Introduction to Compiler Construction

JVM Code Generation: Control, Message, Field Selection, and Array Access Expressions

Outline

1. Generating Code for Control and Logical Expressions

2. Generating Code for Message, Field Selection, and Array Expressions

Generating Code for Control and Logical Expressions

Generating Code for Control and Logical Expressions

Almost all control statements in *j--* are controlled by some Boolean expression

Generating Code for Control and Logical Expressions

Almost all control statements in *j--* are controlled by some Boolean expression

For example, consider the if-then-else statement below

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

The JVM code produced for the statement is 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: ...
```

Generating Code for Control and Logical Expressions

Almost all control statements in *j*-- are controlled by some Boolean expression

For example, consider the if-then-else statement below

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

The JVM code produced for the statement is 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: ...
```

Notice a couple of things

Generating Code for Control and Logical Expressions

Almost all control statements in *j--* are controlled by some Boolean expression

For example, consider the if-then-else statement below

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

The JVM code produced for the statement is 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: ...
```

Notice a couple of things

1. We don't compute a Boolean value onto the stack and then branch on its value, but make use of the underlying JVM instruction set, which makes for more compact code

Generating Code for Control and Logical Expressions

Almost all control statements in *j--* are controlled by some Boolean expression

For example, consider the if-then-else statement below

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

The JVM code produced for the statement is 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: ...
```

Notice a couple of things

1. We don't compute a Boolean value onto the stack and then branch on its value, but make use of the underlying JVM instruction set, which makes for more compact code
2. We branch to the else-part if the condition is `false`

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

Generating Code for Control and Logical Expressions

Generating Code for Control and Logical Expressions

Suppose we wish implement the Java do-while statement in *j--*; for example

```
do {
    a++;
}
while (a < b);
```

The code we generate might have the form

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

Note that we branch when the condition is `true`

Generating Code for Control and Logical Expressions

Suppose we wish implement the Java do-while statement in *j--*; for example

```
do {
    a++;
}
while (a < b);
```

The code we generate might have the form

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

Note that we branch when the condition is `true`

In generating code for a condition, one needs a method specifying three arguments

Generating Code for Control and Logical Expressions

Suppose we wish implement the Java do-while statement in *j--*; for example

```
do {
    a++;
}
while (a < b);
```

The code we generate might have the form

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

Note that we branch when the condition is `true`

In generating code for a condition, one needs a method specifying three arguments

1. The `CLEmitter` instance

Generating Code for Control and Logical Expressions

Suppose we wish implement the Java do-while statement in *j--*; for example

```
do {
    a++;
}
while (a < b);
```

The code we generate might have the form

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

Note that we branch when the condition is `true`

In generating code for a condition, one needs a method specifying three arguments

1. The `CLEmitter` instance
2. The target label for the branch

Generating Code for Control and Logical Expressions

Suppose we wish implement the Java do-while statement in *j--*; for example

```
do {
    a++;
}
while (a < b);
```

The code we generate might have the form

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

Note that we branch when the condition is `true`

In generating code for a condition, one needs a method specifying three arguments

1. The `CLEmitter` instance
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, and if `false`, the branch should be made on the condition's complement

Generating Code for Control and Logical Expressions

Suppose we wish implement the Java do-while statement in *j--*; for example

```
do {
    a++;
}
while (a < b);
```

The code we generate might have the form

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

Note that we branch when the condition is `true`

In generating code for a condition, one needs a method specifying three arguments

1. The `CLEmitter` instance
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, and 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, here is that overloaded `codegen()` method for `JGreaterThanOp`

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

Generating Code for Control and Logical Expressions

Generating Code for Control and Logical Expressions

The three-argument `codegen()` method is invoked on the condition controlling execution

Generating Code for Control and Logical Expressions

The three-argument `codegen()` method is invoked on the condition controlling execution

For example, the `codegen()` method in `JIfStatement` makes use of the three-argument `codegen()` method in producing code for the if-then-else statement

```java
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);
    }
}
```

Generating Code for Control and Logical Expressions

Generating Code for Control and Logical Expressions

The semantics of Java, and so of *j--*, requires that the evaluation of expressions such as `arg1 && arg2` be short-circuited, ie, if `arg1` is `false`, then `arg2` is not evaluated

Generating Code for Control and Logical Expressions

The semantics of Java, and so of *j--*, requires that the evaluation of expressions such as `arg1 && arg2` be short-circuited, ie, if `arg1` is `false`, then `arg2` is not evaluated

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           Branch to target when
    arg1 && arg2 is true:                arg1 && arg2 is false:

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

Generating Code for Control and Logical Expressions

Generating Code for Control and Logical Expressions

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: ...
```

Generating Code for Control and Logical Expressions

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`

```
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);
    }
}
```

Generating Code for Control and Logical Expressions

Generating Code for Control and Logical Expressions

Notice that our method prevents unnecessary branches to branches; for example, consider the slightly more complicated condition in

```
if (a > b && b > c && c > 5) { c = a; } else { c = b; }
```

The JVM code produced for this targets the same exit on `false`, for each of the `&&` operations

```
0:   iload_1
1:   iload_2
2:   if_icmple       16
5:   iload_2
6:   iload_3
7:   if_icmple       16
10:  iload_3
11:  iconst_5
12:  if_icmple       16
13:  iload_1
14:  istore_3
15:  goto            18
16:  iload_2
17:  istore_3
18:  ...
```

Generating Code for Control and Logical Expressions

Notice that our method prevents unnecessary branches to branches; for example, consider the slightly more complicated condition in

```
if (a > b && b > c && c > 5) { c = a; } else { c = b; }
```

The JVM code produced for this targets the same exit on `false`, for each of the `&&` operations

```
0:   iload_1
1:   iload_2
2:   if_icmple       16
5:   iload_2
6:   iload_3
7:   if_icmple       16
10:  iload_3
11:  iconst_5
12:  if_icmple       16
13:  iload_1
14:  istore_3
15:  goto            18
16:  iload_2
17:  istore_3
18:  ...
```

The `codegen()` method in `JLogicalNotOp`

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

Generating Code for Message, Field Selection, and Array Expressions

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` proceeds as follows

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` proceeds as follows

1. If the message expression involves an instance message, `codegen()` generates code for the target

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` 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

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` 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

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` 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
    1. The instruction (`invokevirtual` or `invokestatic`)

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` 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
    1. The instruction (`invokevirtual` or `invokestatic`)
    2. The JVM name for the target's type

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` 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
   1. The instruction (`invokevirtual` or `invokestatic`)
   2. The JVM name for the target's type
   3. The message name

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` 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
    1. The instruction (`invokevirtual` or `invokestatic`)
    2. The JVM name for the target's type
    3. The message name
    4. The descriptor of the invoked method, which was determined in analysis.

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` 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
    1. The instruction (`invokevirtual` or `invokestatic`)
    2. The JVM name for the target's type
    3. The message name
    4. 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 is non-void, then the 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

Generating Code for Message, Field Selection, and Array Expressions

Generating Code for Message, Field Selection, and Array Expressions

For example, the code generated for

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

would be

```
aload s’ # s’ denotes offset of s
bipush  6
invokevirtual   #6; //Method square:(I)I
```

whereas the code generated for

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

would be

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

Generating Code for Message, Field Selection, and Array Expressions

For example, the code generated for

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

would be

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

whereas the code generated for

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

would be

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

We invoke static methods using the `invokestatic` instruction; for example the following *j--* code

```
... = Square.square(5);
```

where `int square(int)` is a static method in `Square`, would generate the following JVM code

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

Generating Code for Message, Field Selection, and Array Expressions

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` works as follows

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` works as follows

1. It generates code for its target; if the target is a class, no code is generated

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` 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; the code generated makes use of the special instruction, `arraylength`

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` 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; the code generated makes 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

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` 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; the code generated makes 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

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` 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; the code generated makes 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
    1. The instruction (`getfield` or `getstatic`)

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` 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; the code generated makes 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
    1. The instruction (`getfield` or `getstatic`)
    2. The JVM name for the target's type

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` 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; the code generated makes 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
    1. The instruction (`getfield` or `getstatic`)
    2. The JVM name for the target's type
    3. The field name

Generating Code for Message, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` 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; the code generated makes 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
    1. The instruction (`getfield` or `getstatic`)
    2. The JVM name for the target's type
    3. The field name
    4. The JVM descriptor for the type of the field, and so the type of the result

Generating Code for Message, Field Selection, and Array Expressions

Generating Code for Message, Field Selection, and Array Expressions

For example, the following code

```
... = s.instanceField;
```

would be translated as

```
aload s’
getfield instanceField
```

whereas the following code

```
... = Square.staticField;
```

would be translated as

```
getstatic staticField
```

Generating Code for Message, Field Selection, and Array Expressions

For example, the following code

```
... = s.instanceField;
```

would be translated as

```
aload s’
getfield instanceField
```

whereas the following code

```
... = Square.staticField;
```

would be translated as

```
getstatic staticField
```

Code generation for array access expressions is straightforward; for example, if the variable `a` references an array object, and `i` is an integer, then the following code

```
... = a[i];
```

is translated to

```
aload a’
iload i’
iaload
```