Jumbo: A Staged Java Compiler

Lars R. Clausen

March 21, 2003
Version 0.8.0

Abstract

Jumbo is a Java compiler that supports staging operations. It can function as a normal Java compiler (except inner classes). It can also perform code creation and manipulation through the addition of extra syntax `$<...>$` for quotation and `‘Expr(...)` for antiquotation. The compiler is designed to allow the creation and transmittal of code fragments to be combined later.

1 Synopsis

jumbo [**options**] *sourcefile...*
stagedjumbo *sourcefile...*
unquotejumbo *sourcefile*

2 Description

Jumbo is a Java compiler that supports staged compilation.

jumbo is a Java compiler that allows staging constructs in the code. See some important limitations in BUGS below.

stagedjumbo is a wrapper around another Java compiler (by default *javac*(1)) that allows the use of constructs not implemented by the Jumbo compiler itself to be used outside of quoted code.

unquotejumbo is a filter that transform quotes into regular Java code, which can then be compiled or passed through other programs.

3 Options

-s Output Java source with quotation syntax removed.

-l Output Java source of the lifted source (for debugging).

-d *dir* Directory where class files will be stored.

-sourcepath ***dir***] Directory to look for source files in.

-v Output the current version and exit.

-h Output usage.

The input files must be Java source files. All files that need to be (re)compiled must be specified on the command line, **Jumbo** will not perform any dependency analysis. If - is the only file listed, **Jumbo** will read source from stdin.

4 Staging

Staging is performed by the manipulation of values of type Code, representing fragments of code. Code fragments can be created and combined using the syntax described here. They can be passed around as any other value. This section describes the syntax of staging, see the Examples section for examples of the usage.

To create a code fragment, enclose Java source in `$<...>$`. This *quotation* syntax is transformed by **Jumbo** into regular Java code that will create and initialize a value of type Code that corresponds to the enclosed source.

To combine code fragments, use the backquote (‘) to indicate *antiquotation*. The backquote must be followed by a syntactic category and a Java expression in parentheses. The Java expression must evaluate to a value that fits the syntactic category,

The presently available syntactic categories are:

Category	Expression value expected (Type)
Expr	Expressions (Code)
Stmt	Statements (Code)
Name	Identifiers (String)
Type	Types (Code)
Case	List of case branches (MonoList containing Code values)
Method	Method declaration (Code)
Field	Field declaration (Code)
Body	List of class members (MonoList containing Code values)
Char	Character constant (char)
Int	Integer constant (int)
Float	Float constant (float)
Long	Long constant (long)
Double	Double constant (double)
Bool	Boolean constant (boolean)
String	String constant (String)

Omitting the syntactic category is equivalent to using Name. When the syntactic category is omitted and the expression is a simple identifier, the parentheses can be omitted. Thus, `$<foo.‘x.>$` is the same as `$<foo.‘Name(x)>$`. Note that the variable in this case must be of type String, not type Code.

There is currently no way to create a Code value of the Field type. This is a limitation in the parser. The Body type can only be made as a list of Methods types.

Code values can be translated into class files only if they contain a class definition, as that is the smallest unit of binary. To generate all class files in a Code value, invoke the `void generate()` method on the Code object. To generate all class files and load one of the classes, invoke the `Object load(String classname)` method on the Code value. Only the class whose name is passed to `load()` is loaded, and it must have a zero-argument constructor.

Hygienic variables (with guaranteed unique names within the scope) can be created with `new Name(<string>)`. `Name` is a subclass of `Code` representing a variable. Note that creating a Name object does not create the variable definition for that name. For example:

```
Name i = new Name("i");
Code c = $<for (int ‘i = 0; ‘i < max; ‘i++) { ... }>$;
```

5 Notes

The preprocesser part of **stagedjumbo** currently performs some shuffling of the source files to satisfy *javac*(1)'s requirements for file naming. When compiling `Foo.java`, the original source is temporarily moved to `Foo.java-orig`, and the version with the quotation syntax transformed into Java syntax is placed in its stead. After compilation, the original source file is moved back. If there were any errors in compilation, the transformed version is retained as `Foo.java-dequoted`.

Do not attempt to break the compilation process of **stagedjumbo**, in particular the last part. It may leave the transformed version in place of the original source, and if you run **stagedjumbo** again, your original source will be lost.

6 Return value

Jumbo programs return 0 on success. **jumbo** returns 1 if the compilation fails. **stagedjumbo** returns 1 if the unquoting prepass fails, and 2 if the call to another compiler fails. **unquotejumbo** returns 1 if the parsing fails.

7 Examples

Staged version of Hello, world:

```
import uiuc.Jumbo.Util.*;
import uiuc.Jumbo.Jaemus.*;
import uiuc.Jumbo.Compiler.*;
```

```
public class HelloStaged {
  public static void main(String[] argv) {
    Code c = $<public class Hello {
                 public static void main(String[] argv) {
                   System.out.println("Hello, world");
                 }
               }>$;
    c.generate();
  }
}
$ jumbo HelloStaged.java
...
$ java HelloStaged
$ ls
HelloStaged.java  HelloStaged.class Hello.class
$ java Hello
Hello, world
```

Staged version of Hello, world that incorporates the first argument as a static string:

```
import uiuc.Jumbo.Util.*;
import uiuc.Jumbo.Jaemus.*;
import uiuc.Jumbo.Compiler.*;

public class HelloStaged {
  public static void main(String[] argv) {
    Code c = $<public class Hello {
                 public static void main(String[] argv) {
                   System.out.println("Hello, "+‘String(argv[0]));
                 }
               }>$;
    c.generate();
  }
}
$ jumbo HelloStaged.java
...
$ java HelloStaged Jim
$ ls
HelloStaged.java  HelloStaged.class Hello.class
$ java Hello
Hello, Jim
```

In this example, we staged the evaluation of a dot product. When executing DotStaged, we give the first vector, which is encoded in an expression in the Dot class. When executing Dot, we give the second vector, and the dot product of the two vectors is returned. Note how a String value is used to pass the name

of a variable, and how an expression is built iteratively by embedding the old expression in a quoted expression.

```
import uiuc.Jumbo.Util.*;
import uiuc.Jumbo.Jaemus.*;
import uiuc.Jumbo.Compiler.*;

public class DotStaged {
  public static Code makeDot(double[] V1, String V2) {
    Code c = $<0.0>$;
    for (int i = 0; i < V1.length; i++) {
      c = $<‘Expr(c) + ‘V2[‘Int(i)] * ‘Double(V1[i])>$;
    }
    return c;
  }
  public static void main(String[] argv) {
    double[] v = new double[argv.length];
    for (int i = 0; i < v.length; i++)
      v[i] = Double.parseDouble(argv[i]);
    Code c =
$<public class Dot {
    public static void main(String[] argv) {
      if (argv.length != ‘Int(v.length))
        throw new Error("Wrong length vector");
      double[] w = new double[argv.length];
      for (int i = 0; i < w.length; i++)
        w[i] = Double.parseDouble(argv[i]);
      System.out.println(‘Expr(makeDot(v, "w")));
    }
  }>$;
    c.generate();
  }
}
$ jumbo DotStaged.java
...
$ java DotStaged 4.0 2.0 3.2 3.0
$ ls
DotStaged.java  DotStaged.class Dot.class
$ java Dot 1.0 2.0 0.0 1.0
11.0
```

8 Files

***javac*(1)** Jumbo requires *javac*(1) compiler (or an equivalent) for the use of inner classes or any other unimplemented feature. This is only necessary to run **stagedjumbo**.

9 Environment

JAVAC Sets the compiler used by **stagedjumbo** internally. Defaults to *javac*(1).

CLASSPATH Must include the directory containing the compiler classes.

10 See also

javac(1)

11 Bugs

Jumbo doesn't support some aspects of inner classes yet, including multiple levels of nesting and access to private outer members. By using **stagedjumbo**, the non-quoted code is compiled with a normal Java compiler ($JAVAC or *javac*(1) by default). This allows missing features or bugs in **Jumbo** to be avoided for the unquoted parts.

Jumbo may create class files that will be rejected by the verifier in the JVM, but only for invalid input. It currently doesn't check for variable initialization and loss of precision.

Jumbo does not do any dependency checking of the source files. If a change in one source file requires another to be recompiled, the user needs to specify the other source file on the command line.

The error messages from **Jumbo** are simply exceptions not caught. They rarely include line numbers, and can be quite cryptic at times.

Some areas of **Jumbo** have not been extensively tested. Caveat hackor.

12 Authors

Lars R. Clausen `lrclause+jumbo@cs.uiuc.edu`
Ava A. Jarvis `ajar@katanalynx.dyndns.org`
Sam N. Kamin `kamin@cs.uiuc.edu`