# CCAL Assignment 2 Report

CA4003: COMPILER CONSTRUCTION

ABSTRACT SYNTAX TREE
SEMANTIC ANALYSER
INTERMEDIATE CODE

Nigel Guven | 14493422 | 16th December 2019

nigel.guven2@mail.dcu.ie

Word Count: 2264

## Table of Contents

## 1. Assignment Objective

**Title:** Assignment 2: Semantic Analysis and Intermediate Representation for the CCAL language.

**Aim**

The aim of this assignment is to add semantic analysis checks and intermediate representation generation to the lexical and syntax analyser you have implemented for the CCAL language in the first assignment. The generated intermediate code should be a 3-address code and stored in a file with the '.ccl' extension.

You will need to extend your submission for Assignment 1 to:

- Generate an Abstract Syntax Tree.
- Add a Symbol Table that can handle scope.
- Perform a set of semantic checks.
- Generate an Intermediate Representation using 3-address code.

*"JJTree operates in one of two modes, simple and multi (for want of better terms). In simple mode each parse tree node is of concrete type SimpleNode; in multi-mode the type of the parse tree node is derived from the name of the node. If you don't provide implementations for the node classes JJTree will generate sample implementations based on SimpleNode for you. You can then modify the implementations to suit."*

**JavaCC: JJTree Reference Documentation**

## 2. The Contents of this Directory

This directory should contain the source files, the main of which is MyCCParser.jjt which is the grammar file to describe the Lexical and Syntax elements of the CCAL grammar and also handles the creation of JJT nodes which are placed into the symbol table. There are also some files generated by the parser which are shown below. And finally, the document of plagiarism and this specific document which is the assignment report.

- CCAL Definition
  This file provides information on the CCAL Language and was created by the examiner. Contained within is the grammar definition of the CCAL Language. It displays the set of all reserved words, tokens, literals in the first section. The second section describes the production rules of the grammar which define the syntax of the language. A list of semantic operators is provided with their order of precedence. A sample of program files are provided for testing the syntactic correctness of the JavaCC grammar file. These include testing empty statements, variables, scope and functions.

- Declaration on Plagiarism
  This statement is the required signed declaration of originality.

- MyCCParser.jjt
  This JJTree file contains the main grammar file specifications and is the most important file in this directory. It contains the parser class and lexical tokens which describe the source of the grammar. A set of grammar productions extend the program and create JJT nodes for each production which are added to the symbol table which stores each token in hash tables for utilization by the Semantic Analyser and Intermediate Code Generator.

- SymbolTable.java
  The symbol table is the complex data structure which is made up three hash tables containing identifiers and their associated type and attribute. The symbol table handles and also has a print function to retrieve all the identities contained within the program.

- DataType.java
  This file holds the definitions of the various types of data to be used by the Visitor classes. These types are held in an enumeration.

- SemanticAnalyser.java
  This visitor class will check an inputted language for semantic errors and outputs either a successful operation message or one or multiple error messages with the type of error that has occurred, and which item might be at fault.

- IntermediateCodeGenerator.java
  This visitor class generates three address code based on an input file that has passed syntactic and semantic checks. The intermediate representation is outputted to a file called 'intermediatecode.ir'

- A list of 27 extended Node classes for use by the Abstract Syntax Tree which define the data types of the CCAL language and various node types based on the JJTree syntax:

| | | |
|---|---|---|
| SimpleNode.java | ASTProgram.java | ASTVariable_Decl.java |
| ASTConstant_Decl.java | ASTTag.java | ASTConstant_Assignment.java |
| ASTFunction.java | ASTFunctionReturn.java | ASTType.java |
| ASTParameter_List.java | ASTNemp_Parameter_List.java | ASTMain.java |
| ASTStatement.java | ASTAssignment.java | ASTNumber.java |
| ASTBoolean.java | ASTAddition.java | ASTSubtraction.java |
| ASTEqualOperator.java | ASTNotEqualOperator.java | ASTGreaterOperator.java |
| ASTGreaterEqualOperator.java | ASTLessOperator.java | ASTLessEqualOperator.java |
| ASTOr.java | ASTAnd.java | ASTArgument_List.java |

  *For ease of clarity these above files will be excluded from the submission.*

- test.ccl
  This type '.ccl' document was used to test upon the grammar file and contains a sample program provided from the CCAL Definition document.

- intermediatecode.ir
  This file contains the generated three address code created by the Intermediate Code Generator visitor. This file should be able to be manipulated by other programs give that it is in correct 3 address code format although I have not tested this.

## 3. Parser Class

The parser class MyCCParser.jjt had to be extended to utilise JJTree functionality. The first steps were to add options that might be relevant for the purpose of this project. By completion, I had added three options which were JJTree specific.

```
//JavaCC Options

DEBUG_PARSER = false;
JAVA_UNICODE_ESCAPE = true;
OUTPUT_DIRECTORY = ".";

//JJTree Options

NODE_DEFAULT_VOID = true;
MULTI = true;
VISITOR = true;
```

'NODE_DEFAULT_VOID' essentially makes non decorated nodes void. Decorated nodes contain children with a certain number of children specified in the following parentheses of a given decorator. Indefinite nodes can lead to ambiguities in the grammar. By default, JJTree treats each nonterminal as an indefinite node and derives the name of the node from the name of its production.

'MULTI' allows for the generation of multi-mode parse trees.

'VISITOR' inserts jjtAccept methods into the node classes and generate visitor implementation with any entry for every node type used in the grammar to thereby extend a node. The intermediate code generator and semantic analyser are visitors which extend each AST node to include a function which either retrieves the node to compare it against semantic checks or generate an intermediate representation from it.

The Parser class is inside the JJTree file was extended from the first assignment to add the Symbol table call, semantic analysis and intermediate code generation. Also, there is a function which prints the generated 3 address code string to a file. Finally, there are appended necessary exceptions which may be relevant to each visitor.

The lexical section is unchanged.

It is at the grammar section where the most alterations occur. I removed some Lookaheads by inserting 'prime' tagged productions to improve performance. There still remains one Lookahead function in the grammar. I also created a new production for tag which holds the identification of a variable.

The most important task to do was to implement JJTree functionality over the grammar.

```
void CompOperator():
{
    Token t;
}
{
        t = <BOOL_EQUALS> Expression()
        {
            jjtThis.value = t.image;
        }
        #EqualOperator(2)
```

Each final production is given a decorator like the one shown above. As can be seen Here, 'EqualOperator' will be created into a node class which can be derived by other visitor classes. It contains 2 children which we may assume are a left and right comparison with the '==' in the middle. The Token class will save this as its image which is the value of the token when it encounters a '==' symbol.

```
void Nemp_Argument_List():
{}
{
    Identifier()
    [
        <COMMA> Nemp_Argument_List()
    ]
    #Argument_List(>1)
}
```

A decorator may also handle arity which may take in more than a certain number of children as can be seen above where the arity is greater than 1, whereby the argument list can hold one or more arguments.

```
type = Type() id = Identifier()
{
    symboltable.put(id, type, "func", scope);
    scope = id;
}
```

The symbol table is referenced from this production whereby it takes the variable item and handles scope by checking if it is in main or a local method.

## 4. Symbol Table

The symbol table is designed to hold the collection of identifiers, types and attributes in a given program. My symbol table is comprised of three hash tables which hold each of the above parameters. The symbol table must handle scope whereby an identity could be a global value or a local value.

```
/** Variables */

    Hashtable<String, LinkedList<String>> symboltable;
    Hashtable<String, String> attributes;
    Hashtable<String, String> types;
```

One of the hash tables contains a linked list. The reason for this is that the linked list contains a list of the variables contains in a certain scope, be it global or local. Local variables should be removed once a local function exits so if a variable is given global access via the term in the grammar productions, then they can be called by the visitor classes using the symbol table. There is an 'isInScope' function which is called by the semantic analyser specifically which checks the scope of an object.

There are two finder functions for type and attribute which are called by the print function. The print function outputs the tag, the type of its variable and its attribute as can be seen below.

```
**********AST Tree**********

ID      | Type          | Attribute
five      integer         const
result    integer         var
arg2      integer         var
arg1      integer         var
i         integer         var
x         boolean         function_parameter
getx      integer         func

**********AST Tree**********
```

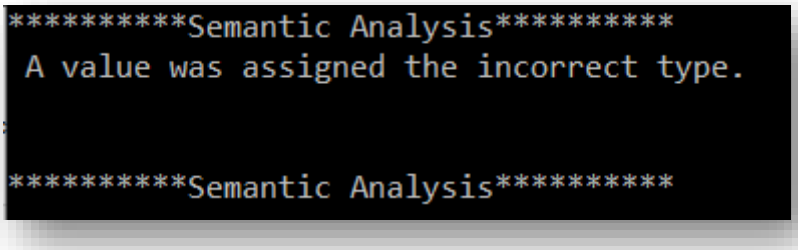
## 5. Semantic Analyser

Semantic Analysis is performed by a visitor class, the Semantic Analyser.java class to be specific. Where the syntax analysis checks for structure and was provided in the previous assignment, semantic analysis derives meaning behind the language and establishes a set of rules based on relating syntax structures. One cannot be utilised without the other in a fully operational programming language.

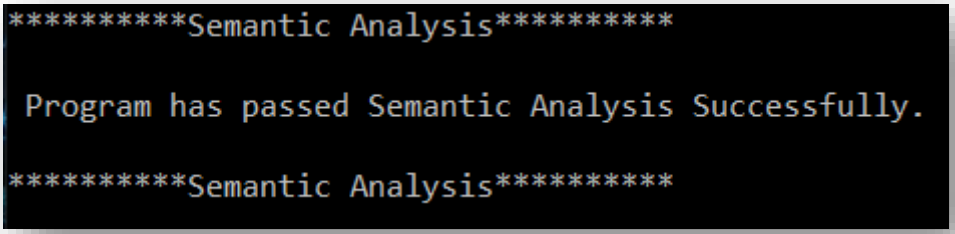
My syntax analyser examines scope, variable assignment and comparison between items of different types. The semantic analyser calls the symbol table once that has been derived from the parser. The semantic analyser checks each item through the use of its visit function which extend each node class. A 'semanticCheckFlag' variable is by default set to true and if one of 6 check methods cause an error it will set to false and with the relevant exception error and item in question being printed to the terminal.

Certain checks are whether an item is being referred out of its scope, whether two arithmetic or logical variables are being incorrectly compared to one another as well as a system for checking if a variable has been assigned the wrong value type. If the symbol table is semantically correct, a successful parse message will be outputted to the terminal.

An incorrect value  was discovered in this parse of a file.




This program parsed successfully without error.



## 6. Intermediate Code Generator

The goal of intermediate code generation is to translate a language of a certain paradigm and abstraction into another language that is also executable. For this task, we are translating the CCAL language, a rather simple but high-level language with basic functions, data types and object orientation construction. This language is translated into an intermediate representation which looks similar to assembly language except that it can be implemented from its representation into any high or low-level language without difficulty.

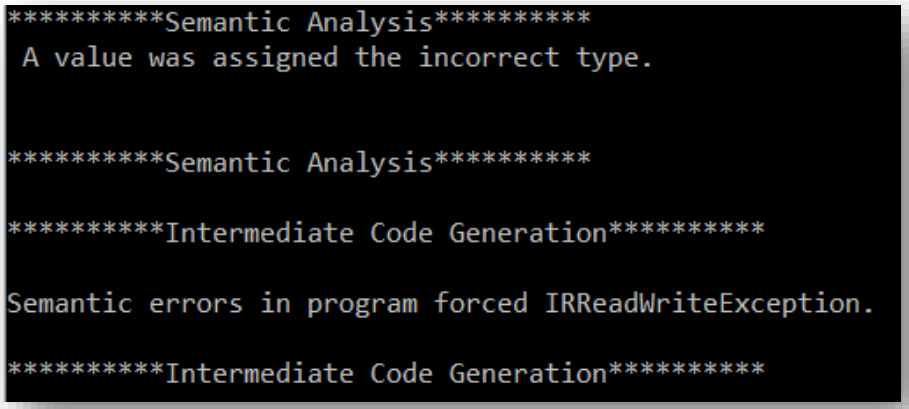
It essentially makes translation easier than having to translate straight from the likes of Java into python whereby the programmer has to understand the structure of both languages and can be quite difficult. Intermediate representations provide a clear instruction routine that can be translated into any language by most novice programmers. This was the goal I had in mind when creating this class.

The intermediate generator, like the semantic analysis visitor, extends classes for each node. There is a 'getInstruction' which retrieves an instruction and appends it to a string holding the list of translated intermediate code. When the code generation visitor ends its revision of the abstract syntax tree, it should contain this string and is then called by the 'MyCCParser.java' class which either outputs the intermediate code to the relevant file or else outputs an error based on the results from the semantic analysis.

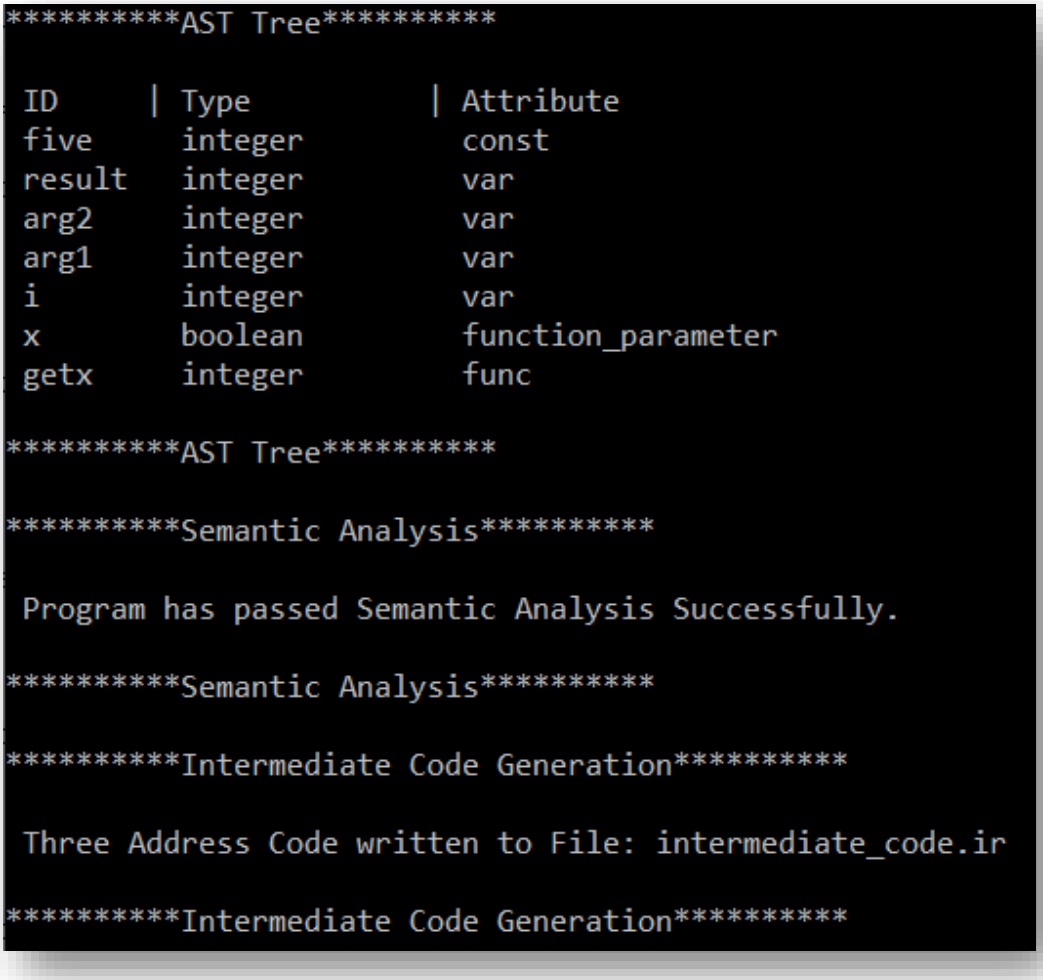
If the semantic analysis check flag has been set to false, then the intermediate generator can check that value and halt printing of the intermediate representation. If successful however a file is generated called intermediate.ir which stores the code and can be translated into another language with ease.

This visitor must make special exceptions for items like if and while statements. This complicates the function. For this use the term 'Seq:' which is short for sequence to describe a block of code and use an increment to append onto each concurrent complex statement. The same is applied to variable identifiers which are assigned the value of 't' and an increment for concurrent items.

An example of ‘IRReadWriteException’ (IR stands for Intermediate Representation) which halted output of generated code as a  result of the semantic analyser catching an incorrect value assignment.




An example below of a successful parse with the printed variables of the Abstract Syntax Tree held on the symbol table. Semantic Analysis has been not detected errors so an intermediate representation can be generated.




The following code diagrams show the final result from a translation of CCAL code into intermediate code. The intermediate code is condensed but still handles the functions and if statements in the more abstract CCAL language. The one criteria missing from intermediate code is scope. With the generated intermediate code blocks, one may be able to translate this into any other language with ease.

```
integer getx (x:boolean)
{
    var i:integer;

    i = 2;
    if(i==2)
    {
        x = false;
    }
    else
    {
        x = true;
    }


    return (x);
}

main
{
    var arg1:integer;
    var arg2:integer;
    var result:integer;

    const five:integer = 5;

    arg1 = -6;
    arg2 = 5;

    result = 3;
}
```

```
getx:
i=2
Seq2:
t1 = i==2
if False
t1 goto Seq3
x=false
Seq3:
main:
five = 5
arg1=-6
arg2=5
result=3

```

## 7. Conclusion

I found this assignment to be of far greater difficulty than the previous assignment. From reading through the samples provided in 'ExprLang.jjt' and 'Using JJTREE and LLVM to build a compiler for a simple language', I was able to build up the compiler one file at a time. I constantly found myself having to remove, add and remove code again in order to make the symbol table work properly. The implementation of JJTree was not too technical as there is plenty of documentation material available found in the JavaCC docs file.

I built a basic semantic analyser visitor following the working of the symbol table and once the Abstract Syntax Tree was finished. I gradually added more rules to the file until I believed it was necessary to include the Intermediate Code Generation visitor alongside. This was not as challenging as the semantic analyser as there is source material provided on the course website.

## 8. References


https://javacc.org/doc

https://www.computing.dcu.ie/~davids/jjtree_example/ExprLang.jjt

https://www.computing.dcu.ie/~davids/courses/CA4003/CA4003_JJTreeVid/index.html

https://www.computing.dcu.ie/~davids/jjtree_example/PrintVisitor.java

https://www.computing.dcu.ie/~davids/jjtree_example/DataType.java

https://www.computing.dcu.ie/~davids/courses/CA4003/jjtree_and_llvm.html

Modern Compiler Implementation in Java
Appel, Andrew W.

Compiler Construction Using Java, JavaCC and YACC
Dos Reis, Anthony.

https://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf


--END OF REPORT--