Phase 2: Intermediate Code Generation for Object-oriented Languages

Student #1 - Micah Galos | SID - 862082339
Student #2 - Najmeh Mohammadi Arani | SID - 862216499
Student #3 - Steven Strickland | SID - 862155853

Due: Nov 7, 2021, 11:59 PM

1. Requirements and Specifications

The requirements for Phase 2 is implementing a translator of input java programs into the vapor language. The vapor language consists of assembly-like language and we have to generate code into a set of functions and data segments. The main objective is utilizing objects and mapping them to memory and object-oriented method calls to function calls.

Specifications for Phase 2 utilize object oriented programming in mapping generated objects from pre-existing typechecker and new additions to proceed with the code generation. The newly created classes to map into memory consist of: ClassRecord and VTable. The ClassRecord maps field names of the class to their offsets while VTable holds the locations of those fields and methods.

From Phase 1, we utilize the FirstPassVisitor to collect the information and insert them into the big symbol table, Table. For the code generation into the Vapor language, we extend GJDepthFirst with another visitor called VaporVisitor. In VaporVisitor, we take the Translator and Table objects and pass them into the TranslateTable. TranslateTable is the bridge to accept the GJDepthFirst grammars and append into the translated Vapor assembly. Checking type ids uses the SymbolTable to determine what kind of type are the field variables and make sure the id exists in the collector.

For the outputting into Vapor's assembly-like language, we created: BuiltInOps, Jump, Translator, and VaporCheck-- appending lines of code via the VaporVisitor during the translation process. Each of these class's main features will be explained at the Design section.

2. Design

a. Phase 1 Updates

i. Table

```
public class Table {
    // Data Structure
    private LinkedHashMap<SymbolTable, Collector> Table = new LinkedHashMap<>();
```

Changed the table data structure from a HashMap to a LinkedHashMap to maintain order for each class's method received in the collector as well as match with the VTable. Since the original implementation order did not matter in type-checking, there were no negative repercussions. Upon converting into Vapor, order did have a factor when translating such that the unordered list in a HashMap may produce incorrect outputs.

```
public Collector findLocal(SymbolTable st) {
    Collector c = Table.get(st);
    boolean b1 = (c != null); // non-empty collector

    if (b1){
        return c;
    }

    else{
        return null;
    }
}
```

Added another method in Table to search for a method's local variables in SymbolTable. Lacking this functionality would result in the translation phase to assume the collector is not empty containing labels and would output the original variable it was assigned in the input java file and not its translation variable with offset.

b. Phase 2

i. ClassRecord

```
public class ClassRecord {
    // Create Hashmap to record classes and their field names to offset
    private LinkedHashMap<String, LinkedList<String> > classRecord = new LinkedHashMap<>();

    public int getVarOffset(String c, String v) {
        // Receive last recorded values of the field offset from VTable
        int offsetVar = classRecord.get(c).lastIndexOf(v);
        return offsetVar * 4 + 4;
    }

    public int getClassSize(String c) {
        boolean b = classRecord.containsKey(c);

        if( b ){ // When class keys exist in the hashmap
            int clazzSize = classRecord.get(c).size();
            return clazzSize * 4 + 4;
        }
        else{ // Initially there is nothing in the hashmap
            return 4;
        }
    }
```

First we initialize with an offset of +4 because we start the stack at some "null" entry point when keeping track of the ordered list of classes and methods contained once visited in the VaporVisitor at Translation. offsetVar and clazzSize is the index order at which they entered the stack, so they take up that entry space above the initial null entry. So, if a class contains methods in an ordered list upon visit, then the order of the class will be recorded at the index along with the methods in the linkedhashmap.

ii. VTable

```
// Create Hashmap to record classes and their vtable
// each entry in vtable has class mapped with their own method and offsets
private LinkedHashMap<String, LinkedHashMap<String, String> > vtable = new LinkedHashMap<>();

public int getMethodOffset(String c, String vt) {
    int offsetMethod = new LinkedList<>(vtable.get(c).keySet()).indexOf(vt);

    return offsetMethod * 4;
}
```

Similar to the ClassRecord, VTable handles their own offset calculation. The VTable handles local variables contained within the methods of the class visited in the ClassRecord. The VTable is not handled by a stack in the visitor, so we keep track of the offset by "reference" in memory at which they were initialized in the ClassRecord's index.

iii. Translator

```
public void appendPrintIntS(Variable v) {
    String Value = v.toString();
    appendNewLine("PrintIntS(" + Value + ")");
}

public void appendAssignment(Variable Location, Variable rhs) {
    String Value = rhs.toString();
    appendAssignment( Location, Value );
}

public void appendAssignment(Variable lhs, String Value) {
    String Location = lhs.toString();
    appendNewLine(Location + " = " + Value);
}

public void appendIf(Variable v, Jump j){
    String Value =  v.toString();
    String CodeLabel = j.toString();
    appendNewLine( "if " + Value + " goto :" + CodeLabel );
}

public void appendIfZero(Variable v, Jump j){
    String Value =  v.toString();
    String CodeLabel = j.toString();
    appendNewLine( "if0 " + Value + " goto :" + CodeLabel );
}

public void appendGoto(Jump j) {
    String CodeLabel = j.toString();
    appendNewLine( "goto :" + CodeLabel );
}

public void appendJump(Jump j) {
    String CodeLabel = j.toString();
    appendNewLine(CodeLabel + ":");
}

public void appendReturn() { appendNewLine("ret"); }

public void appendReturn(Variable v) {
    String Value = v.toString();
    appendNewLine( "ret " + Value );
}
```

Translating ClassRecord and VTable class objects, we append these labels along with their Value and CodeLabel received from VaporVisitor. We also maintain the variable and jump labels by passing them into these methods

and translate into Vapor assembly. The whole implementation is not shown--this is the general idea when translating MiniJava to Vapor.

iv. TranslateTable

```
public class TranslateTable {
    private final Table table;
    private final Translator translator;

    public TranslateTable(Table s, Translator t) {
        table = s;
        translator = t;
    }

    public Table getTable() {
        return table;
    }

    public Translator getTranslator() {
        return translator;
    }
}
```

TranslateTable is utilized in VaporVisitor and VaporCheck, it is a reference to the class as an object in order to accept the grammars in the Visitor at the translation phase. Having a reference to this class ensures the translation process maintains the ordered list of names in the Table and labels when we append MiniJava into Vapor assembly.

v. VaporVisitor

Every related class, method, expression, statement, and array objects from GJDepthFirst are handled by using objects generated by Table, Translator, ClassRecord, VTable, Variable, and TranslateTable. Depending on the grammar type accepted into the visitor, we append their respective functionalities by passing in the parameters and values assigned.

For example, an expression utilizing the Built-In operations consists of the parameter and value being performed. Jump conditions like if-else, and comparison, and while functionalities append the top and end labels depending if the label jumps to a null pointer or not.

vi. VaporCheck

VaporCheck is associated with VaporVisitor such that it checks if object variables are stored in memory or are a function/method call. If the variable reaches a null pointer, then we append an if label to a null label with an error message at translation. For arrays, we check if the array index is out of bounds and proceed to handle it by appending in a similar fashion at translation.

vii. BuiltInOps

```
public static String Add(String a, String b) {
    String addStr = "Add(" + a + " " + b + ")";
    return addStr;
}

public static String Sub(String a, String b) {
    String subStr = "Sub(" + a + " " + b + ")";
    return subStr;
}

public static String MulS(String a, String b) {
    String mulsStr = "MulS(" + a + " " + b + ")";
    return mulsStr;
}

public static String Eq(String a, String b) {
    String eqStr = "Eq(" + a + " " + b + ")";
    return eqStr;
}

public static String Lt(String a, String b) {
    String ltStr = "Lt(" + a + " " + b + ")";
    return ltStr;
}

public static String LtS(String a, String b) {
    String ltsStr = "LtS(" + a + " " + b + ")";
    return ltsStr;
}

public static String HeapAllocZ(int n) {
    String intLabel = Integer.toString(n);
    String heapStr = "HeapAllocZ(" + intLabel + ")";
    return heapStr;
}

public static String AllocArray(Variable size) {
    String intLabel = size.toString();
    String allocStr = "call :AllocArray(" + intLabel + ")";
    return allocStr;
}
```

Vapor supports the set of these built-in operations, the class itself returns these strings methods as output when receiving variables a and b from VaporVisitor for their specific arithmetic functionality.

viii. Jump

```
// Count is incremented globally, starts at 1
private int jumpCount[] = {1,1,1,1,1}; // {null, bounds, if, while, and}
```

```
public Jump appendNull() {
    Jump j = new Jump("null%d", jumpCount[0]++);
    return j;
}

public Jump appendBounds() {
    Jump j = new Jump("bounds%d", jumpCount[1]++);
    return j;
}

// Return these after initializing the jump label chain
public Jump appendIfEnd() {
    Jump j = new Jump("if%d_end", jumpCount[2]++);
    return j;
}

public Jump appendWhileEnd() {
    Jump j = new Jump("while%d_end", jumpCount[3]++);
    return j;
}

public Jump appendAndEnd() {
    Jump j = new Jump("ss%d_end", jumpCount[4]++);
    return j;
}

// Initialize these labels to begin jump label chain
public Jump appendIfElse() {
    Jump j = new Jump("if%d_else", jumpCount[2]);
    return j;
}

public Jump appendWhileTop() {
    Jump j = new Jump("while%d_top", jumpCount[3]);
    return j;
}

public Jump appendAndElse() {
    Jump j = new Jump("ss%d_else", jumpCount[4]);
    return j;
}
```

We keep track of every jump label in an int array and increment each entry globally for every recursive call in VaporCheck and VaporVisitor.

ix. Variable

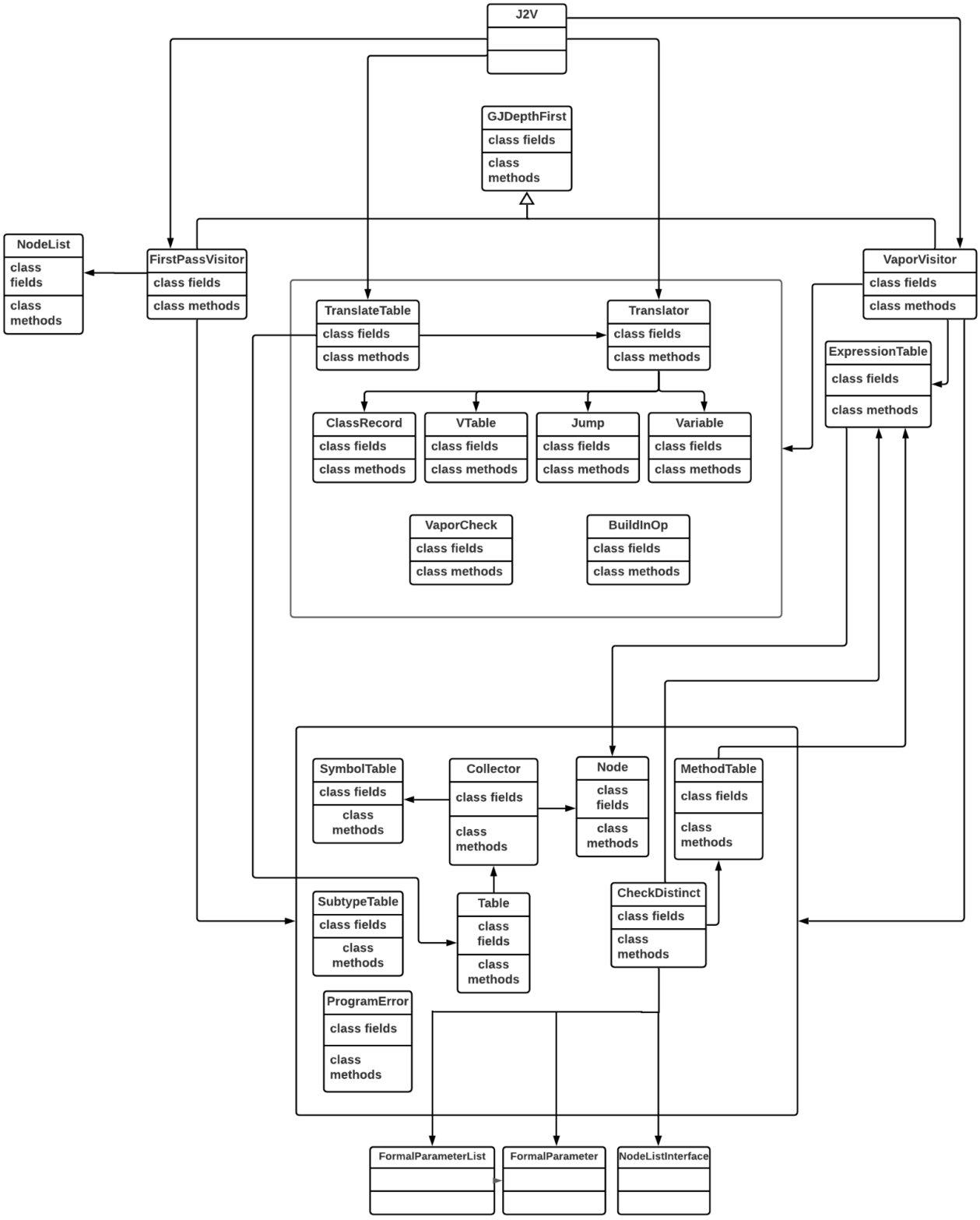
```
private boolean memCallFlag[] = {false, false};
// memCallFlag[0] - variable is in memory
// memCallFlag[1] - variable is parameter in method call
```

```
private Variable(int n, String b, boolean[] d) {
    number = n;
    root = b;
    this.memCallFlag = d;
}
```

```
public Variable memVar(){ // variables have a reference in memory
    Variable refVar = new Variable(this);
    return refVar;
}

public Variable appendFuncCall(String fc) {
    boolean[] flag = {false, true};
    Variable funcCall = new Variable(minusFlag, fc, flag);
    return funcCall;
}

public Variable appendTempVar() {
    boolean[] flag = {false, false};
    Variable tempVar = new Variable(varCount++, null, flag);
    return tempVar;
}

public Variable appendConstant(String c) {
    boolean[] flag = {false, false};
    Variable constVar = new Variable(minusFlag, c, flag);
    return constVar;
}

// Methods in Vapor contain "this" as a reference to the original variable id (this...) or [this...]
public Variable appendThisVar() {
    boolean[] flag = {false, false};
    Variable thisVar = new Variable(minusFlag, "this", flag);
    return thisVar;
}

// each variable in the method has an offset in memory [this+Offset]
public Variable appendThisVar(int offset) {
    boolean[] flag = {false, false};
    Variable thisVar = new Variable(offset, "this", flag);
    return thisVar;
}

public Variable appendLocalVar(String var) {
    boolean[] flag = {false, false};
    Variable localVar = new Variable(minusFlag, var, flag);
    return localVar;
}

public Variable appendLocalVar(int offset, String var) {
    boolean[] flag = {false, false};
    Variable localVar = new Variable(offset, var, flag);
    return localVar;
}
```

The variables in this class are handled by signal flags. The main flag, memCallFlag, holds two signals--memCallFlag[0] determines if the variable is in memory while memCallFlag[1] is when the variable is a method call. Depending on these two signals determine what kind of variable: a literal, local variable, or a variable of type “this” in the vapor language. TempVar is the [t.X] variable in Vapor, such that “X” is the incremented value for variable assignments. In addition, some methods which determine what variables are referenced in memory utilize a minusFlag which indicates a non-incrementing variable. Non-incrementing variables are read from the Translator which received the original table. The original table uses the labels of the MiniJava program, but we want the translation variables in the form of [t.x] or t.x.

c. UML Diagram



3. Testing and Verification