Barista – version 1.4

http://barista.x9c.fr

February 6, 2010

Abstract

This document presents Barista, the library as well as the application, its purpose and the way it works. This document consists of six chapters and an appendix. After a first chapter providing an overview of the Barista project, the second chapter explains how Barista can be built from sources and details its dependencies. Then chapters three, four, and five describe how Barista can be used as an Objective Caml library, as a Java library, and as a command-line program. The last chapter shed light on the format of source file used by the Barista assembler when used as a command-line program. Finally, an appendix summarizes all the instructions recognized by the Barista assembler.

Contents

Chapter 1

Overview

Barista is initially an Objective Caml[1] library designed to load, construct, manipulate and save Java[2] class files. The library supports the whole class file format as defined by Sun. The Barista library is used in the OCaml-Java project (available at http://ocamljava.x9c.fr) for code generation.

Since version 1.0β, Barista also features a Java API allowing to use Barista directly from Java. This API only exposes Java definitions of class file elements but the actual treatment is done by the Objective Caml library, after conversion of elements from their Java representation into their Objective Caml representation. This of course results in a performance penalty but alllows to leverage the robustness of the Objective Caml library. More, the Java API invokes the Objective Caml code when compiled to Java classes using the Cafesterol compiler from the OCaml-Java project; this means that the Java API relies upon a bootstraping of Barista.

A command-line utility (also named "barista") has been developped upon the library: both an assembler and a disassembler for the Java platform. In its 1.4 version, Barista supports Java 1.6 and needs Objective Caml 3.11.2 to build. Code sample 1 shows the canonical basic example coded in the Barista assembler. Chapter 6 describes the format of assembler sources.

Code sample 1 The classical "hello world" in Barista assembler.

```
.class public final pack.Test
.extends java.lang.Object

.method public static void main(java.lang.String[])
        getstatic java.lang.System.out : java.io.PrintStream
        ldc "hello world.\n"
        invokevirtual java.io.PrintStream.println(java.lang.String):void
```

This document assumes familiarity with Java in general and with the class file format in particular; one should refer to the documentation published by Sun for more information (a

[1]The official Caml website can be reached at http://caml.inria.fr and contains the full development suite (compilers, tools, virtual machine, *etc.*) as well as links to third-party contributions.

[2]The official Java website can be reached at http://java.sun.com where most of official Java information can be found.

good entry point for the JVM specification being http://java.sun.com/docs/books/jvms/).

In order to improve the project, I am primarily looking for testers and bug reporters. Pointing errors in documentation and indicating where it should be enhanced is also very helpful.
Bug reports can be made at http://bugs.x9c.fr.
Other requests can be sent to barista@x9c.fr.

Chapter 2

Building Barista from sources

This chapter presents the steps to follow in order to build the various parts of Barista from sources. First, the dependencies for the various parts are listed. Then, the available targets are presented for both the `Makefile` and the `build.xml` files.

2.1 Dependencies

To compile the OCaml sources of Barista, one needs Objective Caml 3.11.2, and GNU make 3.81. To compile the Java sources of Barista, one needs Java 1.6, as well as Ant 1.7.1. Cafesterol[1] 1.4 and Cadmium[2] 1.4 are also needed to compile the Java version of Barista. Additionally, Barista depends upon the following Objective Caml libraries that should hence be installed before build:

- **bigarray**, **str**, **unix** available in the Objective Caml standard distribution;
- **camlzip** (zip/gzip/jar library), at least version 1.04 available at: `http://cristal.inria.fr/~xleroy/software.html`;
- **camomile** (Unicode library), at least version 0.7.2 available at: `http://camomile.sourceforge.net`.

Although not dependent upon, Barista can take advantage of the presence of Bisect to generate coverage information for the test suite shipped with the sources. Bisect, a code coverage tool for the Objective Caml language is available at `http://bisect.x9c.fr`.

2.2 Executable and Objective Caml library

Before invoking `make`, one is advised to edit `Makefile` to check that the paths and definitions match the host configuration. The variables to check are:

- `PATH_OCAML_BIN` that should point to the directory containing the Objective Caml executables (`ocamlc`, `ocamlopt`, *etc.*);
- `CLASSPATH` that should define a Java classpath containing the JDK classes;

[1] Cafesterol, the OCaml to Java compiler is itself based on Barista and is available at `http://cafesterol.x9c.fr`.

[2] Cadmium, the Java-based interpreter and runtime support for Objective Caml programs is available at `http://cadmium.x9c.fr`.

- `JAVA` that should give the command line to be used to execute the Java virtual machine;
- `JAVAC` that should give the command line to be used to execute the Java compiler;
- `JAVAP` that should give the command line to be used to execute the Java decompiler shipped with the JDK;
- `ANT` that should give the command line to be used to execute the Ant tool.

The following targets are available:

`clean`, `clean-doc`, `clean-coverage`, `clean-all` respectively deletes produced binary files, produced documentation files, produced coverage information and all produced files;

`bytecode` compiles the bytecode version of Barista;

`native` compiles the native version of Barista;

`all` compiles both bytecode and native versions of Barista and generates ocamldoc documentation;

`cafesterol` compiles the Java version of Barista;

`html-doc` generates ocamldoc documentation;

`install-library` copies the library files into the Objective Caml directory;

`install-executables` copies the executable files into the binary directory;

`install-all` copies both library and executable files;

`install-cafesterol` copies files generated by Cafesterol;

`install-ocamlfind` installs Barista through `ocamlfind`;

`tests` runs both unit and functional tests;

`report` uses Bisect to generate the coverage report (the test suit should have been run on a Bisect-instrumented version of Barista);

`depend` generates both `depend` and `depend.cafesterol` dependency files to enable `make` depency-based builds.

To build a Bisect-instrumented version of the library, one should pass a non-empty value to the `COVERAGE` variable. This means that an instrumented version can be build by issuing the command `make COVERAGE=ON` *`targets`*. Then the test suite can be run by `make COVERAGE=ON tests`. Finally, the coverage report is generated by `make report`.

2.3 Java library

To be able to compile the Java sources, it is needed to have successfully run `make cafesterol`, and to have a working installation of the Cadmium project. Before invoking `ant`, one is advised to edit `build.properties` to check that the paths and definitions match the host configuration. The variables to check are:

- `path.ocaml` that should point to the directory containing the Objective Caml executables (`ocamlc`, `ocamlopt`, *etc.*);
- `path.jdkdoc` that should point to the directory containing the Javadoc documentation of the JDK;
- `path.cadmiumdoc` that should point to the directory containing the Javadoc documentation of the Cadmium project.

The following targets are available:

`clean-classes`, `clean-javadoc`, `clean` respectively deletes produced binary files, produced documentation files and all produced files;

`compile` compiles the Java files;

`deploy` compiles the Java files and then creates the `jar` file;

`javadoc` generates Javadoc documentation.

Chapter 3

Using Barista as an Objective Caml library

3.1 Overview of the library

The complete documentation for the library can ne generated by issuing `make html-doc` after a successful compilation. The produced documentation is located in the `ocamldoc` subdirectory. The modules classify in three categories: utility modules, modules implementing the features of the Barista command-line program, and modules representing class file elements.

The utility modules are:

- `Utils` providing Unicode functions (based on the Camomile library);
- `InputStream` providing various implementation of input streams;
- `OutputStream` providing various implementation of output streams;
- `Consts` providing the Unicode constants for the whole project.

The modules implementing the features of the Barista command-line program are:

- `Source` providing functions for source file handling;
- `Printer` providing the function called by the `-print` command-line switch;
- `Disassembler` providing the function called by the `-dasm` command-line switch;
- `Lexer` providing the function performing lexical analysis of a source line;
- `Assembler` providing the function called by the `-asm` command-line switch.

The `ClassPath` and `ClassLoader` modules allow easy loading of class definitions. The following modules provide the base elements to be found in a class definition:

- `Name` defines the names for the various kinds of Java elements;
- `Descriptor` defines the descriptors (*i.e.* type informations) for the various kinds of Java elements;

- `Signature` defines the signature (*i.e.* type informations for *generic* types) for the various kinds of Java elements;
- `AccessFlag` defines the access flags for the various kinds of Java elements;
- `ConstantPool` defines the constant pools used for decoding/encoding of class files.

Additionally, the other modules provide types representing the class file elements. Each class file element comes in two flavours: a low-level form (as close as possible to the class file format) and a high-level form (as expressive as can be). Table 3.1 gives the types for these elements.

Element	**Low-level form**	**High-level form**
Annotations	`Annotation.info`	`Annotation.t`
Attributes	`Attribute.into`	`Attribute.t`
Instructions	`ByteCode.t`	`Instruction.t`
Fields	`Field.info`	`Field.t`
Methods	`Method.info`	`Method.t`
Classes	`ClassFile.t`	`ClassDefinition.t`

Table 3.1: Mapping of Java elements to Objective Caml Barista types.

Finally, the module `StackState` is provided for manipulation of locals and operand stacks, while `ControlFlow` and `Code` respectively allow to build a control flow graph for a method and to compute stack elements (`max_locals`, `max_stack`, and stack frames) from such a graph.

3.2 Compilation of a Barista-based source file

To use Barista as a library, it is sufficient to link the program with either `baristaLibrary.cma` (bytecode version), `baristaLibrary.cmxa` (native version), or `baristaLibrary.cmja` (Cafesterol-compiled version). These libraries are crafted in such a way that there is only one top-level module called `BaristaLibrary` that contains all the modules described by the ocamldoc-generated documentation.

Code sample 2 shows an example of an Objective Caml program using Barista to produce a file named `Hello.class` containing the definition of a class named `Hello`. This class contains only a `main` method that prints “Hello” on the standard output.

In order to compile the source file represented by code sample 2, one of the following commands should be used:

- `ocamlc -I +barista -I +zip bigarray.cma camomile.cma unix.cma zip.cma str.cma baristaLibrary.cma source.ml`
- `ocamlopt -I +barista -I +zip bigarray.cmxa camomile.cmxa unix.cmxa zip.cmxa str.cmxa baristaLibrary.cmxa source.ml`
- `ocamljava -I +barista -I +zip bigarray.cmja camomile.cmja unix.cmja zip.cmja str.cmja baristaLibrary.cmja source.ml`

Code sample 2 Using Barista in an Objective Caml program.

```
open BaristaLibrary

let utf8 = Utils.utf8_of_string
let utf8_for_class x = Name.make_for_class_from_external (utf8 x)
let utf8_for_field x = Name.make_for_field (utf8 x)
let utf8_for_method x = Name.make_for_method (utf8 x)

let () =
  let instructions = [
    Instruction.GETSTATIC ((utf8_for_class "java.lang.System"),
                           (utf8_for_field "out"),
                           (`Class (utf8_for_class "java.io.PrintStream")));
    Instruction.LDC (`String (utf8 "hello."));
    Instruction.INVOKEVIRTUAL (`Class_or_interface (utf8_for_class "java.io.PrintStream"),
                               (utf8_for_method "println"),
                               ([(`Class (utf8_for_class "java.lang.String"))],
                                `Void));
    Instruction.RETURN;
  ] in
  let code = {
    Attribute.max_stack = 2;
    Attribute.max_locals = 1;
    Attribute.code = instructions;
    Attribute.exception_table = [];
    Attribute.attributes = [];
  } in
  let main_method =
    Method.Regular ([`Public; `Static],
                    (utf8_for_method "main"),
                    ([`Array (`Class (utf8_for_class "java.lang.String"))], `Void),
                    [`Code code]) in
  let hello = {
    ClassDefinition.access_flags = [`Public; `Super];
    ClassDefinition.name = utf8_for_class "Hello";
    ClassDefinition.extends = Some (utf8_for_class "java.lang.Object");
    ClassDefinition.implements = [];
    ClassDefinition.fields = [];
    ClassDefinition.methods = [main_method];
    ClassDefinition.attributes = [];
  } in
  let cf = ClassDefinition.encode hello in
  ClassFile.write cf (OutputStream.make_of_channel (open_out "Hello.class"))
```

Chapter 4

Using Barista as a Java library

4.1 Overview of the library

The complete documentation for the library can be generated by issuing `ant javadoc`. The produced documentation is located in two subdirectories: `javadoc/public` for the documentation of public elements, and `javadoc/dev` for a comprehensive documentation (including private elements). The classes of the `fr.x9c.barista.api` package classify in three categories: Ant tasks, classes implementing the features of the Barista command-line program, and classes representing class file elements.

The Ant task classes are:

- `AntAssembleTask` providing the task for assembling a class;
- `AntDisassembleTask` providing the task for disassembling a class;
- `AntPrintTask` providing the task for printing information about a class.

The classes implementing the functionalities of the command-line program are:

- `Disassembler` providing the function called by the `-dasm` command-line switch;
- `Printer` providing the function called by the `-print` command-line switch;
- `Assembler` providing the function called by the `-asm` command-line switch.

Finally, the other classes provide represent the class file elements:

- `AccessFlag` defines the access flags for the various kinds of Java elements;
- `Annotation` and inner-classes define the annotations for the various kinds of Java elements;
- `Attribute` and inner-classes define the attributes for the various kinds of Java elements;
- `Desciptor` and inner-classes define the descriptors for the various kinds of Java elements;
- `Instruction` and inner-classes define the instructions for the methods;
- `Field` defines a field;
- `Method` defines a method;
- `ClassDefinition` defines a whole class.

4.2 Compilation of a Barista-based source file

To use Barista as a library, it is sufficient to have both `barista.jar` and `barista-api.jar` in the classpath. Code sample 3 shows an example of a Java program using Barista to produce a file named `Hello.class` containing the definition of a class named `Hello`. This class contains only a `main` method that prints “Hello” on the standard output.

Code sample 3 Using Barista in a Java program.

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

import fr.x9c.barista.api.API;
import fr.x9c.barista.api.AccessFlag;
import fr.x9c.barista.api.Attribute;
import fr.x9c.barista.api.ByteCode;
import fr.x9c.barista.api.ClassDefinition;
import fr.x9c.barista.api.Descriptor;
import fr.x9c.barista.api.Field;
import fr.x9c.barista.api.Instruction;
import fr.x9c.barista.api.Method;

public final class Source {

    public static void main(final String[] args) throws IOException {
        final Descriptor printStream =
            new Descriptor.Class("java.io.PrintStream");
        final Instruction.FieldRef systemOut =
            new Instruction.FieldRef("java.lang.System", "out", printStream);
        final Instruction.MethodRef println =
            new Instruction.MethodRef("java.io.PrintStream",
                                      "println",
                                      Descriptor.VOID,
                                      Arrays.asList(new Descriptor.Class("java.lang.String")));
        final Attribute.CodeValue codeValue =
            new Attribute.CodeValue(4,
                                    4,
                                    Arrays.asList(new Instruction.GETSTATIC(systemOut),
                                                  new Instruction.LDC_STRING("hello."),
                                                  new Instruction.INVOKEVIRTUAL(println),
                                                  Instruction.RETURN),
                                    Collections.<Attribute.ExceptionTableElement>emptyList(),
                                    Collections.<Attribute>emptyList());
        final Attribute code = new Attribute.Code(codeValue);
        final Method mainMethod =
            new Method(Arrays.asList(AccessFlag.Public, AccessFlag.Static),
                       "main",
                       Descriptor.VOID,
                       Collections.singletonList(new Descriptor.Array(new Descriptor.Class("java.lang.String"))
                       Collections.singletonList(code));
        final ClassDefinition cd =
            new ClassDefinition(Arrays.asList(AccessFlag.Public, AccessFlag.Super),
                                "Hello",
                                "java.lang.Object",
                                Collections.<String>emptyList(),
                                Collections.<Field>emptyList(),
                                Collections.singletonList(mainMethod),
                                Collections.<Attribute>emptyList());
        final byte[] bc = ByteCode.encode(cd);
        FileOutputStream out = new FileOutputStream("Hello.class");
        out.write(bc);
        out.flush();
    }

}
```

Chapter 5

Using Barista as a command-line program

The Barista program exists under three forms: `barista`, `barista.opt`, and `barista.jar`. The first one is an Objective Caml bytecode file, the second one is a native executable, and the third one is an executable jar file. All three programs should behave the same way, only differing in execution speed. The Barista executables recognize the following command-line switches:

`-help` shows a short usage help;

`-version` prints the version and exits;

`-cp` *cp* appends *cp* to the classpath;

`-classpath` *cp* sets the classpath to *cp*;

`-output` *d* sets the output path for generated class files to *d*;

`-target` *v* sets the target version for generated class files to *v*;

`-compute-stacks` automatically compute stack information[1] when not provided in source;

`-print` *c* prints a short description of the class named *c*;

`-asm` *f* assembles the file named *f* and produces the corresponding class file;

`-dasm` *c* disassembles the class named *c* to the Barista format;

`-flow` *m* print the control flow graph for the method *m* (using dot format[2]);

`-api` registers the API for the Java library (internal use).

Where:

cp is a colon-separated classpath;

d should point to a directory;

f should point to a file;

[1]`max_locals`, `max_stack`, as well as stack frames.
[2]`http://www.graphviz.org/doc/info/lang.html`

c is a fully-qulified class name;

v is a Java version among `1.0`, `1.1`, `1.2`, `1.3`, `1.4`, `1.5`, `1.6`, `1.7`;

m is a method reference (*cf.* section 6.1).

The other elements of the command line are interpreted as preceded by `-asm` if they end with `.j`, otherwise they are interpreted as preceded by `-print`. The default classpath is set to the value of the environment variable named `CLASSPATH` if it exists, to `.` (dot) otherwise.

Chapter 6

Assembler source format

This chapter describes the format of the source files that the Barista assembler accepts. The very same format is used by the Barista disassembler as its output format. This makes it easy to disassemble a class, modify the disassembled source and then reassemble it.

6.1 Lexical elements

The source files used by the Barista assembler are line-oriented (this means that source elements are analyzed line by line). Over a line, elements are whitespace-separated, meaning that whitespaces are meaningful. The different lexical elements are:

Comments, introduced by the `#` (sharp) character and ending at the end of the line

Assembler directives, introduced by the `.` (dot) character followed by a non-empty sequence of lowercase letters or underscores *e.g.* `.class`

Element attributes, introduced by the `@` (at) character followed by a non-empty capitalized sequence of letters *e.g.* `@ConstantValue`

Labels, non-empty sequences of letters and digits beginning with a letter and ending with a colon *e.g.* `aLabel3:`

Integer constants, following the Objective Caml conventions and thus supporting decimal notation (*e.g.* `15`), hexadecimal notation (*e.g.* `0x0F`), octal notation (*e.g.* `0o17`) and binary notation (*e.g.* `0b1111`). All notations support embedded underscore characters used for increased readability (such underscore characters are ignored). Integer constants can also be defined using the character notation (*e.g.* `'a'`); this notation supports escaped sequence for ASCII characters (*e.g.* `'\t'`, `'\n'`, or `'\\'`) as well as for Unicode character in short (*e.g.* `'\u1234'`) and long formats (*e.g.* `'\U12345678'`)

Floating-point constants, following the Objective Caml conventions and thus consisting of three part: an integral part, a decimal part and an exponent part (*e.g.* `-1.234e+2`). Embedded underscore characters can be used for readability

String constants, between `"` quotes, supporting the escaped sequences presented for integer character constants *e.g.* `"abc\tdef"`

Class names, given in *external* format using dots between packages/classes and dollars between inner classes (*e.g.* `pack.Cls$Inn` for an inner-class *Inn* of a class named *Cls* in package *pack*)

Array types, either a class name or a primitive type name followed by a non-empty list of `[]` pairs (*e.g.* `int[][]` or `java.lang.String[]`)

Field references, defined by a qualified field name followed by a colon and the field type *e.g.* `java.lang.String.CASE_SENSITIVE_ORDER:java.util.Comparator`

Dynamic method references[1], defined by an unqualified method name followed by the parameters between paretheses, then followed by a colon and the return type
e.g. `toString():java.lang.String`

Method references, defined by a qualified method name followed by the parameter types between parentheses, then followed by a colon and the return type
e.g. `java.lang.Object.toString():java.lang.String` or `int[].toString():java.lang.String`

Method signatures, defined by an unqualified method name followed by the parameter types between parentheses *e.g.* `toString()`

Arrows, that are simply the character sequence `=>`

Tildes, that are simply the character `~`

6.2 Assembler directives

The assembler directives define the elements of a Java class as well as their main properties; the following directives are recognized:

`.class` *flags name* where *flags* is a list of class flags[2] (among `public`, `final`, `super`, `interface`, `abstract`, `synthetic`, `annotation`, `enum`), and *name* is a fully qualified class name. This directive should be the first one of the source file.

`.extends` *name* where *name* is a fully qualified class name. This directive sets the class parent and should be present even if the parent is `java.lang.Object`; this directive may be missing if and only if the class defined by the source file has no parent (it should only be true for the `java.lang.Object` class).

`.implements` *name* where *name* is a fully qualified class name (that should be an interface).

`.field` *flags type name* where *flags* is a list of field flags (among `public`, `private`, `protected`, `static`, `final`, `volatile`, `transient`, `synthetic`, `enum`), *type* is either a primitive, a fully qualified class name or an array type, and *name* is a field name. This directive adds a field to the class.

[1] Dynamic method calls are not currently supported, as Java 1.5 does not defines such a concept. The present implementation is based on the instruction list published by Sun for Java 1.6, this information being an early draft to be refined before implementation in Java 1.7.

[2] One should notice that flags discussed in this document are JVM-level flags, and not Java language-level flags. Hence the presence of flag that cannot occur in Java sources.

`.method` *flags returntype signature* where *flags* is a list of method class flags (among `public`, `private`, `protected`, `static`, `final`, `synchronized`, `bridge`, `varargs`, `native`, `abstract`, `strict`, `synthetic`), *returntype* is either a primitive, a fully qualified class name, or an array type, and *signature* is a method signature. This directive adds a method to the class.

`.max_stack` n where n is an integer in the interval from 0 to 65535 (both inclusive). This directive sets the maximum stack size for the current method.

`.max_locals` n where n is an integer in the interval from 0 to 65535 (both inclusive). This directive sets the maximum local size for the current method.

`.catch` *start end handler* [*name*] where *start*, *end* and *handler* are labels refering to respectively the begin and the end of the protected code area, and to the associated exception handler. The optional *name* is a fully qualified class name giving the name of the exception class to be handled by the code located at handler label; if this name is missing, all exceptions will be caught.

`.frame` l def where l is the label where the frame definition def applies. This directive allows to specify elements for the `StackMapFrame` attributes of the class file. The frame definition (noted def above) can take of of the following forms:

- `same` to indicate that the frame definition is the same as the previous one;
- `same_locals` t to indicate that the frame definition is the same as the previous one regarding locals and has one element on the stack whose type is t;
- `chop` n to indicate that the frame definition is the same as the previous one, chopped by n elements (n should be `1`, `2`, or `3`);
- `append` t_1 [t_2 [t_3]] indicate that the frame definition is the same as the previous one, with one to three elements appended (whose types are given by the t_i);
- `full` t_1 ... t_n ~ t'_1 ... t'_m to define a frame with no reference to the previous one, the t_i are the types for locals while the t'_j ones are the types for stack elements (the tilde symbol separating the two lists);

The t, t_i, and t'_j are type definitions whose possible values are:

- `top` for the *top* type;
- `int` for the `int` type;
- `float` for the `float` type;
- `long` for the `long` type;
- `double` for the `double` type;
- `null` for the type associated to the *null* value;
- `uninit_this` for the uninitialized *this* reference;
- `uninit` l for an uninitialized reference whose related `new` instruction is located at the offset given by label l;
- a fully-qualified class frame for the type associated to this class.

6.3 Attributes

The assembler attributes define the properties of a Java element. An attribute is applied to the latest defined Java element by a `.class`, `.field` or `.method` directive. The following attributes are recognized:

`@ConstantValue` *value* (fields only)
defines the initial value of the field. *value* should be compatible with the field type and can be a float, an integer or a string. A `false` boolean is coded by a `0` integer while every other values code a `true` boolean

`@Exceptions` *non-empty-list* (methods only)
defines the list of exceptions (as fully qualified class names) that the method can throw

`@InnerClasses` *ic oc n flags* (classes only)
adds an inner-class information. *ic* is the fully qualified name of the inner-class (or `0` if this information is missing), *oc* is the fully qualified name of the outer-class (or `0` if this information is missing), *n* is the name of the inner-class in the outer-class (or `0` if this information is missing), *flags* is a list of inner-class flags (among `public`, `private`, `protected`, `static`, `final`, `super`, `interface`, `abstract`, `synthetic`, `annotation`, `enum`)

`@EnclosingMethod` *name meth* (classes only)
adds an enlosing-method information. *name* is the fully qualified name of the enclosed class, *meth* is the enclosing method specified as a dynamic method (or `0` if this information is missing)

`@Synthetic` (classes, fields, methods)
marks the element as synthetic (*i.e.* compiler-generated)

`@Signature` *string* (classes, fields, methods)
sets the signature of the element

`@SourceFile` *string* (classes only)
sets the source file name

`@SourceDebugExtension` *string* (classes only)
sets the source debug extension

`@Deprecated` (classes, fields, methods)
marks the element as deprecated

`@RuntimeVisibleAnnotations` *elems* (classes, fields, methods)
adds an annotation to the element (the format of annotations is discussed below)

`@RuntimeInvisibleAnnotations` *elems* (classes, fields, methods)
adds an annotation to the element (the format of annotations is discussed below)

`@RuntimeVisibleParameterAnnotations` *n elems* (methods only)
adds an annotation to the element for the parameter at index *n* (the format of annotations is discussed below)

`@RuntimeInvisibleParameterAnnotations` *n elems* (methods only)
adds an annotation to the element for the parameter at index *n* (the format of annotations is discussed below)

`@AnnotationDefault` *elems* (methods only)
adds an annotation default to the element (the format of annotation defaults is discussed below)

`@LineNumberTable` [*n*] (methods only)
maps the current code offset to a line number. The line number is *n* if provided, otherwise it is the current line of the Barista source file

`@LocalVariableTable` *start end id t idx* (methods only)
adds a type information for a local variable. *id* and *t* are the variable identifier and type, *idx* is its position in the locals, and *start* (inclusive) and *end* (exclusive) are labels defining the portion of code where this variable is defined

`@LocalVariableTypeTable` *start end id s idx* (methods only)
adds a type information for a local variable. *id* and *s* are the variable identifier and signature, *idx* is its position in the locals, and *start* (inclusive) and *end* (exclusive) are labels defining the portion of code where this variable is defined

`@Unknown` *string*1 *string*2 (classes, fields, methods)
adds an unknown (*i.e.* implementation-dependent) attributes to the element. *string*1 is the identifier of the attribute while *string*2 is its value

6.4 Annotations

Annotations with their key-value attributes and possibly embedded annotations form a tree-like structure. The Barista assembler sources being line-oriented, annotations are organized in a way such that lines with the same prefix gives informations for the same subtree.
On a line defining an annotation, the first element should be the class of the annotation. This fully qualified class name is followed by the key identifier and its associated value. The value is itself a couple; the first component is the value type while the second component is the actual value. Code sample 4 shows an annotation (whose class is `pack.AnnotationClass`) with two parameters:

- parameter `a` whose type is *string* and value `"xyz"`;
- parameter `b` whose type is *float* and value `3.14`.

Code sample 4 Annotation example.

```
@RuntimeVisibleAnnotation pack.AnnotationClass a string "xzy"
@RuntimeVisibleAnnotation pack.AnnotationClass b float 3.14
```

The possible types for an annotation attribute are the Java primitive types, extended with the following ones:

`string` for string values (using the format of string constants discussed above);

`enum` for enum values (defined by the fully qualified name of the enum class followed by the identifier of the enum value);

`class` for reference to a given class (followed by the fully qualified name of the class);

`annotation` for embedded annotation (followed by an annotation value using the format explained in this very section);

an array is not introduced by any keyword, each value being introduced by its index[3] (as an integer). The type of each embedded element should be repeated.

Code sample 5 shows the previous annotation enriched with three values:

- parameter `c` whose type is *array* and value is a two-element array containing `5` and `7`;
- parameter `d` whose type is *annotation* and value `java.lang.Deprecated`;
- parameter `e` whose type is *enum* and value `pack.EnumClass.E1`.

Code sample 5 Annotation example.

```
@RuntimeVisibleAnnotation pack.AnnotationClass a string "xzy"
@RuntimeVisibleAnnotation pack.AnnotationClass b float 3.14
@RuntimeVisibleAnnotation pack.AnnotationClass c 0 int 5
@RuntimeVisibleAnnotation pack.AnnotationClass c 1 int 7
@RuntimeVisibleAnnotation pack.AnnotationClass d annotation java.lang.Deprecated
@RuntimeVisibleAnnotation pack.AnnotationClass e enum pack.EnumClass E1
```

6.5 Annotation defaults

Annotation defaults closely follow the notation used for annotation, except that leading annotation class name as well as attribute name should be omitted. Code sample 6 defines an annotation default whose value is the `"xyz"` string.

Code sample 6 Annotation default example.

```
@AnnotationDefault string "xyz"
```

6.6 Instructions

Instructions may, of course, be used only inside methods. An instruction line may contain either a label, or an instruction (along with its parameters), or both. One should refer to the JVM specification for the list of available instructions; this specification also defines the parameters waited by each instruction. The translation of parameters from the specification to their Barista counterpart is straightforward and only detailed in the appendix (one may also read tests cases for examples). When using the *wide* version of an instruction, one has to use

[3] These indexes do not need to be successive, they are only used to sort the values in order to produce the array.

the `wide` keyword before the instruction.

Almost all instructions are declared on one line but *switch* instructions are multi-line ones. Code sample 7 shows the syntax used by the `tableswitch` and `lookupswitch` instructions. A `tableswitch` instruction accepts three parameters: a label, and lower and upper bounds. The label is the destination for the default case; the following lines define the destination for the lower bound, the lower bound plus one, and so on until the upper bound.
A `lookupswitch` instruction accepts two parameters: a label and a number of matchings. The label is the destination for the default case; the following lines define the matchings in the *value* `=>` *label* form.

Code sample 7 Switch examples.

```
tableswitch default: 0 2
        => zero:
        => one:
        => two:

lookupswitch default: 3
        0 => zero:
        1 => one:
        2 => two:
```

6.7 Example

Code sample 8 shows an example of a Barista source file coding a Java class named `pack.Test`. This class contains two fields (the constants named `PREFIX` and `SUFFIX`) as well as two methods (`print` and `main`). The `main` method prints the classical hello world message and then iterates over the elements of the passed string array by applying the `print` method to each element. In turn, the `print` method prints each string with the prefix and suffix specified by the `PREFIX` and `SUFFIX` field values.

Code sample 8 Example of a Barista-coded class file.

```
.class public final pack.Test
.extends java.lang.Object

.field private static final java.lang.String PREFIX
        @ConstantValue "  - << "

.field private static final java.lang.String SUFFIX
        @ConstantValue " >>"

.method public static void print(java.lang.String)
        getstatic java.lang.System.out:java.io.PrintStream
        dup
        dup
        getstatic pack.Test.PREFIX: java.lang.String
        invokevirtual java.io.PrintStream.print(java.lang.String): void
        aload_0
        invokevirtual java.io.PrintStream.print(java.lang.String) :void
        getstatic pack.Test.SUFFIX :java.lang.String
        invokevirtual java.io.PrintStream.println(java.lang.String) : void
        return

.method public static void main(java.lang.String[])
        nop
        getstatic java.lang.System.out : java.io.PrintStream
        ldc "hello\t... \n\t... \"world\""
        invokevirtual java.io.PrintStream.println(java.lang.String):void

        iconst_0
        istore_1
        aload_0
        arraylength
        istore_2
loop:
        iload_1
        iload_2
        if_icmpeq end:
        aload_0
        iload_1
        aaload
        invokestatic pack.Test.print(java.lang.String):void
        iinc 1 1
        goto loop:
end:
        return
```

Appendix A

Instructions and related parameters

This appendix summarizes in alphabetical order the instructions recognized by the Barista assembler. For a complete description, one should refer to the JVM documentation provided by Sun. For examples of actual uses, one is advised to check the sources used as unit tests (they can be found in the `tests` subdirectory of the Barista source distribution).

aaload load reference from array
no parameter

aastore store into reference array
no parameter

aconst_null push null
no parameter

aload load reference from local variable

parameter #1 local index (unsigned 8-bit integer)

aload_0 load reference from local variable
no parameter

aload_1 load reference from local variable
no parameter

aload_2 load reference from local variable
no parameter

aload_3 load reference from local variable
no parameter

anewarray create new array of references

parameter #1 element type (class name, or array type)

areturn return reference from method
no parameter

arraylength get length of array
no parameter

astore store reference into local variable

parameter #1 local index (unsigned 8-bit integer)

astore_0 store reference into local variable
no parameter

astore_1 store reference into local variable
no parameter

astore_2 store reference into local variable
no parameter

astore_3 store reference into local variable
no parameter

athrow throw exception or error
no parameter

baload load byte or boolean from array
no parameter

bastore store into byte or boolean array
no parameter

bipush push byte

parameter #1 byte value (signed 8-bit integer)

caload load char from array
no parameter

castore store into char array
no parameter

checkcast check whether object is of given type

parameter #1 type to test against (class name, or array type)

d2f convert double to float
no parameter

d2i convert double to int
no parameter

d2l convert double to long
no parameter

dadd add double
no parameter

daload load double from array
no parameter

dastore store into double array
no parameter

dcmpg compare double
no parameter

dcmpl compare double
no parameter

dconst_0 push double
no parameter

dconst_1 push double
no parameter

ddiv divide double
no parameter

dload load double from local variable

parameter #1 local index (unsigned 8-bit integer)

dload_0 load double from local variable
no parameter

dload_1 load double from local variable
no parameter

dload_2 load double from local variable
no parameter

dload_3 load double from local variable
no parameter

dmul multiply double
no parameter

dneg negate double
no parameter

drem remainder double
no parameter

dreturn return double from method
no parameter

dstore store double into local variable

parameter #1 local index (unsigned 8-bit integer)

dstore_0 store double into local variable
no parameter

dstore_1 store double into local variable
no parameter

dstore_2 store double into local variable
no parameter

dstore_3 store double into local variable
no parameter

dsub subtract double
no parameter

dup duplicate the top operand stack value
no parameter

dup2 duplicate the top one or two operand stack values
no parameter

dup2_x1 duplicate the top one or two operand stack values and insert two or three values down
no parameter

dup2_x2 duplicate the top one or two operand stack values and insert two, three, or four values down
no parameter

dup_x1 duplicate the top operand stack value and insert two values down
no parameter

dup_x2 duplicate the top operand stack value and insert two or threee values down
no parameter

f2d convert float to double
no parameter

f2i convert float to int
no parameter

f2l convert float to long
no parameter

fadd add float
no parameter

faload load float from array
no parameter

fastore store into float array
no parameter

fcmpg compare float
no parameter

fcmpl compare float
no parameter

fconst_0 push float
no parameter

fconst_1 push float
no parameter

fconst_2 push float
no parameter

fdiv divide float
no parameter

fload load float from local variable

parameter #1 local index (unsigned 8-bit integer)

fload_0 load float from local variable
no parameter

fload_1 load float from local variable
no parameter

fload_2 load float from local variable
no parameter

fload_3 load float from local variable
no parameter

fmul multiply float
no parameter

fneg negate float
no parameter

frem remainder float
no parameter

freturn return float from method
no parameter

fstore store float into local variable

parameter #1 local index (unsigned 8-bit integer)

fstore_0 store float into local variable
no parameter

fstore_1 store float into local variable
no parameter

fstore_2 store float into local variable
no parameter

fstore_3 store float into local variable
no parameter

fsub subtract float
no parameter

getfield fetch field from object

parameter #1 field to get (field reference)

getstatic get static field from class

parameter #1 field to get (field reference)

goto branch always

parameter #1 destination offset (label)

goto_w branch always

parameter #1 destination offset (label)

i2b convert int to byte
no parameter

i2c convert int to char
no parameter

i2d convert int to double
no parameter

i2f convert int to float
no parameter

i2l convert int to long
no parameter

i2s convert int to short
no parameter

iadd add int
no parameter

iaload load int from array
no parameter

iand boolean AND int
no parameter

iastore store into int array
no parameter

iconst_0 push int constant
no parameter

iconst_1 push int constant
no parameter

iconst_2 push int constant
no parameter

iconst_3 push int constant
no parameter

iconst_4 push int constant
no parameter

iconst_5 push int constant
no parameter

iconst_m1 push int constant
no parameter

idiv divide int
no parameter

if_acmpeq branch if reference comparison succeeds

parameter #1 destination offset (label)

if_acmpne branch if reference comparison succeeds

parameter #1 destination offset (label)

if_icmpeq branch if int comparison succeeds

parameter #1 destination offset (label)

if_icmpge branch if int comparison succeeds

parameter #1 destination offset (label)

if_icmpgt branch if int comparison succeeds

parameter #1 destination offset (label)

if_icmple branch if int comparison succeeds

parameter #1 destination offset (label)

if_icmplt branch if int comparison succeeds

parameter #1 destination offset (label)

if_icmpne branch if int comparison succeeds

parameter #1 destination offset (label)

ifeq branch if int comparison with zero succeeds

parameter #1 destination offset (label)

ifge branch if int comparison with zero succeeds

parameter #1 destination offset (label)

ifgt branch if int comparison with zero succeeds

parameter #1 destination offset (label)

ifle branch if int comparison with zero succeeds

parameter #1 destination offset (label)

iflt branch if int comparison with zero succeeds

parameter #1 destination offset (label)

ifne branch if int comparison with zero succeeds

parameter #1 destination offset (label)

ifnonnull branch if reference not null

parameter #1 destination offset (label)

ifnull branch if reference not null

parameter #1 destination offset (label)

iinc increment local variable by constant

parameter #1 local index (unsigned 8-bit integer)

parameter #2 increment value (signed 8-bit integer)

iload load int from local variable

parameter #1 local index (unsigned 8-bit integer)

iload_0 load int from local variable
no parameter

iload_1 load int from local variable
no parameter

iload_2 load int from local variable
no parameter

iload_3 load int from local variable
no parameter

imul multiply int
no parameter

ineg negate int
no parameter

instanceof determine if object is of given type

parameter #1 type to test against (class name, or array type)

invokedynamic invoke instance method; resolve and dispatch based on class

parameter #1 method to invoke (dynamic method reference)

invokeinterface invoke interface method

parameter #1 method to invoke (method reference)

parameter #2 parameter count (unsigned 8-bit integer)

invokespecial invoke instance method; special handling for superclass, private, and instance initialization method invocations

parameter #1 method to invoke (method reference)

invokestatic invoke a class (static) method

parameter #1 method to invoke (method reference)

invokevirtual invoke instance method; dispatch based on class

parameter #1 method to invoke (method reference)

ior boolean OR int
no parameter

irem remainder int
no parameter

ireturn return int from method
no parameter

ishl shift left int
no parameter

ishr arithmetic shift right int
no parameter

istore store int into local variable

parameter #1 local index (unsigned 8-bit integer)

istore_0 store int into local variable
no parameter

istore_1 store int into local variable
no parameter

istore_2 store int into local variable
no parameter

istore_3 store int into local variable
no parameter

isub subtract int
no parameter

iushr logical shift right int
no parameter

ixor boolean XOR int
no parameter

jsr jump subroutine

parameter #1 destination offset (label)

jsr_w jump subroutine (wide index)

parameter #1 destination offset (label)

l2d convert long to double
no parameter

l2f convert long to float
no parameter

l2i convert long to int
no parameter

ladd add long
no parameter

laload load long from array
no parameter

land boolean AND long
no parameter

lastore store into long array
no parameter

lcmp compare long
no parameter

lconst_0 push long constant
no parameter

lconst_1 push long constant
no parameter

ldc push item from runtime constant pool

parameter #1 value to be pushed (signed 32-bit integer, or float, or string literal, or class name, or array type)

ldc2_w push long or double from runtime constant pool (wide index)

parameter #1 value to be pushed (signed 64-bit integer, or float)

ldc_w push item from runtime constant pool (wide index)

parameter #1 value to be pushed (signed 32-bit integer, or float, or string literal, or class name, or array type)

ldiv divide long
no parameter

lload load long from local variable

parameter #1 local index (unsigned 8-bit integer)

lload_0 load long from local variable
no parameter

lload_1 load long from local variable
no parameter

lload_2 load long from local variable
no parameter

lload_3 load long from local variable
no parameter

lmul multiply long
no parameter

lneg negate long
no parameter

lookupswitch access jump table by key match and jump

parameter #1 default destination offset (label)

parameter #2 number of key, offset pairs (signed 32-bit integer)

parameter #3 key, offset pairs (list of (match, label) pairs)

lor boolean OR long
no parameter

lrem remainder long
no parameter

lreturn return long from method
no parameter

lshl shift left long
no parameter

lshr arithmetic shift right long
no parameter

lstore store long into local variable

parameter #1 local index (unsigned 8-bit integer)

lstore_0 store long into local variable
no parameter

lstore_1 store long into local variable
no parameter

lstore_2 store long into local variable
no parameter

lstore_3 store long into local variable
no parameter

lsub subtract long
no parameter

lushr logical shift right long
no parameter

lxor boolean XOR long
no parameter

monitorenter enter monitor for object
no parameter

monitorexit exit monitor for object
no parameter

multianewarray create new multidimensional array

parameter #1 element type (class name, or array type)

parameter #2 number of dimensions (unsigned 8-bit integer)

new create new object

parameter #1 type to create (class name)

newarray create new arrayhandler

parameter #1 element type (primitive array type)

nop do nothing
no parameter

pop pop the top operand stack value
no parameter

pop2 pop the top one or two operand stack values
no parameter

putfield set field in object

parameter #1 field to set (field reference)

putstatic set static field in class

parameter #1 field to set (field reference)

ret return from subroutine

parameter #1 local index (unsigned 8-bit integer)

return return void from method
no parameter

saload load short from array
no parameter

sastore store into short array
no parameter

sipush push short

parameter #1 short value (signed 16-bit integer)

swap swap the top two operand stack values
no parameter

tableswitch access jump table by index and jump

parameter #1 default destination offset (label)

parameter #2 lower bound (signed 32-bit integer)

parameter #3 higher bound (signed 32-bit integer)

parameter #4 destination offsets (list of labels)

wide aload load reference from local variable

parameter #1 local index (unsigned 16-bit integer)

wide astore store reference into local variable

parameter #1 local index (unsigned 16-bit integer)

wide dload load double from local variable

parameter #1 local index (unsigned 16-bit integer)

wide dstore store double into local variable

parameter #1 local index (unsigned 16-bit integer)

wide fload load float from local variable

parameter #1 local index (unsigned 16-bit integer)

wide fstore store float into local variable

parameter #1 local index (unsigned 16-bit integer)

wide iinc increment local variable by constant

parameter #1 local index (unsigned 16-bit integer)

parameter #2 increment value (signed 16-bit integer)

wide iload load int from local variable

parameter #1 local index (unsigned 16-bit integer)

wide istore store int into local variable

parameter #1 local index (unsigned 16-bit integer)

wide lload load long from local variable

parameter #1 local index (unsigned 16-bit integer)

wide lstore store long into local variable

parameter #1 local index (unsigned 16-bit integer)

wide ret return from subroutine

parameter #1 local index (unsigned 16-bit integer)