# **BergVerhelstCompiler Phase 1**

“Scanner Phase”

## Introduction

This submission is the Scanner Phase of the compiler project. We are submitting the code for the compiler, the executable jar and supporting files, test files and this document. The Scanner phase simply takes a character stream and produces tokens.

## Participation

Participation in this phase was split equally among the members of the group. Both members designed the central class, the Scanner, together. Leon created the original structure of the class and Emery implemented the majority of the original framework of this class. Nevertheless, we were both heavily involved in revisions, testing, improving and finalizing the code as the specifications were defined and redefined. We also both coded the supporting libraries with Leon taking the lead in that area of this phase.

Project Status

For this phase we have fully implemented and tested a Scanner class and Token class as well as supporting classes. The scanner will take lexemes, classify them, and produce tokens out of them.The administrative console has been developed insofar as the specification require.

Since this is the first phase of the project, there were no previous phases that could be modified.

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

## Architecture and Design

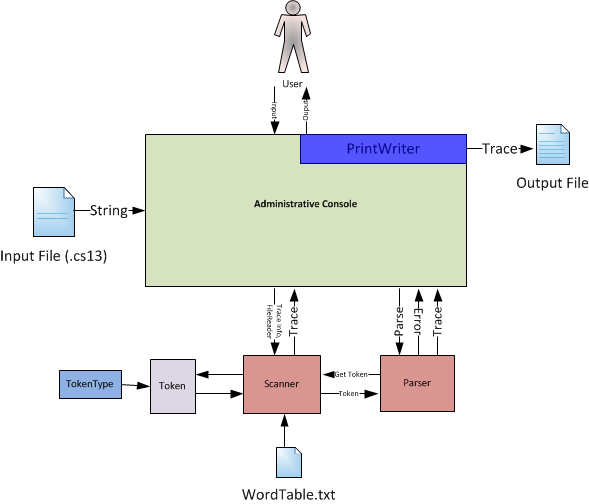
The approach we took was to build the compiler from the bottom up, that is we started with the most granular item of the compiler, the Token. Once the token structure was implemented we coded the Scanner and the logic that went with it, from there we built the supporting components, like the Administrative Console, and the file I/O components.

Execution begins in the compiler with the creation of an Administrative Console (AC). The AC is provided with the array of arguments passed through the command line. The console uses the Common-CLI-1.2 library to parse these arguments. Using these arguments the AC sets up the environment for the compiler. Once the environment is set up the AC creates a print writer to use for the trace of a file. The print writer defaults to System.out, but it can be specified to be an output file. The AC will generate, setup the Scanner and the Parser and execute the Parser.

The Parse takes a scanner and runs the Scanner in a while loop until the Scanner returns an EOF.

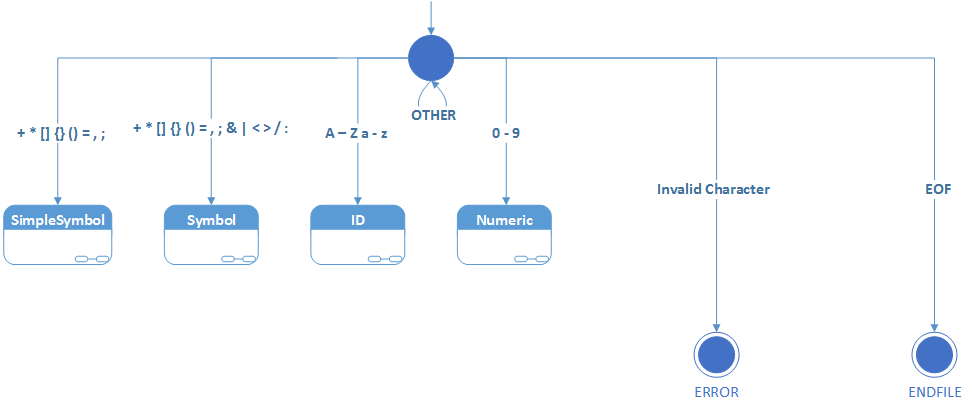
The Scanner is provided with an input file, which it reads into a string. The string is read character by character in order to generate Tokens using the Scanner’s getToken() method. The scanner will decide on the type of a token by peeking at the next character and then executing a method in relation to its decision. This process is visually shown in the state machines.

Tokens are the result of the Scanner’s getToken method. A token has a TokenType which is one of the items in the TokenType enum. Tokens also have an attribute value and a lexeme. For IDs and ID-like tokens the constructor will check if the lexeme exists in the symbol table and then place the index of the token (if it exists) into the attribute value of the lexeme.

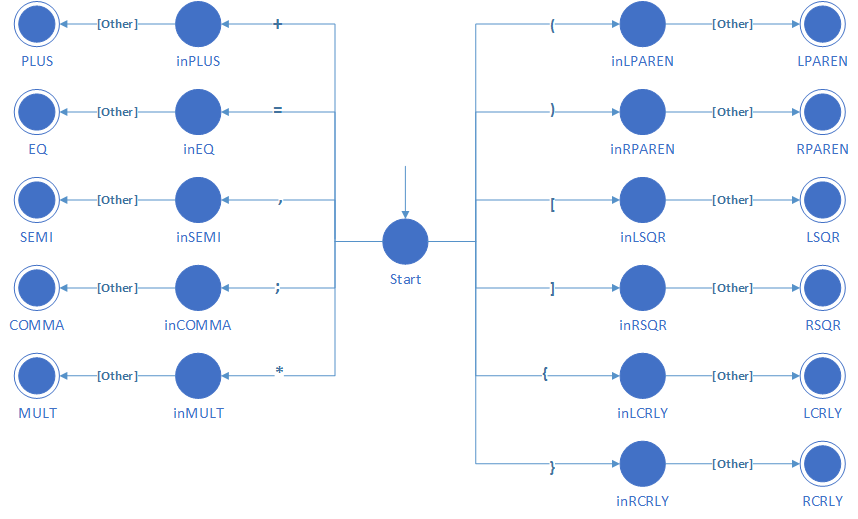


State Machine Diagrams

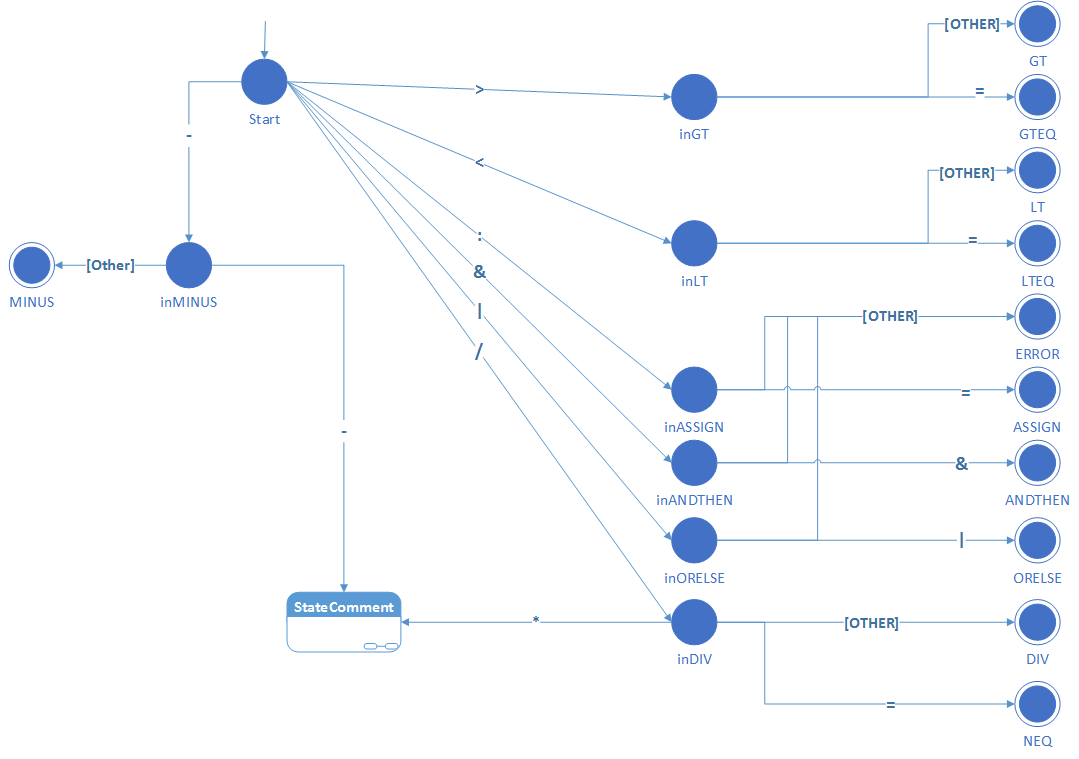
Main



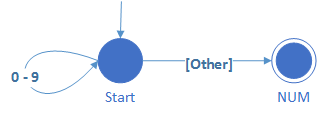
Simple Symbol

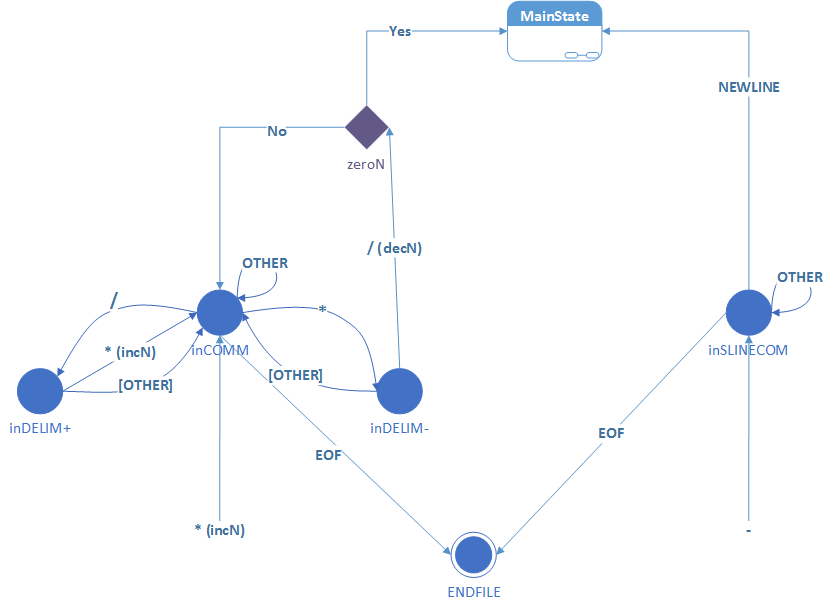


Symbol

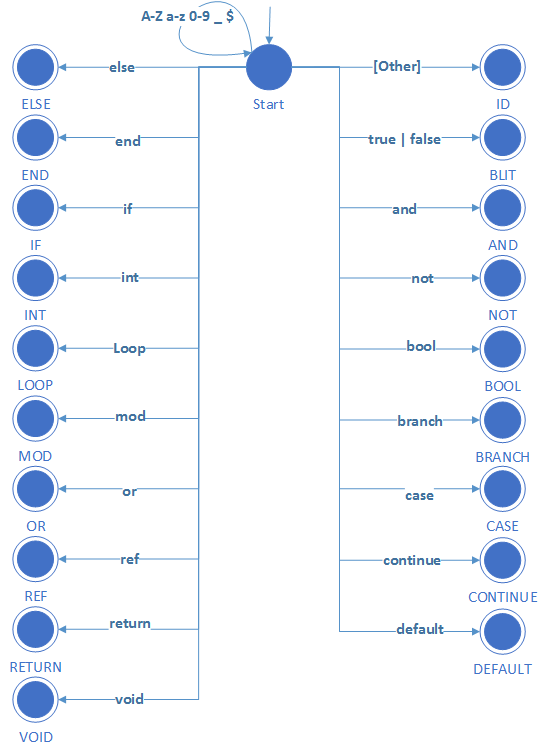


Numeric



Comment

ID



Implementation

Due to the way we handle nested comments, it is possible for a stack overflow error to occur if there are an inordinate amount (millions) of subsequent nested comments.

## 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. At this point you can enter “sh buildjar” into the terminal to compile the project into a jar file. We have specified the verbose option in order that the user may see the progress of this shell script. You can use “sh runTest” to use the program on the provided test files as well as our own. These files are: numerials.cs13, identifiers.cs13, other.cs13, nested.cs13, masterTest.cs13. Each run of the compiler will produce an output file named <inputfile>Output.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.

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 built using the “buildjar” shell file and the most recent version of the jar has been included.

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))

-q Only display error messages (Default)

-scan Process all and print Lexical Phase and errors

-v Display all Trace Messages

**Examples:**

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

Will run the scanner 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 scanner 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 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**

-TokenType is an enumeration of all the possible token types for the c\*13 language.

**Main/Main.java**

-Main instantiates an administrative console and executes the compiler

**Main/AdministrativeConsole.java**

-The administrative console sets the compiler state based on the arguments provided to the compiler. The administrative console creates and initializes each phase of the compilers and provides them with trace options.

**Parser/Parser.java**

-The parser simply runs a loop to retrieve tokens from the scanner

**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. Scanner holds the input file’s contents and is the class that is queried for the input file’s next character.

**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 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.

**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)

For testing we chose to test the output of the component being tested against the expected result. For simplicity and scalability we chose to keep the expected outputs in their respective files. The test cases are as exhaustive as we could come up with for the individual functions. The unit tests complete successfully and the compiler operates as expected.

Additionally the supplied test and our mastertest.cs13 have been used and give the expected output when passed through our scanner.