Kawa, the Java-based Scheme system

19 November 2003

Per Bothner

Kawa is a Scheme environment, written in Java, and that compiles Scheme code into Java byte-codes.

This documents version 1.7.90, updated 19 November 2003.

See the summary of recent changes (`http://www.gnu.org/software/kawa/NEWS`).

The author of Kawa is Per Bothner (`http://www.bothner.com/per/`) `per@bothner.com`. Kawa is a re-write of Kawa 0.2, which was written by R. Alexander Milowski `alex@milowski.com`.

The Kawa home page (which is currently just an on-line version of this document) is `http://www.gnu.org/software/kawa/`.

The Scheme repository (`http://www.cs.indiana.edu/scheme-repository/home.html`) has various useful information on Scheme. but it is not very actively updated. A new repository has been started at www.schemers.org (`http://www.schemers.org/`). It includes pointer to an online copy of R5RS (`http://www.schemers.org/Documents/Standards/`).

A nice quick introduction to Scheme can be found in Greg Badros's lecture notes (`http://www.cs.washington.edu/education/courses/341/99su/lectures/scheme/`). A more in-depth tutorial which also discusses Scheme implementation is Paul Wilson's "An Introduction to Scheme and its Implementation" (`http://www.cs.utexas.edu/users/wilson/schintro/s toc.html`).

Javadoc generated documentation of the Kawa classes (`http://www.gnu.org/software/kawa/api/`) is also available. The packages `gnu.bytecode` (`http://www.gnu.org/software/kawa/api/gnu/bytecode/`
`gnu.math` (`http://www.gnu.org/software/kawa/api/gnu/math/package-summary.html`),
`gnu.lists` (`http://www.gnu.org/software/kawa/api/gnu/lists/package-summary.html`),
`gnu.xml` (`http://www.gnu.org/software/kawa/api/gnu/xml/package-summary.html`),
`gnu.expr` (`http://www.gnu.org/software/kawa/api/gnu/expr/package-summary.html`),
`gnu.mapping` (`http://www.gnu.org/software/kawa/api/gnu/mapping/package-summary.html`),
and `gnu.text` (`http://www.gnu.org/software/kawa/api/gnu/text/package-summary.html`),
are used by Kawa, and distributed with it, but may be independently useful.

For a technical overview of Kawa, see these `http://www.gnu.org/software/kawa/internals.html`.

For copyright information on the software and documentation, see Chapter 17 [License], page 78.

Kawa is partly sponsored by Brainfood (`http://www.brainfood.com/`).

This package has nothing to do with the Kawa commercial Java IDE (`http://www.macromedia.com/sof`

1 Features

Kawa is a full Scheme implementation. It implements almost all of R5RS (for exceptions see Chapter 5 [Restrictions], page 12), plus some extensions. By default, symbols are case sensitive.

It is completely written in Java. Scheme functions and files are automatically compiled into Java byte-codes. Kawa does some optimizations, and the compiled code runs at reasonable speed.

Kawa uses Unicode internally, and uses the Java facilities to convert files using other character encodings.

Kawa provides the usual read-eval-print loop, as well as batch modes.

Kawa provides a framework for implementing other progressing languages, and comes with incomplete support for CommonLisp, Emacs Lisp, and EcmaScript, and the draft XML Query language (`http://www.gnu.org/software/qexo/`).

Kawa is written in an object-oriented style.

Kawa has builtin pretty-printer support, and fancy formatting.

Kawa supports class-definition facilities, and separately-compiled modules.

Kawa implements the full numeric tower, including infinite-precision rational numbers and complex numbers. It also supports "quantities" with units, such as `3cm`.

You can optionally declare the types of variables.

You can conveniently access Java objects, methods, fields, and classes.

Kawa implements most of the features of the expression language of DSSSL, the Scheme-derived ISO-standard Document Style Semantics and Specification Language for SGML. Of the core expression language, the only features missing are character properties, `external-procedure`, the time-relationed procedures, and character name escapes in string literals. Also, Kawa is not generally tail-recursive. From the full expression language, Kawa additionally is missing `format-number`, `format-number-list`, and language objects. Quantities, keyword values, and the expanded `lambda` form (with optional and keyword parameters) are supported.

Kawa implements the following semi-standard SRFIs (Scheme Request for Implementation (`http://srfi.schemers.org/`)):

SRFI 0: Feature-based conditional expansion construct, using `cond-expand` - see Section 7.1 [Syntax and conditional compilation], page 17.

SRFI 1: List Library (`http://srfi.schemers.org/srfi-1/srfi-1.html`), if `(require 'list-lib)`.

SRFI 4: Homogeneous numeric vector datatypes - see Section 7.11 [Uniform vectors], page 30..

SRFI 6: Basic String Ports - see Section 8.2 [Ports], page 41.

SRFI 8: `receive`: Binding to multiple values - see Section 7.2 [Multiple values], page 18.

SRFI 9: Defining Record Types, using `define-record-type` - see ⟨undefined⟩ [Record types], page ⟨undefined⟩.

SRFI 11: Syntax for receiving multiple values, using `let-values` and `let*-value` - see Section 7.2 [Multiple values], page 18.

SRFI 17: Generalized `set!` - see Section 7.13 [Locations], page 34.

SRFI 23: Error reporting mechanism, using `error` - see Section 7.12 [Exceptions], page 33.

SRFI 25: Multi-dimensional Array Primitives - see Section 7.10 [Arrays], page 27.

SRFI 26: Notation for Specializing Parameters without Currying - see Section 7.5 [Procedures], page 21.

SRFI 28: Basic Format Strings - see Section 8.3 [Format], page 44.

SRFI 30: Nested Multi-line Comments.

2 Getting Kawa

You can get Kawa sources and binaries from the Kawa ftp site `ftp://ftp.gnu.org/pub/gnu/kawa/`, or from a mirror site (`http://www.gnu.org/order/ftp.html`).

The latest release of the Kawa source code is `ftp://ftp.gnu.org/pub/gnu/kawa/kawa-1.7.90.tar.gz`. The same sources are available as a zip file `ftp://ftp.gnu.org/pub/gnu/kawa/kawa-1.7.90-src.zip`.

A ready-to-run `.jar` archive of the pre-compiled classes is in `ftp://ftp.gnu.org/pub/gnu/kawa/kawa-1`

You can also check out the very latest version via anonymous cvs.

```
cvs -d :pserver:anoncvs@sources.redhat.com:/cvs/kawa login
     (password is ‘‘anoncvs’’)
cvs -d :pserver:anoncvs@sources.redhat.com:/cvs/kawa co kawa
```

Once you have it checked out, you can update it with `cvs update`.

You can also view the cvs archive (`http://sources.redhat.com/cgi-bin/cvsweb.cgi/kawa/?cvsroot`) via cvsweb.

3 Building and installing Kawa

Before installing Kawa, you must have Java working on your system.

You can compile Kawa from the source distribution. Alternatively, you can install the pre-compiled binary distribution.

3.1 Getting and running Java

You will need a working Java system. Kawa has been reported to work with JDK from 1.1 through 1.4.x, Kaffe, Symantec Cafe, J++, and GCJ.

The discussion below assumes you are using the Java Developer's Kit (JDK) from JavaSoft (Sun). You can download free copies of JDK 1.4 (`http://java.sun.com/j2se/1.4/`) for various platforms.

If you want to run Kawa on a Macintosh, see `http://home.earthlink.net/%7Eathene/scheme/mackawa`

The program `java` is the Java interpreter. The program `javac` is the Java compiler, and is needed if you want to compile the source release yourself. Both programs must be in your `PATH`. If you have the JDK in directory `$JDK`, and you are using a Bourne-shell compatible shell (/bin/sh, ksh, bash, and some others) you can set `PATH` thus:

```
PATH=$JDK/bin:$PATH
export PATH
```

3.2 Installing and using the binary distribution

The binary release includes only the binary compiled '`.class`' versions of the same '`.java`' source files in the source release. It does not include any documentation, so you probably want the source release in addition to the binary release. The purpose of the binary release is just to save you time and trouble of compiling the sources.

The binary release depends on certain "Java 2" features, such as collections. If you have an older Java implementation (including JDK 1.1.x) you will need to get the source distribution.

The binary release comes as a `.jar` archive '`kawa-1.7.90.jar`'.

You can unzip the archive, or you can use it as is. Assuming the latter, copy the archive to some suitable location, such as `/usr/local/lib/kawa.jar`.

Then, before you can actually run Kawa, you need to set `CLASSPATH` so it includes the Kawa archive. On Unix, using a Bourne-style shell:

```
CLASSPATH=/usr/local/lib/kawa.jar
export CLASSPATH
```

On Windows you need to set `classpath` in a DOS console. For example:

```
set classpath=\kawa\kawa-1.7.90.jar
```

Then to run Kawa do:

```
java kawa.repl
```

To run Kawa in a fresh window, you can do:

```
java kawa.repl -w
```

3.3 Installing and using the source distribution

The Kawa release normally comes as a gzip-compressed tar file named '`kawa-1.7.90.tar.gz`'. The same sources are available as a zip file '`kawa-1.7.90-src.zip`'. Two methods are supporting for compiling the Kawa sources; choose whichever is most convenient for you.

One method uses the traditional GNU `configure` script, followed by running `make`. This works well on Unix-like systems, such as GNU/Linux. It does not work well under Microsoft Windows. (Even when using the CygWin Unix-emulation package there are some problems with file paths.)

The other method uses the `ant` command, a Java-based build system released by Apache's Jakarta project. This uses an `build.xml` file in place of `Makefile`s, and works on non-Unix systems such as Microsoft Windows. However, the `ant` method does not support all the features of the `configure+make` method.

3.3.1 Build Kawa using `configure` and `make`

In your build directory do:

```
tar xzf kawa-1.7.90.tar.gz
cd kawa-1.7.90
```

Then you must configure the sources. This you can do the same way you configure most other GNU software. Normally you can just run the configure script with no arguments:

```
./configure
```

This will specify that a later `make install` will install the compiled '`.class`' files into `/usr/local/share/java`. If you want them to be installed someplace else, such as `$PREFIX/share/java`, then specify that when you run configure:

```
./configure --prefix $PREFIX
```

If you have the GNU '`readline`' library installed, you might try adding the '`--enable-kawa-frontend`' flag. This will build the '`kawa`' front-end program, which provides input-line editing and an input history. You can get '`readline`' from archives of GNU programs, including `ftp://www.gnu.org/`.

If you have Swing installed, and want to use JEmacs (Emacs in Java), also pass the `--with-swing` flag to `configure`.

If you have installed Kawa before, make sure your `CLASSPATH` does not include old versions of Kawa, or other classes that may conflict with the new ones.

If you use a very old or bare-bones Java implementation that not have certain "Java 2" features (such as `java.util.List`, `java.lang.ref`, or `ThreadLocal`) then you need to convert the Kawa source-code so it doesn't depend on those features. You do this with the following command:

```
make select-java1
```

Most people should not need to do this. (You don't need to if you're using GCJ, even though it doesn't implement all of Java 2.) (If you need to convert the code back to the default, do: `make select-java2`.)

Then you need to compile all the .java source files. Just run make:

```
make
```

This assumes that '`java`' and '`javac`' are the java interpreter and compiler, respectively. For example, if you are using the Kaffe Java interpreter, you need to instead say:

```
make JAVA=kaffe
```

You can now test the system by running Kawa in place:

```
java kawa.repl
```

or you can run the test suite:

```
make check
```

or you can install the compiled files:

```
make install
```

This will install your classes into `$PREFIX/share/java` (and its sub-directories). Here `$PREFIX` is the directory you specified to configure with the `--prefix` option, or `/usr/local` if you did not specify a `--prefix` option.

To use the installed files, you need to set `CLASSPATH` so that `$PREFIX/share/java/kawa.jar` is in the path:

```
CLASSPATH=$PREFIX/share/java/kawa.jar
export CLASSPATH
```

This is done automatically if you use the '`kawa`' script.

3.3.2 Build Kawa using `ant`

Kawa now includes an Ant buildfile (`build.xml`). Ant (`http://jakarta.apache.org/ant/`) is a part of the Apache Jakarta project. If you don't hava Ant installed, get it from `http://ant.apache.org/bindownload.cgi`. The buildfile should work with Ant 1.3, and

has been tested with 1.4.1. and 1.5.1. The build is entirely Java based and works equally well on *nix, Windows, and presumably most any other operating system.

Once Ant has been installed and configured (you may need to set the `JAVA_HOME`, and `ANT_HOME` environment variables), you should be able to change to the directory containing the `build.xml` file, and invoke the '`ant`' command. With the default settings, a successful build will result in a `kawa-1.7.90.jar` in the current directory

There are a few Ant "targets" of interest (they can be supplied on the Ant command line):

`all` This is the default, it does `classes` and `jar`.

`classes` Compiles all the files into `*.class` files into the directory specified by the `build.dir` property.

`jar` Builds a jar into into the directory specified by the `dist.dir` property.

`runw` Run Kawa in a GUI window.

`clean` Deletes all files generated by the build, including the jar.

There is not yet a `test` target for running the testsuite.

The are various "properties" that control what `ant` does. You can override the on the command line or by editing the `build.properties` file in the same directory as `build.xml`. For example the `build.dir` directory tells `ant` where to build temporary files, and where to leave the resulting `.jar` file. For example, to leave the generated files in the sub-directory named `BUILD` do:

```
ant -Dbuild.dir=BUILD
```

A sample `build.properties` is provided and it contains comments explaining many of the options.

Here are a few general properties that help to customize your build:

`build.dir`
Path to put the temporary files used for building.

`dist.dir` Path to put the resulting jar file.

`version.local`
A suffix to add to the version label for your customized version.

`debug` Whether (true/false) the Javac "-g" option is enabled.

`optimize` Whether (true/false) the Javac "-O" option is enabled.

Here are some Kawa-specific ones (all `true`/`false`): `with-collections`, `with-references`, `with-awt`, `with-swing`, `enable-jemacs`, and `enable-servlet>` See the sample `build.properties` for more information on these.

If you change any of the build properties, you will generally want to do an '`ant clean`' before building again as the build is often not able to notice that kind of change. In the case of changing a directory path, you would want to do the `clean` before changing the path.

A special note for NetBeans users: For some reason the build-tools target which compiles an Ant task won't compile with the classpath provided by NetBeans. You may do '`ant`

`build-tools`' from the command line outside of NetBeans, in which case you will not want to use the `clean` target as that will delete the tool files as well. You can use the `clean-build` and/or `clean-dist` targets as appropriate. Alternatively you can add `ant.jar` to the `build-tools` classpath by copying or linking it into a `lib/ext` directory in Kawa's source directory (the one containing the `build.xml` file).

3.3.3 Using the Jikes compiler

Jikes (`http://oss.software.ibm.com/developerworks/opensource/jikes/project/`) is a Java source-to-bytecode compiler that is much faster than Sun's `javac`. (Note that this only speeds up building Kawa from source, not actually running Kawa.) The instructions for using jikes are as above, except that you need to specify Jikes at `configure` time, setting the `JAVAC` environment variable. If `jikes` is in your execution path, do:

```
JAVAC=jikes ./configure
```

You also need to inform Jikes where it should find the standard Java classes (since Jikes is a compiler only). For example:

```
CLASSPATH=.:/opt/jdk1.3/jre/lib/rt.jar
export CLASSPATH
```

3.3.4 Compiling Kawa to native code with GCJ

The GNU Compiler for the Java(tm) Programming Language (GCJ (`http://gcc.gnu.org/java/`)) is part of the GNU Compiler Collection (GCC (`http://gcc.gnu.org/`)). It can compile Java source or bytecode files into native code on supported systems. Version 3.3 or later of GCC is recommended, and only Intel x86-based Linux/GNU system have been tested with Kawa.

First, get and install GCC 3.3. Set `PREFIX` to where you want to install GCJ, and configure it with these options:

```
./configure --enable-threads --enable-languages=c++,java --prefix $PREFIX
make bootstrap
make install
```

Make sure `gcj` is in your path and refers to the newly-installed version, and if needed, set `LD_LIBRARY_PATH` to point to the directory where `libgcj.so` was installed:

```
PATH=$PREFIX/bin:$PATH
LD_LIBRARY_PATH=$PREFIX/lib
export LD_LIBRARY_PATH
```

To build Kawa, you need to specify `--with-gcj` to `configure` which tells it to use GCJ. Currently you also need to specify `--without-awt --without-swing` because GCJ does not yet support AWT or Swing:

```
./configure --with-gcj --without-awt --without-swing --prefix $PREFIX
```

Then as before:

```
make
make install
```

3.3.5 Building Kawa under MS-Windows

Using the `ant` method is recommended for building Kawa under Microsoft Windows. You may get an error message `"Out of environment space."` See `http://support.microsoft.com/support/kb` for a solution. Alternatively you can run the class `org.apache.tools.ant.Main` directly from the Ant jar.

The Kawa `configure` and `make` process assumes a Unix-like environment. If you want to build Kawa from source under Windows (95, 98, or NT), you could use a Unix empulation package, such as the free Cygwin (`http://sources.redhat.com/cygwin/`). However, there are some problems with filenames that make this more complicated than it should be. It should be possible to build Kawa under Cygwin using `gcj` as descibed above.

4 How to start up and run Kawa

The easiest way to start up Kawa is to run the '`kawa`' program. This finds your java interpreter, and sets up '`CLASSPATH`' correctly. If you have installed Kawa such `$PREFIX/bin` is in your `$PATH`, just do:

```
kawa
```

However, '`kawa`' only works if you have a Unix-like environment. On some platforms, '`kawa`' is a program that uses the GNU '`readline`' library to provide input line editing.

To run Kawa manually, you must start a Java interpreter. How you do this depends on the Java interpreter. For JavaSoft's JDK, you must have the Java interpreter in your `PATH`. You must also make sure that the `kawa/repl.class` file, the rest of the Kawa packages, and the standard Java packages can be found by searching CLASSPATH. See Section 3.1 [Running Java], page 3.

Then you do:

```
java kawa.repl
```

In either case, you will then get the '`#|kawa:1|#`' prompt, which means you are in the Kawa read-eval-print-loop. If you type a Scheme expression, Kawa will evaluate it. Kawa will then print the result (if there is a non-`"void"` result).

4.1 Command-line arguments

You can pass various flags to Kawa, for example:

```
kawa -e '(display (+ 12 4))(newline)'
```

or:

```
java kawa.repl -e '(display (+ 12 4))(newline)'
```

Either causes Kawa to print '`16`', and then exit.

At startup, Kawa executes an init file from the user's home directory. The init file is named `.kawarc.scm` on Unix-like systems (those for which the file separator is `'/'`), and `kawarc.scm` on other systems. This is done before the read-eval-print loop or before the first `-f` or `-c` argument. (It is not run for a `-e` command, to allow you to set options to override the defaults.)

'`-e` *expr*' Kawa evaluates *expr*, which contains one or more Scheme expressions. Does not cause the `~/.kawarc.scm` init file to be run.

'`-c` *expr*' Same as '`-e` *expr*', except that it does cause the `~/.kawarc.scm` init file to be run.

'`-f` *filename-or-url*'
Kawa reads and evaluates expressions from the file named by *filename-or-url*. If the latter is '`-`', standard input is read (with no prompting). Otherwise, it is equivalent to evaluating '`(load "`*filename-or-url*`")`'. The *filename-or-url* is interpreted as a URL if it is absolute - it starts with a "URI scheme" like `http:`.

'`-s`'
'`--`' The global variable '`command-line-arguments`' is set to the remaining arguments (if any), and an interactive read-eval-print loop is started. This uses the same "console" as where you started up Kawa; use '`-w`' to get a new window.

'`-w`' Creates a new top-level window, and runs an interactive read-eval-print in the new window. See Section 4.3 [New-Window], page 11. Same as `-e (scheme-window #t)`. You can specify multiple '`-w`' options, and also use '`-s`'.

'`--help`' Prints out some help.

'`--version`'
Prints out the Kawa version number, and then exits.

'`--server` *portnum*'
Start a server listening from connections on the specified *portnum*. Each connection using the Telnet protocol causes a new read-eval-print-loop to started. This option allows you to connect using any Telnet client program to a remote "Kawa server".

'`--scheme`'
Set the default language to Scheme. (This is the default unless you select another language, or you name a file with a known extension on the command-line.)

'`--elisp`'
'`--emacs`'
'`--emacs-lisp`'
Set the default language to Emacs Lisp. (The implementation is quite incomplete.)

'`--lisp`'
'`--clisp`'
'`--clisp`'
'`--commonlisp`'
'`--common-lisp`'
Set the default language to CommonLisp. (The implementation is *very* incomplete.)

'`--krl`' Set the default language to KRL. See Chapter 13 [KRL], page 76.

'`--brl`' Set the default language to KRL, in BRL-compatibility mode. See Chapter 13 [KRL], page 76.

'`--xquery`'
Set the default language to the draft XML Query language. See the Kawa-XQuery page (`http://www.gnu.org/software/qexo/`) for more information.

'`--xslt`' Set the default language to XSLT (XML Stylesheet Language Transformations). (The implementation is *very* incomplete.) See the Kawa-XSLT page (`http://www.gnu.org/software/qexo/xslt.html`) for more information.

'`--output-format` *format*'
'`--format` *format*'
Change the default output format to that specified by *format*. See ⟨undefined⟩ [Named output formats], page ⟨undefined⟩ for more information and a list.

The following options control which calling conventions are used:

'`--full-tailcalls`'
Use a calling convention that supports proper tail recursion.

'`--no-full-tailcalls`'
Use a calling convention that does not support proper tail recursion. Self-tail-recursion (i.e. a recursive call to the current function) is still implemented correctly, assuming that the called function is known at compile time.

The default is currently `--no-full-tailcalls` because I believe it is faster (though I have not done any measurements yet). It is also closer to the Java call model, so may be better for people primarily interested in using Kawa for scripting Java systems.

Both calling conventions can co-exist: Code compiled with `--full-tailcalls` can call code compiled with `--no-full-tailcalls` and vice versa.

The options '`-C`', '`-d`', '`-T`', '`-P`', '`--main`' '`--applet`', and `--servlet` are used to compile a Scheme file; see Section 6.2 [Files compilation], page 13. The option '`--connect` *portnum*' is only used by the '`kawa`' front-end program.

The following options are useful if you want to debug or understand how Kawa works.

'`--debug-dump-zip`'
Normally, when Kawa loads a soyrce file, or evaluates a non-trivial expression, it generates new internal Java classes but does not write them out. This option asks it to write out generated classes in a '`.zip`' archive whose name has the prefix '`kawa-zip-dump-`'.

'`--debug-print-expr`'
Kawa translates source language forms into an internal `Expression` data structure. This option causes that data structure to be written out in a readable format to the standard output.

'`--debug-print-final-expr`'
Similar to the previous option, but prints out the `Expression` after various transformations and optimizations have been done, and just before code generation.

If there are further command-line arguments after the options have been processed, then the first remaining argument names a file that is read and evaluated. If there is no such argument, then Kawa enters an interactive read-eval-print loop, but only if none of the '`-c`', '`-e`', '`-f`', '`-s`', '`-C`', or '`--`' options were specified.

4.2 Running Command Scripts

Unix-like systems support a mechanism where a *script* can specify a programs that should execute it. The convention is that the first line of the file should start with the two characters '`#!`' followed by the absolute path of the program that should process (interpret) the script.

This is convention works well for script languages that use '`#`' to indicate the start of a comment, since the interpreter will automatically ignore the line specifying the interpreter filename. Scheme, however, uses '`#`' for various special objects, and Kawa specifically uses '`#!`' as a prefix for various Section 7.3 [Special named constants], page 20 such as `#!optional`.

Kawa does recognize the three-character sequence '`#!/`' at the beginning of a file as special, and ignores it. So you can specify command interpreters, as long as you don't put a space between the '`#!`' and the interpreter filename. Here is an example:

```
#!/usr/local/bin/kawa
(format #t "The time is ~s~%" (make <java.util.Date>))
```

If this file has the execute permission set and is in your `PATH`, then you can execute it just my naming it on command line. The system kernel will automatically execute `kawa`, passing it the filename as an argument.

Note that the full path-name of the `kawa` interpreter must be hard-wired into the script. This means you may have to edit the script depending on where Kawa is installed on your system. Another possible problem is that the interpreter must be an actual program, not a shell script. Depending on how you configure and install Kawa, `kawa` can be a real program or a script. You can avoid both problems by the `env` program, available on most modern Unix-like systems:

```
#!/usr/bin/env kawa
(format #t "The time is ~s~%" (make <java.util.Date>))
```

4.3 Running a Command Interpreter in a new Window

An alternative interface runs the Java read-eval-print-loop inside a new window. This is in some ways nicer. One reason is that it provides better editing. You can also create new windows. They can either have different top-level environments or they can share environments. To try it, do:

```
java kawa.repl -w
```

4.4 Exiting Kawa

Kawa normally keeps running as long as there is an active read-eval-print loop still awaiting input or there is an unfinished other computation (such as requested by a '`-e`' of '`-f`' option).

To close a read-eval-print-loop, you can type the special literal `#!eof` at top level. This is recognized as end-of-file. Unfortunately, due to thread-related complications, just typing an end-of-file character (normally ctrl/D until Unix), will not work.

If the read-eval-print-loop is in a new window, you can select '`Close`' from the '`File`' menu.

To exit the entire Kawa session, call the `exit` procedure (with 0 or 1 integer arguments).

5 Features of R5RS not implemented

Kawa implements all the required and optional features of R5RS, with the following exceptions.

The entire "numeric tower" is implemented. However, some transcendental function only work on reals. Integral function do not necessarily work on inexact (floating-point) integers. (The whole idea of "inexact integer" in R5RS seems rather pointless ...)

Also, `call-with-current-continuation` is only "upwards" (?). I.e. once a continuation has been exited, it cannot be invoked. These restricted continuations can be used to implement catch/throw (such as the examples in R4RS), but not co-routines or backtracking.

Kawa now does general tail-call elimination, but only if you use the flag `--full-tail-calls`. (Currently, the `eval` function itself is not fully tail-recursive, in violation of R5RS.) The `--full-tail-calls` flag is not on by default, partly because it is noticably slower (though I have not measured how much), and partly I think it is more useful for Kawa to be compilatible with standard Java calling conventions and tools. Code compiled with `--full-tail-calls` can call code compiled without it and vice versa.

Even without `--full-tail-calls`, if the compiler can prove that the procedure being called is the current function, then the tail call will be replaced by a jump. This means the procedure must be defined using a letrec, not a define (because the compiler does not know if someone might re-define a global definition), and there must be no assignments (using `set!`) to the procedure binding.

6 Compiling Scheme code to byte-code or an executable

All Scheme functions and source files are invisibly compiled into internal Java byte-codes. A traditional evaluator is only used for top-level directly entered expressions *outside* a lambda. (It would have been simpler to also byte-compile top-level expressions by surrounding them by a dummy lambda. However, this would create a new Class object in the Java VM for every top-level expression. This is undesirable unless you have a VM that can garbage collect Class objects.)

To save speed when loading large Scheme source files, you probably want to pre-compile them and save them on your local disk. There are two ways to do this.

You can compile a Scheme source file to a single archive file. You do this using the `compile-file` function. The result is a single file that you can move around and `load`

just like the `.scm` source file. You just specify the name of the archive file to the `load` procedure. Currently, the archive is a "zip" archive and has extension ".zip"; a future release will probably use "Java Archive" (jar) files. The advantage of compiling to an archive is that it is simple and transparent. A minor disadvantage is that it causes the Java "verifier" to be run when functions are loaded from it, which takes a little extra time.

Alternatively, you can compile a Scheme source file to a collection of '`.class`' files. You then use the standard Java class loading mechanism to load the code. The Java "verifier" does not need to get run, which makes loading a little faster. The compiled class files do have to be installed be installed somewhere in the `CLASSPATH`.

You can also compile your Scheme program to native code using GCJ.

6.1 Compiling Scheme to an archive file

compile-file *source-file compiled-archive* Function

Compile the *source-file*, producing a `.zip` archive *compiled-file*.

For example, to byte-compile a file '`foo.scm`' do:

```
(compile-file "foo.scm" "foo")
```

This will create '`foo.zip`', which contains byte-compiled "j-code". You can move this file around, without worrying about class paths. To load the compiled file, you can later `load` the named file, as in either `(load "foo")` or `(load "foo.zip")`. This should have the same effect as loading '`foo.scm`', except you will get the faster byte-compiled versions.

6.2 Compiling Scheme to a set of .class files

Invoking '`kawa`' (or '`java kawa.repl`') with the '`-C`' flag will compile a '`.scm`' source file into one or more '`.class`' files:

```
kawa --main -C myprog.scm
```

You run it as follows:

`kawa [-d` *outdirectory*`] [-P` *prefix*`] [-T` *topname*`] [--main | --applet | --servlet] -C` *infile* `...`

Note the '`-C`' must come last, because '`Kawa`' processes the arguments and options in order,

Here:

'`-C` *infile* `...`'
: The Scheme source files we want to compile.

'`-d` *outdirectory*'
: The directory under which the resulting '`.class`' files will be. The default is the current directory.

'`-P` *prefix*'
: A string to prepend to the generated class names. The default is the empty string.

'`-T` *topname*'
: The name of the `"top"` class - i.e. the one that contains the code for the top-level expressions and definitions. The default is generated from the *infile* and *prefix*.

'`--main`'
: Generate a `main` method so that the resulting `"top"` class can be used as a stand-alone application. See Section 6.4 [Application compilation], page 15.

'`--applet`'
: The resulting class inherits from `java.applet.Applet`, and can be used as an applet. See Section 6.5 [Applet compilation], page 15.

'`--servlet`'
: The resulting class implements `javax.servlet.http.HttpServlet`, and can be used as an servlet in a servlet container like Tomcat.

When you actually want to load the classes, the *outdirectory* must be in your '`CLASSPATH`'. You can use the standard `load` function to load the code, by specifying the top-level class, either as a file name (relative to *outdirectory*) or a class name. E.g. if you did:

```
kawa -d /usr/local/share/java -P my.lib. -T foo -C foosrc.scm
```

you can use either:

```
(load "my.lib.foo")
```

or:

```
(load "my/lib/foo.class")
```

If you are compiling a Scheme source file (say '`foosrc.scm`') that uses macros defined in some other file (say '`macs.scm`'), you need to make sure the definitions are visible to the compiler. One way to do that is with the '`-f`':

```
kawa -f macs.scm -C foosrc.scm
```

6.3 Compilation options

Various named option control how Kawa compiles certain forms.

'`--module-static`'
: If no `module-static` is specified, generate a static module (as if `(module-static #t)` were specified). See Section 10.8 [Module classes], page 61.

'`--warn-invoke-unknown-method`'
: Emit a warning if the `invoke` function calls a named method for which there is no matching method in the compile-time type of the receiver. This (currently) defaults to on; to turn it off use the `--no-warn-invoke-unknown-method` flag.

'`--warn-undefined-variable`'
: Emit a warning if the code references a variable which is neither in lexical scope nor in the compile-time dynamic (global) environment. This is useful for catching typos. (A `define-variable` form can be used to silence warnings. It declares to the compiler that a variable is to be resolved dynamically.)

An option can be followed by a value, as in `--warn-invoke-unknown-method=no`. For boolean options, the values `yes`, `true`, `on`, or `1` enable the option, while `no`, `false`, `off`, or `0` disable it. You can also negate an option by prefixing it with `no-`: The option `--no-warn-invoke-unknown-method` is the same as `--warn-invoke-unknown-method=no`.

You can set the same options (except, for now, `module-static`) within your Scheme source file. (In that case they override the options on the command line.)

module-compile-options [*key: value*] ... Syntax

This sets the value of the `key` option to `value` for the current module (source file). It takes effect as soon it is seen during the first macro-expansion pass, and is active thereafter (unless overridden by `with-compile-options`).

The *key* is one of the above option names. (The following colon make it a Kawa keyword.) The *value* must be a literal value: either a boolean (`#t` or `#f`), a number, or a string, depending on the *key*. (All the options so far are boolean options.)

```
(module-compile-options warn-undefined-variable: #t)
;; This causes a warning message that y is unknown.
(define (func x) (list x y))
```

with-compile-options [*key: value*] ... *body* Syntax

Similar to `module-compile-options`, but the option is only active within *body*.

```
(define (func x)
  (with-compile-options warn-invoke-unknown-method: #f
    (invoke x 'size)))
```

6.4 Compiling Scheme to a standalone application

A Java application is a Java class with a special method (whose name is `main`). The application can be invoked directly by naming it in the Java command. If you want to generate an application from a Scheme program, create a Scheme source file with the definitions you need, plus the top-level actions that you want the application to execute. You can compile in the regular way decribed in the previous section, but add the `--main` option. For example, assuming your Scheme file is `MyProgram.scm`:

```
kawa --main -C MyProgram.scm
```

This will create a `MyProgram.class` which you can either `load` (as decribed in the previous section), or invoke as an application:

```
java MyProgram [args]
```

Your Scheme program can access the command-line arguments *args* by using the global variable '`command-line-arguments`'.

6.5 Compiling Scheme to an applet

An applet is a Java class that inherits from `java.applet.Applet`. The applet can be downloaded and run in a Java-capable web-browser. To generate an applet from a Scheme program, write the Scheme program with appropriate definitions of the functions '`init`',

'`start`', '`stop`' and '`destroy`'. You must declare these as zero-argument functions with a `<void>` return-type.

Here is an example, based on the scribble applet in Flanagan's "Java Examples in a Nutshell" (O'Reilly, 1997):

```
(define-private last-x 0)
(define-private last-y 0)

(define (init) <void>
  (let ((applet :: <java.applet.Applet> (this)))
    (invoke applet 'addMouseListener
    (object (<java.awt.event.MouseAdapter>)
    ((mousePressed (e :: <java.awt.event.MouseEvent>)) <void>
     (set! last-x (invoke e 'getX))
     (set! last-y (invoke e 'getY)))))
    (invoke applet 'addMouseMotionListener
    (object (<java.awt.event.MouseMotionAdapter>)
    ((mouseDragged (e :: <java.awt.event.MouseEvent>)) <void>
     (let ((g :: <java.awt.Graphics>
      (invoke applet 'getGraphics))
   (x :: <int> (invoke e 'getX))
   (y :: <int> (invoke e 'getY)))
       (invoke g 'drawLine last-x last-y x y)
       (set! last-x x)
       (set! last-y y)))))))

(define (start) <void> (format #t "called start.~%~!"))
(define (stop) <void> (format #t "called stop.~%~!"))
(define (destroy) <void> (format #t "called destroy.~%~!"))
```

You compile the program with the '`--applet`' flag in addition to the normal '`-C`' flag:

```
java kawa.repl --applet -C scribble.scm
```

You can then create a '`.jar`' archive containing your applet. You also need to include the Kawa classes in the '`.jar`', or you can include a `MANIFEST` file that specifies `Class-Path` to use a Java 2 download extension (`http://java.sun.com/docs/books/tutorial/ext/basics/download.h`

```
jar cf scribble.jar scribble*.class other-classes ...
```

Finally, you create an '`.html`' page referencing your applet:

```
<html><head><title>Scribble testapp</title></head>
<body><h1>Scribble testapp</h1>
You can scribble here:
<br>
<applet code="scribble.class" archive="scribble.jar" width=200 height=200>
Sorry, Java is needed.</applet>
</body></html>
```

6.6 Compiling Scheme to a native executable

You can compile your Scheme program to native code using GCJ, as long as you have built Kawa using GCJ.

First, you need to compile the Scheme code to a set of `.class` files; see Section 6.2 [Files compilation], page 13.

```
kawa --main -C myprog.scm
```

Then to create an executable `myprog` do:

```
gckawa --main=myprog myprog*.class -o myprog
```

The `gckawa` is a simple shell script that calls `gcj`. The reason for the wildcard in `myprog*.class` is that sometimes Kawa will generate some helper classes in addition to `myprog.class`. The `--main` option tell `gcj` which class contains the `main` method it should use. The `-o` option names the resulting executable program. The `-lkawa` option tells the linker it should link with the kawa shared library, and the `-L$PREFIX/bin` option tells the linker where it can find that library.

7 Extensions

7.1 Syntax and conditional compilation

define-syntax .. Syntax

Pattern ...

defmacro *name lambda-list form ...* Syntax

Defines an old-style macro a la Common Lisp, and installs `(lambda` *lambda-list form* `...)` as the expansion function for *name*. When the translator sees an application of *name*, the expansion function is called with the rest of the application as the actual arguments. The resulting object must be a Scheme source form that is futher processed (it may be repeatedly macro-expanded).

If you define a macro with `defmacro`, you (currently) cannot use the macro in the same compilation as the definition. This restriction does not apply to macros defined by `define-syntax`.

gentemp Function

Returns a new (interned) symbol each time it is called. The symbol names are implementation-dependent. (This is not directly macro-related, but is often used in conjunction with `defmacro` to get a fresh unique identifier.)

cond-expand *cond-expand-clause** [(*else command-or-definition**)] Syntax

cond-expand-clause ::= `(`*feature-requirement command-or-definition*`*)`
feature-requirement ::= *feature-identifier*
| `(and` *feature-requirement*`*)`
| `(or` *feature-requirement*`*)`
| `(not` *feature-requirement*`)`
feature-identifier ::= a symbol which is the name or alias of a SRFI

The `cond-expand` form tests for the existence of features at macro-expansion time. It either expands into the body of one of its clauses or signals an error during syntactic processing. `cond-expand` expands into the body of the

first clause whose feature requirement is currently satisfied; the `else` clause, if present, is selected if none of the previous clauses is selected.

A feature requirement has an obvious interpretation as a logical formula, where the *feature-identifier* variables have meaning true if the feature corresponding to the feature identifier, as specified in the SRFI registry, is in effect at the location of the `cond-expand` form, and false otherwise. A feature requirement is satisfied if its formula is true under this interpretation.

Examples:

```
(cond-expand
    ((and srfi-1 srfi-10)
     (write 1))
    ((or srfi-1 srfi-10)
     (write 2))
    (else))

(cond-expand
  (command-line
   (define (program-name) (car (argv)))))
```

The second example assumes that `command-line` is an alias for some feature which gives access to command line arguments. Note that an error will be signaled at macro-expansion time if this feature is not present.

7.2 Multiple values

The multiple-value feature was added in R5RS.

values *object ...* Function

Delivers all of its arguments to its continuation.

call-with-values *thunk receiver* Function

Call its *thunk* argument with a continuation that, when passed some values, calls the *receiver* procedure with those values as arguments.

let-values ((*formals expression*) *...*) *body* Syntax

Each *formals* should be a formal arguments list as for a `lambda`, cf section 4.1.4 of the R5RS.

The *expression*s are evaluated in the current environment, the variables of the *formals* are bound to fresh locations, the return values of the *expression*s are stored in the variables, the *body* is evaluated in the extended environment, and the values of the last expression of *body* are returned. The *body* is a "tail body", cf section 3.5 of the R5RS.

The matching of each *formals* to values is as for the matching of *formals* to arguments in a `lambda` expression, and it is an error for an *expression* to return a number of values that does not match its corresponding *formals*.

```
(let-values ((a b . c) (values 1 2 3 4)))
  (list a b c))            --> (1 2 (3 4))
```

```
(let ((a 'a) (b 'b) (x 'x) (y 'y))
  (let-values (((a b) (values x y))
               ((x y) (values a b)))
    (list a b x y)))       --> (x y a b)
```

let*-values ((*formals expression*) ...) *body* Syntax

Each *formals* should be a formal arguments list as for a `lambda` expression, cf section 4.1.4 of the R5RS.

`let*-values` is similar to `let-values`, but the bindings are performed sequentially from left to right, and the region of a binding indicated by (*formals expression*) is that part of the `let*-values` expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

```
(let ((a 'a) (b 'b) (x 'x) (y 'y))
  (let*-values (((a b) (values x y))
                ((x y) (values a b)))
    (list a b x y)))       --> (x y x y)
```

receive *formals expression body* Syntax

The *formals*, *expression*, and *body* are as described in R5RS. Specifically, *formals* can have any of three forms:

'`(`*variable1* `...` *variablen*`)`'
: The environment in which the receive-expression is evaluated is extended by binding *variable1*, ..., *variablen* to fresh locations. The *expression* is evaluated, and its values are stored into those locations. (It is an error if *expression* does not have exactly *n* values.)

'*variable*'
: The environment in which the receive-expression is evaluated is extended by binding *variable* to a fresh location. The *expression* is evaluated, its values are converted into a newly allocated list, and the list is stored in the location bound to *variable*.

'`(`*variable1* `...` *variablen* `.` *variablen+1*`)`'
: The environment in which the receive-expression is evaluated is extended by binding *variable1*, ..., *variablen+1* to fresh locations. The *expression* is evaluated. Its first *n* values are stored into the locations bound to *variable1* ... *variablen*. Any remaining values are converted into a newly allocated list, which is stored into the location bound to *variablen+1* (It is an error if *expression* does not have at least *n* values.)

In any case, the expressions in *body* are evaluated sequentially in the extended environment. The results of the last expression in the body are the values of the receive-expression.

values-append *arg1 ...* Function

The values resulting from evaluating each argument are appended together.

7.3 Special named constants

#!optional Constant

Special self-evaluating literal used in lambda parameter lists before optional parameters.

#!rest Constant

Special self-evaluating literal used in lambda parameter lists before the rest parameter.

#!key Constant

Special self-evaluating literal used in lambda parameter lists before keyword parameters.

#!eof Constant

The end-of-file object.

Note that if the Scheme reader sees this literal at top-level, it is returned literally. This is indistinguishable from coming to the end of the input file. If you do not want to end reading, but want the actual value of `#!eof`, you should quote it.

#!void Constant

The void value. Same as `(values)`. If this is the value of an expression in a read-eval-print loop, nothing is printed.

#!null Constant

The Java `null` value. This is not really a Scheme value, but is useful when interfacing to low-level Java code.

7.4 Keywords

Keywords are similar to symbols. The main difference is that keywords are self-evaluating and therefore do not need to be quoted in expressions. They are used mainly for specifying keyword arguments.

keyword ::= *identifier*:

An alternative syntax, with the colon first, is supported for compatibility with Common Lisp and some other Scheme implementations:

keyword ::= :*identifier*

Putting the colon first has exactly the same effect as putting it last; putting is last is recommended, and is how keywords are printed.

A keyword is a single token; therefore no whitespace is allowed between the *identifier* and the colon (which is not considered part of the name of the keyword).

keyword? *obj* Function

Return `#t` if *obj* is a keyword, and otherwise returns `#f`.

keyword->string *keyword* Function
Returns the name of *keyword* as a string. The name does not include the final `#\:`.

string->keyword *string* Function
Returns the keyword whose name is *string*. (The *string* does not include a final `#\:`.)

7.5 Procedures

apply *proc* [*arg1 ...*] *args* Function
Args must be a sequence (list, vector, or string) or a primitive Java array. (This is an extension over standard Scheme, which requires that *args* be a list.) Calls the *proc* (which must be a procedure), using as arguments the *arg1...* values plus all the elements of *args*.

constant-fold *proc arg1 ...* Syntax
Same as (*proc arg1* `...`), unless *proc* and all the following arguments are compile-time constants. (That is: They are either constant, or symbols that have a global binding and no lexical binding.) In that case, *proc* is applied to the arguments at compile-time, and the result replaces the `constant-fold` form. If the application raises an exception, a compile-time error is reported. For example:

```
(constant-fold vector 'a 'b 'c)
```

is equivalent to `(quote #(a b c))`, assuming `vector` has not been re-bound.

7.5.1 Procedure properties

You can associate arbitrary *properties* with any procedure. Each property is a (*key*, *value*)-pair. Usually the *key* is a symbol, but it can be any object.

The system uses certain internal properties: `'name` refers to the name used when a procedure is printed; `'emacs-interactive` is used to implement Emacs `interactive` specification; `'setter` is used to associate a `setter` prcedure.

procedure-property *proc key* [*default*] Function
Get the property value corresponding to the given *key*. If *proc* has no property with the given *key*, return *default* (which defaults to `#f`) instead.

set-procedure-property! *proc key value* Function
Associate the given *value* with the *key* property of *proc*.

To change the print name of the standard `+` procedure (probably not a good idea!), you could do:

```
(set-procedure-property! + 'name 'PLUS)
```

Note this *only* changes the name property used for printing:

```
+ => #<procedure PLUS>
(+ 2 3) => 5
(PLUS 3 4) => ERROR
```

As a matter of style, it is cleaner to use the `define-procedure` form, as it is a more declarative interface.

define-procedure *name* [*propname: propvalue*] *... method ...* Syntax

Defines *name* as a compound procedure consisting of the specified *method*s, with the associated properties. Applying *name* select the "best" *method*, and applies that. See the following section on generic procedures.

For example, the standard `vector-ref` procedure specifies one method, as well as the `setter` property:

```
(define-procedure vector-ref
  setter: vector-set!
  (lambda ((vector :: <vector>) (k :: <int>))
    (invoke vector 'get k)))
```

7.5.2 Generic (dynamically overloaded) procedures

A *generic procedure* is a collection of *method procedures*. (A "method procedure" is not the same as a Java method, but the terms are related.) You can call a generic procedure, which selects the "closest match" among the component method procedures: I.e. the most specific method procedure that is applicable given the actual arguments.

Note: The current implementation of selecting the "best" method is not reliable if there is more than one method. It can select depending on argument count, and it can select between primitive Java methods. However, it cannot yet do what you probably hope for: select between different Scheme procedures based on parameter types.

make-procedure [*keyword: value*]*... method...* Function

Create a generic procedure given the specific methods. You can also specify property values for the result.

The *keyword*s specify how the arguments are used. A `method:` keyword is optional and specifies that the following argument is a method. A `name:` keyword specifies the name of the resulting procedure, when used for printing. Unrecognized keywords are used to set the procedure properties of the result.

```
(define plus10 (make-procedure foo: 33 name: 'Plus10
                               method: (lambda (x y) (+ x y 10))
                               method: (lambda () 10)))
```

7.5.3 Extended Formal Arguments List

The formal arguments list of a lambda expression has two extendsions over standard Scheme: Kawa borrows the extended formal argument list of DSSSL, and Kawa allows you to declare the type of the parameter.

lambda-expression ::= (lambda *formals* [*rtype*] *body*)

where

formals ::= (*formal-arguments*) | *rest-arg*

You can of course also use the extended format in a `define`:

(define (*name formal-arguments*) [*rtype*] *body*)

formal-arguments ::=
  req-opt-args (*rest-key-args* | . *rest-arg*)

req-opt-args ::= *req-arg* ... [#!optional *opt-arg* ...]
rest-key-args ::= [#!rest *rest-arg*] [#!key *key-arg* ...]
req-arg ::= *variable* [:: *type*] | (*variable* [[::] *type*])
opt-arg ::= *arg-with-default*
key-arg ::= *arg-with-default*
arg-with-default ::= *variable* [:: *type*]
  | (*variable* [:: *type* [*initializer*] | *initializer* [[::] *type*]])
rest-arg ::= *variable*

When the procedure is applied to a list of actual arguments, the formal and actual arguments are processed from left to right as follows:

- The *req-arg*s are bound to actual arguments starting with the first actual argument. It shall be an error if there are fewer actual arguments then there are *req-arg*s.
- Next the *opt-arg*s are bound to remaining actual arguemnts. If there are fewer remaining actual arguments than there are *opt-arg*s, then the remaining *variable*s are bound to the corresponding *initializer*, if one was specified, and otherwise to `#f`. The *initializer* is evaluated in an environment in which all the previous formal parameters have been bound.
- If there is a *rest-arg*, it is bound to a list of all the remaining actual arguments. These remaining actual arguments are also eligible to be bound to keyword arguments. If there is no *rest-arg* and there are no *key-arg*s, then it shall be an error if there are any remaining actual arguments.
- If `#!key` was specified, then there shall be an even number of remaining actual arguments. These are interpreted as a series of pairs, where the first member of each pair is a keyword specifying the argument name, and the second is the corresponding value. It shall be an error if the first member of a pair is not a keyword. It shall be an error if the argument name is not the same as a variable in a *key-arg*s, unless there is a *rest-arg*. If the same argument name occurs more than once in the list of actual arguments, then the first value is used. If there is no actual argument for a particular *key-arg*, then the variable is bound to the corresponding *initializer*, if one was specified, and otherwise to `#f`. The *initializer* is evaluated in an environment in which all the previous formal parameters have been bound.

If a *type* is specified, the corresponding actual argument (or the *initializer* default value) is coerced to the specified *type*. In the function body, the parameter has the specified type.

If *rtype* (the first form of the function body) is an unbound identifier of the form `<TYPE>` (that is the first character is '`<`' and the last is '`>`'), then tha specifies the functions return type. It is syntactic sugar for `(as <TYPE> (begin BODY))`.

cut *slot-or-expr slot-or-expr** [`<...>`] Syntax

where each *slot-or-expr* is either an *expression* or the literal symbol `<>`.

It is frequently necessary to specialize some of the parameters of a multi-parameter procedure. For example, from the binary operation `cons` one might want to obtain the unary operation `(lambda (x) (cons 1 x))`. This specialization of parameters is also known as *partial application*, *operator section*, or *projection*. The macro `cut` specializes some of the parameters of its first argument. The parameters that are to show up as formal variables of the result are indicated by the symbol `<>`, pronouced as "slot". In addition, the symbol `<...>`, pronounced as "rest-slot", matches all residual arguments of a variable argument procedure.

A `cut`-expression is transformed into a *lambda expression* with as many formal variables as there are slots in the list *slot-or-expr**. The body of the resulting *lambda expression* calls the first *slot-or-expr* with arguments from the *slot-or-expr** list in the order they appear. In case there is a rest-slot symbol, the resulting procedure is also of variable arity, and the body calls the first *slot-or-expr* with remaining arguments provided to the actual call of the specialized procedure.

Here are some examples:
`(cut cons (+ a 1) <>)` is the same as `(lambda (x2) (cons (+ a 1) x2))`
`(cut list 1 <> 3 <> 5)` is the same as `(lambda (x2 x4) (list 1 x2 3 x4 5))`
`(cut list)` is the same as `(lambda () (list))`
`(cut list 1 <> 3 <...>)` is the same as `(lambda (x2 . xs) (apply list 1 x2 3 xs))`

The first argument can also be a slot, as one should expect in Scheme:
`(cut <> a b)` is the same as `(lambda (f) (f a b))`

cute *slot-or-expr slot-or-expr** [`<...>`] Syntax

The macro `cute` (a mnemonic for "cut with evaluated non-slots") is similar to `cut`, but it evaluates the non-slot expressions at the time the procedure is specialized, not at the time the specialized procedure is called.

For example:
`(cute cons (+ a 1) <>)` is the same as `(let ((a1 (+ a 1))) (lambda (x2) (cons a1 x2)))`

As you see from comparing this example with the first example above, the `cute`-variant will evaluate `(+ a 1)` once, while the `cut`-variant will evaluate it during every invocation of the resulting procedure.

7.6 Quantities and Numbers

As a super-class of numbers, Kawa also provides quantities. A *quantity* is a product of a *unit* and a pure number. The number part can be an arbitrary complex number. The unit is a product of integer powers of base units, such as meter or second.

Kawa quantities are a generalization of the quantities in DSSSL, which only has length-derived quantities.

The precise syntax of quantity literals may change, but some examples are `10pt` (10 points), `5s` (5 seconds), and `4cm^2` (4 square centimeters).

quantity? *object* Function
True iff *object* is a quantity. Note that all numbers are quantities, but not the other way round. Currently, there are no quantities that re not numbers. To distinguish a plain unit-less number from a quantity, you can use `complex?`.

quantity->number *q* Function
Returns the pure number part of the quantity *q*, relative to primitive (base) units. If *q* is a number, returns *q*. If *q* is a unit, yields the magitude of *q* relative to base units.

quantity->unit *q* Function
Returns the unit of the quantity *q*. If *q* is a number, returns the empty unit.

make-quantity *x unit* Function
Returns the product of *x* (a pure number) and *unit*. You can specify a string instead of *unit*, such as `"cm"` or `"s"` (seconds).

define-base-unit *unit-name dimension* Syntax
Define *unit-name* as a base (primitive) unit, which is used to measure along the specified *dimension*.

```
(define-base-unit dollar "Money")
```

define-unit *unit-name expression* Syntax
Define *unit-name* as a unit (that can be used in literals) equal to the quantity *expression*.

```
(define-unit cent 0.01dollar)
```

quotient *x y* Function
Generalized to arbitrary real numbers, using the definition: `(truncate (/ x y))`.

remainder *x y* Function
Generalized to arbitrary real numbers, using the definition: `(- x (* y (truncate (/ x y))))`. If `y` is `0`, the result is `x` - i.e. we take `(* 0 (quotient x 0))` to be `0`. The result is inexact if either argument is inexact, even if `x` is exact and `y` is 0.

modulo *x y* Function
Generalized to arbitrary real numbers, using the definition: `(- x (* y (floor (/ x y))))`. If `y` is `0`, the result is `x`. The result is inexact if either argument is inexact, even if `x` is exact and `y` is 0.

7.7 Logical Number Operations

These functions operate on the 2's complement binary representation of an exact integer.

logand *i ...* Function
Returns the bit-wise logical "and" of the arguments. If no argument is given, the result is -1.

logior *i ...* Function
Returns the bit-wise logical "(inclusive) or" of the arguments. If no argument is given, the result is 0.

logxor *i ...* Function
Returns the bit-wise logical "exclusive or" of the arguments. If no argument is given, the result is 0.

lognot *i* Function
Returns the bit-wise logical inverse of the argument.

logop *op x y* Function
Perform one of the 16 bitwise operations of *x* and *y*, depending on *op*.

logtest *i j* Function
Returns true if the arguments have any bits in common. Same as `(not (zero? (logand i j)))`, but is more efficient.

logbit? *i pos* Function
Returns `#t` iff the bit numbered *pos* in *i* is one.

arithmetic-shift *i j* Function
Shifts *i* by *j*. It is a "left" shift if *j*>0, and a "right" shift if *j*<0.

The result is equal to `(floor (* i (expt 2 j)))`.

ash *i j* Function
Alias for `arithmetic-shift`.

logcount *i* Function
Count the number of 1-bits in *i*, if it is non-negative. If *i* is negative, count number of 0-bits.

integer-length *i* Function
Return number of bits needed to represent *i* in an unsigned field. Regardless of the sign of *i*, return one less than the number of bits needed for a field that can represent *i* as a two's complement integer.

bit-extract *n start end* Function
Return the integer formed from the (unsigned) bit-field starting at *start* and ending just before *end*. Same as `(arithmetic-shift (bitand n (bitnot (arithmetic-shift -1 end))) (- start))`.

7.8 Lists

The SRFI-1 List Library (`http://srfi.schemers.org/srfi-1/srfi-1.html`) is available, though not enabled by default. To use its functions you must `(require 'list-lib)` or `(require 'srfi-1)`.

```
(require 'list-lib)
(iota 5 0 -0.5) ;; => (0.0 -0.5 -1.0 -1.5 -2.0)
```

reverse! *list* Function

The result is a list consisting of the elements of *list* in reverse order. No new pairs are allocated, instead the pairs of *list* are re-used, with `cdr` cells of *list* reversed in place. Note that if *list* was pair, it becomes the last pair of the reversed result.

7.9 Strings

string-upcase *str* Function

Return a new string where the letters in *str* are replaced by their upper-case equivalents.

string-downcase *str* Function

Return a new string where the letters in *str* are replaced by their lower-case equivalents.

string-capitalize *str* Function

Return a new string where the letters in *str* that start a new word are replaced by their title-case equivalents, while non-initial letters are replaced by their lower-case equivalents.

string-upcase! *str* Function

Destructively modify *str*, replacing the letters by their upper-case equivalents.

string-downcase! *str* Function

Destructively modify *str*, replacing the letters by their upper-lower equivalents.

string-capitalize! *str* Function

Destructively modify *str*, such that the letters that start a new word are replaced by their title-case equivalents, while non-initial letters are replaced by their lower-case equivalents.

7.10 Multi-dimensional Arrays

Arrays are heterogeneous data structures whose elements are indexed by integer sequences of fixed length. The length of a valid index sequence is the rank or the number of dimensions of an array. The shape of an array consists of bounds for each index.

The lower bound b and the upper bound e of a dimension are exact integers with `(<=` b e`)`. A valid index along the dimension is an exact integer k that satisfies both `(<=` b k`)` and `(<` k e`)`. The length of the array along the dimension is the difference `(-` e b`)`. The size of an array is the product of the lengths of its dimensions.

A shape is specified as an even number of exact integers. These are alternately the lower and upper bounds for the dimensions of an array.

array? *obj* Function

Returns `#t` if *obj* is an array, otherwise returns `#f`.

shape *bound ...* Function

Returns a shape. The sequence *bound ...* must consist of an even number of exact integers that are pairwise not decreasing. Each pair gives the lower and upper bound of a dimension. If the shape is used to specify the dimensions of an array and *bound ...* is the sequence *b0 e0 ... bk ek ...* of *n* pairs of bounds, then a valid index to the array is any sequence *j0 ... jk ...* of *n* exact integers where each *jk* satisfies `(<= bk jk)` and `(< jk ek)`.

The shape of a *d*-dimensional array is a *d* 2 array where the element at *k 0* contains the lower bound for an index along dimension *k* and the element at *k 1* contains the corresponding upper bound, where *k* satisfies `(<= 0 k)` and `(< k d)`.

make-array *shape* Function
make-array *shape obj* Function

Returns a newly allocated array whose shape is given by *shape*. If *obj* is provided, then each element is initialized to it. Otherwise the initial contents of each element is unspecified. The array does not retain a reference to *shape*.

array *shape obj ...* Function

Returns a new array whose shape is given by *shape* and the initial contents of the elements are *obj ...* in row major order. The array does not retain a reference to *shape*.

array-rank *array* Function

Returns the number of dimensions of *array*.

```
(array-rank
  (make-array (shape 1 2 3 4)))
```

Returns 2.

array-start *array k* Function

Returns the lower bound for the index along dimension *k*.

array-end *array k* Function

Returns the upper bound for the index along dimension *k*.

array-ref *array k ...* Function
array-ref *array index* Function

Returns the contents of the element of *array* at index *k ....* The sequence *k ...* must be a valid index to *array*. In the second form, *index* must be either a vector or a 0-based 1-dimensional array containing *k ....*

```
(array-ref (array (shape 0 2 0 3)
                  'uno 'dos 'tres
                  'cuatro 'cinco 'seis)
   1 0)
```

Returns `cuatro`.

```
(let ((a (array (shape 4 7 1 2) 3 1 4)))
   (list (array-ref a 4 1)
```

```
(array-ref a (vector 5 1))
(array-ref a (array (shape 0 2)
                    6 1))))
```

Returns `(3 1 4)`.

array-set! *array k ... obj* Function
array-set! *array index obj* Function

Stores *obj* in the element of *array* at index *k* .... Returns the void value. The sequence *k* ... must be a valid index to *array*. In the second form, *index* must be either a vector or a 0-based 1-dimensional array containing *k* ....

```
(let ((a (make-array
            (shape 4 5 4 5 4 5))))
   (array-set! a 4 4 4 "huuhkaja")
   (array-ref a 4 4 4))
```

Returns `"huuhkaja"`.

share-array *array shape proc* Function

Returns a new array of *shape* shape that shares elements of *array* through *proc*. The procedure *proc* must implement an affine function that returns indices of *array* when given indices of the array returned by `share-array`. The array does not retain a reference to *shape*.

```
(define i_4
   (let* ((i (make-array
                (shape 0 4 0 4)
                0))
          (d (share-array i
                (shape 0 4)
                (lambda (k)
                   (values k k)))))
      (do ((k 0 (+ k 1)))
          ((= k 4))
         (array-set! d k 1))
      i))
```

Note: the affinity requirement for *proc* means that each value must be a sum of multiples of the arguments passed to *proc*, plus a constant.

Implementation note: arrays have to maintain an internal index mapping from indices *k1* ... *kd* to a single index into a backing vector; the composition of this mapping and *proc* can be recognised as `(+` *n0* `(*` *n1 k1*`) ... (*` *nd kd*`))` by setting each index in turn to 1 and others to 0, and all to 0 for the constant term; the composition can then be compiled away, together with any complexity that the user introduced in their procedure.

Multi-dimensional arrays are specified by SRFI-25 (`http://srfi.schemers.org/srfi-25/srfi-25.htm`
In Kawa, a one-dimensional array whose lower bound is 0 is also a sequence. Furthermore, if such an array is simple (not created `share-array`) it will be implemented using a `<vector>`. Uniform vectors and strings are also arrays in Kawa. For example:

```
(share-array
 (f64vector 1.0 2.0 3.0 4.0 5.0 6.0)
 (shape 0 2 0 3)
 (lambda (i j) (+ (* 2 i) j)))
```

evaluates to a two-dimensionsal array of `<double>`:

```
#2a((1.0 2.0 3.0) (3.0 4.0 5.0))
```

7.11 Uniform vectors

Uniform vectors are vectors whose elements are of the same numeric type. The are defined by SRFI-4 (`http://srfi.schemers.org/srfi-4/srfi-4.html`). However, the type names (such as `<s8vector>`) are a Kawa extension.

<s8vector> Variable

The type of uniform vectors where each element can contain a signed 8-bit integer. Represented using an array of `<byte>`.

<u8vector> Variable

The type of uniform vectors where each element can contain an unsigned 8-bit integer. Represented using an array of `<byte>`, but each element is treated as if unsigned.

<s16vector> Variable

The type of uniform vectors where each element can contain a signed 16-bit integer. Represented using an array of `<short>`.

<u16vector> Variable

The type of uniform vectors where each element can contain an unsigned 16-bit integer. Represented using an array of `<short>`, but each element is treated as if unsigned.

<s32vector> Variable

The type of uniform vectors where each element can contain a signed 32-bit integer. Represented using an array of `<int>`.

<u32vector> Variable

The type of uniform vectors where each element can contain an unsigned 32-bit integer. Represented using an array of `<int>`, but each element is treated as if unsigned.

<s64vector> Variable

The type of uniform vectors where each element can contain a signed 64-bit integer. Represented using an array of `<long>`.

<u64vector> Variable

The type of uniform vectors where each element can contain an unsigned 64-bit integer. Represented using an array of `<long>`, but each element is treated as if unsigned.

<f32vector> Variable

The type of uniform vectors where each element can contain a 32-bit floating-point real. Represented using an array of `<float>`.

<f64vector> Variable

The type of uniform vectors where each element can contain a 64-bit floating-point real. Represented using an array of `<double>`.

s8vector? *value* Function
u8vector? *value* Function
s16vector? *value* Function
u16vector? *value* Function
s32vector? *value* Function
u32vector? *value* Function
s64vector? *value* Function
u64vector? *value* Function
f32vector? *value* Function
f64vector? *value* Function

Return true iff *value* is a uniform vector of the specified type.

make-s8vector *n* [*value*] Function
make-u8vector *n* [*value*] Function
make-s16vector *n* [*value*] Function
make-u16vector *n* [*value*] Function
make-s32vector *n* [*value*] Function
make-u32vector *n* [*value*] Function
make-s64vector *n* [*value*] Function
make-u64vector *n* [*value*] Function
make-f32vector *n* [*value*] Function
make-f64vector *n* [*value*] Function

Create a new uniform vector of the specified type, having room for *n* elements. Initialize each element to *value* if it is specified; zero otherwise.

s8vector *value ...* Function
u8vector *value ...* Function
s16vector *value ..* Function
u16vector *value ...* Function
s32vector *value ...* Function
u32vector *value ...* Function
s64vector *value ...* Function
u64vector *value ...* Function
f32vector *value ...* Function
f64vector *value ...* Function

Create a new uniform vector of the specified type, whose length is the number of *value*s specified, and initialize it using those *value*s.

s8vector-length *v* Function
u8vector-length *v* Function
s16vector-length *v* Function
u16vector-length *v* Function
s32vector-length *v* Function
u32vector-length *v* Function
s64vector-length *v* Function
u64vector-length *v* Function
f32vector-length *v* Function
f64vector-length *v* Function

Return the length (in number of elements) of the uniform vector *v*.

s8vector-ref *v i* Function
u8vector-ref *v i* Function
s16vector-ref *v i* Function
u16vector-ref *v i* Function
s32vector-ref *v i* Function
u32vector-ref *v i* Function
s64vector-ref *v i* Function
u64vector-ref *v i* Function
f32vector-ref *v i* Function
f64vector-ref *v i* Function

Return the element at index *i* of the uniform vector *v*.

s8vector-set! *v i x* Function
u8vector-set! *v i x* Function
s16vector-set! *v i x* Function
u16vector-set! *v i x* Function
s32vector-set! *v i x* Function
u32vector-set! *v i x* Function
s64vector-set! *v i x* Function
u64vector-set! *v i x* Function
f32vector-set! *v i x* Function
f64vector-set! *v i x* Function

Set the element at index *i* of uniform vector *v* to the value *x*, which must be a number coercible to the appropriate type.

s8vector->list *v* Function
u8vector->list *v* Function
s16vector->list *v* Function
u16vector->list *v* Function
s32vector->list *v* Function
u32vector->list *v* Function
s64vector->list *v* Function
u64vector->list *v* Function
f32vector->list *v* Function
f64vector->list *v* Function

Convert the uniform vetor *v* to a list containing the elments of *v*.

list->s8vector *l* Function
list->u8vector *l* Function
list->s16vector *l* Function
list->u16vector *l* Function
list->s32vector *l* Function
list->u32vector *l* Function
list->s64vector *l* Function
list->u64vector *l* Function
list->f32vector *l* Function
list->f64vector *l* Function

Create a uniform vector of the appropriate type, initializing it with the elements of the list *l*. The elements of *l* must be numbers coercible the new vector's element type.

7.11.1 Relationship with Java arrays

Each uniform array type is implemented as an *underlying Java array*, and a length field. The underlying type is `byte[]` for `<u8vector>` or `<s8vector>`; `short[]` for `<u16vector>` or `<u16vector>`; `int[]` for `<u32vector>` or `<s32vector>`; `long[]` for `<u64vector>` or `<s64vector>`; `<float[]` for `<f32vector>`; and `<double[]` for `<f32vector>`. The length field allows a uniform array to only use the initial part of the underlying array. (This can be used to support Common Lisp's fill pointer feature.) This also allows resizing a uniform vector. There is no Scheme function for this, but you can use the `setSize` method:

```
(invoke some-vector 'setSize 200)
```

If you have a Java array, you can create a uniform vector sharing with the Java array:

```
(define arr :: <byte[]> ((primitive-array-new <byte>) 10))
(define vec :: <u8vector> (make <u8vector> arr))
```

At this point `vec` uses `arr` for its underlying storage, so changes to one affect the other. It `vec` is re-sized so it needs a larger underlying array, then it will no longer use `arr`.

7.12 Signalling and recovering from exceptions

catch *key thunk handler* Function

Invoke *thunk* in the dynamic context of *handler* for exceptions matching *key*. If thunk throws to the symbol *key*, then *handler* is invoked this way:

```
(handler key args ...)
```

key may be a symbol. The *thunk* takes no arguments. If *thunk* returns normally, that is the return value of `catch`.

Handler is invoked outside the scope of its own `catch`. If *handler* again throws to the same key, a new handler from further up the call chain is invoked.

If the key is `#t`, then a throw to *any* symbol will match this call to `catch`.

throw *key* &rest *args ...* Function

Invoke the catch form matching *key*, passing *args* to the *handler*.

If the key is a symbol it will match catches of the same symbol or of #t.

If there is no handler at all, an error is signaled.

error *message args ...* procedure
Raise an error with key `misc-error` and a message constructed by displaying *msg* and writing *args*. This normally prints a stack trace, and brings you back to the top level, or exits kawa if you are not running interactively. This procedure is part of SRFI-23, and other Scheme implementations.

primitive-throw *exception* Function
Throws the *exception*, which must be an instance of a sub-class of `<java.lang.Throwable>`.

try-finally *body handler* Syntax
Evaluate *body*, and return its result. However, before it returns, evaluate *handler*. Even if *body* returns abnormally (by throwing an exception), *handler* is evaluated.

(This is implemented just like Java's `try-finally`.)

try-catch *body handler ...* Syntax
Evaluate *body*, in the context of the given *handler* specifications. Each *handler* has the form:

var type exp ...

If an exception is thrown in *body*, the first *handler* is selected such that the thrown exception is an instance of the *handler*'s *type*. If no *handler* is selected, the exception is propagated through the dynamic execution context until a matching *handler* is found. (If no matching *handler* is found, then an error message is printed, and the computation terminated.)

Once a *handler* is selected, the *var* is bound to the thrown exception, and the *exp* in the *handler* are executed. The result of the `try-catch` is the result of *body* if no exception is thrown, or the value of the last *exp* in the selected *handler* if an exception is thrown.

(This is implemented just like Java's `try-catch`.)

dynamic-wind *in-guard thunk out-guard* Function
All three arguments must be 0-argument procedures. First calls *in-guard*, then *thunk*, then *out-guard*. The result of the expression is that of *thunk*. If *thunk* is exited abnormally (by throwing an exception or invoking a continuation), *out-guard* is called.

If the continuation of the dynamic-wind is re-entered (which is not yet possible in Kawa), the *in-guard* is called again.

This function was added in R5RS.

7.13 Locations

A *location* is a place where a value can be stored. An *lvalue* is an expression that refers to a location. (The name "lvalue" refers to the fact that the left operand of `set!` is an lvalue.) The only kind of lvalue in standard Scheme is a *variable*. Kawa also allows *computed lvalues*. These are procedure calls used in "lvalue context", such as the left operand of `set!`.

You can only use procedures that have an associated *setter*. In that case, `(set! (f arg ...) value)` is equivalent to `((setter f) arg ... value)` Currently, only a few procedures have associated `setter`s, and only builtin procedures written in Java can have `setter`s.

For example:

```
(set! (car x) 10)
```

is equivalent to:

```
((setter car) x 10)
```

which is equivalent to:

```
(set-car! x 10)
```

setter *procedure* Function

Gets the "setter procedure" associated with a "getter procedure". Equivalent to `(procedure-property` *procedure* `'setter)`. By convention, a setter procedure takes the same parameters as the "getter" procedure, plus an extra parameter that is the new value to be stored in the location specified by the parameters. The expectation is that following `((setter` *proc*`)` *args* `...` *value*`)` then the value of `(`*proc args* `...)` will be *value*.

The `setter` of `setter` can be used to set the `setter` property. For example the Scheme prologue effectively does the following:

```
(set! (setter vector-set) vector-set!)
```

Kawa also gives you access to locations as first-class values:

location *lvalue* Syntax

Returns a location object for the given *lvalue*. You can get its value (by applying it, as if it were a procedure), and you can set its value (by using `set!` on the application). The *lvalue* can be a local or global variable, or a procedure call using a procedure that has a `setter`.

```
(define x 100)
(define lx (location x))
(set! (lx) (cons 1 2)) ;; set x to (1 . 2)
(lx)  ;; returns (1 . 2)
(define lc (location (car x)))
(set! (lc) (+ 10 (lc)))
;; x is now (11 . 2)
```

define-alias *variable lvalue* Syntax

Define *variable* as an alias for *lvalue*. In other words, makes it so that `(location` *variable*`)` is equivalent to `(location` *lvalue*`)`. This works both top-level and inside a function.

Some people might find it helpful to think of a location as a settable *thunk*. Others may find it useful to think of the `location` syntax as similar to the C '`&`' operator; for the '`*`' indirection operator, Kawa uses procedure application.

7.14 Eval and Environments

eval *expression* [*environment*] Function

`eval` evaluates *expression* in the environment indicated by *environment*.

The default for *environment* is the result of `(interaction-environment)`.

null-environment *version* Function

This procedure returns an environment that contains no variable bindings, but contains (syntactic) bindings for all the syntactic keywords.

The effect of assigning to a variable in this environment (such as `let`) is undefined.

scheme-report-environment *version* Function

The *version* must be an exact non-negative inetger corresponding to a version of one of the Revised*version* Reports on Scheme. The procedure returns an environment that contains exactly the set of bindings specified in the corresponding report.

This implementation supports *version* that is 4 or 5.

The effect of assigning to a variable in this environment (such as `car`) is undefined.

interaction-environment Function

This procedure return an environment that contains implementation-defined bindings, as well as top-level user bindings.

environment-bound? *environment symbol* Function

Return true `#t` if there is a binding for *symbol* in *environment*; otherwise returns `#f`.

fluid-let ((*variable init*) ...) *body ...* Syntax

Evaluate the *init* expressions. Then modify the dynamic bindings for the *variables* to the values of the *init* expressions, and evaluate the *body* expressions. Return the result of the last expression in *body*. Before returning, restore the original bindings. The temporary bindings are only visible in the current thread, and its descendent threads.

base-uri [*node*] Function

If *node* is specified, returns the base-URI property of the *node*. If the *node* does not have the base-URI property, returns `#f`. (The XQuery version returns the empty sequence in that case.)

In the zero-argument case, returns the "base URI" of the current context. By default the base URI is the current working directory (as a URL). While a source file is `load`ed, the base URI is temporarily set to the URL of the document.

load *path* Function

The *path* can be an (absolute) URL or a filename.

load-relative *path* Function

Same as `load`, except that *path* is a URI that is relative to the context's current base URI.

7.15 Debugging

trace *procedure* Syntax

Cause *procedure* to be "traced", that is debugging output will be written to the standard error port every time *procedure* is called, with the parameters and return value.

untrace *procedure* Syntax

Turn off tracing (debugging output) of *procedure*.

7.16 Threads

There is a very preliminary interface to create parallel threads. The interface is similar to the standard `delay`/`force`, where a thread is basically the same as a promise, except that evaluation may be in parallel.

So far, only modest effort has been made into making Kawa thread-safe.

future *expression* Syntax

Creates a new thread that evaluates *expression*.

force *thread* Function

The standard `force` function has generalized to also work on threads. If waits for the thread's *expression* to finish executing, and returns the result.

sleep *time* Function

Suspends the current thread for the specified time. The *time* can be either a pure number (in secords), or a quantity whose unit is a time unit (such as `10s`).

7.17 Processes

make-process *command envp* Function

Creates a `<java.lang.Process>` object, using the specified *command* and *envp*. The *command* is converted to an array of Java strings (that is an object that has type `<java.lang.String[]>`. It can be a Scheme vector or list (whose elements should be Java strings or Scheme strings); a Java array of Java strings; or a Scheme string. In the latter case, the command is converted using `command-parse`. The *envp* is process environment; it should be either a Java array of Java strings, or the special `#!null` value.

system *command* Function

Runs the specified *command*, and waits for it to finish. Returns the return code from the command. The return code is an integer, where 0 conventionally means successful completion. The *command* can be any of the types handled by `make-process`.

command-parse Variable

The value of this variable should be a one-argument procedure. It is used to convert a command from a Scheme string to a Java array of the constituent "words". The default binding, on Unix-like systems, returns a new command to invoke `"/bin/sh" "-c"` concatenated with the command string; on non-Unix-systems, it is bound to `tokenize-string-to-string-array`.

tokenize-string-to-string-array *command* Function

Uses a `java.util.StringTokenizer` to parse the *command* string into an array of words. This splits the *command* using spaces to delimit words; there is no special processing for quotes or other special characters. (This is the same as what `java.lang.Runtime.exec(String)` does.)

7.18 Miscellaneous

scheme-implementation-version Function

Returns the Kawa version number as a string.

command-line-arguments Variable

Any command-line arguments (following flags processed by Kawa itself) are assigned to the global variable '`command-line-arguments`', which is a vector of strings.

home-directory Variable

A string containing the home directory of the user.

exit [*code*] Function

Exits the Kawa interpreter, and ends the Java session. The integer value *code* is returned to the operating system. If *code* is not specified, zero is returned, indicating normal (non-error) termination.

scheme-window [*shared*] Function

Create a read-eval-print-loop in a new top-level window. If *shared* is true, it uses the same environment as the current `(interaction-environment)`; if not (the default), a new top-level environment is created.

You can create multiple top-level window that can co-exist. They run in separate threads.

when *condition form...* Syntax

If *condition* is true, evaluate each *form* in order, returning the value of the last one.

unless *condition form...* Syntax

If *condition* is false, evaluate each *form* in order, returning the value of the last one.

vector-append *arg...* Function

Creates a new vector, containing the elements from all the *arg*s appended together. Each *arg* may be a vector or a list.

instance? *value type* Function
Returns `#t` iff *value* is an instance of type *type*. (Undefined if *type* is a primitive type, such as `<int>`.)

as *type value* Function
Converts or coerces *value* to a value of type *type*. Throws an exception if that cannot be done. Not supported for *type* to be a primitive type such as `<int>`.

synchronized *object form ...* Syntax
Synchronize on the given *object*. (This means getting an exclusive lock on the object, by acquiring its *monitor*.) Then execute the *form*s while holding the lock. When the *form*s finish (normally or abnormally by throwing an exception), the lock is released. Returns the result of the last *form*. Equivalent to the Java `synchronized` statement, except that it may return a result.

8 Input, output, and file handling

Kawa has a number of useful tools for controlling input and output:

A programmable reader.

A powerful pretty-printer.

The `--output-format` (or `--format`) command-line switch can be used to override the default format for how values are printed on the standard output. This format is used for values printed by the read-eval-print interactive interface. It is also used to control how values are printed when Kawa evaluates a file named on the command line (using the `-f` flag or a just a script name). (It also effects applications compiled with the `--main` flag.) It currently effects how values are printed by a `load`, though that may change.

The default format depends on the current programming language. For Scheme, the default is `--scheme` for read-eval-print interaction, and `--ignore` for files that are loaded.

The formats currently supported include the following:

‘`scheme`’ Values are printed in a format matching the Scheme programming language, as if using `display`. `"Groups"` or `"elements"` are written as lists.

‘`readable-scheme`’
Like `scheme`, as if using `write`: Values are generally printed in a way that they can be read back by a Scheme reader. For example, strings have quotation marks, and character values are written like ‘`#\A`’.

‘`elisp`’ Values are printed in a format matching the Emacs Lisp programming language. Mostly the same as `scheme`.

‘`readable-elisp`’
Like `elisp`, but values are generally printed in a way that they can be read back by an Emacs Lisp reader. For example, strings have quotation marks, and character values are written like ‘`?A`’.

'`clisp`'
'`commonlisp`'
: Values are printed in a format matching the Common Lisp programming language, as if written by `princ`. Mostly the same as `scheme`.

'`readable-clisp`'
'`readable-commonlisp`'
: Like `clisp`, but as if written by `prin1`: values are generally printed in a way that they can be read back by a Common Lisp reader. For example, strings have quotation marks, and character values are written like '`#\A`'.

'`xml`'
: Values are printed in XML format. `"Groups"` or `"elements"` are written as using xml element syntax. Plain characters (such as '`<`') are escaped (such as '`<`').

'`xhtml`'
: Same as `xml`, but follows the xhtml compatibility guidelines.

'`html`'
: Values are printed in HTML format. Mostly same as `xml` format, but certain element without body, are written without a closing tag. For example `<img>` is written without `</img>`, which would be illegal for html, but required for xml. Plain characters (such as '`<`') are not escaped inside `<script>` or `<style>` elements.

'`cgi`'
: The output should be a follow the CGI standards. I.e. assume that this script is invoked by a web server as a CGI script/program, and that the output should start with some response header, followed by the actual response data. To generate the response headers, use the `response-header` function. If the `Content-type` response header has not been specified, and it is required by the CGI standard, Kawa will attempt to infer an appropriate `Content-ty[e` depending on the following value.

'`ignore`'
: Top-level values are ignored, instead of printed.

8.1 File System Interface

file-exists? *filename* Function
: Returns true iff the file named *filename* actually exists.

file-directory? *filename* Function
: Returns true iff the file named *filename* actually exists and is a directory.

file-readable? *filename* Function
: Returns true iff the file named *filename* actually exists and can be read from.

file-writable? *filename* Function
: Returns true iff the file named *filename* actually exists and can be writen to. (Undefined if the *filename* does not exist, but the file can be created in the directory.)

delete-file *filename* Function
: Delete the file named *filename*.

rename-file *oldname newname* Function
Renames the file named *oldname* to *newname*.

copy-file *oldname newname-from path-to* Function
Copy the file named *oldname* to *newname*. The return value is unspecified.

create-directory *dirname* Function
Create a new directory named *dirname*. Unspecified what happens on error (such as exiting file with the same name). (Currently returns `#f` on error, but may change to be more compatible with scsh.)

system-tmpdir Function
Return the name of the default directory for temporary files.

make-temporary-file [*format*] Function
Return a file with a name that does not match any existing file. Use *format* (which defaults to `"kawa~d.tmp"`) to generate a unique filename in `(system-tmpdir)`. The current implementation is *not* safe from race conditions; this will be fixed in a future release (using Java2 features).

8.2 Ports

current-error-port Function
Return the port to which errors and warnings should be sent (the *standard error* in Unix and C terminology).

read-line [*port* [*handle-newline*]] Function
Reads a line of input from *port*. The *handle-newline* parameter determines what is done with terminating end-of-line delimiter. The default, `'trim`, ignores the delimiter; `'peek` leaves the delimiter in the input stream; `'concat` appends the delimiter to the returned value; and `'split` returns the delimiter as a second value. You can use the last three options to tell if the string was terminated by end-or-line or by end-of-file.

open-input-string *string* Function
Takes a string and returns an input port that delivers characters from the string. The port can be closed by `close-input-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

```
(define p
  (open-input-string "(a . (b c . ())) 34"))

(input-port? p)                 -->  #t
(read p)                        -->  (a b c)
(read p)                        -->  34
(eof-object? (peek-char p))     -->  #t
```

open-output-string Function
Returns an output port that will accumulate characters for retrieval by `get-output-string`. The port can be closed by the procedure `close-output-port`,

though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

```
(let ((q (open-output-string))
  (x '(a b c)))
    (write (car x) q)
    (write (cdr x) q)
    (get-output-string q))        -->  "a(b c)"
```

get-output-string *output-port* Function
Given an output port created by `open-output-string`, returns a string consisting of the characters that have been output to the port so far.

call-with-input-string *string proc* Function
Create an input port that gets its data from *string*, call *proc* with that port as its one argument, and return the result from the call of *proc*

call-with-output-string *proc* Function
Create an output port that writes its data to a *string*, and call *proc* with that port as its one argument. Return a string consisting of the data written to the port.

force-output [*port*] Function
Forces any pending output on *port* to be delivered to the output device and returns an unspecified value. If the *port* argument is omitted it defaults to the value returned by `(current-output-port)`.

An interactive input port has a prompt procedure associated with it. The prompt procedure is called before a new line is read. It is passed the port as an argument, and returns a string, which gets printed as a prompt.

input-port-prompter *port* Function
Get the prompt procedure associated with *port*.

set-input-port-prompter! *port prompter* Function
Set the prompt procedure associated with *port* to *prompter*, which must be a one-argument procedure taking an input port, and returning a string.

default-prompter *port* Function
The default prompt procedure. It returns `"#|kawa:`*L*`|# "`, where *L* is the current line number of *port*. When reading a continuation line, the result is `"#|`*C*`--:`*L*`|# "`, where `C` is the character returned by `(input-port-read-state` *port*`)`. The prompt has the form of a comment to make it easier to cut-and-paste.

port-column *input-port* Function
port-line *input-port* Function
Return the current column number or line number of *input-port*, using the current input port if none is specified. If the number is unknown, the result is `#f`. Otherwise, the result is a 0-origin integer - i.e. the first character of the first line is line 0, column 0. (However, when you display a file position, for

example in an error message, we recommend you add 1 to get 1-origin integers. This is because lines and column numbers traditionally start with 1, and that is what non-programmers will find most natural.)

set-port-line! *port line* Function
Set (0-origin) line number of the current line of *port* to *num*.

input-port-line-number *port* Function
Get the line number of the current line of *port*, which must be a (non-binary) input port. The initial line is line 1. Deprecated; replaced by `(+ 1 (port-line` *port*`))`.

set-input-port-line-number! *port num* Function
Set line number of the current line of *port* to *num*. Deprecated; replaced by `(set-port-line!` *port* `(-` *num* `1))`.

input-port-column-number *port* Function
Get the column number of the current line of *port*, which must be a (non-binary) input port. The initial column is column 1. Deprecated; replaced by `(+ 1 (port-column` *port*`))`.

input-port-read-state *port* Function
Returns a character indicating the current `read` state of the *port*. Returns `#\Return` if not current doing a *read*, `#\"` if reading a string; `#\|` if reading a comment; `#\(` if inside a list; and `#\Space` when otherwise in a `read`. The result is intended for use by prompt prcedures, and is not necessarily correct except when reading a new-line.

symbol-read-case Variable
A symbol that controls how `read` handles letters when reading a symbol. If the first letter is '`U`', then letters in symbols are upper-cased. If the first letter is '`D`' or '`L`', then letters in symbols are down-cased. If the first letter is '`I`', then the case of letters in symbols is inverted. Otherwise (the default), the letter is not changed. (Letters following a '\' are always unchanged.)

port-char-encoding Variable
Controls how bytes in external files are converted to/from internal Unicode characters. Can be either a symbol or a boolean. If `port-char-encoding` is `#f`, the file is assumed to be a binary file and no conversion is done. Otherwise, the file is a text file. The default is `#t`, which uses a locale-dependent conversion. If `port-char-encoding` is a symbol, it must be the name of a character encoding known to Java. For all text files (that is if `port-char-encoding` is not `#f`), on input a `#\Return` character or a `#\Return` followed by `#\Newline` are converted into plain `#\Newline`.

This variable is checked when the file is opened; not when actually reading or writing. Here is an example of how you can safely change the encoding temporarily:

```
(define (open-binary-input-file name)
  (fluid-let ((port-char-encoding #f)) (open-input-file name)))
```

8.3 Formatted Output (Common-Lisp-style)

format *destination fmt . arguments* Function

An almost complete implementation of Common LISP format description according to the CL reference book *Common LISP* from Guy L. Steele, Digital Press. Backward compatible to most of the available Scheme format implementations.

Returns `#t`, `#f` or a string; has side effect of printing according to *fmt*. If *destination* is `#t`, the output is to the current output port and `#!void` is returned. If *destination* is `#f`, a formatted string is returned as the result of the call. If *destination* is a string, *destination* is regarded as the format string; *fmt* is then the first argument and the output is returned as a string. If *destination* is a number, the output is to the current error port if available by the implementation. Otherwise *destination* must be an output port and `#!void` is returned.

fmt must be a string or an instance of `gnu.text.MessageFormat` or `java.text.MessageFormat`. If *fmt* is a string, it is parsed as if by `parse-format`.

parse-format *format-string* Function

Parses `format-string`, which is a string of the form of a Common LISP format description. Returns an instance of `gnu.text.ReportFormat`, which can be passed to the `format` function.

A format string passed to `format` or `parse-format` consists of format directives (that start with '`~`'), and regular characters (that are written directly to the destination). Most of the Common Lisp (and Slib) format directives are implemented. Neither justification, nor pretty-printing are supported yet.

Plus of course, we need documentation for `format`!

8.3.1 Implemented CL Format Control Directives

Documentation syntax: Uppercase characters represent the corresponding control directive characters. Lowercase characters represent control directive parameter descriptions.

`~A` Any (print as `display` does).

`~@A` left pad.

`~`*mincol*`,`*colinc*`,`*minpad*`,`*padchar*`A` full padding.

`~S` S-expression (print as `write` does).

`~@S` left pad.

`~`*mincol*`,`*colinc*`,`*minpad*`,`*padchar*`S` full padding.

`~C` Character.

`~@C` prints a character as the reader can understand it (i.e. `#\` prefixing).

`~:C` prints a character as emacs does (eg. `^C` for ASCII 03).

8.3.2 Formatting Integers

`~D` Decimal.

- `~@D` print number sign always.
- `~:D` print comma separated.
- `~`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`D` padding.

`~X` Hexadecimal.

- `~@X` print number sign always.
- `~:X` print comma separated.
- `~`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`X` padding.

`~O` Octal.

- `~@O` print number sign always.
- `~:O` print comma separated.
- `~`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`O` padding.

`~B` Binary.

- `~@B` print number sign always.
- `~:B` print comma separated.
- `~`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`B` padding.

`~`*n*`R` Radix *n*.

- `~`*n*`,`*mincol*`,`*padchar*`,`*commachar*`,`*commawidth*`R` padding.

`~@R` print a number as a Roman numeral.

`~:@R` print a number as an "old fashioned" Roman numeral.

`~:R` print a number as an ordinal English number.

`~R` print a number as a cardinal English number.

`~P` Plural.

- `~@P` prints `y` and `ies`.
- `~:P` as `~P but jumps 1 argument backward.`
- `~:@P` as `~@P but jumps 1 argument backward.`

commawidth is the number of characters between two comma characters.

8.3.3 Formatting floating-point (real) numbers

`~F` Fixed-format floating-point (prints a flonum like *mmm.nnn*).

`~`*width*`,`*digits*`,`*scale*`,`*overflowchar*`,`*padchar*`F`

`~@F` If the number is positive a plus sign is printed.

`~E` Exponential floating-point (prints a flonum like *mmm.nnn*`E`*ee*)

`~`*width*`,`*digits*`,`*exponentdigits*`,`*scale*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`E`

`~@E` If the number is positive a plus sign is printed.

`~G` General floating-point (prints a flonum either fixed or exponential).

`~`*width*`,`*digits*`,`*exponentdigits*`,`*scale*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`G`

`~@G` If the number is positive a plus sign is printed.

A slight difference from Common Lisp: If the number is printed in fixed form and the fraction is zero, then a zero digit is printed for the fraction, if allowed by the *width* and *digits* is unspecified.

`~$` Dollars floating-point (prints a flonum in fixed with signs separated).

`~`*digits*`,`*scale*`,`*width*`,`*padchar*`$`

`~@$` If the number is positive a plus sign is printed.

`~:@$` A sign is always printed and appears before the padding.

`~:$` The sign appears before the padding.

8.3.4 Miscellaneous formatting operators

`~%` Newline.

`~`*n*`%` print *n* newlines.

`~&` print newline if not at the beginning of the output line.

`~`*n*`&` prints `~&` and then *n-1* newlines.

`~|` Page Separator.

`~`*n*`|` print *n* page separators.

`~~` Tilde.

`~`*n*`~` print *n* tildes.

`~`<newline>
Continuation Line.

`~:`<newline>
newline is ignored, white space left.

`~@`<newline>
newline is left, white space ignored.

`~T` Tabulation.

- `~@T` relative tabulation.
- `~`*colnum*`,`*colinc*`T` full tabulation.

`~?` Indirection (expects indirect arguments as a list).

- `~@?` extracts indirect arguments from format arguments.

`~(`*str*`~)` Case conversion (converts by `string-downcase`).

- `~:(`*str*`~)` converts by `string-capitalize`.
- `~@(`*str*`~)` converts by `string-capitalize-first`.
- `~:@(`*str*`~)` converts by `string-upcase`.

`~*` Argument Jumping (jumps 1 argument forward).

- `~`*n*`*` jumps *n* arguments forward.
- `~:*` jumps 1 argument backward.
- `~`*n*`:*` jumps *n* arguments backward.
- `~@*` jumps to the 0th argument.
- `~`*n*`@*` jumps to the *n*th argument (beginning from 0)

`~[`*str0*`~;`*str1*`~;...~;`*strn*`~]` Conditional Expression (numerical clause conditional).

- `~`*n*`[` take argument from *n*.
- `~@[` true test conditional.
- `~:[` if-else-then conditional.
- `~;` clause separator.
- `~:;` default clause follows.

`~{`*str*`~}` Iteration (args come from the next argument (a list)).

- `~`*n*`{` at most *n* iterations.
- `~:{` args from next arg (a list of lists).
- `~@{` args from the rest of arguments.
- `~:@{` args from the rest args (lists).

`~^` Up and out.

- `~`*n*`^` aborts if $n = 0$
- `~`*n*`,`*m*`^` aborts if $n = m$
- `~`*n*`,`*m*`,`*k*`^` aborts if $n <= m <= k$

8.3.5 Not Implemented CL Format Control Directives

`~:A` print `#f` as an empty list (see below).

`~:S` print `#f` as an empty list (see below).

`~<~>` Justification.

`~:^`

8.3.5.1 Extended, Replaced and Additional Control Directives

These are not necesasrily implemented in Kawa!

`~I` print a R4RS complex number as `~F~@Fi` with passed parameters for `~F`.

`~Y` Pretty print formatting of an argument for scheme code lists.

`~K` Same as `~?`.

`~!` Flushes the output if format *destination* is a port.

`~_` Print a `#\space` character

`~`*n*`_` print *n* `#\space` characters.

`~`*n*`C` Takes *n* as an integer representation for a character. No arguments are consumed. *n* is converted to a character by `integer->char`. *n* must be a positive decimal number.

`~:S` Print out readproof. Prints out internal objects represented as `#<...>` as strings `"#<...>"` so that the format output can always be processed by `read`.

`~:A` Print out readproof. Prints out internal objects represented as `#<...>` as strings `"#<...>"` so that the format output can always be processed by `read`.

`~F, ~E, ~G, ~$`
may also print number strings, i.e. passing a number as a string and format it accordingly.

9 Types

A *type* is a set of values, plus an associated set of operations valid on those values. Types are useful for catching errors ("type-checking"), documenting the programmer's intent, and to help the compiler generate better code. Types in some languages (such as C) appear in programs, but do not exist at run-time. In such languages, all type-checking is done at compile-time. Other languages (such as standard Scheme) do not have types as such, but they have *predicates*, which allow you to check if a value is a member of certain sets; also, the primitive functions will check at run-time if the arguments are members of the allowed sets. Other languages, including Java and Common Lisp, provide a combination: Types may be used as specifiers to guide the compiler, but also exist as actual run-time values. In Java, for each class, there is a corresponding `java.lang.Class` run-time object, as well as an associated type (the set of values of that class, plus its sub-classes, plus `null`).

Kawa, like Java, has first-class types, that is types exist as objects you can pass around at run-time. For each Java type, there is a corresponding Kawa type (but not necessarily vice versa). It would be nice if we could represent run-time type values using `java.lang.Class` objects, but unfortunately that does not work very well. One reason is that we need to be able to refer to types and classes that do not exist yet, because we are in the processing of compiling them. Another reason is that we want to be able to distinuish between different types that are implemented using the same Java class.

Various Kawa constructs require or allow a type to be specified. Those specifications consist of *type expressions*, which is evaluated to yield a type value. The current Kawa compiler is rather simple-minded, and in many places only allows simple types that the compiler can evaluate at compile-time. More specifically, it only allows simple *type names* that map to primitive Java types or java classes.

9.1 Standard Types

These types are bound to identifiers having the form `<TYPENAME>`. (This syntax and most of the names are as in RScheme.)

To find which Java classes these types map into, look in `kawa/standard/Scheme.java`.

Note that the value of these variables are instances of `gnu.bytecode.Type`, not (as you might at first expect) `java.lang.Class`.

<object> Variable
An arbitrary Scheme value - and hence an arbitrary Java object.

<number> Variable
The type of Scheme numbers.

<quantity> Variable
The type of quantities optionally with units. This is a sub-type of `<number>`.

<complex> Variable
The type of complex numbers. This is a sub-type of `<quantity>`.

<real> Variable
The type of real numbers. This is a sub-type of `<complex>`.

<rational> Variable
The type of exact rationl numbers. This is a sub-type of `<real>`.

<integer> Variable
The type of exact Scheme integers. This is a sub-type of `<rational>`.

<symbol> Variable
The type of Scheme symbols.

<keyword> Variable
The type of keyword values. See Section 7.4 [Keywords], page 20.

<list> Variable
The type of Scheme lists (pure and impure, including the empty list).

<pair> Variable
The type of Scheme pairs. This is a sub-type of `<list>`.

<string> Variable
The type of (mutable) Scheme strings. This is *not* the same as (non-mutable) Java strings (which happen to be the same as `<symbol>`).

<character> Variable
The type of Scheme character values. This is a sub-type of `<object>`, in contrast to type `<char>`, which is the primitive Java `char` type.

<vector> Variable
The type of Scheme vectors.

<procedure> Variable
The type of Scheme procedures.

<input-port> Variable
The type of Scheme input ports.

<output-port> Variable
The type of Scheme output ports.

<String> Variable
This type name is a special case. It specifies the class `<java.lang.String>` (just as `<symbol>` does). However, coercing a value to `<String>` is done by invoking the `toString` method on the value to be coerced. Thus it `"works"` for all objects. It also works for `#!null`.

When Scheme code invokes a Java methods any parameter whose type is `java.lang.String` is converted as if it was decalred as a `<String>`.

More will be added later.

A type specifier can also be one of the primitive Java types. The numeric types `<long>`, `<int>`, `<short>`, `<byte>`, `<float>`, and `<double>` are converted from the corresponding Scheme number classes. Similarly, `<char>` can be converted to and from Scheme characters. The type `boolean` matches any object, and the result is `false` if and only if the actual argument is `#f`. (The value `#f` is identical to `Boolean.FALSE`, and `#t` is identical to `Boolean.TRUE`.) The return type `<void>` indicates that no value is returned.

A type specifier can also be a fully-qualified Java class name (for example `<java.lang.StringBuffer>`). In that case, the actual argument is cast at run time to the named class. Also, `<java.lang.StringBuffer[]` represents an array of references to `java.lang.StringBuffer` objects.

9.2 Declaring Types of Variables

let ((*name* [:: *type*] *init*) ...) *body* Syntax

Declare new locals variables with the given *name*, initial value *init*, and optional type specification *type*. If *type* is specified, then the expression *init* is evaluated, the result coerced to *type*, and then assigned to the variable. If *type* is not specified, it defaults to `<object>`.

let* ((*name* [:: *type*] *init*) ...) *body* Syntax

letrec ((*name* [:: *type*] *init*) ...) *body* Syntax

define [:: *type*] *value* Syntax

See also `define-private`, and `define-constant`.

10 Object, Classes and Modules

Kawa provides various ways to define, create, and access Java objects. Here are the currently supported features.

The Kawa module system is based on the features of the Java class system.

this Syntax

Returns the "this object" - the current instance of the current class. The current implementation is incomplete, not robust, and not well defined. However, it will have to do for now. Note: `"this"` is a macro, not a variable, so you have to write it using parentheses: '`(this)`'. A planned extension will allow an optional class specifier (needed for nested clases).

The `define-record-type` form can be used for creating new data types, called record types. A predicate, constructor, and field accessors and modifiers are defined for each record type. The `define-record-type` feature is specified by SRFI-9 (`http://srfi.schemers.org/srfi-9/srfi`
which is implemented by many modern Scheme implementations.

define-record-type *type-name* (*constructor-name field-tag ...*) Syntax
predicate-name (*field-tag accessor-name* [*modifier-name*]) ...

The form `define-record-type` is generative: each use creates a new record type that is distinct from all existing types, including other record types and Scheme's predefined types. Record-type definitions may only occur at top-level (there are two possible semantics for 'internal' record-type definitions, generative and nongenerative, and no consensus as to which is better).

An instance of `define-record-type` is equivalent to the following definitions:

- The *type-name* is bound to a representation of the record type itself.
- The *constructor-name* is bound to a procedure that takes as many arguments as there are *field-tag*s in the (*constructor-name* `...`) subform and returns a new *type-name* record. Fields whose tags are listed with *constructor-name* have the corresponding argument as their initial value. The initial values of all other fields are unspecified.

- The *predicate-name* is a predicate that returns `#t` when given a value returned by *constructor-name* and `#f` for everything else.
- Each *accessor-name* is a procedure that takes a record of type *type-name* and returns the current value of the corresponding field. It is an error to pass an accessor a value which is not a record of the appropriate type.
- Each *modifier-name* is a procedure that takes a record of type *type-name* and a value which becomes the new value of the corresponding field. The result (in Kawa) is the empty value `#!void`. It is an error to pass a modifier a first argument which is not a record of the appropriate type.

Set!ing the value of any of these identifiers has no effect on the behavior of any of their original values.

Here is an example of how you can define a record type named `pare` with two fields `x` and `y`:

```
(define-record-type pare
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

The above defines `kons` to be a constructor, `kar` and `kdr` to be accessors, `set-kar!` to be a modifier, and `pare?` to be a predicate for `pare`s.

```
(pare? (kons 1 2))        --> #t
(pare? (cons 1 2))        --> #f
(kar (kons 1 2))          --> 1
(kdr (kons 1 2))          --> 2
(let ((k (kons 1 2)))
  (set-kar! k 3)
  (kar k))                --> 3
```

The Kawa compiler creates a new Java class with a name derived from the *type-name*. If the *type-name* is valid Java class name, that becomes the name of the Java class. If the *type-name* has the form `<`*name*`>` (for example `<pare>`), then *name* is used, if possible, for the Java class name. Otherwise, the name of the Java class is derived by "mangling" the *type-name*. In any case, the package is the same as that of the surrounding module.

Kawa generates efficient code for the *predicate-name*, *accessor-name*, and *modifier-name* functions. The *constructor-name* currently compiles into code using run-time reflection, but hopefully that will get fixed.

10.1 Creating New Record Types On-the-fly

Calling the `make-record-type` procedure creates a new record data type at run-time, without any compile-time support. It is primarily provided for compatibility; in most cases it is better to use the `define-record-type` form (see ⟨undefined⟩ [Record types], page ⟨undefined⟩).

make-record-type *type-name field-names* Function

Returns a *record-type descriptor*, a value representing a new data type disjoint from all others. The *type-name* argument must be a string, but is only used for debugging purposes (such as the printed representation of a record of the new type). The *field-names* argument is a list of symbols naming the *fields* of a record of the new type. It is an error if the list contains any duplicates.

record-constructor *rtd* [*field-names*] Function

Returns a procedure for constructing new members of the type represented by *rtd*. The returned procedure accepts exactly as many arguments as there are symbols in the given list, *field-names*; these are used, in order, as the initial values of those fields in a new record, which is returned by the constructor procedure. The values of any fields not named in that list are unspecified. The *field-names* argument defaults to the list of field names in the call to `make-record-type` that created the type represented by *rtd*; if the *field-names* argument is provided, it is an error if it contains any duplicates or any symbols not in the default list.

record-predicate *rtd* Function

Returns a procedure for testing membership in the type represented by *rtd*. The returned procedure accepts exactly one argument and returns a true value if the argument is a member of the indicated record type; it returns a false value otherwise.

record-accessor *rtd field-name* Function

Returns a procedure for reading the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly one argument which must be a record of the appropriate type; it returns the current value of the field named by the symbol *field-name* in that record. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

record-modifier *rtd field-name* Function

Returns a procedure for writing the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly two arguments: first, a record of the appropriate type, and second, an arbitrary Scheme value; it modifies the field named by the symbol *field-name* in that record to contain the given value. The returned value of the modifier procedure is unspecified. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

record? *obj* Function

Returns a true value if *obj* is a record of any type and a false value otherwise.

record-type-descriptor *record* Function

Returns a record-type descriptor representing the type of the given record. That is, for example, if the returned descriptor were passed to `record-predicate`, the resulting predicate would return a true value when passed the given record.

record-type-name *rtd* Function

Returns the type-name associated with the type represented by rtd. The returned value is `eqv?` to the *type-name* argument given in the call to `make-record-type` that created the type represented by *rtd*.

record-type-field-names *rtd* Function

Returns a list of the symbols naming the fields in members of the type represented by *rtd*. The returned value is `equal?` to the field-names argument given in the call to `make-record-type` that created the type represented by *rtd*.

Records are extensions of the class `Record`. These procedures use the Java 1.1 reflection facility.

10.2 Mapping Scheme names to Java names

Programs use "names" to refer to various values and procedures. The definition of what is a "name" is different in different programming languages. A name in Scheme (and other Lisp-like languages) can in principle contain any character (if using a suitable quoting convention), but typically names consist of "words" (one or more letters) separated by hyphens, such as '`make-temporary-file`'. Digits and some special symbols are also used. Standard Scheme is case-insensitive; this means that the names '`loop`', '`Loop`', and '`LOOP`' are all the same name. Kawa is by default case-sensitive, but we recommend that you avoid using upper-case letters as a general rule.

The Java language and the Java virtual machine uses names for classes, variables, fields and methods. These names can contain upper- and lower-case letters, digits, and the special symbols '`_`' and '`$`'.

Given a name in a Scheme program, Kawa needs to map that name into a valid Java name. A typical Scheme name such as '`make-temporary-file`' is not a valid Java name. The convention for Java names is to use "mixed-case" words, such as '`makeTemporaryFile`'. So Kawa will translate a Scheme-style name into a Java-style name. The basic rule is simple: Hyphens are dropped, and a letter that follows a hyphen is translated to its upper-case (actually "title-case") equivalent. Otherwise, letters are translated as is.

Some special characters are handled specially. A final '`?`' is replaced by an *initial* '`is`', with the following letter converted to titlecase. Thus '`number?`' is converted to '`isNumber`' (which fits with Java conventions), and '`file-exists?`' is converted to '`isFileExists`' (which doesn't really). The pair '`->`' is translated to '`$To$`'. For example '`list->string`' is translated to '`list$To$string`'.

Some symbols are mapped to a mnemonic sequence, starting with a dollar-sign, followed by a two-character abbreviation. For example, the less-than symbol '`<`' is mangled as '`$Ls`'. See the source code to the `mangleName` method in the `gnu.expr.Compilation` class for the full list. Characters that do not have a mnemonic abbreviation are mangled as '`$`' followed by a four-hex-digit unicode value. For example '`Tamil vowel sign ai`' is mangled as '`$0bc8`'.

Note that this mapping may map different Scheme names to the same Java name. For example '`string?`', '`String?`', '`is-string`', '`is-String`', and '`isString`' are all mapped to the same Java identifier '`isString`'. Code that uses such "Java-clashing" names is *not*

supported. There is very partial support for renaming names in the case of a clash, and there may be better support in the future. However, some of the nice features of Kawa depend on being able to map Scheme name to Java names naturally, so we urge you to *not* write code that "mixes" naming conventions by using (say) the names '`open-file`' and '`openFile`' to name two different objects.

The above mangling is used to generate Java method names. Each top-level definition is also mapped to a Java field. The name of this field is also mangled, but using a mostly reversible mapping: The Scheme function '`file-exists?`' is mapped to the method name '`file$Mnexists$Qu`'. Because '`$`' is used to encode special characters, you should avoid using it in names in your source file.

10.3 Allocating objects

make *type args ...* Function

Constructs a new object instance of the specified *type*, which must be either a `java.lang.Class` or a `<gnu.bytecode.ClassType>`.

The *args ...* are passed to the constructor of the class type. If there is no applicable constructor, and the *args ...* consist of a set of (*keyword*,*value*)-pairs, then the default constructor is called, and each (*keyword*,*value*)-pair is used to set the correspdong slot of the result, as if by: `(slot-set!` *result keyword value*`)`.

For example, the following are all equivalent:

```
(set! p (make <java.awt.Point> 3 4))

(set! p (make <java.awt.Point> y: 4 x: 3))

(set! p (make <java.awt.Point>))
(slot-set! p 'x 3)
(set! (slot-ref p 'y) 4)
```

10.4 Calling Java methods from Scheme

You can call a Java method as if it were a Scheme procedure using various mechanisms. The most convenient way to do it is to use `define-namespace` to define an alias for a Java class:

```
(define-namespace Int32 "class:java.lang.Integer")
```

In this example the name `Int32` is a *namespace alias* for the namespace whose full name is `"class:java.lang.Integer"`. The full name should be the 6 characters `"class:"` followed by the fully-qualified name of a Java class. You can name a method using a *qualified name* containing a colon. The part of the name before the colon is a namespace alias (in this case `Int32`), and the part of the name after the colon is the method name. For example:

```
(Int32:toHexString 255) => ff
```

This invokes the static method `toHexString` in the Java class `java.lang.Integer`, passing it the argument `255`, and returning the Java String `"ff"`. (Note this is not the same as a Scheme string!)

You can use the method name `new` to construct new objects:

```
(Int32:new '|255|)
```

This is equivalent to the Java expression `new Integer("255")`. You can also write:

```
(Int32:new "255")
```

Kawa is smart enough to convert the Kawa string to a Java String.

You can invoke non-static methods the same way. In that case the first argument is the receiver or `this` argument.

```
(Int32:toString (Int32:new "00255"))
```

This evaluates to the Java String `"255"`.

As a shorthand, you can use the name of a Java class instead of a namespace alias:

```
(java.lang.Integer:toHexString 255)
(java.lang.Object:toString some-value)
```

If Kawa sees a qualified name with a prefix that is not defined *and* that matches the name of an existing class, then Kawa will automaticaly treat the prefix as a nickname for namespace uri like `class:java.lang.Integer`. Both conditions should be true at both compile-time and run-time. However, using an explicit `define-namespace` is recommended.

If you prefer, you can instead use the following functions. (There is also an older deprecated lower-level interface (see Section 11.3 [Low-level Method invocation], page 67.)

invoke-static *class name args ...* Function

The *class* can be a `<java.lang.Class>`, a `<gnu.bytecode.ClassType>`, or a `<symbol>` or `<string>` that names a Java class. The *name* can be `<symbol>` or `<string>` that names one or more methods in the Java class. The name is "mangled" (see Section 10.2 [Mangling], page 54) into a valid Java name.

Any public methods (static or instance) in the specified *class* (or its superclasses) that match "*name*" or "*name*$V" collectively form a generic procedure. When the procedure is applied to the argument list, the most specific applicable method is chosen depending on the argument list; that method is then called with the given arguments. Iff the method is an instance method, the first actual argument is used as the `this` argument. If there are no applicable methods (or no methods at all!), or there is no "best" method, `WrongType` is thrown.

("*name*$V" is used for procedures with `#!rest` or keyword args; the last argument must be an array type; all the "extra" arguments must be compatible with the type of the array elements.)

An example (derived from the Skij FAQ):

```
(invoke-static <java.lang.Thread> 'sleep 100)
```

The behavior of interpreted code and compiled code is not identical, though you should get the same result either way unless you have designed the classes rather strangely. The details will be nailed down later, but the basic idea is that the compiler will "inline" the `invoke-static` call if it can pick a single "best" matching method.

invoke *object name args ...* Function

The *name* can be `<symbol>` or `<string>` that names one or more methods in the Java class. The name is "mangled" (see Section 10.2 [Mangling], page 54) into a valid Java name.

Any public methods (static or instance) in the specified *class* (or its superclasses) that match "*name*" or "*name*$V" collectively form a generic procedure. When the procedure is applied to the argument list, the most specific applicable method is chosen depending on the argument list; that method is then called with the given arguments. Iff the method is an instance method, the *object* is used as the `this` argument; otherwise *object* is prepended to the *args* list. If there are no applicable methods (or no methods at all!), or there is no "best" method, `WrongType` is thrown.

("*name*$V" is used for procedures with `#!rest` or keyword args; the last argument must be an array type; all the "extra" arguments must be compatible with the type of the array elements.)

The behavior of interpreted code and compiled code is not indentical, though you should get the same result either way unless you have designed the classes rather strangely. The details will be nailed down later, but the basic idea is that the compiler will "inline" the `invoke-static` call if it can pick a single "best" matching method.

If the compiler cannot determine the method to call (assuming the method name is constant), the compiler has to generate code at run-time to find the correct method. This is much slower, so the compiler will print a warning. To avoid a waning, you can use a type declaration, or insert a cast:

```
(invoke (as <java.util.Date> my-date) 'setDate cur-date)
```

or

```
(let ((my-date :: <java.util.Date> (calculate-date))
      (cur-date :: <int> (get-cur-date)))
  (invoke my-date 'setDate cur-date))
```

invoke-special *class receiver-object name arg ...* Function

The *class* can be a `<java.lang.Class>`, a `<gnu.bytecode.ClassType>`, or a `<symbol>` or `<string>` that names a Java class. The *name* can be `<symbol>` or `<string>` that names one or more methods in the Java class. The name is "mangled" (see Section 10.2 [Mangling], page 54) into a valid Java name.

This procedure is very similar to `invoke` and `invoke-static` and invokes the specified method, ignoring any methods in subclasses that might overide it. One interesting use is to invoke a method in your super-class like the Java language `super` keyword.

Any methods in the specified *class* that match "*name*" or "*name*$V" collectively form a generic procedure. That generic procedure is then applied as in `invoke` using the `receiver-object` and the arguments (if any).

The compiler must be able to inline this procedure (because you cannot force a specific method to be called using reflection). Therefore the *class* and *name* must resolve at compile-time to a specific method.

```
(define-simple-class <MyClass> (<java.util.Date>)
  ((get-year) :: <int>
   (+ (invoke-special <java.util.Date> (this) 'get-year)) 1900)
  ((set-year (year :: <int>)) :: <void>
   (invoke-special <java.util.Date> (this) 'set-year (- year 1900))))
```

class-methods *class name* Function

Return a generic function containing those methods of *class* that match the name *name*, in the sense of `invoke-static`. Same as:

```
(lambda args (apply invoke-static (cons class (cons name args))))
```

Some examples using these functions are '`vectors.scm`' and '`characters.scm`' the directory '`kawa/lib`' in the Kawa sources.

10.5 Accessing fields of Java objects

Kawa has both a high-level interface and a low-level interface for accessing the fields of Java objects and static fields. The lower-level interfaces are macros that return functions. These functions can be inlined, producing efficient code. The higher-level functions are less verbose and more convenient. However, they can only access public fields.

field *object fieldname* Function

Get the instance field with the given *fieldname* from the given *Object*. Returns the value of the field, which must be public. This procedure has a `setter`, and so can be used as the first operand to `set!`.

The field name is "mangled" (see Section 10.2 [Mangling], page 54) into a valid Java name. If there is no accessible field whose name is "*fieldname*", we look for a no-argument method whose name is "`get`*Fieldname*".

If *object* is a primitive Java array, then *fieldname* can only be `'length`, and the result is the number of elements of the array.

static-field *class fieldname* Function

Get the static field with the given *fieldname* from the given *class*. Returns the value of the field, which must be public. This procedure has a `setter`, and so can be used as the first operand to `set!`.

Examples:

```
(static-field <java.lang.System> 'err)
;; Copy the car field of b into a.
(set! (field a 'car) (field b 'car))
```

slot-ref *object fieldname* Function

A synonym for `(field` *object fieldname*`)`.

slot-set! *object fieldname value* Function

A synonym for `(set! (field` *object fieldname*`)` *value*`)`.

10.6 Defining new classes

Kawa provides various mechanisms for defining new classes. The `define-class` and `define-simple-class` forms will usually be the preferred mechanisms. They have basically the same syntax, but have a couple of differences. `define-class` allows multiple inheritance as well as true nested (first-class) class objects. However, the implementation is more complex: code using it is slightly slower, and the mapping to Java classes is a little less obvious. (Each Scheme class is implemented as a pair of an interface and an implementation class.) A class defined by `define-simple-class` is slightly more efficient, and it is easier to access it from Java code.

The syntax of `define-class` are mostly compatible with that in the Guile and Stk dialects of Scheme.

define-class *name* (*supers ...*) *field-or-method-decl ...* Syntax
define-simple-class *name* (*supers ...*) *field-or-method-decl ...* Syntax
field-or-method ::= *field-decl* | *method-decl*
field-decl ::= (*fname* [:: *ftype*] [*option-keyword option-value*]*)
method-decl ::= ((*method-name formal-arguments*) [:: *rtype*] *body*)

Defines a new class named *name*. If `define-simple-class` is used, creates a normal Java class named *name* in the current package. (If *name* has the form `<xyx>` the Java implementation type is named `xyz`.) If `define-class` the implementation is unspecified. In most cases, the compiler creates a class pair, consisting of a Java interface and a Java implementation class.

The class inherits from the classes and interfaces listed in *supers*. This is a list of names of classes that are in scope (perhaps imported using `require`), or names for existing classes or interfaces surrounded by `<>`, such as `<gnu.lists.Sequence>`. If `define-simple-class` is used, at most one of these may be the name of a normal Java class or classes defined using `define-simple-class`; the rest must be interfaces or classes defined using `define-class`. If `define-class` is used, *all* of the classes listed in *supers* should be interfaces or classes defined using `define-class`.

Each *field-decl* declares a public instance "slot" (field) with the given *fname*. If *ftype* is specified it is the type of the slot. The following *option-keyword*s are implemented:

`type:` *ftype*
: Specifies that *ftype* is the type of (the values of) the field. Equivalent to '`::` *ftype*'.

`allocation: class:`
: Specifies that there should be a single slot shared between all instances of the class (and its sub-classes). Not yet implemented for `define-class`, only for `define-simple-class`. In Java terms this is a `static` field.

`allocation: instance:`
: Specifies that each instance has a separate value "slot", and they are not shared. In Java terms, this is a non-`static` field. This is the default.

`init-form:` *expr*
: An expression used to initialize the slot. The lexical environment of the *expr* is that of the `define-class` or `define-simple-class`. (This is not quite true in the current implementation, as the names of fields and methods of this class are visible.)

`init-value:` *value*
: A value expression used to initialize the slot. For now this is synonymous with *init-form:*, but that may change (depending on what other implementation do), so to be safe only use `init-value:` with a literal.

`init-keyword:` *name*`:`
: A keyword that that can be used to initialize instance in `make` calls. For now, this is ignored, and *name* should be the same as the field's *fname*. `static` field.

Each *method-decl* declares a public non-static method, whose name is *method-name*. (If *method-name* is not a valid Java method name, it is mapped to something reasonable. For example `foo-bar?` is mapped to `isFooBar`.) The types of the method arguments can be specified in the *formal-arguments*. The return type can be specified by *rtype*, or is otherwise the type of the *body*. Currently, the *formal-arguments* cannot contain optional, rest, or keyword parameters. (The plan is to allow optional parameters, implemented using multiple overloaded methods.)

The scope of the *body* of a method includes the *field-decl*s of the object. It does include the surrounding lexical scope. It sort-of also includes the declared methods, but this is not working yet.

A simple example:

```
(define-simple-class <2d-vector> ()
  (x type: <double> init-value: 0.0 init-keyword: x:)
  (y type: <double> init-value: 0.0 init-keyword: y:)
  ((add (other :: <2d-vector>)) :: <2d-vector>
   ;; Kawa compiles this using primitive Java types!
   (make <2d-vector>
     x: (+ x (slot-ref other 'x))
     y: (+ y (slot-ref other 'y))))
  ((scale (factor :: <double>)) :: <2d-vector>
   ;; Unfortunately, multiply is not yet optimized as addition is.
   (make <2d-vector> x: (* factor x) y: (* factor y))))

(define-simple-class <3d-vector> (<2d-vector>)
  (z type: <double> init-value: 0.0 init-keyword: z:)
```

```
((scale (factor :: <double>)) :: <2d-vector>
 ;; Note we cannot override the return type to <3d-vector>
 ;; because Java does not allow that.  Should hide that. .
 (make <3d-vector>
   ;; Unfortunately, slot names of inherited classes are not visible.
   ;; Until this is fixed, use slot-ref.
   x: (* factor (slot-ref (this) 'x))
   y: (* factor (slot-ref (this) 'y))
   z: (* factor z))))
```

10.7 Anonymous classes

object (*supers ...*) *field-or-method-decl ...* Syntax

Returns a new instance of an anonymous (inner) class. The syntax is similar to `define-class`.

field-or-method ::= *field-decl* | *method-decl*
field-decl ::= (*fname* [[[::] *ftype*] *finit*])
| (*fname* [:: *ftype*] [*option-keyword option-value*]*)
method-decl ::= ((*method-name formal-arguments*) [[::] *rtype*] *body*)

Returns a new instance of a unique (anonymous) class. The class inherits from the list of *supers*, where at most one of the elements should be the base class being extended from, and the rest are interfaces.

This is roughly equivalent to:

```
(begin
  (define-simple-class hname (supers ...) field-or-method-decl ...)
  (make hname))
```

A *field-decl* is as for `define-class`, except that we also allow an abbreviated syntax. Each *field-decl* declares a public instance field. If *ftype* is given, it specifies the type of the field. If *finit* is given, it is an expression whose value becomes the initial value of the field. The *finit* is evaluated at the same time as the `object` expression is evaluated, in a scope where all the *fname*s are visible.

A *method-decl* is as for `define-class`.

10.8 Modules and how they are compiled to classes

A *module* is a set of definitions that the module *exports*, as well as some *actions* (expressions evaluated for their side effect). The top-level forms in a Scheme source file compile a module; the source file is the *module source*. When Kawa compiles the module source, the result is the *module class*. Each exported definition is translated to a public field in the module class.

There are two kinds of module class: A *static module* is a class (or gets compiled to a class) all of whose public fields a static, and that does not have a public constructor. A JVM can only have a single global instance of a static module. An *instance module* has a public default constructor, and usually has at least one non-static public field. There can be multiple instances of an instance module; each instance is called a *module instance*.

However, only a single instance of a module can be *registered* in an environment, so in most cases there is only a single instance of instance modules. Registering an instance in an environment means creating a binding mapping a magic name (derived from the class name) to the instance.

In fact, any Java class class that has the properties of either an instance module or a static module, is a module, and can be loaded or imported as such; the class need not have written using Scheme.

10.8.1 Name visibility

The definitions that a module exports are accessible to other modules. These are the "public" definitions, to use Java terminology. By default, all the identifiers declared at the top-level of a module are exported, except those defined using `define-private`. However, a major purpose of using modules is to control the set of names exported. One reason is to reduce the chance of accidental name conflicts between separately developed modules. An even more important reason is to enforce an interface: Client modules should only use the names that are part of a documented interface, and should not use internal implementation procedures (since those may change).

If there is a `module-export` declaration in the module, then only those names listed in a `module-export` are exported. There can be more than one `module-export`, and they can be anywhere in the Scheme file. As a matter of good style, I recommend a single `module-export` near the beginning of the file.

module-export *name ...* Syntax

Make the definition for each *name* be exported. Note that it is an error if there is no definition for *name* in the current module, or if it is defined using `define-private`.

In this module, `fact` is public and `worker` is private:

```
(module-export fact)
(define (worker x) ...)
(define (fact x) ...)
```

Alternatively, you can write:

```
(define-private (worker x) ...)
(define (fact x) ...)
```

10.8.2 Definitions

In addition to `define` (which can take an optional type specifier), Kawa has some extra definition forms.

define-private *name* [*:: type*] *value* Syntax
define-private (*name formals*) *body* Syntax

Same as `define`, except that `name` is not exported.

define-constant *name* [*:: type*] *value* Syntax

Definites *name* to have the given *value*. The value is readonly, and you cannot assign to it. (This is not fully enforced.) If the definition is at module level, then

the compiler will create a `final` field with the given name and type. The *value* is evaluated as normal; however, if it is a compile-time constant, it defaults to being static.

define-variable *name* [*init*] Syntax

If *init* is specified and *name* does not have a global variable binding, then *init* is evaluated, and *name* bound to the result. Otherwise, the value bound to *name* does not change. (Note that *init* is not evaluated if *name* does have a global variable binding.)

Also, declares to the compiler that *name* will be looked up in the dynamic environment. This can be useful for shutting up warnings from `--warn-undefined-variable`.

This is similar to the Common Lisp `defvar` form. However, the Kawa version is (currently) only allowed at module level.

define-namespace *name namespace-uri* Syntax

Defines *name* as *namespace alias* - a lexically scoped "nickname" for the namespace (or package) whose full name is *namespace-uri*, which should be a string literal. Any symbols in the scope of this definitions that contain a colon, and where the part before the colon matches the *name* will be treated as being in the package/namespace whose global unique name is the *namespace-uri*.

The features is currently used for XML (to be documented). In XML terminology, a name containing a colon is a *qualified name*. The part of the name before the colon is a *namespace prefix*, which is an aliases for a locally-visible *namespace uri*. The latter is an arbitrary string, but for uniqueness it is recommended that it be a uri belonging to the organization that defines the namespace. (It need to correspond to an actual browsable location, even though it looks like one.) The part of a name following the colon is the *local part* of the name.

If the namespace starts with the strings `"class:"`, then the *name* can be used for invoking Java methods - see Section 10.4 [Method operations], page 55.

10.8.3 How a module becomes a class

If you want to just use a Scheme module as a module (i.e. `load` or `require` it), you don't care how it gets translated into a module class. However, Kawa gives you some control over how this is done, and you can use a Scheme module to define a class which you can use with other Java classes. This style of class definition is an alternative to `define-class` [not yet implemented] which lets you define classes and instances fairly conveniently.

The default name of the module class is the main part of the filename of the Scheme source file (with directories and extensions sripped off). That can be overridden by the `-T` Kawa command-line flag. The package-prefix specified by the `-P` flag is prepended to give the fully-qualified class name.

module-name <*name*> Syntax

Sets the name of the generated class, overriding the default. If there is no '`.`' in the *name*, the package-prefix (specified by the `-P` Kawa command-line flag) is prepended.

By default, the base class of the generated module class is unspecified; you cannot count on it being more specific than `Object`. However, you can override it with `module-extends`.

module-extends <*class*> Syntax
Specifies that the class generated from the immediately surrounding module should extend (be a sub-class of) the class <*class*>.

module-implements <*interface*> ... Syntax
Specifies that the class generated from the immediately surrounding module should implement the interfaces listed.

Note that the compiler does *not* currently check that all the abstract methods requires by the base class or implemented interfaces are actually provided, and have the correct signatures. This will hopefully be fixed, but for now, if you are forgot a method, you will probably get a verifier error

For each top-level exported definition the compiler creates a corresponding public field with a similar (mangled) name. By default, there is some indirection: The value of the Scheme variable is not that of the field itself. Instead, the field is a `gnu.mapping.Symbol` object, and the value Scheme variable is defined to be the value stored in the `Symbol`. Howewer, if you specify an explicit type, then the field will have the specified type, instead of being a `Symbol`. The indirection using `Symbol` is also avoided if you use `define-constant`.

If the Scheme definition defines a procedure (which is not re-assigned in the module), then the compiler assumes the variable as bound as a constant procedure. The compiler generates one or more methods corresponding to the body of the Scheme procedure. It also generates a public field with the same name; the value of the field is an instance of a subclass of `<gnu.mapping.Procedure>` which when applied will execute the correct method (depending on the actual arguments). The field is used when the procedure used as a value (such as being passed as an argument to `map`), but when the compiler is able to do so, it will generate code to call the correct method directly.

You can control the signature of the generated method by declaring the parameter types and the return type of the method. See the applet (see Section 6.5 [Applet compilation], page 15) example for how this can be done. If the procedures has optional parameters, then the compiler will generate multiple methods, one for each argument list length. (In rare cases the default expression may be such that this is not possible, in which case an `"variable argument list"` method is generated instead. This only happens when there is a nested scope *inside* the default expression, which is very contrived.) If there are `#!keyword` or `#!rest` arguments, the compiler generate a `"variable argument list"` method. This is a method whose last parameter is either an array or a `<list>`, and whose name has `$V` appended to indicate the last parameter is a list.

Top-leval macros (defined using either `define-syntax` or `defmacro`) create a field whose type is currently a sub-class of `kawa.lang.Syntax`; this allows importing modules to detect that the field is a macro and apply the macro at compile time.

module-static *name ...* Syntax
module-static `#t` Syntax
module-static `#f` Syntax

Control whether the generated fields and methods are static. If `#t` is specified, then the module will be a static module, *all* definitions will be static, and the module body is evaluated in the class's static initializer. Otherwise, the module is an instance module. However, the *name*s that are explicitly listed will be compiled to static fields and methods. If `#f` is specified, then all exported names will be compiled to non-static (instance) fields and methods.

By default, if no `module-static` is specified, the following rules apply:

1. If there is a `module-extends` or `module-implements` declaration, then `(module-static #f)` is implied.
2. If the `--module-static` command-line parameter is specified, then `(module-static #t)` is implied.
3. (Not yet implemented: If there are no top-level actions and all definitions are procedure definitions, macro definitions, or constant definitions, then `(module-static #t)` is implied.)
4. Otherwise, a method will be static iff it doesn't need to reference non-static fields or methods of the module instance. In that case, the corresponding field will also be static.

Note `(module-static #t)` usually produces more efficient code, and is recommended if a module contains only procedure or macro definitions. (This may become the default.) However, a static module means that all environments in a JVM share the same bindings, which you may not want if you use multiple top-level environments.

Unfortuntely, the Java class verifier does not allow fields to have arbitrary names. Therefore, the name of a field that represents a Scheme variable is `"mangled"` (see Section 10.2 [Mangling], page 54) into an acceptable Java name. The implementation can recover the original name of a field `X` as `((gnu.mapping.Named) X).getName()` because all the standard compiler-generate field types implemented the `Named` interface.

The top-level actions of a module will get compiled to a `run` method. If there is an explicit `method-extends`, then the module class will also automatically implement `java.lang.Runnable`. (Otherwise, the class does not implement `Runnable`, since in that case the `run` method return an `Object` rather than `void`. This will likely change.)

10.8.4 Requiring (importing) a module

You can import a module into the current namespace with `require`.

require *modulespec* Syntax

The *modulespec* can be either a `<`*classname*`>` or a `'`*featurename*. In either case the names exported by the specified module (class) are added to the current set of visible names.

If *modulespec* is `<`*classname*`>` where *classname* is an instance module (it has a public default constructor), and if no module instance for that class has

been registered in the current environment, then a new instance is created and registered (using a "magic" identifier). If the module class either inherits from `gnu.expr.ModuleBody` or implements `java.lang.Runnable` then the corresponding `run` method is executed. (This is done *after* the instance is registered so that cycles can be handled.) These actions (creating, registering, and running the module instance) are done both at compile time and at run time, if necessary.

All the public fields of the module class are then incorporated in the current set of local visible names in the current module. This is done at compile time - no new bindings are created at run-time (except for the magic binding used to register the module instance), and the imported bindings are private to the current module. References to the imported bindings will be compiled as field references, using the module instance (except for static fields).

If the *modulespec* is '*featurename* then the *featurename* is looked up (at compile time) in the "feature table" which yields the implementing <*classname*>.

For some examples, you may want to look in the `gnu/kawa/slib` directory.

11 The Scheme-Java interface

Kawa has extensive features so you can work with Java objects and call Java methods.

11.1 Scheme types in Java

All Scheme values are implemented by sub-classes of '`java.lang.Object`'.

Scheme symbols are implemented using `java.lang.String`. (Don't be confused by the fact the Scheme sybols are represented using Java Strings, while Scheme strings are represented by `gnu.lists.FString`. It is just that the semantics of Java strings match Scheme symbols, but do not match mutable Scheme strings.) Interned symbols are presented as interned Strings. (Note that with JDK 1.1 string literals are automatically interned.)

Scheme integers are implemented by `gnu.math.IntNum`. Use the make static function to create a new IntNum from an int or a long. Use the intValue or longValue methods to get the int or long value of an IntNum.

A Scheme "flonum" is implemented by `gnu.math.DFloNum`.

A Scheme pair is implemented by `gnu.lists.Pair`.

A Scheme vector is implemented by `gnu.lists.FVectror`.

Scheme characters are implemented using `gnu.text.Char`.

Scheme strings are implemented using `gnu.lists.FString`.

Scheme procedures are all sub-classes of `gnu.mapping.Procedure`. The "action" of a '`Procedure`' is invoked by using one of the '`apply*`' methods: '`apply0`', '`apply1`', '`apply2`', '`apply3`', '`apply4`', or '`applyN`'. Various sub-class of '`Procedure`' provide defaults for the various '`apply*`' methods. For example, a '`Procedure2`' is used by 2-argument procedures. The '`Procedure2`' class provides implementations of all the '`apply*`' methods *except* '`apply2`', which must be provided by any class that extends `Procedure2`.

11.2 Low-level Operations on Java Arrays

The following macros evaluate to procedures that can be used to manipulate primitive Java array objects. The compiler can inline each to a single bytecode instruction (not counting type conversion).

primitive-array-new *element-type* Syntax

Evaluates to a one-argument procedure. Applying the resulting procedure to an integer count allocates a new Java array of the specified length, and whose elements have type *element-type*.

primitive-array-set *element-type* Syntax

Evaluates to a three-argument procedure. The first argument of the resulting procedure must be an array whose elements have type *element-type*; the second argument is an index; and the third argument is a value (coercible to *element-type*) which replaces the value specified by the index in the given array.

primitive-array-get *element-type* Syntax

Evaluates to a two-argument procedure. The first argument of the resulting procedure must be an array whose elements have type *element-type*; the second argument is an index. Applying the procedure returns the element at the specified index.

primitive-array-length *element-type* Syntax

Evaluates to a one-argument procedure. The argument of the resulting procedure must be an array whose elements have type *element-type*. Applying the procedure returns the length of the array. (Alternatively, you can use `(field` *array* `'length)`.)

11.3 Low-level Method invocation

The following lower-level primitives require you to specify the parameter and return types explicitly. You should probably use the functions `invoke` and `invoke-static` (see Section 10.4 [Method operations], page 55) instead.

primitive-constructor *class* (*argtype ...*) Syntax

Returns a new anonymous procedure, which when called will create a new object of the specified class, and will then call the constructor matching the specified argument types.

primitive-virtual-method *class method rtype* (*argtype ...*) Syntax

Returns a new anonymous procedure, which when called will invoke the instance method whose name is the string *method* in the class whose name is *class*.

primitive-static-method *class method rtype* (*argtype ...*) Syntax

Returns a new anonymous procedure, which when called will invoke the static method whose name is the string *method* in the class whose name is *class*.

primitive-interface-method *interface method rtype* (*argtype ...*) Syntax
Returns a new anonymous procedure, which when called will invoke the matching method from the interface whose name is *interface*.

The macros return procedure values, just like `lambda`. If the macros are used directly as the procedure of a procedure call, then kawa can inline the correct bytecodes to call the specified methods. (Note also that neither macro checks that there really is a method that matches the specification.) Otherwise, the Java reflection facility is used.

11.4 Low-level Operations on Object Fields

The following macros evaluate to procedures that can be used to access or change the fields of objects or static fields. The compiler can inline each to a single bytecode instruction (not counting type conversion).

These macros are deprecated. The `fields` and `static-field` functions (see Section 10.5 [Field operations], page 58) are easier to use, more powerful, and just as efficient. (One exception is for `primitive-set-static`; while its functionality can be expressed using `(set! (static-field ...) ...)`, that idiom is currently less efficient.) Also, the high-level functions currently do not provide access to private fields.

primitive-get-field *class fname ftype* Syntax
Use this to access a field named *fname* having type *type* in class *class*. Evaluates to a new one-argument procedure, whose argument is a reference to an object of the specified *class*. Calling that procedure returns the value of the specified field.

primitive-set-field *class fname ftype* Syntax
Use this to change a field named *fname* having type *type* in class *class*. Evaluates to a new two-argument procedure, whose first argument is a reference to an object of the specified *class*, and the second argument is the new value. Calling that procedure sets the field to the specified value. (This macro's name does not end in a '`!`', because it does not actually set the field. Rather, it returns a function for setting the field.)

primitive-get-static *class fname ftype* Syntax
Like `primitive-get-field`, but used to access static fields. Returns a zero-argument function, which when called returns the value of the static field.

primitive-set-static *class fname ftype* Syntax
Like `primitive-set-field`, but used to modify static fields. Returns a one-argument function, which when called sets the value of the static field to the argument.

11.5 Loading Java functions into Scheme

When `kawa -C` compiles (see Section 6.2 [Files compilation], page 13) a Scheme module it creates a class that implements the `java.lang.Runnable` interface. (Usually it is a

class that extends the `gnu.expr.ModuleBody`.) It is actually fairly easy to write similar "modules" by hand in Java, which is useful when you want to extend Kawa with new "primitive functions" written in Java. For each function you need to create an object that extends `gnu.mapping.Procedure`, and then bind it in the global environment. We will look at these two operations.

There are multiple ways you can create a `Procedure` object. Below is a simple example, using the `Procedure1` class, which is class extending `Procedure` that can be useful for one-argument procedure. You can use other classes to write procedures. For example a `ProcedureN` takes a variable number of arguments, and you must define `applyN(Object[] args)` method instead of `apply1`. (You may notice that some builtin classes extend `CpsProcedure`. Doing so allows has certain advantages, including support for full tail-recursion, but it has some costs, and is a bit trickier.)

```
import gnu.mapping.*;
import gnu.math.*;
public class MyFunc extends Procedure1
{
  // An "argument" that is part of each procedure instance.
  private Object arg0;

  public MyFunc(String name, Object arg0)
  {
    super(name);
    this.arg0 = arg0;
  }

  public Object apply1 (Object arg1)
  {
    // Here you can so whatever you want. In this example,
    // we return a pair of the argument and arg0.
    return gnu.lists.Pair.make(arg0, arg1);
  }
}
```

You can create a `MyFunc` instance and call it from Java:

```
Procedure myfunc1 = new MyFunc("my-func-1", Boolean.FALSE);
Object aresult = myfunc1.apply1(some_object);
```

The name `my-func-1` is used when `myfunc1` is printed or when `myfunc1.toString()` is called. However, the Scheme variable `my-func-1` is still not bound. To define the function to Scheme, we can create a "module", which is a class intended to be loaded into the top-level environment. The provides the definitions to be loaded, as well as any actions to be performed on loading

```
public class MyModule
{
  // Define a function instance.
  public static final MyFunc myfunc1
    = new MyFunc("my-func-1", IntNum.make(1));
}
```

If you use Scheme you can use `require`:

```
#|kawa:1|# (require <MyModule>)
#|kawa:2|# (my-func-1 0)
(1 0)
```

Note that `require` magically defines `my-func-1` without you telling it to. For each public final field, the name and value of the field are entered in the top-level environment when the class is loaded. (If there are non-static fields, or the class implements `Runnable`, then an instance of the object is created, if one isn't available.) If the field value is a `Procedure` (or implements `Named`), then the name bound to the procedure is used instead of the field name. That is why the variable that gets bound in the Scheme environment is `my-func-1`, not `myfunc1`.

Instead of `(require <MyModule>)`, you can do `(load "MyModule")` or `(load "MyModule.class")`. If you're not using Scheme, you can use Kawa's `-f` option:

```
$ kawa -f MyModule --xquery --
#|kawa:1|# my-func-1(3+4)
<list>1 7</list>
```

If you need to do some more complex calculations when a module is loaded, you can put them in a `run` method, and have the module implement `Runnable`:

```
public class MyModule implements Runnable
{
  public void run ()
  {
    Interpreter interp = Interpreter.getInterpreter();
    Object arg = Boolean.TRUE;
    interp.defineFunction (new MyFunc ("my-func-t", arg));
    System.err.println("MyModule loaded");
  }
}
```

Loading `MyModule` causes `"MyModule loaded"` to be printed, and `my-func-t` to be defined. Using `Interpreter`'s `defineFunction` method is recommended because it does the righ things even for languages like Common Lisp that use separate "namespaces" for variables and functions.

A final trick is that you can have a `Procedure` be its own module:

```
import gnu.mapping.*;
import gnu.math.*;
public class MyFunc2 extends Procedure2
{
  public MyFunc(String name)
  {
    super(name);
  }

  public Object apply2 (Object arg1, arg2)
  {
    return gnu.lists.Pair.make(arg1, arg2);
  }
```

```
  public static final MyFunc myfunc1 = new MyFunc("my-func-2);
}
```

11.6 Evaluating Scheme expressions from Java

The following methods are recommended if you need to evaluate a Scheme expression from a Java method. (Some details (such as the '`throws`' lists) may change.)

`void` **Scheme.registerEnvironment** () Static method
Initializes the Scheme environment. Maybe needed if you try to load a module compiled from a Scheme source file.

`Object` **Scheme.eval** (`InPort` *port*, `Environment` *env*) Static method
Read expressions from *port*, and evaluate them in the *env* environment, until end-of-file is reached. Return the value of the last expression, or `Interpreter.voidObject` if there is no expression.

`Object` **Scheme.eval** (`String` *string*, `Environment` *env*) Static method
Read expressions from *string*, and evaluate them in the *env* environment, until the end of the string is reached. Return the value of the last expression, or `Interpreter.voidObject` if there is no expression.

`Object` **Scheme.eval** (`Object` *sexpr*, `Environment` *env*) Static method
The *sexpr* is an S-expression (as may be returned by `read`). Evaluate it in the *env* environment, and return the result.

For the `Environment` in most cases you could use '`Environment.current()`'. Before you start, you need to initialize the global environment, which you can with

```
Environment.setCurrent(new Scheme().getEnvironment());
```

Alternatively, rather than setting the global environment, you can use this style:

```
Scheme scm = new Scheme();
Object x = scm.eval("(+ 3 2)");
System.out.println(x);
```

12 Tools for working with XML and HTML

Kawa has a number of experimental features for working with XML. These are built on the concepts of the `gnu.lists` package.

12.0.1 Scheme functions

make-element *tag* [*attribute ...*] *child ...* Function
Create a representation of a XML element, corresponding to

<*tag* *attribute*...>*child*...</*tag*>

The result is a `TreeList`, though if the result context is a consumer the result is instead "written" to the consumer. Thus nested calls to `make-element` only result in a single `TreeList`. More generally, whether an *attribute* or *child* is includded by copying or by reference is (for now) undefined. The *tag* should currently be a symbol, though in the future it should be a qualified name. An *attribute* is typically a call to `make-attribute`, but it can be any attribute-valued expression.

```
(make-element 'p
      "The time is now: "
      (make-element 'code (make <java.util.Date>)))
```

make-attribute *name value...* Function

Create an "attribute", which is a name-value pair. For now, *name* should be a symbol

as-xml *value* Function

Return a value (or multiple values) that when printed will print *value* in XML syntax.

```
(require 'xml)
(as-xml (make-element 'p "Some " (make-element 'em "text") "."))
```

prints `<p>Some <em>text</em>.</p>`.

12.0.2 xquery language

W3C is working an an XML Query language (`http://www.w3c.org/XML/Query`), and a draft has been released. If you start Kawa with the `--xquery` it selects the "XQuery" source language; this also prints output using XML syntax. See the Qexo (Kawa-XQuery) home page (`http://www.gnu.org/software/qexo/`) for examples and more information.

12.0.3 XSL transformations

There is an experimental implementation of the XSLT (XML Stylesheet Language Transformations) language. Selecting `--xslt` at the Kawa command line will parse a source file according to the syntax on an XSLT stylesheet. See the Kawa-XSLT page (`http://www.gnu.org/software/qexo/xslt.html`) for more information.

12.1 Writing web-server-side Kawa scripts

You can compile a Kawa program (written in any supported Kawa language, including Scheme, BRL, KRL, or XQuery), and run it as either servlet engine (using a web server that supports servlets), or as a "CGI script" on most web servers.

In either case, the result of evaluating the top-level expressions becomes the HTTP response that the servlet sends back to the browser. The result is typically an HTML/XML element code object; Kawa will automatically format the result as appropriate for the type. The initial result values may be special "response header values", as created by the

`response-header` function. Kawa will use the response header values to set various required and optional fields of the HTTP response. Note that `response-header` does not actually do anything until it is "printed" to the standard output. Note also that if a `"Content-Type"` response value is printed that controls the formatting of the following non-response-header values.

Here is a simple program `hello.scm`:

```
(require 'http) ; Required for Scheme, though not BRL/KRL.
(response-content-type 'text/html) ; Optional
(make-element 'p
  "The request URL was: " (request-url))
(make-element 'p
  (let ((query (request-query-string)))
    (if query
      (values-append "The query string was: " query)
      "There was no query string.")))
#\newline ; emit a new-line at the end
```

The same program using KRL is shorter:

```
<p>The request URL was: [(request-url)]</p>,
<p>[(let ((query (request-query-string)))
    (if query
      (begin ]The query string was: [query)
      ]There was no query string.[))]</p>
```

You can also use XQuery:

```
<p>The request URL was: {request-url()}</p>
<p>{let $query := request-query-string() return
    if ($query)
    then ("The query string was: ",$query)
    else "There was no query string."}</p>
```

Either way, you compile your program to a servlet:

```
kawa --servlet -C hello.scm
```

or:

```
kawa --servlet --krl -C hello.krl
```

or:

```
kawa --servlet --xquery -C hello.xql
```

The next two sections will explain how you can install this script as either a servlet or a CGI script.

12.2 Installing Kawa programs as Servlets

You can compile a Kawa program to a `Servlet`, and run it in a servlet engine (a Servlet-aware web server). I assume you have compiled your program to a servlet as descrbibed in the previous section.

If you have Tomcat 4.x installed, and you want `hello` to be part of the `myutils` "web application", copy `hello*.class` into `$CATALINA_HOME/webapps/myutils/WEB-INF/classes/`. You also need to copy Kawa somewhere where Tomcat can find it, for

example `$CATALINA_HOME/lib/kawa-1.7.90.jar`. You can then run the `hello` servlet using the URL `http://localhost:8080/myutils/servlet/hello`.

current-servlet Function

When called from a Kawa servlet's handler, returns the actual `javax.servlet.http.HttpServlet` instance.

current-servlet-context Function

Returns the `ServletContext` of the currently executing servlet.

current-servlet-config Function

Returns the `ServletConfig` of the currently executing servlet.

servlet-context-realpath Function

Returns the file path of the current servlet's "Web application".

12.3 Installing Kawa programs as CGI scripts

The recommended way to have a web-server run a Kawa program as a CGI script is to compile the Kawa program to a servlet (as explained in Section 12.1 [Server-side scripts], page 72, and then use Kawa's supplied CGI-to-servlet bridge.

First, compile your program to one or more class files as explain in Section 12.1 [Server-side scripts], page 72. For example:

```
kawa --servlet --xquery -C hello.xql
```

Then copy the resulting `.class` files to your server's CGI directory. On Red Hat GNU/Linux, you can do the following (as root):

```
cp hello*.class /var/www/cgi-bin/
```

Next find the `cgi-servlet` program that Kawa builds and installs. If you installed Kawa in the default place, it will be in `/usr/local/bin/cgi-servlet`. (You'll have this if you installed Kawa from source, but not if you're just using Kawa `.jar` file.) Copy this program into the same CGI directory:

```
cp /usr/local/bin/cgi-servlet /var/www/cgi-bin/
```

You can link instead of copying:

```
ln -s /usr/local/bin/cgi-servlet /var/www/cgi-bin/
```

However, because of security issues this may not work, so it is safer to copy the file. However, if you already have a copy of `cgi-servlet` in the CGI-directory, it is safe to make a hard link instead of making an extra copy.

Make sure the files have the correct permissions:

```
chmod a+r /var/www/cgi-bin/hello*.class /var/www/cgi-bin/hello
chmod a+x /var/www/cgi-bin/hello
```

Now you should be able to run the Kawa program, using the URL `http://localhost/cgi-bin/hello`. It may take a few seconds to get the reply, mainly because of the start-up time of the Java VM. That is why servlets are preferred. Using the CGI interface can still be useful for testing or when you can't run servlets. We hope to soon be able to run Kawa CGI scripts compiled using GCJ, which should have much reduced start-up time, making Kawa servlets more practical.

12.4 Functions for accessing HTTP requests

The following functions are useful for accessing properties of a HTTP request, in a Kawa program that is run either as a servlet or a CGI script. These functions can be used from plain Scheme, from KRL (whether in BRL-compatible mode or not), and from XQuery.

If using plain Scheme, you need to do the following before using these functions.

```
(require 'http)
```

This is not needed for KRL or XQuery.

request-method Function

Returns the method of the HTTP request, usually `"GET"` or `"POST"`. Corresponds to the CGI variable `REQUEST_METHOD`.

request-path-info Function

Corresponds to the CGI variable `PATH_INFO`.

request-path-translated Function

Corresponds to the CGI variable `PATH_TRANSLATED`.

request-uri Function

Returns the URI of the request, not including the query string, or server specification. The is the combination of CGI variables `SCRIPT_NAME` and `PATH_INFO`.

request-url Function

Returns the complete URL of the request, except the query string.

request-query-string Function

Returns the query string from an HTTP request. The query string is the part of the request URL after a quetion mark. Returns false if there was no query string. Corresponds to the CGI variable `QUERY_STRING`.

12.5 Functions for generating HTTP responses

unescaped-data *data* Function

Creates a special value which causes `data` to be printed, as is, without normal escaping. For example, when the output format is XML, then printing `"<?xml?>"` prints as '`<?xml?>`', but `(unescaped-data "<?xml?>")` prints as '`<?xml?>`'.

If using plain Scheme, you need to do the following before using these functions.

```
(require 'http)
```

This is not needed for KRL or XQuery.

response-header *key value* Function

Create the response header '*key*`:` *value*' in the HTTP response. The result is a `"response header value"` (of some unspecified type). It does not directly set or print a response header, but only does so when you actually `"print"` its value to the response output stream.

response-content-type *type* Function
Species the content-type of the result - for example `"text/plain"`. Convenience function for `(response-header "Content-Type"` *type*`)`.

error-response *code* [*message*] Function
Creates a response-header with an error code of *code* and a response message of *message*. (For now this is the same as `response-status`.)

Note this also returns a response-header value, which does not actually do anything unless it is returned as the result of executing a servlet body.

response-status *code* [*message*] Function
Creates a response-header with an status code of *code* and a response message of *message*. (For now this is the same as `error-response`.)

13 KRL - The Kawa Report Language for generating XML/HTML

KRL (the "Kawa Report Language") is powerful Kawa dialect for embedding Scheme code in text files such as HTML or XML templates. You select the KRL language by specifying `--krl` on the Kawa command line.

KRL is based on on BRL (`http://brl.sourceforge.net/`), Bruce Lewis's "Beautiful Report Language", and uses some of BRL's code, but there are some experimental differences, and the implementation core is different. You can run KRL in BRL-compatility-mode by specifying `--brl` instead of `--krl`.

13.1 Differences between KRL and BRL

This section summarizes the known differences between KRL and BRL. Unless otherwise specified, KRL in BRL-compatibility mode will act as BRL.

- In BRL a normal Scheme string `"mystring"` is the same as the inverted quote string `]mystring[`, and both are instances of the type `<string>`. In KRL `"mystring"` is a normal Scheme string of type `<string>`, but `]mystring[` is special type that suppresses output escaping. (It is equivalent to `(unescaped-data "mystring")`.)
- When BRL writes out a string, it does not do any processing to escape special characters like `<`. However, KRL in its default mode does normally escape characters and strings. Thus `"<a>"` is written as `<a&gr;`. You can stop it from doing this by overriding the output format, for example by specifying `--output-format scheme` on the Kawa command line, or by using the `unescaped-data` function.
- Various Scheme syntax forms, including `lambda`, take a `<body>`, which is a list of one or more declarations and expressions. In normal Scheme and in BRL the value of a `<body>` is the value of the last expression. In KRL the value of a `<body>` is the concatenation of all the values of the expressions, as if using `values-append`.
- In BRL a word starting with a colon is a keyword. In KRL a word starting with a colon is an identifier, which by default is bound to the `make-element` function specialized to take the rest of the word as the tag name (first argument).

- BRL has an extensive utility library. Most of this has not yet been ported to KRL, even in BRL-compatibility mode.

14 Where to report bugs, discuss changes, etc

14.1 Reporting bugs

To report a bug or feature request for Kawa (including Qexo or JEmacs), it is best to use the bug-submission page (`http://savannah.gnu.org/bugs/?func=addbug&group=kawa`). You can browse and comment on existing bug reports using the Kawa Bugzilla page (`http://savannah.gnu.org/bugs/?group=kawa`).

When a bug report is created or modified, mail is automatically sent to the `bug-kawa@gnu.org` list. You can subscribe, unsubscribe, or browse the archives through the `bug-kawa` web interface (`http://mail.gnu.org/mailman/listinfo/bug-kawa`).

14.2 General Kawa email and discussion

The general Kawa email list is `kawa@sources.redhat.com`. This mailing list is used for announcements, questions, patches, and general discussion relating to Kawa. If you wish to subscribe, send a blank message request to `kawa-subscribe@sources.redhat.com`. To unsubscribe, send a blank message to `kawa-unsubscribe@sources.redhat.com`. (If your mail is forwarded and you're not sure which email address you're subscribed as send mail to the address following `mailto:` in the `List-Unsubscribe` line in the headers of the messages you get from the list.)

You can browse the archive of past messages (`http://sources.redhat.com/ml/kawa/`).

There are separate mailing lists for Qexo (`http://mail.gnu.org/mailman/listinfo/qexo-general`) and JEmacs (`http://lists.sourceforge.net/mailman/listinfo/jemacs-info`).

15 Technical Support for Kawa

If you have a project that depends on Kawa or one of its component packages, you might do well get get paid priority support from Kawa's author.

The base price is $2400 for one year. This entitles you to basic support by email or phone. Per `per@bothner.com` will answer techical questions about Kawa or its implementation, investigate bug reports, and suggest work-arounds. I may (at my discretion) provide fixes and enhancements (patches) for simple problems. Reponse for support requests received using the day (California time) will normally be within a few hours.

All support requests must come through a single designated contact person. If Kawa is important to your business, you probably want at least two contact people, doubling the price.

If the support contract is cancelled (by either party), remaining time will be prorated and refunded.

Per is also available for development projects.

16 Projects using Kawa

JEmacs is included in the Kawa distribution. It is a project to re-implement Emacs, allowing a mix of Java. Scheme, and Emacs Lisp. It has its own home-page (`http://www.jemacs.net/`).

BRL ("the Beautiful Report Language") is a database-oriented language to embed in HTML and other markup. BRL (`http://brl.sourceforge.net/`) allows you to embed Scheme in a an HTML file on a web server.

The Health Media Research Laboratory, part of the Comprehensive Cancer Center at the University of Michigan, is using Kawa as an integral part of its core tailoring technologies. Java programs using Kawa libraries are used to administer customized web-based surveys, generate tailored feedback, validate data, and "characterize," or transform, data. Kawa code is embedded directly in XML-formatted surveys and data dictionaries. Performance and ease of implementation has far exceeded expectations. For more information contact Paul R. Potts, Technical Director, Health Media Research Lab, `<potts@umich.edu>`.

Mike Dillon (`mdillon@gjt.org`) did the preliminary work of creating a Kawa plugin for jEdit. It is called SchemeShell and provides a REPL inside of the jEdit console for executing expressions in Kawa (much as the BeanShell plugin does with the BeanShell scripting language). It is currently available only via CVS from:

```
CVSROOT=:pserver:anonymous@cvs.jedit.sourceforge.net:/cvsroot/jedit
MODULE=plugins/SchemeShell
```

STMicroelectronics (`marco.vezzoli@st.com`) uses Kawa in a prototypal intranet 3tier information retrival system as a communication protocol between server and clients and to do server agents programming.

The Nice Programming Language is an open source research language that has a Java-like syntax. It features multiple dispatch methods, parametric types, higher-order functions, tuples, ..., and the new concept of "abstract interfaces". The Nice compiler (`nicec`) uses Kawa's `gnu.expr` and `gnu.bytecode` packages to generate Java bytecode. You can find more about Nice at `http://nice.sourceforge.net`. For more information feel free to contact Daniel Bonniot (`d.bonniot@mail.dotcom.fr`).

17 License

17.1 License for the Kawa software

The license for Kawa and the packages it depends on is a "modified Gnu Public License". You can find it in the file `COPYING` in the Kawa sources, and also quoted here:.

```
The Java classes (with related files and documentation) in these packages
are copyright (C) 1996, 1997, 1998, 1999  Per Bothner.

These classes are distributed in the hope that they will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
```

```
These classes are free software.  You can use and re-distribute a class
without restriction (in source or binary form) as long as you use a
version that has not been modified in any way from a version released
by Per Bothner, Red Hat inc, or the Free Software Foundation.
You may make and distribute a modified version, provided you follow
the terms of the GNU General Public License; either version 2,
or (at your option) any later version.
```

The file `COPYING` also contains a copy of the GNU General Public License version 2.

People have asked what the Kawa license means in practice. Informally, you get to pick between these choices:

1. Use Kawa as distributed by Per Bothner, Red Hat Inc, or the Free Software Foundation (or their approved agents), with no modifications. In that case, you can use Kawa for any purpose you like, and distribute your application with any license you like. (This basically gives you the same rights as a typical commercial royalty-free re-distribution license.)
2. Obey the terms of the standard Gnu Public License. (See `http://www.gnu.org/copyleft/gpl.html`). Informally, this means that if you distribute any application that is based on Kawa, you must also make available to all your recipients (customers) the source code for your entire application, giving them the modification and re-distribution rights they have under the GPL. In a Java context, I take "entire application" to mean all classes (and native code) that run in the same Java virtual machine, except for the Java runtime system itself (the virtual machine, low-level run-time system, and any classes in a `java` or `javax` package).
3. If you need to make a change to Kawa, you can submit them to Per Bothner, and convince him to include them in future Kawa releases.
4. You can negotiate some other (commercial) license with Per Bothner.

In general, if the license of Kawa or associated packages causes difficulties, let me know.

Kawa uses some math routines from fdlib's libf77, which bear the following copyright:

> Copyright 1990, 1991, 1992, 1993 by AT&T Bell Laboratories and Bellcore.
>
> Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the names of AT&T Bell Laboratories or Bellcore or any of their entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.
>
> AT&T and Bellcore disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall AT&T or Bellcore be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

17.2 Copyright for this manual

Here is the license for this manual:

Index

#

`#!eof` 20

`#!key` 20

#!null 20
#!optional 20
#!rest 20
#!void 20

<

<character> 50
<complex> 49
<f32vector> 31
<f64vector> 31
<input-port> 50
<integer> 49
<keyword> 49
<list> 50
<number> 49
<object> 49
<output-port> 50
<pair> 50
<procedure> 50
<quantity> 49
<rational> 49
<real> 49
<s16vector> 30
<s32vector> 30
<s64vector> 30
<s8vector> 30
<string> 50
<String> 50
<symbol> 49
<u16vector> 30
<u32vector> 30
<u64vector> 30
<u8vector> 30
<vector> 50

A

apply 21
arithmetic-shift 26
array 28
array-end 28
array-rank 28
array-ref 28
array-set! 29
array-start 28
array? 27
as 39
as-xml 72
ash 26

B

base-uri 36
bit-extract 26

C

call-with-input-string 42
call-with-output-string 42
call-with-values 18
catch 33
class-methods 58
command-line-arguments 38
command-parse 38
compile-file 13
cond-expand 17
constant-fold 21
copy-file 41
create-directory 41
current-error-port 41
current-servlet 74
current-servlet-config 74
current-servlet-context 74
cut 23
cute 24

D

default-prompter 42
define 51
define-alias 35
define-base-unit 25
define-class 59
define-constant 62
define-namespace 63
define-private 62
define-procedure 22
define-record-type 51
define-simple-class 59
define-syntax 17
define-unit 25
define-variable 63
defmacro 17
delete-file 40
dynamic-wind 34

E

environment-bound? 36
error 34
error-response 76
eval 36
exit 38

F

f32vector ... 31
f32vector->list ... 32
f32vector-length ... 32
f32vector-ref ... 32
f32vector-set! ... 32
f32vector? ... 31
f64vector ... 31
f64vector->list ... 32
f64vector-length ... 32
f64vector-ref ... 32
f64vector-set! ... 32
f64vector? ... 31
field ... 58
file-directory? ... 40
file-exists? ... 40
file-readable? ... 40
file-writable? ... 40
fluid-let ... 36
force ... 37
force-output ... 42
format ... 44
future ... 37

G

gentemp ... 17
get-output-string ... 42

H

home-directory ... 38

I

input-port-column-number ... 43
input-port-line-number ... 43
input-port-prompter ... 42
input-port-read-state ... 43
instance? ... 39
integer-length ... 26
interaction-environment ... 36
invoke ... 57
invoke-special ... 57
invoke-static ... 56

K

keyword->string ... 21
keyword? ... 20

L

let ... 51
let* ... 51
let*-values ... 19
let-values ... 18
letrec ... 51
list->f32vector ... 33
list->f64vector ... 33
list->s16vector ... 33
list->s32vector ... 33
list->s64vector ... 33
list->s8vector ... 33
list->u16vector ... 33
list->u32vector ... 33
list->u64vector ... 33
list->u8vector ... 33
load ... 36
load-relative ... 36
location ... 35
logand ... 25
logbit? ... 26
logcount ... 26
logior ... 26
lognot ... 26
logop ... 26
logtest ... 26
logxor ... 26

M

make ... 55
make-array ... 28
make-attribute ... 72
make-element ... 71
make-f32vector ... 31
make-f64vector ... 31
make-procedure ... 22
make-process ... 37
make-quantity ... 25
make-record-type ... 53
make-s16vector ... 31
make-s32vector ... 31
make-s64vector ... 31
make-s8vector ... 31
make-temporary-file ... 41
make-u16vector ... 31
make-u32vector ... 31
make-u64vector ... 31
make-u8vector ... 31
module-compile-options ... 15
module-export ... 62
module-extends ... 64
module-implements ... 64

module-name 63
module-static 64, 65
modulo 25

N

null-environment 36

O

object 61
open-input-string 41
open-output-string 41
options 8

P

parse-format 44
port-char-encoding 43
port-column 42
port-line 42
primitive-array-get 67
primitive-array-length 67
primitive-array-new 67
primitive-array-set 67
primitive-constructor 67
primitive-get-field 68
primitive-get-static 68
primitive-interface-method 68
primitive-set-field 68
primitive-set-static 68
primitive-static-method 67
primitive-throw 34
primitive-virtual-method 67
procedure-property 21

Q

quantity->number 25
quantity->unit 25
quantity? 25
quotient 25

R

read-line 41
receive 19
record-accessor 53
record-constructor 53
record-modifier 53
record-predicate 53
record-type-descriptor 53
record-type-field-names 54
record-type-name 54
record? 53
remainder 25
rename-file 41
request-method 75
request-path-info 75
request-path-translated 75
request-query-string 75
request-uri 75
request-url 75
require 65
response-content-type 76
response-header 75
response-status 76
reverse! 27

S

s16vector 31
s16vector->list 32
s16vector-length 32
s16vector-ref 32
s16vector-set! 32
s16vector? 31
s32vector 31
s32vector->list 32
s32vector-length 32
s32vector-ref 32
s32vector-set! 32
s32vector? 31
s64vector 31
s64vector->list 32
s64vector-length 32
s64vector-ref 32
s64vector-set! 32
s64vector? 31
s8vector 31
s8vector->list 32
s8vector-length 31
s8vector-ref 32
s8vector-set! 32
s8vector? 31
scheme-implementation-version 38
scheme-report-environment 36
scheme-window 38
Scheme.eval 71
Scheme.registerEnvironment 71
servlet-context-realpath 74
set-input-port-line-number! 43
set-input-port-prompter! 42
set-port-line! 43
set-procedure-property! 21
setter 35

shape ... 28
share-array ... 29
sleep ... 37
slot-ref ... 58
slot-set! ... 58
static-field ... 58
string->keyword ... 21
string-capitalize ... 27
string-capitalize! ... 27
string-downcase ... 27
string-downcase! ... 27
string-upcase ... 27
string-upcase! ... 27
symbol-read-case ... 43
synchronized ... 39
system ... 37
system-tmpdir ... 41

T

this ... 51
throw ... 33
tokenize-string-to-string-array ... 38
trace ... 37
try-catch ... 34
try-finally ... 34

U

u16vector ... 31
u16vector->list ... 32
u16vector-length ... 32
u16vector-ref ... 32
u16vector-set! ... 32
u16vector? ... 31
u32vector ... 31
u32vector->list ... 32
u32vector-length ... 32
u32vector-ref ... 32
u32vector-set! ... 32
u32vector? ... 31
u64vector ... 31
u64vector->list ... 32
u64vector-length ... 32
u64vector-ref ... 32
u64vector-set! ... 32
u64vector? ... 31
u8vector ... 31
u8vector->list ... 32
u8vector-length ... 32
u8vector-ref ... 32
u8vector-set! ... 32
u8vector? ... 31
unescaped-data ... 75
unless ... 38
untrace ... 37

V

values ... 18
values-append ... 19
vector-append ... 38

W

when ... 38
with-compile-options ... 15