Introduction to Compiler Construction

JVM Code Generation: Classes and their Members

Outline

Generating Code for Classes and their Members

Generating Code for Classes and their Members

`JCompilationUnit.codegen()` drives the generation of code for classes; for each type (ie, class) declaration, it

Generating Code for Classes and their Members

`JCompilationUnit.codegen()` drives the generation of code for classes; for each type (ie, class) declaration, it

- invokes `codegen()` on the `JClassDeclaration` for generating the code for that class,

Generating Code for Classes and their Members

`JCompilationUnit.codegen()` drives the generation of code for classes; for each type (ie, class) declaration, it

- invokes `codegen()` on the `JClassDeclaration` for generating the code for that class,
- writes out the class to a class file in the destination directory, and

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

Generating Code for Classes and their Members

Generating Code for Classes and their Members

`JClassDeclaration.codegen()` does the following

Generating Code for Classes and their Members

`JClassDeclaration.codegen()` does the following

- It computes the fully-qualified name for the class, taking any package name into account

Generating Code for Classes and their Members

`JClassDeclaration.codegen()` 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

Generating Code for Classes and their Members

`JClassDeclaration.codegen()` 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

Generating Code for Classes and their Members

`JClassDeclaration.codegen()` 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.

Generating Code for Classes and their Members

`JClassDeclaration.codegen()` 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

Generating Code for Classes and their Members

Generating Code for Classes and their Members

```java
public void codegen(CLEmitter output) {
    // The class header.
    String qualifiedName = JAST.compilationUnit.packageName().isEmpty() ? name : JAST.compilationUnit.packageName() + "/" + name;
    output.addClass(mods, qualifiedName, superType.jvmName(), null, false);

    // The implicit empty constructor?
    if (!hasExplicitConstructor) {
        codegenImplicitConstructor(output);
    }

    // The members.
    for (JMember member : classBlock) {
        ((JAST) member).codegen(output);
    }

    // Generate a class initialization method.
    if (!staticFieldInitializations.isEmpty()) {
        codegenClassInit(output);
    }
}
```

Generating Code for Classes and their Members

Generating Code for Classes and their Members

JMethodDeclaration.codegen()

```
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);
    }
}
```

Generating Code for Classes and their Members

JMethodDeclaration.codegen()

```
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);
    }
}
```

JConstructorDeclaration.codegen()

```
public void codegen(CLEmitter output) {
    output.addMethod(mods, "<init>", descriptor, null, false);
    if (!invokesConstructor) {
        output.addNoArgInstruction(ALOAD_0);
        output.addMemberAccessInstruction(INVOKESPECIAL, ((JTypeDecl) context.classContext().definition())
                        .superType().jvmName(), "<init>", "()V");
    }

    // Field initializations
    for (JFieldDeclaration field : definingClass.instanceFieldInitializations()) {
        field.codegenInitializations(output);
    }

    // And then the body
    body.codegen(output);
    output.addNoArgInstruction(RETURN);
}
```

Generating Code for Classes and their Members

Generating Code for Classes and their Members

Since the analysis phase has moved initializations, `codegen()` for `JFieldDeclaration` need only generate code for the field declaration itself

Generating Code for Classes and their Members

Since the analysis phase has moved initializations, `codegen()` for `JFieldDeclaration` need only generate code for the field declaration itself

`JFieldDeclaration.codegen()`

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