Introduction to Compiler Construction

JVM Code Generation: Assignment, String Concatenation, Cast, and Other Operations

Outline

1. Generating Code for Assignment and Similar Operations
2. Generating Code for String Concatenation
3. Generating Code for Casts

Generating Code for Assignment and Similar Operations

Generating Code for Assignment and Similar Operations

Consider the simple assignment statement

```
x = y;
```

which asks that the value of the variable `y` be stored in variable `x`

Generating Code for Assignment and Similar Operations

Consider the simple assignment statement

```
x = y;
```

which asks that the value of the variable `y` be stored in variable `x`

We want the *l*-value (address or location) for `x` and the *r*-value (content or value) for `y`

Generating Code for Assignment and Similar Operations

Consider the simple assignment statement

```
x = y;
```

which asks that the value of the variable `y` be stored in variable `x`

We want the *l*-value (address or location) for `x` and the *r*-value (content or value) for `y`

All expressions have *r*-values, but 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, the following have both *l*-values and *r*-values

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

while the following have *r*-values, but not *l*-values

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

Generating Code for Assignment and Similar Operations

Generating Code for Assignment and Similar Operations

The right-hand-side expression is compiled to produce code for computing its *r*-value and leaving it on the stack

Generating Code for Assignment and Similar Operations

The right-hand-side expression is compiled to produce code for computing its *r*-value and leaving it on the stack

For the left-hand-side, sometimes no code needs to be generated, as in the following example

```
x = y;
```

produces

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

Generating Code for Assignment and Similar Operations

The right-hand-side expression is compiled to produce code for computing its *r*-value and leaving it on the stack

For the left-hand-side, sometimes no code needs to be generated, as in the following example

```
x = y;
```

produces

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

On the other hand, compiling

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

produces

```
aload a'
iload x'
iload y'
iastore
```

Generating Code for Assignment and Similar Operations

The right-hand-side expression is compiled to produce code for computing its *r*-value and leaving it on the stack

For the left-hand-side, sometimes no code needs to be generated, as in the following example

```
x = y;
```

produces

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

On the other hand, compiling

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

produces

```
aload a'
iload x'
iload y'
iastore
```

An assignment may act as a statement, as shown below

```
x = y;
```

or as an expression, as shown below

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

Generating Code for Assignment and Similar Operations

Generating Code for Assignment and Similar Operations

In the first case, no value is left on the stack

Generating Code for Assignment and Similar Operations

In the first case, no value is left on the stack

In the second case, `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`, ie, the code might look something like

```
iload y'
dup
istore x'
istore z'
```

Generating Code for Assignment and Similar Operations

In the first case, no value is left on the stack

In the second case, `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`, ie, the code might look something like

```
iload y'
dup
istore x'
istore z'
```

In parsing, when an expression is used as a statement, `Parser`'s `statementExpression()` method sets a flag `isStatementExpression` in the expression node to `true`, and the code generation phase makes use of this flag in deciding when code must be produced for duplicating *r*-values on the run-time stack

Generating Code for Assignment and Similar Operations

In the first case, no value is left on the stack

In the second case, `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`, ie, the code might look something like

```
iload y'
dup
istore x'
istore z'
```

In parsing, when an expression is used as a statement, `Parser`'s `statementExpression()` method sets a flag `isStatementExpression` in the expression node to `true`, and the code generation phase makes use of this flag in deciding when code must be produced for duplicating *r*-values on the run-time stack

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

Generating Code for Assignment and Similar Operations

In the first case, no value is left on the stack

In the second case, `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`, ie, the code might look something like

```
iload y'
dup
istore x'
istore z'
```

In parsing, when an expression is used as a statement, `Parser`'s `statementExpression()` method sets a flag `isStatementExpression` in the expression node to `true`, and the code generation phase makes use of this flag in deciding when code must be produced for duplicating *r*-values on the run-time stack

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

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

Generating Code for Assignment and Similar Operations

Generating Code for Assignment and Similar Operations

The table below compares the various operations (labeled down the left), with an assortment of left-hand sides (labeled across the top)

	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_x2] iastore	aload o' dup getfield f iload y' iadd [dup_x1] putfield f	getstatic sf iload y' iadd [dup] putstatic sf
++lhs	iinc x',1 [iload x']	aload a' iload i' dup2 iaload iconst_1 iadd [dup_x2] iastore	aload o' dup getfield f iconst_1 iadd [dup_x1] putfield f	getstatic sf iconst_1 iadd [dup] putstatic sf
lhs--	[iload x'] iinc x',-1	aload a' iload i' dup2 iaload [dup_x2] iconst_1 isub iastore	aload o' dup getfield f [dup_x1] iconst_1 isub putfield f	getstatic sf [dup] iconst_1 isub putstatic sf

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

Generating Code for Assignment and Similar Operations

Generating Code for Assignment and Similar Operations

The table above suggests four sub-operations common to most of the assignment-like operations in *j--*

Generating Code for Assignment and Similar Operations

The table above suggests four sub-operations common to most of the assignment-like operations in *j--*

1. `codegenLoadLhsLvalue()` - this generates code to load any up-front data for the left-hand side of an assignment needed for an eventual store, ie, its *l*-value

Generating Code for Assignment and Similar Operations

The table above suggests four sub-operations common to most of the assignment-like operations in *j--*

1. `codegenLoadLhsLvalue()` - this generates code to load any up-front data for the left-hand side of an assignment needed for an eventual store, ie, 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

Generating Code for Assignment and Similar Operations

The table above suggests four sub-operations common to most of the assignment-like operations in *j--*

1. `codegenLoadLhsLvalue()` - this generates code to load any up-front data for the left-hand side of an assignment needed for an eventual store, ie, 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

Generating Code for Assignment and Similar Operations

The table above suggests four sub-operations common to most of the assignment-like operations in *j--*

1. `codegenLoadLhsLvalue()` - this generates code to load any up-front data for the left-hand side of an assignment needed for an eventual store, ie, 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

Generating Code for Assignment and Similar Operations

The table above suggests four sub-operations common to most of the assignment-like operations in *j--*

1. `codegenLoadLhsLvalue()` - this generates code to load any up-front data for the left-hand side of an assignment needed for an eventual store, ie, 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`

Generating Code for Assignment and Similar Operations

The table above suggests four sub-operations common to most of the assignment-like operations in *j--*

1. `codegenLoadLhsLvalue()` - this generates code to load any up-front data for the left-hand side of an assignment needed for an eventual store, ie, 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 the table below

	x	a[i]	o.f	C.sf
codegenLoadLhsLvalue()	[none]	aload a' iload 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

Generating Code for Assignment and Similar Operations

Generating Code for Assignment and Similar Operations

Our compiler defines an interface `JLhs`, which declares four abstract methods for these four sub-operations; each of `JVariable`, `JArrayExpression` and `JFieldSelection` implements `JLhs`

Generating Code for Assignment and Similar Operations

Our compiler defines an interface `JLhs`, which declares four abstract methods for these four sub-operations; each of `JVariable`, `JArrayExpression` and `JFieldSelection` implements `JLhs`

Of course, one must also be able to generate code for the right-hand side expression, but `codegen()` is sufficient for that

Generating Code for Assignment and Similar Operations

Our compiler defines an interface `JLhs`, which declares four abstract methods for these four sub-operations; each of `JVariable`, `JArrayExpression` and `JFieldSelection` implements `JLhs`

Of course, one must also be able to generate code for the right-hand side expression, but `codegen()` is sufficient for that

For example, `JPlusAssignOp`'s `codegen()` is shown below

```
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);
}
```

Generating Code for String Concatenation

Generating Code for 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

Generating Code for 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

The compiler's analysis phase determines whether or not string concatenation is implied, and when it is, the concatenation is made explicit, ie, the operation's AST is rewritten, replacing `JPlusOp` with a `JStringConcatenationOp`

Generating Code for 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

The compiler's analysis phase determines whether or not string concatenation is implied, and when it is, the concatenation is made explicit, ie, the operation's AST is rewritten, replacing `JPlusOp` with a `JStringConcatenationOp`

Also, when `x` is a string, analysis replaces

```
x += <expression>
```

by

```
x = x + <expression>
```

Generating Code for String Concatenation

Generating Code for String Concatenation

To implement string concatenation, the compiler generates code to do the following

Generating Code for String Concatenation

To implement string concatenation, the compiler generates code to do the following

1. Create an empty string buffer, ie, a `StringBuffer` object, and initialize it

Generating Code for String Concatenation

To implement string concatenation, the compiler generates code to do the following

1. Create an empty string buffer, ie, 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

Generating Code for String Concatenation

To implement string concatenation, the compiler generates code to do the following

1. Create an empty string buffer, ie, 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`

Generating Code for String Concatenation

To implement string concatenation, the compiler generates code to do the following

1. Create an empty string buffer, ie, 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 operations, which eliminates the instantiation of unnecessary string buffers

Generating Code for String Concatenation

To implement string concatenation, the compiler generates code to do the following

1. Create an empty string buffer, ie, 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 operations, which 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;
```

Generating Code for Casts

Generating Code for Casts

Analysis determines both the validity of a cast and the necessary `Converter`, which encapsulates the code generated for the particular cast

Generating Code for Casts

Analysis determines both the validity of a cast and the necessary `Converter`, which encapsulates the code generated for the particular cast

Each `Converter` implements a method `codegen()`, which generates any code necessary to the cast

Generating Code for Casts

Analysis determines both the validity of a cast and the necessary `Converter`, which encapsulates the code generated for the particular cast

Each `Converter` implements a method `codegen()`, which generates any code necessary to the cast

For example, consider the converter for casting a reference type to one of its sub-types (narrowing cast) which 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());
    }
}
```

Generating Code for Casts

Analysis determines both the validity of a cast and the necessary `Converter`, which encapsulates the code generated for the particular cast

Each `Converter` implements a method `codegen()`, which generates any code necessary to the cast

For example, consider the converter for casting a reference type to one of its sub-types (narrowing cast) which 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

Generating Code for Casts

Generating Code for Casts

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;
```

Generating Code for Casts

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
```

Generating Code for Casts

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`

Generating Code for Casts

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--*