CS179E Compiler Project

By Kris Tsuchiyama

Phase 1: Type-Checking

Goal: Given a program, the type checker checks at compile time that a type mismatch does not appear based on the type system

Main Function

```java
import syntaxtree.*;
import java.io.*;
import java.util.Scanner;

public class Typecheck {
    public static void main(String[] args) {
        /*Scanner sc = new Scanner(System.in);
        System.out.println("Printing the file passed in:");
        while(sc.hasNextLine()) System.out.println(sc.nextLine());
         */

        InputStream fileStream = System.in;
        new MiniJavaParser(fileStream);
        SymbolTableVisitor sv = new SymbolTableVisitor();
        TypeVisitor tv = new TypeVisitor();
        try {
            Node root = MiniJavaParser.Goal();
            root.accept(sv);
            root.accept(tv, sv);
            tv.checkTypeError();
            //sv.print();
            if(!tv.hasError) {
                System.out.println("Program type checked successfully");
                System.exit( status: 0);
            }
            System.exit( status: 1);
        } catch (Exception e) {
            e.printStackTrace();
            //System.out.println("Type error");
        }
    }
}
```

Visitor Pattern

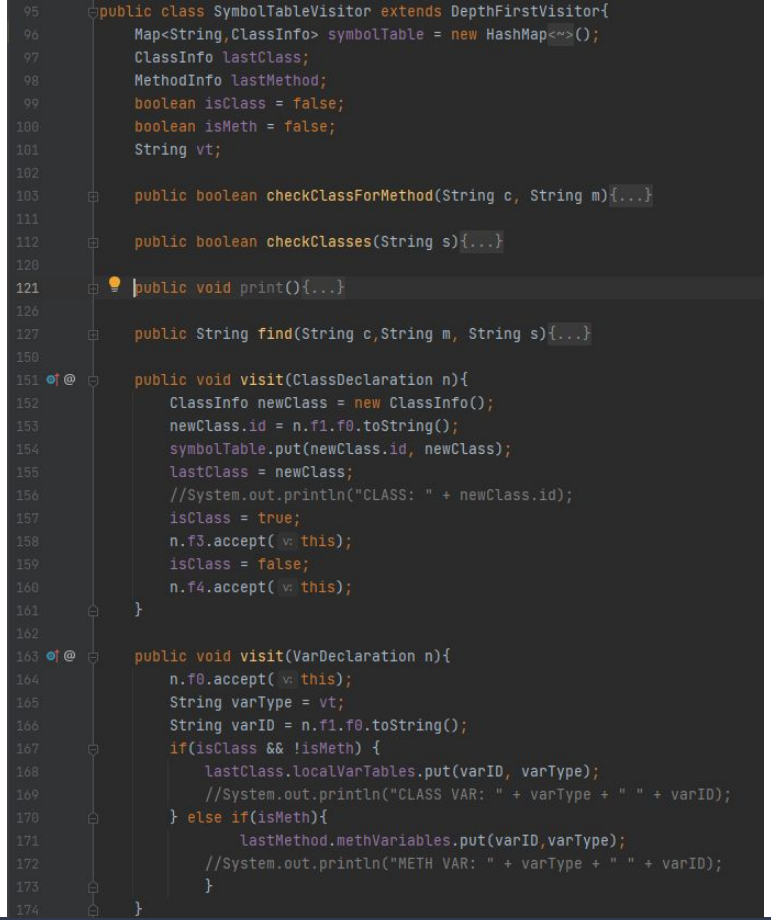
```java
import syntaxtree.*;
import visitor.DepthFirstVisitor;
import java.util.*;

class MethodInfo{...}

class ClassInfo{...}

public class SymbolTableVisitor extends DepthFirstVisitor{
    Map<String,ClassInfo> symbolTable = new HashMap<~>();
    ClassInfo lastClass;
    MethodInfo lastMethod;
    boolean isClass = false;
    boolean isMeth = false;
    String vt;

    public boolean checkClassForMethod(String c, String m){
        for(Map.Entry<String,ClassInfo> entry : symbolTable.entrySet()){
            if(entry.getKey().equals(c)){
                return entry.getValue().checkMethods(m);
            }
        }
        return false;
    }

    public boolean checkClasses(String s){
        for(Map.Entry<String,ClassInfo> entry : symbolTable.entrySet()){
            if(entry.getKey().equals(s)){
                return true;
            }
        }
        return false;
    }
    public void print(){
        for(Map.Entry<String,ClassInfo> entry : symbolTable.entrySet()){
            entry.getValue().print();
        }
    }

    public String find(String c,String m, String s){
        String result = "error";
        for(Map.Entry<String,ClassInfo> entry : symbolTable.entrySet()){
            if(entry.getKey().equals(c)) {
```

```java
import syntaxtree.*;
import visitor.DepthFirstVisitor;
import visitor.GJDepthFirst;

public class TypeVisitor extends GJDepthFirst<String, SymbolTableVisitor> {
    boolean hasError = false;
    String errorMsg = "";
    String currentClass = "";
    String currentMethod = "";


    public void checkTypeError() {
        if(hasError) {
            System.out.println("Type error");
        }
    }

    public String visit(ClassDeclaration n, SymbolTableVisitor s){
        currentClass = n.f1.f0.toString();
        n.f3.accept( v: this, s);
        n.f4.accept( v: this, s);
        return n.f1.f0.toString();
    }

    public String visit(MethodDeclaration n, SymbolTableVisitor s){
        currentMethod = n.f2.f0.toString();
        String methType = n.f1.accept( v: this, s);
        n.f4.accept( v: this, s);
        n.f7.accept( v: this, s);
        n.f8.accept( v: this, s);
        String returnType = n.f10.accept( v: this, s);

        if(methType.equals(returnType)){
            //System.out.println(methType + ", " + returnType + " : " + currentMethod + " : " + currentClass);
            currentMethod = "";
            return returnType;
        } else{
            //System.out.println(methType + ", " + returnType + " : " + currentMethod + " : " + currentClass);
            hasError = true;
            return "error";
        }
    }
```

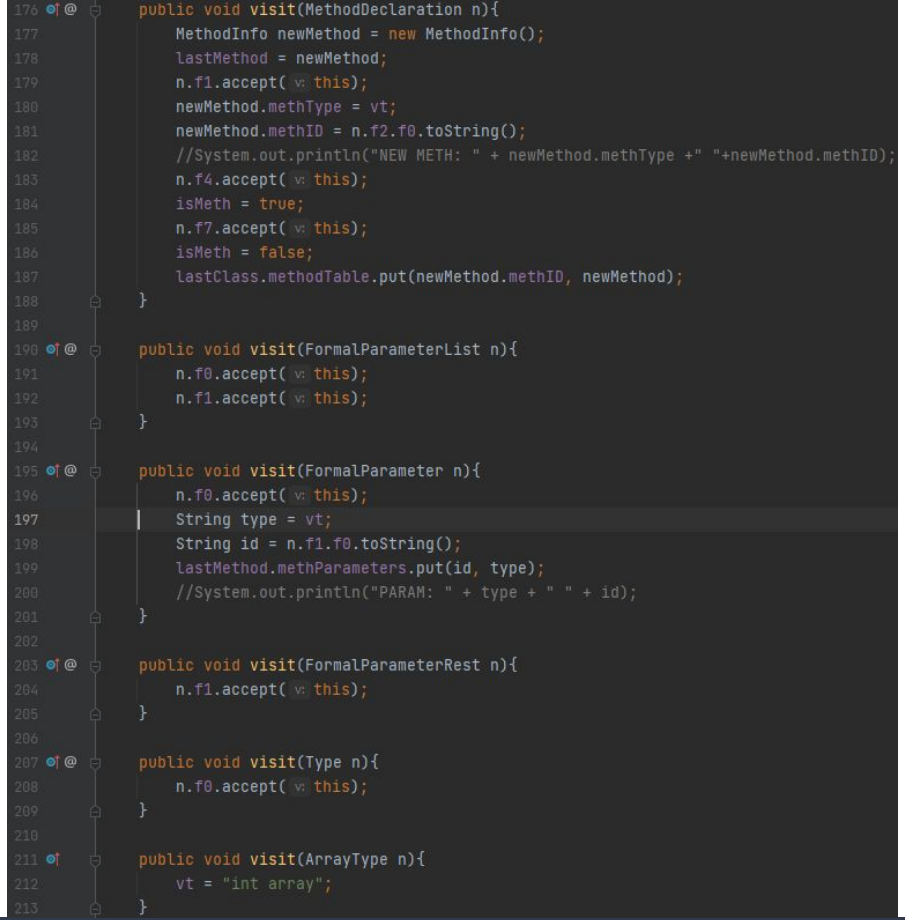
Symbol Table Visitor

Purpose: Visit each node and add contents into the symbol table

```java
import syntaxtree.*;
import visitor.DepthFirstVisitor;
import java.util.*;

class MethodInfo{
    String methType;
    String methID;
    Map<String,String> methVariables = new HashMap<~>();
    Map<String,String> methParameters = new HashMap<String, String>();

    public void print(){...}
    public String find(String c, String m, String s){...}

    public String getNumParams(){...}
}

class ClassInfo{
    String id;
    Map<String,MethodInfo> methodTable = new HashMap<~>();
    Map<String,String> localVarTables = new HashMap<~>();

    public void print(){...}

    public String find(String c, String m, String s){...}

    public boolean checkMethods(String s){...}
}

public class SymbolTableVisitor extends DepthFirstVisitor{
    Map<String,ClassInfo> symbolTable = new HashMap<~>();
    ClassInfo lastClass;
    MethodInfo lastMethod;
    boolean isClass = false;
    boolean isMeth = false;
    String vt;

    public boolean checkClassForMethod(String c, String m){
        for(Map.Entry<String,ClassInfo> entry : symbolTable.entrySet()){
            if(entry.getKey().equals(c)){
                return entry.getValue().checkMethods(m);
            }
        }
        return false;
    }
```



```
public class SymbolTableVisitor extends DepthFirstVisitor{
    Map<String,ClassInfo> symbolTable = new HashMap<~>();
    ClassInfo lastClass;
    MethodInfo lastMethod;
    boolean isClass = false;
    boolean isMeth = false;
    String vt;

    public boolean checkClassForMethod(String c, String m){...}

    public boolean checkClasses(String s){...}

    public void print(){...}

    public String find(String c,String m, String s){...}

    public void visit(ClassDeclaration n){
        ClassInfo newClass = new ClassInfo();
        newClass.id = n.f1.f0.toString();
        symbolTable.put(newClass.id, newClass);
        lastClass = newClass;
        //System.out.println("CLASS: " + newClass.id);
        isClass = true;
        n.f3.accept( v: this);
        isClass = false;
        n.f4.accept( v: this);
    }

    public void visit(VarDeclaration n){
        n.f0.accept( v: this);
        String varType = vt;
        String varID = n.f1.f0.toString();
        if(isClass && !isMeth) {
            lastClass.localVarTables.put(varID, varType);
            //System.out.println("CLASS VAR: " + varType + " " + varID);
        } else if(isMeth){
                lastMethod.methVariables.put(varID,varType);
            //System.out.println("METH VAR: " + varType + " " + varID);
            }
    }
```



```
    public void visit(MethodDeclaration n){
        MethodInfo newMethod = new MethodInfo();
        lastMethod = newMethod;
        n.f1.accept( v: this);
        newMethod.methType = vt;
        newMethod.methID = n.f2.f0.toString();
        //System.out.println("NEW METH: " + newMethod.methType +" "+newMethod.methID);
        n.f4.accept( v: this);
        isMeth = true;
        n.f7.accept( v: this);
        isMeth = false;
        lastClass.methodTable.put(newMethod.methID, newMethod);
    }

    public void visit(FormalParameterList n){
        n.f0.accept( v: this);
        n.f1.accept( v: this);
    }

    public void visit(FormalParameter n){
        n.f0.accept( v: this);
        String type = vt;
        String id = n.f1.f0.toString();
        lastMethod.methParameters.put(id, type);
        //System.out.println("PARAM: " + type + " " + id);
    }

    public void visit(FormalParameterRest n){
        n.f1.accept( v: this);
    }

    public void visit(Type n){
        n.f0.accept( v: this);
    }

    public void visit(ArrayType n){
        vt = "int array";
    }
}
```

Class/Variable Declaration

Method Declaration/etc

Type Visitor

Purpose: Type-checking based on type system rules

```java
import syntaxtree.*;
import visitor.DepthFirstVisitor;
import visitor.GJDepthFirst;

public class TypeVisitor extends GJDepthFirst<String, SymbolTableVisitor> {
    boolean hasError = false;
    String errorMsg = "";
    String currentClass = "";
    String currentMethod = "";


    public void checkTypeError() {
        if(hasError) {
            System.out.println("Type error");
        }
    }

    public String visit(ClassDeclaration n, SymbolTableVisitor s){
        currentClass = n.f1.f0.toString();
        n.f3.accept( v: this, s);
        n.f4.accept( v: this, s);
        return n.f1.f0.toString();
    }

    public String visit(MethodDeclaration n, SymbolTableVisitor s){
        currentMethod = n.f2.f0.toString();
        String methType = n.f1.accept( v: this, s);
        n.f4.accept( v: this, s);
        n.f7.accept( v: this, s);
        n.f8.accept( v: this, s);
        String returnType = n.f10.accept( v: this, s);

        if(methType.equals(returnType)){
            //System.out.println(methType + ", " + returnType + " : " + currentMethod + " : " + currentClass);
            currentMethod = "";
            return returnType;
        } else{
            //System.out.println(methType + ", " + returnType + " : " + currentMethod + " : " + currentClass);
            hasError = true;
            return "error";
        }
    }
```

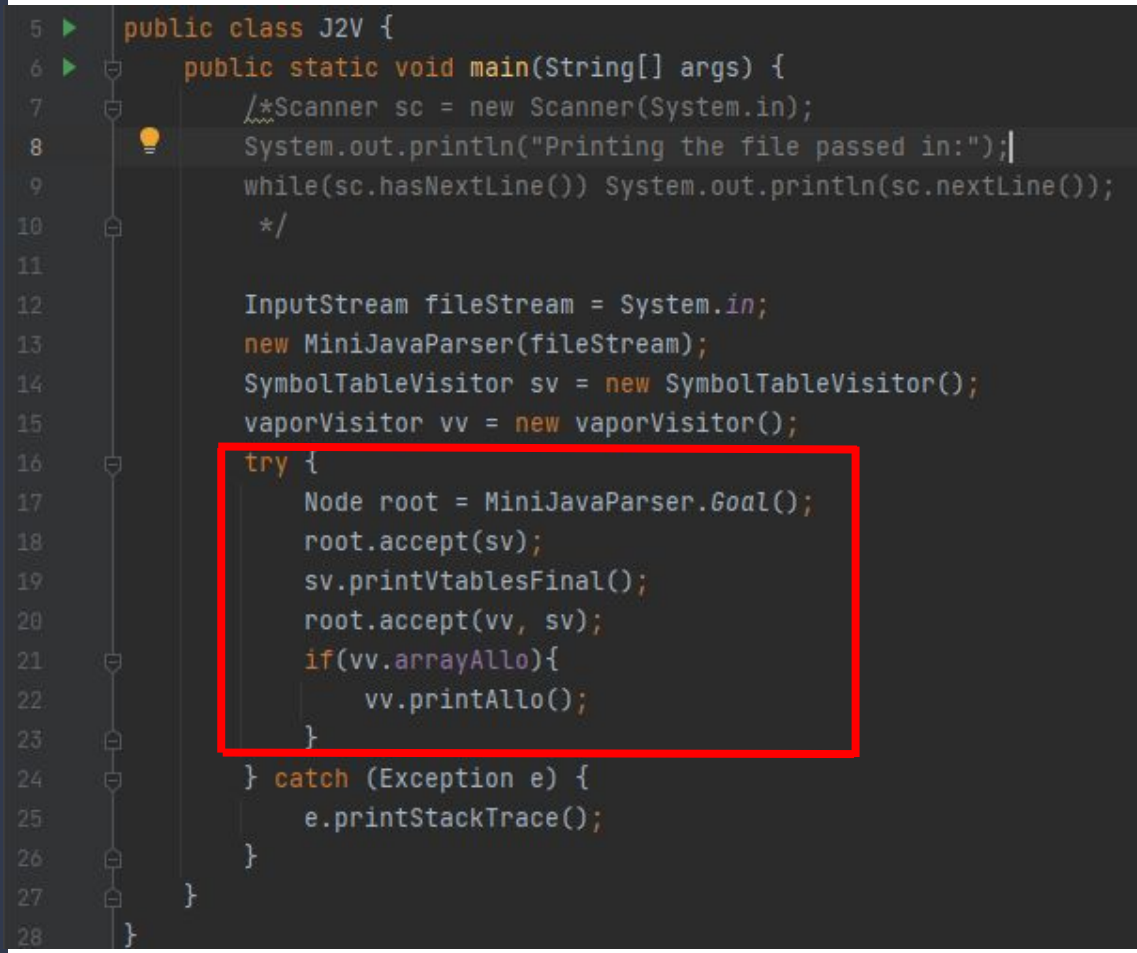
```
62  public String visit(CompareExpression n, SymbolTableVisitor s){
63      String arg1 = n.f0.accept( v: this, s);
64      String arg2 = n.f2.accept( v: this, s);
65      if((arg1 == arg2) && arg1 == "int"){
66          return "boolean";
67      } else{
68          hasError = true;
69          //errorMsg = arg1 + arg2 + "CompareExpression";
70          //System.out.println(errorMsg);
71          return "error";
72      }
73  }
74
75  public String visit(PlusExpression n, SymbolTableVisitor s){
76      String arg1 = n.f0.accept( v: this, s);
77      String arg2 = n.f2.accept( v: this, s);
78      if((arg1 == arg2) && arg1 == "int"){
79          return "int";
80      } else{
81          hasError = true;
82          //errorMsg = arg1 + arg2 + "PlusExpression";
83          //System.out.println(arg1);
84          //System.out.println(arg2);
85          //System.out.println(errorMsg);
86          return "error";
87      }
88  }
89
90  public String visit(MinusExpression n, SymbolTableVisitor s){
91      String arg1 = n.f0.accept( v: this, s);
92      String arg2 = n.f2.accept( v: this, s);
93      if((arg1 == arg2) && arg1 == "int"){
94          return "int";
95      } else{
96          hasError = true;
97          //errorMsg = arg1 + arg2 + "MinusExpression";
98          //System.out.println(errorMsg);
99          return "error";
100     }
101 }
102
```

```
218 public String visit(MessageSend n, SymbolTableVisitor s){
219     String arg1 = n.f0.accept( v: this, s); //class type
220     String arg2 = n.f2.accept( v: this, s); //method type
221     String arg3 = n.f4.accept( v: this, s); //method parameters
222     String methName = n.f2.f0.toString();
223     //System.out.println(arg1);
224     //System.out.println(methName);
225     //System.out.println(arg2);
226
227     if(!s.checkClasses(arg1)){
228         hasError = true;
229         //errorMsg = arg1 + " " +  methName + " MessageSend, Primitive Class";
230         //System.out.println(errorMsg);
231         return "error";
232     }
233     if(arg3 != null) {
234         if (!s.symbolTable.get(arg1).methodTable.get(methName).getNumParams().equals(arg3)) {
235             hasError = true;
236             //errorMsg = arg3 + " " + methName + " MessageSend, Bad Params";
237             //System.out.println(s.symbolTable.get(arg1).methodTable.get(methName).getNumParams());
238             //System.out.println(errorMsg);
239             return "error";
240         }
241     }
242
243     if(!s.checkClassForMethod(arg1, methName)){
244         hasError = true;
245         //errorMsg = arg1 + " " + methName + " MessageSend, Class Doesn't have Method";
246         //System.out.println(errorMsg);
247         return "error";
248     } else{
249         //System.out.println(arg2);
250         return arg2;
251     }
252 }
```

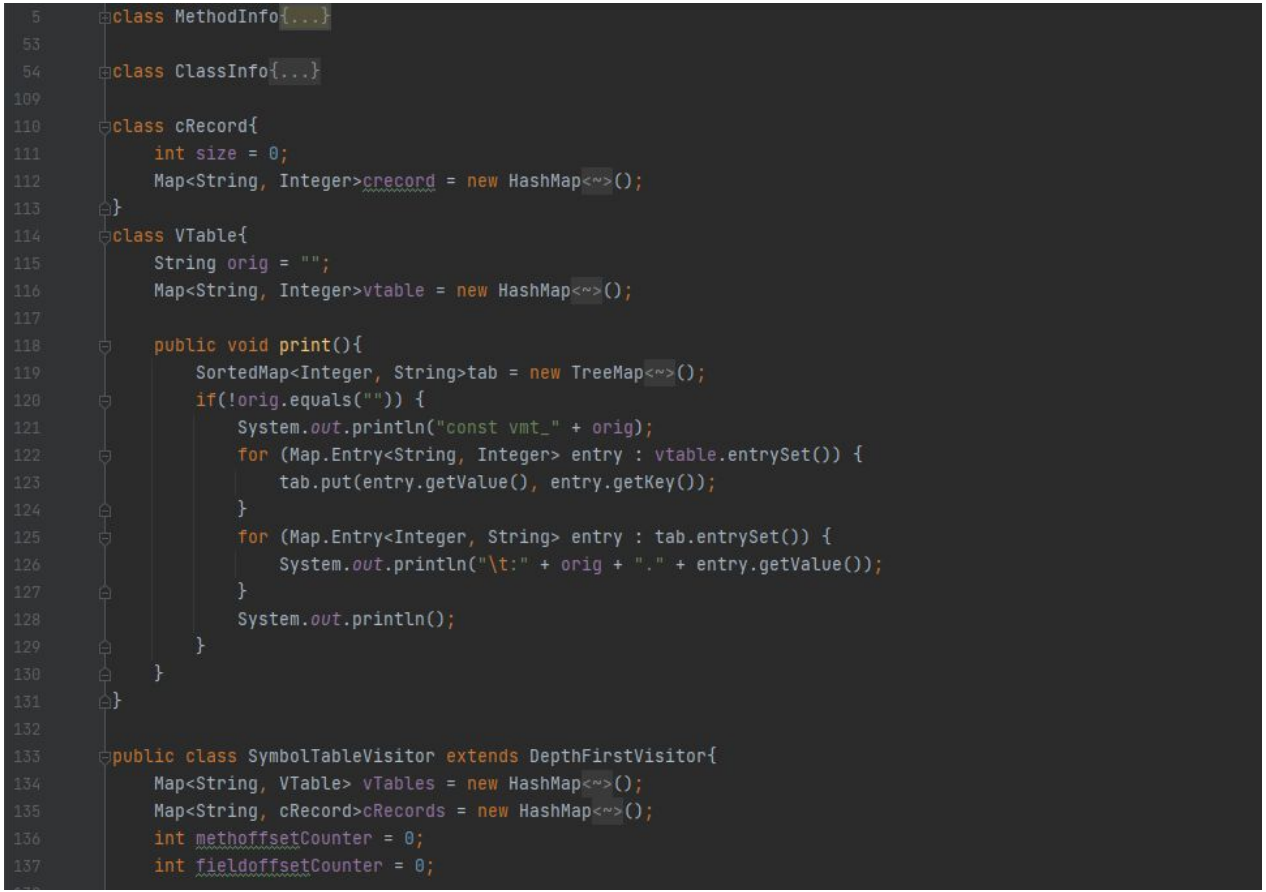
Phase 2: Code Gen

Goal: Translate MiniJava language into Vapor language

Main Function



```
public class J2V {
    public static void main(String[] args) {
        /*Scanner sc = new Scanner(System.in);
        System.out.println("Printing the file passed in:");
        while(sc.hasNextLine()) System.out.println(sc.nextLine());
         */

        InputStream fileStream = System.in;
        new MiniJavaParser(fileStream);
        SymbolTableVisitor sv = new SymbolTableVisitor();
        vaporVisitor vv = new vaporVisitor();
        try {
            Node root = MiniJavaParser.Goal();
            root.accept(sv);
            sv.printVtablesFinal();
            root.accept(vv, sv);
            if(vv.arrayAllo){
                vv.printAllo();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



```
class MethodInfo{...}

class ClassInfo{...}

class cRecord{
    int size = 0;
    Map<String, Integer>crecord = new HashMap<~>();
}
class VTable{
    String orig = "";
    Map<String, Integer>vtable = new HashMap<~>();

    public void print(){
        SortedMap<Integer, String>tab = new TreeMap<~>();
        if(!orig.equals("")) {
            System.out.println("const vmt_" + orig);
            for (Map.Entry<String, Integer> entry : vtable.entrySet()) {
                tab.put(entry.getValue(), entry.getKey());
            }
            for (Map.Entry<Integer, String> entry : tab.entrySet()) {
                System.out.println("\t:" + orig + "." + entry.getValue());
            }
            System.out.println();
        }
    }
}

public class SymbolTableVisitor extends DepthFirstVisitor{
    Map<String, VTable> vTables = new HashMap<~>();
    Map<String, cRecord>cRecords = new HashMap<~>();
    int methoffsetCounter = 0;
    int fieldoffsetCounter = 0;
```

Added class record and vtable to SymbolTableVisitor

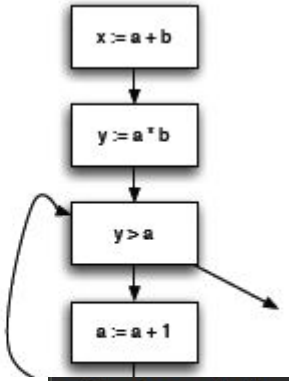
```
65  public String visit(MethodDeclaration n, SymbolTableVisitor s){
66      System.out.print("func " + currentClass + "." + n.f2.f0.toString());
67      System.out.println("(" + s.symbolTable.get(currentClass).methodTable.get(n.f2.f0.toString()).getParams() +")");
68      n.f8.accept( v: this, s);
69      varCounter = 0;
70
71      String returnValue = n.f10.accept( v: this, s);
72      if(returnValue == null){
73          returnValue = "";
74      }
75      if(returnValue.contains("this")){
76          String tempVar = "t." + varCounter;
77          varCounter++;
78          System.out.println(tempVar + " = " + returnValue);
79          returnValue = tempVar;
80      }
81      System.out.println("\tret " + returnValue + "\n");
82      return "";
83  }
84
85  public String visit(Statement n, SymbolTableVisitor s){
86      n.f0.accept( v: this, s);
87      return "";
88  }
89
90  public String visit(AssignmentStatement n, SymbolTableVisitor s){
91      isLeft = true;
92      String ident = n.f0.accept( v: this, s);
93      isLeft = false;
94      String exp = n.f2.accept( v: this,s);
95      for(int i = 0; i < tabCount; ++i){
96          System.out.print("\t");
97      }
98      System.out.println(ident + " = " + exp);
99      return "";
100 }
```

Vapor Visitor

Phase 3: Reg Allocation

Goal: Translate Vapor language into Vapor-M language, which uses registers and stacks instead of local variables

```
ic static Vector<instruction> createFlowAnalysis(VFunction s) throws Exception {
boolean isConv = false;
//Create new instruction nodes for each instruction and put list
Vector<instruction> insList = new Vector<>();
for(int i = 0; i < s.body.length; ++i){
    instruction newIns = new instruction();
    newIns.insNum = i;
    insList.add(newIns);
    vmVisitor vm = new vmVisitor();
    //System.out.println(s.body[i].getClass().getSimpleName());
    s.body[i].accept(insList, vm);
    if(s.body[i].getClass().getSimpleName().equals("VCall")){
        containsCall = true;
    }
}
//Connect all nodes to create flow diagram
for(int i = 0; i < insList.size() - 1; ++i){
    insList.elementAt(i).out_nodes.add(insList.elementAt( index: i+1));
    //System.out.println("Node " + i + "->" + "Node " + (i+1));
}
for(int i = 1; i < insList.size(); ++i){
    insList.elementAt(i).in_nodes.add(insList.elementAt( index: i-1));
    //System.out.println("Node " + i + "<-" + "Node " + (i-1));
}

//Connect nodes with branch and goto statements to the correct nodes
for(int i = 0; i < insList.size(); i++){
    if(insList.elementAt(i).goto_label != 0){
        /*System.out.print(insList.elementAt(i).goto_label_name + ":");
        System.out.println(insList.elementAt(i).goto_label);
        for(int a = 0; a < insList.elementAt(insList.elementAt(i).goto_label).def.
            System.out.println(insList.elementAt(insList.elementAt(i).goto_label).
        }*/
        insList.elementAt(i).out_nodes.add(insList.elementAt(insList.elementAt(i).
        insList.elementAt(insList.elementAt(i).goto_label).in_nodes.add(insList.el
    }
}
```




```
public class instruction{
    int insNum = 0;
    boolean converge = false;
    Vector<String> in = new Vector<~>();
    Vector<String> out = new Vector<~>();
    Vector<String> inPrime = new Vector<~>();
    Vector<String> outPrime = new Vector<~>();
    Vector<String> use = new Vector<~>();
    Vector<String> def = new Vector<~>();
    Vector<instruction> in_nodes = new Vector<>();
    Vector<instruction> out_nodes = new Vector<>();
    Vector<String> active_variables = new Vector<>();
    String goto_label_name = "";
    int goto_label = 0;
}
```

```
public class vmVisitor extends VInstr.VisitorP<Vector<instruction>, Exception> {

    @Override
    public void visit(Vector<instruction> o, VAssign vAssign) throws Exception {
        //Define use and def sets
        if(vAssign.source.toString().matches( regex: "(.*[a-z].*)|(.*[A-Z].*)")) {
            o.lastElement().use.add(vAssign.source.toString());
        }
        o.lastElement().def.add(vAssign.dest.toString());
    }

    @Override
    public void visit(Vector<instruction> o, VCall vCall) throws Exception {
        o.lastElement().def.add(vCall.dest.toString());
        for(int i = 0; i < vCall.args.length; ++i){
            if(vCall.args[i].toString().matches( regex: "(.*[a-z].*)|(.*[A-Z].*)")) {
                if(!vCall.args[i].toString().contains(":")) {
                    o.lastElement().use.add(vCall.args[i].toString());
                } else{
                    System.out.println("const " + vCall.args[i].toString().substring(1));
                }
            }
        }
        if(!vCall.addr.toString().contains(":")) {
            o.lastElement().use.add(vCall.addr.toString());
        }
```

```
//Change all in and out sets until convergence
while(!isConv){
    for(int i = 0; i < insList.size(); ++i){
        //Clear in' and Out'
        insList.elementAt(i).inPrime.clear();
        insList.elementAt(i).outPrime.clear();
        //Copy in and out into in' and out'
        insList.elementAt(i).inPrime.addAll(insList.elementAt(i).in);
        insList.elementAt(i).outPrime.addAll(insList.elementAt(i).out);

        for(int c = 0; c < insList.elementAt(i).use.size(); ++c){ //add use set
            String newval1 = insList.elementAt(i).use.elementAt(c);
            if(!insList.elementAt(i).in.contains(newval1)){
                insList.elementAt(i).in.add(newval1);
            }
        }
        for(int c = 0; c < insList.elementAt(i).out.size(); ++c){ //add out set
            String newval = insList.elementAt(i).out.elementAt(c);
            if((!insList.elementAt(i).in.contains(newval)) && (!insList.elementAt(i).def.contains(newval))){
                insList.elementAt(i).in.add(newval);
            }
        }
        //Add values into out set ->go to each node connected outwards from current one and add its in set
        for(int a = 0; a < insList.elementAt(i).out_nodes.size(); ++a){
            for(int b = 0; b < insList.elementAt(i).out_nodes.elementAt(a).in.size(); ++b){
                String newVar = insList.elementAt(i).out_nodes.elementAt(a).in.elementAt(b);
                if(!insList.elementAt(i).out.contains(newVar)) {
                    insList.elementAt(i).out.add(newVar);
                }
            }
        }
        //Check if in/out sets are equal to in'/out' sets
        if(insList.elementAt(i).inPrime.equals(insList.elementAt(i).in) && insList.elementAt(i).outPrime.equals(insList.elementAt(i).out)){
            insList.elementAt(i).converge = true;
        } else{
            insList.elementAt(i).converge = false;
        }
    }
    //Check if all instruction sets have converged
    int count = 0;
```

```java
public static Vector<instruction> activeVariables(VFunction s) throws Exception {
    Vector<instruction> flowDiagram = createFlowAnalysis(s);
    //Add in[n] U def[n] into active set of each node n
    for(int i = 0; i < flowDiagram.size(); ++i){
        flowDiagram.elementAt(i).active_variables.addAll(flowDiagram.elementAt(i).in);
        for(int c = 0; c < flowDiagram.elementAt(i).def.size(); ++c){
            String newval = flowDiagram.elementAt(i).def.elementAt(c);
            if(!flowDiagram.elementAt(i).active_variables.contains(newval)){
                flowDiagram.elementAt(i).active_variables.add(newval);
            }
        }
    }
    // This prints the active variables for testing
    /*for(int i = 0; i < flowDiagram.size(); ++i){
        System.out.println("Node " + i);
        for(int a = 0; a < flowDiagram.elementAt(i).active_variables.size(); ++a){
            System.out.print(flowDiagram.elementAt(i).active_variables.elementAt(a) + ", ");
        }
        System.out.println();
    }*/
    return flowDiagram;
}
```

```java
public static Vector<interval> activeIntervals(VFunction s) throws Exception{
    Vector<instruction> flowDiagram = activeVariables(s);
    Vector<String> variables = new Vector<>();
    Vector<interval> intervalList = new Vector<>();

    for(int i = 0; i < flowDiagram.size(); ++i){
        for(int j = 0; j < flowDiagram.elementAt(i).active_variables.size(); ++j){
            if(!variables.contains(flowDiagram.elementAt(i).active_variables.elementAt(j))){
                variables.add(flowDiagram.elementAt(i).active_variables.elementAt(j));
            }
        }
    }

    //Print out list of variables which we will be analyzing
    /*for(int i = 0; i < variables.size(); ++i){
        System.out.println(variables.elementAt(i));
    }*/

    for(int i = 0; i < variables.size(); ++i){
        String var = variables.elementAt(i);
        for(int a = 0; a < flowDiagram.size(); ++a){
            if(flowDiagram.elementAt(a).active_variables.contains(var)){
                interval newInterval = new interval();
                newInterval.var = var;
                newInterval.start = a;
                for(int b = a; b < flowDiagram.size(); ++b){
                    if(!flowDiagram.elementAt(b).active_variables.contains(var)){
                        newInterval.end = b-1;
                        intervalList.add(newInterval);
                        a=b;
                        break;
                    }
                    if(b == flowDiagram.size()-1){
                        newInterval.end = b;
                        intervalList.add(newInterval);
                        a=b;
                        break;
                    }
                }
            }
        }
    }
```

```
public static func linearScan(VFunction s) throws Exception{
    Vector<interval> intervalList = activeIntervals(s);
    Vector<interval> activeList = new Vector<>();
    Vector<String> regList = new Vector<String>();
    Vector<String> inStack = new Vector<String>();
    Vector<String> localStack = new Vector<String>();
    HashMap<String, String> regAllocation = new HashMap<>();
    func currentFunc = new func();

    //Add $s and $t registers to regList
    for(int i = 0; i < 9; ++i){
        regList.add("$t" + i);
    }
    for(int i = 0; i < 8; ++i){
        regList.add("$s" + i);
    }

    //Parameters are mapped to $a, excess params left in inStack, excess $a registers
    int paramRegisters = 0;
    for(int i = 0; i < s.params.length; ++i){
        inStack.add(s.params[i].toString());
        if(paramRegisters < 4){
            regAllocation.put(s.params[i].toString(),"$a" + paramRegisters);
            //Remove parameters from interval list so they don't get removed
            for(int z = 0; z < intervalList.size(); ++z){
                if((intervalList.elementAt(z).var).equals(s.params[i].toString())){
                    //intervalList.removeElementAt(z);
                    intervalList.elementAt(z).var = "$a" + paramRegisters;
                }
            }
            ++paramRegisters;
        } else{
            regAllocation.put(s.params[i].toString(),"in["+ i +"]");
            for(int z = 0; z < intervalList.size(); ++z){
                if((intervalList.elementAt(z).var).equals(s.params[i].toString())){
                    //intervalList.removeElementAt(z);
                    intervalList.elementAt(z).var = "in["+ i +"]";
                }
            }
        }
```

```
    //LinearScanRegisterAllocation
    for(int i = 0; i < intervalList.size(); ++i){
        //ExpireOldIntervals
            //Sort activeList by increasing endpoint
        for(int a = 0; a < activeList.size()-1; ++a){
            for(int b = a+1; b < activeList.size(); ++b){
                if(activeList.elementAt(b).end < activeList.elementAt(a).end){
                    temp = activeList.elementAt(a);
                    activeList.setElementAt(activeList.elementAt(b), a);
                    activeList.setElementAt(temp, b);
                }
            }
        }
        //foreach interval j in active, in order or increasing end point
        for(int j = 0; j < activeList.size(); ++j){
            if(activeList.elementAt(j).end > intervalList.elementAt(i).start){
                break;
            } else{
                regList.add(regAllocation.get(activeList.elementAt(j).var));
                activeList.remove(activeList.elementAt(j));
            }
        }
        //End of ExpireOldIntervals
        if(activeList.size() == regList.size()){
            //SpillAtInterval(i)
            localStack.add(activeList.lastElement().var);
            if(activeList.lastElement().end > intervalList.elementAt(i).end){
                String spillReg = regAllocation.get(localStack.lastElement());
                regAllocation.put(intervalList.elementAt(i).var,spillReg);
                regAllocation.remove(localStack.lastElement());
                regAllocation.put(localStack.lastElement(),"local[" + (localStack.size()-1) + "]");
                activeList.remove(activeList.lastElement());
                activeList.add(intervalList.elementAt(i));
                //Resort Active List by end point
                for(int d = 0; d < activeList.size()-1; ++d){
                    for(int f = d+1; f < activeList.size(); ++f){
                        if(activeList.elementAt(f).end < activeList.elementAt(d).end){
                            temp = activeList.elementAt(d);
                            activeList.setElementAt(activeList.elementAt(f), d);
                            activeList.setElementAt(temp, f);
                        }
                    }
```

```
public static void main(String[] args) throws Exception {
    InputStream fileStream = System.in;
    VaporProgram root = parseVapor(fileStream);
    printHeader(root);
    for(int i = 0; i < root.functions.length; ++i){
        containsCall = false;
        func currentFunc = linearScan(root.functions[i]);
        printVisitor pv = new printVisitor();

        //Print out the func header with array sizes
        System.out.print("func " + root.functions[i].ident + " [");
        System.out.print("in " + currentFunc.inStack.size() + ", ");
        System.out.print("out " + currentFunc.outStack.size() + ", ");
        System.out.println("local " + currentFunc.localStack.size() + "]");

        //Print out statements to save all $s register values we use
        for(int s = 0; s < currentFunc.localStack.size() ; ++s){
            if(currentFunc.localStack.elementAt(s).contains("$s")){
                System.out.println("\tlocal[" + s + "] = " + currentFunc.localStack.elementAt(s));
            }
        }

        //load parameters
        for(int p = 0; p < root.functions[i].params.length; ++p){
            for(Map.Entry<String,String> entry : currentFunc.regAlloc.entrySet()){
                if(entry.getKey().equals(root.functions[i].params[p].toString())){
                    if(!(currentFunc.regAlloc.get(entry.getValue()) == null)) {
                        System.out.println("\t" + currentFunc.regAlloc.get(entry.getValue()) + " = " + entry.getValue());
                    }
                }
            }
        }

        //Print out the translated body instructions
        for(int a = 0; a < root.functions[i].body.length; ++a){
            //Print label
            for(int b = 0; b < root.functions[i].labels.length; ++b) {
                if (a == root.functions[i].labels[b].instrIndex) {
                    System.out.println(root.functions[i].labels[b].ident + ":");
                }
            }
```

```
public class func {
    int currInst = 0;
    HashMap<String, String>regAlloc = new HashMap<>();
    Vector<String> inStack = new Vector<>();
    Vector<String> outStack = new Vector<>();
    Vector<String> localStack = new Vector<>();
    Vector<interval> intervalList = new Vector<>();
}
```

```
            //Print Translated Instruction
            currentFunc.currInst = a;
            root.functions[i].body[a].accept(currentFunc, pv);
        }

        //Print out the loading of the old $s values into appropriate registers
        for(int s = 0; s < currentFunc.localStack.size() ; ++s){
            if(currentFunc.localStack.elementAt(s).contains("$s")){
                System.out.println("\t" + currentFunc.localStack.elementAt(s) + " = local[" + s + "]");
            }
        }

        //Now we can print the ret statement
        System.out.println("\tret\n");
    }
}
```


Main Function

```
lic class printVisitor extends VInstr.VisitorP<func, Exception> {
@Override
public void visit(func o, VAssign vAssign) throws Exception {
    if(o.regAlloc.containsKey(vAssign.source.toString())) {
        System.out.println("\t" + o.regAlloc.get(vAssign.dest.toString()) + " = " + o.regAlloc.get(vAssign.source.toString()));
    } else{
        System.out.println("\t" + o.regAlloc.get(vAssign.dest.toString()) + " = " + vAssign.source.toString());
    }
}

@Override
public void visit(func o, VCall vCall) throws Exception {
    //Save $t variables
    for(int t = 0; t < o.localStack.size(); ++t){
        if(o.localStack.elementAt(t).contains("$t")) {
            System.out.println("\tlocal[" + t + "] = " + o.localStack.elementAt(t));
        }
    }
    //Save current argument registers to the in stack
    int regCount = 0;
    for(int a = 0; a < o.inStack.size(); ++a){
        if(regCount < 4){
            System.out.println("\tin[" + a + "] = $a" + a);
            regCount++;
        }
    }
    //Store params
    regCount = 0;
    for(int i = 0; i < vCall.args.length; ++i){
        String ar = vCall.args[i].toString();
        if(regCount < 4){
            if(o.regAlloc.containsKey(ar)) {
                if(o.regAlloc.get(ar).contains("$a")){
                    System.out.println("\t$a" + regCount + " = " + o.regAlloc.get(o.regAlloc.get(ar)));
                    //System.out.println("\t$a" + regCount + " = in[" + regCount + "]");
                }else {
                    System.out.println("\t$a" + regCount + " = " + o.regAlloc.get(ar));
                }
            } else{
                System.out.println("\t$a" + regCount + " = " + ar);
            }
```

```
@Override
public void visit(func o, VBuiltIn vBuiltIn) throws Exception {
    System.out.print("\t");
    if(vBuiltIn.dest != null){
        System.out.print(o.regAlloc.get(vBuiltIn.dest.toString()) + " = ");
    }
    System.out.print(vBuiltIn.op.name + "(");
    for(int i = 0; i < vBuiltIn.args.length; ++i){
        if(o.regAlloc.containsKey(vBuiltIn.args[i].toString())){
            if(o.regAlloc.get(vBuiltIn.args[i].toString()).contains("in")){
                System.out.print(o.regAlloc.get(o.regAlloc.get(vBuiltIn.args[i].toString())));
            }else {
                System.out.print(o.regAlloc.get(vBuiltIn.args[i].toString()));
            }
        } else {
            System.out.print(vBuiltIn.args[i]);
        }
        if(!(i == vBuiltIn.args.length-1)){
            System.out.print(" ");
        }
    }
    System.out.println(")");
}

@Override
public void visit(func o, VMemWrite vMemWrite) throws Exception {
    int byteOff = ((VMemRef.Global)vMemWrite.dest).byteOffset;
    if(byteOff != 0){
        if(o.regAlloc.get(((VMemRef.Global) vMemWrite.dest).base.toString()).contains("in")) {
            System.out.print("\t[" + o.regAlloc.get(o.regAlloc.get(((VMemRef.Global) vMemWrite.dest).base.toString())) + "+" + byteOff + "] = ");
        } else{
            if(o.regAlloc.get(((VMemRef.Global) vMemWrite.dest).base.toString()).contains("$a")){
                String temp = o.regAlloc.get(((VMemRef.Global) vMemWrite.dest).base.toString());
                System.out.print("\t[" + o.regAlloc.get(temp) + "+" + byteOff + "] = ");
            } else {
                System.out.print("\t[" + o.regAlloc.get(((VMemRef.Global) vMemWrite.dest).base.toString()) + "+" + byteOff + "] = ");
            }
        }
    }else {
        System.out.print("\t[" + o.regAlloc.get(((VMemRef.Global) vMemWrite.dest).base.toString()) + "] = ");
```

printVisitor

Phase 4: Activation Records and Instruction Selection

Goal: Vapor-M registers and stacks are mapped to MIPS registers and runtime stack. Vapor-M instructions mapped to MIPS instructions.

Main Function

```
public static void main(String[] args) throws Exception{
    InputStream fileStream = System.in;
    VaporProgram root = parseVapor(fileStream);
    mipsVisitor mv = new mipsVisitor();
    printHeader(root);
    for(int i = 0; i < root.functions.length; ++i){
        VFunction temp = root.functions[i];
        callsFunction = false;

        //Check if function calls another function
        for(int a = 0; a < root.functions[i].body.length; ++a){
            if(temp.body[a].getClass().getSimpleName().contains("VCall") || temp.body[a].getClass().getSimpleName().contains
                callsFunction = true;
            }
        }

        System.out.println(temp.ident.toString() + ":");
        //Save $fp to location $sp - 2
        System.out.println("  sw $fp -8($sp)");
        //   Move $fp to point to where $sp is pointing
        System.out.println("  move $fp $sp");
        //   Pushing the frame
        //      Decrease $sp by size = Local + Out + 2
        //         1 for Return address
        //         1 for frame pointer
        System.out.println("  subu $sp $sp " + (temp.stack.local + temp.stack.out + 2)*4);
        //   Saving the return register $ra at $fp - 1 (if the function calls any other function)
        //      Note that $fp now holds the old $sp
        if(callsFunction){
            System.out.println("  sw $ra -4($fp)");
        }

        //translate rest of instructions
        for(int a = 0; a < root.functions[i].body.length; ++a){
            //Print labels in correct spots
            for(int b = 0; b< root.functions[i].labels.length; ++b){
                if(root.functions[i].labels[b].instrIndex == a){
                    System.out.println(root.functions[i].labels[b].ident + ":");
                }
            }

            root.functions[i].body[a].accept(temp,mv);
        }
```

```java
public class mipsVisitor extends VInstr.VisitorP<VFunction, Exception> {

    @Override
    public void visit(VFunction func, VAssign vAssign) throws Exception {
        if(!vAssign.source.toString().contains("$")){
            if(!vAssign.source.toString().contains(":")) {
                System.out.println("  li " + vAssign.dest.toString() + " " + vAssign.source.toString());
            }else{
                System.out.println("  la " + vAssign.dest.toString() + " " + vAssign.source.toString().substring(1));
            }
        }else {
            System.out.println("  move " + vAssign.dest.toString() + " " + vAssign.source.toString());
        }

    }

    @Override
    public void visit(VFunction func, VCall vCall) throws Exception {
        if(vCall.addr.toString().contains("$")){
            System.out.print("  jalr ");
            System.out.println(vCall.addr.toString());
        } else{
            System.out.println("  jal " + vCall.addr.toString().substring(1));
        }
    }

    @Override
    public void visit(VFunction func, VBuiltIn vBuiltIn) throws Exception {
        if(vBuiltIn.op.name.toString().equals("HeapAllocZ")){
            if(vBuiltIn.args[0].toString().contains("$")){
                System.out.println("  move $a0 " + vBuiltIn.args[0].toString());
            } else {
                System.out.println("  li $a0 " + vBuiltIn.args[0].toString());
            }
            System.out.println("  jal _heapAlloc");
            System.out.println("  move " + vBuiltIn.dest.toString() + " $v0");
        }
        if(vBuiltIn.op.name.toString().equals("Error")){
            System.out.print("  la $a0 ");
            if(vBuiltIn.args[0].toString().contains("null")){
                System.out.println("_str0");
            }
            if(vBuiltIn.args[0].toString().contains("array index out of bounds")){
```

```java
    @Override
    public void visit(VFunction func, VBuiltIn vBuiltIn) throws Exception {
        if(vBuiltIn.op.name.toString().equals("HeapAllocZ")){
            if(vBuiltIn.args[0].toString().contains("$")){
                System.out.println("  move $a0 " + vBuiltIn.args[0].toString());
            } else {
                System.out.println("  li $a0 " + vBuiltIn.args[0].toString());
            }
            System.out.println("  jal _heapAlloc");
            System.out.println("  move " + vBuiltIn.dest.toString() + " $v0");
        }
        if(vBuiltIn.op.name.toString().equals("Error")){
            System.out.print("  la $a0 ");
            if(vBuiltIn.args[0].toString().contains("null")){
                System.out.println("_str0");
            }
            if(vBuiltIn.args[0].toString().contains("array index out of bounds")){
                System.out.println("_str1");
            }
            System.out.println("  j _error");
        }
        if(vBuiltIn.op.name.toString().equals("PrintIntS")) {
            if(!vBuiltIn.args[0].toString().contains("$")){
                System.out.println("  li $a0 " + vBuiltIn.args[0].toString());
            }else {
                System.out.println("  move $a0 " + vBuiltIn.args[0].toString());
            }
            System.out.println("  jal _print");
        }
        if(vBuiltIn.op.name.toString().equals("LtS")){...}
        if(vBuiltIn.op.name.toString().equals("Sub")){...}
        if(vBuiltIn.op.name.toString().equals("MulS")){...}
        if(vBuiltIn.op.name.toString().equals("Lt")){...}
        if(vBuiltIn.op.name.toString().equals("Add")){...}
    }

    @Override
    public void visit(VFunction func, VMemWrite vMemWrite) throws Exception {
        if(vMemWrite.source.toString().contains(":")){
            System.out.println("  la $t9 " + vMemWrite.source.toString().substring(1));
            System.out.println("  sw $t9 " + ((VMemRef.Global)vMemWrite.dest).byteOffset + "(" + ((VMemRef.Globa
```

mipsVisitor

```java
private static boolean callsFunction = false;
public static void printHeader(VaporProgram root){
    //Get the class names
    Vector<String> functionNames = new Vector<>();
    Vector<String> classNames = new Vector<>();
    for(int i = 0; i < root.functions.length; ++i){
        if(root.functions[i].ident.contains(".")) {
            String func_name = root.functions[i].ident;
            functionNames.add(func_name);
            String[] parsedName = func_name.split( regex: "\\.", limit: 2);
            if(!classNames.contains(parsedName[0])) {
                classNames.add(parsedName[0]);
            }
        }
    }
    //Print Header Info
    System.out.println(".data\n");
    for(int i = 0; i < root.dataSegments.length; ++i){
        System.out.println(root.dataSegments[i].ident + ":");
        for(int j = 0; j < functionNames.size(); ++j){
            if(functionNames.elementAt(j).contains(classNames.elementAt(i))){
                System.out.println("  " + functionNames.elementAt(j));
            }
        }
    }
    System.out.println();
    System.out.println(".text");
    System.out.println("  jal Main");
    System.out.println("  li $v0 10");
    System.out.println("  syscall");
    System.out.println();
}
```

```java
public static void printTail(){
    System.out.println("_print:");
    System.out.println("  li $v0 1");
    System.out.println("  syscall");
    System.out.println("  la $a0 _newline");
    System.out.println("  li $v0 4");
    System.out.println("  syscall");
    System.out.println("  jr $ra");
    System.out.println();

    System.out.println("_error:");
    System.out.println("  li $v0 4");
    System.out.println("  syscall");
    System.out.println("  li $v0 10");
    System.out.println("  syscall");
    System.out.println();

    System.out.println("_heapAlloc:");
    System.out.println("  li $v0 9");
    System.out.println("  syscall");
    System.out.println("  jr $ra");

    System.out.println();
}
public static VaporProgram parseVapor(InputStream in) throws IOException {...}
```

Header and Tail

Challenges