Name: David Uzumaki

Student number: 150330326

Programme: CASE

Module: CA4003

Assignment: Semantic Analysis and Intermediate representation for the Cal language

Submission date: 17/12/18 at 0:300

Contents

[**Intro** 3](#_Toc532778695)

[**Abstract Syntax Tree** 3](#_Toc532778696)

[**Symbol Table & DataType** 5](#_Toc532778697)

[**Semantic Analysis** 6](#_Toc532778698)

[Intermediate Representation using 3-address code. 7](#_Toc532778699)

[How to Run 7](#_Toc532778700)

# **Intro**

This report will discuss:

1. Abstract Syntax Tree
2. Symbol Table
3. Semantic Analysis
4. 3-adress code

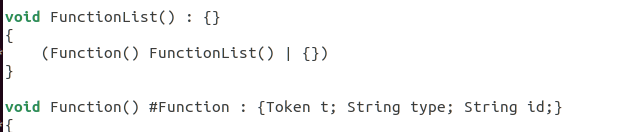
# **Abstract Syntax Tree**

To create this, this was tagging the production rules with child names and in some cases a number of nodes.

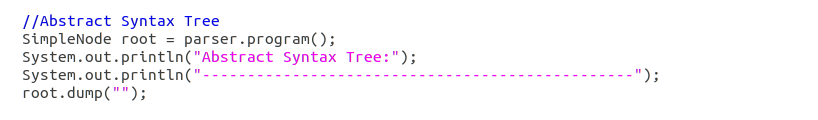
Example.



I also had to change rules that are called 0 or more times from assignment one. In Assignment 1 function was called like “Function()\*” but now it’s split like the original cal file in the documentation:

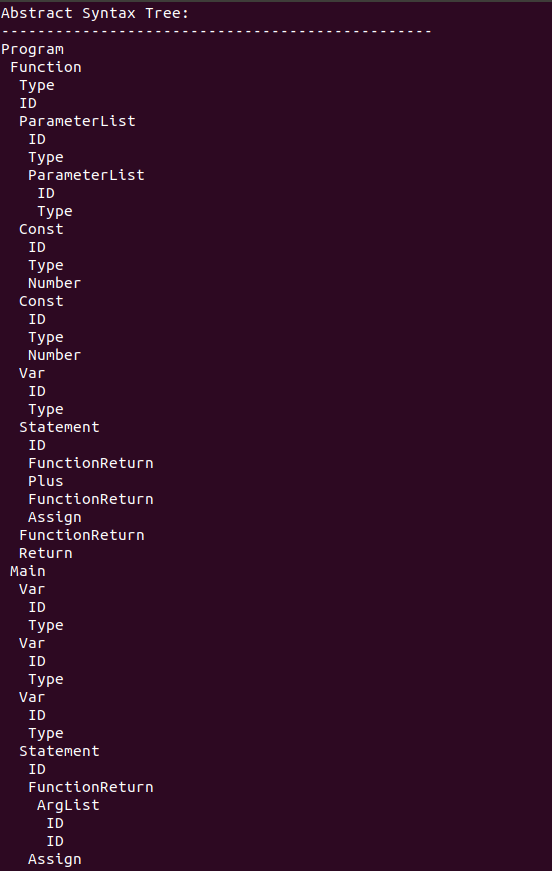


This process was applied to all other 0 or more time production calls from assignment 1 and can be seen in the accompanied cal.jjt file. The AST is generated from:



Specifically, root.dump(“”).

Example AST output:



# **Symbol Table & DataType**

The symbol table is generated in my SymbolTable file. This was a mechanical process for the most part. It’s made up of three hashtables. The first one deals with scope keys matched to a linked list of variables in that scope. The second was a map of keys which is id + scope matched to a data type. The Data Types that can be enumerated are defined in the “DataType” file accompanied with this report.

The third hashtable managed key to phrase(variables, constants, functions and the function paramters).

For efficiency I used conditional node descriptions. Example:



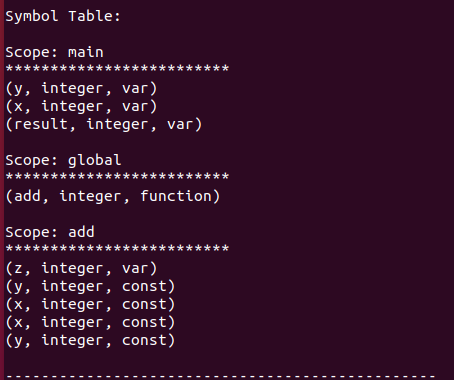
The “>1” means it only appears in the AST if it has more than one child.

The symbol table consists of a few functions in order to manage scope and duplication. It was build like the AST and everywhere ID is called, it is followed by:



The scope is stored as a static variable in the jjtree file and connected to “global”. It is then updated depending on the test file that is ran.

Example output:



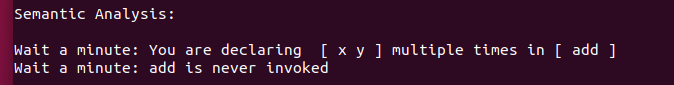
# **Semantic Analysis**

The semantic analysis is generated from the TypeCheckVisitor file and it implements from the automatically generated calVisitor file. It defined static variables which control the semantic check statuses.

The following checks are carried out:

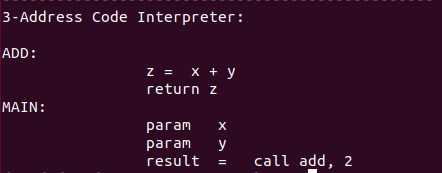
* Is every identifier declared within scope before its is used?
* Is no identifier declared more than once in the same scope?
* Is the left-hand side of an assignment a variable of the correct type?
* Are the arguments of an arithmetic operator the integer variables or integer constants?
* Are the arguments of a boolean operator boolean variables or boolean constants?
* Is there a function for every invoked identifier?
* Does every function call have the correct number of arguments?
* Is every variable both written to and read from?
* Is every function called?

Every “Wait a minute” error displays a message saying what the problem is and if so, where the problem is occurring. (scope wise).



# Intermediate Representation using 3-address code.

This file is made the same way as TypeCheckVisitor excluding printing out any errors. That is why it done after the semantic check so you only have to worry about printing. Example out put below:



# How to Run

1. jjtree cal.jjt
2. javacc cal.jj
3. javac \*.java
4. java cal testfile