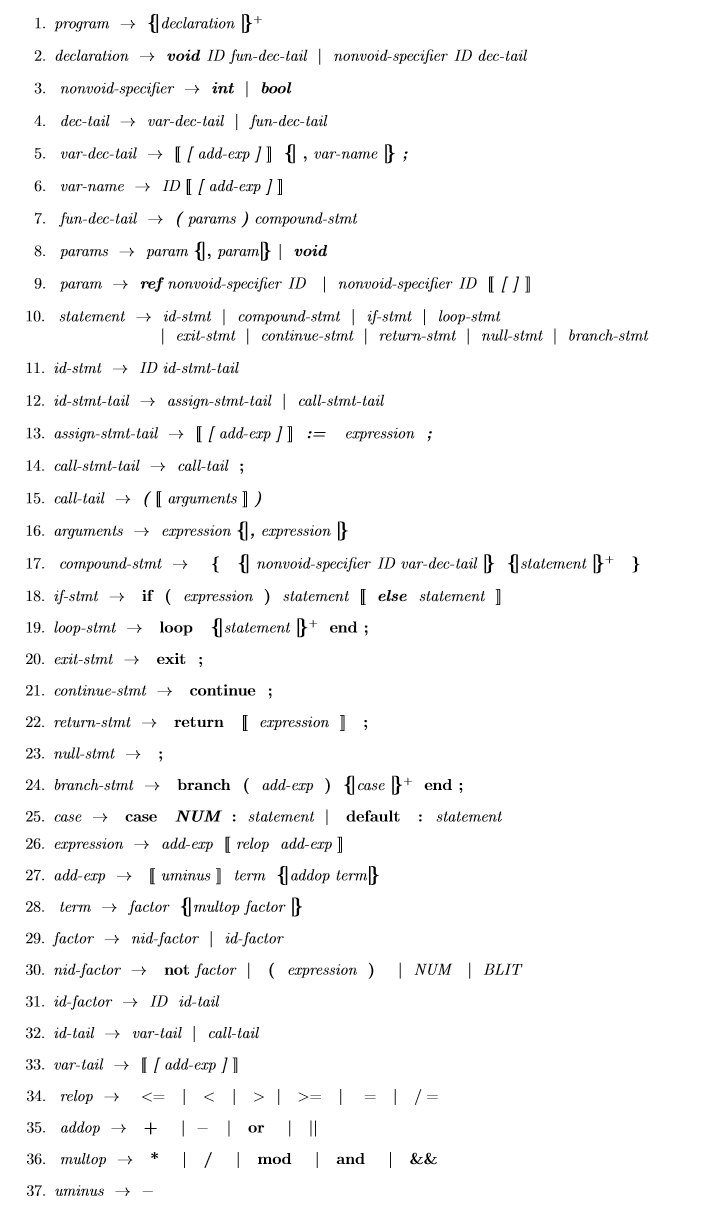
The following page contains the LL(1) EBNF grammar used to create the parser which was supplied by Jernej for this phase. Any EBNF grammar is EBNF if it holds to the following four specifications:



Since none of the first sets of the for any option of productions that go to multiple non-terminals intersect because of the left side factoring of the rules, then LL1.a holds. The first and follow sets for all nonterminals are disjoint, making LL1.b hold. LL1.c and LL1.d are assumed to hold.

**The LL(1) Grammar**

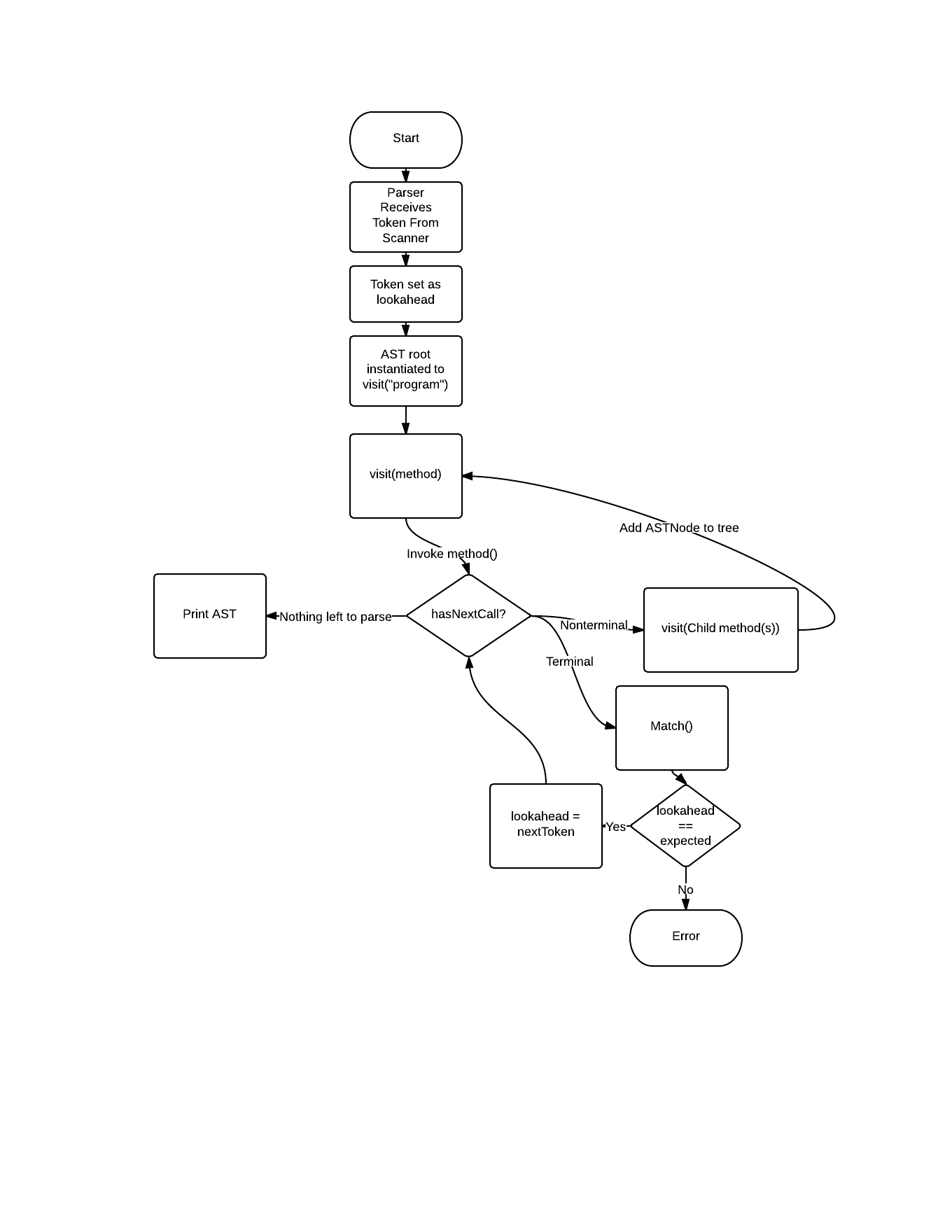
The grammar used to make a leftmost derivation is shown below. This grammar has the dangling else ambiguity. The disambiguation rule used to resolve the dangling else ambiguity is that an ‘else’ matches with the closest previous ‘if’.



Deviations in Parser Routines

Case Stmt has “NUM” and “COLON” added to its synch set for the **case NUM :** *statement* option.

AST Generation



Implementation

There are no notes for implementations issues for this phase.

## 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. 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.

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 all and print Lexical and Parser Phase and error

-q Only display error messages (Default)

-scan Process all 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

java -jar BergVerhelstCompiler.jar -v ../output-test/output.cs13 -o Out.txt

Will verbosely run the parser and save the output to out.txt.

## Code

Format: <packagename>/<filename.java>

**FileIO/FileReader.java**

-FileReader takes a filename and provides a method to read in that file to a string.

**FileIO/Writer.java**

-Writer writes lines to a file. The file is either the default file (ouput.txt) or it is supplied in the class’s constructor. If the file already exists, the output is appended.

**Lexeme/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).

**Lexeme/TokenType.java**

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

**Lexeme/TNSet.java**

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

**Main/Main.java**

-Main instantiates an administrative console and executes the compiler

**Main/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.

**Parser/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.

**Parser/ASTNode.java**

-The ASTNode class has inner classes for the different Node types, as wells as 2 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: Statement, Expression

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

**Scanner/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.

**UnitTests/UnitTester.java**

-The UnitTester run all the tests of all the respective unit test classes.

**UnitTests/UTResult.java**

-The UTResult class houses a result from a unit test, it holds the method name and the true/false value from the test.

**UnitTests/<otherclassname>Test.java**

-Classes following this format test the methods of the class referenced by this class’s name.

## Tests and Observations

**Main Scanner Test Files**

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

**prereq/cs13test.cs13**

- This is our file we used while creating the program to test random features, and have sample code chucks.

**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.

**unit/adminconsoletest.cs13**

-This class was created to test the administrative console.

**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

**Support Test Files**

These files were used to verify the output of the selected methods for the Scanner unit tests.

unit/scannerToken.target (Scanner.getToken)

unit/scannerCharacter.target (Scanner.isCharacter)

unit/scannerInvisible.target (Scanner.isInvisible)

unit/scannerNumeric.target (Scanner.isNumeric)

unit/scannerSimpleChar.target (Scanner.isSimpleChar)

unit/scannerSimpleSymbol.target (Scanner.isSimpleSymbol)

unit/scannerSymbol.target (Scanner.isSymbol)

unit/scannerWhiteSpace.target (Scanner.isWhiteSpace)

**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 individual functions and production rules. There was a challenge in creating tests for the parser as it required a firm grasp of the cs13 language. The broken test file is exhaustive. The unit tests currently only exist up to the scanner phase.