5. JVM Code Generation

5.1 Introduction

Once the AST has been fully analyzed, all variables and expressions have been typed and any necessary tree rewriting has been done. Also a certain amount of setup, needed for code generation has been accomplished. The compiler is now ready to traverse the AST one more time to generate the Java Virtual Machine (JVM) code, that is build the class file, for the program.

For example, consider the following very simple program.

```
public class Square {

    public int square(int x) {
        return x * x;
    }
}
```

Compiling this with our *j--* compiler,

```
$ $j/bin/j-- Square.java[1]
```

produces a class file, **`Square.class`**. If we run the **`javap`** program on this, that is

```
$ javap -verbose Square
```

we get the following symbolic representation of the class file.

```
public class Square extends java.lang.Object
  minor version: 0
  major version: 49
  Constant pool:
const #1 = Asciz Square;
const #2 = class #1;   //  Square
const #3 = Asciz java/lang/Object;
const #4 = class #3;   //  java/lang/Object
const #5 = Asciz <init>;
const #6 = Asciz ()V;
const #7 = NameAndType      #5:#6;//  "<init>":()V
const #8 = Method     #4.#7;     //
java/lang/Object."<init>":()V
```

[1] $j stands for the location of your *j--* code tree. You might consider butting $j/bin in your PATH by setting your PATH environment variable appropriately.

```
const #9 = Asciz Code;
const #10 = Asciz     square;
const #11 = Asciz     (I)I;

{
public Square();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial   #8; //Method java/lang/Object."<init>":()V
   4: return

public int square(int);
  Code:
   Stack=2, Locals=2, Args_size=2
   0: iload_1
   1: iload_1
   2: imul
   3: ireturn

}
```

Of course, we cannot *run* this program because there is no **`main()`** method. But this is not our purpose; we simply wish to look at some code.

Now, the first line is the class header:

```
public class Square extends java.lang.Object
```

It tells us that the name of the class is **`Square`** and that its super class is **`java.lang.Object`**. The next two lines tell us something about the version of the JVM that we are using; we are not interested in this. Following this is a constant pool, which in this case defines eleven constants. The numerical tags **`#1`**,…,**`#11`** are used in the code for referring to these constants. For example, **`#10`** refers to the method name square; **`#11`** refers to its descriptor **`(I)I`**[2]. The use of the constant pool may appear somewhat archaic, but it makes for a compact class file.

Notice there are *two* methods defined. The first is a *constructor*.

```
public Square();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
```

[2] Recall, a *descriptor* describes a method's signature. In this case, **`(I)I`** describes a method that takes one **`int`** as an argument and returns an **`int`** result.

```
1: invokespecial    #8; //Method java/lang/Object."<init>":()V
4: return
```

This constructor (generated as a method named **<init>**) is the implied *empty* constructor (having no arguments), which the compiler must generate if the user has not defined an explicit version. The first line of this code isn't really code at all but a sequence of values that the run-time JVM will want to know when it allocates a stack frame:

```
Stack=1, Locals=1, Args_size=1
```

The **Stack=1** indicates that 1 memory location is required for partial results (in this case, for loading **this** onto the stack). The **Locals=1** says just one local variable is allocated in the stack from, in this instance, the location at offset 0 for **this**. The **Args_size=1** says there is just one actual argument to this constructor (in this case, the implicit argument, **this**).

Then there is the code, proper:

```
0: aload_0
1: invokespecial    #8; //Method java/lang/Object."<init>":()V
4: return
```

At location 0 is a one-byte **aload_0** instruction, which loads **this** onto the stack. Following that, at location 1, is a three-byte **invokespecial** instruction. The first byte holds the opcode for **invokespecial** and the next two bytes hold a reference to the constant #8 in the constant pool, **Method #4.#7;**. This constant makes reference to additional constant entries **#4** and **#7**. Taken as a whole, the **invokespecial** instruction invokes the constructor **<init>,** with descriptor **()V**, of **Square**'s super class **java/lang/Object**, on the single argument **this**. After invoking the super constructor, the constructor simply returns, obeying the **return** instruction at location 4.

The second, explicit method **square()** is even simpler. The stack frame information,

```
Stack=2, Locals=2, Args_size=2
```

says we need two locations for computation (the two operands to the multiplication operation), two local variables allocated (for the implicit **this**, and the explicit parameter **x**), and that there will be two actual arguments on the stack: **this** and the argument for **x**. The code consists of four one-byte instructions:

```
0: iload_1
1: iload_1
2: imul
3: ireturn
```

The instruction at location 0 loads the value for **x** onto the stack; the instruction at location 1 does the same. The instruction at location 2 pops those two values for **x** off from the stack and performs an integer multiplication on them, leaving its result on the stack. The instruction at location 3 returns the integer (**x*x**) on the stack as the result of the method invocation.

Of course, to emit these instructions, we firstly create a **CLEmitter** instance, which is an abstraction of the class file we wish to build, and then call upon **CLEmitter**'s methods for generating the necessary headers and instructions.

For example, to generate the class header,

```
public class Square extends java.lang.Object
```

one would invoke the **addClass()** method on **output**, an instance of **CLEmitter**:

```
output.addClass(mods, "Square", "java/lang/Object", false);
```

where,

- The **mods** denotes an **ArrayList** containing the single string **"public"**,
- **Square** is the class name,
- **java/lang/Object** is the *internal form* for the fully qualified super class, and
- **false** indicates that the class is *not* synthetic.

As a simpler example, the one-byte, no-argument instruction **aload_1** may be generated by

```
output.addNoArgInstruction(ALOAD_1);
```

To fully understand **CLEmitter** and all of its methods for generating code, one should read through the **CLEmitter.java** source file, which is distributed as part of the *j--* compiler. One must sometimes be careful to pay attention to what the **CLEmitter** is expecting. For example, sometimes a method requires a fully qualified name in Java form such as **java.lang.Object**, but other times an internal form is required: **java/lang/Object**.

For another more involved example of code generation, we revisit the **Factorial** class from chapters 2 and 3. Recall the source code for **Factorial**,

```
package pass;

import java.lang.System;
```

```
public class Factorial
{
    // Two methods and a field

    public static int factorial(int n)
    {
         // position 1:
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args)
    {
        int x = n;
         // position 2:
        System.out.println(n + "! = " + factorial(x));
    }

    static int n = 5;
}
```

Running **javap** on the class produced for this by the *j*-- compiler gives us,

```
public class pass.Factorial extends java.lang.Object
  minor version: 0
  major version: 49
  Constant pool:
…

{
static int n;

public pass.Factorial();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial   #8; //Method java/lang/Object."<init>":()V
   4: return

public static int factorial(int);
  Code:
   Stack=3, Locals=1, Args_size=1
   0:  iload_0
   1:  iconst_0
   2:  if_icmpgt 10
   5:  iconst_1
```

```
   6:  ireturn
   7:  goto 19
   10: iload_0
   11: iload_0
   12: iconst_1
   13: isub
   14: invokestatic   #13; //Method factorial:(I)I
   17: imul
   18: ireturn
   19: nop

public static void main(java.lang.String[]);
  Code:
   Stack=3, Locals=2, Args_size=1
   0:  getstatic #19; //Field n:I
   3:  istore_1
   4:  getstatic #25; //Field
                      java/lang/System.out:Ljava/io/PrintStream;
   7:  new  #27; //class java/lang/StringBuilder
   10: dup
   11: invokespecial  #28; //Method
                           //java/lang/StringBuilder."<init>":()V
   14: getstatic#19; //Field n:I
   17: invokevirtual  #32; //Method
           java/lang/StringBuilder.append:
           (I)Ljava/lang/StringBuilder;
   20: ldc #34; //String ! =
   22: invokevirtual  #37; //Method
           java/lang/StringBuilder.append:
           (Ljava/lang/String;)Ljava/lang/StringBuilder;
   25: iload_1
   26: invokestatic   #13; //Method factorial:(I)I
   29: invokevirtual  #32; //Method
           // java/lang/StringBuilder.append:
           // (I)Ljava/lang/StringBuilder;
   32: invokevirtual  #41; //Method
           // java/lang/StringBuilder.toString:
           // ()Ljava/lang/String;
   35: invokevirtual  #47; //Method
           // java/io/PrintStream.println:
           // (Ljava/lang/String;)V
   38: return

public static {};
  Code:
   Stack=2, Locals=0, Args_size=0
   0:  iconst_5
   1:  putstatic#19; //Field n:I
   4:  return
```

```
}
```

We have removed the constant pool, but the comments contain the constants that the program refers to. Notice the last method, **`static`**. That basically implements what is known as the static block – where class initializations can go. Here, the static field **`n`** is initialized to 5. The method that does this in the JVM code is **`<clinit>`** (for class initialization).

The following sections address the task of generating code for various *j--* constructs.

5.2 Generating Code for Classes and their Members

`JCompilationUnit`'s **`codegen()`** drives the generation of code for classes. For each type (that is, class) declaration, it

- invokes **`codegen()`** on the **`JClassDeclaration`** for generating the code for that class, and
- writes out the class to a class file in the destination directory.
- adds the in-memory representation of the class to a list that stores such representations for all the classes within a compilation unit; this list is used in translating JVM bytecode to native (SPIM) code.

```
public void codegen(CLEmitter output) {
    for (JAST typeDeclaration : typeDeclarations) {
         typeDeclaration.codegen(output);
         output.write();
         clFiles.add(output.clFile());
    }
}
```

5.2.1 Class Declarations

The **`codegen()`** method for **`JClassDeclaration`** does the following.

- It computes the fully qualified name for the class, taking any package name into account.
- It invokes an **`addClass()`** on the **`CLEmitter`** for adding the class header to the start of the class file.
- If there is no explicit constructor with no arguments defined for the class, it invokes the private method **`codegenImplicitConstructor()`** to generate code for the implicit constructor as required by the language.
- It generates code for its members, by sending the *codegen()* message to each of them.
- If there are any static field initializations in the class declaration, then it invokes the private method **`codegenClassInit()`** to generate the code necessary for defining a *static block*, a block of code that is executed after a class is loaded.

In the case of our **Factorial** example, there is no explicit constructor, so one is generated. The method, **codegenImplicitConstructor()** is invoked to generate the following JVM code[3]:

```
public <init>();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial   #8; //Method java/lang/Object."<init>":()V
   4: return
```

The constructor is simply invoking the superclass constructor, i.e. **Object()**. In Java, such a constructor would look like,

```
public Factorial() {
    this.super();
}
```

The **Factorial** example has one static field with an initialization,

```
static int n = 5;
```

During analysis, the initialization was transformed into an explicit assignment, which must be executed after the **Factorial** class is loaded. Seasoned Java programmers will recognize this to be in a *static block*; in Java, the static block would look like,

```
static {
    n = 5;
}
```

and would occur as a member of **Factorial**. Of course, *j--* does not have static blocks[4], they may be represented in the JVM as, e.g.[5]

```
public <clinit> {};
  Code:
```

---

[3] Actually, **javap** shows the name as **Factorial()** but when invoking the **addMethod()** method on the **CLEmitter**, one passes the argument "**<init>**" for the constructor's name; a constructor is simply an initialization method. The JVM expects this internal name.

[4] Its implementation is left as an exercise, in Chapter 6.

[5] Again, **javap** represents **<clinit>** as **static**, but in the argument to the **addMethod()** method is "**<clinit>**"; **<clinit>** stands for *class initialization*. Again, the JVM expects this internal name.

```
   Stack=2, Locals=0, Args_size=0
   0:  iconst_5
   1:  putstatic #19; //Field n:I
   4:  return
}
```

5.2.2 Method Declarations

The code generated for a **JMethodDeclaration** is pretty simple:

```
public void codegen(CLEmitter output) {
    output.addMethod(mods, name, descriptor, null, false);
    if (body != null) {
        body.codegen(output);
    }

    // Add implicit RETURN
    if (returnType == Type.VOID) {
        output.addNoArgInstruction(RETURN);
    }
}
```

It generates code for the method header, the body and then, for void methods, an implicit return statement. Of course, the return may be superfluous if one already exits in the source code, but any good optimizer would remove the extra one.

5.2.3 Constructor Declarations

A constructor declaration is handled very much like a method declaration, but requires two additional tasks:

1. After the method header, with name **<init>**, has been emitted, **JConstructorDeclaration**'s **codegen()** looks to see if a super class constructor has been explicitly invoked:

   ```
   if (!invokesConstructor) {
       output.addNoArgInstruction(ALOAD_0);
       output.addMemberAccessInstruction(INVOKESPECIAL,
           ((JTypeDecl) context.classContext().definition())
              .superType().jvmName(), "<init>", "()V");
   }
   ```

   The flag, **invokesConstructor**, is set **false** by default and is set to **true** during analysis of the body if the first statement in that body is an invocation of the super class constructor.
2. Any instance field initializations (after analysis, represented as assignments) are generated:

```
    for (JFieldDeclaration field : definingClass
            .instanceFieldInitializations()) {
        field.codegenInitializations(output);
    }
```

Notice that **JConstructorDeclaration** extends **JMethodDeclaration**.

5.2.4 Field Declarations

Given that any initializations have been moved elsewhere, in the analysis phase, **codegen()** for **JFieldDeclaration** need only generate code for the field declaration itself:

```
public void codegen(CLEmitter output) {
    for (JVariableDeclarator decl : decls) {
        // Add field to class
        output.addField(mods, decl.name(), decl.type()
            .toDescriptor(), false);
    }
}
```

5.3 Generating Code for Control and Logical Expressions

5.3.1 Branching on Condition

Just about all control statements in *j--* are controlled by some Boolean expression. Indeed, control and Boolean expressions are intertwined in *j--*. For example, consider the if-then-else-statement and the while-statement below.

```
if (a > b)
    c = a;
else
    c = b;
```

```
while (a > b)
    b++;
```

The code produced for the two are as follows.

```
0:    iload_1
1:    iload_2
2:    if_icmple  10
5:    iload_1
6:    istore_3
7:    goto  12
10:   iload_2
11:   istore_3
12:   …
```

```
0:    goto  6
3:    iinc  1, 1
6:    iload_1
7:    iload_2
8:    if_icmpgt  3
11:   …
```

Notice a couple of things about this code.

1. In both examples, rather than to compute a **boolean** value (**true** or **false**) onto the stack depending on the condition and then branching on the value of *that*, the code branches directly on the condition itself. This is faster, makes use of the underlying JVM instruction set, and makes for more compact code.

2. In the while-statement a single unconditional branch is made to test condition, which appears at the *bottom* of the loop. Then a branch on that condition is made back to the top of the loop each time that condition is true. Compare this to the alternative:

```
0:      iload_1
1:      iload_2
2:      if_icmple  11
5:      iinc  1, 1
8:      goto  0
11:     …
```

   Here we execute both an **if_cmple** and a **goto**, *each time around the loop*.

3. In the if-then-else-statement, a branch is made over the code for the then-part to the else-part if the condition's *complement* is **true** -- that is, if the condition is **false**; in our example, an **if_icmple** instruction is used. But in the while-statement, a branch is made back to the top of the loop (the body) if the *condition itself* is **true**.

```
   branch to elseLabel
    if <condition> is false
   <code for thenPart>
   branch to endLabel
elseLabel:
  <code for elsePart>
endLabel:
```

```
    branch to test
top:
    <code for body>
test:
    branch to top
     if <condition> is true
```

In generating code for a condition, one needs a method specifying three arguments;
1. The **CLEmitter**.
2. The target label for the branch.
3. A **boolean** flag, **onTrue**. If **onTrue** is **true** then the branch should be made on the condition; if **false**, the branch should be made on the condition's complement.

Thus, every **boolean** expression must support a version of **codegen()** with these three arguments. For example consider that for **JGreaterThanOp** is:

```
public void codegen(CLEmitter output, String targetLabel,
    boolean onTrue) {
    lhs.codegen(output);
    rhs.codegen(output);
    output.addBranchInstruction(onTrue ? IF_ICMPGT : IF_ICMPLE,
            targetLabel);
}
```

A method of this sort is invoked on the condition controlling execution; for example, the following **`codegen()`** for **`JIfStatement`** makes use of such a method in producing code for the if-then-else statement.

```
public void codegen(CLEmitter output) {
    String elseLabel = output.createLabel();
    String endLabel = output.createLabel();
    condition.codegen(output, elseLabel, false);
    thenPart.codegen(output);
    if (elsePart != null) {
        output.addBranchInstruction(GOTO, endLabel);
    }
    output.addLabel(elseLabel);
    if (elsePart != null) {
        elsePart.codegen(output);
        output.addLabel(endLabel);
    }
}
```

Notice that method **`createLabel()`** creates a unique label each time it is invoked, and the **`addLabel()`** and **`addBranchInstruction()`** methods compute the necessary offsets for the actual JVM branches.

5.3.2 Short-circuited &&

Consider the && operator. In a logical expression like

arg_1 **`&&`** arg_2

The semantics of Java, and so of *j--*, require that the evaluation of such an expression be *short-circuited*. That is, if arg_1 is **`false`**, arg_2 is not evaluated; **`false`** is returned. If arg_1 is **`true`**, the value of the entire expression depends on arg_2. This can be expressed using the Java conditional expression, as

arg_1 **`?`** arg_2 **`: false`**

How do we generate code for this operator? The code to be generated depends of whether the branch for the entire expression is to be made on **`true`**, or on **`false`**:

Branch to target when arg_1 **`&&`** arg_2 is **`true`**:

```
    branch to skip if
        arg1 is false
    branch to target when
        arg2 is true
skip: …
```

Branch to target when arg_1 **`&&`** arg_2 is **`false`**:

```
    branch to target if
        arg1 is false
    branch to target if
        arg2 is false
```

For example, the code generated for

```
if (a > b && b > c)
    c = a;
else
    c = b;
```

would be

```
   0:      iload_1
   1:      iload_2
   2:      if_icmple  15
   5:      iload_2
   6:      iload_3
   7:      if_icmple  15
   10:     iload_1
   11:     istore_3
   12:     goto  17
   15:     iload_2
   16:     istore_3
   17:     …
```

The **`codegen()`** method in **`JLogicalAndOp`** generates the code for the **`&&`** operator:

```
public void codegen(CLEmitter output, String targetLabel,
    boolean onTrue) {
    if (onTrue) {
        String falseLabel = output.createLabel();
        lhs.codegen(output, falseLabel, false);
        rhs.codegen(output, targetLabel, true);
        output.addLabel(falseLabel);
    } else {
        lhs.codegen(output, targetLabel, false);
        rhs.codegen(output, targetLabel, false);
    }
```

```
}
```

The implementation of a `||` operator for a short-circuited logical or operation (not yet in the *j--* language) is left as an exercise.

5.3.3 Logical Not: !

The code generation for the **`JLogicalNot`** operator **`!`** is trivial: a branch on **`true`** becomes a branch on **`false`**, and a branch on **`false`** becomes a branch on **`true`**:

```
public void codegen(CLEmitter output, String targetLabel,
    boolean onTrue) {
    arg.codegen(output, targetLabel, !onTrue);
}
```

5.4 Generating Code for Message Expressions, Field Selection and Array Expressions

5.4.1 Message Expressions

Message expressions are what distinguish object-oriented programming languages from the more traditional programming languages such as C. *j--,* like Java, is object-oriented.

Much of the work in decoding a message expression is done in analysis. By the time the compiler has entered the code generation phase, any ambiguous targets have been reclassified, instance messages have been distinguished from class (static) messages, implicit targets have been made explicit, and the message's signature has been determined.

In fact, by examining the target's type (and if necessary, its sub-types) analysis can determine whether or not a viable message matching the signature of the message exists. Of course, for instance messages, the actual message that is invoked cannot be determined until run time. That depends on the actual run-time object that is the target of the expression and so is determined by the JVM. But the code generated for a message expression does *state* everything that the compiler *can* determine.

`JMessageExpression`'s **`codegen()`** proceeds as follows:

1. If the message expression involves an instance message, **`codegen()`** generates code for the target.
2. The message invocation instruction is determined: **`invokevirtual`** for instance messages and **`invokestatic`** for static messages.

3. The **`addMemberAccessInstruction()`** method is invoked to generate the message invocation instruction; this method takes the following arguments.
    a. The instruction (**`invokevirtual`** or **`invokestatic`**).
    b. The JVM name for the target's type.
    c. The message name.
    d. The descriptor of the invoked method, which was determined in analysis.
4. If the message expression is being used as a statement expression *and* the return type of the method referred to in (d), and so the type of the message expression itself is non-void, then method **`addNoArgInstruction()`** is invoked for generating a **`pop`** instruction. This is necessary because executing the message expression will produce a result on top of the stack, and this result is to be thrown away.

For example, the code generated for the message expression,

```
… = s.square(6);
```

where **`square`** is an instance method, would be

```
aload s'
bipush     6
invokevirtual    #6; //Method square:(I)I
```

Notice this leaves an integer result 36 on the stack. But if the message expression were used as a statement, as in

```
s.square(6);
```

our compiler generates a **`pop`** instruction to dispose of the result:

```
aload s'[6]
bipush     6
invokevirtual    #6; //Method square:(I)I
pop
```

We invoke static methods using **`invokestatic`** instructions. For example, say we wanted to invoke the static method **`csquare`**, the message expression

```
.. = Square1.csquare(5);
```

would be translated as

```
iconst_5
invokestatic     #5; //Method csquare:(I)I
```

[6] **`s'`** (and other uses of ') indicates the stack frame offset for the local variable **`s`**.

5.4.2 Field Selection

When we talk about generating code for a field selection, we are interested in the case where we want the *value*[7] for the field selection expression. The case where the field selection occurs as the target of an assignment statement is considered in the next section.

As for message expressions, much of the work in decoding a field selection occurs during analysis. By the time the code generation phase begins, the compiler has determined all of what it needs to know to generate code. **`JFieldSelection`**'s **`codegen()`** works as follows.

1. It generates code for its target. (If the target is a class, no code is generated.)
2. The compiler must again treat the special case, **`a.length`** where **`a`** is an array. As was noted in the discussion of analysis, this is a language hack. The code generated is also a hack, making use of the special instruction, **`arraylength`**.
3. Otherwise, it is treated as a proper field selection. The field selection instruction is determined: **`getfield`** for instance fields and **`getstatic`** for static fields.
4. The **`addMemberAccessInstruction()`** method is invoked with the following arguments.
    a. The instruction (**`getfield`** or **`getstatic`**).
    b. The JVM name for the target's type.
    c. The field name.
    d. The JVM descriptor for the type of the field, and so the type of the result.

For example, given that **`instanceField`** is indeed an instance field, the field selection

```
s.instanceField
```

would be translated as

```
aload s'
getfield instanceField:I
```

On the other hand, for a static field selection such as

```
Square1.staticField
```

our compiler would generate

```
getstatic staticField:I
```

---

[7] Actually, as we shall see in the next section, the field expression's *r-value*.

5.4.3 Array Access Expressions

Array access expressions in Java and so in *j--* are pretty straightforward. For example, if the variable **a** references an array object, and **i** is an integer, then

```
a[i]
```

is an array access expression. The variable **a** may be replaced by any primary expression that evaluates to an array object, and **i** may be replaced by any expression evaluating to an integer. The JVM code is just as straightforward. For example, if **a** is an integer array, then the code produced for the array access expression would be

```
aload a'
iload i'
iaload
```

Instruction **iaload** pops an index **i** and an (integer) array **a** off the stack and replaces those two items by **a[i]**.

In *j--*, as in Java, multiply dimensioned arrays are implemented as arrays of arrays.[8] That is, *j--* arrays are vectors.

5.5 Generating Code for Assignment and Similar Operations

5.5.1 Issues in Compiling Assignment

5.5.1.1 l-values and r-values

Consider a simple assignment statement.

```
x = y;
```

It does not seem to say very much; it asks that the value of variable **y** be stored (as the new value) in variable **x**. Notice that we want different values for **x** and **y** when interpreting such an assignment. We often say we want the *l-value* for **x**, and the *r-value* for **y**. We may think of a variable's l-value as its address or location (or in the case of local variables, its stack frame offset), and we may think of a variable's r-value as its

[8] Such arrays are known as Illife vectors, after their inventor, John K. Illife.

content or the value stored in (or, as the value of) the variable. This relationship is illustrated in Figure 5.1.

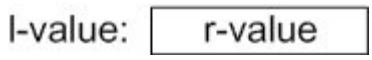

Figure 5.1 A Variable's l-value and r-value

The names, l-value and r-value come from the corresponding positions in the assignment expression: the left-hand side of the **=**, and the right-hand side of the **=**. Think of,

l-value **=** r-value

Notice that while all expressions have r-values, many have no l-values. For example, if **a** is an array of ten integers, and **o** is an object with field **f**, **C** is a class with static field **sf**, and **x** is a local variable, all of

```
a[3]
o.f
C.sf
x
```

have both l-values and r-values. That is, any of these may appear on the left-hand side or on the right-hand side of an assignment operator. On the other hand, while all of the following have r-values, *none* has an l-value:

```
5
x+5
Factorial.factorial(5)
```

None of these may meaningfully appear on the left-hand side of an assignment statement.

In compiling an assignment, compiling the right-hand-side expression to produce code for computing its r-value and leaving it on the stack is straightforward. But what code must be generated for the left-hand side? In some cases, no code need be generated; for example, assuming **x** and **y** are local integer variables, compiling

```
x = y;
```

produces

```
iload y'
istore x'
```

where **x′** and **y′** are the stack offsets for **x** and **y** respectively. On the other hand, compiling

```
a[x] = y;
```

produces

```
aload a′
iload x′
iload y′
iastore
```

This code loads (a reference to) the array **a**, and an index **x** onto the stack. It then loads the r-value for **y** onto the stack. The **iastore** (integer array store) instruction pops the value for **y**, the index **x**, and the array **a** from the stack, and it stores **y** in the element indexed by **x** in array **a**.

5.5.1.2 Assignment Expressions vs. Assignment Statements

Another issue is that an assignment may act as an expression producing a result (on the stack) or as a statement (syntactically, as a statement expression). That is, we can have

```
x = y;
```

or

```
z = x = y;
```

In the first case, the assignment **x = y;** is a statement; no value is left on the stack. But in the second case, the **x = y** must assign the value of **y** to **x** but also leave a value (the r-value for **y**) on the stack so that it may be popped off and assigned to **z**. That is, the code might look something like

```
iload y′
dup
istore x′
istore z′
```

The r-value for **y** is duplicated on top of the stack so there are two copies. Each of the **istore** operations pops a value and stores it in the appropriate variable.

During analysis, when an assignment is used as a statement, **JStatementExpression**'s **analyze()** method sets a flag **isStatementExpression** in the expression node to

true. Now, the code generation phase must make use of this flag in deciding when code must be produced for duplicating r-values on the run-time stack.

5.5.1.3 Additional Assignment-Like Operations

The most important property of the assignment is its *side effect*. One uses the assignment operation for its side effect: overwriting a variable's r-value with another. Indeed, the assignment statement is the most important statement in any programming language.[9] There are other Java (and *j*--) operations having the same sort of side effect. For example,

```
x--
++x
x += 6
```

are all expressions, which have side effects. But they also denote values. The context in which they are used determines whether or not we want the value left on the stack, or we simply want the side effect.

5.5.2 Comparing Left-Hand Sides and Operations

The table in Figure 5.2 compares the various operations (labeled down the left), with an assortment of left-hand sides (labeled across the top). We don't deal with string concatenation here, but leave that to a later section.

	lhs			
Operation	**x**	**a[i]**	**o.f**	**C.sf**
lhs = y	**iload y′** **[dup]** **istore x′**	**aload a′** **iload i′** **iload y′** **[dup_x2]** **iastore**	**aload o′** **iload y** **[dup_x1]** **putfield f**	**iload y′** **[dup]** **putstatic sf**
lhs += y	**iload x′** **iload y′** **iadd** **[dup]** **istore x′**	**aload a′** **iload i′** **dup2** **iaload** **iload y′** **iadd** **[dup_x1]**	**aload o′** **dup** **getfield f** **iload y′** **iadd** **[dup_x1]**	**getstatic sf** **iload y′** **iadd** **[dup]** **putstatic sf**

[9] The assignment statement has been called the *ultimate imperative*.

```
                    [dup_x2]    putfield f
                    iastore

++lhs      iinc x',1   aload a'    aload o'       getstatic sf
           [iload x']  iload i'    dup            iconst_1
                       dup2        getfield f     iadd
                       iaload      iconst_1       [dup]
                       iconst_1    iadd           putstatic sf
                       iadd       [dup_x1]
                       [dup_x2]    putfield f
                       iastore

lhs--      [iload x']  aload a'    aload o'       getstatic sf
           iinc x',1   iload i'    dup            [dup]
                       dup2        getfield f     iconst_1
                       iaload      [dup_x1]       isub
                       [dup_x2]    iconst_1       putstatic sf
                       iconst_1    isub
                       isub        putfield f
                       iastore
```

Figure 5.2 Comparing Code for Assignment-like Operations

The instructions in brackets [...] must be generated if and only if the operation is a sub-expression of some other expression, that is if the operation is not a statement expression.

Notice that there are all sorts of instructions for duplicating values and moving them into various locations on the stack.

dup	simply duplicates the top value on the stack (pushing a copy on top)
dup2	duplicates the top pair of items on the stack (pushing a second pair on top)
dup_x2	duplicates the top value, but copies it below the two items that are below the original top value.
dup_x1	duplicates the top value. but copies it below the single item that is below the original top value.

Figure 5.3 illustrates the effect of each of these. As one might guess, these instructions were included in the JVM instruction set for just these sorts of operations.

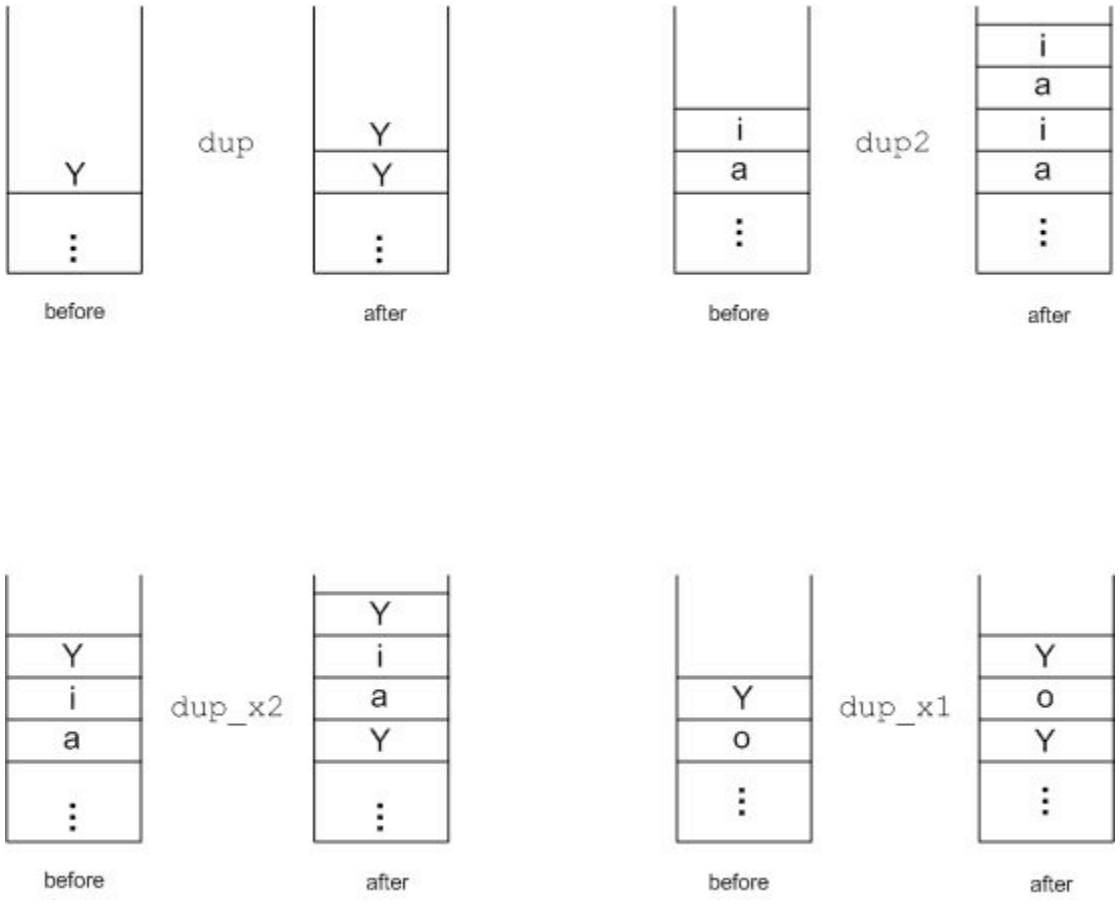

Figure 5.3 The Effects of Various Duplication Instructions

The table in Figure 5.2 suggests that all of this is very complicated, but *there i*s a pattern here, which we may take advantage of in clarifying what code to generate.

5.5.3 Factoring Assignment-like Operations

The table in Figure 5.2 suggests four sub-operations common to most of the assignment-like operations in *j--*. They may also be useful if and when we want to extend *j--* and adding additional operations. These four sub-operations are:

1. **`codegenLoadLhsLvalue`** -- this generates code to load any up-front data for the left-hand side of an assignment needed for an eventual store, i.e. its l-value.
2. **`codegenLoadLhsRvalue`** -- this generates code to load the r-value of the left-hand side, needed for implementing, for example the **+=** operator.
3. **`codegenDuplicateRvalue`** -- this generates code to duplicate an r-value on the stack and put it in a place where it will be on top of the stack once the store is executed.
4. **`codegenStore`** -- this generates the code necessary to perform the actual store.

The code needed for each of these differs for each potential left-hand side of an assignment: a simple local variable **x**, an indexed array element **a[i]**, an instance field **o.f**, and a static field **C.sf**. The code necessary for each of the four operations, and for each left-hand-side form, is illustrated in Figure 5.4.

	lhs			
	x	**a[i]**	**o.f**	**C.sf**
codegenLoadLhsLvalue:				
	<none>	**aload a′** **aload i′**	**aload o′**	**<none>**
codegenLoadLhsRvalue:				
	iload x′	**dup2** **iaload**	**dup** **getfield f**	**getstatic sf**
codegenDuplicateRvalue:				
	dup	**dup_x2**	**dup_x1**	**dup**
codegenStore:				
	istore x′	**iastore**	**putfield f**	**putstatic sf**

Figure 5.4 The Code for Four Sub-Operations

Our compiler defines an interface **JLhs**, which defines four abstract methods for these four sub-operations. Each of **JVariable**, **JArrayExpression** and **JFieldSelection** implement **JLhs**. Of course, one must also be able to generate code for the right-hand side expression. But **codegen()** is sufficient for that -- indeed, that is its purpose.

For example, **JPlusAssignOp**'s **codegen()** makes use of all of these operations:

```
public void codegen(CLEmitter output) {
    ((JLhs) lhs).codegenLoadLhsLvalue(output);
    if (lhs.type().equals(Type.STRING)) {
        rhs.codegen(output);
    } else {
        ((JLhs) lhs).codegenLoadLhsRvalue(output);
        rhs.codegen(output);
        output.addNoArgInstruction(IADD);
    }
```

```
    if (!isStatementExpression) {
        // Generate code to leave the r-value atop stack
        ((JLhs) lhs).codegenDuplicateRvalue(output);
    }
    ((JLhs) lhs).codegenStore(output);
}
```

5.6 Generating Code for String Concatenation

The implementation of most unary and binary operators is straightforward; there is a JVM instruction for each *j--* operation. A case for which this does not apply is string concatenation.

In *j--,* as in Java, the binary **+** operator is overloaded. If both of its operands are integers, it denotes addition. But if either operand is a string then the operator denotes string concatenation and the result is a string. String concatenation is the only *j--* operation where the operand types don't have to match[10].

The compiler's analysis phase determines whether or not string concatenation is implied. When it is, the concatenation is made explicit; that is, the operation's AST is rewritten, replacing **`JAddOp`** with a **`JStringConcatenationOp`**. Also, when **`x`** is a string, analysis replaces

`x +=` <expression>

by

`x = x +` <expression>

So, code generation is left with generating code for only the explicit string concatenation operation.

To implement string concatenation, the compiler generates code to do the following.
1. Create an empty string buffer, i.e. a **`StringBuffer`** object, and initialize it.
2. Append any operands to that buffer. That **`StringBuffer`**'s **`append()`** method is overloaded to deal with any type makes handling operands of mixed types easy.
3. Invoke the **`toString()`** method on the string buffer to produce a **`String`**.

`JStringConcatenationOp`'s **`codegen()`** makes use of a helper method, **`nestedCodegen()`** for performing only step 2 for any nested string concatenation

[10] We leave the implementation of implicit type conversions as an exercise in Chapter 6.

operations. This eliminates the instantiation of unnecessary string buffers. For example, given the *j--* expression

```
x + true + "cat" + 0
```

the compiler generates the following JVM code.

```
new   java/lang/StringBuilder
dup
invokespecial StringBuilder."<init>":()V
aload x'
invokevirtual append:(Ljava/lang/String;)StringBuilder;
iconst_1
invokevirtual append:(Z)Ljava/lang/StringBuilder;
ldc   "cat"
invokevirtual
        append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
iconst_0
invokevirtual append:(I)Ljava/lang/StringBuilder;
invokevirtual StringBuilder.toString:()Ljava/lang/String;
```

5.7 Generating Code for Casts

Analysis determines both the validity of a cast and the necessary **`Converter`**, which encapsulates the code generation for the particular cast. Each **`Converter`** implements a method **`codgen()`**, which generates any code necessary to the cast. For example, consider the converter for casting a reference type to one of its sub-types: such a cast is called *narrowing* and requires that a **`checkcast`** instruction be generated.

```
class NarrowReference implements Converter {

    private Type target;

    public NarrowReference(Type target) {
        this.target = target;
    }

    public void codegen(CLEmitter output) {
        output.addReferenceInstruction(CHECKCAST,
            target.jvmName());
    }
}
```

On the other hand, when any type is cast to itself (the identity cast), or when a reference type is cast to one of its super types (called *widening*), no code need be generated.

Casting an **`int`** to an **`Integer`** is called *boxing*, and requires an invocation of the **`Integer.valueOf()`** method:

```
invokestatic java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
```

Casting an **`Integer`** to an **`int`** is called *unboxing*, and requires an invocation of the **`Integer.intValue()`** method:

```
invokevirtual java/lang/Integer.intValue:()I
```

Certain casts, from one primitive type to another require that a special instruction be executed. For example, the **`i2c`** instruction converts an **`int`** to a **`char`**.

There is a **`Converter`** defined for each valid conversion in *j--*.

Further Reading

(Lindholm and Yellin, 1999) is the authoritative reference for the JVM and its instruction set. Chapter 7 discusses some of the issues in compiling a high-level language such as Java to the JVM.

An excellent introduction to compiling the Microsoft's .Net Common Language Runtime (CLR) is (Gough, 2001).

(Strachey, 1967) introduces the notion of l-values and r-values.

Exercises

Exercises in code generation are discussed in Chapter 6.