Oracle Labs Morocco

Toy Language Compiler Project

Task Report

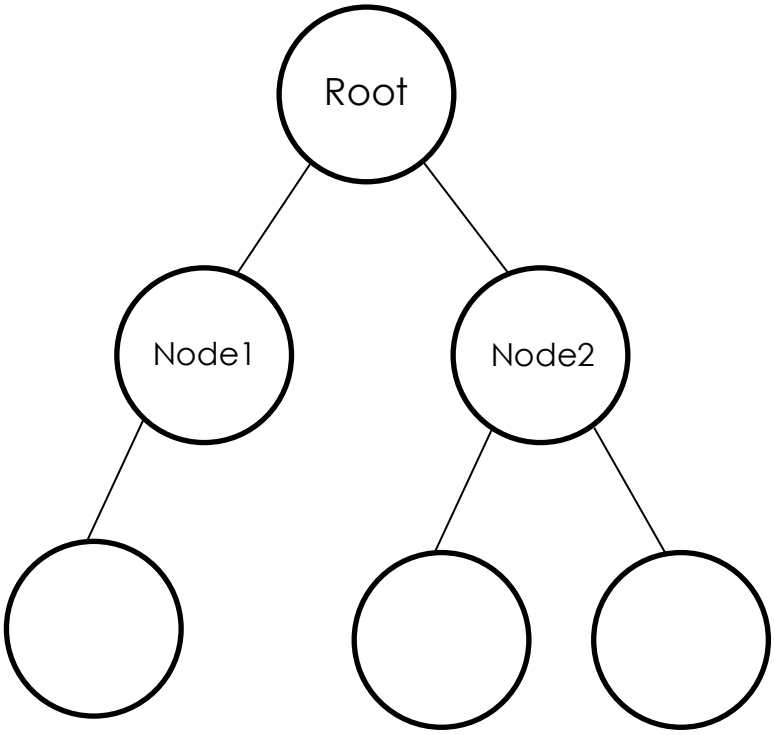
GHAILAN Oumaima

01/12/2019

Table of contents :

Assignment 1: AST Data Structure

i.



- An **Abstract Syntax Tree** is generally a tree, then what we need to represent in **JAVA** is a data structure that contains all the usual tree functionalities. We're going to make a class that represent the **root** and another that represents the **nodes**.

Nodes:

```
public class Nodes {
    public String name; //name is a declared as local variable
    public List<Nodes> children= new ArrayList<>();

    //generated constructor
    public Nodes(String name, List<Nodes> children) {
        super();
        this.name = name;
        this.children = children;
    }

    public Nodes() {
        super();
    }

}
```

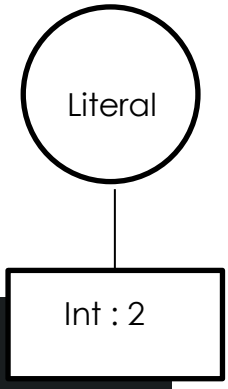
Root:

Our Root is always a function. A root is a node, its particularity is that it doesn't have a parent. As we're still going to introduce our data structure, we're going to leave this for later.

Type of data:

Literal:

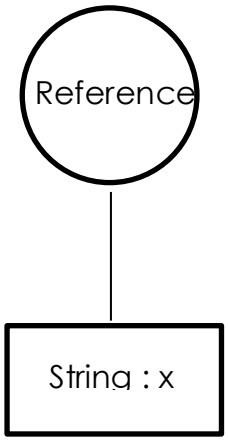
Literal is constant value. We're defining the ASTliteral Class bellow in such a way that it has a child with the String name "int: value" with value is our constant.




```
public class ASTliteral extends Nodes {
    int value;

    public ASTliteral(int value) {
        super();
        this.name="Lit";
        this.value = value;
        Nodes valueNode = new Nodes("int:"+value);
        this.children = Arrays.asList(valueNode);
    }

}
```

Reference:

A **reference** is a variable, in which we store a certain value, and we access to that value by its name instead of the value itself. This is how we implemented it as a class in JAVA.

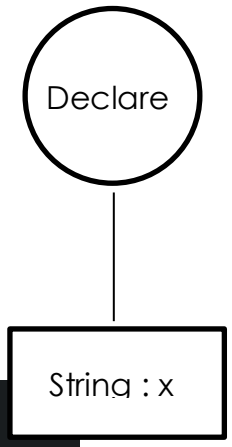



```
public class ASTreference extends Nodes {

    String var;

    public ASTreference(String var) {
        super();
        this.name="Ref";
        this.var = var;
        Nodes variableNode = new Nodes("String:"+ var);
        this.children = Arrays.asList(variableNode);
    }
}
```

Functionalities:

Declare:

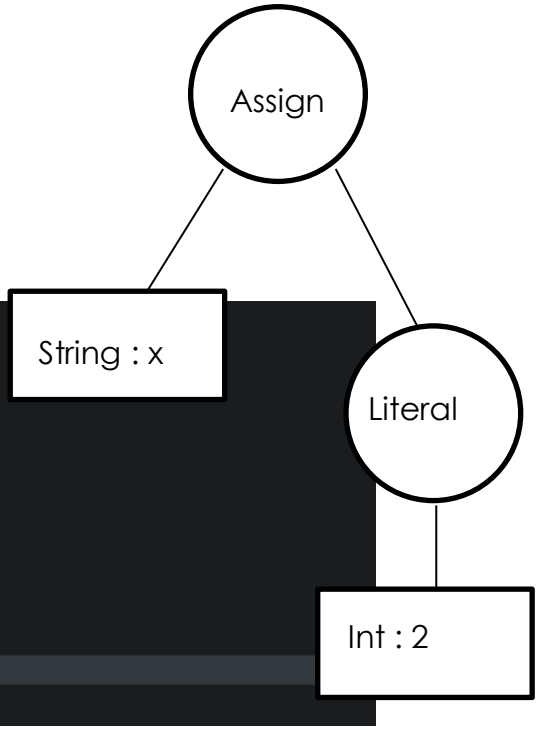
Our variable is a string which has a value of an int, so our **declaration** should store that variable in the **AST**.




```
public class ASTdeclare extends Nodes {
    String var;

    public ASTdeclare(String var) {
        super();
        this.name="Decl";
        this.var = var;
        Nodes variableNode = new Nodes("String:" + var);
        this.children = Arrays.asList(variableNode);
    }


}
```

Assign:

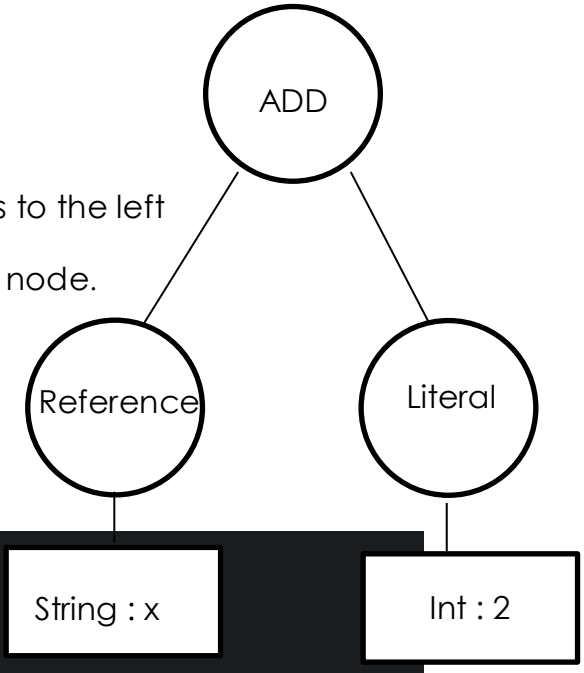
We **assign** an expression, which is on the right, to the variable on the left side of the node.




```
    String var;
    Nodes expression;

    public ASTassign(String var, Nodes expression) {
        super();
        this.name="Assign";
        this.var = var;
        Nodes variableNode = new Nodes(var);
        this.expression = expression;
        this.children = Arrays.asList(variableNode, expression);
    }

}
```

ADD:

- Our **ADD** node should add the right elements to the left elements. One of them can be another **ADD** node.



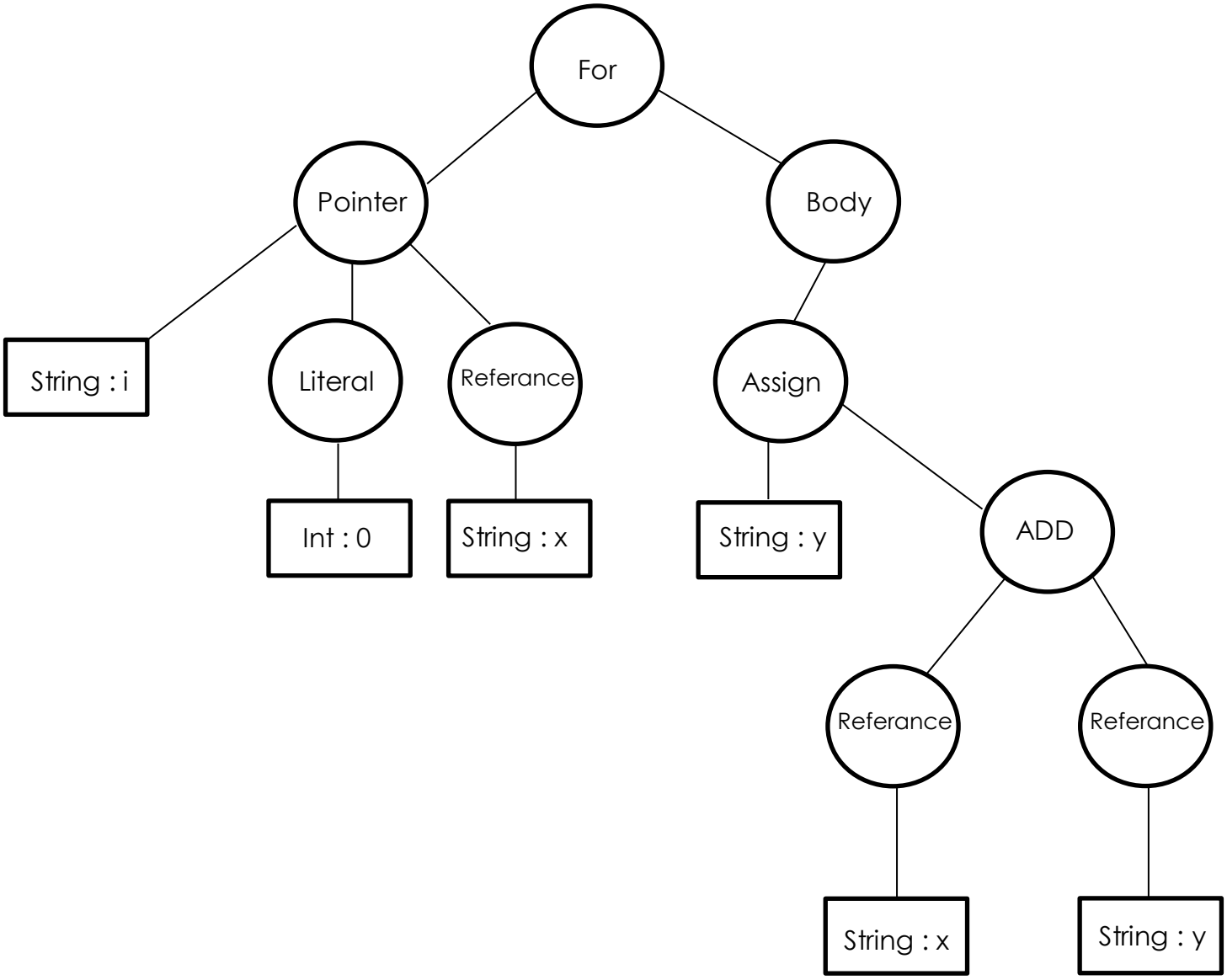

```
public class ASTadd extends Nodes{

    Nodes right;
    Nodes left;
    public ASTadd( Nodes left, Nodes right) {
        this.name="Add";

        this.right = right;
        this.left = left;
        this.children = Arrays.asList(left, right);
    }
```

ii. After we set our **AST** data structure (tree structure, assign, declare, ADD), now we will need to implement a **"FOR"** and **"RETURN"** functions as well as our main **"FUNCTION"** which should be also the root of our **AST**.

For simplicity, we divided the tree into multiple structures, which should be implemented first; see the following for more explanation.

FOR loop:

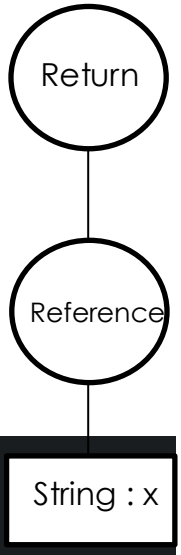
- The for loop generally contains a part of a pointer that indicates where our loop begins and ends, and another part for the body of our loop which has different instructions. So our implementation should contain two nodes; one for our pointer and another for our body.




```
public class ASTfor extends Nodes{

    public String i;
    public Nodes start;
    public Nodes stop;
    public List<Nodes> Body;


    public ASTfor( String i, Nodes start, Nodes stop, List<Nodes> Body) {

        this.i = i;
        this.start = start;
        this.stop = stop;
        this.Body = Body;
        this.name="for";
        Nodes pointerNode = new Nodes("iterator");
        List<Nodes> loopcondition ;
        loopcondition= Arrays.asList(pointerNode, start, stop);
        Nodes loopParametersNode = new Nodes("List", loopcondition);
        Nodes loopBodyNode = new Nodes("List", Body);
        this.children = Arrays.asList(loopParametersNode, loopBodyNode);


    }
}
```

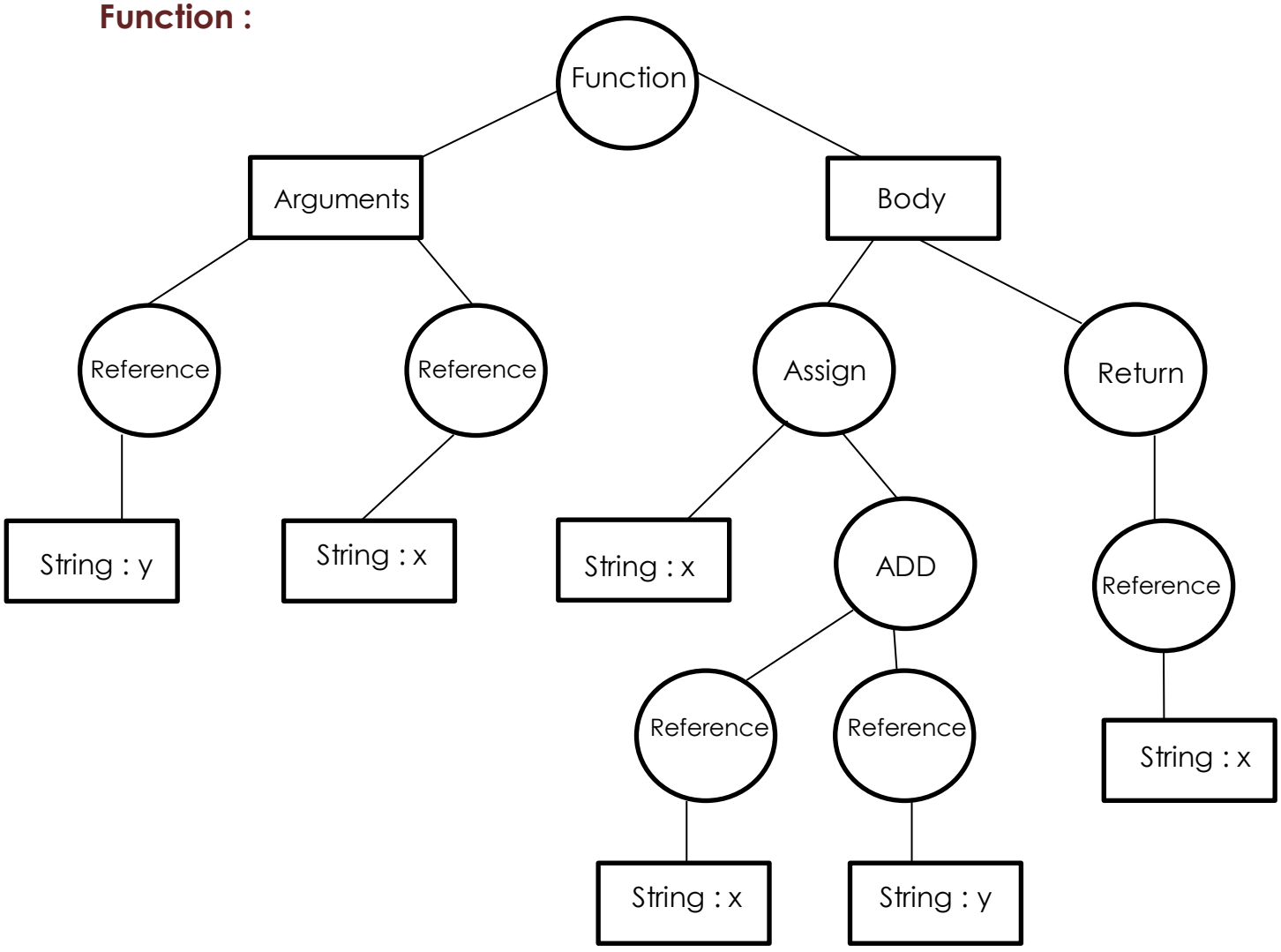
Return:

- It returns the value of a function.




```
public class ASTreturn {

    Nodes returnValue;

    public ASTreturn(Nodes returnValue) {
        super();
        this.returnValue = returnValue;
    }

}
```

Function :



- Our function has two nodes, one for arguments and another for the body. In our toy language, the function is the root of the AST. Here in our code, we made a class that realizes this functionality.

```
public class ASTfunction extends Nodes {

    public List<String> args;
    Nodes argsNode;
    public List<Nodes> body;
    public Nodes bodyNode;
    public String functionName;



    public ASTfunction(String functionName, List<String> args, List<Nodes> body) {
        super();
        this.name="Func";
        this.functionName = functionName;
        this.args = args;
        List<Nodes> argsNodes = new ArrayList<Nodes>();
        for (String argsName : args) {
            Nodes arg = new Nodes(argsName);
            argsNodes.add(arg);
        }
        this.argsNode = new Nodes("Args", argsNodes);
        this.body = body;
        this.bodyNode = new Nodes("Body", body);
        this.children = Arrays.asList(argsNode, bodyNode);

    }
```

Assignment 2: Simple Code Generation

i. What "mathematical function" is realized by the test2 function in the previous toy language program?

✓ The "mathematical function" realized by the test2 function is the **multiplication**.

*ii. Now we have to make our code generator (**NodetoCode**):*

Based on our structure, we decided to make a function that translate each node structure to a string, and then; the **FUNCTION(root)** browse each of its nodes to generate a code of the whole function.

Nodes:

```
public String NodetoCode() {
    return "";
}
```

Literal:

```
public String NodetoCode() {
    String s;
    s=""+value;
    return s;
    }
```

Reference:

```
@Override
public String NodetoCode() {

    String s;
    s=""+var;
    return s;
    }
```

Declare:

```
public String NodetoCode() {
    return "int " + var;
}
```

Assign:

```
@Override
public String NodetoCode() {
    return var + "= " + expression.NodetoCode();
    }
```

ADD:

```
@Override

public String NodetoCode() {
    return left.NodetoCode() + " + " + right.NodetoCode();
    }
```

For loop:

```
public String NodetoCode() {
    String b = new String();
    for (Nodes node:Body) {
        b += "\t\t" + node.NodetoCode();
        if (!b.substring(b.length()-1).equals(";") && !b.substring(b.length()-1).equals("}") )
        {
            b += ";";
        }
        b += "\n" ;
    }
    return "for (int " + this.i + " = "  + this.start.NodetoCode() + "; " + this.i + "<" +
    this.stop.NodetoCode() + "; " + this.i + "++) {\n" + b + "\t}" ;
}
```

Return:

```
@Override
public String NodetoCode() {
    return "return " +returnValue.NodetoCode();
    }
```

Function:

```
@Override
public String NodetoCode() {
    String argsString = new String();
    String bodyString = new String();

    Iterator<String> iterator = args.iterator();
    while (iterator.hasNext()) {
        argsString += "int " +iterator.next();
        if (iterator.hasNext()) {
            argsString += ", ";
        }
    }

    for (Nodes node:body) {
        bodyString += "\t" + node.NodetoCode();
        if (!bodyString.substring(bodyString.length()-1).equals(";") &&
                !bodyString.substring(bodyString.length()-1).equals("}") ) {
            bodyString += ";";
        }
        bodyString += "\n" ;
    }
    return "public static int " + functionName + "(" + argsString +") {\n" + bodyString +"}";
}
```

- Now that we have generated the code of each node, all that's left is to generate the code of the whole AST.

```
public void ASTCodeGenerator(ASTfunction f) {
    String JavaCode=new String();
    JavaCode=f.NodetoCode();
    System.out.println(JavaCode);
}
```

Test of function test2:

```
public class test2 {

    public static List<String> args = Arrays.asList("a");
    public static Nodes addref1 = new ASTreference("a");
    public static Nodes addref2 = new ASTreference("b");
    public static Nodes declare = new ASTdeclare("b");
    public static Nodes lit2 = new ASTliteral(0);
    public static Nodes assign1 = new ASTassign("b", lit2);
    public static Nodes add = new ASTadd(addref1,addref2);
    public static Nodes assign2 = new ASTassign("b", add);
    public static Nodes lit1 = new ASTliteral(0);
    public static Nodes endfor = new ASTreference("a");
    public static List<Nodes> Body = Arrays.asList(assign2);
    public static Nodes for1 = new ASTfor("i",lit1,endfor,Body);
    public static Nodes returnref = new ASTreference("b");
    public static Nodes ret= new ASTreturn(returnref);
    public static List<Nodes> body = Arrays.asList(declare,assign1,for1,ret);
    public static ASTfunction func1 = new ASTfunction("test2", args,body);



    public static void main(String[] args) {
        func1.ASTCodeGenerator();
    }
```

This is the output of our function test2:

```
public static int test2(int a) {
        int b;
        b= 0;
        for (int i = 0; i<a; i++) {
                b= a + b;
        }
        return b;
}
```

Assignment 3: Simple Optimization 1

➔ The for loop in this function doesn't have any body, which makes it useless and unnecessary for our function. So I would remove it from the code.

➔ What we did here, is that we made a function that remove all the empty loops, considering the next part which includes removing empty loops within loops.

```
public void removeuselessfor(Nodes node) {
    for (Iterator<Nodes> itererator = node.children.iterator(); itererator.hasNext(); ) {
        Nodes temp = itererator.next();
        if (temp instanceof ASTfor) {
            if (temp.children.size()==1) {
                itererator.remove();
            }
            else {
                removeuselessfor(temp.children.get(1));
                itererator.remove();
            }
            }
        }
    }
public ASTfunction Optimizer1() {
    removeuselessfor(this.bodyNode);
    return this;
}
```

- This is how we're going to run our test:

```java
public class test3 {
    public static List<String> args = Arrays.asList("a");
    public static Nodes lit1 = new ASTliteral(0);
    public static Nodes endfor = new ASTreference("a");
    public static List<Nodes> Body = Arrays.asList();
    public static Nodes for1 = new ASTfor("i",lit1,endfor,Body);

    public static Nodes returnref = new ASTreference("a");
    public static Nodes lit2 = new ASTliteral(1);
    public static Nodes add = new ASTadd(returnref,lit2);
    public static Nodes ret = new ASTreturn(add);

    public static List<Nodes> body = new LinkedList<Nodes>(Arrays.asList(for1,ret));

    public static ASTfunction func1 = new ASTfunction("test3",args,body);




    public static void main(String[] args) {
        System.out.println("before our optimization:");
        func1.ASTCodeGenerator();

        func1.Optimizer1();
        System.out.println("after our optimization:");
        func1.ASTCodeGenerator();


    }

}
```

Test of function test3:

- This is the output of our test3:

```
before our optimization:
public static int test3(int a) {
        for (int i = 0; i<a; i++) {
        }
        return a + 1;
}
after our optimization:
public static int test3(int a) {
        return a + 1;
}
```

Test of function test4:

- This is our structure :

```
public static List<String> args = Arrays.asList("a");           //parameters


//for2
public static Nodes lit3 = new ASTliteral(0);
public static Nodes endfor2 = new ASTreference("a");
public static List<Nodes> Body2 = Arrays.asList();
public static Nodes for2 = new ASTfor("i2",lit3,endfor2,Body2);


public static Nodes lit1 = new ASTliteral(0);
public static Nodes endfor = new ASTreference("a");
public static List<Nodes> Body = Arrays.asList(for2);
public static Nodes for1 = new ASTfor("i1",lit1,endfor,Body);

public static Nodes returnref = new ASTreference("a");
public static Nodes lit2 = new ASTliteral(1);
public static Nodes add = new ASTadd(returnref,lit2);
public static Nodes ret = new ASTreturn(add);

public static List<Nodes> body = new LinkedList<Nodes>(Arrays.asList(for1,ret));

public static ASTfunction func1 = new ASTfunction("test4",args,body);
```

- This is the output of our test4:

```
before our optimization:
public static int test4(int a) {
        for (int i1 = 0; i1<a; i1++) {
                for (int i2 = 0; i2<a; i2++) {
        }
        }
        return a + 1;
}
after our optimization:
public static int test4(int a) {
        return a + 1;
}
```

➔ Which means our optimization is working for this case too.

Assignment 4: Simple Optimization 2

The idea here, is to track the return values, so in case the return node has values that are independents from unnecessary declarations, we remove those unnecessary declarations. What we did here is we first created a list of return values and then compare them to a usecase of declared values, taking in consideration other functions like ADD and ASSIGN.

```
public List<String> returnedvaluesgenerator(){

    for (Iterator<Nodes> itererator = this.returnValue.children.iterator(); itererator.hasNext(
        Nodes temp = itererator.next();
        if (temp instanceof ASTadd) {
            returnedvalues.addAll(this.addvaluesnode((ASTadd) temp));

        }

        else {

            if (temp instanceof ASTreference) {
                returnedvalues.add(refvar((ASTreference) temp));
            }
        }

        }
        return returnedvalues;
    }
```

The usecase of declarations values.

```
public List<String> usecase= new ArrayList<>();
```

This is our function that deletes unnecessary declarations.

```
public void unnecessarydecldelete(Nodes node) {

    for (Iterator<Nodes> itererator = node.children.iterator(); itererator.hasNext(); ) {
        Nodes temp = itererator.next();
        if (temp instanceof ASTdeclare) {

            if (declarationtest((ASTdeclare) temp,node)==false) {
                itererator.remove();
                for (Iterator<Nodes> it = node.children.iterator(); itererator.hasNext(); ) {
                    Nodes t=itererator.next();
                    if (t instanceof ASTassign) {
                        if (((ASTassign) t).var==((ASTdeclare) temp).var) {
                            itererator.remove();

                        }
                    }
                }
            }

        }
    }

}
```

This is where we test if we should delete the declaration or not.

```
public boolean declarationtest(ASTdeclare declarationnode,Nodes node) {

    for (Iterator<Nodes> itererator = node.children.iterator(); itererator.hasNext(); ) {
        Nodes temp = itererator.next();
        if (temp instanceof ASTadd) {
            declarationnode.usecase.addAll(adddectest((ASTadd) temp));

        }
        if (temp instanceof ASTreturn) {
            List<String> L=returndectest((ASTreturn) temp);
            declarationnode.usecase.removeAll(L);

            if (L.equals(returndectest((ASTreturn) temp))) {

                return false;
            }
            else {
                return true;
            }

        }
    }
    return true;
```

This is the values used in any ADD node.

```
public List<String> addvalueslist(){
    if (this.right instanceof ASTreference) {
        addedvalues.add(refgetvar((ASTreference)this.right));

    }
        else {
            if (this.right instanceof ASTadd){
                ((ASTadd) this.right).addvalueslist();

            }
        }

return addedvalues;

    }
```

Test of function test5:

Now we will run our test5:

```
public static List<String> args = Arrays.asList("a");

public static Nodes declare = new ASTdeclare("b");
public static Nodes lit2 = new ASTliteral(1337);
public static Nodes assign1 = new ASTassign("b", lit2);
public static Nodes returnref = new ASTreference("a");
public static Nodes lit = new ASTliteral(1);
public static Nodes add = new ASTadd(returnref,lit);
public static Nodes ret = new ASTreturn(add);

public static List<Nodes> body = new LinkedList<Nodes>(Arrays.asList(declare,assign1,ret));
public static ASTfunction func1 = new ASTfunction("test5",args,body);

public static void main(String[] args) {
    System.out.println("before our optimization:");
    func1.ASTCodeGenerator();

    func1.Optmizer2();
    System.out.println("after our optimization:");
    func1.ASTCodeGenerator();
```

And this is the output of our test5:

```
before our optimization:
public static int test5(int a) {
        int b;
        b= 1337;
        return a + 1;
}
after our optimization:
public static int test5(int a) {
        return a + 1;
}
```