11/12/2017

CA4003

Compiler Construction Assignment 2

 Ross.franey3@mail.dcu.ie

Ross Franey

14302851

CASE4

Contents

[Introduction 2](#_Toc500744395)

[AST Generation & Precedence 2](#_Toc500744396)

[Symbol Table & Scope 2](#_Toc500744397)

[Semantic Check Visitor 3](#_Toc500744398)

[Check: No Const is assigned wrong type 3](#_Toc500744399)

[Check: No Var is assigned wrong type 4](#_Toc500744400)

[Check: Identifier declarations are all in scope 5](#_Toc500744401)

[Check: Children of an assignment have been declared before used 5](#_Toc500744402)

[Check: Arithmetic operands are of the correct type 6](#_Toc500744403)

[Check: No Boolean operation uses incorrect type 6](#_Toc500744404)

[Check: No duplicate declarations in any given scope 6](#_Toc500744405)

[Check: Each function has been called 7](#_Toc500744406)

[Check: Error Counter 7](#_Toc500744407)

[Example 8](#_Toc500744408)

[Three Address Code Visitor 8](#_Toc500744409)

[Conclusion/Personal thoughts 10](#_Toc500744410)

## Introduction

In this document, each section of the assignment will be outlined, and some specific problems will be chosen in each section, giving an example of how they were handled. Finally, a test program along with the output of that program will be provided.

## AST Generation & Precedence

The first challenge of this assignment was to convert the previous grammar to a jjt file and then build an Abstract Syntax Tree. During a previous discussion with the lecturer it was mentioned that precedence would be required for Assignment 2. As such, the first step for me in this assignment was to re-arrange my Grammar to that the precedence was correct. Luckily, my previous grammar was not far off and all that needed to be done was re-arrange the order of arithmetic and Boolean operations.

As well as this, new functions were added in the jjt file in order to capture data such as token names, types, and scopes.

Some Token types in the JJT file such as <FUNCTION>, and <BOOLEAN>, <VOID>, <INTEGER>, were changed to functions in order to specify the token and ID of each.

In order to print the AST, the dump() method in SimpleNode was modified in order to print more information about the tokens, such as their values.

In decorating the nodes for the AST, care was taken not to include any unnecessary data, as the point of the Abstract syntax tree is to display only the bare necessary data.

Please find attached with this document, a file containing the CCAL program that was run against the grammar, and the resulting AST.

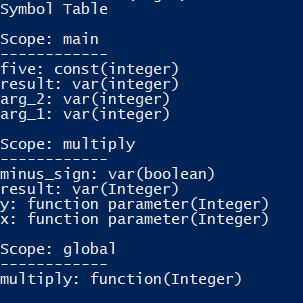
## Symbol Table & Scope

The next requirement for this Assignment was to create a Symbol Table which would accommodate the concept of scope.

After watching the provided videos on YouTube, I began to work on an implementation of the symbol table using a Hash table. A Hash table with keys representing the various scopes in the program, and linked list values representing the list of variables/IDs inside that scope, was added. Unlike the videos however, it was clear that further structures would be necessary. This is to ensure that the next part of the assignment (semantic checking) can be performed. Type checking means a separate hash table would be required for the various types related to each, as well as a hash table for the values of each identifier so that during the semantic checking phase, we could be sure that things like double declarations of the same variable (for example) could not occur.

Finally, a print method was added to the Symbol table class in order to display the scope and respective values and types inside each scope. This simply iterates through each linked list printing the contents.

The following is the symbol table produced from test4 as given in the assignment specification:



## Semantic Check Visitor

The third aspect of this assignment was semantic checking. During this section, there were a number of semantic checks that were carried out, with the intention of covering as closely as possible, all of the checks that the java compiler performs (exceptions are complicated checks such as unsafe operations, etc.).

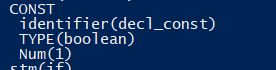
A visitor class was created in order to perform these checks, along with Boolean values for each check that would be created.

Importantly, the semantic checker has been created to ensure that only one error message will print for a specific troublesome area of code. This means that there is a tiered ordering in which checks will fail. If this was not the case, as an example, if this was not the case, a variable that had not been declared in scope would also throw a type error because its value in the symbol table would be “type\_unknown”. Fixing this issue necessitated greater code complexity but the result is a much more specific error message, so the user knows exactly what needs to be addressed first.

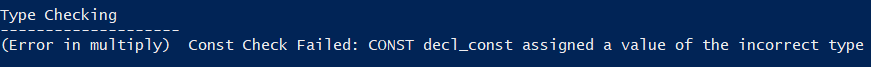
The following checks are carried out:

### Check: No Const is assigned wrong type

Ensuring that each const declared is assigned the correct type was the first check created. As an example, a const defined as an integer should be assigned an integer value only, otherwise an error would be thrown. Here is an example of a Bool const being assigned an integer.



This clearly should not be allowed (if 0 and 1 are not representative of true and false, of course). The resulting error message is printed:

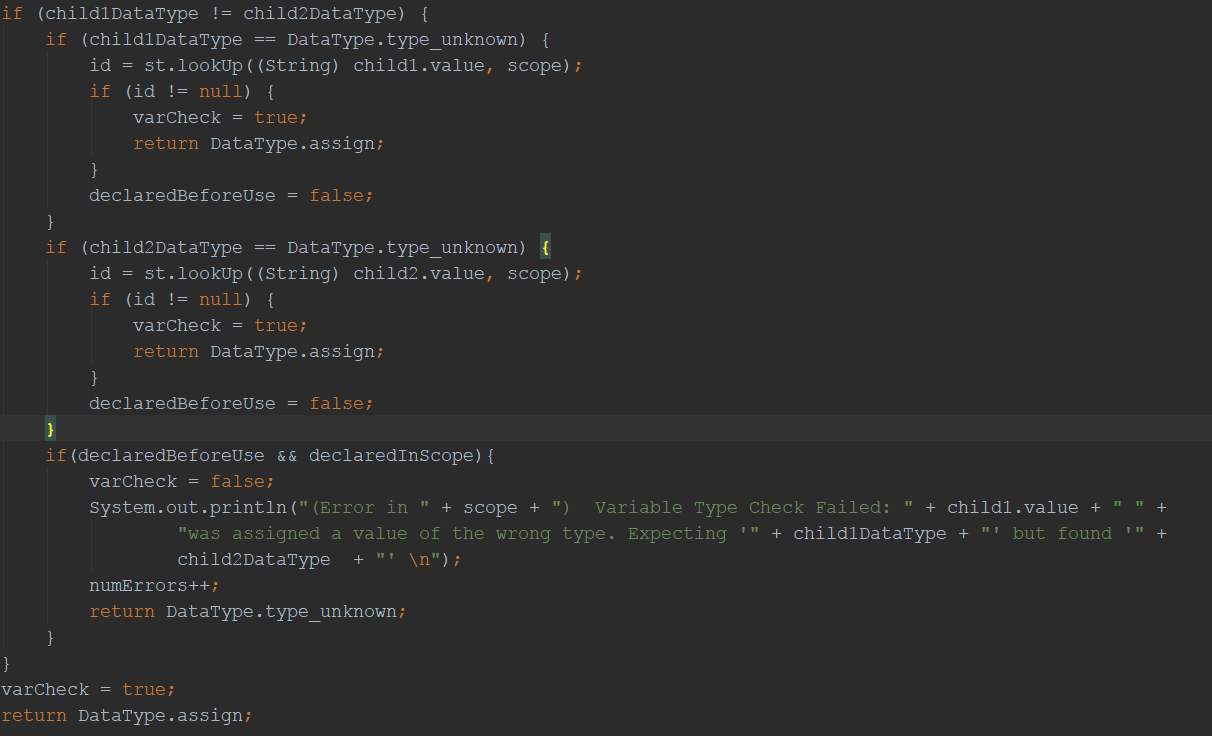


As can be seen, the semantic checker also informs the user of the part of their program (i.e, the scope) in which an error occurs.

### Check: No Var is assigned wrong type

The next Check is the Var check which is checking if a variable is assigned the correct type. In order to prevent the program from printing all failed checks, the program only checks this if a number of other conditions are true.

1. The children of an assign node are of different types
   1. If child 1 has datatype unknown, check the symbol table for an entry. If it is not null, then the check will pass. Do the same for datatype of child 2. If either of the above are not null in the symbol table, then the variable check passes. If they are null in the table, then the declared before use check\*\* fails, as it means there is no entry for a variable that is attempted to be used.
   2. If declared before use check passes, and the variable is declared in the current scope\*\*, the variable check fails and an unknown datatype is returned.
2. The children of an assign node are of the same type. In this case, the check passes and the “Assign” datatype is returned.



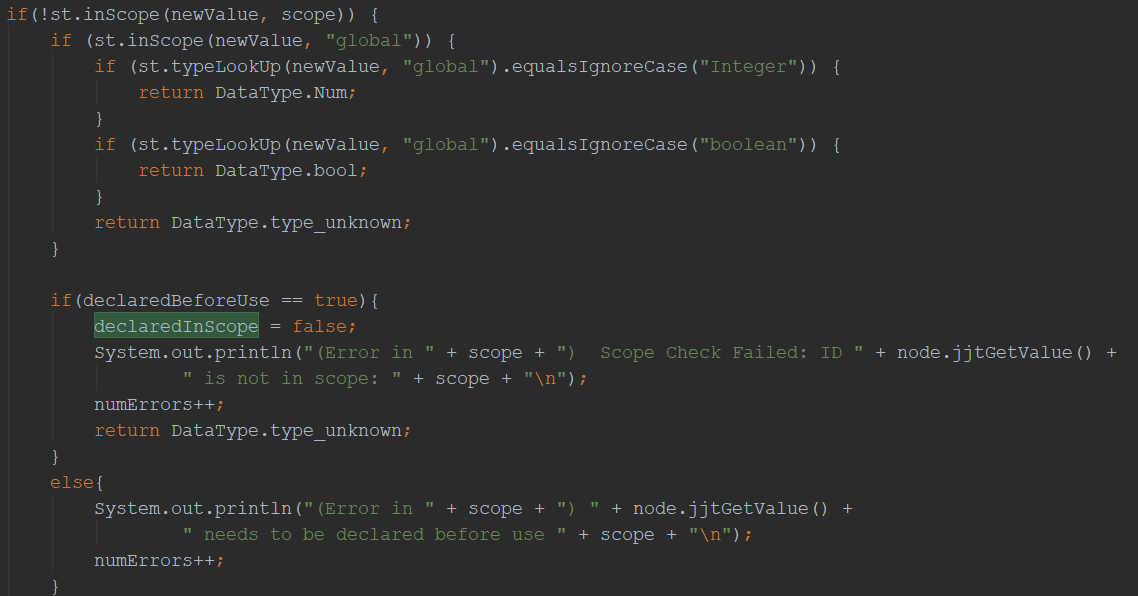
### Check: Identifier declarations are all in scope

The next check is the declared in scope check. This ensures that an ID being referenced is present either in global scope, or in the function which is trying to reference it.

There are a number of conditions that need to be true in order to carry out this check.

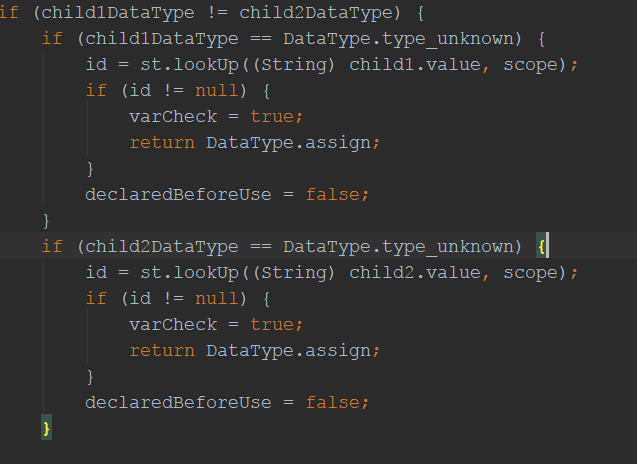
1. Firstly, the parent of the identifier needs to be either a variable, const or function.
2. Second, the node must not be in scope. This is checked using an inScope Boolean method in the symbol table, which essentially indexes the symbol table with the scope and id and makes sure there is an entry. If there is not, the function returns false.
3. it is ensured that the declared before use check has passed, as otherwise we do not want to print a scope error, but rather that there is simply no declaration anywhere in the program, not just this scope.

If there is a declaration somewhere else in the program (but not in scope) which matches, the scope check fails and the corresponding message prints. Otherwise, if there is no match in the entire code, we ignore the scope check and print a more serious error, corresponding to the failure of the declared before Use check.



### Check: Children of an assignment have been declared before used

The next check is the declared before use check. This check ensures that children of an assignment have been declared before the program attempts to use them. In order to carry out this check, the id is looked up in the symbol table using a look up function which simply indexes the symbol table using the value and scope of the current node. If the resulting string is not null, then the declared before use check passes.



### Check: Arithmetic operands are of the correct type

The next check is for arithmetic arguments, in order to ensure they are of the same type. Inside the add and minus operation visit functions, the data types of child1 and 2 of the node is checked to ensure they are equal. If they are, the check passes, if not it fails and returns an unknown datatype.

### Check: No Boolean operation uses incorrect type

A Boolean check also exists in order to ensure that no Boolean comparison operation uses the wrong type. The code for this Is implemented in all the comparison operation visitor functions, and checks that both children are of type num.

### Check: No duplicate declarations in any given scope

Finally, a check was created to ensure that duplicate identifiers were not declared in the same scope. This was by far the most challenging of all the checks performed.

In order to achieve this, I first created a “duplicates” Boolean in the semantic check visitor class which is initialized as false, and an Array List to hold all of the scopes that will be visited during the course of the program. After this, at each point where the scope changes in the visitor code, the name of the new scope is added to the Array List.

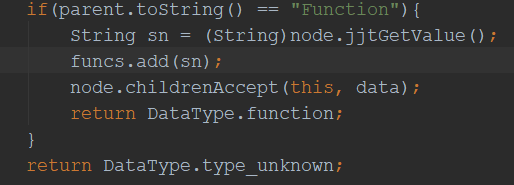
Once the visitor is complete, a function aptly named dupesInScope() is called, which has been created in the ASTSymbolTable class, with the input being each individual string (scope) stored in the aforementioned Array.

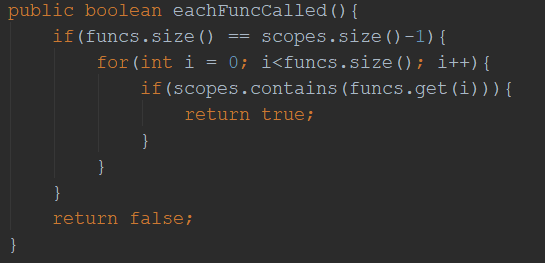
This method checks the linked list of the symbol table whose key is the scope input, and returns a string of the correct error message to print if there are duplicates, or null if there are none. If the return string is not null, the error will be printed and the duplicate Boolean will be set to true.

### Check: Each function has been called

To achieve this check, an eachFuncCalled() Boolean has been created. Each identifier that is called which has a parent of type “function” will be added to a new Array List called funcs. As well as this, each scope that is added will be added to an Array List called scopes. At the end of the program, the semantic checker will run the eachFuncCalled() function and ensure that:

1. The length of the scopes array is exactly one item larger than the funcs array (accounting for scope global which is omitted)
2. Each element in the funcs array is also present in the scope array (to ensure the array isn’t simply the same length but with incorrect function names).



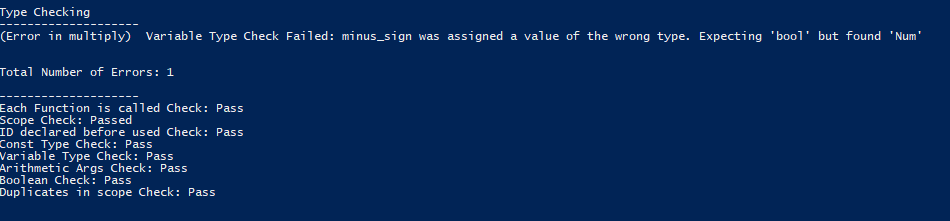




### Check: Error Counter

A counter has been created to establish the number of errors that occurred in the program, and is incremented every time a check is set to false.

### Example

The semantic checker called on the program used for testing throughout this document yielded the following result:

* To test the error messages, simply purposely create any of the errors listed above in the test4.ccal program provided in order to compare the result with the above (correct) output.

## Three Address Code Visitor

This was perhaps the most time-consuming aspect of the assignment, as initially a visitor which printed the code as if it were java was created, only to notice three address code is in fact the specified format. This was beneficial however as it forced the revision of the lecture notes on three address code, and granted me a greater familiarity which will no doubt be beneficial.

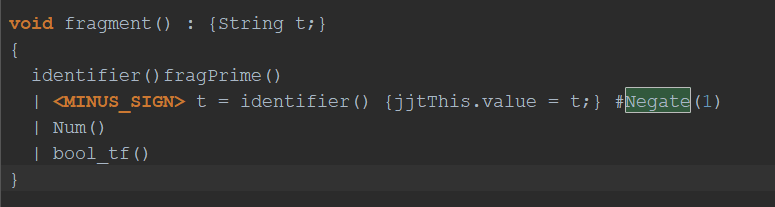
The class itself contains the same visitor functions as the semantic check visitor, however rather than returning a datatype, they simply print the required information in the format which is as close to three address code as possible, then return null.

In order to track the labels, a simple label counter was created. This label counter appends its value to the end of “L” each time a new label is created. This occurs each time there is a “while” or “if” statement in the code.

In the while loop deviation, the condition of the statement is checked and “ifFalse” is printed followed by the condition, followed by a GOTO, which goes back to the start of the statement (the label).

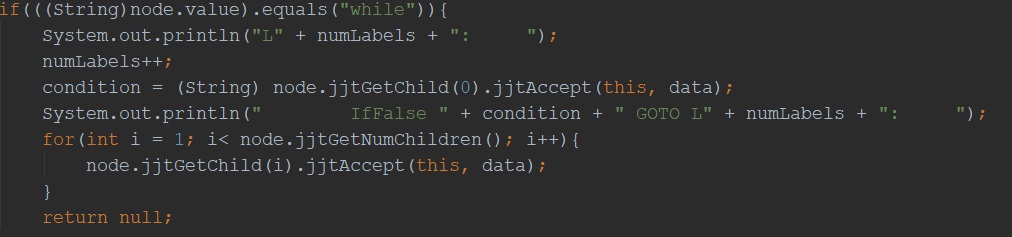
The if statement deviation is handled similarly, however the looping function of the above while statement is replaced with a GOTO to an ”else” label.

One issue encountered here was printing out negative integer variables. To tackle this issue, a String was added to the fragment rule in the JJT file, which would be the value of the identifier and be assigned as the value of the Negate node in the 3-address code visitor.

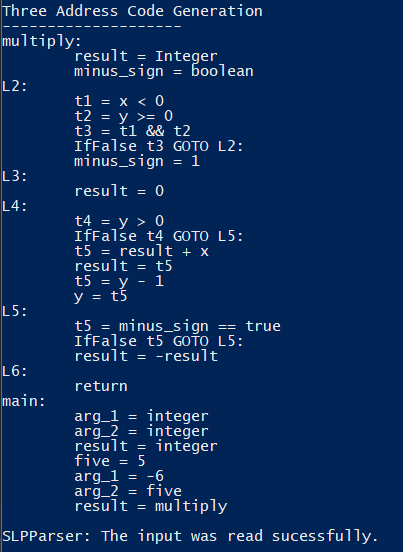


Then, in the 3-address code visitor, the Assign function was modified to check that if the second child is null (this would mean that child 2 was the Negate node), A simplenode for that null child is created, and it’s value, passed as mentioned above, will be the ID string. This can now be printed with a minus sign in order to represent the negative variable in 3 address code.

For statements in the 3-address code visitor, two separate cases were created. One for while loops and one for if statements. Originally, the issue of only one child being printed in each of the statements. To resolve this, a for loop was used with the jjtGetNumChildren() method of the SimpleNode class.



This ensures that all children in the statement are printed.

The following is the 3-address code generated from the test4 program:

## Conclusion/Personal thoughts

Although very time consuming, this assignment was enjoyable to work on, and through my investigation and research as part of its completion, I acquired a far greater appreciation of the complexities involved in compiler construction, the surface of which, it would seem, this project has barely skimmed.

Despite out of the scope/requirements of the assignment, noteworthy is my attempt to implement line numbers in error messages using token counting, and is my largest disappointment with the submission. As the core requirements were prioritized, I did not have the capacity to find a solution in the given timeframe.

This aside, I am happy with my submission both in terms of what has been learned, and indeed the final result of each section.