```
1  // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import java.io.BufferedOutputStream;
6  import java.io.ByteArrayOutputStream;
7  import java.io.File;
8  import java.io.FileNotFoundException;
9  import java.io.FileOutputStream;
10 import java.io.IOException;
11 import java.io.DataOutputStream;
12 import java.io.OutputStream;
13 import java.util.ArrayList;
14 import java.util.Hashtable;
15 import java.util.Stack;
16 import java.util.StringTokenizer;
17 import java.util.TreeMap;
18 import static jminusminus.CLConstants.*;
19 import static jminusminus.CLConstants.Category.*;
20
21 /**
22  * This class provides a high level interface for creating (in-memory and file
23  * based) representation of Java classes.
24  *
25  * j-- uses this interface to produce target JVM bytecode from a j-- source
26  * program. During the pre-analysis and analysis phases, j-- produces partial
27  * (in-memory) classes for the type declarations within the compilation unit,
28  * and during the code generation phase, it produces file-based classes for the
29  * declarations.
30  */
31
32 public class CLEmitter {
33
34     /** Name of the class. */
35     private String name;
36
37     /**
38      * If true, the in-memory representation of the class will be written to the
39      * file system. Otherwise, it won't be saved as a file.
40      */
41     private boolean toFile;
42
43     /** Destination directory for the class. */
44     private String destDir;
45
46     /** In-memory representation of the class. */
47     private CLFile clFile;
48
49     /** Constant pool of the class. */
50     private CLConstantPool constantPool;
51
52     /** Direct super interfaces of the class. */
53     private ArrayList<Integer> interfaces;
54
55     /** Fields in the class. */
56     private ArrayList<CLFieldInfo> fields;
57
58     /** Attributes of the field last added. */
59     private ArrayList<CLAttributeInfo> fAttributes;
60
61     /** Methods in the class. */
62     private ArrayList<CLMethodInfo> methods;
63
64     /** Attributes of the method last added. */
65     private ArrayList<CLAttributeInfo> mAttributes;
66
```

```
67      /** Attributes of the class. */
68      private ArrayList<CLAttributeInfo> attributes;
69
70      /** Inner classes of the class. */
71      private ArrayList<CLInnerClassInfo> innerClasses;
72
73      /** Code (instruction) section of the method last added. */
74      private ArrayList<CLInstruction> mCode;
75
76      /**
77       * Table containing exception handlers in the method last added.
78       */
79      private ArrayList<CLException> mExceptionHandlers;
80
81      /** Access flags of the method last added. */
82      private int mAccessFlags;
83
84      /**
85       * Index into the constant pool, the item at which specifies the name of the
86       * method last added.
87       */
88      private int mNameIndex;
89
90      /**
91       * Index into the constant pool, the item at which specifies the descriptor
92       * of the method last added.
93       */
94      private int mDescriptorIndex;
95
96      /** Number of arguments for the method last added. */
97      private int mArgumentCount;
98
99      /** Code attributes of the method last added. */
100     private ArrayList<CLAttributeInfo> mCodeAttributes;
101
102     /** Whether the method last added needs closing. */
103     private boolean isMethodOpen;
104
105     /**
106      * Stores jump labels for the method last added. When a label is created, a
107      * mapping from the label to an Integer representing the pc of the next
108      * instruction is created. Later on, when the label is added, its Integer
109      * value is replaced by the value of pc then.
110      */
111     private Hashtable<String, Integer> mLabels;
112
113     /** Counter for creating unique jump labels. */
114     private int mLabelCount;
115
116     /**
117      * Whether there there was an instruction added after the last call to
118      * addLabel( String label ). If not, the branch instruction that was added
119      * with that label would jump beyond the code section, which is not
120      * acceptable to the runtime class loader. Therefore, if this flag is false,
121      * we add a NOP instruction at the end of the code section to make the jump
122      * valid.
123      */
124     private boolean mInstructionAfterLabel = false;
125
126     /**
127      * Location counter; index of the next instruction within the code section
128      * of the method last added.
129      */
130     private int mPC;
131
132     /** Name of the method last added; used for error reporting. */
133     private String eCurrentMethod;
134
135     /**
```

```
136      * Whether an error occurred while creating/writing the class.
137      */
138     private boolean errorHasOccurred;
139
140     /**
141      * Class loader to use for creating in-memory representation of classes from
142      * byte streams.
143      */
144     private static ByteClassLoader byteClassLoader;
145
146     /**
147      * Initialize all variables used for adding a method to the ClassFile
148      * structure to their appropriate values.
149      */
150
151     private void initializeMethodVariables() {
152         mAccessFlags = 0;
153         mNameIndex = -1;
154         mDescriptorIndex = -1;
155         mArgumentCount = 0;
156         mPC = 0;
157         mAttributes = new ArrayList<CLAttributeInfo>();
158         mExceptionHandlers = new ArrayList<CLException>();
159         mCode = new ArrayList<CLInstruction>();
160         mCodeAttributes = new ArrayList<CLAttributeInfo>();
161         mLabels = new Hashtable<String, Integer>();
162         mLabelCount = 1;
163         mInstructionAfterLabel = false;
164     }
165
166     /**
167      * Add the method created using addMethod() to the ClassFile structure. This
168      * involves adding an instance of CLMethodInfo to ClassFile.methods for the
169      * method.
170      */
171
172     private void endOpenMethodIfAny() {
173         if (isMethodOpen) {
174             isMethodOpen = false;
175             if (!mInstructionAfterLabel) {
176                 // Must jump to an instruction
177                 addNoArgInstruction(NOP);
178             }
179
180             // Resolve jump labels in exception handlers
181             ArrayList<CLExceptionInfo> exceptionTable = new
ArrayList<CLExceptionInfo>();
182             for (int i = 0; i < mExceptionHandlers.size(); i++) {
183                 CLException e = mExceptionHandlers.get(i);
184                 if (!e.resolveLabels(mLabels)) {
185                     reportEmitterError(
186                             "%s: Unable to resolve exception handler "
187                                     + "label(s)", eCurrentMethod);
188                 }
189
190                 // We allow catchType to be null (mapping to
191                 // index 0),
192                 // implying this exception handler is called for
193                 // all
194                 // exceptions. This is used to implement
195                 // "finally"
196                 int catchTypeIndex = (e.catchType == null) ? 0 : constantPool
197                         .constantClassInfo(e.catchType);
198                 CLExceptionInfo c = new CLExceptionInfo(e.startPC, e.endPC,
199                         e.handlerPC, catchTypeIndex);
200                 exceptionTable.add(c);
201             }
202
203             // Convert Instruction objects to bytes
```

```
204             ArrayList<Integer> byteCode = new ArrayList<Integer>();
205             int maxLocals = mArgumentCount;
206             for (int i = 0; i < mCode.size(); i++) {
207                 CLInstruction instr = mCode.get(i);
208 
209                 // Compute maxLocals
210                 int localVariableIndex = instr.localVariableIndex();
211                 switch (instr.opcode()) {
212                 case LLOAD:
213                 case LSTORE:
214                 case DSTORE:
215                 case DLOAD:
216                 case LLOAD_0:
217                 case LLOAD_1:
218                 case LLOAD_2:
219                 case LLOAD_3:
220                 case LSTORE_0:
221                 case LSTORE_1:
222                 case LSTORE_2:
223                 case LSTORE_3:
224                 case DLOAD_0:
225                 case DLOAD_1:
226                 case DLOAD_2:
227                 case DLOAD_3:
228                 case DSTORE_0:
229                 case DSTORE_1:
230                 case DSTORE_2:
231                 case DSTORE_3:
232                     // Each long and double occupies two slots in
233                     // the
234                     // local variable table
235                     localVariableIndex++;
236                 }
237                 maxLocals = Math.max(maxLocals, localVariableIndex + 1);
238 
239                 // Resolve jump labels in flow control
240                 // instructions
241                 if (instr instanceof CLFlowControlInstruction) {
242                     if (((CLFlowControlInstruction) instr)
243                             .resolveLabels(mLabels)) {
244                         reportEmitterError(
245                                 "%s: Unable to resolve jump label(s)",
246                                 eCurrentMethod);
247                     }
248                 }
249 
250                 byteCode.addAll(instr.toBytes());
251             }
252 
253             // Code attribute; add only if method is neither
254             // native
255             // nor abstract
256             if (!((mAccessFlags & ACC_NATIVE) == ACC_NATIVE || (mAccessFlags &
ACC_ABSTRACT) == ACC_ABSTRACT)) {
257                 addMethodAttribute(codeAttribute(byteCode, exceptionTable,
258                         stackDepth(), maxLocals));
259             }
260 
261             methods.add(new CLMethodInfo(mAccessFlags, mNameIndex,
262                     mDescriptorIndex, mAttributes.size(), mAttributes));
263         }
264 
265         // This method could be the last method, so we need
266         // the following wrap up code
267 
268         // Add the InnerClass attribute if this class has inner
269         // classes
270         if (innerClasses.size() > 0) {
271             addClassAttribute(innerClassesAttribute());
```

```
272         }
273
274         // Set the members of the ClassFile structure to their
275         // appropriate values
276         clFile.constantPoolCount = constantPool.size() + 1;
277         clFile.constantPool = constantPool;
278         clFile.interfacesCount = interfaces.size();
279         clFile.interfaces = interfaces;
280         clFile.fieldsCount = fields.size();
281         clFile.fields = fields;
282         clFile.methodsCount = methods.size();
283         clFile.methods = methods;
284         clFile.attributesCount = attributes.size();
285         clFile.attributes = attributes;
286     }
287
288     /**
289      * Add a field.
290      *
291      * @param accessFlags
292      *            access flags for the field.
293      * @param name
294      *            name of the field.
295      * @param type
296      *            type descriptor for the field.
297      * @param isSynthetic
298      *            is this a synthetic field?
299      * @param c
300      *            index into the constant pool at which there is a
301      *            ConstantIntegerInfo, ConstantFloatInfo, ConstantLongInfo,
302      *            ConstantDoubleInfo or ConstantStringInfo object. c is -1 if
303      *            the field does not have an initialization.
304      */
305
306     private void addFieldInfo(ArrayList<String> accessFlags, String name,
307             String type, boolean isSynthetic, int c) {
308         if (!validTypeDescriptor(type)) {
309             reportEmitterError("'%s' is not a valid type descriptor for field",
310                     type);
311         }
312         int flags = 0;
313         int nameIndex = constantPool.constantUtf8Info(name);
314         int descriptorIndex = constantPool.constantUtf8Info(type);
315         fAttributes = new ArrayList<CLAttributeInfo>();
316         if (accessFlags != null) {
317             for (int i = 0; i < accessFlags.size(); i++) {
318                 flags |= CLFile.accessFlagToInt(accessFlags.get(i));
319             }
320         }
321         if (isSynthetic) {
322             addFieldAttribute(syntheticAttribute());
323         }
324         if (c != -1) {
325             addFieldAttribute(constantValueAttribute(c));
326         }
327         fields.add(new CLFieldInfo(flags, nameIndex, descriptorIndex,
328                 fAttributes.size(), fAttributes));
329     }
330
331     /**
332      * Return the number of units a type with the specified descriptor produces
333      * or consumes from the operand stack. 0 is returned if the specified
334      * descriptor is invalid.
335      *
336      * @param descriptor
337      *            a type descriptor.
338      * @return the number of units produced or consumed in/from the operand
339      *         stack.
340      */
```

```
341
342     private int typeStackResidue(String descriptor) {
343         int i = 0;
344         char c = descriptor.charAt(0);
345         switch (c) {
346         case 'B':
347         case 'C':
348         case 'I':
349         case 'F':
350         case 'L':
351         case 'S':
352         case 'Z':
353         case '[':
354             i = 1;
355             break;
356         case 'J':
357         case 'D':
358             i = 2;
359             break;
360         }
361         return i;
362     }
363
364     /**
365      * Return the difference between the number of units consumed from the
366      * operand stack by a method with the specified descriptor and the number of
367      * units produced by the method in the operand stack. 0 is returned if the
368      * descriptor is invalid.
369      *
370      * @param descriptor
371      *            a method descriptor.
372      * @return number of units consumed from the operand stack - number of units
373      *         produced in the stack.
374      */
375
376     private int methodStackResidue(String descriptor) {
377         int i = 0;
378
379         // Extract types of arguments and the return type from
380         // the method descriptor
381         String argTypes = descriptor.substring(1, descriptor.lastIndexOf(")"));
382         String returnType = descriptor
383                 .substring(descriptor.lastIndexOf(")") + 1);
384
385         // Units consumed
386         for (int j = 0; j < argTypes.length(); j++) {
387             char c = argTypes.charAt(j);
388             switch (c) {
389             case 'B':
390             case 'C':
391             case 'I':
392             case 'F':
393             case 'S':
394             case 'Z':
395                 i -= 1;
396                 break;
397             case '[':
398                 break;
399             case 'J':
400             case 'D':
401                 i -= 2;
402                 break;
403             case 'L':
404                 int k = argTypes.indexOf(";", j);
405                 j = k;
406                 i -= 1;
407                 break;
408             }
409         }
```

```
410
411         // Units produced
412         i += typeStackResidue(returnType);
413         return i;
414     }
415
416     /**
417      * Return the argument count (number of formal parameters) for the specified
418      * method. 0 is returned if the descriptor is invalid.
419      *
420      * @param descriptor
421      *            method descriptor.
422      * @return argument count for the specified method.
423      */
424
425     private int argumentCount(String descriptor) {
426         int i = 0;
427
428         // Extract types of arguments and the return type from
429         // the method descriptor
430         String argTypes = descriptor.substring(1, descriptor.lastIndexOf(")"));
431
432         // Find number of arguments
433         for (int j = 0; j < argTypes.length(); j++) {
434             char c = argTypes.charAt(j);
435             switch (c) {
436             case 'B':
437             case 'C':
438             case 'I':
439             case 'F':
440             case 'S':
441             case 'Z':
442                 i += 1;
443                 break;
444             case '[':
445                 break;
446             case 'J':
447             case 'D':
448                 i += 2;
449                 break;
450             case 'L':
451                 int k = argTypes.indexOf(";", j);
452                 j = k;
453                 i += 1;
454                 break;
455             }
456         }
457         return i;
458     }
459
460     /**
461      * Return true if the specified name is in the internal form of a fully
462      * qualified class or interface name [JVMS 4.3], false otherwise.
463      *
464      * @param name
465      *            fully qualified class name in internal form.
466      * @return true if the specified name is in the internal form of a fully
467      *         qualified class or interface name, false otherwise.
468      */
469
470     private boolean validInternalForm(String name) {
471         if ((name == null) || name.equals("") || name.startsWith("/")
472                 || name.endsWith("/")) {
473             return false;
474         }
475         StringTokenizer t = new StringTokenizer(name, "/");
476         while (t.hasMoreTokens()) {
477             String s = t.nextToken();
478             for (int i = 0; i < s.length(); i++) {
```

```
                if (i == 0) {
                    if (!Character.isJavaIdentifierStart(s.charAt(i))) {
                        return false;
                    }
                } else {
                    if (!Character.isJavaIdentifierPart(s.charAt(i))) {
                        return false;
                    }
                }
            }
        }
        return true;
    }

    /**
     * Return true if the specified descriptor is a valid type descriptor [JVMS
     * 4.4.2], false otherwise.
     *
     * @param descriptor
     *            type descriptor.
     * @return true if the specified descriptor is a valid type descriptor,
     *         false otherwise.
     */

    private boolean validTypeDescriptor(String descriptor) {
        if (descriptor != null) {
            try {
                char c = descriptor.charAt(0);
                switch (c) {
                case 'B':
                case 'C':
                case 'I':
                case 'F':
                case 'S':
                case 'Z':
                case 'J':
                case 'D':
                    return (descriptor.length() == 1);
                case 'L':
                    if (descriptor.endsWith(";")) {
                        return validInternalForm(descriptor.substring(1,
                                descriptor.length() - 1));
                    }
                    return false;
                case '[':
                    return validTypeDescriptor(descriptor.substring(1));
                }
            } catch (IndexOutOfBoundsException e) {
                return false;
            }
        }
        return false;
    }

    /**
     * Return true if the specified descriptor is a valid method descriptor
     * [JVMS 4.4.3], false otherwise.
     *
     * @param descriptor
     *            method descriptor.
     * @return true if the specified descriptor is a valid method descriptor,
     *         false otherwise.
     */

    private boolean validMethodDescriptor(String descriptor) {
        if ((descriptor != null) && (descriptor.length() > 0)) {
            try {
                // Extract types of arguments and the return type
                // from
```

```
548                 // the method descriptor
549                 String argTypes = descriptor.substring(1, descriptor
550                         .lastIndexOf(")"));
551                 String returnType = descriptor.substring(descriptor
552                         .lastIndexOf(")") + 1);
553
554                 // Validate argument type syntax
555                 if (argTypes.endsWith("[")) {
556                     return false;
557                 }
558                 for (int i = 0; i < argTypes.length(); i++) {
559                     char c = argTypes.charAt(i);
560                     switch (c) {
561                     case 'B':
562                     case 'C':
563                     case 'I':
564                     case 'F':
565                     case 'S':
566                     case 'Z':
567                     case 'J':
568                     case 'D':
569                     case '[':
570                         break;
571                     case 'L':
572                         int j = argTypes.indexOf(";", i);
573                         String s = argTypes.substring(i, j + 1);
574                         i = j;
575                         if (!validTypeDescriptor(s)) {
576                             return false;
577                         }
578                         break;
579                     default:
580                         return false;
581                     }
582                 }
583
584                 // Validate return type syntax
585                 return (returnType.equals("V") ||
586 validTypeDescriptor(returnType));
587             } catch (IndexOutOfBoundsException e) {
588                 return false;
589             }
590         }
591         return false;
592     }
593
594     /**
595      * Return the instruction with the specified pc within the code array of the
596      * current method being added.
597      *
598      * @param pc
599      *            pc of the instruction.
600      * @return the instruction with the specified pc, or null.
601      */
602
603     private CLInstruction instruction(int pc) {
604         for (int j = 0; j < mCode.size(); j++) {
605             CLInstruction i = mCode.get(j);
606             if (i.pc() == pc) {
607                 return i;
608             }
609         }
610         return null;
611     }
612
613     /**
614      * Return the index of the instruction with the specified pc, within the
615      * code array of the current method being added.
```

```
     * @param pc
     *            pc of the instruction.
     * @return index of the instruction with the specified pc.
     */

    private int instructionIndex(int pc) {
        int j = 0;
        for (; j < mCode.size(); j++) {
            CLInstruction i = mCode.get(j);
            if (i.pc() == pc) {
                return j;
            }
        }
        return j;
    }

    /**
     * Compute the maximum depth of the operand stack for the method last added,
     * and return the value.
     *
     * @return maximum depth of operand stack.
     */

    private int stackDepth() {
        CLBranchStack branchTargets = new CLBranchStack();
        for (int i = 0; i < mExceptionHandlers.size(); i++) {
            CLException e = mExceptionHandlers.get(i);
            CLInstruction h = instruction(e.handlerPC);
            if (h != null) {
                // 1 because the exception that is thrown is
                // pushed
                // on top of the operand stack
                branchTargets.push(h, 1);
            }
        }
        int stackDepth = 0, maxStackDepth = 0, c = 0;
        CLInstruction instr = (mCode.size() == 0) ? null : mCode.get(c);
        while (instr != null) {
            int opcode = instr.opcode();
            int stackUnits = instr.stackUnits();
            if (stackUnits == EMPTY_STACK) {
                stackDepth = 0;
            } else if (stackUnits == UNIT_SIZE_STACK) {
                stackDepth = 1;
            } else {
                stackDepth += stackUnits;
            }
            if (stackDepth > maxStackDepth) {
                maxStackDepth = stackDepth;
            }

            // For tracing purposes
            // System.out.println( instr.mnemonic() + ", " +
            // stackUnits + ", " +
            // stackDepth + ", " + maxStackDepth );

            if (instr instanceof CLFlowControlInstruction) {
                CLFlowControlInstruction b = (CLFlowControlInstruction) instr;
                int jumpToIndex = b.pc() + b.jumpToOffset();
                CLInstruction instrAt = null;
                switch (opcode) {
                case JSR:
                case JSR_W:
                case RET:
                    instr = null;
                    break;
                case GOTO:
                case GOTO_W:
                    instr = null;
```

```
                default:
                    instrAt = instruction(jumpToIndex);
                    if (instrAt != null) {
                        branchTargets.push(instrAt, stackDepth);
                    }
                }
            } else {
                if ((opcode == ATHROW)
                        || ((opcode >= IRETURN) && (opcode <= RETURN))) {
                    instr = null;
                }
            }
            if (instr != null) {
                c++;
                instr = (c >= mCode.size()) ? null : mCode.get(c);
            }
            if (instr == null) {
                CLBranchTarget bt = branchTargets.pop();
                if (bt != null) {
                    instr = bt.target;
                    stackDepth = bt.stackDepth;
                    c = instructionIndex(instr.pc());
                }
            }
        }
        return maxStackDepth;
    }

    /**
     * Add LDC or LDC_W if index is wider. Instruction.
     *
     * @param index
     *            index into the constant pool, the item at which is a
     *            ConstantIntegerInfo, ConstantFloatInfo or ConstantStringInfo
     *            instance. If index is wide, i.e., greater than 255, LDC_W
     *            version of the instruction is added.
     */

    private void ldcInstruction(int index) {
        CLLoadStoreInstruction instr = null;
        if (index <= 255) {
            instr = new CLLoadStoreInstruction(LDC, mPC++, index);
        } else {
            instr = new CLLoadStoreInstruction(LDC_W, mPC++, index);
        }
        mPC += instr.operandCount();
        mCode.add(instr);
        mInstructionAfterLabel = true;
    }

    /**
     * Add LDC2_W instruction -- used for long and double constants. Note that
     * only a wide-index version of LDC2_W instruction exists.
     *
     * @param index
     *            index into the constant pool, the item at which is a
     *            ConstantLongInfo or ConstantDoubleInfo item.
     */

    private void ldc2wInstruction(int index) {
        CLLoadStoreInstruction instr = new CLLoadStoreInstruction(LDC2_W,
                mPC++, index);
        mPC += instr.operandCount();
        mCode.add(instr);
        mInstructionAfterLabel = true;
    }

    /**
     * Construct and return a ConstantValue attribute given the constant value
```

```
     * index.
     *
     * @param c
     *            index into the constant pool, the item at which is the
     *            constant value.
     * @return a ConstantValue attribute.
     */

    private CLConstantValueAttribute constantValueAttribute(int c) {
        int attributeNameIndex = constantPool
                .constantUtf8Info(ATT_CONSTANT_VALUE);
        return new CLConstantValueAttribute(attributeNameIndex, 2, c);
    }

    /**
     * Construct and return a Code attribute given the list of bytes that make
     * up the instructions and their operands, exception table, maximum depth of
     * operand stack, and maximum number of local variables.
     *
     * @param byteCode
     *            list of bytes that make up the instructions and their
     *            operands.
     * @param exceptionTable
     *            exception table.
     * @param stackDepth
     *            maximum depth of operand stack.
     * @param maxLocals
     *            maximum number of local variables.
     * @return a Code attribute.
     */

    private CLCodeAttribute codeAttribute(ArrayList<Integer> byteCode,
            ArrayList<CLExceptionInfo> exceptionTable, int stackDepth,
            int maxLocals) {
        int codeLength = byteCode.size();
        int attributeNameIndex = constantPool.constantUtf8Info(ATT_CODE);
        int attributeLength = codeLength + 8 * exceptionTable.size() + 12;
        for (int i = 0; i < mCodeAttributes.size(); i++) {
            attributeLength += 6 + mCodeAttributes.get(i).attributeLength;
        }
        return new CLCodeAttribute(attributeNameIndex, attributeLength,
                stackDepth, maxLocals, (long) codeLength, byteCode,
                exceptionTable.size(), exceptionTable, mCodeAttributes.size(),
                mCodeAttributes);
    }

    /**
     * Construct and return an ExceptionsAttribute given the list of exceptions.
     *
     * @param exceptions
     *            list of exceptions in internal form.
     * @return an Exceptions attribute.
     */

    private CLExceptionsAttribute exceptionsAttribute(
            ArrayList<String> exceptions) {
        int attributeNameIndex = constantPool.constantUtf8Info(ATT_EXCEPTIONS);
        ArrayList<Integer> exceptionIndexTable = new ArrayList<Integer>();
        for (int i = 0; i < exceptions.size(); i++) {
            String e = exceptions.get(i);
            exceptionIndexTable.add(new Integer(constantPool
                    .constantClassInfo(e)));
        }
        return new CLExceptionsAttribute(attributeNameIndex,
                exceptionIndexTable.size() * 2 + 2, exceptionIndexTable.size(),
                exceptionIndexTable);
    }

    /**
```

```
     * Construct and return InnerClasses attribute.
     *
     * @return an InnerClasses attribute.
     */

    private CLInnerClassesAttribute innerClassesAttribute() {
        int attributeNameIndex = constantPool
                .constantUtf8Info(ATT_INNER_CLASSES);
        long attributeLength = innerClasses.size() * 8 + 2;
        return new CLInnerClassesAttribute(attributeNameIndex, attributeLength,
                innerClasses.size(), innerClasses);
    }

    /**
     * Construct and return a Synthetic attribute.
     *
     * @param a
     *            Synthetic attribute.
     */

    private CLAttributeInfo syntheticAttribute() {
        int attributeNameIndex = constantPool.constantUtf8Info(ATT_SYNTHETIC);
        return new CLSyntheticAttribute(attributeNameIndex, 0);
    }

    /**
     * Used to report an error if the opcode used for adding an instruction is
     * invalid, or if an incorrect method from CLEmitter is used to add the
     * opcode.
     *
     * @param opcode
     *            opcode of the instruction.
     */

    private void reportOpcodeError(int opcode) {
        if (!CLInstruction.isValid(opcode)) {
            reportEmitterError("%s: Invalid opcode '%d'", eCurrentMethod,
                    opcode);
        } else {
            reportEmitterError(
                    "%s: Incorrect method used to add instruction '%s'",
                    eCurrentMethod,
                    CLInstruction.instructionInfo[opcode].mnemonic);
        }
    }

    /**
     * Report any error that occurs while creating/writing the class, to STDERR.
     *
     * @param message
     *            message identifying the error.
     * @param args
     *            related values.
     */

    private void reportEmitterError(String message, Object... args) {
        System.err.printf(message, args);
        System.err.println();
        errorHasOccurred = true;
    }

    // ////////////////////////////////////////////////
    // CLEmitter proper begins here
    // ////////////////////////////////////////////////

    /**
     * Construct a CLEmitter instance.
     *
     * @param toFile
```

```
892      *            if true, the in-memory representation of the class file will
893      *            be written to the file system. Otherwise, it won't be saved as
894      *            a file.
895      */
896
897     public CLEmitter(boolean toFile) {
898         destDir = ".";
899         this.toFile = toFile;
900     }
901
902     /**
903      * Set the destination directory for the class file to the specified value.
904      *
905      * @param destDir
906      *            destination directory.
907      */
908
909     public void destinationDir(String destDir) {
910         this.destDir = destDir;
911     }
912
913     /**
914      * Has an emitter error occurred up to now?
915      *
916      * @return true or false.
917      */
918
919     public boolean errorHasOccurred() {
920         return errorHasOccurred;
921     }
922
923     /**
924      * Add a class or interface. This method instantiates a class file
925      * representation in memory, so must be called prior to methods that add
926      * information (fields, methods, instructions, etc.) to the class.
927      *
928      * @param accessFlags
929      *            the access flags for the class or interface.
930      * @param thisClass
931      *            fully qualified name of the class or interface in internal
932      *            form.
933      * @param superClass
934      *            fully qualified name of the parent class in internal form.
935      * @param superInterfaces
936      *            list of direct super interfaces of this class or interface as
937      *            fully qualified names in internal form.
938      * @param isSynthetic
939      *            whether the class or interface is synthetic.
940      */
941
942     public void addClass(ArrayList<String> accessFlags, String thisClass,
943             String superClass, ArrayList<String> superInterfaces,
944             boolean isSynthetic) {
945         clFile = new CLFile();
946         constantPool = new CLConstantPool();
947         interfaces = new ArrayList<Integer>();
948         fields = new ArrayList<CLFieldInfo>();
949         methods = new ArrayList<CLMethodInfo>();
950         attributes = new ArrayList<CLAttributeInfo>();
951         innerClasses = new ArrayList<CLInnerClassInfo>();
952         errorHasOccurred = false;
953         clFile.magic = MAGIC;
954         clFile.majorVersion = MAJOR_VERSION;
955         clFile.minorVersion = MINOR_VERSION;
956         if (!validInternalForm(thisClass)) {
957             reportEmitterError("'%s' is not in internal form", thisClass);
958         }
959         if (!validInternalForm(superClass)) {
960             reportEmitterError("'%s' is not in internal form", superClass);
```

```
961         }
962         if (accessFlags != null) {
963             for (int i = 0; i < accessFlags.size(); i++) {
964                 clFile.accessFlags |= CLFile
965                         .accessFlagToInt(accessFlags.get(i));
966             }
967         }
968         name = thisClass;
969         clFile.thisClass = constantPool.constantClassInfo(thisClass);
970         clFile.superClass = constantPool.constantClassInfo(superClass);
971         for (int i = 0; superInterfaces != null && i < superInterfaces.size(); i+
+) {
972             if (!validInternalForm(superInterfaces.get(i))) {
973                 reportEmitterError("'%s' is not in internal form",
974                         superInterfaces.get(i));
975             }
976             interfaces.add(new Integer(constantPool
977                     .constantClassInfo(superInterfaces.get(i))));
978         }
979         if (isSynthetic) {
980             addClassAttribute(syntheticAttribute());
981         }
982     }
983 
984     /**
985      * Add an inner class. Note that this only registers the inner class with
986      * its parent and does not create the class.
987      *
988      * @param accessFlags
989      *            access flags for the inner class
990      * @param innerClass
991      *            fully qualified name of the inner class in internal form.
992      * @param outerClass
993      *            fully qualified name of the outer class in internal form.
994      * @param innerName
995      *            simple name of the inner class.
996      */
997 
998     public void addInnerClass(ArrayList<String> accessFlags, String innerClass,
999             String outerClass, String innerName) {
1000         int flags = 0;
1001         if (accessFlags != null) {
1002             for (int j = 0; j < accessFlags.size(); j++) {
1003                 flags |= CLFile.accessFlagToInt(accessFlags.get(j));
1004             }
1005         }
1006         CLInnerClassInfo innerClassInfo = new CLInnerClassInfo(constantPool
1007                 .constantClassInfo(innerClass), constantPool
1008                 .constantClassInfo(outerClass), constantPool
1009                 .constantUtf8Info(innerName), flags);
1010         innerClasses.add(innerClassInfo);
1011     }
1012 
1013     /**
1014      * Add a field without initialization.
1015      *
1016      * @param accessFlags
1017      *            access flags for the field.
1018      * @param name
1019      *            name of the field.
1020      * @param type
1021      *            type descriptor of the field.
1022      * @param isSynthetic
1023      *            is this a synthetic field?
1024      */
1025 
1026     public void addField(ArrayList<String> accessFlags, String name,
1027             String type, boolean isSynthetic) {
1028         addFieldInfo(accessFlags, name, type, isSynthetic, -1);
```

```
1029    }
1030
1031    /**
1032     * Add an int, short, char, byte, or boolean field with initialization. If
1033     * the field is final, the initialization is added to the constant pool. The
1034     * initializations are all stored as ints, where boolean true and false are
1035     * 1 and 0 respectively, and short, char, and byte must be cast to int.
1036     *
1037     * @param accessFlags
1038     *            access flags for the field.
1039     * @param name
1040     *            name of the field.
1041     * @param type
1042     *            type descriptor of the field.
1043     * @param isSynthetic
1044     *            is this a synthetic field?
1045     * @param i
1046     *            int value.
1047     */
1048
1049    public void addField(ArrayList<String> accessFlags, String name,
1050            String type, boolean isSynthetic, int i) {
1051        addFieldInfo(accessFlags, name, type, isSynthetic, constantPool
1052                .constantIntegerInfo(i));
1053    }
1054
1055    /**
1056     * Add a float field with initialization. If the field is final, the
1057     * initialization is added to the constant pool.
1058     *
1059     * @param accessFlags
1060     *            access flags for the field.
1061     * @param name
1062     *            name of the field.
1063     * @param isSynthetic
1064     *            is this a synthetic field?
1065     * @param f
1066     *            float value.
1067     */
1068
1069    public void addField(ArrayList<String> accessFlags, String name,
1070            boolean isSynthetic, float f) {
1071        addFieldInfo(accessFlags, name, "F", isSynthetic, constantPool
1072                .constantFloatInfo(f));
1073    }
1074
1075    /**
1076     * Add a long field with initialization. If the field is final, the
1077     * initialization is added to the constant pool.
1078     *
1079     * @param accessFlags
1080     *            access flags for the field.
1081     * @param name
1082     *            name of the field.
1083     * @param isSynthetic
1084     *            is this a synthetic field?
1085     * @param l
1086     *            long value.
1087     */
1088
1089    public void addField(ArrayList<String> accessFlags, String name,
1090            boolean isSynthetic, long l) {
1091        addFieldInfo(accessFlags, name, "J", isSynthetic, constantPool
1092                .constantLongInfo(l));
1093    }
1094
1095    /**
1096     * Add a double field with initialization. If the field is final, the
1097     * initialization is added to the constant pool.
```

```
1098     *
1099     * @param accessFlags
1100     *            access flags for the field.
1101     * @param name
1102     *            name of the field.
1103     * @param isSynthetic
1104     *            is this a synthetic field?
1105     * @param d
1106     *            double value.
1107     */
1108
1109    public void addField(ArrayList<String> accessFlags, String name,
1110            boolean isSynthetic, double d) {
1111        addFieldInfo(accessFlags, name, "D", isSynthetic, constantPool
1112                .constantDoubleInfo(d));
1113    }
1114
1115    /**
1116     * Add a String type field with initialization. If the field is final, the
1117     * initialization is added to the constant pool.
1118     *
1119     * @param accessFlags
1120     *            access flags for the field.
1121     * @param name
1122     *            name of the field.
1123     * @param isSynthetic
1124     *            is this a synthetic field?
1125     * @param s
1126     *            String value
1127     */
1128
1129    public void addField(ArrayList<String> accessFlags, String name,
1130            boolean isSynthetic, String s) {
1131        addFieldInfo(accessFlags, name, "Ljava/lang/String;", isSynthetic,
1132                constantPool.constantStringInfo(s));
1133    }
1134
1135    /**
1136     * Add a method. Instructions can subsequently be added to this method using
1137     * the appropriate methods for adding instructions.
1138     *
1139     * @param accessFlags
1140     *            access flags for the method.
1141     * @param name
1142     *            name of the method.
1143     * @param descriptor
1144     *            descriptor specifying the return type and the types of the
1145     *            formal parameters of the method.
1146     * @param exceptions
1147     *            exceptions thrown by the method, each being a name in fully
1148     *            qualified internal form.
1149     * @param isSynthetic
1150     *            whether this is a synthetic method?
1151     */
1152
1153    public void addMethod(ArrayList<String> accessFlags, String name,
1154            String descriptor, ArrayList<String> exceptions, boolean isSynthetic)
1155        if (!validMethodDescriptor(descriptor)) {
1156            reportEmitterError(
1157                    "'%s' is not a valid type descriptor for method",
1158                    descriptor);
1159        }
1160        endOpenMethodIfAny(); // close any previous method
1161        isMethodOpen = true;
1162        initializeMethodVariables();
1163        eCurrentMethod = name + descriptor;
1164        if (accessFlags != null) {
1165            for (int i = 0; i < accessFlags.size(); i++) {
```

```
                mAccessFlags |= CLFile.accessFlagToInt(accessFlags.get(i));
            }
        }
        mArgumentCount = argumentCount(descriptor)
                + (accessFlags.contains("static") ? 0 : 1);
        mNameIndex = constantPool.constantUtf8Info(name);
        mDescriptorIndex = constantPool.constantUtf8Info(descriptor);
        if (exceptions != null && exceptions.size() > 0) {
            addMethodAttribute(exceptionsAttribute(exceptions));
        }
        if (isSynthetic) {
            addMethodAttribute(syntheticAttribute());
        }
    }

    /**
     * Add an exception handler.
     *
     * @param startLabel
     *            the exception handler is active from the instruction following
     *            this label in the code section of the current method being
     *            added ...
     * @param endLabel
     *            to the instruction following this label. Formally, the handler
     *            is active while the program counter is within the interval
     *            [startLabel, endLabel).
     * @param handlerLabel
     *            the handler begins with instruction following this label.
     * @param catchType
     *            the exception type that this exception handler is designated
     *            to catch, as a fully qualified name in internal form. If null,
     *            this exception handler is called for all exceptions; this is
     *            used to immplement "finally".
     */

    public void addExceptionHandler(String startLabel, String endLabel,
            String handlerLabel, String catchType) {
        if (catchType != null && !validInternalForm(catchType)) {
            reportEmitterError("'%s' is not in internal form", catchType);
        }
        CLException e = new CLException(startLabel, endLabel, handlerLabel,
                catchType);
        mExceptionHandlers.add(e);
    }

    /**
     * Add a no argument instruction. Following instructions can be added using
     * this method:
     *
     * <p/>
     * Arithmetic Instructions:
     *
     * <pre>
     *   IADD, LADD, FADD, DADD, ISUB, LSUB, FSUB, DSUB, IMUL,
     *   LMUL, FMUL, DMUL, IDIV, LDIV, FDIV, DDIV, IREM, LREM, FREM,
     *   DREM, INEG, LNEG, FNEG, DNEG
     * </pre>
     *
     * <p/>
     * Array Instructions:
     *
     * <pre>
     *   IALOAD, LALOAD, FALOAD, DALOAD, AALOAD, BALOAD, CALOAD,
     *   SALOAD, IASTORE, LASTORE, FASTORE, DASTORE, AASTORE,
     *   BASTORE, CASTORE, SASTORE, ARRAYLENGTH
     * </pre>
     *
     * <p/>
     * Bit Instructions:
```

```
     *
     * <pre>
     *   ISHL, ISHR, IUSHR, LSHL, LSHR, LUSHR, IOR, LOR, IAND, LAND,
     *   IXOR, LXOR
     * </pre>
     *
     * <p/>
     * Comparison Instructions:
     *
     * <pre>
     *   DCMPG, DCMPL, FCMPG, FCMPL, LCMP
     * </pre>
     *
     * <p/>
     * Conversion Instructions:
     *
     * <pre>
     *   I2B, I2C, I2S, I2L, I2F, I2D, L2F, L2D, L2I, F2D, F2I,
     *   F2L, D2I, D2L, D2F
     * </pre>
     *
     * <p/>
     * Load Store Instructions:
     *
     * <pre>
     *   ILOAD_0, ILOAD_1, ILOAD_2, ILOAD_3, LLOAD_0, LLOAD_1,
     *   LLOAD_2, LLOAD_3, FLOAD_0, FLOAD_1, FLOAD_2, FLOAD_3,
     *   DLOAD_0, DLOAD_1, DLOAD_2, DLOAD_3, ALOAD_0, ALOAD_1,
     *   ALOAD_2, ALOAD_3, ISTORE_0, ISTORE_1, ISTORE_2, ISTORE_3,
     *   LSTORE_0, LSTORE_1, LSTORE_2, LSTORE_3, FSTORE_0, FSTORE_1,
     *   FSTORE_2, FSTORE_3, DSTORE_0, DSTORE_1, DSTORE_2, DSTORE_3,
     *   ASTORE_0, ASTORE_1, ASTORE_2, ASTORE_3, ICONST_0, ICONST_1,
     *   ICONST_2, ICONST_3, ICONST_4, ICONST_5, ICONST_M1, LCONST_0,
     *   LCONST_1, FCONST_0, FCONST_1, FCONST_2, DCONST_0, DCONST_1,
     *   ACONST_NULL, WIDE (added automatically where necesary)
     * </pre>
     *
     * <p/>
     * Method Instructions:
     *
     * <pre>
     *   IRETURN, LRETURN, FRETURN, DRETURN, ARETURN, RETURN
     * </pre>
     *
     * <p/>
     * Stack Instructions:
     *
     * <pre>
     *   POP, POP2, DUP, DUP_X1, DUP_X2, DUP2, DUP2_X1, DUP2_X2, SWAP
     * </pre>
     *
     * <p/>
     * Miscellaneous Instructions:
     *
     * <pre>
     *   NOP, ATHROW, MONITORENTER, MONITOREXIT
     * </pre>
     *
     * The opcodes for instructions are defined in CLConstants class.
     *
     * @param opcode
     *            opcode of the instruction.
     */

    public void addNoArgInstruction(int opcode) {
        CLInstruction instr = null;
        switch (CLInstruction.instructionInfo[opcode].category) {
        case ARITHMETIC1:
            instr = new CLArithmeticInstruction(opcode, mPC++);
```

```
            break;
        case ARRAY2:
            instr = new CLArrayInstruction(opcode, mPC++);
            break;
        case BIT:
            instr = new CLBitInstruction(opcode, mPC++);
            break;
        case COMPARISON:
            instr = new CLComparisonInstruction(opcode, mPC++);
            break;
        case CONVERSION:
            instr = new CLConversionInstruction(opcode, mPC++);
            break;
        case LOAD_STORE1:
            instr = new CLLoadStoreInstruction(opcode, mPC++);
            break;
        case METHOD2:
            instr = new CLMethodInstruction(opcode, mPC++);
            break;
        case MISC:
            instr = new CLMiscInstruction(opcode, mPC++);
            break;
        case STACK:
            instr = new CLStackInstruction(opcode, mPC++);
            break;
        default:
            reportOpcodeError(opcode);
        }
        if (instr != null) {
            mPC += instr.operandCount();
            mCode.add(instr);
            mInstructionAfterLabel = true;
        }
    }

    /**
     * Add a one argument instruction. Wideable instructions are widened if
     * necessary by adding a WIDE instruction before the instruction. Following
     * instructions can be added using this method.
     *
     * <p/>
     * Load Store Instructions:
     *
     * <pre>
     *   ILOAD, LLOAD, FLOAD, DLOAD, ALOAD, ISTORE, LSTORE, FSTORE,
     *   DSTORE, ASTORE, BIPUSH, SIPUSH
     * </pre>
     *
     * <p/>
     * Flow Control Instructions:
     *
     * <pre>
     * RET
     * </pre>
     *
     * The opcodes for instructions are defined in CLConstants class.
     *
     * @param opcode
     *            opcode of the instruction.
     * @param arg
     *            the argument. For the instructions that deal with local
     *            variables, the argument is the local variable index; for
     *            BIPUSH and SIPUSH instructions, the argument is the constant
     *            byte or short value.
     */

    public void addOneArgInstruction(int opcode, int arg) {
        CLInstruction instr = null;
        boolean isWidened = false;
```

```
1373        switch (CLInstruction.instructionInfo[opcode].category) {
1374        case LOAD_STORE2:
1375            isWidened = arg > 255;
1376            if (isWidened) {
1377                CLLoadStoreInstruction wideInstr = new CLLoadStoreInstruction(
1378                        WIDE, mPC++);
1379                mCode.add(wideInstr);
1380            }
1381            instr = new CLLoadStoreInstruction(opcode, mPC++, arg, isWidened);
1382            break;
1383        case LOAD_STORE3:
1384            instr = new CLLoadStoreInstruction(opcode, mPC++, arg);
1385            break;
1386        case FLOW_CONTROL2:
1387            isWidened = arg > 255;
1388            if (isWidened) {
1389                CLLoadStoreInstruction wideInstr = new CLLoadStoreInstruction(
1390                        WIDE, mPC++);
1391                mCode.add(wideInstr);
1392            }
1393            instr = new CLFlowControlInstruction(mPC++, arg, isWidened);
1394            break;
1395        default:
1396            reportOpcodeError(opcode);
1397        }
1398        if (instr != null) {
1399            mPC += instr.operandCount();
1400            mCode.add(instr);
1401            mInstructionAfterLabel = true;
1402        }
1403    }
1404
1405    /**
1406     * Add an IINC instruction to increment a variable by a constant. The
1407     * instruction is widened if necessary by adding a WIDE instruction before
1408     * the instruction.
1409     *
1410     * @param index
1411     *            local variable index.
1412     * @param constVal
1413     *            increment value.
1414     */
1415
1416    public void addIINCInstruction(int index, int constVal) {
1417        boolean isWidened = index > 255 || constVal < Byte.MIN_VALUE
1418                || constVal > Byte.MAX_VALUE;
1419        if (isWidened) {
1420            CLLoadStoreInstruction wideInstr = new CLLoadStoreInstruction(WIDE,
1421                    mPC++);
1422            mCode.add(wideInstr);
1423        }
1424        CLArithmeticInstruction instr = new CLArithmeticInstruction(IINC,
1425                mPC++, index, constVal, isWidened);
1426        mPC += instr.operandCount();
1427        mCode.add(instr);
1428        mInstructionAfterLabel = true;
1429    }
1430
1431    /**
1432     * Add a member (field & method) access instruction. Following instructions
1433     * can be added using this method:
1434     *
1435     * <p/>
1436     * Field Instructions:
1437     *
1438     * <pre>
1439     *   GETSTATIC, PUTSTATIC, GETFIELD, PUTFIELD
1440     * </pre>
1441     *
```

```
1442     * <p/>
1443     * Method Instructions:
1444     *
1445     * <pre>
1446     *   INVOKEVIRTUAL, INVOKESPECIAL, INVOKESTATIC, INVOKEINTERFACE,
1447     *   INVOKEDYNAMIC
1448     * </pre>
1449     *
1450     * The opcodes for instructions are defined in CLConstants class.
1451     *
1452     * @param opcode
1453     *            opcode of the instruction.
1454     * @param target
1455     *            fully qualified name in internal form of the class to which
1456     *            the member belongs.
1457     * @param name
1458     *            name of the member.
1459     * @param type
1460     *            type descriptor of the member.
1461     */
1462
1463    public void addMemberAccessInstruction(int opcode, String target,
1464            String name, String type) {
1465        if (!validInternalForm(target)) {
1466            reportEmitterError("%s: '%s' is not in internal form",
1467                    eCurrentMethod, target);
1468        }
1469        CLInstruction instr = null;
1470        int index, stackUnits;
1471        switch (CLInstruction.instructionInfo[opcode].category) {
1472        case FIELD:
1473            if (!validTypeDescriptor(type)) {
1474                reportEmitterError(
1475                        "%s: '%s' is not a valid type descriptor for field",
1476                        eCurrentMethod, type);
1477            }
1478            index = constantPool.constantFieldRefInfo(target, name, type);
1479            stackUnits = typeStackResidue(type);
1480            if ((opcode == GETFIELD) || (opcode == PUTFIELD)) {
1481                // This is because target of this method is also
1482                // consumed from the operand stack
1483                stackUnits--;
1484            }
1485            instr = new CLFieldInstruction(opcode, mPC++, index, stackUnits);
1486            break;
1487        case METHOD1:
1488            if (!validMethodDescriptor(type)) {
1489                reportEmitterError(
1490                        "%s: '%s' is not a valid type descriptor for "
1491                                + "method", eCurrentMethod, type);
1492            }
1493            if (opcode == INVOKEINTERFACE) {
1494                index = constantPool.constantInterfaceMethodRefInfo(target,
1495                        name, type);
1496            } else {
1497                index = constantPool.constantMethodRefInfo(target, name, type);
1498            }
1499            stackUnits = methodStackResidue(type);
1500            if (opcode != INVOKESTATIC) {
1501                // This is because target of this method is also
1502                // consumed from the operand stack
1503                stackUnits--;
1504            }
1505            instr = new CLMethodInstruction(opcode, mPC++, index, stackUnits);
1506
1507            // INVOKEINTERFACE expects the number of arguments in
1508            // the method to be specified explicitly.
1509            if (opcode == INVOKEINTERFACE) {
1510                // We add 1 to account for "this"
```

```
1511                ((CLMethodInstruction) instr)
1512                        .setArgumentCount(argumentCount(type) + 1);
1513            }
1514            break;
1515        default:
1516            reportOpcodeError(opcode);
1517        }
1518        if (instr != null) {
1519            mPC += instr.operandCount();
1520            mCode.add(instr);
1521        }
1522    }
1523
1524    /**
1525     * Add a reference (object) instruction. Following instructions can be added
1526     * using this method:
1527     *
1528     * <pre>
1529     *   NEW, CHECKCAST, INSTANCEOF
1530     * </pre>
1531     *
1532     * The opcodes for instructions are defined in CLConstants class.
1533     *
1534     * @param opcode
1535     *            opcode of the instruction.
1536     * @param type
1537     *            reference type in internal form.
1538     */
1539
1540    public void addReferenceInstruction(int opcode, String type) {
1541        if (!validTypeDescriptor(type) && !validInternalForm(type)) {
1542            reportEmitterError("%s: '%s' is neither a type descriptor nor in "
1543                    + "internal form", eCurrentMethod, type);
1544        }
1545        CLInstruction instr = null;
1546        switch (CLInstruction.instructionInfo[opcode].category) {
1547        case OBJECT:
1548            int index = constantPool.constantClassInfo(type);
1549            instr = new CLObjectInstruction(opcode, mPC, index);
1550            break;
1551        default:
1552            reportOpcodeError(opcode);
1553        }
1554        if (instr != null) {
1555            mPC += instr.operandCount();
1556            mCode.add(instr);
1557        }
1558    }
1559
1560    /**
1561     * Add an array instruction. Following instructions can be added using this
1562     * method:
1563     *
1564     * <pre>
1565     *   NEWARRAY, ANEWARRAY
1566     * </pre>
1567     *
1568     * The opcodes for instructions are defined in CLConstants class.
1569     *
1570     * @param opcode
1571     *            opcode of the instruction.
1572     * @param type
1573     *            array type. In case of NEWARRAY, the primitive types are
1574     *            specified as: "Z" for boolean, "C" for char, "F" for float,
1575     *            "D" for double, "B" for byte, "S" for short, "I" for int, "J"
1576     *            for long. In case of ANEWARRAY, reference types are specified
1577     *            in internal form.
1578     */
1579
```

```
1580    public void addArrayInstruction(int opcode, String type) {
1581        CLInstruction instr = null;
1582        switch (CLInstruction.instructionInfo[opcode].category) {
1583        case ARRAY1:
1584            int index = 0;
1585            if (opcode == NEWARRAY) {
1586                if (type.equalsIgnoreCase("Z")) {
1587                    index = 4;
1588                } else if (type.equalsIgnoreCase("C")) {
1589                    index = 5;
1590                } else if (type.equalsIgnoreCase("F")) {
1591                    index = 6;
1592                } else if (type.equalsIgnoreCase("D")) {
1593                    index = 7;
1594                } else if (type.equalsIgnoreCase("B")) {
1595                    index = 8;
1596                } else if (type.equalsIgnoreCase("S")) {
1597                    index = 9;
1598                } else if (type.equalsIgnoreCase("I")) {
1599                    index = 10;
1600                } else if (type.equalsIgnoreCase("J")) {
1601                    index = 11;
1602                } else {
1603                    reportEmitterError(
1604                            "%s: '%s' is not a valid primitive type",
1605                            eCurrentMethod, type);
1606                }
1607            } else {
1608                if (!validTypeDescriptor(type) && !validInternalForm(type)) {
1609                    reportEmitterError(
1610                            "%s: '%s' is not a valid type descriptor "
1611                                    + "for an array", eCurrentMethod, type);
1612                }
1613                index = constantPool.constantClassInfo(type);
1614            }
1615            instr = new CLArrayInstruction(opcode, mPC++, index);
1616            break;
1617        default:
1618            reportOpcodeError(opcode);
1619        }
1620        if (instr != null) {
1621            mPC += instr.operandCount();
1622            mCode.add(instr);
1623        }
1624    }
1625
1626    /**
1627     * Add a MULTIANEWARRAY instruction for creating multi-dimensional arrays.
1628     *
1629     * @param type
1630     *            array type in internal form.
1631     * @param dim
1632     *            number of dimensions.
1633     */
1634
1635    public void addMULTIANEWARRAYInstruction(String type, int dim) {
1636        CLInstruction instr = null;
1637        if (!validTypeDescriptor(type)) {
1638            reportEmitterError(
1639                    "%s: '%s' is not a valid type descriptor for an array",
1640                    eCurrentMethod, type);
1641        }
1642        int index = constantPool.constantClassInfo(type);
1643        instr = new CLArrayInstruction(MULTIANEWARRAY, mPC++, index, dim);
1644        if (instr != null) {
1645            mPC += instr.operandCount();
1646            mCode.add(instr);
1647        }
1648    }
```

```
1649
1650    /**
1651     * Add a branch instruction. Following instructions can be added using this
1652     * method:
1653     *
1654     * <pre>
1655     *   IFEQ, IFNE, IFLT, IFGE, IFGT, IFLE, IF_ICMPEQ, IF_ICMPNE,
1656     *   IF_ICMPLT, IF_ICMPGE, IF_ICMPGT, IF_ICMPLE, IF_ACMPEQ,
1657     *   IF_ACMPNE, GOTO, JSR, IF_NULL, IF_NONNULL, GOTO_W, JSR_W
1658     * </pre>
1659     *
1660     * The opcodes for instructions are defined in CLConstants class.
1661     *
1662     * @param opcode
1663     *            opcode of the instruction.
1664     * @param label
1665     *            branch label.
1666     */
1667
1668    public void addBranchInstruction(int opcode, String label) {
1669        CLInstruction instr = null;
1670        switch (CLInstruction.instructionInfo[opcode].category) {
1671        case FLOW_CONTROL1:
1672            instr = new CLFlowControlInstruction(opcode, mPC++, label);
1673            break;
1674        default:
1675            reportOpcodeError(opcode);
1676        }
1677        if (instr != null) {
1678            mPC += instr.operandCount();
1679            mCode.add(instr);
1680            mInstructionAfterLabel = true;
1681        }
1682    }
1683
1684    /**
1685     * Add a TABLESWITCH instruction -- used for switch statements.
1686     *
1687     * @param defaultLabel
1688     *            jump label for default value.
1689     * @param low
1690     *            smallest value of index.
1691     * @param high
1692     *            highest value of index.
1693     * @param labels
1694     *            list of jump labels for each index value from low to high, end
1695     *            values included.
1696     */
1697
1698    public void addTABLESWITCHInstruction(String defaultLabel, int low,
1699            int high, ArrayList<String> labels) {
1700        CLFlowControlInstruction instr = new CLFlowControlInstruction(
1701                TABLESWITCH, mPC++, defaultLabel, low, high, labels);
1702        mPC += instr.operandCount();
1703        mCode.add(instr);
1704        mInstructionAfterLabel = true;
1705    }
1706
1707    /**
1708     * Add a LOOKUPSWITCH instruction -- used for switch statements.
1709     *
1710     * @param defaultLabel
1711     *            jump label for default value.
1712     * @param numPairs
1713     *            number of pairs in the match table.
1714     * @param matchLabelPairs
1715     *            key match table.
1716     */
1717
```

```
1718    public void addLOOKUPSWITCHInstruction(String defaultLabel, int numPairs,
1719            TreeMap<Integer, String> matchLabelPairs) {
1720        CLFlowControlInstruction instr = new CLFlowControlInstruction(
1721                LOOKUPSWITCH, mPC++, defaultLabel, numPairs, matchLabelPairs);
1722        mPC += instr.operandCount();
1723        mCode.add(instr);
1724        mInstructionAfterLabel = true;
1725    }
1726
1727    /**
1728     * Add an LDC instruction to load an int constant on the operand stack.
1729     *
1730     * @param i
1731     *            int constant.
1732     */
1733
1734    public void addLDCInstruction(int i) {
1735        ldcInstruction(constantPool.constantIntegerInfo(i));
1736    }
1737
1738    /**
1739     * Add an LDC instruction to load a float constant on the operand stack.
1740     *
1741     * @param f
1742     *            float constant.
1743     */
1744
1745    public void addLDCInstruction(float f) {
1746        ldcInstruction(constantPool.constantFloatInfo(f));
1747    }
1748
1749    /**
1750     * Add an LDC instruction to load a long constant on the operand stack.
1751     *
1752     * @param l
1753     *            long constant.
1754     */
1755
1756    public void addLDCInstruction(long l) {
1757        ldc2wInstruction(constantPool.constantLongInfo(l));
1758    }
1759
1760    /**
1761     * Add an LDC instruction to load a double constant on the operand stack.
1762     *
1763     * @param d
1764     *            double constant.
1765     */
1766
1767    public void addLDCInstruction(double d) {
1768        ldc2wInstruction(constantPool.constantDoubleInfo(d));
1769    }
1770
1771    /**
1772     * Add an LDC instruction to load a String constant on the operand stack.
1773     *
1774     * @param s
1775     *            String constant.
1776     */
1777
1778    public void addLDCInstruction(String s) {
1779        ldcInstruction(constantPool.constantStringInfo(s));
1780    }
1781
1782    /**
1783     * Add the specified class attribute to the attribyte section of the class.
1784     *
1785     * @param attribute
1786     *            class attribute.
```

```
1787     */
1788
1789    public void addClassAttribute(CLAttributeInfo attribute) {
1790        if (attributes != null) {
1791            attributes.add(attribute);
1792        }
1793    }
1794
1795    /**
1796     * Add the specified method attribute to the attribute section of the method
1797     * last added.
1798     *
1799     * @param attribute
1800     *            method attribute.
1801     */
1802
1803    public void addMethodAttribute(CLAttributeInfo attribute) {
1804        if (mAttributes != null) {
1805            mAttributes.add(attribute);
1806        }
1807    }
1808
1809    /**
1810     * Add the specified field attribute the attribute section of the field last
1811     * added.
1812     *
1813     * @param attribute
1814     *            field attribute.
1815     */
1816
1817    public void addFieldAttribute(CLAttributeInfo attribute) {
1818        if (fAttributes != null) {
1819            fAttributes.add(attribute);
1820        }
1821    }
1822
1823    /**
1824     * Add the specified code attribute to the attribute section of the code for
1825     * the method last added.
1826     *
1827     * @param attribute
1828     *            code attribute.
1829     */
1830
1831    public void addCodeAttribute(CLAttributeInfo attribute) {
1832        if (mCodeAttributes != null) {
1833            mCodeAttributes.add(attribute);
1834        }
1835    }
1836
1837    /**
1838     * Add a jump label to the code section of the method being added. A flow
1839     * control instruction that was added with this label will jump to the
1840     * instruction right after the label.
1841     *
1842     * @param label
1843     *            jump label.
1844     */
1845
1846    public void addLabel(String label) {
1847        mLabels.put(label, mPC);
1848        mInstructionAfterLabel = false;
1849    }
1850
1851    /**
1852     * Construct and return a unique jump label.
1853     *
1854     * @return unique jump label.
1855     */
```

```
1856
1857    public String createLabel() {
1858        return "Label" + mLabelCount++;
1859    }
1860
1861    /**
1862     * Return the pc (location counter). The next instruction will be added with
1863     * this pc.
1864     *
1865     * @return the pc.
1866     */
1867
1868    public int pc() {
1869        return mPC;
1870    }
1871
1872    /**
1873     * Return the constant pool of the class being built.
1874     *
1875     * @return constant pool.
1876     */
1877
1878    public CLConstantPool constantPool() {
1879        return constantPool;
1880    }
1881
1882    /**
1883     * Set a new ByteClassLoader for loading classes from byte streams.
1884     */
1885
1886    public static void initializeByteClassLoader() {
1887        byteClassLoader = new ByteClassLoader();
1888    }
1889
1890    /**
1891     * Return the CLFile instance corresponding to the class built by this
1892     * emitter.
1893     */
1894
1895    public CLFile clFile() {
1896        return clFile;
1897    }
1898
1899    /**
1900     * Return the class being constructed as a Java Class instance.
1901     *
1902     * @return Java Class instance.
1903     */
1904    public Class toClass() {
1905        endOpenMethodIfAny();
1906        Class theClass = null;
1907        try {
1908            // Extract the bytes from the class representation in
1909            // memory into an array of bytes
1910            ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
1911            CLOutputStream out = new CLOutputStream(new BufferedOutputStream(
1912                    byteStream));
1913            clFile.write(out);
1914            out.close();
1915            byte[] classBytes = byteStream.toByteArray();
1916            byteStream.close();
1917
1918            // Load a Java Class instance from its byte
1919            // representation
1920            byteClassLoader.setClassBytes(classBytes);
1921            theClass = byteClassLoader.loadClass(name, true);
1922        } catch (IOException e) {
1923            reportEmitterError("Cannot write class to byte stream");
1924        } catch (ClassNotFoundException e) {
```

```
            reportEmitterError("Cannot load class from byte stream");
        }
        return theClass;
    }

    /**
     * Write out the class to the file system as a .class file if toFile is
     * true. The destination directory for the file can be set using the
     * destinationDir(String dir) method.
     */

    public void write() {
        endOpenMethodIfAny();
        if (!toFile) {
            return;
        }
        String outFile = destDir + File.separator + name + ".class";
        try {
            File file = new File(destDir + File.separator
                    + name.substring(0, name.lastIndexOf("/") + 1));
            file.mkdirs();
            CLOutputStream out = new CLOutputStream(new BufferedOutputStream(
                    new FileOutputStream(outFile)));
            clFile.write(out);
            out.close();
        } catch (FileNotFoundException e) {
            reportEmitterError("File %s not found", outFile);
        } catch (IOException e) {
            reportEmitterError("Cannot write to file %s", outFile);
        }
    }
}

/**
 * Representation of an exception handler
 */

class CLException {

    /**
     * The exception handler is active from this instruction in the code section
     * of the current method being added to ...
     */
    public String startLabel;

    /**
     * this instruction. Formally, the handler is active while the program
     * counter is within the interval [startPC, endPC).
     */
    public String endLabel;

    /**
     * Instruction after this label is first instruction of the handler.
     */
    public String handlerLabel;

    /**
     * The class of exceptions that this exception handler is designated to
     * catch.
     */
    public String catchType;

    /** startLabel is resolved to this value. */
    public int startPC;

    /** endLabel is resolved to this value. */
    public int endPC;

    /** handlerLabel is resolved to this value. */
```

```
1994    public int handlerPC;
1995
1996    /**
1997     * Construct a CLException object.
1998     *
1999     * @param startLabel
2000     *            the exception handler is active from the instruction following
2001     *            this label in the code section of the current method being
2002     *            added ...
2003     * @param endLabel
2004     *            to the instruction following this label. Formally, the handler
2005     *            is active while the program counter is within the interval
2006     *            [startLabel, endLabel).
2007     * @param handlerLabel
2008     *            the handler begins with instruction following this label.
2009     * @param catchType
2010     *            the exception type that this exception handler is designated
2011     *            to catch, as a fully qualified name in internal form.
2012     */
2013
2014    public CLException(String startLabel, String endLabel, String handlerLabel,
2015            String catchType) {
2016        this.startLabel = startLabel;
2017        this.endLabel = endLabel;
2018        this.handlerLabel = handlerLabel;
2019        this.catchType = catchType;
2020    }
2021
2022    /**
2023     * Resolve the symbolic labels to the corresponding pc values using the given
2024     * label to pc mapping. If unable to resolve a label, the corresponding pc
2025     * is set to 0.
2026     *
2027     * @param labelToPC
2028     *            label to pc mapping.
2029     * @return true if all labels were resolved successfully; false otherwise.
2030     */
2031
2032    public boolean resolveLabels(Hashtable<String, Integer> labelToPC) {
2033        boolean allLabelsResolved = true;
2034        if (labelToPC.containsKey(startLabel)) {
2035            startPC = labelToPC.get(startLabel);
2036        } else {
2037            startPC = 0;
2038            allLabelsResolved = false;
2039        }
2040        if (labelToPC.containsKey(endLabel)) {
2041            endPC = labelToPC.get(endLabel);
2042        } else {
2043            endPC = 0;
2044            allLabelsResolved = false;
2045        }
2046        if (labelToPC.containsKey(handlerLabel)) {
2047            handlerPC = labelToPC.get(handlerLabel);
2048        } else {
2049            handlerPC = 0;
2050            allLabelsResolved = false;
2051        }
2052        return allLabelsResolved;
2053    }
2054
2055}
2056
2057/**
2058 * Instances of this class form the elements of the CLBranchStack which is used
2059 * for control flow analysis to compute maximum depth of operand stack for a
2060 * method.
2061 */
2062
```

```
2063class CLBranchTarget {
2064
2065    /** Target instruction. */
2066    public CLInstruction target;
2067
2068    /** Depth of stack before the target instruction is executed. */
2069    public int stackDepth;
2070
2071    /**
2072     * Construct a CLBranchTarget object.
2073     *
2074     * @param target
2075     *            the target instruction.
2076     * @param stackDepth
2077     *            depth of stack before the target instruction is executed.
2078     */
2079
2080    public CLBranchTarget(CLInstruction target, int stackDepth) {
2081        this.target = target;
2082        this.stackDepth = stackDepth;
2083    }
2084
2085}
2086
2087/**
2088 * This class is used for control flow analysis to compute maximum depth of
2089 * operand stack for a method.
2090 */
2091
2092class CLBranchStack {
2093
2094    /** Branch targets yet to visit. */
2095    private Stack<CLBranchTarget> branchTargets;
2096
2097    /** Branch targets already visited. */
2098    private Hashtable<CLInstruction, CLBranchTarget> visitedTargets;
2099
2100    /**
2101     * Return an instance of CLBranchTarget with the specified information and
2102     * record the target as visited.
2103     *
2104     * @param target
2105     *            the target instruction.
2106     * @param stackDepth
2107     *            depth of stack before the target instruction is executed.
2108     * @return an instance of CLBranchTarget.
2109     */
2110
2111    private CLBranchTarget visit(CLInstruction target, int stackDepth) {
2112        CLBranchTarget bt = new CLBranchTarget(target, stackDepth);
2113        visitedTargets.put(target, bt);
2114        return bt;
2115    }
2116
2117    /**
2118     * Return true if the specified instruction has been visited, false
2119     * otherwise.
2120     *
2121     * @param target
2122     *            the target instruction.
2123     * @return true if the specified instruction has been visited, false
2124     *         otherwise.
2125     */
2126
2127    private boolean visited(CLInstruction target) {
2128        return (visitedTargets.get(target) != null);
2129    }
2130
2131    /**
```

```
2132     * Construct a CLBranchStack object.
2133     */
2134
2135    public CLBranchStack() {
2136        this.branchTargets = new Stack<CLBranchTarget>();
2137        this.visitedTargets = new Hashtable<CLInstruction, CLBranchTarget>();
2138    }
2139
2140    /**
2141     * Push the specified information into the stack as a CLBranchTarget
2142     * instance if the target has not been visited yet.
2143     *
2144     * @param target
2145     *            the target instruction.
2146     * @param stackDepth
2147     *            depth of stack before the target instruction is executed.
2148     */
2149
2150    public void push(CLInstruction target, int stackDepth) {
2151        if (visited(target)) {
2152            return;
2153        }
2154        branchTargets.push(visit(target, stackDepth));
2155    }
2156
2157    /**
2158     * Pop and return an element from the stack. null is returned if the stack
2159     * is empty.
2160     *
2161     * @return an element from the stack.
2162     */
2163
2164    public CLBranchTarget pop() {
2165        if (!branchTargets.empty()) {
2166            CLBranchTarget bt = (CLBranchTarget) branchTargets.pop();
2167            return bt;
2168        }
2169        return null;
2170    }
2171
2172}
2173
2174/**
2175 * A class loader to be able to load a class from a byte stream.
2176 */
2177
2178class ByteClassLoader extends ClassLoader {
2179
2180    /** Bytes representing the class. */
2181    private byte[] bytes;
2182
2183    /** Has a package been defined for this class loader? */
2184    private boolean pkgDefined = false;
2185
2186    /**
2187     * Set the bytes representing the class.
2188     *
2189     * @param bytes
2190     *            bytes representing the class.
2191     */
2192
2193    public void setClassBytes(byte[] bytes) {
2194        this.bytes = bytes;
2195    }
2196
2197    /**
2198     * @inheritDoc
2199     */
2200
```

```
2201    public Class<?> loadClass(String name, boolean resolve)
2202            throws ClassNotFoundException {
2203        Class cls = findLoadedClass(name);
2204        if (cls == null) {
2205            try {
2206                cls = findSystemClass(name);
2207            } catch (Exception e) {
2208                // Ignore these
2209            }
2210        }
2211        if (cls == null) {
2212            name = name.replace("/", ".");
2213            String pkg = name.lastIndexOf('.') == -1 ? "" : name.substring(0,
2214                    name.lastIndexOf('.'));
2215            if (!pkgDefined) {
2216                // Packages must be created before the class is
2217                // defined, and package names must be unique
2218                // within
2219                // a class loader and cannot be redefined or
2220                // changed once created
2221                definePackage(pkg, "", "", "", "", "", "", null);
2222                pkgDefined = true;
2223            }
2224            cls = defineClass(name, bytes, 0, bytes.length);
2225            if (resolve && cls != null) {
2226                resolveClass(cls);
2227            }
2228        }
2229        return cls;
2230    }
2231
2232}
2233
2234/**
2235 * Inherits from java.out.DataOutputStream and provides an extra function for
2236 * writing unsigned int to the output stream, which is required for writing Java
2237 * class files.
2238 */
2239
2240class CLOutputStream extends DataOutputStream {
2241
2242    /**
2243     * Construct a CLOutputStream from the specified output stream.
2244     *
2245     * @param out
2246     *            output stream.
2247     */
2248
2249    public CLOutputStream(OutputStream out) {
2250        super(out);
2251    }
2252
2253    /**
2254     * Write four bytes to the output stream to represent the value of the
2255     * argument. The byte values to be written, in the order shown, are:
2256     *
2257     * <pre>
2258     *     (byte) ( 0xFF &amp; ( v &gt;&gt; 24 ) )
2259     *     (byte) ( 0xFF &amp; ( v &gt;&gt; 16 ) )
2260     *     (byte) ( 0xFF &amp; ( v &gt;&gt; 8 ) )
2261     *     (byte) ( 0xFF &amp; v )
2262     * </pre>
2263     *
2264     * @param v
2265     *            the int value to be written.
2266     * @throws IOException
2267     *             if an error occurs while writing.
2268     */
2269
```

```
2270    public final void writeInt(long v) throws IOException {
2271        long mask = 0xFF;
2272        out.write((byte) (mask & (v >> 24)));
2273        out.write((byte) (mask & (v >> 16)));
2274        out.write((byte) (mask & (v >> 8)));
2275        out.write((byte) (mask & v));
2276    }
2277
2278}
2279
```