```
// Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import java.util.ArrayList;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeMap;
import java.util.Map.Entry;
import static jminusminus.CLConstants.*;
import static jminusminus.CLConstants.Category.*;

/**
 * Representation of a JVM instruction. It stores the opcode and mnenomic of an
 * instruction, its operand count (DYNAMIC if the instruction has variable
 * number operands), pc (location counter), stack units (words produced - words
 * consumed from the operand stack), and local variable index (IRRELEVANT if the
 * instruction does not operate on local variables).
 */

abstract class CLInstruction {

    /** Opcode for this instruction. */
    protected int opcode;

    /** Mnemonic for this instruction. */
    protected String mnemonic;

    /**
     * Number of operands for this instruction; determined statically for all
     * instructions except TABLESWITCH and LOOKUPSWITCH.
     */
    protected int operandCount;

    /**
     * Location counter; index of this instruction within the code array of a
     * method.
     */
    protected int pc;

    /**
     * Stack units; words produced - words consumed from the operand stack by
     * this instruction.
     */
    protected int stackUnits;

    /**
     * Index of the local variable that this instruction refers to; applies only
     * to instructions that operate on local variables.
     */
    protected int localVariableIndex;

    /**
     * For each JVM instruction, this array stores its opcode, mnemonic, number
     * of operands (DYNAMIC for instructions with variable attribute count),
     * local variable index (IRRELEVANT where not applicable), stack units, and
     * instruction category. For example, for IMUL, these parameters are IMUL,
     * "imul", 0, IRRELEVANT, -1, ARITHMETIC1.
     */
    public static final CLInsInfo[] instructionInfo = {
            new CLInsInfo(NOP, "nop", 0, IRRELEVANT, 0, MISC),
            new CLInsInfo(ACONST_NULL, "aconst_null", 0, IRRELEVANT, 1,
                    LOAD_STORE1),
            new CLInsInfo(ICONST_M1, "iconst_m1", 0, IRRELEVANT, 1, LOAD_STORE1),
            new CLInsInfo(ICONST_0, "iconst_0", 0, IRRELEVANT, 1, LOAD_STORE1),
```

```
67              new CLInsInfo(ICONST_1, "iconst_1", 0, IRRELEVANT, 1, LOAD_STORE1),
68              new CLInsInfo(ICONST_2, "iconst_2", 0, IRRELEVANT, 1, LOAD_STORE1),
69              new CLInsInfo(ICONST_3, "iconst_3", 0, IRRELEVANT, 1, LOAD_STORE1),
70              new CLInsInfo(ICONST_4, "iconst_4", 0, IRRELEVANT, 1, LOAD_STORE1),
71              new CLInsInfo(ICONST_5, "iconst_5", 0, IRRELEVANT, 1, LOAD_STORE1),
72              new CLInsInfo(LCONST_0, "lconst_0", 0, IRRELEVANT, 2, LOAD_STORE1),
73              new CLInsInfo(LCONST_1, "lconst_1", 0, IRRELEVANT, 2, LOAD_STORE1),
74              new CLInsInfo(FCONST_0, "fconst_0", 0, IRRELEVANT, 1, LOAD_STORE1),
75              new CLInsInfo(FCONST_1, "fconst_1", 0, IRRELEVANT, 1, LOAD_STORE1),
76              new CLInsInfo(FCONST_2, "fconst_2", 0, IRRELEVANT, 1, LOAD_STORE1),
77              new CLInsInfo(DCONST_0, "dconst_0", 0, IRRELEVANT, 2, LOAD_STORE1),
78              new CLInsInfo(DCONST_1, "dconst_1", 0, IRRELEVANT, 2, LOAD_STORE1),
79              new CLInsInfo(BIPUSH, "bipush", 1, IRRELEVANT, 1, LOAD_STORE3),
80              new CLInsInfo(SIPUSH, "sipush", 2, IRRELEVANT, 1, LOAD_STORE3),
81              new CLInsInfo(LDC, "ldc", 1, IRRELEVANT, 1, LOAD_STORE4),
82              new CLInsInfo(LDC_W, "ldc_w", 2, IRRELEVANT, 1, LOAD_STORE4),
83              new CLInsInfo(LDC2_W, "ldc2_w", 2, IRRELEVANT, 2, LOAD_STORE4),
84              new CLInsInfo(ILOAD, "iload", 1, DYNAMIC, 1, LOAD_STORE2),
85              new CLInsInfo(LLOAD, "lload", 1, DYNAMIC, 2, LOAD_STORE2),
86              new CLInsInfo(FLOAD, "fload", 1, DYNAMIC, 1, LOAD_STORE2),
87              new CLInsInfo(DLOAD, "dload", 1, DYNAMIC, 2, LOAD_STORE2),
88              new CLInsInfo(ALOAD, "aload", 1, DYNAMIC, 1, LOAD_STORE2),
89              new CLInsInfo(ILOAD_0, "iload_0", 0, 0, 1, LOAD_STORE1),
90              new CLInsInfo(ILOAD_1, "iload_1", 0, 1, 1, LOAD_STORE1),
91              new CLInsInfo(ILOAD_2, "iload_2", 0, 2, 1, LOAD_STORE1),
92              new CLInsInfo(ILOAD_3, "iload_3", 0, 3, 1, LOAD_STORE1),
93              new CLInsInfo(LLOAD_0, "lload_0", 0, 0, 2, LOAD_STORE1),
94              new CLInsInfo(LLOAD_1, "lload_1", 0, 1, 2, LOAD_STORE1),
95              new CLInsInfo(LLOAD_2, "lload_2", 0, 2, 2, LOAD_STORE1),
96              new CLInsInfo(LLOAD_3, "lload_3", 0, 3, 2, LOAD_STORE1),
97              new CLInsInfo(FLOAD_0, "fload_0", 0, 0, 1, LOAD_STORE1),
98              new CLInsInfo(FLOAD_1, "fload_1", 0, 1, 1, LOAD_STORE1),
99              new CLInsInfo(FLOAD_2, "fload_2", 0, 2, 1, LOAD_STORE1),
100             new CLInsInfo(FLOAD_3, "fload_3", 0, 3, 1, LOAD_STORE1),
101             new CLInsInfo(DLOAD_0, "dload_0", 0, 0, 2, LOAD_STORE1),
102             new CLInsInfo(DLOAD_1, "dload_1", 0, 1, 2, LOAD_STORE1),
103             new CLInsInfo(DLOAD_2, "dload_2", 0, 2, 2, LOAD_STORE1),
104             new CLInsInfo(DLOAD_3, "dload_3", 0, 3, 2, LOAD_STORE1),
105             new CLInsInfo(ALOAD_0, "aload_0", 0, 0, 1, LOAD_STORE1),
106             new CLInsInfo(ALOAD_1, "aload_1", 0, 1, 1, LOAD_STORE1),
107             new CLInsInfo(ALOAD_2, "aload_2", 0, 2, 1, LOAD_STORE1),
108             new CLInsInfo(ALOAD_3, "aload_3", 0, 3, 1, LOAD_STORE1),
109             new CLInsInfo(IALOAD, "iaload", 0, IRRELEVANT, -1, ARRAY2),
110             new CLInsInfo(LALOAD, "laload", 0, IRRELEVANT, 0, ARRAY2),
111             new CLInsInfo(FALOAD, "faload", 0, IRRELEVANT, -1, ARRAY2),
112             new CLInsInfo(DALOAD, "daload", 0, IRRELEVANT, 0, ARRAY2),
113             new CLInsInfo(AALOAD, "aaload", 0, IRRELEVANT, -1, ARRAY2),
114             new CLInsInfo(BALOAD, "baload", 0, IRRELEVANT, -1, ARRAY2),
115             new CLInsInfo(CALOAD, "caload", 0, IRRELEVANT, -1, ARRAY2),
116             new CLInsInfo(SALOAD, "saload", 0, IRRELEVANT, -1, ARRAY2),
117             new CLInsInfo(ISTORE, "istore", 1, DYNAMIC, -1, LOAD_STORE2),
118             new CLInsInfo(LSTORE, "lstore", 1, DYNAMIC, -2, LOAD_STORE2),
119             new CLInsInfo(FSTORE, "fstore", 1, DYNAMIC, -1, LOAD_STORE2),
120             new CLInsInfo(DSTORE, "dstore", 1, DYNAMIC, -2, LOAD_STORE2),
121             new CLInsInfo(ASTORE, "astore", 1, DYNAMIC, -1, LOAD_STORE2),
122             new CLInsInfo(ISTORE_0, "istore_0", 0, 0, -1, LOAD_STORE1),
123             new CLInsInfo(ISTORE_1, "istore_1", 0, 1, -1, LOAD_STORE1),
124             new CLInsInfo(ISTORE_2, "istore_2", 0, 2, -1, LOAD_STORE1),
125             new CLInsInfo(ISTORE_3, "istore_3", 0, 3, -1, LOAD_STORE1),
126             new CLInsInfo(LSTORE_0, "lstore_0", 0, 0, -2, LOAD_STORE1),
127             new CLInsInfo(LSTORE_1, "lstore_1", 0, 1, -2, LOAD_STORE1),
128             new CLInsInfo(LSTORE_2, "lstore_2", 0, 2, -2, LOAD_STORE1),
129             new CLInsInfo(LSTORE_3, "lstore_3", 0, 3, -2, LOAD_STORE1),
130             new CLInsInfo(FSTORE_0, "fstore_0", 0, 0, -1, LOAD_STORE1),
131             new CLInsInfo(FSTORE_1, "fstore_1", 0, 1, -1, LOAD_STORE1),
132             new CLInsInfo(FSTORE_2, "fstore_2", 0, 2, -1, LOAD_STORE1),
133             new CLInsInfo(FSTORE_3, "fstore_3", 0, 3, -1, LOAD_STORE1),
134             new CLInsInfo(DSTORE_0, "dstore_0", 0, 0, -2, LOAD_STORE1),
135             new CLInsInfo(DSTORE_1, "dstore_1", 0, 1, -2, LOAD_STORE1),
```

```
            new CLInsInfo(DSTORE_2, "dstore_2", 0, 2, -2, LOAD_STORE1),
            new CLInsInfo(DSTORE_3, "dstore_3", 0, 3, -2, LOAD_STORE1),
            new CLInsInfo(ASTORE_0, "astore_0", 0, 0, -1, LOAD_STORE1),
            new CLInsInfo(ASTORE_1, "astore_1", 0, 1, -1, LOAD_STORE1),
            new CLInsInfo(ASTORE_2, "astore_2", 0, 2, -1, LOAD_STORE1),
            new CLInsInfo(ASTORE_3, "astore_3", 0, 3, -1, LOAD_STORE1),
            new CLInsInfo(IASTORE, "iastore", 0, IRRELEVANT, -3, ARRAY2),
            new CLInsInfo(LASTORE, "lastore", 0, IRRELEVANT, -4, ARRAY2),
            new CLInsInfo(FASTORE, "fastore", 0, IRRELEVANT, -3, ARRAY2),
            new CLInsInfo(DASTORE, "dastore", 0, IRRELEVANT, -4, ARRAY2),
            new CLInsInfo(AASTORE, "aastore", 0, IRRELEVANT, -3, ARRAY2),
            new CLInsInfo(BASTORE, "bastore", 0, IRRELEVANT, -3, ARRAY2),
            new CLInsInfo(CASTORE, "castore", 0, IRRELEVANT, -3, ARRAY2),
            new CLInsInfo(SASTORE, "sastore", 0, IRRELEVANT, -3, ARRAY2),
            new CLInsInfo(POP, "pop", 0, IRRELEVANT, -1, STACK),
            new CLInsInfo(POP2, "pop2", 0, IRRELEVANT, -2, STACK),
            new CLInsInfo(DUP, "dup", 0, IRRELEVANT, 1, STACK),
            new CLInsInfo(DUP_X1, "dup_x1", 0, IRRELEVANT, 1, STACK),
            new CLInsInfo(DUP_X2, "dup_x2", 0, IRRELEVANT, 1, STACK),
            new CLInsInfo(DUP2, "dup2", 0, IRRELEVANT, 2, STACK),
            new CLInsInfo(DUP2_X1, "dup2_x1", 0, IRRELEVANT, 2, STACK),
            new CLInsInfo(DUP2_X2, "dup2_x2", 0, IRRELEVANT, 2, STACK),
            new CLInsInfo(SWAP, "swap", 0, IRRELEVANT, 0, STACK),
            new CLInsInfo(IADD, "iadd", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(LADD, "ladd", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(FADD, "fadd", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(DADD, "dadd", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(ISUB, "isub", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(LSUB, "lsub", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(FSUB, "fsub", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(DSUB, "dsub", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(IMUL, "imul", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(LMUL, "lmul", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(FMUL, "fmul", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(DMUL, "dmul", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(IDIV, "idiv", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(LDIV, "ldiv", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(FDIV, "fdiv", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(DDIV, "ddiv", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(IREM, "irem", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(LREM, "lrem", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(FREM, "frem", 0, IRRELEVANT, -1, ARITHMETIC1),
            new CLInsInfo(DREM, "drem", 0, IRRELEVANT, -2, ARITHMETIC1),
            new CLInsInfo(INEG, "ineg", 0, IRRELEVANT, 0, ARITHMETIC1),
            new CLInsInfo(LNEG, "lneg", 0, IRRELEVANT, 0, ARITHMETIC1),
            new CLInsInfo(FNEG, "fneg", 0, IRRELEVANT, 0, ARITHMETIC1),
            new CLInsInfo(DNEG, "dneg", 0, IRRELEVANT, 0, ARITHMETIC1),
            new CLInsInfo(ISHL, "ishl", 0, IRRELEVANT, -1, BIT),
            new CLInsInfo(LSHL, "lshl", 0, IRRELEVANT, -2, BIT),
            new CLInsInfo(ISHR, "ishr", 0, IRRELEVANT, -1, BIT),
            new CLInsInfo(LSHR, "lshr", 0, IRRELEVANT, -2, BIT),
            new CLInsInfo(IUSHR, "iushr", 0, IRRELEVANT, -1, BIT),
            new CLInsInfo(LUSHR, "lushr", 0, IRRELEVANT, -2, BIT),
            new CLInsInfo(IAND, "iand", 0, IRRELEVANT, -1, BIT),
            new CLInsInfo(LAND, "land", 0, IRRELEVANT, -2, BIT),
            new CLInsInfo(IOR, "ior", 0, IRRELEVANT, -1, BIT),
            new CLInsInfo(LOR, "lor", 0, IRRELEVANT, -2, BIT),
            new CLInsInfo(IXOR, "ixor", 0, IRRELEVANT, -1, BIT),
            new CLInsInfo(LXOR, "lxor", 0, IRRELEVANT, -2, BIT),
            new CLInsInfo(IINC, "iinc", 2, DYNAMIC, 0, ARITHMETIC2),
            new CLInsInfo(I2L, "i2l", 0, IRRELEVANT, 1, CONVERSION),
            new CLInsInfo(I2F, "i2f", 0, IRRELEVANT, 0, CONVERSION),
            new CLInsInfo(I2D, "i2d", 0, IRRELEVANT, 1, CONVERSION),
            new CLInsInfo(L2I, "l2i", 0, IRRELEVANT, -1, CONVERSION),
            new CLInsInfo(L2F, "l2f", 0, IRRELEVANT, -1, CONVERSION),
            new CLInsInfo(L2D, "l2d", 0, IRRELEVANT, 0, CONVERSION),
            new CLInsInfo(F2I, "f2i", 0, IRRELEVANT, 0, CONVERSION),
            new CLInsInfo(F2L, "f2l", 0, IRRELEVANT, 1, CONVERSION),
            new CLInsInfo(F2D, "f2d", 0, IRRELEVANT, 1, CONVERSION),
```

```
            new CLInsInfo(D2I, "d2i", 0, IRRELEVANT, -1, CONVERSION),
            new CLInsInfo(D2L, "d2l", 0, IRRELEVANT, 0, CONVERSION),
            new CLInsInfo(D2F, "d2f", 0, IRRELEVANT, -1, CONVERSION),
            new CLInsInfo(I2B, "i2b", 0, IRRELEVANT, 0, CONVERSION),
            new CLInsInfo(I2C, "i2c", 0, IRRELEVANT, 0, CONVERSION),
            new CLInsInfo(I2S, "i2s", 0, IRRELEVANT, 0, CONVERSION),
            new CLInsInfo(LCMP, "lcmp", 0, IRRELEVANT, -3, COMPARISON),
            new CLInsInfo(FCMPL, "fcmpl", 0, IRRELEVANT, -1, COMPARISON),
            new CLInsInfo(FCMPG, "fcmpg", 0, IRRELEVANT, -1, COMPARISON),
            new CLInsInfo(DCMPL, "dcmpl", 0, IRRELEVANT, -3, COMPARISON),
            new CLInsInfo(DCMPG, "dcmpg", 0, IRRELEVANT, -3, COMPARISON),
            new CLInsInfo(IFEQ, "ifeq", 2, IRRELEVANT, -1, FLOW_CONTROL1),
            new CLInsInfo(IFNE, "ifne", 2, IRRELEVANT, -1, FLOW_CONTROL1),
            new CLInsInfo(IFLT, "iflt", 2, IRRELEVANT, -1, FLOW_CONTROL1),
            new CLInsInfo(IFGE, "ifge", 2, IRRELEVANT, -1, FLOW_CONTROL1),
            new CLInsInfo(IFGT, "ifgt", 2, IRRELEVANT, -1, FLOW_CONTROL1),
            new CLInsInfo(IFLE, "ifle", 2, IRRELEVANT, -1, FLOW_CONTROL1),
            new CLInsInfo(IF_ICMPEQ, "if_icmpeq", 2, IRRELEVANT, -2,
                    FLOW_CONTROL1),
            new CLInsInfo(IF_ICMPNE, "if_icmpne", 2, IRRELEVANT, -2,
                    FLOW_CONTROL1),
            new CLInsInfo(IF_ICMPLT, "if_icmplt", 2, IRRELEVANT, -2,
                    FLOW_CONTROL1),
            new CLInsInfo(IF_ICMPGE, "if_icmpge", 2, IRRELEVANT, -2,
                    FLOW_CONTROL1),
            new CLInsInfo(IF_ICMPGT, "if_icmpgt", 2, IRRELEVANT, -2,
                    FLOW_CONTROL1),
            new CLInsInfo(IF_ICMPLE, "if_icmple", 2, IRRELEVANT, -2,
                    FLOW_CONTROL1),
            new CLInsInfo(IF_ACMPEQ, "if_acmpeq", 2, IRRELEVANT, -2,
                    FLOW_CONTROL1),
            new CLInsInfo(IF_ACMPNE, "if_acmpne", 2, IRRELEVANT, -2,
                    FLOW_CONTROL1),
            new CLInsInfo(GOTO, "goto", 2, IRRELEVANT, 0, FLOW_CONTROL1),
            new CLInsInfo(JSR, "jsr", 2, IRRELEVANT, 1, FLOW_CONTROL1),
            new CLInsInfo(RET, "ret", 1, IRRELEVANT, 0, FLOW_CONTROL2),
            new CLInsInfo(TABLESWITCH, "tableswitch", DYNAMIC, IRRELEVANT, -1,
                    FLOW_CONTROL3),
            new CLInsInfo(LOOKUPSWITCH, "lookupswitch", DYNAMIC, IRRELEVANT,
                    -1, FLOW_CONTROL4),
            new CLInsInfo(IRETURN, "ireturn", 0, IRRELEVANT, EMPTY_STACK,
                    METHOD2),
            new CLInsInfo(LRETURN, "lreturn", 0, IRRELEVANT, EMPTY_STACK,
                    METHOD2),
            new CLInsInfo(FRETURN, "freturn", 0, IRRELEVANT, EMPTY_STACK,
                    METHOD2),
            new CLInsInfo(DRETURN, "dreturn", 0, IRRELEVANT, EMPTY_STACK,
                    METHOD2),
            new CLInsInfo(ARETURN, "areturn", 0, IRRELEVANT, EMPTY_STACK,
                    METHOD2),
            new CLInsInfo(RETURN, "return", 0, IRRELEVANT, EMPTY_STACK, METHOD2),
            new CLInsInfo(GETSTATIC, "getstatic", 2, IRRELEVANT, DYNAMIC, FIELD),
            new CLInsInfo(PUTSTATIC, "putstatic", 2, IRRELEVANT, DYNAMIC, FIELD),
            new CLInsInfo(GETFIELD, "getfield", 2, IRRELEVANT, DYNAMIC, FIELD),
            new CLInsInfo(PUTFIELD, "putfield", 2, IRRELEVANT, DYNAMIC, FIELD),
            new CLInsInfo(INVOKEVIRTUAL, "invokevirtual", 2, IRRELEVANT,
                    DYNAMIC, METHOD1),
            new CLInsInfo(INVOKESPECIAL, "invokespecial", 2, IRRELEVANT,
                    DYNAMIC, METHOD1),
            new CLInsInfo(INVOKESTATIC, "invokestatic", 2, IRRELEVANT, DYNAMIC,
                    METHOD1),
            new CLInsInfo(INVOKEINTERFACE, "invokeinterface", 4, IRRELEVANT,
                    DYNAMIC, METHOD1),
            new CLInsInfo(INVOKEDYNAMIC, "invokedynamic", 2, IRRELEVANT,
                    DYNAMIC, METHOD1),
            new CLInsInfo(NEW, "new", 2, IRRELEVANT, 1, OBJECT),
            new CLInsInfo(NEWARRAY, "newarray", 1, IRRELEVANT, 0, ARRAY1),
            new CLInsInfo(ANEWARRAY, "anewarray", 2, IRRELEVANT, 0, ARRAY1),
            new CLInsInfo(ARRAYLENGTH, "arraylength", 0, IRRELEVANT, 0, ARRAY2),
```

```
            new CLInsInfo(ATHROW, "athrow", 0, IRRELEVANT, UNIT_SIZE_STACK,
                    MISC),
            new CLInsInfo(CHECKCAST, "checkcast", 2, IRRELEVANT, 0, OBJECT),
            new CLInsInfo(INSTANCEOF, "instanceof", 2, IRRELEVANT, 0, OBJECT),
            new CLInsInfo(MONITORENTER, "monitorenter", 0, IRRELEVANT, -1, MISC),
            new CLInsInfo(MONITOREXIT, "monitorexit", 0, IRRELEVANT, -1, MISC),
            new CLInsInfo(WIDE, "wide", 3, IRRELEVANT, 0, LOAD_STORE1),
            new CLInsInfo(MULTIANEWARRAY, "multianewarray", 3, IRRELEVANT, 0,
                    ARRAY3),
            new CLInsInfo(IFNULL, "ifnull", 2, IRRELEVANT, -1, FLOW_CONTROL1),
            new CLInsInfo(IFNONNULL, "ifnonnull", 2, IRRELEVANT, -1,
                    FLOW_CONTROL1),
            new CLInsInfo(GOTO_W, "goto_w", 4, IRRELEVANT, 0, FLOW_CONTROL1),
            new CLInsInfo(JSR_W, "jsr_w", 4, IRRELEVANT, 1, FLOW_CONTROL1) };

    /**
     * Return true if the opcode is valid; false otherwise.
     *
     * @param opcode
     *            instruction opcode.
     * @return true or false.
     */

    public static boolean isValid(int opcode) {
        return NOP <= opcode && opcode <= JSR_W;
    }

    /**
     * Return the opcode for this instruction.
     *
     * @return the opcode.
     */

    public int opcode() {
        return opcode;
    }

    /**
     * Return the mnemonic for this instruction.
     *
     * @return the mnemonic.
     */

    public String mnemonic() {
        return mnemonic;
    }

    /**
     * Return the number of operands for this instruction.
     *
     * @return number of operands.
     */

    public int operandCount() {
        return operandCount;
    }

    /**
     * Return the pc for this instruction.
     *
     * @return the pc.
     */

    public int pc() {
        return pc;
    }

    /**
     * Return the stack units for this instruction.
```

```
343      *
344      * @return the stack units.
345      */
346
347     public int stackUnits() {
348         return stackUnits;
349     }
350
351     /**
352      * Return the local variable index for this instruction.
353      *
354      * @return the local variable index.
355      */
356
357     public int localVariableIndex() {
358         return localVariableIndex;
359     }
360
361     /**
362      * Return the bytecode for this instruction.
363      *
364      * @return bytecode.
365      */
366
367     public abstract ArrayList<Integer> toBytes();
368
369     /**
370      * Return the byte from i at position byteNum.
371      *
372      * @param i
373      *            number whose individual byte is required.
374      * @param byteNum
375      *            the byte to return; 1 (lower) - 4 (higher) instructions.
376      * @return the byte at the specified position
377      */
378
379     protected int byteAt(int i, int byteNum) {
380         int j = 0, mask = 0xFF;
381         switch (byteNum) {
382         case 1: // lower order
383             j = i & mask;
384             break;
385         case 2:
386             j = (i >> 8) & mask;
387             break;
388         case 3:
389             j = (i >> 16) & mask;
390             break;
391         case 4: // higher order
392             j = (i >> 24) & mask;
393             break;
394         }
395         return j;
396     }
397
398 }
399
400 /**
401  * Representation for OBJECT instructions.
402  */
403
404 class CLObjectInstruction extends CLInstruction {
405
406     /**
407      * Index into the constant pool, the item at which identifies the object
408      * type.
409      */
410     private int index;
411
```

```
412     /**
413      * Construct a CLObjectInstruction object.
414      *
415      * @param opcode
416      *            the opcode for this instruction.
417      * @param pc
418      *            index of this instruction within the code array of a method.
419      * @param index
420      *            index into the constant pool, the item at which identifies the
421      *            object.
422      */
423
424     public CLObjectInstruction(int opcode, int pc, int index) {
425         super.opcode = opcode;
426         super.pc = pc;
427         mnemonic = instructionInfo[opcode].mnemonic;
428         operandCount = instructionInfo[opcode].operandCount;
429         stackUnits = instructionInfo[opcode].stackUnits;
430         localVariableIndex = instructionInfo[opcode].localVariableIndex;
431         this.index = index;
432     }
433
434     /**
435      * @inheritDoc
436      */
437
438     public ArrayList<Integer> toBytes() {
439         ArrayList<Integer> bytes = new ArrayList<Integer>();
440         bytes.add(opcode);
441         bytes.add(byteAt(index, 2));
442         bytes.add(byteAt(index, 1));
443         return bytes;
444     }
445
446 }
447
448 /**
449  * Representation for FIELD instructions.
450  */
451
452 class CLFieldInstruction extends CLInstruction {
453
454     /**
455      * Index into the constant pool, the item at which contains the name and
456      * descriptor of the field.
457      */
458     private int index;
459
460     /**
461      * Construct a CLFieldInstruction object.
462      *
463      * @param opcode
464      *            the opcode for this instruction.
465      * @param pc
466      *            index of this instruction within the code array of a method.
467      * @param index
468      *            index into the constant pool, the item at which contains the
469      *            name and descriptor of the field.
470      * @param stackUnits
471      *            words produced - words consumed from the operand stack by this
472      *            instruction.
473      */
474
475     public CLFieldInstruction(int opcode, int pc, int index, int stackUnits) {
476         super.opcode = opcode;
477         super.pc = pc;
478         mnemonic = instructionInfo[opcode].mnemonic;
479         operandCount = instructionInfo[opcode].operandCount;
480         super.stackUnits = stackUnits;
```

```
481         localVariableIndex = instructionInfo[opcode].localVariableIndex;
482         this.index = index;
483     }
484 
485     /**
486      * @inheritDoc
487      */
488 
489     public ArrayList<Integer> toBytes() {
490         ArrayList<Integer> bytes = new ArrayList<Integer>();
491         bytes.add(opcode);
492         bytes.add(byteAt(index, 2));
493         bytes.add(byteAt(index, 1));
494         return bytes;
495     }
496 
497 }
498 
499 /**
500  * Representation for METHOD1 and METHOD2 instructions.
501  */
502 
503 class CLMethodInstruction extends CLInstruction {
504 
505     /**
506      * Index into the constant pool, the item at which contains the name and
507      * descriptor of the method.
508      */
509     private int index;
510 
511     /**
512      * Number of arguments in case of INVOKEINTERFACE instruction.
513      */
514     private int nArgs;
515 
516     /**
517      * Construct a CLMethodInstruction object for METHOD1 instructions.
518      *
519      * @param opcode
520      *            the opcode for this instruction.
521      * @param pc
522      *            index of this instruction within the code array of a method.
523      * @param index
524      *            index into the constant pool, the item at which contains the
525      *            name and descriptor of the method.
526      * @param stackUnits
527      *            words produced - words consumed from the operand stack by this
528      *            instruction.
529      */
530 
531     public CLMethodInstruction(int opcode, int pc, int index, int stackUnits) {
532         super.opcode = opcode;
533         super.pc = pc;
534         mnemonic = instructionInfo[opcode].mnemonic;
535         operandCount = instructionInfo[opcode].operandCount;
536         super.stackUnits = stackUnits;
537         localVariableIndex = instructionInfo[opcode].localVariableIndex;
538         this.index = index;
539     }
540 
541     /**
542      * Construct a CLMethodInstruction object for METHOD2 instructions.
543      *
544      * @param opcode
545      *            the opcode for this instruction.
546      * @param pc
547      *            index of this instruction within the code array of a method.
548      */
549 
```

```
    public CLMethodInstruction(int opcode, int pc) {
        super.opcode = opcode;
        super.pc = pc;
        mnemonic = instructionInfo[opcode].mnemonic;
        operandCount = instructionInfo[opcode].operandCount;
        stackUnits = instructionInfo[opcode].stackUnits;
        localVariableIndex = instructionInfo[opcode].localVariableIndex;
    }

    /**
     * Set the number of arguments for the method for INVOKEINTERFACE
     * instruction.
     *
     * @param nArgs
     *            number of arguments for the method.
     */

    public void setArgumentCount(int nArgs) {
        this.nArgs = nArgs;
    }

    /**
     * @inheritDoc
     */

    public ArrayList<Integer> toBytes() {
        ArrayList<Integer> bytes = new ArrayList<Integer>();
        bytes.add(opcode);
        if (instructionInfo[opcode].category == METHOD1) {
            bytes.add(byteAt(index, 2));
            bytes.add(byteAt(index, 1));

            // INVOKEINTERFACE expects the number of arguments of
            // the method as the third operand and a fourth
            // argument which must always be 0.
            if (opcode == INVOKEINTERFACE) {
                bytes.add(byteAt(nArgs, 1));
                bytes.add(0);
            }
        }
        return bytes;
    }

}

/**
 * Representation for ARRAY1, ARRAY2 and ARRAY3 instructions.
 */

class CLArrayInstruction extends CLInstruction {

    /**
     * A number identifying the type of primitive array, or an index into the
     * constant pool, the item at which specifies the reference type of the
     * array.
     */
    private int type;

    /** Number of dimensions in case of a multi-dimensional array. */
    private int dim;

    /**
     * Construct a CLArrayInstruction object for ARRAY1 instructions.
     *
     * @param opcode
     *            the opcode for this instruction.
     * @param pc
     *            index of this instruction within the code array of a method.
     * @param type
```

```
619      *            number identifying the type.
620      */
621
622     public CLArrayInstruction(int opcode, int pc, int type) {
623         super.opcode = opcode;
624         super.pc = pc;
625         mnemonic = instructionInfo[opcode].mnemonic;
626         operandCount = instructionInfo[opcode].operandCount;
627         stackUnits = instructionInfo[opcode].stackUnits;
628         localVariableIndex = instructionInfo[opcode].localVariableIndex;
629         this.type = type;
630     }
631
632     /**
633      * Construct a CLArrayInstruction object for ARRAY2 instructions.
634      *
635      * @param opcode
636      *            the opcode for this instruction.
637      * @param pc
638      *            index of this instruction within the code array of a method.
639      * @param type
640      *            number identifying the type.
641      * @param dim
642      *            number of dimensions.
643      */
644
645     public CLArrayInstruction(int opcode, int pc, int type, int dim) {
646         super.opcode = opcode;
647         super.pc = pc;
648         mnemonic = instructionInfo[opcode].mnemonic;
649         operandCount = instructionInfo[opcode].operandCount;
650         stackUnits = instructionInfo[opcode].stackUnits;
651         localVariableIndex = instructionInfo[opcode].localVariableIndex;
652         this.type = type;
653         this.dim = dim;
654     }
655
656     /**
657      * Construct a CLArrayInstruction object for ARRAY3 instructions.
658      *
659      * @param opcode
660      *            the opcode for this instruction.
661      * @param pc
662      *            index of this instruction within the code array of a method.
663      */
664
665     public CLArrayInstruction(int opcode, int pc) {
666         super.opcode = opcode;
667         super.pc = pc;
668         mnemonic = instructionInfo[opcode].mnemonic;
669         operandCount = instructionInfo[opcode].operandCount;
670         stackUnits = instructionInfo[opcode].stackUnits;
671         localVariableIndex = instructionInfo[opcode].localVariableIndex;
672     }
673
674     /**
675      * @inheritDoc
676      */
677
678     public ArrayList<Integer> toBytes() {
679         ArrayList<Integer> bytes = new ArrayList<Integer>();
680         bytes.add(opcode);
681         switch (opcode) {
682         case NEWARRAY:
683             bytes.add(byteAt(type, 1));
684             break;
685         case ANEWARRAY:
686             bytes.add(byteAt(type, 2));
687             bytes.add(byteAt(type, 1));
```

```
            break;
        case MULTIANEWARRAY:
            bytes.add(byteAt(type, 2));
            bytes.add(byteAt(type, 1));
            bytes.add(byteAt(dim, 1));
            break;
        }
        return bytes;
    }

}

/**
 * Representation for ARITHMETIC1 and ARITHMETIC2 instructions.
 */

class CLArithmeticInstruction extends CLInstruction {

    /**
     * Whether this instruction is preceeded by a WIDE instruction; applies only
     * to IINC.
     */
    private boolean isWidened;

    /** Increment value for IINC instruction. */
    private int constVal;

    /**
     * Construct a CLArithmeticInstruction object for ARITHMETIC1 instructions.
     *
     * @param opcode
     *            the opcode for this instruction.
     * @param pc
     *            index of this instruction within the code array of a method.
     */

    public CLArithmeticInstruction(int opcode, int pc) {
        super.opcode = opcode;
        super.pc = pc;
        mnemonic = instructionInfo[opcode].mnemonic;
        operandCount = instructionInfo[opcode].operandCount;
        stackUnits = instructionInfo[opcode].stackUnits;
        localVariableIndex = instructionInfo[opcode].localVariableIndex;
    }

    /**
     * Construct a CLArithmeticInstruction object for IINC instruction.
     *
     * @param opcode
     *            the opcode for this instruction.
     * @param pc
     *            index of this instruction within the code array of a method.
     * @param localVariableIndex
     *            index of the local variable to increment.
     * @param constVal
     *            increment value.
     * @param isWidened
     *            whether this instruction is preceeded by the WIDE (widening)
     *            instruction.
     */

    public CLArithmeticInstruction(int opcode, int pc, int localVariableIndex,
            int constVal, boolean isWidened) {
        super.opcode = opcode;
        super.pc = pc;
        super.localVariableIndex = localVariableIndex;
        mnemonic = instructionInfo[opcode].mnemonic;
        operandCount = instructionInfo[opcode].operandCount;
        stackUnits = instructionInfo[opcode].stackUnits;
```

```
757         this.constVal = constVal;
758         this.isWidened = isWidened;
759     }
760 
761     /**
762      * @inheritDoc
763      */
764 
765     public ArrayList<Integer> toBytes() {
766         ArrayList<Integer> bytes = new ArrayList<Integer>();
767         bytes.add(opcode);
768         if (opcode == IINC) {
769             if (isWidened) {
770                 bytes.add(byteAt(localVariableIndex, 2));
771                 bytes.add(byteAt(localVariableIndex, 1));
772                 bytes.add(byteAt(constVal, 2));
773                 bytes.add(byteAt(constVal, 1));
774             } else {
775                 bytes.add(byteAt(localVariableIndex, 1));
776                 bytes.add(byteAt(constVal, 1));
777             }
778         }
779         return bytes;
780     }
781 
782 }
783 
784 /**
785  * Representation for BIT instructions.
786  */
787 
788 class CLBitInstruction extends CLInstruction {
789 
790     /**
791      * Constructs a CLBitInstruction object.
792      *
793      * @param opcode
794      *            the opcode for this instruction.
795      * @param pc
796      *            index of this instruction within the code array of a method.
797      */
798 
799     public CLBitInstruction(int opcode, int pc) {
800         super.opcode = opcode;
801         super.pc = pc;
802         mnemonic = instructionInfo[opcode].mnemonic;
803         operandCount = instructionInfo[opcode].operandCount;
804         stackUnits = instructionInfo[opcode].stackUnits;
805         localVariableIndex = instructionInfo[opcode].localVariableIndex;
806     }
807 
808     /**
809      * @inheritDoc
810      */
811 
812     public ArrayList<Integer> toBytes() {
813         ArrayList<Integer> bytes = new ArrayList<Integer>();
814         bytes.add(opcode);
815         return bytes;
816     }
817 
818 }
819 
820 /**
821  * Representation for COMPARISON instructions.
822  */
823 
824 class CLComparisonInstruction extends CLInstruction {
825 
```

```
826     /**
827      * Construct a CLComparisonInstruction object.
828      *
829      * @param opcode
830      *            the opcode for this instruction.
831      * @param pc
832      *            index of this instruction within the code array of a method.
833      */
834
835     public CLComparisonInstruction(int opcode, int pc) {
836         super.opcode = opcode;
837         super.pc = pc;
838         mnemonic = instructionInfo[opcode].mnemonic;
839         operandCount = instructionInfo[opcode].operandCount;
840         stackUnits = instructionInfo[opcode].stackUnits;
841         localVariableIndex = instructionInfo[opcode].localVariableIndex;
842     }
843
844     /**
845      * @inheritDoc
846      */
847
848     public ArrayList<Integer> toBytes() {
849         ArrayList<Integer> bytes = new ArrayList<Integer>();
850         bytes.add(opcode);
851         return bytes;
852     }
853
854 }
855
856 /**
857  * Representation for CONVERSION instructions.
858  */
859
860 class CLConversionInstruction extends CLInstruction {
861
862     /**
863      * Construct a CLConversionInstruction object.
864      *
865      * @param opcode
866      *            the opcode for this instruction.
867      * @param pc
868      *            index of this instruction within the code array of a method.
869      */
870
871     public CLConversionInstruction(int opcode, int pc) {
872         super.opcode = opcode;
873         super.pc = pc;
874         mnemonic = instructionInfo[opcode].mnemonic;
875         operandCount = instructionInfo[opcode].operandCount;
876         stackUnits = instructionInfo[opcode].stackUnits;
877         localVariableIndex = instructionInfo[opcode].localVariableIndex;
878     }
879
880     /**
881      * @inheritDoc
882      */
883
884     public ArrayList<Integer> toBytes() {
885         ArrayList<Integer> bytes = new ArrayList<Integer>();
886         bytes.add(opcode);
887         return bytes;
888     }
889
890 }
891
892 /**
893  * Representation for FLOW_CONTROL1, FLOW_CONTROL2, FLOW_CONTROL3 and
894  * FLOW_CONTROL4 instructions.
```

```
 */

class CLFlowControlInstruction extends CLInstruction {

    /**
     * Jump label; this flow control instruction will jump to an instruction
     * after this label.
     */
    private String jumpToLabel;

    /** jumpLabel is resolved to this offset. */
    private int jumpToOffset;

    /**
     * Index of the local variable containing the return address; applies only
     * to RET instruction.
     */
    private int index;

    /**
     * Whether this instruction is preceeded by a WIDE instruction; applies only
     * to RET instruction.
     */
    private boolean isWidened;

    /**
     * These many (0-3) bytes are added before default offset so that the index
     * of the offset is divisible by 4.
     */
    private int padd;

    /**
     * Jump label for default value for TABLESWITCH and LOOKUPSWITCH
     * instructions.
     */
    private String defaultLabel;

    /** defaultLabel is resolved to this offset. */
    private int defaultOffset;

    /**
     * Number of pairs in the match table for LOOKUPSWITCH instruction.
     */
    private int numPairs;

    /** Key and label table for LOOKUPSWITCH instruction. */
    private TreeMap<Integer, String> matchLabelPairs;

    /**
     * Key and offset (resolved labels from matchLabelPairs) table for
     * LOOKUPSWITCH instruction.
     */
    private TreeMap<Integer, Integer> matchOffsetPairs;

    /** Smallest value of index for TABLESWITCH instruction. */
    private int low;

    /** Highest value of index for TABLESWITCH instruction. */
    private int high;

    /**
     * List of jump labels for TABLESWITCH instruction for each index value from
     * low to high, end values included.
     */
    private ArrayList<String> labels;

    /**
     * List of offsets (resolved labels from labels) for TABLESWITCH
     * instruction.
```

```
964      */
965     private ArrayList<Integer> offsets;
966
967     /**
968      * Construct a CLFlowControlInstruction object for FLOW_CONTROL1
969      * instructions.
970      *
971      * @param opcode
972      *             the opcode for this instruction.
973      * @param pc
974      *             index of this instruction within the code array of a method.
975      * @param jumpToLabel
976      *             the label to jump to.
977      */
978
979     public CLFlowControlInstruction(int opcode, int pc, String jumpToLabel) {
980         super.opcode = opcode;
981         super.pc = pc;
982         mnemonic = instructionInfo[opcode].mnemonic;
983         operandCount = instructionInfo[opcode].operandCount;
984         stackUnits = instructionInfo[opcode].stackUnits;
985         localVariableIndex = instructionInfo[opcode].localVariableIndex;
986         this.jumpToLabel = jumpToLabel;
987     }
988
989     /**
990      * Construct a CLFlowControlInstruction object for RET instruction.
991      *
992      * @param pc
993      *             index of this instruction within the code array of a method.
994      * @param index
995      *             index of the local variable containing the return address.
996      * @param isWidened
997      *             whether this instruction is preceeded by the WIDE (widening)
998      *             instruction.
999      */
1000
1001    public CLFlowControlInstruction(int pc, int index, boolean isWidened) {
1002        super.opcode = RET;
1003        super.pc = pc;
1004        mnemonic = instructionInfo[opcode].mnemonic;
1005        operandCount = instructionInfo[opcode].operandCount;
1006        stackUnits = instructionInfo[opcode].stackUnits;
1007        localVariableIndex = instructionInfo[opcode].localVariableIndex;
1008        this.index = index;
1009        this.isWidened = isWidened;
1010    }
1011
1012    /**
1013     * Construct a CLFlowControlInstruction object for TABLESWITCH instruction.
1014     *
1015     * @param opcode
1016     *             the opcode for this instruction.
1017     * @param pc
1018     *             index of this instruction within the code array of a method.
1019     * @param defaultLabel
1020     *             jump label for default value.
1021     * @param low
1022     *             smallest value of index.
1023     * @param high
1024     *             highest value of index.
1025     * @param labels
1026     *             list of jump labels for each index value from low to high, end
1027     *             values included.
1028     */
1029
1030    public CLFlowControlInstruction(int opcode, int pc, String defaultLabel,
1031            int low, int high, ArrayList<String> labels) {
1032        super.opcode = opcode;
```

```
        super.pc = pc;
        mnemonic = instructionInfo[opcode].mnemonic;
        stackUnits = instructionInfo[opcode].stackUnits;
        localVariableIndex = instructionInfo[opcode].localVariableIndex;
        this.defaultLabel = defaultLabel;
        this.low = low;
        this.high = high;
        this.labels = labels;
        pad = 4 - ((pc + 1) % 4);
        operandCount = pad + 12 + 4 * labels.size();
    }

    /**
     * Construct a CLFlowControlInstruction object for LOOKUPSWITCH instruction.
     *
     * @param opcode
     *            the opcode for this instruction.
     * @param pc
     *            index of this instruction within the code array of a method.
     * @param defaultLabel
     *            jump label for default value.
     * @param numPairs
     *            number of pairs in the match table.
     * @param matchLabelPairs
     *            key match table.
     */

    public CLFlowControlInstruction(int opcode, int pc, String defaultLabel,
            int numPairs, TreeMap<Integer, String> matchLabelPairs) {
        super.opcode = opcode;
        super.pc = pc;
        mnemonic = instructionInfo[opcode].mnemonic;
        stackUnits = instructionInfo[opcode].stackUnits;
        localVariableIndex = instructionInfo[opcode].localVariableIndex;
        this.defaultLabel = defaultLabel;
        this.numPairs = numPairs;
        this.matchLabelPairs = matchLabelPairs;
        pad = 4 - ((pc + 1) % 4);
        operandCount = pad + 8 + 8 * numPairs;
    }

    /**
     * Resolve the jump labels to the corresponding offset values using the
     * given label to pc mapping. If unable to resolve a label, the offset is
     * set such that the next instruction will be executed.
     *
     * @param labelToPC
     *            label to pc mapping.
     * @return true if all labels were resolved successfully; false otherwise.
     */

    public boolean resolveLabels(Hashtable<String, Integer> labelToPC) {
        boolean allLabelsResolved = true;
        if (instructionInfo[opcode].category == FLOW_CONTROL1) {
            if (labelToPC.containsKey(jumpToLabel)) {
                jumpToOffset = labelToPC.get(jumpToLabel) - pc;
            } else {
                jumpToOffset = operandCount;
                allLabelsResolved = false;
            }
        } else if (opcode == LOOKUPSWITCH) {
            if (labelToPC.containsKey(defaultLabel)) {
                defaultOffset = labelToPC.get(defaultLabel) - pc;
            } else {
                defaultOffset = operandCount;
                allLabelsResolved = false;
            }
            matchOffsetPairs = new TreeMap<Integer, Integer>();
            Set<Entry<Integer, String>> matches = matchLabelPairs.entrySet();
```

```
            Iterator<Entry<Integer, String>> iter = matches.iterator();
            while (iter.hasNext()) {
                Entry<Integer, String> entry = iter.next();
                int match = entry.getKey();
                String label = entry.getValue();
                if (labelToPC.containsKey(label)) {
                    matchOffsetPairs.put(match, labelToPC.get(label) - pc);
                } else {
                    matchOffsetPairs.put(match, operandCount);
                    allLabelsResolved = false;
                }
            }
        } else if (opcode == TABLESWITCH) {
            if (labelToPC.containsKey(defaultLabel)) {
                defaultOffset = labelToPC.get(defaultLabel) - pc;
            } else {
                defaultOffset = operandCount;
                allLabelsResolved = false;
            }
            offsets = new ArrayList<Integer>();
            for (int i = 0; i < labels.size(); i++) {
                if (labelToPC.containsKey(labels.get(i))) {
                    offsets.add(labelToPC.get(labels.get(i)) - pc);
                } else {
                    offsets.add(operandCount);
                    allLabelsResolved = false;
                }
            }
        }
        return allLabelsResolved;
    }

    /**
     * Return the pc of instruction to jump to.
     *
     * @return pc to jump to.
     */

    public int jumpToOffset() {
        return jumpToOffset;
    }

    /**
     * @inheritDoc
     */

    public ArrayList<Integer> toBytes() {
        ArrayList<Integer> bytes = new ArrayList<Integer>();
        bytes.add(opcode);
        switch (opcode) {
        case RET:
            if (isWidened) {
                bytes.add(byteAt(index, 2));
                bytes.add(byteAt(index, 1));
            } else {
                bytes.add(byteAt(index, 1));
            }
            break;
        case TABLESWITCH:
            for (int i = 0; i < pad; i++) {
                bytes.add(0);
            }
            bytes.add(byteAt(defaultOffset, 4));
            bytes.add(byteAt(defaultOffset, 3));
            bytes.add(byteAt(defaultOffset, 2));
            bytes.add(byteAt(defaultOffset, 1));
            bytes.add(byteAt(low, 4));
            bytes.add(byteAt(low, 3));
            bytes.add(byteAt(low, 2));
```

```
            bytes.add(byteAt(low, 1));
            bytes.add(byteAt(high, 4));
            bytes.add(byteAt(high, 3));
            bytes.add(byteAt(high, 2));
            bytes.add(byteAt(high, 1));
            for (int i = 0; i < offsets.size(); i++) {
                int jumpOffset = offsets.get(i);
                bytes.add(byteAt(jumpOffset, 4));
                bytes.add(byteAt(jumpOffset, 3));
                bytes.add(byteAt(jumpOffset, 2));
                bytes.add(byteAt(jumpOffset, 1));
            }
            break;
        case LOOKUPSWITCH:
            for (int i = 0; i < pad; i++) {
                bytes.add(0);
            }
            bytes.add(byteAt(defaultOffset, 4));
            bytes.add(byteAt(defaultOffset, 3));
            bytes.add(byteAt(defaultOffset, 2));
            bytes.add(byteAt(defaultOffset, 1));
            bytes.add(byteAt(numPairs, 4));
            bytes.add(byteAt(numPairs, 3));
            bytes.add(byteAt(numPairs, 2));
            bytes.add(byteAt(numPairs, 1));
            Set<Entry<Integer, Integer>> matches = matchOffsetPairs.entrySet();
            Iterator<Entry<Integer, Integer>> iter = matches.iterator();
            while (iter.hasNext()) {
                Entry<Integer, Integer> entry = iter.next();
                int match = entry.getKey();
                int offset = entry.getValue();
                bytes.add(byteAt(match, 4));
                bytes.add(byteAt(match, 3));
                bytes.add(byteAt(match, 2));
                bytes.add(byteAt(match, 1));
                bytes.add(byteAt(offset, 4));
                bytes.add(byteAt(offset, 3));
                bytes.add(byteAt(offset, 2));
                bytes.add(byteAt(offset, 1));
            }
            break;
        case GOTO_W:
        case JSR_W:
            bytes.add(byteAt(jumpToOffset, 4));
            bytes.add(byteAt(jumpToOffset, 3));
            bytes.add(byteAt(jumpToOffset, 2));
            bytes.add(byteAt(jumpToOffset, 1));
            break;
        default:
            bytes.add(byteAt(jumpToOffset, 2));
            bytes.add(byteAt(jumpToOffset, 1));
        }
        return bytes;
    }

}

/**
 * Representation for LOAD_STORE1, LOAD_STORE2, LOAD_STORE3 and LOAD_STORE4
 * instructions.
 */

class CLLoadStoreInstruction extends CLInstruction {

    /**
     * Whether this instruction is preceeded by a WIDE instruction; applies only
     * to ILOAD, LLOAD, FLOAD, DLOAD, ALOAD, ISTORE, LSTORE, FSTORE, DSTORE,
     * ASTORE.
     */
```

```
1240    private boolean isWidened;
1241
1242    /**
1243     * A byte (for BIPUSH), a short (for SIPUSH), or a constant pool index for
1244     * LDC, LDC_W, LDC2_W instructions.
1245     */
1246    private int constVal;
1247
1248    /**
1249     * Construct a CLLoadStoreInstruction object for LOAD_STORE1 instructions.
1250     *
1251     * @param opcode
1252     *            the opcode for this instruction.
1253     * @param pc
1254     *            index of this instruction within the code array of a method.
1255     */
1256
1257    public CLLoadStoreInstruction(int opcode, int pc) {
1258        super.opcode = opcode;
1259        super.pc = pc;
1260        mnemonic = instructionInfo[opcode].mnemonic;
1261        operandCount = instructionInfo[opcode].operandCount;
1262        stackUnits = instructionInfo[opcode].stackUnits;
1263        localVariableIndex = instructionInfo[opcode].localVariableIndex;
1264    }
1265
1266    /**
1267     * Construct a CLLoadStoreInstruction object for LOAD_STORE2 instructions.
1268     *
1269     * @param opcode
1270     *            the opcode for this instruction.
1271     * @param pc
1272     *            index of this instruction within the code array of a method.
1273     * @param localVariableIndex
1274     *            index of the local variable to increment.
1275     * @param isWidened
1276     *            whether this instruction is preceeded by the WIDE (widening)
1277     *            instruction.
1278     */
1279
1280    public CLLoadStoreInstruction(int opcode, int pc, int localVariableIndex,
1281            boolean isWidened) {
1282        super.opcode = opcode;
1283        super.pc = pc;
1284        mnemonic = instructionInfo[opcode].mnemonic;
1285        operandCount = instructionInfo[opcode].operandCount;
1286        stackUnits = instructionInfo[opcode].stackUnits;
1287        super.localVariableIndex = localVariableIndex;
1288        this.isWidened = isWidened;
1289    }
1290
1291    /**
1292     * Construct a CLLoadStoreInstruction object for LOAD_STORE3 and LOAD_STORE4
1293     * instructions.
1294     *
1295     * @param opcode
1296     *            the opcode for this instruction.
1297     * @param pc
1298     *            index of this instruction within the code array of a method.
1299     * @param constVal
1300     *            a byte (for BIPUSH), a short (for SIPUSH), or a constant pool
1301     *            index for LDC instructions.
1302     */
1303
1304    public CLLoadStoreInstruction(int opcode, int pc, int constVal) {
1305        super.opcode = opcode;
1306        super.pc = pc;
1307        mnemonic = instructionInfo[opcode].mnemonic;
1308        operandCount = instructionInfo[opcode].operandCount;
```

```
1309        stackUnits = instructionInfo[opcode].stackUnits;
1310        localVariableIndex = instructionInfo[opcode].localVariableIndex;
1311        this.constVal = constVal;
1312    }
1313
1314    /**
1315     * @inheritDoc
1316     */
1317
1318    public ArrayList<Integer> toBytes() {
1319        ArrayList<Integer> bytes = new ArrayList<Integer>();
1320        bytes.add(opcode);
1321        if (instructionInfo[opcode].operandCount > 0) {
1322            if (localVariableIndex != IRRELEVANT) {
1323                if (isWidened) {
1324                    bytes.add(byteAt(localVariableIndex, 2));
1325                }
1326                bytes.add(byteAt(localVariableIndex, 1));
1327            } else {
1328                switch (opcode) {
1329                case BIPUSH:
1330                case LDC:
1331                    bytes.add(byteAt(constVal, 1));
1332                    break;
1333                case SIPUSH:
1334                case LDC_W:
1335                case LDC2_W:
1336                    bytes.add(byteAt(constVal, 2));
1337                    bytes.add(byteAt(constVal, 1));
1338                }
1339            }
1340        }
1341        return bytes;
1342    }
1343
1344}
1345
1346/**
1347 * Representation for STACK instructions
1348 */
1349
1350class CLStackInstruction extends CLInstruction {
1351
1352    /**
1353     * Construct a CLStackInstruction object.
1354     *
1355     * @param opcode
1356     *            the opcode for this instruction.
1357     * @param pc
1358     *            index of this instruction within the code array of a method.
1359     */
1360
1361    public CLStackInstruction(int opcode, int pc) {
1362        super.opcode = opcode;
1363        super.pc = pc;
1364        mnemonic = instructionInfo[opcode].mnemonic;
1365        operandCount = instructionInfo[opcode].operandCount;
1366        stackUnits = instructionInfo[opcode].stackUnits;
1367        localVariableIndex = instructionInfo[opcode].localVariableIndex;
1368    }
1369
1370    /**
1371     * @inheritDoc
1372     */
1373
1374    public ArrayList<Integer> toBytes() {
1375        ArrayList<Integer> bytes = new ArrayList<Integer>();
1376        bytes.add(opcode);
1377        return bytes;
```

```
1378    }
1379
1380}
1381
1382/**
1383 * Representation for MISC instructions.
1384 */
1385
1386class CLMiscInstruction extends CLInstruction {
1387
1388    /**
1389     * Construct a CLMiscInstruction object.
1390     *
1391     * @param opcode
1392     *            the opcode for this instruction.
1393     * @param pc
1394     *            index of this instruction within the code array of a method.
1395     */
1396
1397    public CLMiscInstruction(int opcode, int pc) {
1398        super.opcode = opcode;
1399        super.pc = pc;
1400        mnemonic = instructionInfo[opcode].mnemonic;
1401        operandCount = instructionInfo[opcode].operandCount;
1402        stackUnits = instructionInfo[opcode].stackUnits;
1403        localVariableIndex = instructionInfo[opcode].localVariableIndex;
1404    }
1405
1406    /**
1407     * @inheritDoc
1408     */
1409
1410    public ArrayList<Integer> toBytes() {
1411        ArrayList<Integer> bytes = new ArrayList<Integer>();
1412        bytes.add(opcode);
1413        return bytes;
1414    }
1415
1416}
1417
1418/**
1419 * This class stores static information about a JVM instruction.
1420 */
1421
1422class CLInsInfo {
1423
1424    /** Opcode for this instruction. */
1425    public int opcode;
1426
1427    /** Mnemonic for this instruction. */
1428    public String mnemonic;
1429
1430    /** Number of operands for this instruction. */
1431    public int operandCount;
1432
1433    /**
1434     * Words produced - words consumed from the operand stack by this
1435     * instruction.
1436     */
1437    public int stackUnits;
1438
1439    /**
1440     * Index of the local variable that this instruction refers to; applies only
1441     * to instructions that operate on local variables.
1442     */
1443    public int localVariableIndex;
1444
1445    /** The category under which instruction belongs. */
1446    public Category category;
```

```
1447
1448    /**
1449     * Construct a CLInsInfo object.
1450     *
1451     * @param opcode
1452     *            opcode for this instruction.
1453     * @param mnemonic
1454     *            name for this instruction.
1455     * @param operandCount
1456     *            number of operands for this instruction.
1457     * @param localVariableIndex
1458     *            index of the local variable that this instruction refers to.
1459     * @param stackUnits
1460     *            words produced - words consumed from the operand stack by this
1461     *            instruction.
1462     * @param category
1463     *            category under which this instruction belogs.
1464     */
1465
1466    public CLInsInfo(int opcode, String mnemonic, int operandCount,
1467            int localVariableIndex, int stackUnits, Category category) {
1468        this.opcode = opcode;
1469        this.mnemonic = mnemonic;
1470        this.operandCount = operandCount;
1471        this.localVariableIndex = localVariableIndex;
1472        this.stackUnits = stackUnits;
1473        this.category = category;
1474    }
1475
1476}
1477
```