Surya Singh
ssing072
862033627

Neha Gupta
ngupt009
862049507

Phase 1 Report

Requirements:

Phase 1 is a typechecker for miniJava. When running a file the typechecker verifies at compile time that discrepancies do not occur during run-time. It will result in a pass or fail after reviewing all possible scenarios. The typechecker follows a typesystem. In this we extend two visitors in order to type check the java rules. If the types are implemented correctly in the java file, we print “`Program type checked successfully`” or if it doesn’t we have a “`Type error`”.

Design

Type checking

Our typechecker has two parts. First part uses the DepthFirstVisitor, in order to check a few rules of method declaration and id declaration along with building a symbol tree. Second Part extends the GJDepthFirst visitor in order to type check statements and expressions.

Project Overview and File Structure

- **Typecheck.java** - Main file which calls two of the parts in order to typecheck a java file.

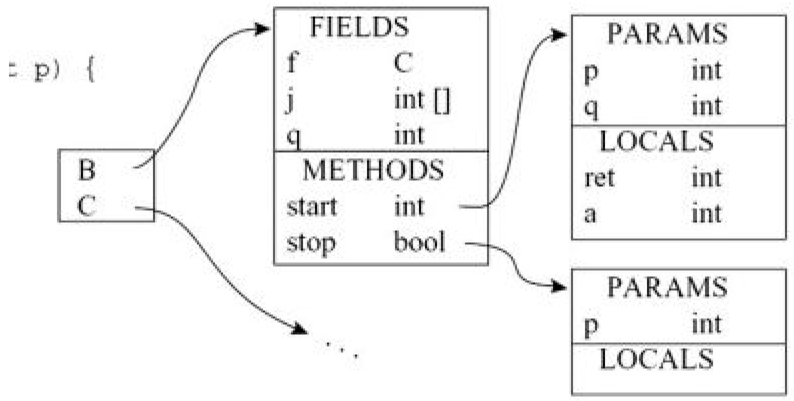
Created a package typechecker which holds inside:

- **FirstVisitor.java** - This is part 1 which extends the DepthFirstVisitor to create Symbol Table
- **SecondVisitor.java** - This is the second part which extends GJDepthFirstVisitor to type check statements.
- **Helper.java** - This file contains the helper functions such as distinct or getID
- **SymbolTable.java** - This is the symbol table implementation described later.
- **ClassSymbol.java** - This holds the implementation of a class symbol.
- **MethodSymbol.java** - This holds the implementation of a method symbol.
- **Symbol.java** - This file is used to create symbols.

Main File

In our typecheck file, we first get the root node(goal) of the file passed in, and then create a FirstVisitor(ft) object. Then we call the accept function on the root, passing in the Firstvisitor object. Which begins our part 1 of building the symbol table. Once building the symbol table is completed, we check if there were any errors, and if there were, then we have a type error. If there are no errors, then we move on to the next part which is type checking the statements and expressions. We create a SecondVisitor object, and this time when we call accept on the root, we pass in the second visitor object, along with the symboltable, that we fetch from the FirstVisitor object. Once the type checking completes, we check if there are any errors, if so then we have a type error. If there are no errors, that completes the type checking.

How does our SymbolTable work?

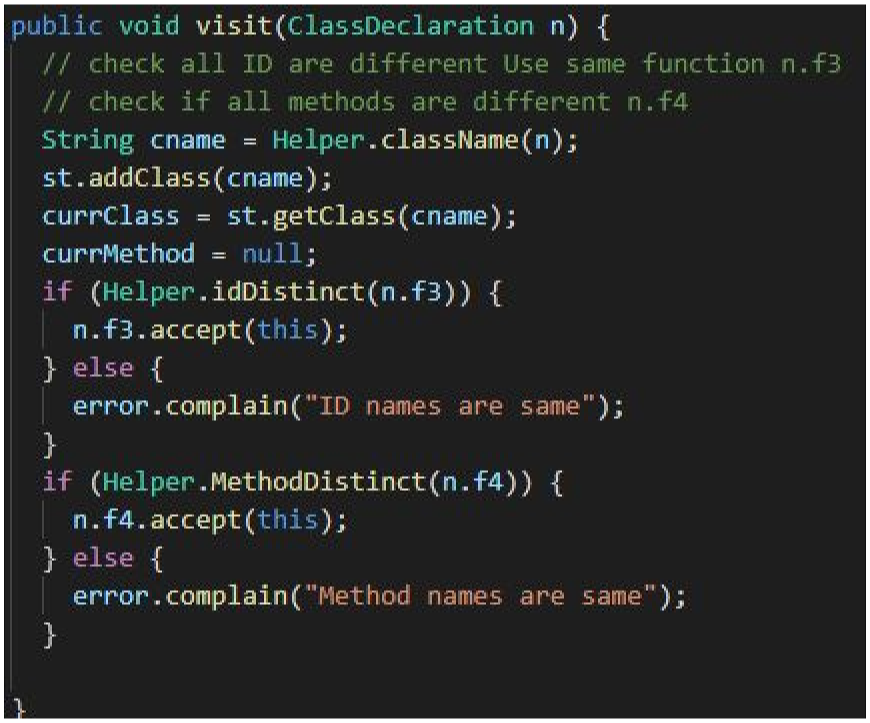


In order to build the SymbolTable, we used this implementation that's described in the book[1]. A class has fields and methods, and the methods have their own params and locals. So we decided to create objects for each of these. In a SymbolTable, it has a HashMap that maps a String to a ClassSymbol Object. Which allows us to map a class name to a ClassSymbol object. Our ClassSymbol object contains the name of the class, a hashmap for fields which is the variable declaration inside of classes, and a hashmap for methods, which maps a name of the method to its method symbol. Our MethodSymbol object contains the name of the method along with its type, a hashmap for the local variables and hashmap for parameter variables. Each of these hashmaps hash a Symbol(name) to a String(type). Each of these classes have their respective setter and getter functions which allow us to later traverse through the symbolTable.

First Part - FirstVisitor extends DepthFirstVisitor

In part 1 of the type checking phase we extended the DepthFirstVisitor class and implemented some of the functions that look at type rules:- Goal, MainClass, ClassDeclaration, ClassExtendsDeclaration, MethodDeclaration, VariableDeclaration, and more. We created visit methods of each of the types that needed to be processed and contained information for the symbol table. This class has a private SymbolTable, which it builds as the visitor extends deeper into the program. Using the accept(this) method we were able to extend specific methods and classes and look for symbols within them. This class uses the Helper function, in order to process a Node before it is implemented inside the symbol table. First we check for things such as if the classnames are all unique and then if all methods are unique and look for any errors within each implementation. After everything is correct with the names, we process the current thing to be implemented inside the symbol table. We also created two private variables, currClass and currMethod, in order to keep track of which class or which method is currently being processed, so that when we build the symbol table we know, which class to add the fields and methods and which method to add the params and locals.

Example of a function in FirstVisitor.java



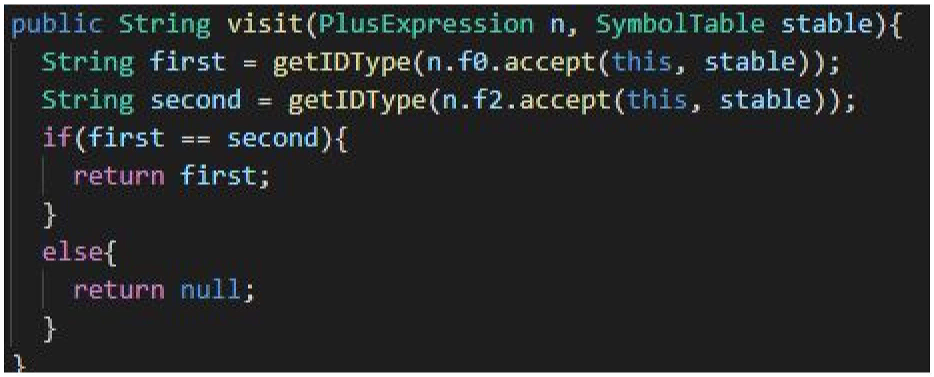
```java
public void visit(ClassDeclaration n) {
  // check all ID are different Use same function n.f3
  // check if all methods are different n.f4
  String cname = Helper.className(n);
  st.addClass(cname);
  currClass = st.getClass(cname);
  currMethod = null;
  if (Helper.idDistinct(n.f3)) {
    n.f3.accept(this);
  } else {
    error.complain("ID names are same");
  }
  if (Helper.MethodDistinct(n.f4)) {
    n.f4.accept(this);
  } else {
    error.complain("Method names are same");
  }


}
```

This is one example of how the functions in first visitor work, this function looks at the ClassDeclaration node type and before processing the methods and variables inside, we first check if all id are distinct which we call using Helper.idDistinct and pass in, and same thing for Methods we call Helper.methodDistinct and pass in the MethodDeclaration NodeList.

Second Part- SecondVisitor extends GJDepthVistor

In part 1 of the type checking phase we extended the DepthFirstVisitor class and implemented some of the functions that expand statements and check the type rules using the symbol table. This file also has a private currClass and currMethod, these will be used to keep track of which class the statement is in or the class and the method the statement is in. All of the visit functions in this file are a String type, this is done because later on we can return the type of an object using the string. For example, if there is an assignment statement, x = 1, we can visit the Identifier x = 1, which can get the Id of x using the symbol table and then return "int" for x and same thing goes for the 1, which is IntegerLiteral, and return int for that and then assignment will check if these two are correct. If at any point we don't find something in the symbol table or if a type doesn't match the rule, we have a boolean that is set to true, which we check later in the typecheck.java.

Example for a visit function within SecondVisitor.java



```java
public String visit(PlusExpression n, SymbolTable stable){
  String first = getIDType(n.f0.accept(this, stable));
  String second = getIDType(n.f2.accept(this, stable));
  if(first == second){
    return first;
  }
  else{
    return null;
  }
}
```

This is an example of the visit functions for the PlusExpression, where we look at the left side and the right side and call this getIDType, function which looks at the string passed in, and looks it up in the symbol table and if it exists it return the type of the object that exists in the current class or current method.

Helper Functions

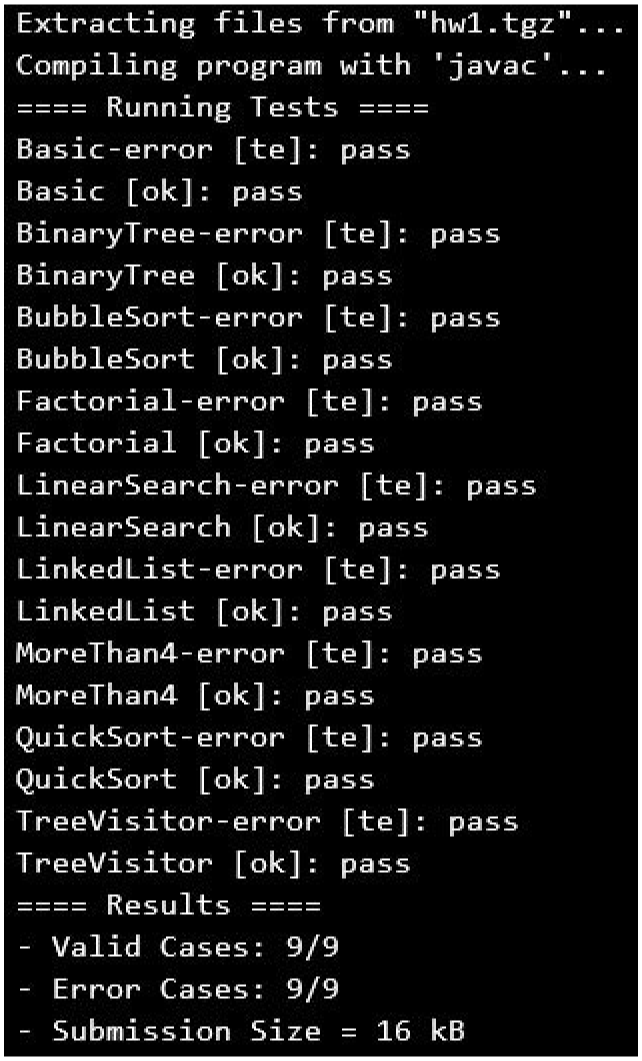
<u>Get name Functions: </u>The get name functions takes a type method, class, and returns a string. GetIntegerType, getID, getMethodName.

<u>ClassName: </u>The class name helper function checks first for the type of class declaration whether it is a main class, class, or class extends. Then once the type is determined the function returns the name.

<u>Distinct: </u>This reads in a list of class names and we use a hashSet to keep track of duplicate names. If any duplicate name occurs the function returns false and returns an error.
For the method, class, and id distinct we call the respective helper functions which return the name. Then check if the name already exists in the Set.
For parameter distinct we have a formal parameter and a nodeList optional types. The formal parameter refers to one parameter and nodeList optional is the possible different parameters in the List.

NoOverloading: This function checks the extended class and if there are no overloaded functions. It takes a string set parameter which is the method name of the super class and it has a NodeList which has the methods from the current class.

Testing and Verification



```
Extracting files from "hw1.tgz"...
Compiling program with 'javac'...
==== Running Tests ====
Basic-error [te]: pass
Basic [ok]: pass
BinaryTree-error [te]: pass
BinaryTree [ok]: pass
BubbleSort-error [te]: pass
BubbleSort [ok]: pass
Factorial-error [te]: pass
Factorial [ok]: pass
LinearSearch-error [te]: pass
LinearSearch [ok]: pass
LinkedList-error [te]: pass
LinkedList [ok]: pass
MoreThan4-error [te]: pass
MoreThan4 [ok]: pass
QuickSort-error [te]: pass
QuickSort [ok]: pass
TreeVisitor-error [te]: pass
TreeVisitor [ok]: pass
==== Results ====
- Valid Cases: 9/9
- Error Cases: 9/9
- Submission Size = 16 kB
```

We used the given test files to check our program, and oftentimes we also expanded on them in order to test specific functions as we created them.

References

1. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.461.6592&rep=rep1&type=pdf