Semantic Analysis and Intermediate Representation for the CCAL Language

CA4003 Compiler Construction
Assignment 2

Name: Hannah O’Connor
ID: 16382283
Date: 16/12/2019

Declaration

I declare that this material, which I now submit for assessment, is entirely my 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 work. I understand that plagiarism, collusion, and copying is a grave and serious offence in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, 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.
I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.
This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. I have read and understood the referencing guidelines found at http://www.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.
I understand that I may be required to discuss with the module lecturer/s the contents of this submission.
I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.

Signed: Hannah O’Connor
Date: 16/12/2019

Introduction

This purpose of this assignment was to continue my previous work on implementing a Lexical and Syntax Analyser to accept the language CCAL. Here, I added various semantic analysis checks and intermediate representation generation. However, in order to be able to implement those checks, I need to first create an Abstract Syntax Tree and Symbol Table.

In order to complete the given workload within this given assignment I used the module's notes as a basis and used some additional online resources which I've appended within *References.*

Options

```
options
{
    JAVA_UNICODE_ESCAPE = true;
    IGNORE_CASE = true;
    MULTI = true;
    VISITOR = true;
    NODE_DEFAULT_VOID = true;
    NODE_PREFIX="";
    // enable below option for debugging
    //DEBUG_PARSER=true;
}
```

- ❖ **NODE_DEFAULT_VOID** is set in order to make the compiler not create nodes by default for each production rule. Instead, it'll only create node for those who have the #<node_name> declaration.
- ❖ **VISITOR** is set in order to allow the HannahParserVistor interface to be generated.

Abstract Syntax Tree

At the beginning, implementing my abstract syntax tree was quite a simple process.
I started converting my *.jj* parser file to *.jjt* and convert my **program** rule, the first rule, to be of type **SimpleNode** in order to implement the **Node** class. This meant that **program** became the parent node to all of the others that are declared after. As shown in the below snippet, it returns a type SimpleNode which is assigned to root. By then invoking it's dump method, it prints the AST to the terminal.

```
SimpleNode program() #program : {}
```

```
{
  decl_list() function_list() main() <EOF> {return jjtThis;}
}
```

```
        SimpleNode root = parser.program();
        root.dump("");
```

I could then proceed to add # decorations to all my production rules to grasp an understanding of the complete code flow of my inputted test files. It showed how the parser iterates through each of the various files based on the given input.

Once I could examine and analyse the printed tree I eliminated the nodes in which I felt weren't relevant or just redundant. For example, I accessed what were simply just sub functions like **other_statement()** rather than being children of a given node. Here, I sectioned off those who wouldn't be required to be classed as its own node. Taking **other_statement()** as an example again, it would just fall under #statement and I would take a closer exam at the various token paths for each rule, as shown below.

Example:

```
 void other_statement() : {Token token;} {
     token = <ASSIGN> expression() <SEMICOLON> {jjtThis.value = token.image;} #assign
   | token = <LPARENS> arg_list() <RPARENS> <SEMICOLON> {jjtThis.value = token.image;}
#function_call
}

void statement() #statement : {Token token;} {
      <ID> other_statement()
    | <LBRACE> statement_block() <RBRACE>
    | token = <IF> condition() <LBRACE> statement_block() <RBRACE> <ELSE> <LBRACE>
statement_block() <RBRACE> {jjtThis.value = token.image;}
    | token = <WHILE> condition() <LBRACE> statement_block() <RBRACE> {jjtThis.value =
token.image;}
    | <SKIP_TOKEN> <SEMICOLON>
}
```

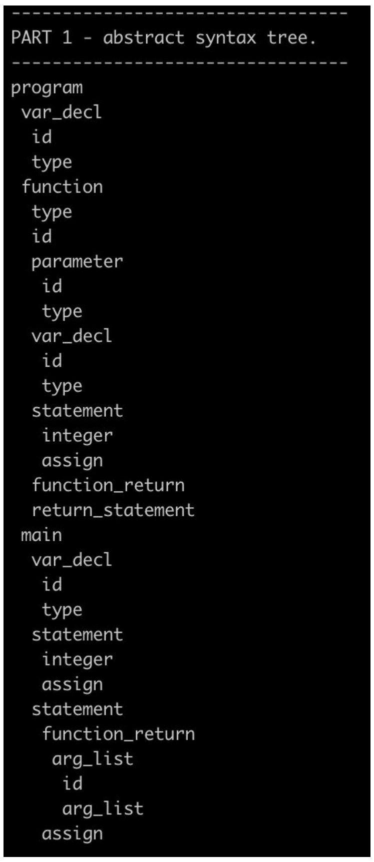
As shown in the above snippet, the token paths are treated independent, given their different purposes, yet both are still children of statement.
Given that the AST shows the path and from several debugging sessions setting **DEBUG_PARSER**, I've earned a relatively good understanding of the path and structure of my test cases. This aided my greatly when it came to planning my later semantic checks.

From reading the notes, I learned that I could have conditional nodes, by including how much nodes to include. For instance, within my production rule **nemp_parameter_list(),** it means that this node will only be shown in the AST if its' child count is greater than one. This helped enhance the efficiency greatly as for that rule in particular it can also allows for the empty string too which I would have ended up having a node for.

Example:

```
void nemp_parameter_list() #parameter(>1) : {String id; String type;} {
    id = id() <COLON> type = type() other_nemp_parameter_list() {
      symbolTable.put(id, type, "parameter", scope);
    }
}
```

Below shows the returned AST from my test file **scopes.ccl**



```
--------------------------------
PART 1 - abstract syntax tree.
--------------------------------
program
 var_decl
  id
  type
 function
  type
  id
  parameter
   id
   type
  var_decl
   id
   type
  statement
   integer
   assign
  function_return
  return_statement
 main
  var_decl
   id
   type
  statement
   integer
   assign
  statement
   function_return
    arg_list
     id
     arg_list
   assign
```

Symbol Table

For my Symbol Table, I created it's own java class which extends **Object.**
I chose to use Java's Hash Table structures for its implementation given that upon researching possible variations, It was the most common data structure used, especially given that Insertion and lookup can be very fast at O(1). Not just that, it seemed like a more straightforward approach then a binary search tree, for example, which is another said common implementation. [1]

My symbol table has 3 Hash Tables. The first "***symbolTable***" maps each declared scope to its individual linked list of all the ids that are contained within that given scope based on the given input. It's initialised with the global scope to an empty linked list, as all given input will have this scope by default and all other scopes will be a child to it. The second "***typeTable***" whose keys are the concatenation of a given id and the current scope. This table maps it's keys to a defined data type (integer, boolean). The last, "***valueTable***" keys are also the concatenation of the given id and current scope, which is mapped is a specified qualifier, such as variable, parameter, etc.

I also needed to consider some other tokens instead would act as their own nodes, but not append to my symbol tree. Those in which I hadn't considered in my previous section upon original construction of abstract syntax tree.

When developing my symbol table, it made me rethink my previous implementation of the AST. For example, when inserting items into the table, I originally had:

```
void var_decl() #var_decl : {Token token; String id;} {
  token = <VAR> <ID> <COLON> type() {
     jjtThis.value = token.image;
     symbolTable.put(<ID>, type(), "variable", scope);
 }
```

However, after finishing the code for my symbol table, I was presented with choice conflicts caused by <ID> token, this led to me having to refactor and essentially, rewrite a lot of my previous rules.

Example:

```
Warning: Choice conflict in [...] construct at line 219, column 4.
         Expansion nested within construct and expansion following
construct
         have common prefixes, one of which is: <ID>
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict involving two expansions at
         line 223, column 5 and line 223, column 18 respectively.
         A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier expansion.
```

A work around for this was to assign <ID> to a string first through making it it's own production rule and node. This meant that I could easily access the id itself, which was particularly useful for the latter semantic checks.

Solution:

```
void var_decl() #var_decl : {Token token; String id;} {
    id = <ID>
    token = <VAR> <ID> <COLON> type() {
      jjtThis.value = token.image;
      symbolTable.put(id, type(), "variable", scope);
    }
}

String id() #id : {Token token;} {
  token = <ID> {
    jjtThis.value = token.image;
    return token.image;}
 }
```

But it wasn't as straightforward as that with all of my production rules. For example:

```
Error: Line 365, Column 1: id occurs on the left hand side of more than
one production.
```

This was originally caused by the below rule:

```
void nemp_parameter_list() #parameter(>1) : {Token token; String id;} {
      id = id() <COLON> type() other_nemp_parameter_list() {
      symbolTable.put(id, type(), "parameters", scope);
    }
}
```

Solution:

```
void nemp_parameter_list() #parameter(>1) : {Token token; String id;} {
    <ID> <COLON> type() other_nemp_parameter_list() {
      symbolTable.put(id(), type(), "parameters", scope);
    }
}
```

By including the assignment id = id() within the token assignment it meant that it was longer on the left hand side. However, by doing this meant that I also had to change the return type of the type() rule to be String too and treated as its' own separate node.

```
String type() #type : {Token token;} {
   (token = <INTEGER> | token = <VOID> | token = <BOOLEAN> | token = <INT>)
{jjtThis.value = token.image; return token.image;}
}
```

In terms of the various scopes, by having them it meant I could logically separate the variables and functions within their own. Apart from global, the other two scopes that I dealt with was **main** and **<function_id>**.

For main:

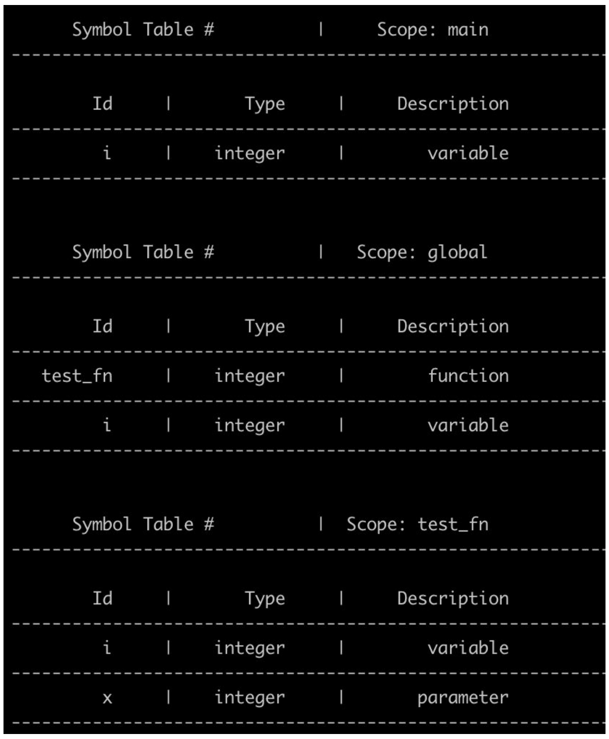
```
void main() #main : {} {
    {scope = "main";}(
    <MAIN> <LBRACE> decl_list() statement_block() <RBRACE>)
}
```

For function:

```
void function() #function : {String id; String type;} {
    // function scope !
    (type = type() id = id() {symbolTable.put(id, type, "function", scope);
scope = id;
     if(!scope.equals("global")) {
        symbolTable.put(id, type, "function", "global"); }
      }
    <LPARENS> parameter_list() <RPARENS> <LBRACE>
    decl_list()
    statement_block()
    <RETURN> <LPARENS> (expression())? <RPARENS> <SEMICOLON> #return_statement
    <RBRACE>)
}
```

A variable could be declared as either global and be accessible from any point in the program. Not only variables but functions are classed as global too, hence why each is added to the symbol table with the value “function”. It ensures that there won’t be any duplicates, as expected in a regular program. But as every scope can declare and assign their own variables, function needs to be considered a scope. You can see this done in the line `scope = id;`

Example Symbol table using input file **scopes.ccal:**



```
      Symbol Table #             |     Scope: main
-------------------------------------------------------------
        Id      |        Type      |      Description
-------------------------------------------------------------
         i      |     integer      |          variable
-------------------------------------------------------------


      Symbol Table #             |   Scope: global
-------------------------------------------------------------
        Id      |        Type      |      Description
-------------------------------------------------------------
     test_fn    |     integer      |          function
-------------------------------------------------------------
         i      |     integer      |          variable
-------------------------------------------------------------


      Symbol Table #             |  Scope: test_fn
-------------------------------------------------------------
        Id      |        Type      |      Description
-------------------------------------------------------------
         i      |     integer      |          variable
-------------------------------------------------------------
         x      |     integer      |         parameter
-------------------------------------------------------------
```

Semantic Checks

For this section, I created a SemanticAnalysis class which implements **HannahParserVistor.** It's responsible for performing the given semantics checks for each of my nodes using the AST and symbol table. The HannahParserVistor interface was automatically generated once running my *jjt* file. Here, it allows my class to *visit* each of the defined nodes in my AST and easily access the children of each too.

- ### Is every identifier declared within scope before its is used?

For this check, I needed to assess that each given identifier was declared first, before being accessed. To do this, I implemented a function **declaredCheck()** into my class. It checks that either the global or current scope contain the given id.

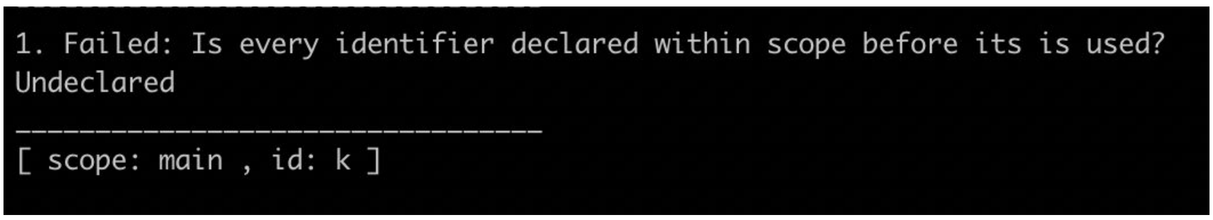
```java
private static boolean declaredCheck(final String id, final String scope) {
  final LinkedList<String> list = symbolTable.getSymbolTable(scope);
  final LinkedList<String> global_list = symbolTable.getSymbolTable("global");
  if (list != null) {
    if (!global_list.contains(id) && !list.contains(id)) {
      return false;
    } else {
      return true;
    }
  }
  return true;
}
```

That function was then called within my statement node as there is where variables will be assigned values. Hence why I need to check that they're declared first.

```
try {
//    // TODO - Fix
//    // Some reason my ids are printing as
//    // id = x
//    // id = 0
//    // id y
//    // id -6
//    // Adding a work around check to make sure the id isn't an int!
  Integer.parseInt(id);
} catch (final Exception e) {
  if (!declaredCheck(id, scope)) {
    noUndeclared = false;
    unDeclared.put(scope, id);
  }
}
```

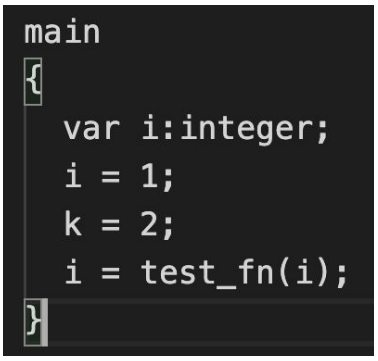
As seen within the above, the comment show work around to implementing this check. I was retrieving both the id and it's value when traversing. Due to the time constraint I was unable to provide an efficient solution. Instead, I added a check to ensure that the current id wasn't an integer. If I found an identifier which wasn't declared, I would set noUndeclared to false and add each current id and the scope to a hash set where they could be printed to the terminal.

Example failed output:



```
1. Failed: Is every identifier declared within scope before its is used?
Undeclared
----------------------------------------
[ scope: main , id: k ]
```

Corresponding test file:



```
main
{
  var i:integer;
  i = 1;
  k = 2;
  i = test_fn(i);
}
```

Is no identifier declared more than once in the same scope?

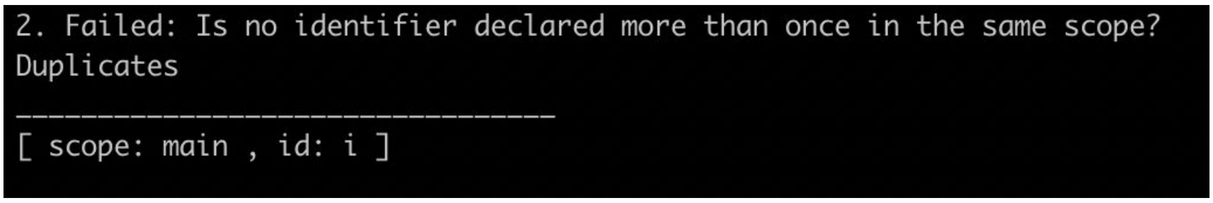
The first obvious way to implement this for me was to create a symbol tree class method that could check the symbol tables for the id entries for each scope, removing any that occur twice.

```java
public boolean duplicates(){
  Enumeration e = symbolTable.keys();
  while (e.hasMoreElements()) {
    String current_scope = (String) e.nextElement();
    LinkedList<String> idList = symbolTable.get(current_scope);
    while (idList.size() > 0) {
      String id = idList.pop();
      if(idList.contains(id)){
        noDuplicates = false;
        dupsTable.put(current_scope, id);
      }
    }
  }
  return noDuplicates;
  }
```

Within the duplicates method, I get a list of all of the ids in the given scope. I iterate through that list and pop them off as I go. I then check to see if that identifier is still within the list after popping. Here, if I encounter any duplicates I add them to the Hashtable dupsTables to keep track and be able to print to the screen;

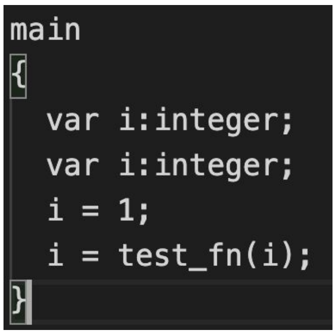
```java
Hashtable<String, String> dupsTable;
```

Example failed output:



```
2. Failed: Is no identifier declared more than once in the same scope?
Duplicates

-----------------------------------
[ scope: main , id: i ]
```

Corresponding test file:



```
main
{
  var i:integer;
  var i:integer;
  i = 1;
  i = test_fn(i);
}
```

- Is the left-hand side of an assignment a variable of the correct type?

The production rule responsible for the assignment of variable is statement().

Where

```
<ID> <ASSIGN> expression() <SEMICOLON>
```

So to ensure that both the right and the left hand side are compatible, I created a check within the statement node.

```
if(type.equals("integer")) {
         // integer
         if(rhs.equals("integer"))
         {
             node.jjtGetChild(1).jjtAccept(this, data);
         }
         else if(rhs.equals("bool")) {
           System.out.println("expected integer got boolean");
         }
         //check that
       }else if(type.equals("boolean")) {
          if(rhs.equals("bool")){
             node.jjtGetChild(1).jjtAccept(this, data);
          }else if(rhs.equals("integer")){
             System.out.println("expected boolean got integer");
         }
       }
```

Here it checks for if the type of the current identifier is integer or boolean than it’s right hand side must also be the same.

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

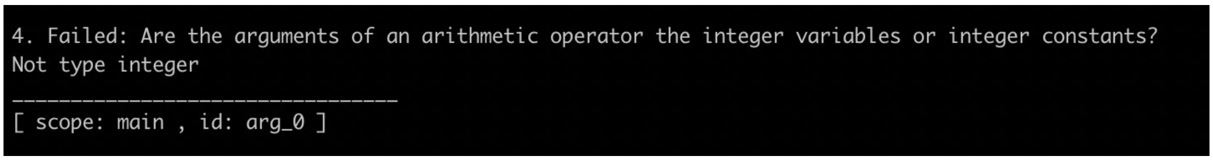
For this check, the purpose was to ensure that both of the arguments of an arithmetic operator(+/-) were both of type integer. Both the operators + and - are of node type **arith_ops** as seen below:

```
void binary_arith_op() : {Token token;} {
      token = <PLUS_SIGN> {jjtThis.value = token.image;} #arith_ops
    | token = <MINUS_SIGN> {jjtThis.value = token.image;} #arith_ops
}
```

The node itself is a child of statement, which is where I’ve placed this check. Below shows my solution:

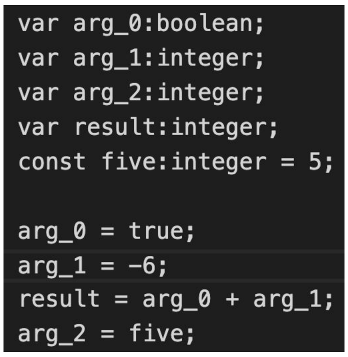
```
         int numChildren = node.jjtGetNumChildren();
        for (int i = 1; i < numChildren; i++) {
            String before = (String) node.jjtGetChild(i-1).jjtAccept(this, data);
            String childstr = (String) node.jjtGetChild(i).toString();
            String after = (String) node.jjtGetChild(i+1).jjtAccept(this, data);
          if(childstr.equals("arith_ops")){
            //lhs is id
            // get type of next arg
            String type1 = symbolTable.getType(before, scope);
            String type2 = symbolTable.getType(after, scope);
            if(!type1.equals(type2)){
              noWrongArithOps = false;
              // add whichever one doesnt have type integer to set.
              // put scope & id.
              if(type1.equals("integer")){
                // add type 2
                wrongArithOps.put(scope,after);
              }else if(type2.equals("integer")){
                // add type 1
                wrongArithOps.put(scope,before);
              }else{
                wrongArithOps.put(scope,before);
                wrongArithOps.put(scope,after);
              }

          }
```

The logic behind this was to check that if the statement node had an arith_ops child. If so, I would also retrieve the node before and after the arith_ops node. I then proceeded to get their ids and check if they were of type “integer”. If either wasn’t, then noWrongArgs would be set to false and the incorrect id would be added to the wrongArithOps Hashtable. Shown below is an example failed output:



```
4. Failed: Are the arguments of an arithmetic operator the integer variables or integer constants?
Not type integer
---------------------------------
[ scope: main , id: arg_0 ]
```

Corresponding test file:



```
var arg_0:boolean;
var arg_1:integer;
var arg_2:integer;
var result:integer;
const five:integer = 5;

arg_0 = true;
arg_1 = -6;
result = arg_0 + arg_1;
arg_2 = five;
```

● Are the arguments of a boolean operator boolean variables or boolean constants?

For this check, it was essentially the same as the previous and I check if binary_ops is a child of statement like I did with arith_ops. Instead, I needed to check that both of the arguments of a binary operator(and/or) were both of type boolean. Both the operators && and || are of node type **binary_ops** as seen below:

```
void and_or() : {Token token;} {
    token = <OR> {jjtThis.value = token.image;} #binary_ops
  | token = <AND> {jjtThis.value = token.image;} #binary_ops
}
```

```
}else if(childstr == "binary_ops"){
          // TODO make this to a function & only pass the rhs & excepted type - same
code as arithops
          String type1 = symbolTable.getType(before, scope);
          String type2 = symbolTable.getType(after, scope);
          if(!type1.equals(type2)){
            noWrongBinaryOps = false;
            // add whichever one doesnt have type integer to set.
            // put scope & id.
            if(type.equals("boolean")){
              // add type 2
              wrongBinaryOps.put(scope,after);
            }else if(type2.equals("boolean")){
              // add type 1
```

```
                wrongBinaryOps.put(scope,before);
             }else{
               wrongBinaryOps.put(scope,before);
               wrongBinaryOps.put(scope,after);
             }

         }
       }
```

- ## Is there a function for every invoked identifier?

The purpose of this check was to see that for every time that a function is invoked, that it is assigned to an identifier.

This was quite a straightforward check to put in place. Firstly, I ran a few different test cases and investigated the AST each time to understand the possible flows.

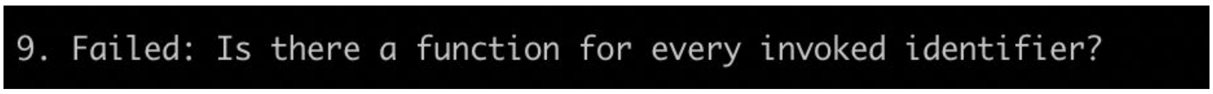
The correct AST structure was:

```
statement
 id
 function_return
  arg_list
```

I put added a check to ensure that there was more than 3 children present and added a simple check to see if the structure accepted functions called without an invoked id.

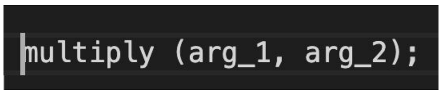
```
    String child1 = (String) node.jjtGetChild(0).toString();
    String child2 = (String) node.jjtGetChild(1).toString();
    String child3 = (String) node.jjtGetChild(2).toString();
     if(child1.equals("id") && child2.equals("arg_list") &&
child3.equals("function_call")){
      funcInvokedId = false;
    }
```

Failed sample output:



```
9. Failed: Is there a function for every invoked identifier?
```

Corresponding test file:



```
multiply (arg_1, arg_2);
```

- Does every function call have the correct number of arguments?

For this, I created a function within my symbol table class that would return the number of parameters for a given function identifier:

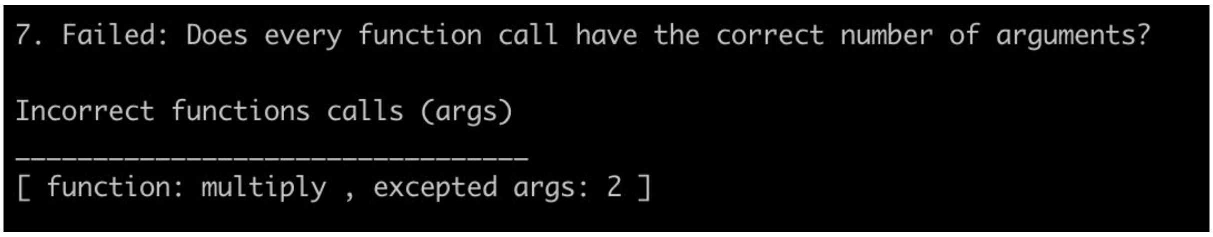
```
// gets how parameter there are based on func_id
public int getParameterCount(String func_id) {
  LinkedList<String> list = symbolTable.get(func_id);
  int count = 0;
  for(String id : list) {
    String value = valuesTable.get(id+func_id);
    if(value.equals("parameter")) {
      count++;
    }
  }
  return count;
  }
```

I could then call that function from within my semantic analysis class within the statement node. I had looked to if the current function was of type “function” and if so, I retrieved the number of children that it had.

```
          int parametersNumber = symbolTable.getParameterCount(id);
          //need to now access the child "arg_list" to examine it's no. args
          //in format
          //statement
          // function_return
          //   arg_list
          // func -> node.jjtGetChild(1) <- so get 1st child of that child
          int argumentsNumber= node.jjtGetChild(0).jjtGetChild(0).jjtGetNumChildren();
          if(!(parametersNumber == argumentsNumber)){
            equalParamArgs = false;
            // add the func id & expected
            wrongParamsArgs.put(id, String.valueOf(parametersNumber));
          }
```

All that was left was to compare the two numbers together. If they weren’t equal equalParmArgs would be set to false and the function identifier and it’s expected number of arguments would be added to a hash table too.

Sample failed output:



```
7. Failed: Does every function call have the correct number of arguments?

Incorrect functions calls (args)
----------------------------------
[ function: multiply , excepted args: 2 ]
```

- Is every variable both written to and read from?

I interpreted this check to ensure that all the variables assigned were also written to. I didn't include constants given as they are already written to.

I created this function within the semantic analysis class:

```java
public void isWrote(String scope, String id){
  scopes_vars = new ArrayList<String>();
  LinkedList<String> idList = symbolTable.getSymbolTable(scope);
  for(String s : idList){
    String vals = symbolTable.getValue(s, scope);
    if(vals.equals("variable")){
      scopes_vars.add(s);
    }
    }
  if(!scopes_vars.contains(id)){
    varsReadWrite = false;
    nonWroteVarsTable.put(scope,id);
  }
}
```

Here it would check that for a given scope and identifier, that the identifier is declared. I used that function within the Id node to then check if it was declared.

- Is every function called?

Here I added a method within my Symbol table class which iterates through, making lists of all of the defined functions.

```java
public ArrayList<String> getFuncTable(){
  Enumeration e = symbolTable.keys();
  while(e.hasMoreElements()) {
    try{
      String scope = (String) e.nextElement();
      String val = this.getValue(scope, scope);
      if(val.equals("function")){
        funcsTable.add(scope);
      }
    }catch(NullPointerException ex){
      //continue
    }
  }
  return funcsTable;
}
```

Within my Semantic Analysis class, I was then able to implement a function which could retrieve that list and check again the calledFuncList.

```
public boolean checkUncalledFuncs(){
  allFuncsCalled = true;
  allFuncsList = symbolTable.getFuncTable();
  for(String func : allFuncsList){
    if(!calledFuncList.contains(func)){
      allFuncsCalled = false;
      UncalledFuncList.add(func);
    }
  }
  return allFuncsCalled;
}
```

For calledFuncList, each function identifier was added within the statement node as a function can only be invoked within the statement production rule. So each time the current identifier was of type identifier, I added it to the list.

Intermediate Representation

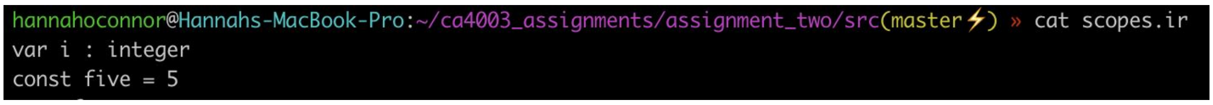
For this section, I created a class **IntermediateRep** which also implements the **HannahParserVistor** interface, therefore, also traversing the AST and performing operations on each of the given nodes.

The class itself has two functions printLabel() and printInstruction(). I added an extra argument to printLabel() where I could specify that I didn't want to include ':' after it's label. The class also includes the static int labelCount in order to keep track of the current line to print to.
All of the mentioned above, ensures that the resulting code in the ir file is correctly indented and inline.

Variables

```
  public Object visit(var_decl node, Object data){
      String id = (String) node.jjtGetChild(0).jjtAccept(this, data);
      String val = (String) node.jjtGetChild(1).jjtAccept(this, data);
      if(node.jjtGetParent().toString().equals("program")) {
        printInstruction("var " + id + " : " + val);
      }
```

```
        return node.value;
    }
```



```
hannahoconnor@Hannahs-MacBook-Pro:~/ca4003_assignments/assignment_two/src(master⚡) » cat scopes.ir
var i : integer
const five = 5
```

Constants

```
    public Object visit(const_decl node, Object data){
        String id = (String) node.jjtGetChild(0).jjtAccept(this, data);
        String val = (String) node.jjtGetChild(2).jjtAccept(this, data);
        printInstruction("const " + id + " = " + val);
        return node.value;
    }
```

Functions

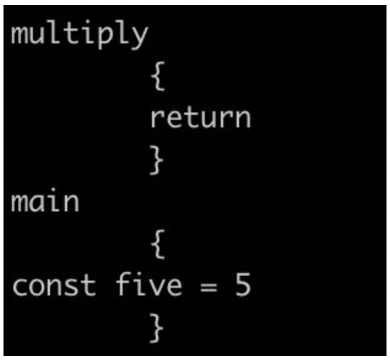
For function node, I ensured that upon retrieving the function identifier, I would add a **{** bracket and recursively call on its children until there are no more and end with a **}** bracket. I did the same practice with the main node also.

```
    public Object visit(function node, Object data){
        SimpleNode id = (SimpleNode) node.jjtGetChild(1);
        printLabel((String) id.value,true);
        System.out.println("\t{");
        int num = node.jjtGetNumChildren();
        for(int i = 0; i < num; i++) {
            node.jjtGetChild(i).jjtAccept(this, data);
        }
        System.out.println("\t}");
        return data;
    }
```

In order to get the number of args for a given function, I reused the same below recursive function like within my semantic check for function arguments (that is if it's invoked through an identifier). Here it accepts the child of the current function node, which should be a node a type arg_list, checking if it contains more children while increasing the count each level down the tree.

```
    private int getArgs(arg_list node, Object data){
        int count = 0;
        while(node.jjtGetNumChildren() != 0) {
            count++;
            System.out.println("parameter " + node.jjtGetChild(0).jjtAccept(this, data));
            // get next node
            node = (arg_list) node.jjtGetChild(1);
        }
        return count;
    }
```

Example file output:



```
multiply
          {
          return
          }
main
          {
const five = 5
          }
```

References & Sources

[1] https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm

- https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm
- https://tomassetti.me/parse-tree-abstract-syntax-tree/
- https://web.cs.wpi.edu/~cS544/PLTprojectast.html
-
- https://www.d.umn.edu/~rmaclin/cs5641/Notes/L15_SymbolTable.pdf
-
- http://www.cs.um.edu.mt/~sspi3/CSA2201_Lecture-2012.pdf
-
- https://stackoverflow.com/questions/10331641/how-to-use-visitor-for-an-ast-in-java
-
- https://www.geeksforgeeks.org/variable-scope-in-java/