Declaration on Plagiarism

Assignment Submission Form

This form must be filled in and completed by the student(s)submitting an assignment

Name(s): Killian Connolly
Programme: Computer Applications & Software Engineering
Module Code: CA4003
Assignment Title: Digital Semantic Analysis & Intermediate Representation
Submission Date: 14th December 2020
Module Coordinator: David Sinclair

I/We declare that this material, which I/Wenow submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/ourwork. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/Wehave read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/usor any other person for assessment on this or any other course of study.

I/Wehave read and understood the referencing guidelines found at http://www.dcu.ie/info/regulations/plagiarism.shtml, https://www4.dcu.ie/students/az/plagiarismand/or recommended in the assignment guidelines.

Name(s): Killian Connolly Date: 14th December 2020

Assignment

Description of your abstract syntax tree structure and symbol table, how you implemented them and how you implemented the semantic checking and intermediate code generation

Generating Abstract Syntax Tree

The first task of this assignment was to generate an Abstract Syntax Tree for the grammar. Using the resources in the notes, I assigned the parsed programme to a ParseTree object, therefore creating an Abstract Syntax Tree provided by Antlr4. I understand that this is not a perfect AST, however it was mentioned that it is sufficient for this assignment.

Semantic Symbol Table

The Symbol Table file contains a Symbol table hashmap and an Undo Stack. The Symbol Table hash map consists of String to a LinkedList of Strings relationship and the Undo Stack consisting of Strings.

The hashmap insert function, inserts ids and types. Types are added to the start of a LinkedList. The Undo Stack commences once a user has entered a function. If the user assigns a different type to the same id before and during a function, the LinkedList will then consist of two values of different types. During the function, the ID is pushed onto the Undo Stack. Once the function is completed the Undo Stack cleared by iterating over, and identifiers that were in the Stack will have their first value in their associated LinkedList removed.
The Symbol table class also contains a hashmap of string and linkedlist for functions. This is explained why later in the report. LinkedLists functions for storing, obtaining and editing LinkedLists which is required for my implementation of some semantic checks are also stored in this class.

After implementing the symbol table I came to the realization that implementing Node class for my Symbol Table would result in more dynamic, easy to use and robust code.

Error Handling

The Semantic check visitor will return a string best describing the error that has just occurred and program is exited. Attempts were made to implement the program to continue after initial error, however, identifiers that cause an initial error cause exponentially more errors.

Semantic Analyser Visitor
The calVisitor class that I have implemented as my semantic analyser is extended from the BaseVisitor that was created when calling the ‘-visitor’ argument when running the grammar file with Antlr4.
From there, the process was to append every type to a given value using the symbol table and ensuring that the program is semantically compatible with

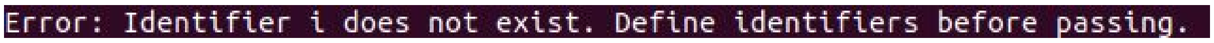
Scope

As mentioned in meetings regarding the assignment, it was specified that scope was not required so no implementation of main scope was included in my assignment. However, later in the process of completing the assignment, scope would have been helpful for some of the semantic checks. Undo Stack covered the majority of the scope handling. Especially when it came to functions where the scope is one dimensional.

<u>Semantic Checks</u>

<u>Is every identifier declared within scope before it is used?</u>

If an identifier is called upon and is not declared within the symbol table at that given time, in any scope, then an error is reported declaring that you must define the identifier before using them:

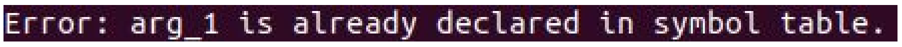


```
Error: Identifier i does not exist. Define identifiers before passing.
```

A function was created inside the Symbol Table class that obtains the most recent type that has been added to the identifier. If nothing is contained in that LinkedList a string is returned symbolising that the entry is nonexistent. In the visitor class if this string comes up when identifying a fragment the error is produced.

<u>Is no identifier declared more than once in the same scope?</u>

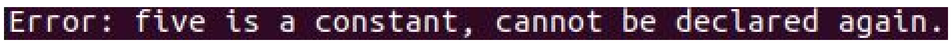
Declaring identifiers twice in the same scope is unnecessary and can cause issues later in your program. Hence, it will throw an error declaring the identifier is already declared.



```
Error: arg_1 is already declared in symbol table.
```

In order to implement this error I stored every time a new variable is declared in each scope. A checker is created before declaring if the variable is already stored, if it is, the error is thrown. After each scope the storage is clear ensuring variables do not carry on to the next scope.

If an identifier that has previously been identified as a constant, this will throw an error since constants cannot be reassigned.

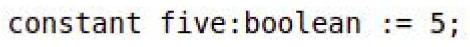


```
Error: five is a constant, cannot be declared again.
```

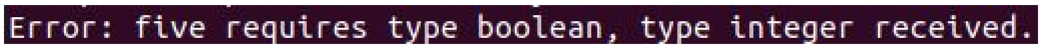
Similar to above, every time constant declaration is made, the identifier is stored ensuring that the identifier does not get declared again. A check is made in variable declarations for constant declarations as well, making sure that constant variables are not being reassigned.

<u>Is the left-hand side of an assignment a variable of the correct type?</u>

Each possible value is hard coded to return their type. For example, visitNumberFrag() will return an “integer” string. In doing this, allows for proper variable assignment checking. If a variable previously declared as an integer and was then received a boolean. The fragment will return a boolean value to the assignment rule method and an error is thrown.
For the example below, the identifier is declared as boolean, when assigned an integer:



```
constant five:boolean := 5;
```



```
Error: five requires type boolean, type integer received.
```

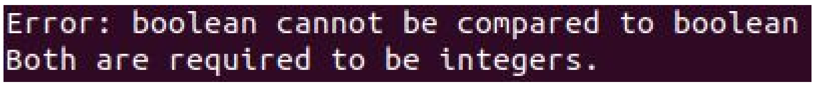
The implementation of this varies for constant declaration and variable declaration.

For constant declaration, the type is received as a string and the type is obtained from visiting the expression child. These types are compared and if they are not the same it throws an error.
For variable declarations whenever an assignment is made to an identifier, a check is made with the symbol table against the type of the assigned variable.

Are the arguments of an arithmetic operator the integer variables or integer constants?

The results of the expressions on each side of an arithmetic operation must result in them being integer variables or constants. Simply by obtaining the type from the result of the expression from each side of the arithmetic operation, we can compare that both results are integers. Code found in visitBinOpExpr().
Example:

```
arg_1 := true - true;
```



```
Error: boolean cannot be compared to boolean
Both are required to be integers.
```

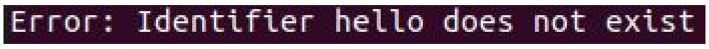
Is every variable both written to and read from? Is every function called?

Both of these semantic checks are signs of poorly written code on the user's behalf. As specified in the meeting regarding this assignment, most languages accept code that is written and is never made use of. I made the decision that it was okay to proceed in creating the language without these semantic checks.
Are the arguments of a boolean operator boolean variables or boolean constants?

Boolean operators within the language implemented included & and | operations and Equal(=) to and Not Equal(!=) to operators. These checks are implemented in visitOrAndCond() and visitExprCompOpCond(). Both of these operations ensure that the values from the result of the expressions are both boolean. Any other type that breaches the operation, results in an error expressing that the types are incompatible with the requested operation. The output error is similar to the one displayed above.

Is there a function for every invoked identifier?

Once an identifier is invoked to a type, and the identifier is assigned to a called function, that function must exist. If a function is called and does not exist the following error is produced.



```
Error: Identifier hello does not exist
```

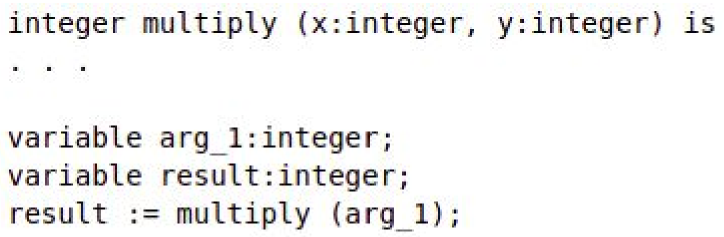
This checker is implemented within visitorIDExpr() methods. Since I have created a separate hashmap for storing functions, I request the list of parameters that the list has stored. If the request comes out as a null pointer error, there is nothing in the function list, meaning that function has not been added to it. This ensures that the function does not exist then throwing this error.

Does every function call have the correct number and type of arguments?

By obtaining the saved parameters types from the functions Hashmap allows for a comparison to be made with the given arguments. These checks are implemented in the

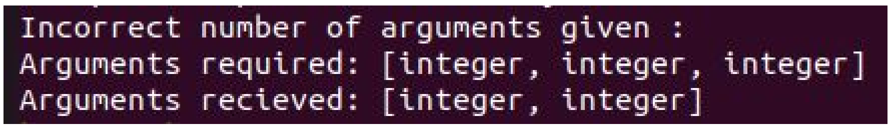
visitIDExpr() function. The arguments are stored in a global list. Each time the children enter the visitNempArgList() function, the type of the argument is added to the global list. As an additional semantic check, I made sure that the identifier arguments given are existing in the symbol table. The type of identifier that is currently being assigned to the output of the function appended to this list also. At the end of this, the list should contain arguments and the type returned and can then be compared to the linked list in the functions hashmap. This global list is cleared once the assignment has been made for the next time a function is called.
The first check involved checking the size of each of these lists.
The second check is ensuring that the types of both arguments and return type are the same in both arguments and required parameters.

Example showing number of arguments are insufficient:



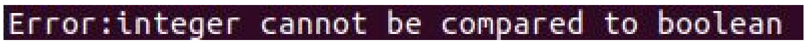
```
integer multiply (x:integer, y:integer) is
. . .

variable arg_1:integer;
variable result:integer;
result := multiply (arg_1);
```



```
Incorrect number of arguments given :
Arguments required: [integer, integer, integer]
Arguments recieved: [integer, integer]
```

If the result was declared as a boolean and the correct number of arguments are given, then the following error will occur:



```
Error:integer cannot be compared to boolean
```

Intermediate Representation Generation

Unfortunately, I could not get this assignment to its completion.

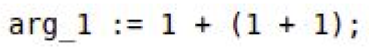
Intermediate Representation Symbol Table

Understanding the purpose for the symbol table for the intermediate code representation visitor was a mystery until in the meeting where it was specified that the symbol table will hold temporary variables. I made a few attempts at implementing this method but I was unable to fully implement temps dues time constraints. For clarity, I understand for optimization, using the symbol table to store temporary variables is the main purpose. Once you have your symbol table filled with temporary variables, I could optimise it to reduce the number of these variables, hence increasing run time and reducing memory. As a simplistic solution, I chose that I will store identifiers and values into the Symbol Table.

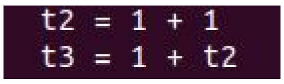
Intermediate Representation Visitor

Temporary variables

My only implementation of temporary variables are handled within the expression that calculates binary arithmetic operations(visitBinOpExpr()). There is a global temp counter that iterates every time you assign a temporary variable. These variables should then be stored in the symbol table used for later us.
Example showing an assignment to a nested expression and the result of the Intermediate Representation code.



```
arg_1 := 1 + (1 + 1);
```

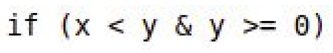


```
t2 = 1 + 1
t3 = 1 + t2
```

Conditions

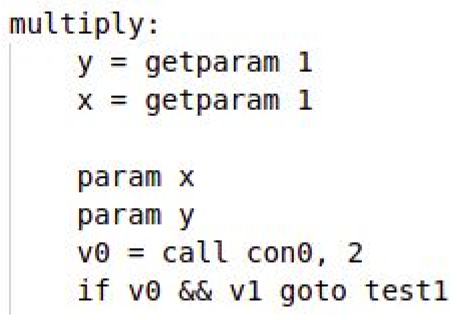
Conditions were handled in visitExprCompOpCond(). My thought process was each condition is required to have its own condition block and the scope that is currently being executed calls upon that block. Hence, the condition block should be printed out after the execution of the current block. Implementation involved taking note of the number of expressions that were identifiers rather than integers. If the answer is two, like the example below, two conditional blocks are required.

The example below show the taci code for the if statement:
If Statement:



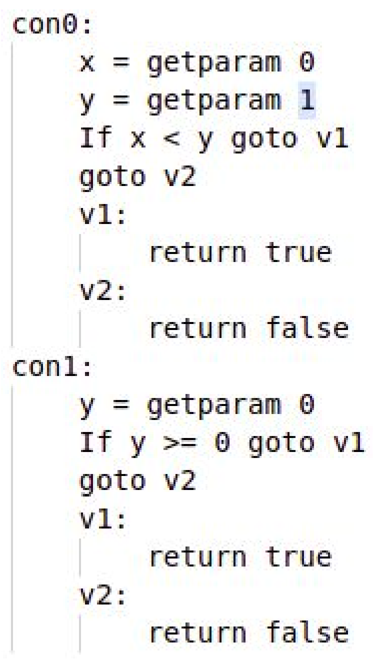
```
if (x < y & y >= 0)
```

Taci Code:



```
multiply:
    y = getparam 1
    x = getparam 1

    param x
    param y
    v0 = call con0, 2
    if v0 && v1 goto test1
```



```
con0:
    x = getparam 0
    y = getparam 1
    If x < y goto v1
    goto v2
    v1:
        return true
    v2:
        return false
con1:
    y = getparam 0
    If y >= 0 goto v1
    goto v2
    v1:
        return true
    v2:
        return false
```

Unfortunately nested if statements do not correctly output the right taci code. My current implementation creates a block for each condition it visits. The stack for the parameters for calling both function and condition blocks is working correctly to my knowledge. The condition blocks are outputting the correct compound operation, assigning the correct parameters and naming conventions for the parameters are also operational.

<u>While Loops & Functions</u>
Unfortunately, I could not get loops or functions implemented after many attempts.

<u>Output to file</u>
The taci 3 bit code interpreter is printed to a file called example.taci. This code is written in the cal.java file. I utilised the library PrintStream that outputs everything that has been System.out to a file.

<u>Conclusion</u>

All example files given in the first assignment successfully parse and no semantic errors arise. Throughout this assignment, I created my own cal files that I used for testing. These files purposely threw errors where they were supposed to. Most of the last given file in the first assignment was given edits to throw these semantic errors.

I have provided in the tests folder files that should throw specific errors(except example.cal) and my attempt at a taci file.