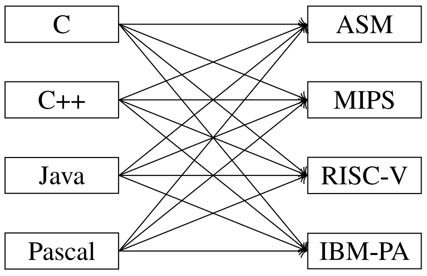
CS179E: Project in Computer Science – Compiler Document for Phase 2

Jiamin Pan, Yunqing Xiao

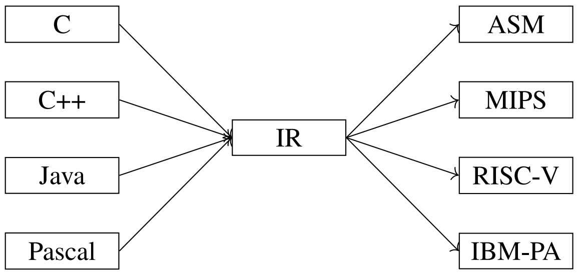
1 Introduction

We have already successfully implemented type-checking in the last phase. Then we will need to convert the MiniJava source file to a source file written in an intermediate language – Vapor. From the homepage [1] we can find some tutorials about Vapor which are sufficient for us to implement the translating process.

The reason why we need to generate intermediate code is that we will utilize it to generate codes written in assembly language. There are various processor architectures (Suppose the number of them is m) and multiple programming languages (Suppose the number of them is n). If we write a compiler for each advanced programming language and processor, just like in Figure 1a, there will be mn compiler with which we are not so satisfied. However, just as depicted in Figure 1b, if we add an intermediate language, we just need to write compilers for converting advanced programming language to it and converting this language to several types of assembly languages. The total number of them is $m+n$, which is very efficient for us.



(a) Without Intermediate Language



(b) With Intermediate Language

Figure 1: Relationship Between Advanced Programming Languages and Processors, abstracted from textbook [2]

2 Dependencies

1. JDK version $\geq$ JDK-8.
2. JavaCC package and MiniJava Parser.
3. Vapor interpreter.

3 Methods for Translating

3.1 Implementation for V-Table

We will use V-Table to determine which method to call since method overriding is also allowed in MiniJava. For the details about V-Table, you may refer to the textbook [2]. Here we just provide some details about our implementation.

Actually we just creat a new class called `CMPair`. Just like its name says, each instance of `CMPair` represents a pair of class and method. Note the method may be overriden, so the class will be changed to the subclass instead of superclass.

We take a sample code segment for example:

```
class A {
    public int run() { ... }
    public int get() { ... }
}

class B extends A {
    public int get() { ... }
}

class C extends B {
    public int run() { ... }
}
```

Then the V-Table for them is like

A

A	run
A	get

B

A	run
B	get

C

C	run
B	get

Figure 2: V-Table for sample codes

where each line represents an instance for `CMPair`.

3.2 Mapping in Scope

We also notice another problem. Since we will use temporary variable name in intermediate codes, it will be difficult for us to refer to them if we want to use some variables for multiple time. Therefore, for each scope, we will maintain a table recording a mapping from each identifier in java codes to an identifier in vapor code.

We will also provide an example

```
1 a = 2;
2 b = 3;
3 System.out.println(a + b);
```

whose corresponding vapor code will be

```
1 t.1 = 2
2 t.2 = 3
3 t.3 = Add(t.1 t.2)
4 PrintInt(t.3)
```

Then the mapping will be $a \rightarrow t.1$ and $b \rightarrow t.2$. When we want to translate the statement in line 3, we will first find if there exists a record for a and b. If we find a matching result, then we will use the allocated identifier in the following translation; otherwise we will create a new identifier for this variable and add this pair to the mapping list. This idea is easy to be implemented and will never occupy so much space.

Besides, it will reduce the number of identifiers in each scope since we do not need to create a new identifiere

3.3 Printer Class

In order to make indent and identifier allocation for vapor, we create a `Printer` class to deal with it. `Printer` class contains several methods including:

1. Record indent number and add indent to the line of the class.

```
// Add indent to the statement
public void printStmt(String stmt) {
    String statement = "";
    for (int i = 0; i < depth; ++i) {
        statement += "\t";
    }
    statement += stmt;
    System.out.println(statement);
}
```

2. Create new identifiers without conflicts. Here we use a counter to generate new identifier.

```
// Return variable identifier
public String newVariable() {
    String var = "t." + Integer.toString(this.counter);
    incCounter();
    return var;
}
```

3. Create label for `if` and `while` statements. The idea is similar to the one above. We also set a counter for it.

3.4 Dynamic Generation

During the process of translation, we take a dynamic method instead of a static way. Basically, the “dynamic” here means that we generate codes during the visiting process instead output codes after we have traversed the whole code. Actually we do not need to traverse the code again since we have completed symbol table construction and type-checking in last phase. We just need to make use of the symbol table and add V-table and mapping list to the new visitor so that we can directly generate code without refer to other part for the java source code. We provide the generation for some type of expressions. The principle are detailed in homepage [1].

```
/**
 * f0 -> "while"
 * f1 -> "("
 * f2 -> Expression()
 * f3 -> ")"
 * f4 -> Statement()
 */
public String visit(WhileStatement n) {
    String _ret=null;
    n.f0.accept(this);
    String label = printer.getLCounter();
    printer.printStmt("loop" + label + "_begin:");
    n.f1.accept(this);
    String condition = getStrType(n.f2.accept(this));
    printer.printStmt("if0␣" + condition + "␣goto␣:loop" + label + "_end");
    n.f3.accept(this);
    printer.incDepth();
    n.f4.accept(this);
    printer.printStmt("goto␣:loop" + label + "_begin");
    printer.decDepth();
    printer.printStmt("loop" + label + "_end:");
    return _ret;
}

/**
 * f0 -> PrimaryExpression()
 * f1 -> "+"
 * f2 -> PrimaryExpression()
 */
public String visit(PlusExpression n) {
    String _ret=null;
    String left = getStrType(n.f0.accept(this));
    n.f1.accept(this);
    String right = getStrType(n.f2.accept(this));
    _ret = printer.newVariable();
    printer.printStmt(_ret + "␣=␣Add(" + left + "␣" + right + ")");
    return _ret;
}
```

3.5 Some Other Difficulties

3.5.1 Fields VS Variables/Parameters

However, even though we have settled down almost all the problems stated before, we also need to face another two dilemmas. Here is the first one.

We may encounter some utilizations of a field of class which just look like a normal local variable or parameter. However, though we can allocate a new identifier for the variable and add the mapping to the list of the scope, we cannot apply this method to fields. We need to get the entry address of the instance and add the offset in the class to it. Surprisingly, this is easy since we just need to refer to the record in `currClass` (For the details of `currClass`, please refer to the document for phase 1.)

```
private String getStrType(String str) {
    if (this.currClass.findField(str) != null) {     // If it is a field
        int idx = this.currClass.getClassRecord().indexOf(str) * 4 + 4;
        String newVar = printer.newVariable();
        printer.printStmt(newVar + "␣=␣[this+" + Integer.toString(idx) + "]");
        return newVar;
    } else if (this.currScope.get(str) != null) {    // If it is an identifier
        return this.currScope.get(str);
    } else {     // It is a value
        return str;
    }
}
```

3.5.2 Variables Not in V-Table

There is another annoying method call in MiniJava:

```
b = new A().run();
```

Yes, we can initially treat it just like what we do for

```
a = new A();
||
t.1 = HeapAllocZ(...)
...
```

just return t.1 representing the object, but we cannot add it to the mapping list since there is no matching local varibles or paremeters to it. What a pity!

If we could find the type of it, then we will be able to determine the offset of the method. Therefore, we need to add a supporting method to determine the class type of it. Since it will be a universal method, then it should also deal with the condition about an instance of a class. We can implement the idea like this:

```
// a method to get type of a class's instance
private String findClassName(PrimaryExpression n) {
    String className = null;
    if (n.f0.choice instanceof AllocationExpression) {  // Class allocation
        AllocationExpression expr = (AllocationExpression)n.f0.choice;
        className = expr.f1.f0.toString();
    } else if (n.f0.choice instanceof ThisExpression) { // This expression
        className = this.currClass.getName();
    } else if (n.f0.choice instanceof BracketExpression) {
        BracketExpression expr1 = (BracketExpression)n.f0.choice;
        if (expr1.f1.f0.choice instanceof MessageSend) {
            MessageSend expr = (MessageSend)expr1.f1.f0.choice;
            return findClassName(expr.f0);
        }
    } else {    // Parameter / Variable name
        String varName = n.accept(this);
        if (this.currMethod.findParam(varName) != null) {
            className = this.currMethod.findParam(varName).getType();
        } else if (this.currMethod.findVar(varName) != null) {
            className = this.currMethod.findVar(varName).getType();
        } else if (this.currClass.findField(varName) != null) {
            className = this.currClass.findField(varName).getType();
        }
    }
    return className;
}
```

4 Review for Phase 2

```
==== Results ====
Passed 13/13 test cases
- Submission Size = 67 kB
```

Just like phase 1, we also passes all the test cases but I also find something to be improved: how to reduce the number of identifiers we use in vapor code. I checked some outputs of our program and I found some identifiers were never used even though it had been created. That is another optimization problem for us and I am convinced that we can solve it in the future.

References

[1] Course's Homepage: Compiler Project.

[2] Andrew W. Appel, Jens Palsberg, *Modern Compiler Implementation in Java*, Cambridge University Press, Second Edition, 2002.