HUAZHONG UNIVERSITY
OF SCIENCE & TECHNOLOGY

Senior Project [Undergraduate Thesis]

Modularized Design and Implementation of MiniJava-MIPS Compiler in Java

School	School of Engineering Science
Major	Biomedical Engineering.
Name	Jiaqi Xiong
Student ID	U201514474
Instructor	Yong Deng and Mohsen Lesani

June 27th, 2019

Abstract

In this internet era, our daily life depends on various software. But before a program can be run, its original code must be translated into binary machine code. Compiler plays a key role during this translation process. It's significant to design and compilers for the development of computer compiling in the future.

A four-phase design of.MiniJava-MIPS compiler programmed in Java language is presented. Firstly the compiler type check a program written in MiniJava language, then the program is translated to vapor code during intermediate code generation phase, next the Vapor code is translated to the Vapor-M code in register allocation phase, and eventually to MIPS instruction in instruction selection phase. Four phases are modularized with the output of the previous module to be the input of the next module, thereby facilitating porting and local optimization.

Key Words： Modularized compiler; MiniJava language; Visitor Pattern; Type-Che -cking

Content

1 Introduction

1.1 Significance of Self-developed Compiler

People are increasingly inseparable from various software written in some programming language. But a program must be translated into a form in which it can be executed by a computer at first. This is where the compiler plays an important role. Generally, higher-level programming languages are friendly to programmers, but are less efficient. Optimizing compilers include techniques to improve the performance of generated code, thus compensating the inefficiency. Since programming in high-level languages is normal, the performance of a computer system is determined not only by its raw speed but also by how well compilers can exploit its features. Thus, in modern computer architecture development, compiled code is used to evaluate the proposed architectural features. While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translation between different kinds of languages. Compiled Simulation is such an important application. In electronic circuit course, students are always required using FPGA to implement a digital clock. Simulation is used to validate this design written in Verilog. But in practical situation, it can be very expensive in some experiment with many possible designs on many different input sets. Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code[1]. Considering applications of compilers mentioned above, the necessity to develop compilers is obvious.

The purpose that people need compile a new source language, or create a new target language, or both, stimulates programmers to write new compilers. Regardless of profitable motivation, writing a compiler is a kind of training in software engineering. Some lessons and experience in software development can be used to improve the reliability and maintainability of the final product[2]. For students major in computer science or related fields, they should have learned knowledge like object-oriented programming language, data structure, algorithm and computer architecture. A design and implementation of a compiler is not only to see the further theory behind different components of a compiler but also a study of how to digest theories and put them into practice. As writing a compiler is a large-scale project, it also provides students with chance to cooperate in a group and think moral and professional issue.

1.2 Critical Techniques in MiniJava-MIPS Compiler

1.2.1 Multi-pass

A multi-pass compiler is a type of compiler that processes the source code or abstract syntax tree of a program several times. It converts the program into one or more intermediate representations in steps between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass. Each pass takes the result of the previous pass as the input, and creates an intermediate output. In this way, the intermediate code is improved pass by pass, until the final pass produces the final code. This contrasts with a one-pass compiler, which traverses the program only once[3].

One-pass compilers are unable to generate as efficient programs as multi-pass compilers due to the limited scope of available information. Many effective compiler optimizations require multiple passes over a block, subroutine, module or even an entire program. Some programming languages simply cannot be compiled in a single pass, as a result of their design. For example, a language may allow references to the not-yet-declared items, so no code can be generated until the entire program has been scanned. In contrast, some programming languages include special constructs to allow one-pass compilation. For MIL- MINI-L compiler[4], that project mainly focuses on lexical analysis and parsing. Performing one-pass code generation is a simple choice since it avoids building and traversing a syntax tree. However, we are provided JavaCC parser generator[5] and Java Tree Builder[6], which means we will have a complete parser for MiniJava and a set of classes used for traversing the syntax tree. Two- pass is a better choice.

1.2.2 Visitor Pattern

Java Tree Builder from Purdue University, is a syntax tree builder to be used with the Java Compiler Compiler (JavaCC) parser generator from Sun Microsystems. It utilizes the visitor design pattern, which enables the definition of a new operation on an object structure without changing the classes of the objects for object-oriented programming.

The visitor pattern divides the code into an object structure and a Visitor, which is akin to functional programming. A Visitor overloads a visit method for each class. The visit methods describe both actions, and access of subobjects. A visitor method for a class “A” takes an argument of type “A”. Each class must have an accept method, each of which takes a visitor as argument. The purpose of the accept methods is to invoke the visit method in the visitor which can handle the current object. The control flow goes back and forth between the visit methods in the visitor and the accept methods in the object structure[7].

When using the Visitor pattern, the set of classes must be fixed in advance, thus adding new classes to the object structure is hard. Visitor’s approach assumes that the interface of the data structure classes is powerful enough to let visitors do their job. As a result, the pattern often forces providing public operations that access internal state, which may compromise its encapsulation[7]. However, visitor makes adding new operations easy by simply writing a new visitor. And the visitor pattern has an edge over Instanceof and Type Casts and Dedicated methods by adding new methods without recompilation.

1.3 Purposes of the Research

The goal of this work is to design and implement the main phases of a modern MiniJava-to-MIPS compiler. This paper consists of four phases: type-checking, intermediate code generation, register allocation, and activation records and instruction selection. For each phase, the objective will be introduced firstly. The specification for each language related to this work, such as MiniJava, Vapor and Vapor-M, will also be presented. The design part will simulate the flow of thought when facing the task to explain how to deal with the task and how to improve the work in the future. It also includes challenges encountered during implementation and how to cope with them through several examples. To assess the validation of each phase, results from testers[8] provided by professor Lesani will be shown.

2 Type Check

2.1 Introduction

The goal of phase one is to write a type checker for MiniJava. MiniJava is a subset of java that includes the bare minimum of Java. Given a program, the type checker checks at compile time that type mismatch does not happen at run time. It either approves or rejects the program. The set of rules that the type checker checks are represented as a type system which will be introduced in chapter 2.2.2.

With the help of JavaCC parser generator and Java Tree Builder, we will have a complete parser for MiniJava, a set of classes used for traversing the syntax tree, and two different default visitors: DepthFirstVisitor and GJDepthFirst. Thus, all the remaining work for type checking a MiniJava program is based on this abstract syntax tree by extending GJDepthFirst. The first thing we need to do is to get familiar with the grammar of MiniJava and the set of rules that the type checker checks.

2.2 MiniJava Language

MiniJava is a subset of Java. The meaning of a MiniJava program is given by its meaning as a Java program. MiniJava differs from Java mainly in three aspects[9]:

1) Overloading is not allowed in MiniJava.
2) The MiniJava statement “System.out.println(...);” can only print integers.
3) The MiniJava expression “e.length” only applies to expressions of type int[].

2.2.1 Grammar

To be specific, the grammar is specified by figure 2-1 using the following metanotation[10]:

1) Non-terminal symbols are words written in italic form;
2) A production is of the form lhs ::= rhs, where lhs is a nonterminal symbol and rhs is a sequence of nonterminal and terminal symbols, with choices separated by |, and sometimes using “. . . " to denote a possibly empty list;

3) Superscripts and subscripts are used to distinguish metavariables.

$$
\begin{array}{rrcl}
(\mathit{Goal}) & g & ::= & mc\ d_1\ \ldots\ d_n \\
(\mathit{MainClass}) & mc & ::= & \texttt{class}\ id\ \{\ \texttt{public static void main (String []}\ id^S\texttt{)}\{ \\
 & & & \quad t_1\ id_1;\ \ldots;\ t_r\ id_r;\ s_1 \ldots\ s_q\}\} \\
(\mathit{TypeDeclaration}) & d & ::= & \texttt{class}\ id\ \{\ t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots\ m_k\ \} \\
 & & | & \texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots\ m_k\ \} \\
(\mathit{MethodDeclaration}) & m & ::= & \texttt{public}\ t\ id^M\ (t_1^F\ id_1^F,\ \ldots,\ t_n^F\ id_n^F)\ \{ \\
 & & & \quad t_1\ id_1;\ \ldots;\ t_r\ id_r;\ s_1\ \ldots\ s_q\ \texttt{return}\ e;\ \} \\
(\mathit{Type}) & t & ::= & \texttt{int[]}\ |\ \texttt{boolean}\ |\ \texttt{int}\ |\ id \\
(\mathit{Statement}) & s & ::= & \{\ s_1\ \ldots\ s_q\ \}\ |\ id\ \texttt{=}\ e;\ |\ id\ \texttt{[}\ e_1\ \texttt{]}\ \texttt{=}\ e_2; \\
 & & | & \texttt{if}\ (\ e\ )\ s_1\ \texttt{else}\ s_2\ |\ \texttt{while}\ (\ e\ )\ s\ |\ \texttt{System.out.println(}\ e\ \texttt{);} \\
(\mathit{Expression}) & e & ::= & p_1\ \texttt{\&\&}\ p_2\ |\ p_1\ \texttt{<}\ p_2\ |\ p_1\ \texttt{+}\ p_2\ |\ p_1\ \texttt{-}\ p_2\ |\ p_1\ \texttt{*}\ p_2\ |\ p_1\ \texttt{[}\ p_2\ \texttt{]} \\
 & & | & p\ \texttt{.length}\ |\ p\ .id\ (e_1,\ \ldots,\ e_n)\ |\ p \\
(\mathit{PrimaryExpression}) & p & ::= & c\ |\ \texttt{true}\ |\ \texttt{false}\ |\ id\ |\ \texttt{this}\ |\ \texttt{new int[}e\texttt{]}\ |\ \texttt{new}\ id\texttt{()}\ |\ \texttt{!}e\ |\ (e) \\
(\mathit{IntegerLiteral}) & c & ::= & \langle \text{INTEGER\_LITERAL} \rangle \\
(\mathit{Identifier}) & id & ::= & \langle \text{IDENTIFIER} \rangle
\end{array}
$$

Figure 2-1 Grammar for MiniJava

2.2.2 Items need checking

After learning the grammar for MiniJava, then the items we need to check can be summarized. The complete nodes that require type checking in the abstract syntax tree are shown in the table 2-2. For those who want to flick through the items, the table 2-1 and an example for ArrayAssignmentStatement are enough.

Table 2-1 Typical Type Rules Need Checking

Syntax tree node	Items need checking
MethodDeclaration	return value type
ArrayAssignmentStatement	id type, offset type, right hand value type, ids used has been declared
IfStatement, WhileStatement	condition type

AndExpression, CompareExpression, NotExpression	primary expression type

For example, for the array assignment statement “b[0] = a + 1； ”, we need check whether b is an array type, whether the offset of the array is an int type, whether the right hand side value is an int type, and whether a and b has been declared before. But how can we know the type of a and b? The symbol table helps us.

Table 2-2 The Whole Type Rules Need Checking

Syntax Tree Node	Items need checking
MethodDeclaration	The type of return value doesn’t match the type of method declaration
AssignmentStatement	The type of right hand value doesn’t match that of left hand value Identifier used has not been declared
ArrayAssignmentStatement	The identifier is not an array type The offset of the array is not an int type Right hand value is not an int type Identifier used has not been declared
IfStatement	Condition is not a Boolean
WhileStatement	Condition is not a Boolean
PrintStatement	The content printed is not an int type
AndExpression	PrimaryExpression is not a Boolean type
CompareExpression	PrimaryExpression is not a Boolean type
PlusExpression	PrimaryExpression is not a Boolean type
MinusExpression	PrimaryExpression is not a Boolean type
TimesExpression	PrimaryExpression is not a Boolean type

ArrayLookup	The identifier is not an array type The offset of the array is not an int type
ArrayLength	The identifier is not an array type
MessageSend	ExpressionList doesn’t match parameters of method declaration The method called has not been declared
ArrayAllocationExpression	The offset of the array is not an int type
AllocationExpression	The class used has not been declared
NotExpression	PrimaryExpression is not a Boolean type

2.3 Symbol Table

An environment is a set of bindings denoted by the ↦ arrow. For example, we could say that the environment σ0 contains the bindings {g ↦ string, a ↦ int}, meaning that the identifier a is an integer variable and g is a string variable. This first step is characterized by the maintenance of symbol tables (also called environments) mapping identifiers to their types and locations. As the declarations of types, variables, and functions are processed, these identifiers are bound to "meanings" in the symbol tables. When uses of identifiers are found, they are looked up in the symbol tables. Each local variable in a program has a scope in which it is visible. For example, in a MiniJava method “m”, all formal parameters and local variables declared in “m” are visible only until the end of “m”. As the semantic analysis reaches the end of each scope, the identifier bindings local to that scope are discarded[11].

In terms of structure, the symbol table is used to describe and save the structure of the entire MiniJava program, which contains the class structure, method structure and variable types. In terms of usage, the symbol table needs to have an approach to query its contents, so that it can be used in the second step. The symbol table consists of an abstract class and four instance classes, as shown in figure 2-2.

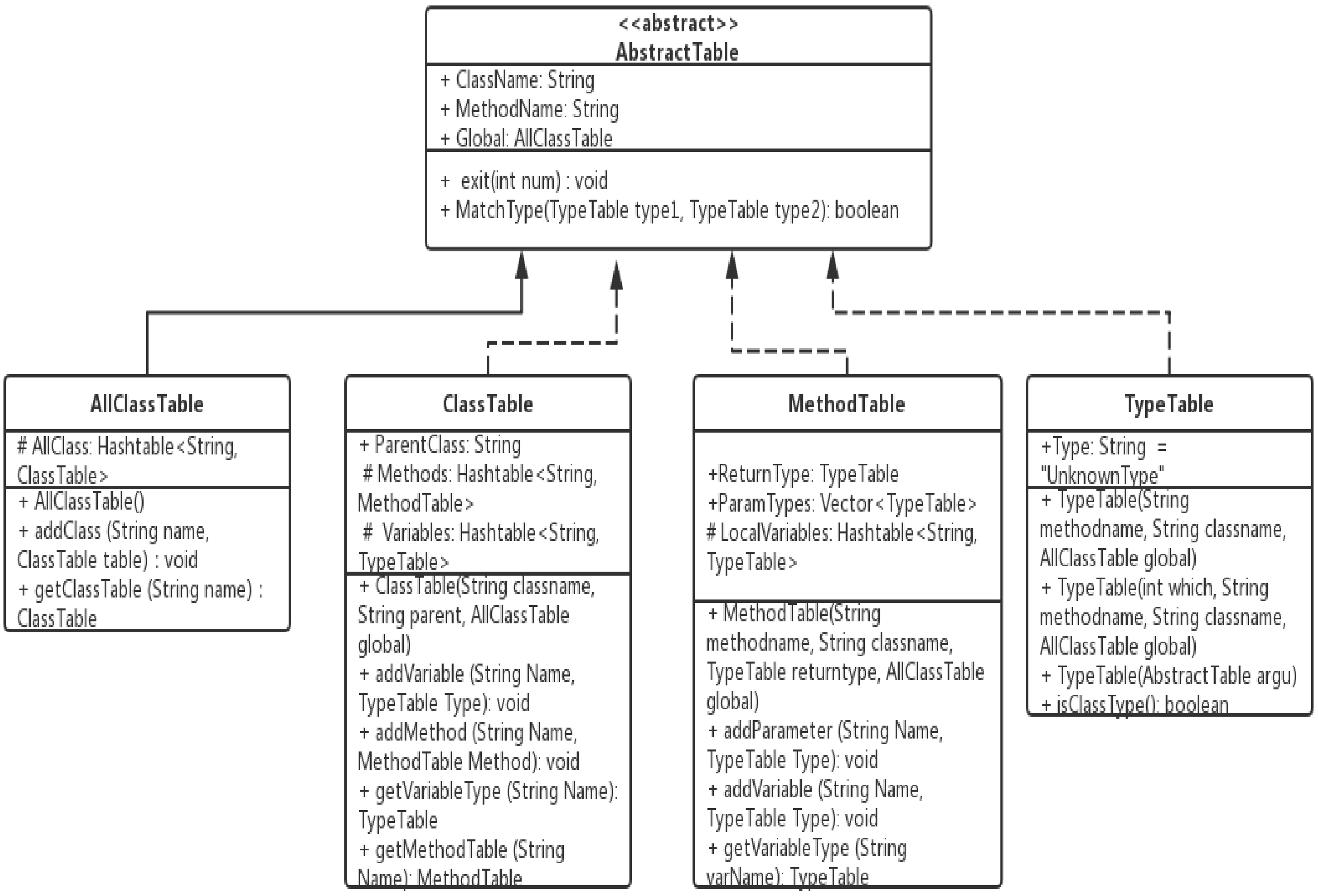

<<abstract>>
AbstractTable
+ ClassName: String
+ MethodName: String
+ Global: AllClassTable
+ exit(int num) : void
+ MatchType(TypeTable type1, TypeTable type2): boolean
AllClassTable
# AllClass: Hashtable<String, ClassTable>
+ AllClassTable()
+ addClass (String name, ClassTable table) : void
+ getClassTable (String name) : ClassTable
ClassTable
+ ParentClass: String
# Methods: Hashtable<String, MethodTable>
# Variables: Hashtable<String, TypeTable>
+ ClassTable(String classname, String parent, AllClassTable global)
+ addVariable (String Name, TypeTable Type): void
+ addMethod (String Name, MethodTable Method): void
+ getVariableType (String Name): TypeTable
+ getMethodTable (String Name): MethodTable
MethodTable
+ReturnType: TypeTable
+ParamTypes: Vector<TypeTable>
# LocalVariables: Hashtable<String, TypeTable>
+ MethodTable(String methodname, String classname, TypeTable returntype, AllClassTable global)
+ addParameter (String Name, TypeTable Type): void
+ addVariable (String Name, TypeTable Type): void
+ getVariableType (String varName): TypeTable
TypeTable
+Type: String = "UnknownType"
+ TypeTable(String methodname, String classname, AllClassTable global)
+ TypeTable(int which, String methodname, String classname, AllClassTable global)
+ TypeTable(AbstractTable argu)
+ isClassType(): boolean

Figure 2-2 Class diagram for symbol table

To enable type checking of MiniJava programs, the symbol table should contain all declared type information:

1) each variable name and formal-parameter name should be bound to its type;

2) each method name should be bound to its parameters, result type, and local variables;

3) each class name should be bound to its variable and method declarations.

or

1) variable name ↦ type

2) formal-parameter name ↦ type

3) method name ↦ {parameters, result type, and local variables}

4) class name ↦ {variable and method declarations}

For example, consider figure 2-3, which shows a program and its symbol table.

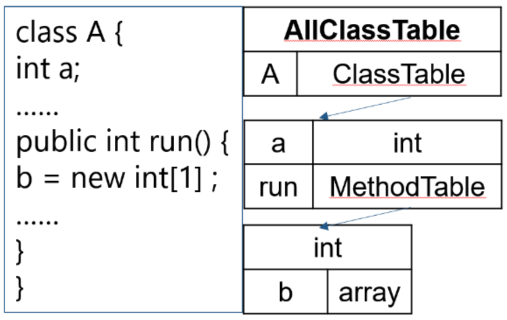


Figure 2-3 An example program and its symbol table

The class “A” is mapped to a ClassTables for its field “a” and method “run”. The Method run is mapped to a MethodTable with both its result type integer and local variable “b”.

The abstract table class defines three instance variables: the class name, the method name, and a AllClassTable(Global). The class name and method name are for the convenience of each ClassTable, MethodTable, and variable type (class members have no method name) when they are created, specifying the class name to which they belong (the ClassTable has no method name) and the method name. The AllClassTable is used to call the AllClassTable at any time to query the class and MethodTables. There are two methods defined in the abstract class, namely the exit function and the MatchType function, which is a comparison of the two variable types.

There are four instance classes, all of which extended from abstract classes, which are AllClassTable, ClassTable, MethodTable, and variable types (the variable type is called TypeTable but it is not a table). Note that in order to facilitate the query, all the data that needs to be stored in this phase is stored using HashTable. The AllClassTable uses a HashTable to store all classes. The storage method is that the class name corresponds to its ClassTable object, and two methods are provided, namely, a method for storing the ClassTable and another for querying the ClassTable. The method for storing the ClassTable can also check whether the class name is repeated. A ClassTable is used to store information about a class, which contains member variables, member methods, and its parent class name for this class. Both member variables and member methods use HashTable, and can store or query the corresponding variable type or

MethodTable based on the variable name or method name. The MethodTable is used to store information about a method, which contains the return type, parameter type, and local variables of the method. Here the parameter type is stored using a vector to compare whether the parameter type matches. The local variable uses HashTable, the storage method is the variable name corresponding to the variable type, it should be noted that the method parameters also belong to the local variable, so the method variable should be added when adding the local variable. The variable type has only one member variable of type String, which means that it represents a variable type. It only needs to represent the type. There is a method defined in the variable type to check if the type is a class type (not a type of int, boolean, and int[]).

2.4 Implementation

As mentioned in Chapter 1.2.2, if we tried to do type checking and intermediate code generation in a single pass, then we might need to type check a call to a method that is not yet put into the symbol table. To avoid such situations, two passes are adopted. Correspondingly, the implementation is divided into two steps. The first step is to construct a symbol table by analyzing the MiniJava program. The second step is type checking.

In the construction of the symbol table, we need to design a visitor to traverse the syntax tree, and store the symbol types of each definition statement one by one. Thus, this visitor inherits the GJDepthFirst class (because it contains definitions of generic incoming parameters and outgoing parameters) and overrides the access methods of the nodes that need to be manipulated.

The generic incoming and outgoing parameters of these visitors are AbstractTable, so that all tables can be passed in to store information and return with stored information.

In this visitor, to traverse and get information of a node only need to accept this node. For example, when traversing to the VarDeclaration node “n” (Figure 2-4), this node “n” is a variable declaration. And node “n: has three child nodes “f0”, “f1” and “f2”, such as “int a”; In this definition, “int” is “f0”, “a” is “f1”, and the semicolon is “f2”. At this point we need the type and name of this variable, the type can be obtained by “n.f0.f0.which”, which is 0, which means the type is int, the name can be obtained by “n.f1.f0.tokenimage”. Finally, we need to save the type and name to the object, then determine whether the incoming “argu” is a ClassTable or a MethodTable, because the variable is definitely defined in a class as a member variable or defined as a local variable in a method, so this The VarDeclaration node must be traversed by a ClassDeclaration node or a MethodDeclaration node, and a ClassTable or MethodTable is passed in the process of the two nodes parse. After judging the properties of “argu”, variables can be added to it.

```
public AbstractTable visit(VarDeclaration n, AbstractTable argu) {
      TypeTable type;
      type   =   new   TypeTable(n.f0.f0.which,   argu.MethodName,
argu.ClassName, argu.Global);
      n.f0.accept(this, type);
    //To check whether this variable is a method local variable
    if(argu instanceof MethodTable)
       ((MethodTable)argu).addVariable(n.f1.f0.tokenImage, type);
    else if(argu instanceof ClassTable)
    ((ClassTable)argu).addVariable(n.f1.f0.tokenImage, type);
    n.f1.accept(this, argu);
    n.f2.accept(this, argu);
    return type;
    }
```

Figure 2-4 Visiting VarDeclaration node

The above process implements traversing to a variable definition node and storing the type and name of the variable definition in the corresponding class node or method node.

In the process of constructing the table, we not only need to pay attention to various Declaration nodes, but also pay attention to the Identifier node. This is because the specific class of the definition of the class object cannot be judged in the previous process. For example, there is a class. “A”, a variable “b” is defined in the code with “A”, which is “A b”. In VarDeclaration (Figure 2-4), its class name cannot be simply judged by judging “which”, because “which”s of class variables are all 3, so when traversing the “f0” of VarDeclaration, it will traverse to an Identifier, and the second parameter passed in is TypeTable, so this Identifier is a variable type. But in the visit method of the Identifier node (Figure 2-5), it is only necessary to determine whether the parameter passed is a TypeTable. If so, it indicates that this is a class name, and then the class name is saved to the TypeTable and passed to the previous VarDeclaration node, and the VarDeclaration node is the b variable and the A type can be saved to the corresponding ClassTable or MethodTable, and the definition of the class variable is saved.

```
    public AbstractTable visit(Identifier n, AbstractTable argu) {
        //Only if the ID's type is TypeTable can edit, if it's
MethodName or ClassName, it wouldn't work
        if(argu          instanceof          TypeTable          &&
((TypeTable)argu).isClassType())
            ((TypeTable)argu).Type = n.f0.tokenImage;
        n.f0.accept(this, argu);
        return null;
    }
```

Figure 2-5 Visiting Identifier node

After traversing all the nodes and adding information to the key nodes, we complete the construction of the symbol table, followed by type checking. In the work of type check, we also need to design a visitor to traverse the syntax tree and check for errors in the code during this traversal. This visitor is also extends from the GJDepthFirst class, because we also need to pass in the global table we just saved.

For example, in the most complex node of all——MessageSend node (Figure 2-6), there are five child nodes, “f0” to “f5”. Where “f0” is a PrimaryExpression, “f2” is an Identifier, and “f4” is an ExpressionList. In fact, this node is a method using such as “f0.f2(f4)”. At this point we need to first determine whether this “f0” is a class variable, first set a TypeTable type variable and pass it to the accept method of “f0” to get its type, and then judge whether the type is a class type, if not, then return the type error. Then determine whether the class type has been defined, and if so, determine whether the “f2” method exists, and if it exists, continue. Then set a MethodTable type variable and pass it to the accept method of “f4”, a temporary MethodTable object can be obtained with the parameters whose type are the same as ExpressionList, and determine whether this object’s parameters match parameters type of the “f2” method object. If all of them matches, the type check of the MessageSend node is completed, and the TypeTable of the incoming argu is set to the return type of the “f2” method, because it may be judged by the upper node.

```
    public String visit(MessageSend n, AbstractTable argu) {
        TypeTable exp = new TypeTable(argu);
        String str1 = n.f0.accept(this, exp);
        MethodTable method = null;
        if(!(exp.isClassType())) //If exp is not class
            argu.exit(1);
        else {
            ClassTable temp = argu.Global.getClassTable(exp.Type);
            if(temp == null)  //If there's no exp's declaration
                argu.exit(1);
            else {
                method = temp.getMethodTable(n.f2.f0.tokenImage);
                if(method == null)   //If  there's  no  this  method's
declaration
                    argu.exit(1);
            }
        }
        TypeTable type = new TypeTable(argu);
        n.f1.accept(this, argu);
        String str2 = n.f2.accept(this, type);
        n.f3.accept(this, argu);
        //Get a temp method as f4's parameters formal and compare with
calling method
        MethodTable  tempmethod  =  new  MethodTable(argu.MethodName,
argu.ClassName, method.ReturnType, argu.Global);
        String str3 = n.f4.accept(this, tempmethod);
        //Check whether two method's parameters number same
        if(method.ParamTypes.size() != tempmethod.ParamTypes.size())
            argu.exit(1);
        //Check whether each parameter in two method match
        for(int i = 0; i < method.ParamTypes.size(); i++)
            if(!argu.MatchType(method.ParamTypes.get(i),
tempmethod.ParamTypes.get(i)))
                argu.exit(1);
        ((TypeTable)argu).Type = method.ReturnType.Type;
        n.f5.accept(this, argu);
        return str1 + "." + str2 + "(" + str3 + ")";
                                        }
```

Figure 2-6 Visiting MessageSend node

Another example is about the following statement: “i=i+(i+1)” (Figure 2-7).

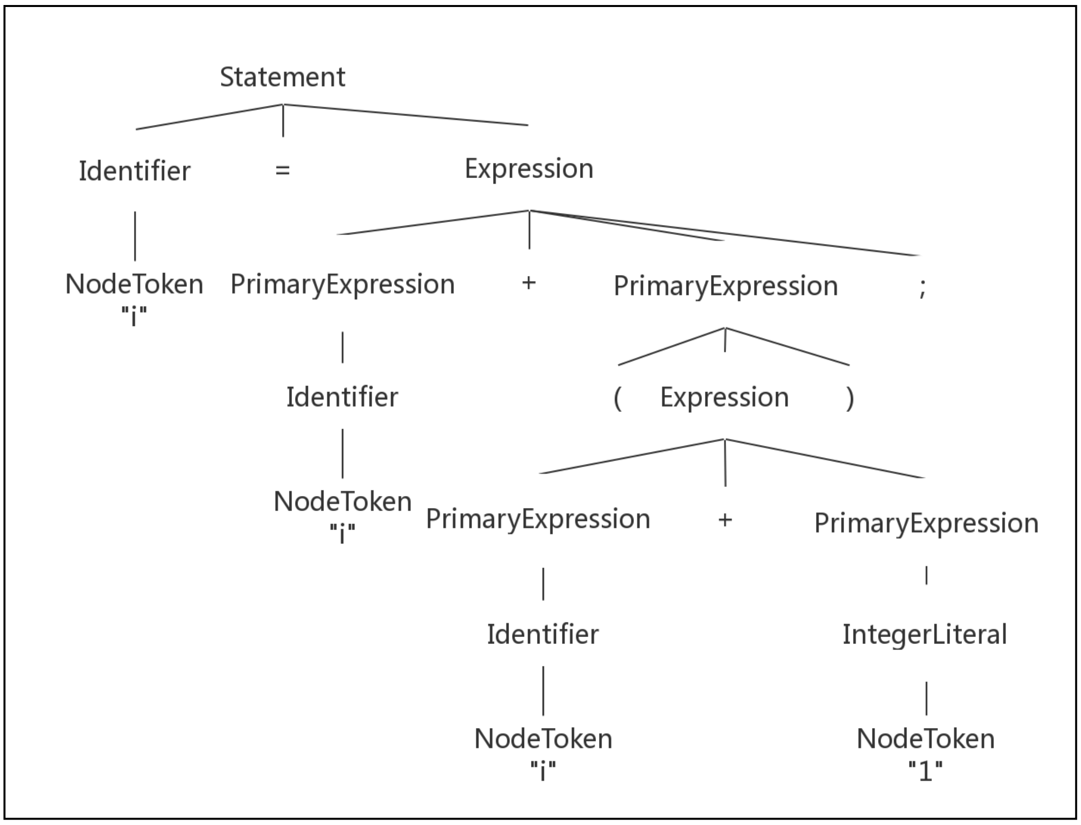
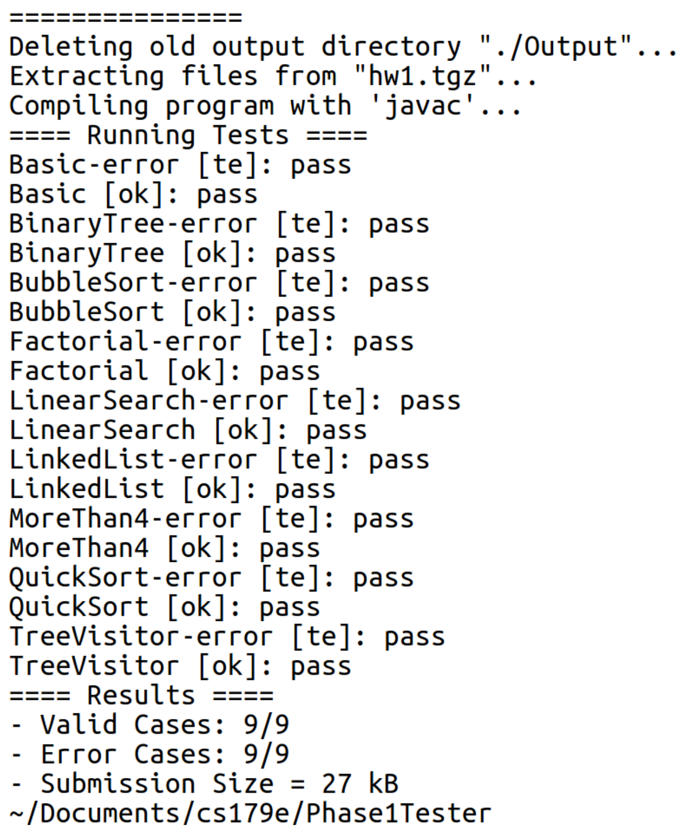


Figure 2-7 Parse tree for the statement “i=i+(i+1)”

This is a recursive process. Each Expression node has undergone a visit and accept process, and each time they visit and accept the node, the TypeTable is passed in order to represent the type of the node. In the override method, this process is also implemented. After TypeTable types of the left and right nodes of the bottom of the PlusExpression are both determined and are int types, the int type is returned as the TypeTable of the node, and then in the previous PlusExpression, the type on the left has determined as the int type, and the type on the right has just been determined as int. The TypeTable of this node is also set to the int type. Finally, in the topmost AssignmentStatement node, we have determined the type determined by the identifier on the left, and the type of the Expression on the right is just determined. If they are the same, the statement node is checked.

2.5 Test Results

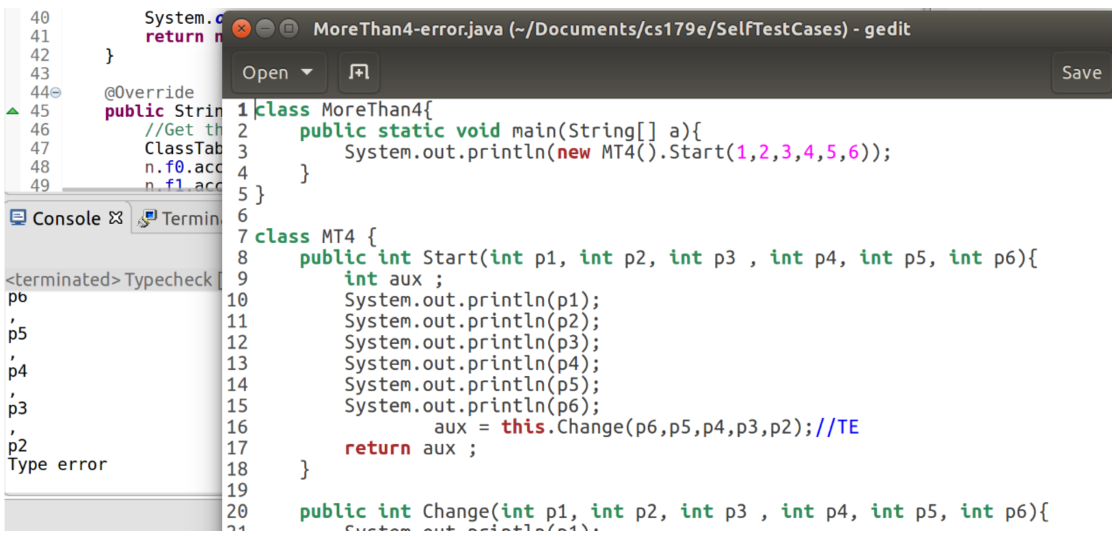
Firstly, the result for the Phase1Tester[8] is shown as figure 2-8.



```
===============
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 = 27 kB
~/Documents/cs179e/Phase1Tester
```

Figure 2-8 Result for the Phase1Tester

Secondly, by overriding the visit method of NodeToken, it can print out each NodeToken visited for debugging, thus we can know what nodes are traversed. In this way, we can also determine whether the location of the program error during the inspection is consistent with the preset error point. The results are shown in the figure 2-9 and figure 2-10, it can be seen that the return nodes of the wrong types are exactly the same as the preset nodes.



MoreThan4-error.java (~/Documents/cs179e/SelfTestCases) - gedit
Open
Save
Console
Terminal
<terminated> Typecheck [

```
        System.c
        return n
    }

    @Override
    public Strin
        //Get th
        ClassTab
        n.f0.acc
        n.f1.acc
```

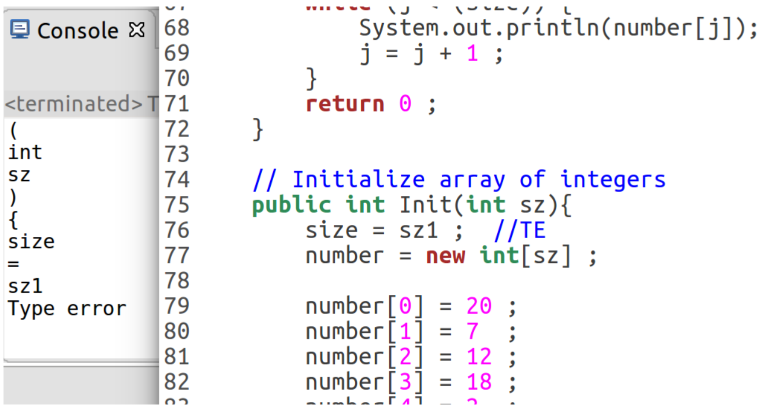
```
p6
,
p5
,
p4
,
p3
,
p2
Type error
```

```
class MoreThan4{
    public static void main(String[] a){
        System.out.println(new MT4().Start(1,2,3,4,5,6));
    }
}

class MT4 {
    public int Start(int p1, int p2, int p3 , int p4, int p5, int p6){
        int aux ;
        System.out.println(p1);
        System.out.println(p2);
        System.out.println(p3);
        System.out.println(p4);
        System.out.println(p5);
        System.out.println(p6);
            aux = this.Change(p6,p5,p4,p3,p2);//TE
        return aux ;
    }

    public int Change(int p1, int p2, int p3 , int p4, int p5, int p6){
```

Figure 2-9 Type error caused by mismatch of parameters for Change method



```
<terminated> T
(
int
sz
)
{
size
=
sz1
Type error
```

```
            System.out.println(number[j]);
            j = j + 1 ;
        }
        return 0 ;
    }

    // Initialize array of integers
    public int Init(int sz){
        size = sz1 ;  //TE
        number = new int[sz] ;

        number[0] = 20 ;
        number[1] = 7  ;
        number[2] = 12 ;
        number[3] = 18 ;
```

Figure 2-10 Type error caused by unknown “sz1”

2.6 Conclusion

To sum up, in this phase two visitors extend GJDepthFirst: FirstVisitor and SecondVisitor to type check a MiniJava program. The former is used to visits nodes in the syntax tree, builds a symbol table, which stores all the defined variables, methods, and classes, that is, constructs a symbol table so that the program can use it to query existed variables, methods, and classes at anytime, anywhere during the second visit; and detects redeclaration error. The latter consults symbol table to type check the statements and expressions, then either approves or rejects the program.

Although result seems to be positive regards to the Phase1Tester, improvement about error handling can be made in the future. Specifically, when the type-checker detects a type error or an undeclared identifier, it should print an appropriate error message for each error. After that, it should recover an error as if a valid expression had been encountered and continue, because the programmer would like to be told of all the errors in the program[11].

3 Intermediate Code Generation

3.1 Introduction

The goal of phase two is to translate programs in the MiniJava language to programs in a low level assembly-like language—Vapor language. Vapor programs are described as a list of functions and data segments. Since in contrast to architecutal assembly languages that support a finite number of registers, Vapor functions can use an unbounded number of variables, the main challenge in the translation from MiniJava to Vapor lies in mapping objects to memory and object-oriented method calls to function calls. We firstly created a symbol table by analyzing the MiniJava program which is similar to the first step in the previous phase. Then we generated intermediate Vapor code according to that symbol table.

3.2 Vapor Language

Vapor program is a set of functions and data segments. Vapor has two types of global data segments. A const segment is for read-only data (like virtual function tables). A var segment is for global mutable data. Each section starts with a data label and is followed by static data values. Each entry in a data segment is four bytes long. The syntax for a function definition is shown in figure 3-1[12].

```
func FunctionLabel(params...)
    body...

where Each line of the body of a function is one of:
•  code label: Label:
•  assignment: Location = Value
•  branch: if Value goto CodeAddress
•  goto: goto CodeAddress
•  function call: call FunctionAddress (Args...)
•  function return: ret Value
•  call to built-in operation: OpName (Args...)
```

Figure 3-1 Syntax for a function definition

3.3 Methodology

To translate programs in the MiniJava language to programs in the Vapor language, the overall procedure can be divided into two steps. The first step is almost the same as the first step in phase one (Chapter 2.4), which is constructing a symbol table by analyzing the MiniJava program. The second step is making use of the symbol table constructed in the first step to generate intermediate Vapor code.

Compared to the symbol table in the phase one, the symbol table in this phase was slightly modified, for example, in the following two aspects:

1) Changing the type of hash style in the ClassTable from HashTable to LinkedHashMap, thereby ensuring the sequence of key-value pair of methods is the same as the sequence that methods are stored in the ClassTable.
2) It's convenient to put methods which add new classes and methods into the AllClassTable when calling them.

In the second step, we built four classes: Counter, Exp, Printer, and Generator. The aim of class Counter is keeping int type counter and flags, for instance, field tCount is applied to calculate the value of num of all t.num variables in each method, field indentation is devoted to calculating the degree of indentation of codes, and field boolean alloc is to indicate whether the method arrayallocz and its feature codes need printing. The class Exp records the corresponding type which each Expression belongs to. This is because expressions are not allowed being nested in the grammar of Vapor language. If there is a expression which consists of two primary-expressions in a MiniJava program, since a primary-expression trivially belongs to the expression, we need firstly change the primary-expression on the right-hand side to a t.num variable, then perform operation on the primary-expression on the left-hand side and t.num bariable on the right-hand side. In this context, the class Exp appears to be attractive as it stores the type of expressions, the value or string of expressions, and the class an expression belongs to. Whenever we need check which type an expression belongs to or compare expressions, we can call class Exp in class Generator which will be discussed below. The class Printer is used to store all kinds of print methods, which are based on customized formatting print method printf(), and three methods which are likely to be used repeatedly: printVar(), printNullPointer() and printOutOfBounds(), which are used to print expression on the right-hand side when nested as mentioned above, error about NULL pointer, and contents regarding array respectively. The class Generator extending the class DepthFirstVisitor is the most important class in this phase. Generator receives symbol table and initializes counter and printer during the initialization, then the vapor code can be generated by it.

3.4 Implementation

Here we focus on several visiting methods and sketch how to implement them. In "public void visit(MainClass n)" method (Figure 3-2), the first thing we need to do is to establish a list of functions. Thus, we firstly traverse HashTable in AllClassTable. As mentioned in previous phase, the AllClassTable uses a HashTable to store all classes whose names correspond to their ClassTable object. Therefore, after traverse, we can obtain ClassTable objects of all classes, and then obtain the name of each class through its ClassTable. In the vapor language, every class owns their functions. Thus, when traversing a specific class, we need enter its ClassTable searching every MethodTable to gain information of all its functions. In this way, we can format print a list of functions of that class. What should be highlighted is that we need control the indentation all the time so that a more readable code can be generated. Finally, the method will print information of Main function.

```
public void visit(MainClass n) {
        //At first, print the classes except main class
        for (ClassTable classtable : Global.AllClass.values()) {
            if (!classtable.ClassName.equals(n.f1.f0.tokenImage) ) {
                printer.printf("const vmt_%s", classtable.ClassName);
                counter.indentation++;
                //for each class, print its methods
                for (String methodName : classtable.Methods.keySet())
                    printer.printf(":%s.%s",     classtable.ClassName,
methodName);
                counter.indentation--;
                System.out.println();
            }
        }
        //print the main function
        printer.printf("func Main()");
        counter.indentation++;
        //start recursion
        n.f15.accept(this);
        //At last, print the ret
        printer.printf("ret");
        counter.indentation--;
    }
```

Figure 3-2   Code for visiting MainClass method

As for “public void visit(MethodDeclaration n)” method (Figure 3-3), we need construct declaration of a method. At the beginning, there is a series of initialization for that method, including initializing tCount to 0, which is because t.num of every method counts from 0, and adjust format of indentation. Next, through the method accept undergo iteration. After the iteration, namely, at the end of that method, we print the content related to return value. A declaration of a method accomplished.

```
public void visit(MethodDeclaration n) {
        System.out.println();
        MethodTable                     methodtable                     =
Global.nowclass.Methods.get(n.f2.f0.tokenImage);
        //set the working methodtable
        Global.nowmethod = methodtable;
        StringBuilder str = new StringBuilder();
        //print the params of a method
        for (String p : methodtable.Params.keySet()) str.append(" "
+ p);
        printer.printf("func                            %s.%s(this%s)",
Global.nowclass.ClassName,                    methodtable.MethodName,
str.toString());
        //for each method, vapor code need to reset the t count for
variables
        counter.tCount = 0;
        counter.indentation++;
        n.f8.accept(this);
        n.f10.accept(this);
        printer.printf("ret %s", printer.printVar(temp));
        counter.indentation--;
        Global.nowmethod = null;
    }
```

Figure 3-3   Code for visiting MethodDeclaration method

Let’s have a closer look at the iteration. During the iterative visit process, different statements and expressions will be visited. For example, as for ifstatement (Figure 3-4) consisting of expression(f2), if statement(f4) and else statement(f6). Because vapor language can’t make expression as if expression, the if can only judge the variable like “t.num”, so we need to get “t.num = if expression” firstly and then put the t.num in the beginning of if statement. Then get into iteration of if statement and else statement. At last, we need to add the “if end” code.

```
public void visit(IfStatement n) {
       //get real if expression
       n.f2.accept(this);
       String elsestat = "if1_else_" + counter.ifelse++;
       String endstat = "if1_end_" + counter.ifend++;
       //print if expression like "t.num expression"
       String condVar = printer.printVar(temp);
       //print if expression like "if0 t.num goto : somewhere"
       printer.printf("if0 %s goto :%s", condVar, elsestat);
       counter.indentation++;
       //print if statement
       n.f4.accept(this);
       printer.printf("goto :%s", endstat);
       counter.indentation--;
       printer.printf("%s:",  elsestat);
       counter.indentation++;
       //print else statment
       n.f6.accept(this);
       counter.indentation--;
       printer.printf("%s:", endstat);
   }
```

Figure 3-4 Code for visiting IfStatement method

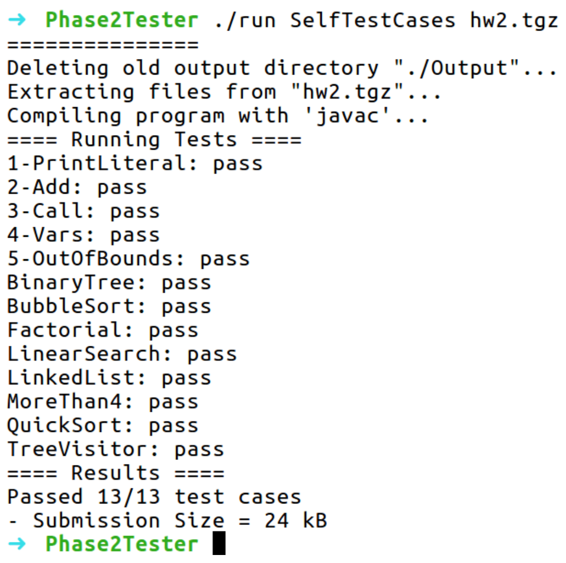
A complicated example about expression is and expression (Figure 3-5). The and expression can be translated to two if0 instructions that check if each of the operands is zero, assign zero to the result and goto the end label. Before the end label, one is assigned to the result. The not expression can be similarly translated. To implement this process, we need firstly gain the left-hand side expression in and expression and keep it as a t.num, then use if statement to judge this t.num. Similarly, we gain the right-hand side expression in and expression and keep it as a t.num. Finally, we use two if and else judgement to realize the whole process. It should be pointed out that after the process of handling and expression, the resulting expression need passing on to the upper level. That's to say, storing the resulting expression in the Exp temp for the visitor of upper level convenience of using.

```
public void visit(AndExpression n) {
       //get left expression
       n.f0.accept(this);
       Exp left = temp;
       String andElseLabel = "and_else_" + counter.andelse++;
       String andEndLabel = "and_end_" + counter.andend++;
       //print left expression like "t.num = expression"
       String tleft = printer.printVar(left);
       //print first if expression like "if0 t.num goto : somewhere"
       printer.printf("if0 %s goto :%s", tleft, andElseLabel);
       counter.indentation++;
       //get right expression
       n.f2.accept(this);
       Exp right = temp;
       //print right expression like "t.num = expression"
       String tright = printer.printVar(right);
       //give result a "t.num" label
       String res = "t." + counter.tCount++;
       //print this expression like "t.num(result) = t.num(right
expression)"
       printer.printf("%s = %s", res, tright);
       printer.printf("goto :%s", andEndLabel);
       counter.indentation--;
       printer.printf("%s:", andElseLabel);
       printer.printf("%s = 0", res);
       printer.printf("%s:", andEndLabel);
       //save this expression and pass to next visitor
       temp = new Exp(res, Exp.Type.ID);
   }
```

Figure 3-5 Code for visiting AndExpression method

3.5 Test Results

The result for the Phase2Tester[8] is shown as figure 3-6. All passes indicate the Vapor programs translated from preset MiniJava programs by our translator are exactly consistent with the Vapor language specification.



```
➜  Phase2Tester ./run SelfTestCases hw2.tgz
===============
Deleting old output directory "./Output"...
Extracting files from "hw2.tgz"...
Compiling program with 'javac'...
==== Running Tests ====
1-PrintLiteral: pass
2-Add: pass
3-Call: pass
4-Vars: pass
5-OutOfBounds: pass
BinaryTree: pass
BubbleSort: pass
Factorial: pass
LinearSearch: pass
LinkedList: pass
MoreThan4: pass
QuickSort: pass
TreeVisitor: pass
==== Results ====
Passed 13/13 test cases
- Submission Size = 24 kB
➜  Phase2Tester █
```

Figure 3-6   Result for the Phase2Tester

4 Register Allocation

4.1 Introduction

The goal of phase three is to translate programs in the Vapor language to programs in the Vapor-M language. Since Vapor-M provides registers and stacks rather than local variables, the local variables provided by Vapor should be mapped to registers and run-time stack elements. We firstly created a register table by data flow analysis of each function in a Vapor program. Then we generated Vapor-M code according to that register table. The main challenge in the translation from Vapor to Vapor-M is Vapor-M Language Specification and considering the finite registers during arguments passing.

4.2 Vapor-M Language

A Vapor-M program is almost the same as a Vapor program except that instead of local variables, registers and stack memory are used. Rather than local variables we have 23 registers: $s0..$s7, $t0..$t8, $a0..$a3, $v0, $v1. Registers are global to all functions (whereas local variables were local to a function activation). To follow MIPS calling conventions in phase 4 discussed in Chapter 5, we use the registers as table 4-1 indicates.

Table 4-1 Usage of registers in Vapor-M

Register(s)	Usage
$s0..$s7	general use callee-saved
$t0..$t8	general use caller-saved
$a0..$a3	reserved for argument passing
$v0	returning a result from a call
$v0, $v1	can be used as temporary registers for loading values from the stack

Each function has three stack arrays called in, out, and locals. The in and out arrays are for passing arguments between functions. The in array refers to the out array of the caller. The local array is for function-local storage that can be used for spilled registers.

The sizes of these arrays are declared at the top of every function (instead of a parameter list). Each element of each array is a 4-byte word. The indexes into the array is the word-offset (not the byte offset). Array references can be used wherever memory references can be used. So in[1] refers to the second element of the in stack array[9].

4.3 Register Table

To translate programs in the Vapor language to programs in the Vapor-M language, the overall procedure can be divided into two parts. The first part is constructing a register table, which maps variables to registers and function-local (spilled registers). The second part is making use of the register table constructed in the first step to generate Vapor-M code.

Each function has a register table, which maps variables to registers and function-local (spilled registers). Considering the data structure, we used the varNode class to store information of each variable (this, t.n, aux, etc.) in Vapor code. Information recorded in the varNode class includes the name of the variable, its corresponding register, live interval (specify the range by the number of line), and other information needed by visitor pattern in the first part. Register table connected these varNodes acting as a linked list to support the translation in the second part.

4.4 Linear Scan Algorithm

In the first part, we assign four tables——varTable, varList, localVarList, and spilledParamList to each function. For example, as to the first function, i.e. the function with the VFunction.index equal to 0, localVarList should store varNodes whose corresponding variable cannot be stored in registers due to the limitation that all registers are not available, thus can be only stored in the local array. VarTable is a HashMap which should store names of all variables in this function and their corresponding varNodes. VarList is similar to the varTable but records all variables in the order of sequence of their appearance. spilledParamList The purpose of the first part is to fulfill information of each variable in the function from the beginning to the end.

It can be divided into 3 steps to maintain information of every variables in a function. The first step is to find out all variables in a function and their corresponding live ranges, which can be pictured as figure 4-1 below. The live ranges can be calculated using the active sets by a top-down scan of the instructions.

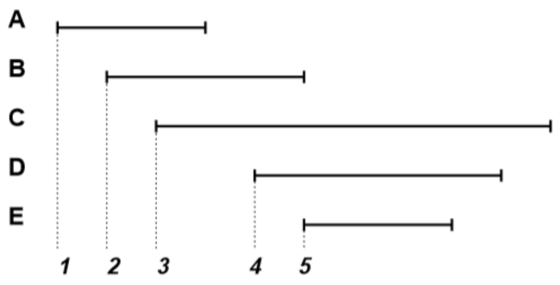
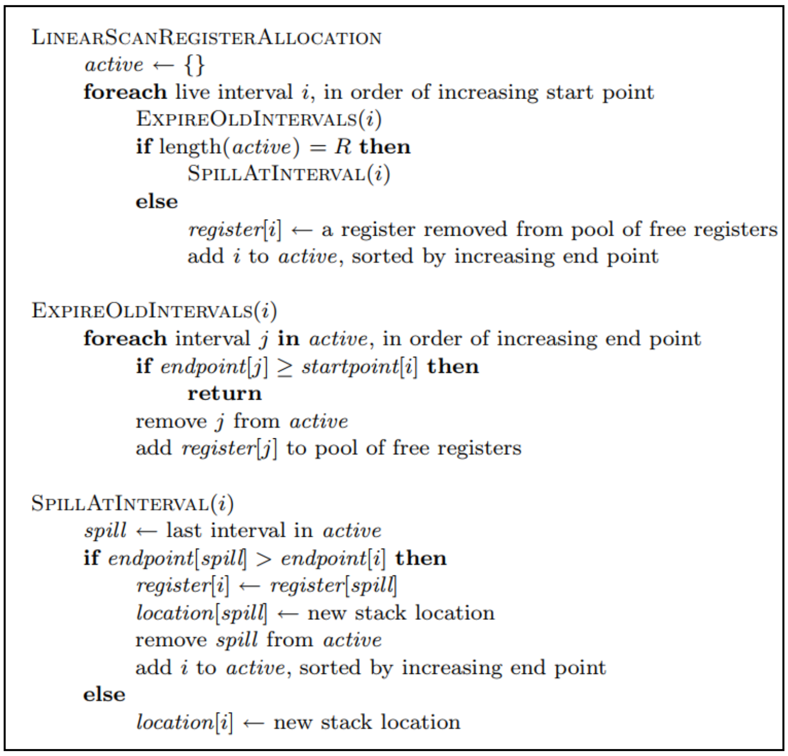


Figure 4-1 An example for liveness analysis (The number of line indicates the liveness period)

In the second step, we then used the linear scan algorithm from Massimiliano Poletto and Vivek Sarkar' paper[13] for register allocation. It is easy to implement this algorithm as figure 4-2 shows below since we have already obtained live intervals in the first step.



```
LINEARSCANREGISTERALLOCATION
    $active \leftarrow \{\}$
    foreach live interval $i$, in order of increasing start point
        EXPIREOLDINTERVALS($i$)
        if length($active$) = $R$ then
            SPILLATINTERVAL($i$)
        else
            $register[i] \leftarrow$ a register removed from pool of free registers
            add $i$ to $active$, sorted by increasing end point

EXPIREOLDINTERVALS($i$)
    foreach interval $j$ in $active$, in order of increasing end point
        if $endpoint[j] \geq startpoint[i]$ then
            return
        remove $j$ from $active$
        add $register[j]$ to pool of free registers

SPILLATINTERVAL($i$)
    $spill \leftarrow$ last interval in $active$
    if $endpoint[spill] > endpoint[i]$ then
        $register[i] \leftarrow register[spill]$
        $location[spill] \leftarrow$ new stack location
        remove $spill$ from $active$
        add $i$ to $active$, sorted by increasing end point
    else
        $location[i] \leftarrow$ new stack location
```

Figure 4-2 Pseudo-code for linear scan algorithm

For example, the implementation for LINEARSCANREGISTERALLOCATION function is shown in figure 4-3 below.

```
        // LinearRegisterAllocation Start
        // active <- {}
        LinkedList<VarNode> active = new LinkedList<VarNode>();
        int funcIndex = 0;
        for (LinkedList<VarNode> varList : varLists) {
            // For each function, linear scan it initially
            workIndex = funcIndex++;
            for (VarNode var : varList) {
                // For each live interval (variable)
                expireOldIntervals(var, active);
                // Note that we need to know whether this variable
overlap a call instuction
                if  (active.size()  ==  17  ||  (var.overLapCall  &&
freeSRegs == 0)) {
                    spillAtInterval(var, active);
                } else {
                    // A register removed from pool of free registers
                    var.register = getFreeReg(var.overLapCall);
                    // Add variable to active set, sorted by increasing
end point
                    increasingAdd(var, active);
                }
            }
        }
```

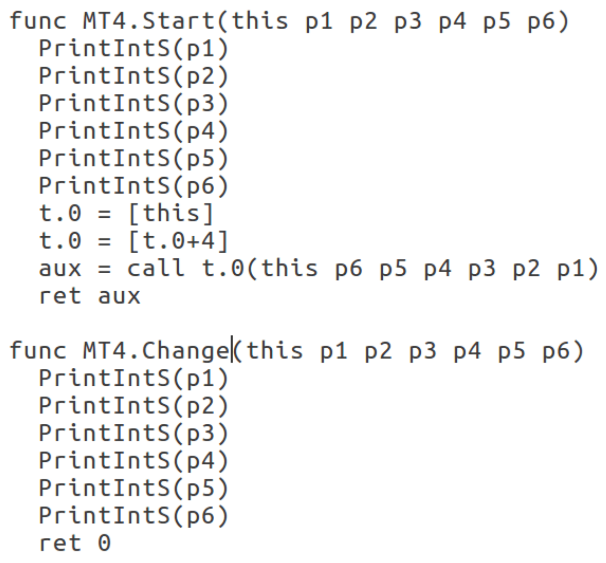
Figure 4-3 Implementation for LINEARSCANREGISTERALLOCATION function

Besides the three functions metioned in the algorithm, to meet the requirement, adding interval i to active and being sorted by increasing end point or sorting intervals in order of start point, we need designe an increasingAdd function to insert intervals.

Finally in the third step, we combined information collected after register allocation to construct register table.

It should be noticed that this phase also used visitor pattern, and extended VInstr.Visitor. However, in contrast to visitor pattern for MiniJava that traverses the syntax tree from the root, the program visits Vapor code line by line. This is because the aim of the first step is to gain live ranges, thus we only need focus on variables in each line of the Vapor code.

In the second part, translation was mainly based on variable and register pairs obtained from linear scan or local information table, which is the hash table of variable and register pairs changed from that of variable and varNode pairs. Through visitor pattern visiting each line of Vapor code, variables such as this and t.n in that line will be translated into the form in registers or local. During the initialization of translation, global variables in the local, or namely, in the stack, and code related to the offset of functions should be retained in their original form. We regarded the smallest unit during the translation part as the same as in the previous part——function. For each function, we should firstly check the number of parameters passed in in case that we need adjust the value of in, out and local, which was indicated by MoreThan4.vapor test. If the number of parameters passed in n is greater than 4, that’s to say, n is greater than the number of $a registers reserved for argument passing, then the value of in should be n-4 and the value of out should be the number of parameters passed to other functions called in current function minus 4. Take the MoreThan4.vapor (Figure 4-4) as an example, in and out for MT4.Start function both are 3. This is because this function receives 7 parameters but the first 4 parameters have already used 4 $a registers, thus the last 3 parameters should be saved in in. Meanwhile, MT4.Change function also needed 7 parameters is called in this function, thus before calling that function at the time of register allocation, 4 parameters passed to MT4.Change function should be saved in $a registers, and the remaining 3 parameters then are saved in out. As for MT4.Change function, similarly, the value of in is 3 as analyzed before. Since this function doesn’t call other function whose parameters are greater than 4, the value of out is 0.

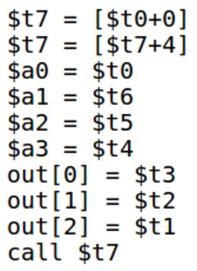


```
func MT4.Start(this p1 p2 p3 p4 p5 p6)
  PrintIntS(p1)
  PrintIntS(p2)
  PrintIntS(p3)
  PrintIntS(p4)
  PrintIntS(p5)
  PrintIntS(p6)
  t.0 = [this]
  t.0 = [t.0+4]
  aux = call t.0(this p6 p5 p4 p3 p2 p1)
  ret aux

func MT4.Change(this p1 p2 p3 p4 p5 p6)
  PrintIntS(p1)
  PrintIntS(p2)
  PrintIntS(p3)
  PrintIntS(p4)
  PrintIntS(p5)
  PrintIntS(p6)
  ret 0
```

Figure 4-4　Part of Vapor code for MoreThan4 program

Part of Vapor-M code for MoreThan4.vapor is shown in figure 4-5 below, where $t7 is MT4.Change function. At this time 7 parameters should be passed in, 4 of which are passed in through registers, and the remaining 3 of them are passed through out. Here the out of MT4.Start is just the in of MT4.Change.



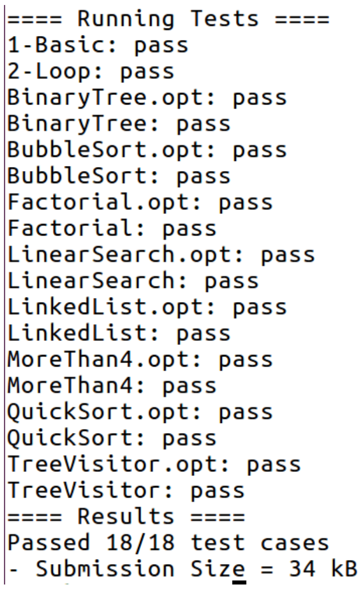
```
$t7 = [$t0+0]
$t7 = [$t7+4]
$a0 = $t0
$a1 = $t6
$a2 = $t5
$a3 = $t4
out[0] = $t3
out[1] = $t2
out[2] = $t1
call $t7
```

Figure 4-5　Part of Vapor-M code for MoreThan4 program

Eventually it is the time to translate instructions in functions through visitor pattern. Generally, when there are enough available registers, we can directly replace the variables with registers. But special cases may occur, for example, when visiting VCall (call function in vapor) instruction, we need check the number of parameters passed in call firstly. The first 4 parameters are saved in 4 $a registers. If the number of parameters is greater than 4, then from the fifth parameter to the last one should be passed out through out. Meanwhile, when the call instruction is translated, we need check if the address of input is for a register. If not for a register but for a local, then the local value should be passed to $v0 register, then $v0 register should be called, finally save the return value to $v0 register.

4.5 Test Results

The result for the Phase3Tester[8] is shown as figure 4-6. All passes indicate that the Vapor-M programs translated from preset Vapor programs are exactly consistent with the Vapor-M language specification.



```
==== Running Tests ====
1-Basic: pass
2-Loop: pass
BinaryTree.opt: pass
BinaryTree: pass
BubbleSort.opt: pass
BubbleSort: pass
Factorial.opt: pass
Factorial: pass
LinearSearch.opt: pass
LinearSearch: pass
LinkedList.opt: pass
LinkedList: pass
MoreThan4.opt: pass
MoreThan4: pass
QuickSort.opt: pass
QuickSort: pass
TreeVisitor.opt: pass
TreeVisitor: pass
==== Results ====
Passed 18/18 test cases
- Submission Size = 34 kB
```

Figure 4-6 Result for the Phase3Tester

5 Activation Records and Instruction Selection

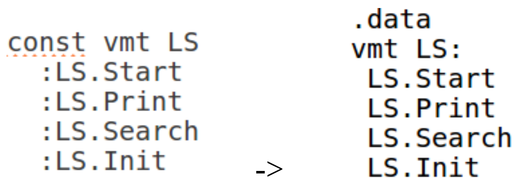
5.1 Introduction

In this phase, the Vapor-M registers and stacks should be mapped to MIPS registers and runtime stack. In addition, the Vapor-M instructions should be mapped to MIPS instructions[14]. Translating a Vapor-M program to a MIPS program is the simplest phase among four phases since we only need to visit the program once and do some simple operation. The main challenge is about the understanding and implementation of the stack frame constructing, the computation of stack offset and computation related to in and out. During the implementation, we need be familiar with the developer document, so we can find small skill of every class in this parser through the document, such as finding the offset of the stack itself.

5.2 Implementation

To translate a Vapor-M program to a MIPS program, the overall procedure can be divided into three steps. The first step is according to the MIPS instruction descriptions translating the .data segment and .text segment, which are known before visited. The second step used visitor pattern to visit the Vapor-M code line by line. Different from visitors in phase three, here the visitor requires registers string array as the parameter. Namely, parser will parse these strings as registers. The third step is to translate strings related to align after visited.

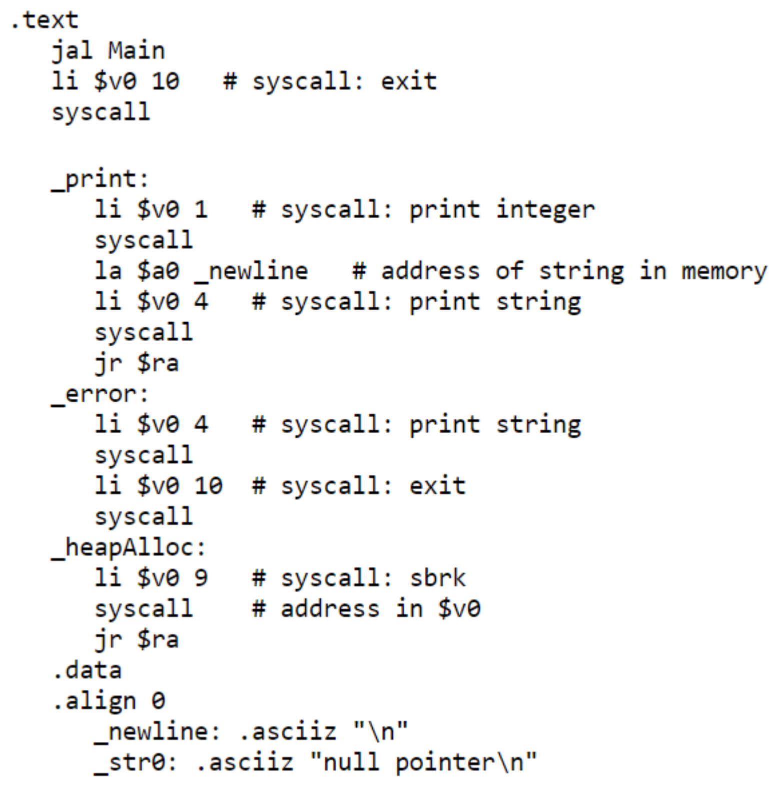
As for the first step, we only need the information from parser to translate line by line. Consider the .data segment at first as the figure 5-1 shows below, it should be noticed that there is a ‘:’ at the beginning of each line within the data in Vapor-M language, thus what should we do is just remove those ‘:’.



```
                            .data
const vmt LS                vmt LS:
   :LS.Start                 LS.Start
   :LS.Print                 LS.Print
   :LS.Search                LS.Search
   :LS.Init          ->      LS.Init
```

Figure 5-1 The left is data segment for Vapor-M program LinearSearch, the right is MIPS instructions after translation.

Next for the .text segment (code segment), there is a standard format for the beginning of .text segment in the MIPS. At first we enter the Main function and print the “jal Main” directly. Then for the normal exit from a program, it should print “li $v0 10” and “syscall” directly. Now the return value of the program store in $v0, although it will never be used in the phase. Because we will use three syscalls: print, error and heapAlloc in the following translation, we put these syscalls in the .text segment as shown in figure 5-2.

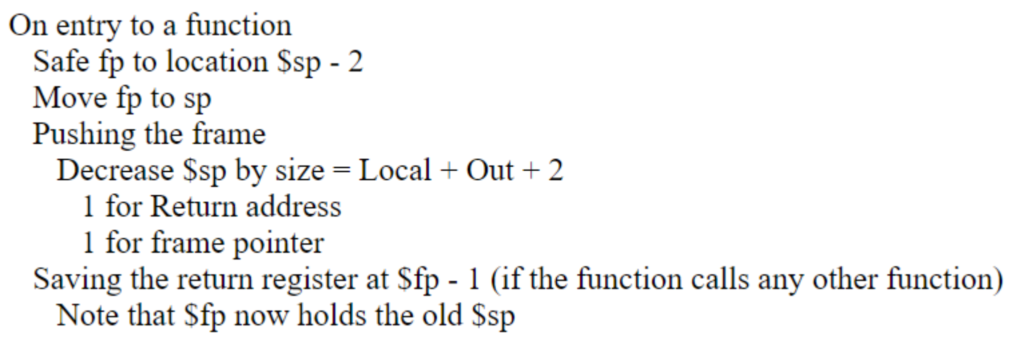


```
.text
   jal Main
   li $v0 10   # syscall: exit
   syscall

   _print:
      li $v0 1   # syscall: print integer
      syscall
      la $a0 _newline   # address of string in memory
      li $v0 4   # syscall: print string
      syscall
      jr $ra
   _error:
      li $v0 4   # syscall: print string
      syscall
      li $v0 10  # syscall: exit
      syscall
   _heapAlloc:
      li $v0 9   # syscall: sbrk
      syscall    # address in $v0
      jr $ra
   .data
   .align 0
      _newline: .asciiz "\n"
      _str0: .asciiz "null pointer\n"
```

Figure 5-2 Text segment includes syscalls: print, error and heapAlloc.

In the second step, we need translate functions. In each function, there is also a standard heading, which conducts operations as shown in the figure 5-3.



```
On entry to a function
  Safe fp to location $sp - 2
  Move fp to sp
  Pushing the frame
    Decrease $sp by size = Local + Out + 2
      1 for Return address
      1 for frame pointer
  Saving the return register at $fp - 1 (if the function calls any other function)
    Note that $fp now holds the old $sp
```

Figure 5-3 Pseudo-code for functions standard heading translation[9]

We visit the Vapor-M code instruction by instruction. Since we have already allocated registers in phase three, and Vapor-M language is a low level assembly-like language, so the translation during the visit is obvious. However, there still several things we need care about. The first thing need caring is similar to what mentioned before that within .data segment in Vapor-M, which contains the function name and its offset, there is a ':' at the beginning of each line. Thus, when we translate instructions including these functions or syscalls, we need remove these colons as shown below. (The right in MIPS is translated from the left in Vapor-M.)

[$t0] = :vmt LS   ->   la $v0 vmt LS

The second thing need focusing is translating three-operand instruction. Take the addu and addiu instructions as an example. These two instructions both have three operands——dest, arg1 and arg2. We can print dest and arg1 directly. But when translating arg2, we need check if arg2 is an instance of VVarRef class. If the answer is yes, that means arg2 is a register, then we need print addu. If not, that means arg2 is an immediate value, then we need print addiu. The case of translating Lt instruction is similar. It also has three operands. But this time we need check arg1, which means arg1 can be a register or an immediate value. If arg1 is not an instance of VVarRef class, that means it is an immediate value. Since an immediate value cannot be a destination operand, we need exchange the position of arg1 and arg2, then use sgtu to print, as shown below.

$t1 = Lt(0 $t1)   ->   sgtu $t1 $t2 0

The third thing need concerning is that we need store the string after Error into a string list when translating the Error instruction in Vapor-M. These strings are exactly related to align.

The fourth thing as the most difficult thing lies in the translation of memory write instruction (sw instruction in MIPS). When the dest operand is a address, we need consider the property of the dest as well as the souce operands. For a destination operand, if it contains a $sp register, which means it is an address related to registers, then we need regard stack as the base address, and we need compute the offset of the destination operand through the offset of the stack itself and the answer of the question that whether the area the stack is in is Local. Meanwhile, if the source operand is an immediate value, then we need store this immediate value in $v0 register, and then assign the value of $v0 to the destination operand as shown in figure 5-4 below. In this example, the destination operand is [$t1+4], an address about registers. Because the the source operand is an immediate value, the immediate value should be put into $v0 at first, and then put the value in $v0 into the the memory location of $t1+4.

```
                          la $v0 20
[$t1+4] = 20      ->      sw $v0 4($t1)
```

Figure 5-4 An example for the translation of memory write instruction when the source operand is an immediate value.

The last thing should be noticed is that the process of change of stack frame at the time of exit from a function should be translated for the translation of ret instruction as shown below:

```
On exit from a function
  Restoring the return register $ra (if the function calls any other function)
    from $fp - 1
  Restore $fp from $fp - 2
  Popping the frame
    Increase $sp by size = Local + Out + 2
  Jumping to the return register
```

Figure 5-5 Pseudo-code for ret instruction translation[9]

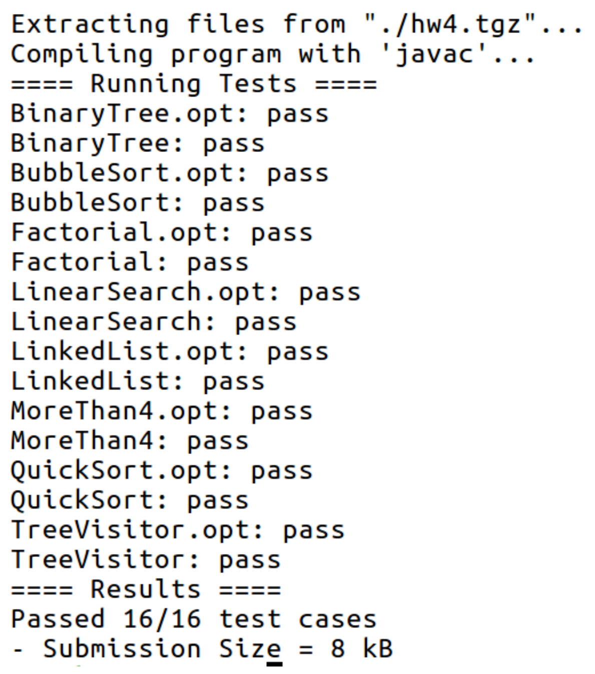
The final step is to translate strings related to align after visited. In the second step, we have already stored the string after Error into a string list. At this time, we just print them out as shown in the figure 5-6.

```
printer(".data");
printer(".align 0");
indentation++;
printer("_newline: .asciiz \"\\n\"");
for(int i = 0; i < strList.size(); i++)
    printer("_str%d: .asciiz \"%s\\n\"", i, strList.get(i));
indentation--;
```

Figure 5-6 Part of code translatING strings related to align

5.3 Test Results

The result for the Phase4Tester[8] is shown as figure 5-7. All passes indicate the MIPS programs translated from preset Vapor-M programs are exactly consistent with the MIPS instructions.



```
Extracting files from "./hw4.tgz"...
Compiling program with 'javac'...
==== Running Tests ====
BinaryTree.opt: pass
BinaryTree: pass
BubbleSort.opt: pass
BubbleSort: pass
Factorial.opt: pass
Factorial: pass
LinearSearch.opt: pass
LinearSearch: pass
LinkedList.opt: pass
LinkedList: pass
MoreThan4.opt: pass
MoreThan4: pass
QuickSort.opt: pass
QuickSort: pass
TreeVisitor.opt: pass
TreeVisitor: pass
==== Results ====
Passed 16/16 test cases
- Submission Size = 8 kB
```

Figure 5-7 Result for the Phase4Tester

6 Summary

A MiniJava-MIPS compiler is a big program, careful attention to modules and interfaces between phases prevents chaos. Thanks to the tools JavaCC parser generator and Java Tree Builder (JTB), lighten the burden on lexical analysis and parsing. The JTB uses visitor pattern make it easier to use two passes to generate vapor code from program in the MiniJava language as well as to do type-checking. Then through analyzing the grammar for three languages and instructions in MIPS, translating the Vapor language to the Vapor-M language and eventually to MIPS machine code becomes follow a rational line.

Four phases are modularized with the output of the previous module to be the input of the next module, thereby facilitating porting and local optimization. However, it can be considered to integrate all phases and design a GUI for freshman in the future, making this work to be regarded as a more mature, complete and user-friendly work. It should also be pointed that although it may not improve the performance of the compiler, an alternative to JBT and JavaCC is to do lexical analyzer generation using flex[15] and parser generation using bison[16].

Acknowledgements

作为本科专业是生物医学工程的学生，我在大四毕业设计选择计算机科学相关的工作是新奇而富有挑战的，如今能够圆满交出答卷，是我何其幸运这一路而来得到许多支持与帮助。首先感谢我的母校 HUST、我的学院 SES 提供了众多资源与平台，让我有机会参加 UCR GPP-E 项目，学习计算机相关的核心专业课程，认识到人生还可以拥有更多可能性。特别感谢我的导师邓勇教授对我的指引教导，对论文的悉心修改，哪怕我身在彼岸也能感受到关怀温暖；感谢 UCR 的 Mohsen Lesani 副教授在实践层面的指导和 Chengyu Song 副教授在理论方面的教学；感谢学院老师对论文进度的关注；感谢助教高激在具体问题上的耐心解答；感谢队友李国稔的共同奋斗和有益讨论；感谢曹婷婷同学对我论文格式调整的帮助，感谢 allen、sili 同学无形助攻；是你们让这篇论文更加成熟高质。另外还要感谢相伴成长的同学好友，朝夕相处的室友，祝福你们前程似锦，学业有成。最后感谢我的父母，你们是我在异国他乡的亲情支柱，感谢你们的默默付出与辛勤培养，文尽，思念之情难止，尤甚。

References

[1] Aho, A.V. Compilers: principles, techniques and tools (for Anna University), 2/e. Pearson Education India, 2003.

[2] Brooks Jr, F.P. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E. Pearson Education India, 1995.

[3] Bornat, R. Understanding and writing compilers: a do-it-yourself guide. Macmillan; 1979.

[4] Sanders, A. MINI-L-Compiler-Project, Github website, https://github.com/asand017/MINI-L-Compiler-Project. 2017-03-26

[5] 姚励, 束永安. "用 JavaCC 构造编译器的方法." 计算机工程 29.9(2003):39-41

[6] UCLA Compilers Group. Java Tree Builder, UCLA Compilers Group website, http://compilers.cs.ucla.edu/jtb/. 2004

[7] Palsberg, J. and Jay, C.B. "The essence of the visitor pattern." In Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241), pp. 9-15. IEEE, 1998.

[8] Xiong, J.Q. Testers for four Phases, Github website, https://github.com/BEAR000777/PhaseXTester, 2019-06-27

[9] Lesani, M. Compiler from Java to MIPS, assistant professor Mohsen Lesani teaching website, https://www.cs.ucr.edu/~lesani/teaching/cp/cp.html, 2019-01-07

[10] Palsberg, J. The MiniJava Type System, professor Jens Palsberg teaching website, http://web.cs.ucla.edu/~palsberg/course/cs132/miniJava-typesystem.pdf, 2014-10-02

[11] Andrew, W.A. and Jens, P. "Modern compiler implementation in Java." (2002).

[12] Palsberg, J. Vapor Language Specification, professor Jens Palsberg teaching website, http://web.cs.ucla.edu/classes/spring11/cs132/kannan/vapor.html, 2011-03-30

[13] Poletto, M. and Sarkar, V. "Linear scan register allocation." ACM Transactions on Programming Languages and Systems (TOPLAS) 21, no. 5 (1999): 895-913.

[14] Britton, R.L. MIPS assembly language programming. Pearson/Prentice Hall, 2004.

[15] Gao, L. FLEX Tutorial, Lan Gao teaching website, http://alumni.cs.ucr.edu/~lgao/teaching/flex.html, 2007-06-29

[16] Gao, L. Bison Tutorial, Lan Gao teaching website, http://alumni.cs.ucr.edu/~lgao/teaching/bison.html, 2007-06-29