Phase 1: Type-checking

Student #1 - Micah Galos | SID - 862082339
Student #2 - Najmeh Mohammadi Arani | SID - 862216499

Due: `Oct 24, 2021`, 11:59 PM

(1)Requirements and Specifications

The requirements needed for Phase 1 is creating a typechecker for various miniJava programs to be converted into MIPs instruction architecture. For type-checking we first build the AST for a given program using the built-in tools included which are generated by JTB and JavaCC. After generating the necessary tools, we extend the generated visitor classes to traverse the AST and perform the type checking utilizing the JTB and JavaCC classes.

The specifications in our design consists of using several HashMaps to store the data in different tables for any given java program as input. These inputs are then parsed depending on the object type and such types are inserted into different linked lists for distinct type comparison or direct instance checking. These checks are maintained in the second pass of the typechecker while the first pass scans the input programs.

(2)Design

Our type checker consists of two visitor extended classes: FirstPassVisitor and SecondPassVisitor--these two classes extend the GJDepthFirst visitor class generated by the JTB tool. First pass scans through the input program and builds the final symbol table, Table, via Hashmaps. From Table, the first pass determines whether the class, method, variable, and parameter ids are distinct based on their class object type.

```
@Override
public Table visit(MainClass n, Table t) {
    SymbolTable id = SymbolTable.setTypeId(CheckDistinct.classId(n));
    SubtypeTable.addSub(id, id); // throw id in case class is subclass or superclass

    // generate big symbol table
    Table tt = new Table(t, n);
    t.addTable(id, n, tt); // add main class id into big symbol table

    // Checks if each variable id are distinct
    NodeListOptional nloNode1 = n.f14; // ( VarDeclaration() )*
    boolean b = CheckDistinct.distinctVar(nloNode1);

    if(b) {
        nloNode1.accept(this, tt);
    }
    else {
        ProgramError.detectError();
    }
    return null;
}
```

Figure 2.1 - Preview FirstPassVisitor; Main Class Implementation

The second pass is the main course of the design; it determines whether there is subclassing, expressions, statements, among other program objects involved in the input program via a series of checks handled by a helper class to facilitate these checking the return types. These checks are referenced by the MiniJavaTypeSystem type rules and are handled by various flags and some use linked lists to read in parameters and methods.

```
// THE MOST CRITICAL TYPE CHECK, THE METHOD CALL IS CONTAINED BELOW

// p is an identifier of some class D
else {
    Collector c = t.find( SymbolTable.setTypeId(p.getType()) );
    // A,C |- p : D
    if (c != null) {
        Table tt = c.getTable();
        Identifier iNode = n.f2;
        SymbolTable m = SymbolTable.setTypeId(CheckDistinct.identifierId(iNode));
        Collector cc = tt.find(m);

        // methodtype(D, id) = (t_1', ..., t_n') --> t
        if (cc != null) { // non empty method type--has some list of types
            MethodDeclaration mdCast = (MethodDeclaration)cc.getTypeNode();
            MethodTable mt = CheckDistinct.methodType(mdCast);
            NodeOptional noNode = n.f4;
            boolean b2 = noNode.present();

            // A,C |- e_i : t_i
            if ( b2 ) {
                LinkedList<String> referenceParams = mt.getParams(); // t_i
                LinkedList<String> actualParams = new LinkedList<>(); // t_i'
                ExpressionList el = (ExpressionList)n.f4.node;

                actualParams.add(el.f0.accept(this, t).getType()); // Expression()

                for ( Enumeration<Node> e = el.f1.elements(); e.hasMoreElements(); ) {
                    ExpressionRest er = (ExpressionRest) e.nextElement();
                    actualParams.add(er.f1.accept(this, t).getType());
                }
                boolean b3 = ( referenceParams.size() == actualParams.size() );
```

Figure 2.2 - Preview Method Call Type Check in Second Pass

If these checks are not satisfied, then the typechecker sends a signal to let the system know there exists a type error somewhere in one the two visitor classes. For this version of the typechecker, the user only needs to know if the program being typechecked is successful or fails.

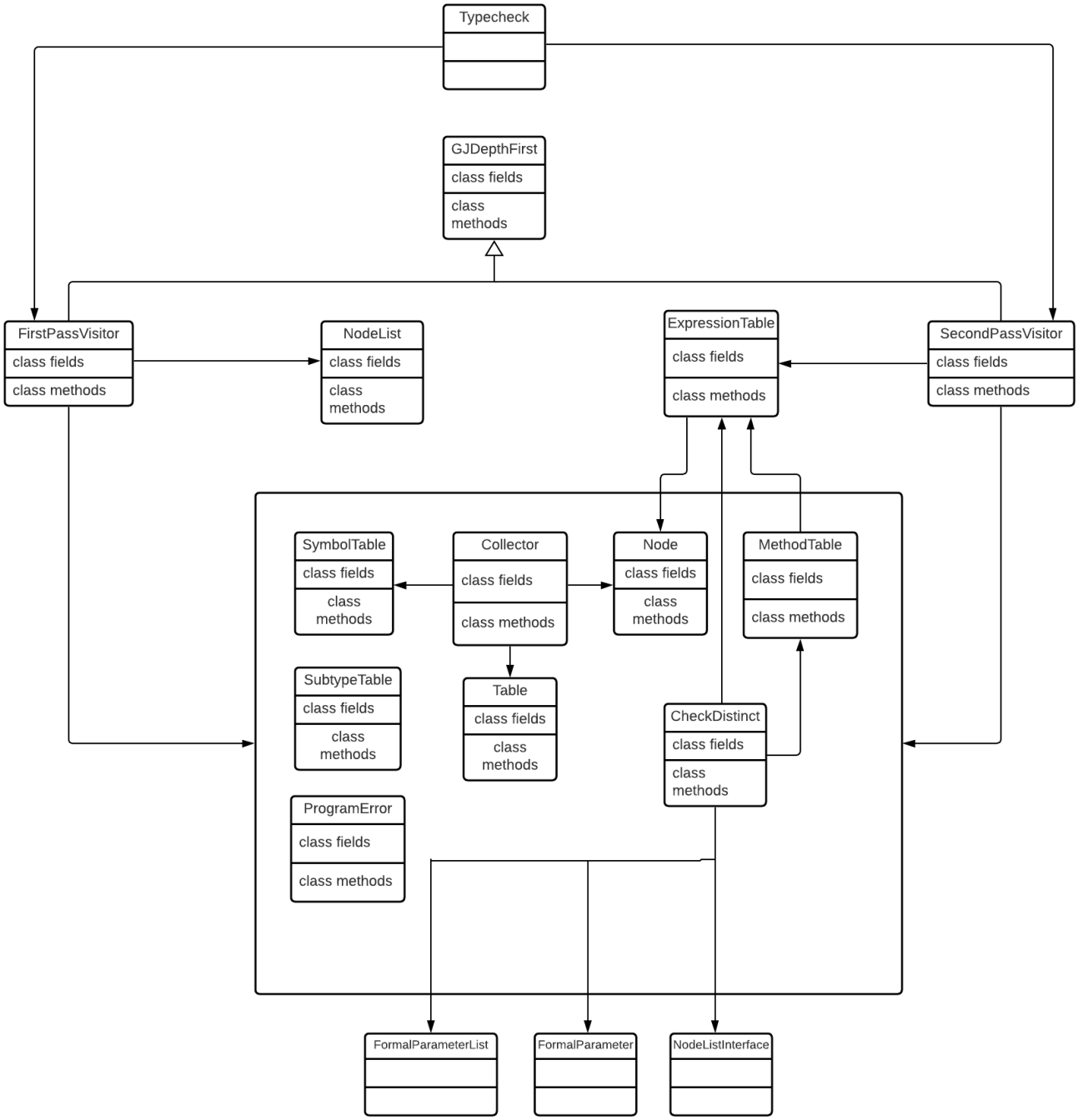

Figure 2.3 - UML Java Diagram

Here is a simple UML diagram of our typechecker design, while it is not in-depth in terms of what fields and methods are called. The basic run down is as shown in following the structure as where each class is communicating from one another.

(3)Testing and Verification

Utilizing the given test cases, our only requirement is to make sure whether the program outputs “Program type checked successfully” or “Type error” in the typecheck main class. As mentioned before, this version allows the user to only know whether the program is type checked successfully or not--assuming such program works as intended. In our design, the typechecker passes all the test cases shown below.

```
mgalo001@bolt $ ./run SelfTestCases/ ../hw1.tgz
===============
Deleting old output directory "./Output"...
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 = 48 kB
~/CS179E/Project/Phase_1/test
mgalo001@bolt $
```

Figure 3.1 - Auto Grader Test Cases

(4)Technical Problem(s)

No doubt in our minds the planning phase of this project was the most difficult. While understanding the MiniJavaTypeSystem did help with understanding the second pass in flag checking for return types, the outside work involved was rigorous.

Upon realizing the scope of the second pass is a series of flag checks was simple, yet tedious. The Wednesday discussions were helpful in getting an idea, but between the two of us trying to approach ideas, it was hard to grasp at first glance-especially with the first session with the TA. Continuously asking questions to Prof. Lesani helped narrow our ideas, alleviating any confusion or concerns we had during the planning process.

Figuring out what data structure, while hypothetically straightforward in idea, was difficult in implementing because of our somewhat deep understanding in what functionalities are utilized in the java libraries. We gave ourselves the first two weeks of the quarter to understand how to store and handle the data via a hashmap for each table. Having the first pass read and store the keys and their values into the big symbol table and making sure each feature is distinct was the “easy” part.

However, creating the checks to make sure the data stored in the hashmap was tedious and difficult during the debug process. The java libraries definitely helped when utilizing hashmaps and linked lists for our type checking purposes. Parsing each feature and their object types was difficult to overcome--micromanaging what object type each are instanced from and confirming the data they are from the correct object class.

The debug process consisted of us having class and null point exception errors which made us very confused as to why some of our objects were sending a null value especially during the second pass. To which it was our fault for separating some classes into their own folders and ended up losing the data upon calling an object’s method.

Nonetheless, we overcame these challenges faced before us and were able to complete a working product of a typechecker. Albeit with the extra week of the original submission, but that extra time we did quality assurance to make sure any changes we made still maintained the required specifications.