|  |
| --- |
|  |
| Berg-Verhelst Compiler |
| CPSC 425 F13 C\*13 Compiler |
|  |
| **Emery Berg & Leon Verhelst** |
| **11/27/2013** |

CPSC 425 Fall 2013

Professor: Jernej Polajnar

|  |
| --- |
|  |

Berg-Verhelst Compiler Report

Contents

[Introduction 1](#_Toc373340457)

[Participation 1](#_Toc373340458)

[Project Status 2](#_Toc373340459)

[Change log 3](#_Toc373340460)

[Architecture and Design 7](#_Toc373340461)

[Overview 7](#_Toc373340462)

[Scanner 8](#_Toc373340463)

[State Machine Diagrams 9](#_Toc373340464)

[Parser 13](#_Toc373340467)

[Production Rules and Comments 13](#_Toc373340468)

[The LL(1) Grammar 14](#_Toc373340469)

[Deviations in Parser Routines 15](#_Toc373340470)

[AST Generation 15](#_Toc373340471)

[Parser: Error Recovery 16](#_Toc373340472)

[Semantic Analysis 16](#_Toc373340473)

[Initial and Full Traversals 16](#_Toc373340474)

[Scope Checks 16](#_Toc373340475)

[Type Checking 17](#_Toc373340476)

[Semantic Rules 17](#_Toc373340477)

[Code Generation 18](#_Toc373340478)

[Traversal of Annotated AST 18](#_Toc373340479)

[Computation of Array Indexes 18](#_Toc373340480)

[Finding Local Sizes 18](#_Toc373340481)

[Finding Level and Displacement 19](#_Toc373340482)

[Implementation 19](#_Toc373340483)

[Building and Use 20](#_Toc373340484)

[Command Line Arguments 20](#_Toc373340485)

[Code 21](#_Toc373340486)

[Tests and Observations 23](#_Toc373340487)

[Main Scanner Test Files 23](#_Toc373340488)

[Main Parser Test Files 23](#_Toc373340489)

[Main Semantic Analyzer Test Files 23](#_Toc373340490)

[Main Code Generation Test Files 23](#_Toc373340491)

[Observations 24](#_Toc373340492)

[What we learned 25](#_Toc373340493)

[Table of Figures 25](#_Toc373340494)

# Introduction

This report covers the completed Berg-Verhelst compiler which was produced for CPSC 425 in the Fall Semester of the 2013/14 academic year. The Berg-Verhelst-Compiler is a compiler that translates source code written in the *C\*13* programming language into target code in the Quadruples language which is three-address code. The output code is meant to be executed on the interpreter supplied for the CPSC 425 F13 class, but due to errors in the interpreter, the generated code only works if the level does not exceed nesting level of a called function. The compiler has four major components, the scanner, the parser, the semantic analyzer and the intermediate code generator. The scanner performs lexical analysis on the source code and converts the character stream into useful tokens, the parser takes the produced stream of tokens and performs syntax analysis to build an Abstract Syntax Tree (AST) that represents the syntactic structure of the source program according to the production rules of the C\*13 grammar. The semantic analyzer then performs semantic analysis on the AST to ensure the semantics of the C\*13 language are satisfied and annotates the AST accordingly. Finally the intermediate code generator takes the annotated AST from the semantic analyzer and generates three address intermediate code in the form of Quadruples.

This submission includes the entire source code for the compiler, an executable jar, supporting files, test files and this document.

# Participation

This project required an even split in the workload in order to balance the available time and resources of both members. Each phase began with a, sometimes lengthy, discussion on ways to tackle the component. Once the two of us agreed on the strategy, we would outline the code and split it up by method; this is reflected in the comments for individual methods in the code. Code written by one member would often be checked over and improved by the other member. Amendments and revisions to previous phases were also split evenly between both members of the team. This ensured that each member worked on and understood the core concepts and algorithms of each component of the compiler. Overall, the work in the project was split evenly among both members of the team.

# Project Status

In this submission we present a complete compiler for the C\*13 language. Every component of the compiler has been fully completed and has been exhaustively tested to the best of both member’s abilities and knowledge. The following itemized list briefly describes the status of each component:

**Scanner**

* Fully implemented and tested
* Produces Tokens
* Classes: Scanner.java, Token.java, TokenType.java

**Parser**

* Fully implemented and tested
* Produces AST
* Classes: Parser, ASTNode, TNSet, FFSet
* 17 ASTNode inheriting private classes, 3 logical groupings (represented by interfaces)
* Panic Mode with Synchronized Sets Error Recovery

**Semantic Analyzer**

* Fully implemented and tested
* Produces Annotated AST
* Classes: SemAnalyzer

**Code Generator**

* Fully Implemented
* Produces Quadruples
* **Note:** Faulty interpreter did not allow for exhaustive testing of generated code. Generated quadruples are valid according to manual checking
* Classes: CodeGen

**All Components:**

* **Note:** All components have their environment set up by the Administrative Console, which also is used to execute each component and pass data between the components.
* Classes: AdministrativeConsole

## Change log

**Current Version [Nov 27, 2013]**

Scanner:

* Completed

Parser Basic:

* Completed

Parser Full:

* Completed

Semantic Analyzer:

* Completed

Code Generator:

* Completed
* Not Operator code generation added
* (Level, Displacement) code generation added

**Phase 5: Code Generator [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 for the error handling of redeclarations of variables within the same scope
* 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
* Binary Operations completed
* Flow Control completed
* Structure completed
* Assignments involving array elements completed
* Unary Operations (lacks NOT function)
* (Level, Displacement) representation of variables is not implemented.
* Temporary Variables are not reused

**Phase 4: Semantic Analyzer [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 3: Parser with Error Recovery [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 2: Basic Parser [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 1: Scanner [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

# Architecture and Design

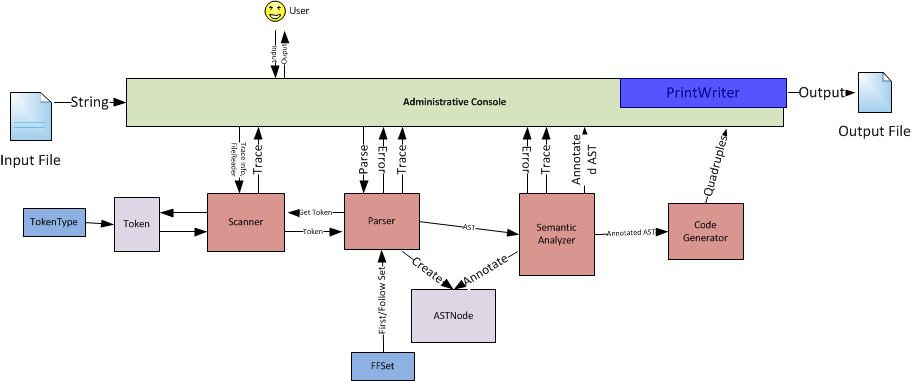


Figure 1-Compiler Architecture

## Overview

The Administrative Console is used to take input, generate output and manage all components and passes of the compiler. If a pass fails the Console stops execution of the next pass, in order to prevent unnecessary error messages. The first pass of the compiler covers the lexical and syntactical analysis of the code given to the compiler. The lexical pass uses the Scanner object to generate tokens which are then passed to the Parser object to ensure they are syntactically valid tokens. If an error is found the parser attempts to recover using panic mode recovery with synchronized sets and if no error is found the parser generates an Abstract Syntax Tree (AST). Once generated the AST is then passed by the administrative console to the Semantic Analyzer, which scans the AST to ensure the semantics rules of the language are followed. While scanning the AST the node contained within are annotated to include additional information for the next pass, which is generation of code. The Administrative Console provides the Code Generator with the annotated AST from the Semantic Analyzer as input and creates quadruple code from it. This is done by traversing the annotated AST and creating the quadruples which use level and displacement instead of the notation given by the other passes of the compiler.

## Scanner

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.

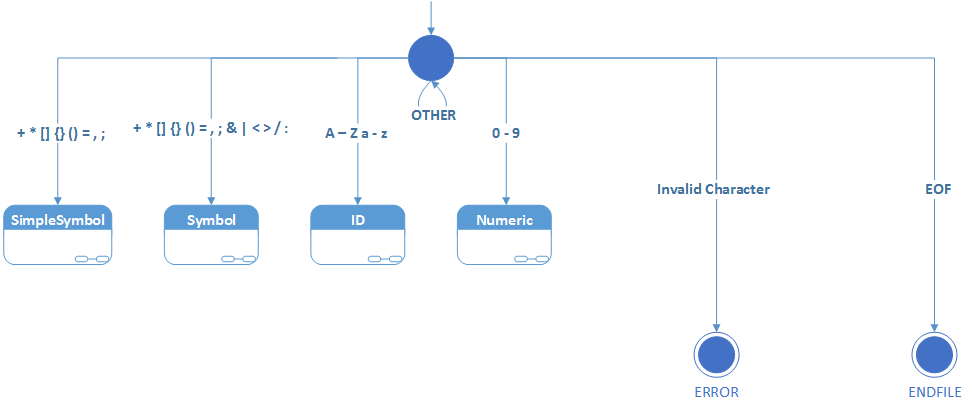
Execution begins in the compiler with the creation of an Administrative Console (AC). The AC is provided with the array of arguments passed through the command line. The console uses the Common-CLI-1.2 library to parse these arguments. Using these arguments the AC sets up the environment for the compiler. The setup of the environment includes setting each component of the compiler’s printing options, there is a quiet option that will only print errors while a verbose option specifies to display what the compiler is doing, the compiler can also be provided with a hidden parameter (-dev) that will show additional information for certain components, such as the Parser. Once the environment is set up the AC creates a printwriter to use for the trace of a file. The print writer defaults to Java’s System.out, but it can be specified to be an output file. The AC will generate and set the environment for all components of the compiler and being execution by calling the Parser (which in turn calls the Scanner). Since the Parser and the Scanner are both used for the first pass of the compiler, they are sometimes viewed as a single component. The execution of the compiler will continue by serially executing the next component of the compiler.

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.

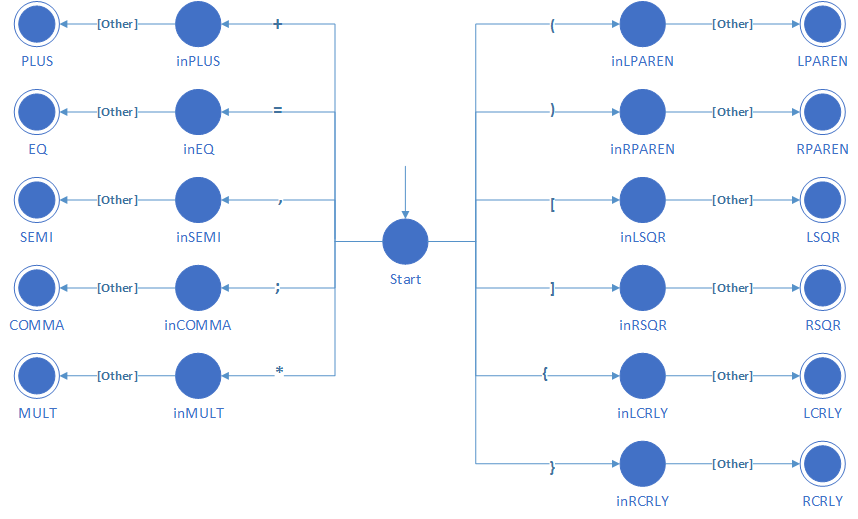
Tokens are the result of the Scanner’s getToken() method. A token has a TokenType which is one of the items in the TokenType enumeration. Tokens also have an attribute value and a lexeme. For IDs and ID-like tokens the constructor will check if the lexeme exists in the symbol table and then place the index of the token (if it exists) into the attribute value of the token which is denoted by the lexeme. If the lexeme does not exist in the symbol table the lexeme is added to the symbol table and the newly created index for that lexeme will be put into the attribute value of the token corresponding to that lexeme.

The state machine diagrams shown below describe how the Scanner classifies sets of characters into their respective Token.

## State Machine Diagrams



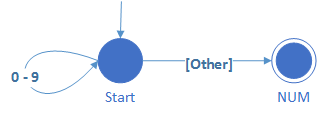
**Figure 2- Main State Machine**: Used to control the follow of the scanner class



**Figure 3- Simple Symbol State Machine:** Identifies simple character symbols

### 

**Figure 4- Symbol State Machine:** Used when dealing with more complex symbols where a look ahead is need to determine the final token.



**Figure 5- Numeric State Machine:** Identifies numbers until a character is no longer numeric

### 

**Figure 6- Comment State Machine:** Used to deal with single line comments and block comments which can be nested

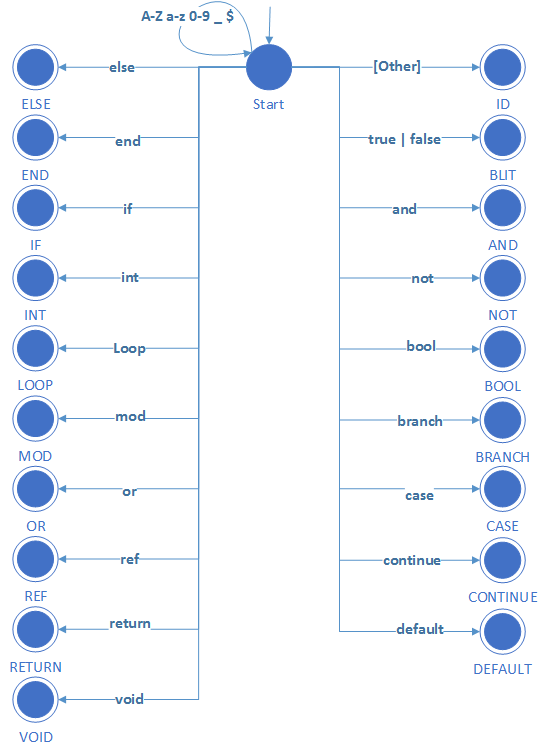


Figure 7-ID State Machine: Used when it has been determined the string is a identifier, this will check against keywords until a match cannot be found, in this case it is set as a basic identifier instead of a keyword. This will consume characters till an invalid character is found.

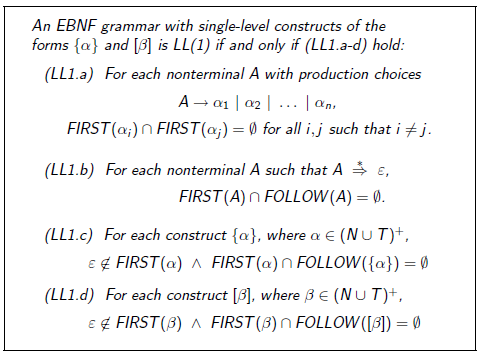
# Parser

To parse a program the Parser grabs the first token of the source file that the Scanner finds and begins to check syntax through recursive productions. This syntax checking all boils down to matching 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 theParser reports an error and the compiler will print ‘FAIL’ once the Parser executes its error recovery procedure and completes its operation.

The Parser uses 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), Declarations, 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 to represent the Boolean logic, a Statement that represents the ‘then’ component and an optional Statement representing an ‘else’ component. Since the nodes are composed of one another, the AST can be traversed using simple recursion.

## Production Rules and Comments

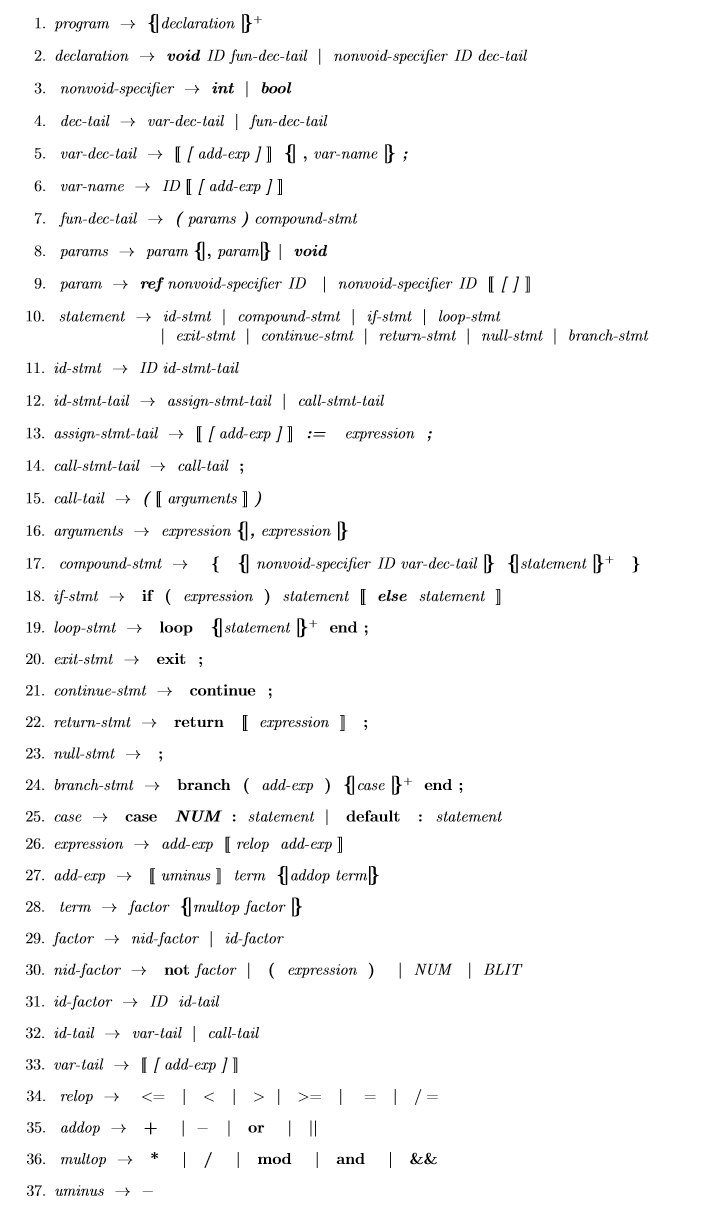
The following page contains the LL(1) EBNF grammar used to create the parser which was supplied by Jernej for this phase. Any EBNF grammar is EBNF if it holds to the following four specifications:



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 nonterminal 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

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

## AST Generation

The below figure visually describes the process by which the parser generates the abstract syntax tree. The diagram is noticeably missing the error recovery module, this is to make the diagram simple and understandable.

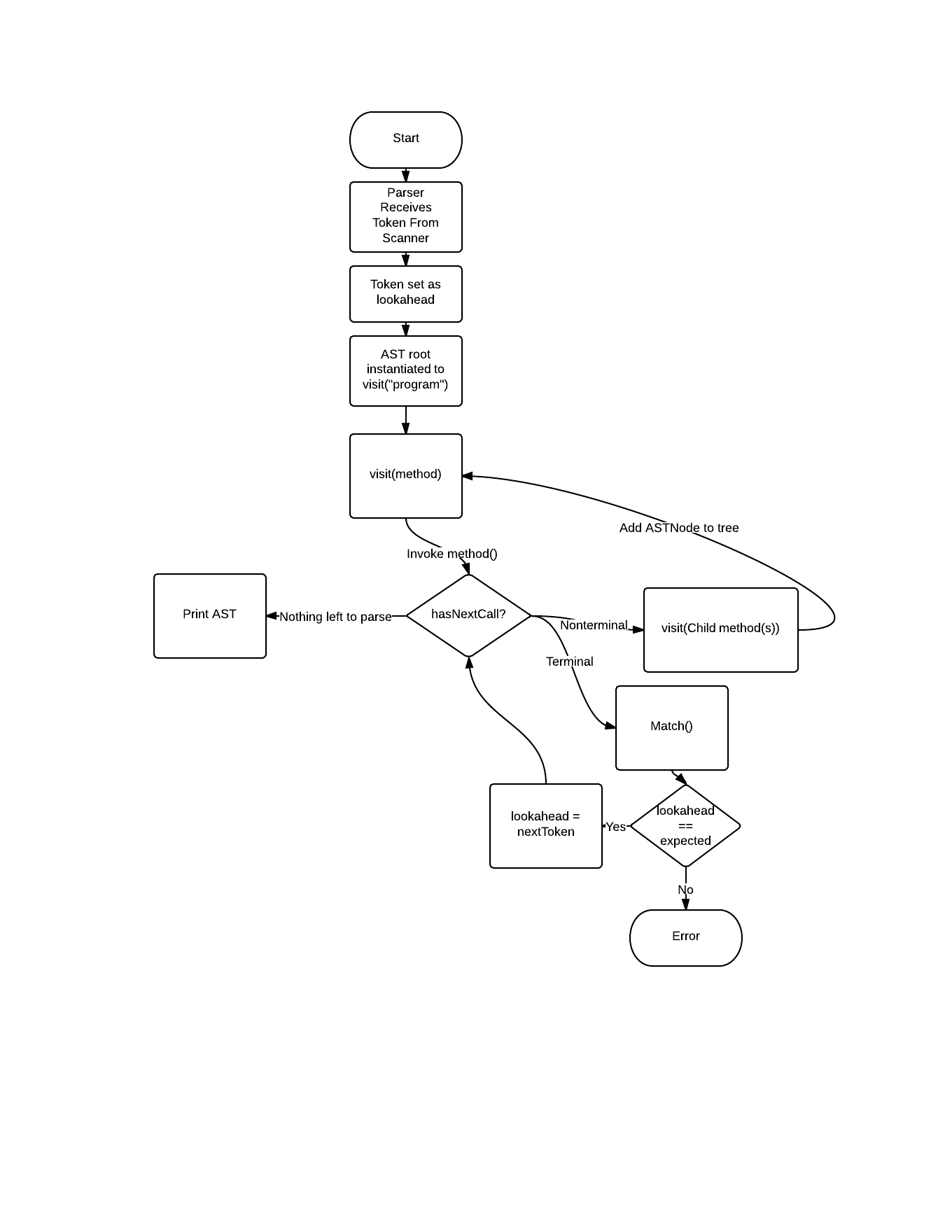


Figure 8: AST Generation Flow Chart

## Parser: Error Recovery

The parser uses panic-mode error recovery with synchronized sets. Synchronized sets or sync sets are sets of token types that can follow certain language constructs (ASTNodes).In the match() routine the parser checks to ensure that the lookahead matches with the token that the scanner has returned from the source. If the lookahead token type does not match the expected token type we initiate our error recovery method. The parser will display an error to the user at this point. The error recovery technique is quite simple, since we know that a token from the sync set is expected after we have parsed the current language construct we can consume tokens from the scanner until we find the a token that is contained in the sync set. From here the parser can resume normal operation.

# Semantic Analysis

The parse() method of the parser returns the root node of the generated AST. This root node is then passed to a 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 within 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 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 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
* Function calls call existing functions
* 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

# Code Generation

In order to execute the Code Generation phase the administrative console 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.

# Implementation

When implementing the semantic analyzer and the code generator we chose to separate the logic from the ASTNodes, this was done to ensure that changes of semantic rules and changes to code generation do not directly affect the ASTNodes, the compiler’s class structure also becomes clearer, with each component being encapsulated within their own class.

One of the semantic rules which was not fully implemented was checking if a variable had been initialized. A basic attempt as this was done which would check if at any point the variable had been initialized, but this did not take into account control structures such as if statements. This level of check would require the semantic analyzer to provide dynamic checking of the variable which is normally done at runtime.

The compiler modifies the semantic rule for the ‘exit’ statement with regards to the ‘loop’ structure of the C\*13 language. 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.

In the code generation phase we did not optimize the generation of temporary variables in that they are not reused. This was done to keep the code simple and understandable.

# 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’ 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
* 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

Throughout designing the different phases of the compiler, we used the process of creating test cases and then comparing them to the results we expected. These test cases were based on our knowledge of each phase and tested the phase to our understandings at the time. Some of the test cases were missed, so changes had to be made to previous phases in order to correct these issues. Early on the knowledge of the C\*13 language also provided challenges in the earlier phase when generating test code. Once we were able to better understand what was required the test cases became easier and generating code to test the later phases was easier.

The last phase of the compiler was difficult, as the interpreter we were supposed to use, did not compile correctly on the newer version of Ubuntu and would give a segmentation fault when a call to a function was made. The basic functionality was tested without level and displacement and by modifying the code to run at the global level to prevent addressing issues and the call crash. Once the modified interpreter was given, level and displacement was tested along with functions in the proper structure. This lead to the discovered that if while in a function a compound state is entered, any returns will cause the interpreter to back trace too far and in some cases cause the program to be stuck in an endless loop. Luckily this still allowed the majority of the functionality to be tested, while the remaining functionality was checked by hand.

# What we learned

One of the first mistakes we made during this project was planning on making the background language modifiable. We ended up removing this code as it was not practical and it caused dependencies on external files. Another area we made mistakes was our implementation of the Administrative Console, which made the Scanner dependent on it. This was removed to fix the recursive dependency which it created and allowed the scanner to work independently of the Administrative Console.

Our original implementation of synchronized sets for the Parser component used two hash maps of TNSets with string identifiers to store the first and follow sets of each of the language’s LL(1) constructs. This proved to be incredibly ugly when it came to applying the GEN(A, synch) rules to the LL(1) grammar. Our code suffered from the confusion of handling the synchronous sets in this ugly manner. This made it difficult to get the parser’s error recovery and syntax analysis working properly. The second mistake we made in this regard was making the union of two TNSets (which is used to join synch sets) modify the TNSets being joined. This made temporary TNSets to hold the original synch sets necessary, which only added to the confusion of handling synchronous sets. Eventually we sorted out the confusion by implementing the FFSet (First/FollowSet) class and using enumerations of TNsets to represent synchronous sets. We also modified the union operation to return a new instance of a TNSet to avoid unnecessary temporary variables.

We also learned new coding techniques which we had not known of previously, for example, the visitor pattern and the … parameter option. In addition we learned better techniques for structure and how to properly make coordinate member to string functions for pretty printing recursively defined data structures. We found static imports in Java to be very useful for directly referring to enumerated items without needing to specify their containing class.

# Table of Figures

[Figure 1-Compiler Architecture 7](#_Toc373339010)

[Figure 2- Main State Machine 9](#_Toc373339011)

[Figure 3- Simple Symbol State Machine 9](#_Toc373339012)

[Figure 4- Symbol State Machine. 10](#_Toc373339013)

[Figure 5- Numeric State Machine 10](#_Toc373339014)

[Figure 6- Comment State Machine 11](#_Toc373339015)

[Figure 7-ID State Machine. 12](#_Toc373339016)

[Figure 8: AST Generation Flow Chart 15](#_Toc373339017)