The Kawa Scheme language

16 January 2020

Per Bothner

Table of Contents

Kawa is a general-purpose programming language that runs on the Java platform. It aims to combine:

- the benefits of dynamic scripting languages (non-verbose code with less boilerplate, fast and easy start-up, a REPL (`http://en.wikipedia.org/wiki/Read-eval-print_loop`), no required compilation step); with
- the benefits of traditional compiled languages (fast execution, static error detection, modularity, zero-overhead Java platform integration).

It is an extension of the long-established Scheme (`http://www.schemers.org/`) language, which is in the Lisp family of programming languages. Kawa has many Chapter 2 [Features], page 42.

Kawa is also a useful Chapter 23 [Framework], page 387, for implementing other programming languages on the Java platform. It has many useful utility classes.

This manual describes version 3.1.1, updated 16 January 2020. See the summary of Chapter 1 [News], page 1.

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

The Chapter 5 [Tutorial], page 67, is useful to get stated. While it is woefully incomplete, it does link to some other more in-depth (but not Kawa-specific) Scheme tutorials.

For copyright information on the software and documentation, see Chapter 24 [License], page 387.

Various people and orgnizations Section 3.3 [Acknowledgements], page 47.

This package has nothing to do with the defunct Kawa commercial Java IDE.

1 News - Recent Changes

These changes are in more-or-less reverse chronological order, with the most recent changes first.

See also the list of Qexo (XQuery)-specific changes (`../qexo/news.html`).

Kawa 3.1.1 (January 16, 2020)

- Various bug-fixes, mostly related to packaging and `--browse-manual`.

Kawa 3.1 (January 7, 2020)

- Updates for Java 9 and newer.
- Support justification `~<...~>` in `format` (thanks to Helmut Eller).
- Partial (and highly experimental) support for the Language Server Protocol (`https://langserver.org`) (used by editors and IDEs for on-the-fly syntax checking and more).
- Revert 3.0 change in allocating closure objects for inlined functions.
- Enhancements to arrays to match SRFI 163 (`http://srfi.schemers.org/srfi-163/srfi-163.html`) and SRFI 164 (`http://srfi.schemers.org/srfi-163/srfi-164.html`):
  - The type `gvector` is a “generalized vector”.

  - Add optional port parameter to `format-array`.
  - The `build-array` procedure takes an optional setter procedure.
  - New procedures `array-shape`, `->shape`.
- In array constructors, index keywords must be all or none: `[int[] length: 5 11 22]` or `[int[] length: 5 1: 22 0: 11]`.
- Various improvements in the Common Lisp implementation, by Helmut Eller.
- New `--max-errors` option.
- The classes created by `define-record-type` now extends `kawa.lang.Record`, which adds some conveniences, such as printing.
- Support for source location ranges with an end position.
- Various improvements when running under DomTerm include clickable error message locations.
- Many bug-fixes and minor improvements.

Kawa 3.0 (October 2, 2017)

- Binary release are now built for Java 8. The repository source code is now set up for Java 8. (Building for Java 6 or 7 is still supported. Java 5 might also work, but has not been tested recently.)
- Tested and updated for Java 9.
- Most places where you could declare a new identifier binding have been generalized to accept Section 8.3 [Variables and Patterns], page 140, including literals and boolean *guards*.

  Related changes:
  - The form `(! `*`pattern expression`*`)` creates [exclam-syntax], page 141, by matching the *expression* against the *pattern*. It is like `define-constant` but generalized to patterns.
  - The conditional match form `(? `*`pattern expression`*`)` is similar to `!` but can only be used Section 8.2 [Conditionals], page 135. If the match fails, the condition is false.
  - Section 8.7 [Repeat forms], page 149, is a powerful experimental feature, similar to list comprehensions.
  - The new form [def-match], page 138, form is a generalization of `case` using patterns.
  - The internal calling convention used for "apply" (ie. calling an unknown-at-compile-time procedure) has been completely changed.
  - New types for arguments list (possibly with keywords) [Explicit argument list objects], page 167, along with [Argument list library], page 168, for using them.
- Major changes to strings:
  - *Incompatible change:* String literals are now `gnu.lists.IString` rather than `java.lang.String`. The advantage of using `gnu.lists.IString` is that `string-ref` and `string-length` are (roughly) constant-time, rather than having to linearly scan the string.

  - *Incompatible change:* The procedures `string-append`, `string-map`, `substring`, `list->string`, `vector->string`, `string-downcase`, `string-upcase`, `string-foldcase`, `string-titlecase`, and the constructor `string` return an immutable string (an `IString`). (The function `string-copy` is similar to `substring`, but returns a mutable string.) This is a work-in-progress with the goal of implementing SRFI-140 (`http://srfi.schemers.org/srfi-140/srfi-140.html`): Other procedures will be changed to return immutable strings.

    If you `(import (scheme base))` standard procedures such as `string-append` will return mutable strings; if you `(import (kawa base))` the procedures will return immutable strings. The command-line options `--r5rs` or `--r6rs` or `--r7rs` override the default so these procedures return mutable strings.
  - *Incompatible change:* Treating a string as a sequence is now simpler but possibly slower: The *I*'th element is now the *I*'th Unicode code point. Indexing with function-call syntax `(`*`string i`*`)` is the same as `(string-ref `*`string i`*`)` and `(length `*`string`*`)` is the same as `(string-length `*`string`*`)`. This applies to all classes that implement `java.lang.CharSequence`. Indexing may be a linear-time operation (thus much slower), unless the string is an `IString` (in which case it is constant-time),
  - *Incompatible change:* Before, if a Java parameter type was `java.lang.String` Kawa would accept any value, converting it using Object's `toString` method. Now Kawa will reject an argument if it is not a `java.lang.CharSequence`.
  - New procedures: `istring?`, `reverse-list->string`, `string-any`, `string-concatenate`, `string-concatenate-reverse`, `string-contains`, `string-contains-right`, `string-count`, `string-drop`, `string-drop-right`, `string-every`, `string-filter`, `string-fold`, `string-fold-right`, `string-for-each-index`, `string-index`, `string-index-right`, `string-join`, `string-map-index`, `string-null?`, `string-prefix?`, `string-prefix-length`, `string-repeat`, `string-remove`, `string-replace`, `string-skip`, `string-skip-right`, `string-split`, `string-suffix?`, `string-suffix-length`, `string-tabulate`, `string-take`, `string-take-right`, `string-trim`, `string-trim-right`, `string-trim-both`, `string-unfold`, `string-unfold-right`, `string->utf16`, `string->utf16be`, `string->utf16le`, `utf16->string`, `utf16be->string`, `utf16le->string`, `xsubstring`. These follow SRFI-140 and return immutable strings. (Some of these had previously been available in SRFI-13, but the older versions return mutable strings.)
- *Incompatible change:* Kawa traditionally followed Java in allowing you to pass an array with the "rest" arguments to a varargs method. For example, you could write:

  ```
  (define args (Object[] 3 "cm"))
  (java.lang.String:format "length:%s%s" args)
  ```

  This is no longer allowed. Instead, use the splice operator:

  ```
  (java.lang.String:format "length:%s%s" @args)
  ```
- *Incompatible change:* You used to be able to write a type-specifier in a formal parameter or return type without using '`::`', as in:

  ```
    (define (incr (x int)) int (+ x 1))
  ```

This is no longer allowed, because it conflicts with the syntax for patterns. Instead you have to write:

```
(define (incr (x ::int)) ::int (+ x 1))
```

- New type aliases `bitvector` and `c16vector`. The latter is a Section 14.4 [Uniform vectors], page 239, type for wrapping `char[]` arrays.
- You can convert a Java array (for example a `int[]` to the corresponing uniform vector type (for example `u32vector`) using the `as` pseudo-function or the corresponding conversion procedure (for example `->u32vector`). The result shares storage with the array, so changes in one will update the other.
- The expression `(module-class)` evaluates to the containing module class.
- Change the Section 19.12 [Mangling], page 330, for field and local variables names to match the Symbolic Freedom (`https://blogs.oracle.com/jrose/entry/symbolic_freedom_in_the_vm`) style.
- Internally, expressions now record their ending position (line/column), in addition to the starting position.
- The new procedure `environment-fold` can be used to iterate over the bindings of an environment.
- Change in how closure objects are allocated for inlined functions. This sometimes reduces the number of objection allocations and also of helper classes, though in pathological cases it could cause objects to be retained (leak) where it didn't before.
- New command-line flag `--warn-uninitialized` (by default on) to control warning about using uninitialized variables.
- The pre-defined Section 13.2 [Character sets], page 204, are now based on Java 9's Unicode 8 support.

Kawa 2.4 (April 30, 2017)

- Final 2.x release. Minor updates and fixes.

Kawa 2.3 (January 13, 2017)

- Moved Kawa's source code repository (version control system) to use git, hosted at GitLab (`https://gitlab.com/kashell/Kawa`).
- Issues (bugs, feature requests, etc) should now be reported using the GitLab Issue Tracker (`https://gitlab.com/kashell/Kawa/issues`).
- New `with-docbook-stylesheets` to make it easier to build the documentation with better functionality and look.
- The command-line option `console:jline-mouse=yes` enables moving the input cursor using a mouse click, when using JLine in the REPL on common xterm-like terminals. This is disabled by default because it conflicts with other mouse actions, such as making a selection for copying text. You can press shift to get the terminal's standard mouse handling.

Kawa 2.2 (November 12, 2016)

- A binary release is no longer just a Kawa `.jar` file, but is now a `zip` archive that also includes shell/batch scripts for running Kawa, useful third-party libraries, and the complete documentation in EPUB format. The archives are named `kawa-version.zip`.

- The `kawa --browse-manual` switch makes it easy to [browse-manual-option], page 89.
- The Section 21.1 [Composable pictures], page 356, lets you create "picture" objects, display them, transform them, combine them, and more.
- There is a new Section 17.7 [Pretty-printing], page 292.
- Basic support for Java 9 (though still some issues).
- Generated files like `Makefile.in` and `configure` are no longer in the Subversion source code repository, though they are still included in the distributed `kawa-version.tar.gz` releases. The new top-level script `autogen.sh` should be run before `configure`.
- Kawa traditionally followed Java in allowing you to pass an array with the "rest" arguments to a varargs method. (A "varargs" method includes Java varargs methods, as well as Kawa methods with a `#!rest` parameter that is explicitly typed to be an array type.) For example, you could write:

  ```
  (define args (Object[] 3 "cm"))
  (java.lang.String:format "length:%s%s" args)
  ```

  This is deprecated, and may stop working in a future release. Instead, use the splice operator:

  ```
  (java.lang.String:format "length:%s%s" @args)
  ```
- More options for Section 14.6 [Ranges], page 246. For example, you can write `[1 by: 2 <=: 9]`.
- Many enhancements to Section 14.8 [Arrays], page 247, and vectors:
  - Shape specifiers (used when creating an array) can now be one of a rank-2 array of low/high-bounds, as in SRFI-25; a vector of upper bounds; or a vector of ranges.
  - New type specifiers for array: `array` is any array (i.e. any `gnu.lists.Array`); `array`*N* is the same restricted to rank *N*; `array[etype]` or `arrayN[etype]` restrict the types of elements to `etype`.

    If the `etype` is a primitive type (for example `array2[double]`) then indexing is optimized to method calls that avoid object allocation.
  - Generalized array indexing: If A is an array (or a vector), then the expression:
    `(A I J K ...)`
    in general evaluates to an array B such that:
    `(B i1 i2 ... j1 j2 ... k1 k2 ... ...)` is
    `(A (I i1 i2 ..) (J j1 j2 ...) (K k1 k2 ...) ...)`

    If an index I is an integer, it is treated as a zero-index array - a scalar.

    For example: if `(define B (A 2 [4 <: 10]))` then `(B i)` is `(A 2 (+ i 4))`.
  - The procedure `array-index-ref` is does the above indexing explicitly: `(array-index-ref A I J K ...)` is `(A I J K ...)`. The result is a read-only snapshot.
  - The procedure `array-index-share` is like `array-index-ref` but creates a modifiable view into argument array.
  - `(build-array shape procedure)` is a general constructor for lazy arrays: If `A` is the result, then `(A i j k ...)` is `(procedure [I J K ...])`.
  - `array-transform` creates a view, with a mapping of the indexes.

  - Other new procedures (like those in the Racket math package): `array-size`, `array-fill!`, `array-copy!`, `array-transform`, `array-reshape`, `array-flatten`, `array->vector`, `index-array`, `build-array`.
  - Add Common Lisp array reader syntax (`#rankA`) with Guile extensions (`https://www.gnu.org/software/guile/manual/html_node/Array-Syntax.html`), including reader sypport for multi-dimensional uniform (primitive) arrays. This is also used when printing arrays.
  - New `format-array` procedure print an array a tabular 2-dimensional (APL-like) format. This format is used by default in the top-level of the REPL.
- Print bit-vectors using the Common Lisp (and Guile) reader syntax. For example `#*1100110`. Enhanced the reader to read this format.
- Various REPL enhancements and new features:
  - The `-w` switch to create a new REPL window can be followed by various sub-options to control *how* and where the window is created. For example `-wbrowser` creates a new window using your default web browser.
  - Prompts are now normally specified using `printf`-style templates. The normal prompt template is specified by the `input-prompt1` variable, while continuation lines use `input-prompt2`. These can be initialized by command-line options `console:prompt1` and `console:prompt2`, or otherwise use language-specific defaults. You can still use `set-input-port-prompter!` to set a more general prompt-procedure, but it is now only called for the initial line of a command, not continuation lines.
  - The new `--with-jline3` configure option builds support for the JLine (version 3) (`https://github.com/jline/jline3`) library for handling console input, similar to GNU readline.
  - Context-dependent command-completion (tab-completion) works when using JLine.
- Various REPL enhancements when using DomTerm (`http://domterm.org/`).
  - If you “print” an XML/HTML node, it gets inserted into the DomTerm objects. You print images, tables, fancy text, and more.
  - If you “print” a picture object or a `BuferredImage` the picture is shown in the DomTerm console.
  - You can load or modify styles with the `domterm-load-stylesheet` procedure.
  - When pretty-printing, calculation of line-breaks and indentation is handled by DomTerm. If you change the window width, DomTerm will dynamically recalculate the line-breaks of previous pretten output. This works even in the case of a session saved to an HTML file, as long as JavaScript is enabled.
  - Hide/show buttons are emitted as part of the default prompt.
- Multiple literals that have the same value (as in `equal?`) get compiled to the same object.
- The syntax `&<[expr]` is now equivalent to `&<{&[expr]}`, assuming `expr` is an expression that evaluates to a string that named an existing file. That file is read is the result is the contents of the file (as if by `(path-data expr)`).

Kawa 2.1 (October 26, 2015)

Lots of little changes, and some big changes to sequences and strings.

- Enhancements to the Kawa tutorial.
- Added `parameter` as a new typename, for Scheme parameter objects. It can be parameterized (for example `parameter[string]`) for better type inference when "calling" (reading) the parameter.
- We now define "interactive mode" as a REPL or a source module that uses the default global top-level environment *or* a source module imported/required by a interactive module. Interactive mode attempts to support dynamic re-definition and re-loading of function and other definitions. This is a work-in-progres; interactive mode currently uses extra indirection to support re-definitions (at a slight performance cost).
- Various changes and fixes in Path/URI handling. Most significantly, the resolve argorithm used by `resolve-uri` was re-written to use the algorithm from RFC-3986, rather than the obsolete RFC-2396 algorithm used by `java.net.URI.resolve`.
- Change to mangle class and package name in Symbolic Freedom (`https://blogs.oracle.com/jrose/entry/symbolic_freedom_in_the_vm`) style. This means that class names and class filenames usually match the source file, even if special charaters are used, except for a small number of disallowed characters. Note this is currently *only* used for class and package names.
- Allow `'synchronized` and `'strictfp` as access flags for methods.
- You can now have a type-specifier for `define-variable`.
- Better support for forward references between macros.
- Added unsigned primitive integer types `ubyte`, `ushort`, `uint`, and `ulong`. These are represented at run-time by the corresponding signed types, but Kawa generates code to do unsigned arithmethic and comparisons. Corresponding boxed classes are `gnu.math.UByte`, `gnu.math.UShort`, `gnu.math.UInt`, and `gnu.math.ULong`.
- Improvements and unification of sequences and strings:
  - The new `sequence` type generalizes lists, vectors, arrays, strings, and more. It is implemented as the `java.util.List` interface, but strings (`java.lang.CharSequence`) and Java arrays are compatible with `sequence` and converted as needed.
  - The `length` function is generalized to arbitrary sequences. (For strings it uses the `CharSequence.length` method, which returns the number of (16-bit) code units. This is different from the `string-length` function, which returns the number of Unicode code points.)
  - A new pseudo-character value `#\ignorable-char` is introduced. It is ignored in string-construction contexts.
  - The function-call syntax for indexing works for all sequences. If the sequence is a string, the result is the Unicode (20-bit) scalar value at the specified index. If index references the trailing surrogate of a surrogate pair the result is `#\ignorable-char`. This allows efficient indexing of strings: Handing of surrogate pairs are handled automatically as long as `#\ignorable-char` is skipped.
  - Indexing of uniform vector types (such as `s64vector` or `f64vector` or `u16vector`) now return the "standard" primitive type (such as `long` or `double`) or the

new unsigned primitive (such as `ushort`). This improves performance (since we can generally use primitive types), and improves compatibility with Java arrays. Specifically, `s64vector` now implements `Sequence<Long>`, and thus `java.util.List<Long>` Note that indexing a `f64vector` returns a `double` which as an object is a `java.lang.Double`, not the Kawa floating-point type `gnu.math.DFloNum`. The result is usually the same, but `eqv?` might return a different result than previously.

  - The arguments to `map`, `for-each`, and `vector-for-each` can now be any sequence (including strings and native arrays). The arguments to `vector-for-each` can now be arbitrary `java.util.List` values. All of these are inlined. If the sequence type is known, more efficient custom code is generated.
  - A range represents an enumerable sequence, normally integers, but it is represented compactly using the start value, the step (usually 1), and size. There is a new convenient syntax for writing a range: if `i` and `j` are integers then `[i <=: j]` is the sequence of integers starting at `i` and ending at `j` (inclusive). You can also write `[i <=: j]` (excludes the upper bound), `[i >: j]` (counts down to `j`, exclusive), and `[i >=: j]` (counts down to `j`, inclusive).
  - You can use a sequences of integers to index a sequence. The result is the sequence of the selected elements. In general `(seq [i0 ... in])` is `[(seq i0) ... (seq in)]`. This work well with ranges: `(seq [i <: j])` is the subsequence of `seq` from `i` to `j` (exclusive).

    If the `seq` is a string (a `CharSequence`) then the result is also a string. In this case the indexing behavior is slightly different in that indexing selects (16-bit) code units, which are combined to a string.

- A new `dynamic` type is like `Object`. However, it forces runtime lookup and type-checking, and supresses compile-time type check and errors. (This is similar to C#. It is useful as an escape hatch if we ever implement traditional strict static type-checking.)
- Specifying the parameter type or return type of a function or method without a '`::`' is deprecated and results in a warning.
- In `--r7rs` mode: The '`l`' exponent suffix of a number literal creates a floating-point double, rather than a `BigInteger`.
- Added the hyperbolic functions: sinh, cosh, tanh, asinh, acosh, atanh.
- The `equal?` function can now handle cyclic lists and vectors. So can `equal-hash`.
- The command-line option `--with-arg-count=N` allows finer control of command-line-processing. It is used before an "action", and specifies the `N` arguments following the action are set as the command-line-arguments. After the action, command-line-processing continues following those `N` arguments.
- Added the R6RS module `(rnrs arithmetic bitwise)`.
- The `kawa.repl` argument processor now handles `-D` options.
- The new `class` sub-form of `import` allows you to import classes, and give them abbreviated names, like the Java `import` statement. The new form is more compact and convenient than `define-alias`.

  You can also use a classname directly, as a symbol, instead of writing it in the form of a list:

```
(import (only java.lang.Math PI))
```

- In the `only` clause of the `import` syntax you can now directly rename, without having to write a `rename` clause.
- Changes in the calling-convention for `--full-tailcalls` yields a substantial speed-up in some situations.
- The type of boolean literals `#f` and `#t` is now primitive `boolean` rather than `java.lang.Boolean`.
- General multi-dimensional arrays can be indexed with function call notation. E.g. `(arr i j k)` is equivalent to `(array-ref a i j k)`. You can also use `set!` with either `array-ref` or function call notation.
- The `#!null` value (Java `null`) is now considered false, not true. Likewise for non-canonical false Boolean objects (i.e. all instances of `java.lang.Boolean` for which `booleanValue` returns false, not just `Boolean.FALSE`).
- New standard libraries `(kawa base)` and `(kawa reflect)`.
- You can now use patterns in the `let` form and related forms.
- Implemented the lambda lifting (`http://en.wikipedia.org/wiki/Lambda_lifting`) optimzation.
- An expression that has type T is now considered compatible with a context requiring an interface type I only if T implements I (or T is Object). (Before they were considered possibly-compatible if T was non-final because the run-time class might be a subclass of T that implements I.)
- New `--console` flag forces input to be treated as an interactive console, with prompting. This is needed on Windows under Emacs, where `System.console()` gives the wrong result.
- You can now in a sub-class reference fields from not-yet-compiled super-classes. (This doesn't work for methods yet.)
- The `(? name::type value)` operator supports conditional binding. The `(! name::type value)` operator supports unconditional binding; it is similar to `define-constant`, but supports patterns.
- More efficient implementation of `call-with-values`: If either argument is a fixed-arity lambda expression it is inlined. Better type-checking of both `call-with-values` and `values`.
- Jamison Hope enhanced the support for quaternions, primarily the new `(kawa rotations)` library.

Kawa 2.0 (December 2 2014)

There are many new features, but the big one is R7RS compatibility.

- New `define-alias` can define aliases for static class members.
- The treatment of keywords is changing to not be self-evaluating (in Scheme). If you want a literal keyword, you should quote it. Unquoted keywords should only be used for keyword arguments. (This will be enforced in a future release.) The compiler now warns about badly formed keyword arguments, for example if a value is missing following a keyword.

- The default is now Java 7, rather than Java 6. This means the checked-in source code is pre-processed for Java 7, and future binary releases will require Java 7.
- The behavior of parameters and fluid variables has changed. Setting a parameter no longer changes its value in already-running sub-threads. The implementation is simpler and should be more efficient.
- The form `define-early-constant` is similar to `define-constant`, but it is evaluated in a module's class initializer (or constructor in the case of a non-static definition).
- Almost all of R7RS is now working:
  - Importing a SRFI library can now use the syntax `(import (srfi N [name]))`
  - The various standard libraries such as `(scheme base)` are implemented.
  - The functions `eval` and `load` can now take an environment-specifier. Implemented the `environment` function.
  - Extended `numerator`, `denominator`, `gcd`, and `lcm` to inexacts.
  - The full R7RS library functionality is working, including `define-library` The keyword `export` is now a synonym for `module-export`, and both support the `rename` keyword. The `prefix` option of `import` now works.
  - The `cond-expand` form now supports the `library` clause.
  - Implemented `make-promise` and `delay-force` (equivalent to the older name `lazy`).
  - Changed `include` so that by default it first seaches the directory containing the included file, so by default it has the same effect as `include-relative`. However, you can override the search path with the `-Dkawa.include.path` property. Also implemented `include-ci`.
  - Implemented `define-values`.
  - Fixed `string->number` to correctly handle a radix specifier in the string.
  - The `read` procedure now returns mutable pairs.
  - If you need to use `...` in a `syntax-rules` template you can use `(... template)`, which disables the special meaning of `...` in `template`. (This is an extension of the older `(... ...)`.)
  - Alternatively, you can can write `(syntax-rules dots (literals) rules)`. The symbol `dots` replaces the functionality of `...` in the `rules`.
  - An underscore `_` in a `syntax-rules` pattern matches anything, and is ignored.
  - The `syntax-error` syntax (renamed from `%syntax-error`) allows error reporting in `syntax-rules` macros. (The older Kawa-specific `syntax-error` procedure was renamed to `report-syntax-error`.)
  - Implemented and documented R7RS exception handling: The syntax `guard` and the procedures `with-exception-handler`, `raise`, and `raise-continuable` all work. The `error` procedure is R7RS-compatible, and the procedures `error-object?`, `error-object-message`, `error-object-irritants`, `file-error?`, and `read-error?` were implemented.
  - Implemented `emergency-exit`, and modified `exit` so finally-blocks are executed.
  - Implemented `exact-integer?`, `floor/`, `floor-quotient`, `floor-remainder`, `truncate/`, `truncate-quotient`, and `truncate-remainder`.

  - The `letrec*` syntax is now supported. (It works the same as `letrec`, which is an allowed extension of `letrec`.)
  - The functions `utf8->string` and `string->utf8` are now documented in the manual.
- The changes to characters and strings are worth covering separately:
  - The `character` type is now a new primitive type (implemented as `int`). This can avoid boxing (object allocation)
  - There is also a new `character-or-eof`. (A union of `character` and the EOF value, except the latter is encoded as -1, thus avoiding object allocation.) The functions read-char and `peek-char` now return a `character-or-eof` value.
  - Functions like `string-ref` that take a character index would not take into account non-BMP characters (those whose value is greater than `#xffff`, thus requiring two surrogate characters). This was contrary to R6RS/R7RS. This has been fixed, though at some performance cost . (For example `string-ref` and `string-length` are no longer constant-time.)
  - Implemented a `string-cursor` API (`Strings.html#String-Cursor-API`) (based on Chibi Scheme). Thes allow efficient indexing, based on opaque cursors (actually counts of 16-bits `chars`).
  - Optimized `string-for-each`, which is now the preferred way to iterate through a string.
  - Implemented `string-map`.
  - New function `string-append!` for in-place appending to a mutable string.
  - New function `string-replace!` for replacing a substring of a string with some other string.
  - The SRFI-13 function `string-append/shared` is no longer automatically visible; you have to `(import (srfi :13 strings))` or similar.
- The `module-name` form allows the name to be a list, as in a R6RS/R7RS-style library name.
- The syntax `@expression` is a *splicing form*. The `expression` must evaluate to a sequence (vector, list, array, etc). The function application or constructor form is equivalent to all the elements of the sequence.
- The parameter object `current-path` returns (or sets) the default directory of the current thread.
- Add convenience procedures and syntax for working with processes (`Processes.html`): `run-process`, `process-exit-wait`, `process-exit-ok?`, `&cmd`, ``&` ``, `&sh`.
- The functions `path-bytes`, and `path-data` can read or write the entire contents of a file (`http://www.gnu.org/software/kawa/Reading-and-writing-whole-files.html`). Alternatively, you can use the short-hand syntax: `&<{pname}` `&>{pname}` `&>>{pname}`. These work with "blobs" which may be text or binary depending on context.
- The initial values of `(current-output-port)` and `(current-error-port)` are now hybrid textual/binary ports. This means you can call `write-bytevector` and `write-u8` on them, making it possible for an application to write binary data to standard output. Similarly, initial value of `(current-input-port)` is a hybrid textual/binary port, but only if there is no console (standard input is not a tty).

- Jamison Hope contributed support for quaternions (`http://en.wikipedia.org/wiki/Quaternion`), a generalization of complex numbers containing 4 real components.
- Andrea Bernardini contributed an optimized implementation of `case` expressions. He was sponsored by Google Summer of Code.
- The `kawa.sh` shell script (which is installed as `kawa` when *not* configuring with `--enable-kawa-frontend`) now handles `-D` and `-J` options. The `kawa.sh` script is now also built when usint Ant.
- The `cond-expand` features `java-6` though `java-9` are now set based on the `System` property `"java.version"` (rather than how Kawa was configured).
- An Emacs-style `coding` declaration allows you to specify the encoding of a Scheme source file.
- The command-line option `--debug-syntax-pattern-match` prints logging importation to standard error when a `syntax-rules` or `syntax-case` pattern matches.
- SRFI-60 (Integers as Bits) (`http://srfi.schemers.org/srfi-60/srfi-60.html`) is now fully implemented.
- Ported SRFI-101 (`http://srfi.schemers.org/srfi-101/srfi-101.html`). These are immutable (read-only) lists with fast (logarithmic) indexing and functional update (i.e. return a modified list). These are implemented by a `RAPair` class which extends the generic `pair` type, which means that most code that expects a standard list will work on these lists as well.
- The class `kawa.lib.kawa.expressions` contains an experimental Scheme API for manipulating and validating expressions.
- Internal: Changed representation used for multiple values to an abstract class with multiple implementations.
- Internal: Started converting to more standard Java code formatting and indentation conventions, rather than GNU conventions. Some files converted; this is ongoing work.
- Internal: Various I/O-related classes moved to new package `gnu.kawa.io`.
- Various changes to the `configure+make` build framework: A C compiler is now only needed if you configure with `--enable-kawa-frontend`. Improved support for building under Windows (using MinGW/MSYS).
- Support for building with GCJ (`http://gcc.gnu.org/java/`) was removed.

Kawa 1.14 (October 4, 2013)

- You can pass flags from the `kawa` front-end to the `java` launcher using `-J` and `-D` flags. The `kawa` front-end now passes the `kawa.command.line` property to Java; this is used by the `(command-line)` procedure.
- Various improvements to the shell-script handling, including re-written documentation (`Scripts.html`).
- Some initial support for Java 8.
- More of R7RS is now working:
  - After adding list procedures `make-list`, `list-copy`, `list-set!` all the R7RS list procedures are implemented.

   - Other added procedures: `square`, `boolean=?`, `string-copy!`, `digit-value`, `get-environment-variable`, `get-environment-variables`, `current-second`, `current-jiffy`, `jiffies-per-second`, and `features`.
   - The predicates `finite?`, `infinite?`, and `nan?` are generalized to complex numbers.
   - The procedures `write`, `write-simple`, and `write-shared` are now consistent with R7RS.
   - String and character comparison functions are generalized to more than two arguments (but restricted to strings or characters, respectively).
   - The procedures `string-copy`, `string->list`, and `string-fill!` now take optional (start,end)-bounds. All of the R7RS string functions are now implemented.
   - Support `=>` syntax in `case` form.
   - Support backslash-escaped special characters in symbols when inside vertical bars, such as `'|Hello\nworld|`.
   - The new functions and syntax are documented in the Kawa manual (`index.html`); look for the functions in the index (`Overall-Index.html`).
- Added `define-private-alias` keyword.
- Extended string quasi-literals (templates) (`Strings.html#String-templates`) as specified by SRFI-109 (`http://srfi.schemers.org/srfi-109/srfi-109.html`). For example, if `name` has the value `"John"`, then:

  ```
  &{Hello &[name]!}
  ```

  evaluates to: `"Hello John!"`.
- Named quasi-literal constructors as specified by SRFI-108 (`http://srfi.schemers.org/srfi-108/srfi-108.html`).
- A symbol having the form `->type` is a type conversion function that converts a value to `type`.
- New and improved check for void-valued expressions in a context requiring a value. This is controlled by the new option `--warn-void-used`, which defaults to true.
- The `datum->syntax` procedure takes an optional third parameter to specify the source location. See `testsuite/srfi-108-test.scm` for an example.
- Instead of specifying `--main` the command line, you can now specify `(module-compile-options: main: #t)` in the Scheme file. This makes it easier to compile one or more application (main) modules along with other modules.
- A change to the data structure used to detect never-returning procedure uses a lot less memory. (Kawa 1.13 implemented a conservative detection of when a procedure cannot return. This analysis would sometimes cause the Kawa compiler to run out of memory. The improved analysis uses the same basic algorithm, but with a more space-efficient "inverted" data structure.)
- Multiple fixes to get Emacs Lisp (JEmacs) working (somewhat) again.

Kawa 1.13 (December 10, 2012)

- We now do a simple (conservative) analysis of when a procedure cannot return. This is combined with earlier and more precise analysis of reachable code. Not only does this catch programmer errors better, but it also avoids some internal compiler errors, because Kawa could get confused by unreachable code.

- Implement 2-argument version of `log` function, as specified by R6RS and R7RS (and, prematurely, the Kawa documentation).
- Implement the R7RS `bytevector` functions. The `bytevector` type is a synonym for older `u8vector` type.
- Implement R7RS `vector` procedures. Various procedures now take (start,end)-bounds.
- Implement most of the R7RS input/output proecdures. Most significant enhancement is support for R7RS-conforming binary ports.
- Various enhancements to the manual, including merging in lots of text from R7RS.
- Improved Android support, including a more convenient Ant script contributed by Julien Rousseau. Also, documentation merged into manual.

Kawa 1.12 (May 30, 2012)

- Implement a compile-time data-flow framework, similar to Single Static Assignment. This enables better type inference, improves some warnings/errors, and enables some optimizations.
- Jamison Hope added support for co-variant return types and bridge methods for generics.
- Macros were improved and more standards-conforming:
  - `datum->syntax` and `syntax->datum` are preferred names for `datum->syntax-object` and `syntax-object->datum`.
  - Implemented `bound-identifier=?` and re-wrote implementation of `free-identifier=?`.
  - Implement `unsyntax` and `unsyntax-splicing`, along with the reader prefixes `#,` and `#,@`.
- New and improved lazy evaluation functionality:
  - Lazy values (resulting from `delay` or `future`) are implicitly forced as needed. This makes “lazy programming” more convenient.
  - New type `promise`.
  - The semantics of promises (`delay` etc) is now compatible with SRFI 45 (`http://srfi.schemers.org/srfi-45/srfi-45.html`).
  - “Blank promises” are useful for passing data between processes, logic programmming, and more. New functions `promise-set-value!`, `promise-set-alias!`, `promise-set-exception!`, and `promise-set-thunk!`.
  - The stream functions of SRFI-41 (`http://srfi.schemers.org/srfi-41/srfi-41.html`) were re-implemented to use the new promise functionality.
- Different functions in the same module can be compiled with or without full tailcall support. You can control this by using `full-tailcalls` in `with-compile-options`. You can also control `full-tailcalls` using `module-compile-options`.
- Charles Turner (sponsored by Google’s Summer of Code (`http://code.google.com/soc/`)) enhanced the printer with support for SRFI-38: External Representation for Data With Shared Structure (`http://srfi.schemers.org/srfi-38/`).

- Optimize tail-recursion in module-level procedures. (We used to only do this for internal functions, for reasons that are no longer relevant.)
- Add support for building Kawa on Windows using configure+make (autotools) and Cygwin.
- Some support for parameterized (generic) types:

  ```
  Type[Arg1 Arg2 ... ArgN]
  ```

  is more-or-less equivalent to Java's:

  ```
  Type<Arg1, Arg2, ..., ArgN>
  ```
- New language options `--r5rs`, `--r6rs`, and `--r7rs` provide better compatibility with those Scheme standards. (This is a work-in-progress.) For example `--r6rs` aims to disable Kawa extensions that conflict with R6RS. It does not aim to disable all extensions, only incompatible extensions. So far these extensions disable the colon operator and keyword literals. Selecting `--r5rs` makes symbols by default case-insensitive.
- The special tokens `#!fold-case` and `#!no-fold-case` act like comments except they enable or disable case-folding of symbols. The old `symbol-read-case` global is now only checked when a LispReader is created, not each time a symbol is read.
- You can now use square brackets to construct immutable sequences (vectors).
- A record type defined using `define-record-type` is now compiled to a class that is a member of the module class.
- Annotations are now supported. This example (`http://per.bothner.com/blog/2011/Using-JAXB-annotations/`) shows how to use JAXB (`http://java.sun.com/xml/downloads/jaxb.html`) annotations to automatically convert between between Java objects and XML files.
- Prevent mutation of vector literals.
- More R6RS procedures: `vector-map`, `vector-for-each`, `string-for-each`, `real-valued?`, `rational-valued?`, `integer-valued?`, `finite?`, `infinite?`, `nan?`, `exact-integer-sqrt`.
- SRFI-14 (`http://srfi.schemers.org/srfi-14/srfi-14.html`) ("character sets") and SRFI-41 (`http://srfi.schemers.org/srfi-41/srfi-41.html`) ("streams") are now supported, thanks to porting done by Jamison Hope.
- Kawa now runs under JDK 1.7. This mostly involved fixing some errors in `StackMapTable` generation.
- You can now have a class created by `define-simple-class` with the same name as the module class. For example `(define-simple-class foo ...)` in a file `foo.scm`. The defined class will serve dual-purpose as the module class.
- Improvements in separating compile-time from run-time code, reducing the size of the runtime jar used for compiled code.
- In the `cond-expand` conditional form you can now use `class-exists:ClassName` as a feature "name" to tests that `ClassName` exists.

Kawa 1.11 (November 11, 2010)

- A new Kawa logo, contributed by Jakub Jankiewicz (`http://jcubic.pl`).

- A new `--warn-unknown-member` option, which generalizes `--warn-invoke-unknown-method` to fields as well as methods.
- A new `kawac` task (`ant-kawac.html`), useful for Ant `build.xml` files, contributed by Jamison Hope.
- Updated Android support (`http: / / per . bothner . com / blog / 2010 / AndroidHelloScheme`).
- New `define-enum` macro (`Enumerations.html`) contributed by Jamison Hope.
- Access specifiers `'final` and `'enum` are now allowed in `define-class` and related forms.
- Optimized `odd?` and `even?`.
- If you specify the type of a `#!rest` parameter as an array type, that will now be used for the "varargs" method parameter. (Before only object arrays did this.)
- When constructing an object and there is no matching constructor method, look for `"add"` methods in addition to `"set"` methods. Also, allow passing constructor args as well as keyword setters. See here (`Allocating-objects.html`) for the gory details.
- New `expand` function (contributed by Helmut Eller, and enabled by `(require 'syntax-utils)`) for converting Scheme expressions to macro-expanded forms.
- SAM-conversion (`Anonymous-classes.html#SAM-conversion`): In a context that expects a Single Abstract Method (SAM) type (for example `java.lang.Runnable`), if you pass a lambda you will get an `object` where the lambda implements the abstract method.
- In interactive mode allow dynamic rebinding of procedures. I.e. if you re-define a procedure, the old procedure objects gets modified in-place and re-used, rather than creating a new procedure object. Thus calls in existing procedures will call the new version.
- Fix various threading issues related to compilation and eval.
- When `format` returns a string, return a `java.lang.String` rather than a `gnu.lists.FString`. Also, add some minor optimization.
- Inheritance of environments and fluid variables now work properly for all child threads, not just ones created using `future`.

Kawa 1.10 (July 24, 2010)

- Now defaults to using Java 6, when compiling from source. The pre-built `jar` works with Java 5, but makes use of some Java 6 features (`javax.script`, built-in HTTP server) if available.
- You can write XML literals (`XML-literals.html`) in Scheme code prefixed by a `#`, for example:

  ```
  #<p>The result is &{result}.</p>
  ```
- New functions `element-name` and `attribute-name`.
- Various Web server improvements (`Server-side-scripts.html`). You have the option of using JDK 6's builtin web-server (`Options.html#Options-for-web-servers`) for auto-configued web pages (`Self-configuring-page-scripts.html`). Automatic import of web server functions, so you should not need to `(import 'http)` any more.

- Kawa hashtables (`Hash-tables.html`) now extend `java.util.Map`.
- If a source file is specified on the `kawa` command line without any options, it is read and compiled as a whole module before it is run. In contrast, if you want to read and evaluate a source file line-by-line you must use the `-f` flag.
- You can specify a class name on the `kawa` command line:

  ```
  $ kawa fully.qualified.name
  ```

  This is like the `java` command. but you don't need to specify the path to the Kawa runtime library, and you don't need a `main` method (as long as the class is `Runnable`).
- The usual bug-fixes, including better handling of the `~F format` directive; and fix in handling of macro hygiene of the `lambda` (bug #27042 (`https://savannah.gnu.org/bugs/index.php?27042`)).
- Spaces are now optional before and after the '::' in type specifiers. The preferred syntax leave no space after the '::', as in:

  ```
  (define xx ::int 1)
  ```
- `define-for-syntax` and `begin-for-syntax` work.
- You can now use `car`, `cdr` etc to work with `syntax` objects that wrap lists, as in SRFI-72.
- You can now define a package alias:

  ```
  (define-alias jutil java.util)
  (define mylist :: jutil:List (jutil:ArrayList))
  ```
- `--module-static` is now the default. A new `--module-nonstatic` (or `--no-module-static`) option can be used to get the old behavior.
- You can use `access:` to specify that a field is `'volatile` or `'transient`.
- You can now have type-specifiers for multiple variables in a `do`.
- Imported variables are read-only.
- Exported variables are only made into Locations when needed.
- The letter used for the exponent in a floating-point literal determines its type: `12s2` is a `java.lang.Float`, `12d2` is a `java.lang.Double`, `12l2` is a `java.math.BigInteger`, `12e2` is a `gnu.math.DFloat`.
- Internal: Asking for a `.class` file using `getResourceAsStream` on an `ArrayClassLoader` will now open a `ByteArrayInputStream` on the class bytes.
- A new `disassemble` function.
- If `exp1` has type `int`, the type of `(+ exp1 1)` is now (32-bit) `int`, rather than (unlimited-precision) `integer`. Similar for `long` expressions, other arithmetic operations (as appropriate), and other untyped integer literals (as long as they fit in 32/64 bits respectively).
- Many more oprimization/specializations of arithmetic, especially when argument types are known.
- Top-level bindings in a module compiled with `--main` are now implicitly module-private, unless there is an explicit `module-export`.
- SRFI-2 (`http://srfi.schemers.org/srfi-2/srfi-2.html`) (`and-let*`: an `and` with local bindings, a guarded `*` special form) is now supported.

- The reader now supports shared sub-objects, as in SRFI-38 (`http://srfi.schemers.org/srfi-38/srfi-38.html`) and Common Lisp: `(#2=(3 4) 9 #2# #2#)`. (Writing shared sub-objects is not yet implemented.)
- A module compiled with `--main` by default exports no bindings (unless overriden by an explicit `module-export`).
- Factor out compile-time only code from run-time code. The new `kawart-version.jar` is smaller because it has less compile-time only code. (Work in progress.)
- More changes for R6RS compatibility:
  - The reader now recognizes `+nan.0`, `+inf.0` and variations.
  - The `div`, `mod`, `div0`, `mod0`, `div-and-mod`, `div0-and-mod0`, `inexact` and `exact` functions were implemented.
  - `command-line` and `exit`.

Kawa 1.9.90 (August 8, 2009)

- Support for `javax.script`.
- Support for regular expressions (`Regular-expressions.html`).
- Performance improvements:
  - Emit `iinc` instruction (to increment a local `int` by a constant).
  - Inline the `not` function if the argument is constant.
  - If `call-with-current-continuation` is only used to exit a block in the current method, optimize to a `goto`.
  - Generate `StackMapTable` attributes when targeting Java 6.
  - Kawa can now inline a function with multiple calls (without code duplication) if all call sites have the same return location (continuation). For example: `(if p (f a) (f b))`. Also mutually tail-recursive functions are inlined, so you get constant stack space even without `--full-tailcalls`. (Thanks for Helmut Eller for a prototype.)
- A number of changes for R6RS compatibility:
  - The `char-titlecase`, `char-foldcase`, `char-title-case?` library functions are implemented.
  - Imported variables are read-only.
  - Support the R6RS `import` keyword, including support for renaming.
  - Support the R6RS `export` keyword (though without support for renaming).
  - Implemented the `(rnrs hashtables)` library.
  - Implemented the `(rnrs sorting)` library.
  - CommonLisp-style keyword syntax is no longer supported (for Scheme): A colon followed by an identifier is no longer a keyword (though an identifier followed by a colon is still a keyword). (One reason for this change is to support SRFI-97.)
  - The character names `#\delete`, `#\alarm`, `#\vtab` are now supported. The old names `#\del`, `#\rubout`, and `#\bel` are deprecated.
  - Hex escapes in character literals are supported. These are now printed where we before printed octal escapes.

  - A hex escape in a string literal should be terminated by a semi-colon, but for compatibily any other non-hex-digit will also terminate the escape. (A terminating semi-colon will be skipped, though a different terminator will be included in the string.)
  - A backslash-whitespace escape in a string literal will not only ignore the whitespace through the end of the line, but also any initial whitespace at the start of the following line.
  - The comment prefix `#;` skips the following S-expression, as specified by SRFI-62 (`http://srfi.schemers.org/srfi-62/srfi-62.html`).
  - All the R6RS exact bitwise arithmetic (`http://www.r6rs.org/final/html/r6rs-lib/r6rs-lib-Z-H-12.html#node_sec_11.4`) functions are now implemented and documented in the manual (`Logical-Number-Operations.html`). The new standard functions (for example `bitwise-and`) are now preferred over the old functions (for example `logand`).
  - If `delete-file` fails, throws an exception instead of returning `#f`.
- The code-base now by default assumes Java 5 (JDK 1.5 or newer), and pre-built `jar` files will require Java 5. Also, the Kawa source code now uses generics, so you need to use a generics-aware `javac`, passing it the appropriate `--target` flag.
- New SRFIs supported:
  - SRFI-62 (`http://srfi.schemers.org/srfi-62/srfi-62.html`) - S-expression comments.
  - SRFI-64 (`http://srfi.schemers.org/srfi-64/srfi-64.html`) - Scheme API for test suites.
  - SRFI-95 (`http://srfi.schemers.org/srfi-95/srfi-95.html`) - Sorting and Merging.
  - SRFI-97 (`http://srfi.schemers.org/srfi-97/srfi-97.html`) - Names for SRFI Libraries. This is a naming convention for R6RS `import` statements to reference SRFI libraries.
- In BRL text outside square brackets (or nested like `]this[`) now evaluates to `UnescapedData`, which a Scheme quoted string evaluates to `String`, rather than an `FString`. (All of the mentioned types implement `java.lang.CharSequence`.)
- You can now run Kawa Scheme programs on Android (`http://per.bothner.com/blog/2009/AndroidHelloScheme/`), Google's mobile-phone operating system.
- The macro `resource-url` is useful for accessing resources.
- A new command-line option `--target` (or `-target`) similar to `javac`'s `-target` option.
- If there is no console, by default create a window as if `-w` was specificed.
- If a class method (defined in `define-class`, `define-simple-class` or `object`) does not have its parameter or return type specified, search the super-classes/interfaces for matching methods (same name and number of parameters), and if these are consistent, use that type.
- Trying to modify the `car` or `cdr` of a literal list now throws an exception.
- The `.zip` archive created by `compile-file` is now compressed.
- Java5-style varargs-methods are recognized as such.

- When evaluating or loading a source file, we now always compile to bytecode, rather than interpreting “simple” expressions. This makes semantics and performance more consistent, and gives us better exception stack traces.
- The Scheme type specifier `<integer>` now handles automatic conversion from `java.math.BigInteger` and the `java.lang` classes `Long`, `Integer`, `Short`, and `Byte`. The various standard functions that work on `<integer>` (for example `gcd` and `arithmetic-shift`) can be passed (say) a `java.lang.Integer`. The generic functions such as `+` and the real function `modulo` should also work. (The result is still a `gnu.math.IntNum`.)
- If a name such as (`java.util`) is lexically unbound, and there is a known package with that name, return the `java.lang.Package` instance. Also, the colon operator is extended so that `package:name` evaluates to the `Class` for `package.name`.
- `` `prefix:,expression `` works - it finds a symbol in `prefix`’s package (aka namespace), whose local-name is the value of `expression`.
- A quantity `3.0cm` is now syntactic sugar for `(* 3.0 unit:cm)`. Similarly:
  `(define-unit name value)`
  is equivalent to:
  `(define-constant unit:name value)`
  This means that unit names follow normal name-lookup rules (except being in the `unit` “package”), so for example you can have local unit definitions.
- You can specify whether a class has public or package access, and whether it is translated to an interface or class.
- You can declare an abstract method by writing `#!abstract` as its body.
- If a name of the form `type?` is undefined, but `type` is defined, then treat the former as `(lambda (x) (instance? x type))`.
- A major incompatible (but long-sought) change: Java strings (i.e. `java.lang.String` values) are now Scheme strings, rather than Scheme symbols. Since Scheme strings are mutable, while Java `String`s are not, we use a different type for mutable strings: `gnu.lists.FString` (this is not a change). Scheme string literals are `java.lang.String` values. The common type for Scheme string is `java.lang.CharSequence` (which was introducted in JDK 1.4).

  Scheme symbols are now instances of `gnu.mapping.Symbol` (`api/gnu/mapping/Symbol.html`), specifically the `SimpleSymbol` class.
- A fully-qualified class name such as `java.lang.Integer` now evaluates to the corresponding `java.lang.Class` object. I.e. it is equivalent to the Java term `java.lang.Integer.class`. This assumes that the name does not have a lexical binding, *and* that it exists in the class-path at compile time.

  Array class names (such as `java.lang.Integer[]`) and primitive types (such as `int`) also work.

  The older angle-bracket syntax `<java.lang.Integer>` also works and has the same meaning. It also evaluates to a `Class`. It used to evaluate to a `Type` (`api/gnu/bytecode/Type.html`), so this is a change.

  The name bound by a `define-simple-class` now evaluates to a `Class`, rather than a `ClassType` (`api/gnu/bytecode/ClassType.html`). A `define-simple-class` is not allowed to reference non-static module-level bindings; for that use `define-class`.

- New convenience macro `define-syntax-case` (`Syntax-and-conditional-compilation.html`).

Kawa 1.9.1 (January 23, 2007)

- Fix some problems building Kawa from source using `configure+make`.

Kawa 1.9.0 (January 21, 2007)

- New types and functions for working with paths and URIs (`Paths.html`).
- Reader macros URI, namespace, duration.
- Simplified build using gcj (`Source-distribution.html`), and added configure flag –with-gcj-dbtool.
- If two "word" values are written, a space is written between them. A word is most Scheme values, including numbers and lists. A Scheme string is treated as a word by `write` but by not `display`.
- A new `--pedantic` command-line flag. It currently only affects the XQuery parser.
- The `load-compile` procedure was removed.
- The string printed by the `--version` switch now includes the Subversion revision and date (but only if Kawa was built using `make` rather than `ant` from a checked-out Subversion tree).
- Kawa development now uses the Subversion (svn) (`http://subversion.tigris.org/`) version control system instead of CVS.
- Show file/line/column on unbound symbols (both when interpreted and when compiled).
- Cycles are now allowed between `require`'d modules. Also, compiling at set of modules that depend on each other can now specified on the compilation command line in any order, as long as needed `require` forms are given.
- The "colon notation" has been generalized. (`PathExpressions.html`). The syntax `object:name` generally means to extract a component with a given `name` from `object`, which may be an object, a class, or a namespace.
- New command-line options `--debug-error-prints-stack-trace` and `--debug-warning-prints-stack-trace` provide stack trace on static error messages.
- The license for the Kawa software (`Software-License.html`) has been changed to the X11/MIT license (`http://opensource.org/licenses/mit-license.php`).
- A much more convenient syntax for working with Java arrays (`Array-operations.html`).

  The same function-call syntax also works for Scheme vectors, uniform vectors, strings, lists - and anything else that implements `java.util.List`.
- The fields and methods of a class and its bases classes are in scope within methods of the class.
- Unnamed procedures (such as lambda expressions) are printed with the source filename and line.
- The numeric compare functions (`=`, `<=`, etc) and `number->string` now work when passed standard Java `Number` objects (such as `java.lang.Long` or `java.math.BigDecimal`).

- SRFI-10 (`http://srfi.schemers.org/srfi-10/srfi-10.html`) is now implemented, providing the `#,(name args ...)` form. Predefined constructor `names` so far are `URI` and `namespace`. The `define-reader-ctor` function is available if you `(require 'srfi-10)`.
- A new `--script` option makes it easier to write Unix shell scripts.
- Allow general URLs for loading (including the `-f` flag), compilation and `open-input-file`, if the "file name" starts with a URL "scheme" like `http:`.
- Classes defined (*e.g.* with `define-simple-class`) in a module can now mutually reference each other. On the other hand, you can no longer `define-class` if the class extends a class rather than an interface; you must use `define-simple-class`.
- `KawaPageServlet` now automatically selects language.
- `provide` macro.
- `quasisyntax` and the convenience syntax `#``, from SRFI-72 (`http://srfi.schemers.org/srfi-72/srfi-72.html`).
- `define-for-syntax`, `syntax-source`, `syntax-line`, and `syntax-column`, for better compatibility with mzscheme.
- SRFI-34 (`http://srfi.schemers.org/srfi-34/srfi-34.html`) (Exception Handling for Programs), which implements `with-exception-handler`, `guard`, and `raise`, is now available, if you `(require 'srfi-34)`.
  Also, SRFI-35 (`http://srfi.schemers.org/srfi-35/srfi-35.html`) (Conditions) is available, if you `(require 'srfi-35)`.
- The `case-lambda` form from SRFI-16 (`http://srfi.schemers.org/srfi-16/srfi-16.html`) is now implemented more efficiently.

Kawa 1.8 (October 18, 2005)

SRFI-69 "Basic hash tables" (`http://srfi.schemers.org/srfi-69/srfi-69.html`) is now available, if you `(require 'hash-table)` or `(require 'srfi-69)`. This is an optimized and Java-compatible port whose default hash function calls the standard `hashCode` method.

A `define-simple-class` can now have one (or more) explicit constructor methods. These have the spcial name `*init*`. You can call superclass constructors or sibling constructors (`this` constructor calls) using the (admittedly verbose but powerful) `invoke-special` form.

The `runnable` function creates a `Runnable` from a `Procedure`. It is implemented using the new class `RunnableClosure`, which is now also used to implement `future`.

The `kawa` command can now be run "in-place" from the build directory: `$build_dir/bin/kawa`.

The special field name `class` in `(static-name type 'class)` or `(prefix:.class)` returns the `java.lang.Class` object corresponding to the `type` or `prefix`. This is similar to the Java syntax.

Contructing an instance (perhaps using `make`) of a class defined using `define-simple-class` in the current module is much more efficient, since it no longer uses reflection. (Optimizing classes defined using `define-class` is more difficult.) The constructor function defined by the `define-record-type` macro is also optimized.

You can now access instance methods using this short-hand: `(*:methodname instance arg ...)`
This is equivalent to: `(invoke instance 'methodname arg ...)`

You can now also access a fields using the same colon-notation as used for accessing methods, except you write a dot before the field name:
`(type:.fieldname)` `;;` is like: `(static-field type 'fieldname)`.
`(*:.fieldname instance)` `;;` is like: `(field 'fieldname instance)`
`(type:.fieldname instance)` `;;` is like: `(*:.fieldname (as instance type))`
These all work with `set!` - for example: `(set! (*:.fieldname instance) value)`.

In the above uses of colon-notation, a `type` can be any one of:
- a namespace prefix bound using `define-namespace` to a namespace uri of the form `"class:classname"`;
- a namespace prefix using `define-namespace` bound to a `<classname>` name, which can be a fully-qualified class name or a locally-declared class, or an alias (which might be an imported class);
- a fully qualified name of a class (that exists at compile-time), as in `(java.lang.Integer:toHexString 123)`; or
- a `<classname>` variable, for example: `(<list>:list3 11 12 13)`.

New fluid variables `*print-base*`, `*print-radix*`, `*print-right-margin*`, and `*print-miser-width*` can control output formatting. (These are based on Common Lisp.)

You can new emit elipsis (`...`) in the output of a `syntax` template using the syntax `(... ...)`, as in other `syntax-case` implementations.

The `args-fold` program-argument processor from SRFI-37 (`http://srfi.schemers.org/srfi-37/srfi-37.html`) is available after you `(require 'args-fold)` or `(require 'srfi-37)`.

The `fluid-let` form now works with lexical bindings, and should be more compatible with other Scheme implementations.

`(module-export namespace:prefix)` can be used to export a namespace prefix.

Static modules are now implemented more similarly to non-static modules. Specifically, the module body is not automatically run by the class initializer. To get the old behavior, use the new `--module-static-run` flag. Alternatively, instead of `(module-static #t)` use `(module-static 'init-run)`.

Implement SRFI-39 (`http://srfi.schemers.org/srfi-39/srfi-39.html`) "Parameter-objects". These are like anonymous fluid values and use the same implementation. `current-input-port`, `current-output-port`, and `current-error-port` are now parameters.

Infer types of variables declared with a `let`.

Character comparisons (such as `char-=?`, `char-ci<?`) implemented much more efficiently — and (if using Java5) work for characters not in the Basic Multilingual Plane.

Major re-write of symbol and namespace handling. A `Symbol` (`api/gnu/mapping/Symbol.html`) is now immutable, consisting of a "print-name" and a pointer to a `Namespace` (`api/gnu/mapping/Namespace.html`) (package). An `Environment` (`api/gnu/mapping/

Environment.html) is a mapping from Symbol to Location (api/gnu/mapping/Location.html).

Rename Interpreter to Language (api / gnu / expr / Language . html) and LispInterpreter to LispLanguage (api/gnu/kawa/lispexpr/LispLanguage.html).

Constant-time property list operations.

Namespace-prefixes are now always resolved at compile-time, never at run-time.

(define-namespace PREFIX <CLASS>) is loosely the same as (define-namespace PREFIX "class:CLASS") but does the right thing for classes defined in this module, including nested or non-simple classes.

Macros capture proper scope automatically, not just when using require. This allows some internal macros to become private.

Major re-write of the macro-handling and hygiene framework. Usable support for syntax-case; in fact some of the primitives (such as if) are now implemented using syntax-case. (syntax form) (or the short-cut #!form) evaluates to a syntax object. (define-syntax (mac x) tr) same as (define-syntax mac (lambda (x) tr)). The following non-hygienic forms are equivalent:

```
(define-macro (macro-name (param ...) transformer)
(define-macro macro-name (lambda (param ...) transformer))
(defmacro macro-name (PARAM ...) transformer)
```

Allow vectors and more general ellipsis-forms in patterns and templates.

A new configure switch --with-java-source=version allows you to tweak the Kawa sources to match Java compiler and libraries you're using. The default (and how the sources are distributed) is 2 (for "Java 2" – jdk 1.2 or better), but you can also select "1" (for jdk 1.1.x), and "5" for Java 5 (jdk 1.5). You can also specify a jdk version number: "1.4.1" is equivalent to "2" (for now). Note the default source-base is incompatible with Java 5 (or more generally JAXB 1.3 or DOM 3), unless you also --disable-xml.

Configure argument --with-servlet[=servlet-api.jar] replaces --enable-servlet.

Function argument in error message are now numbered starting at one. Type errors now give better error messages.

A new function calling convention, used for --full-tailcalls. A function call is split up in two parts: A match0/.../matchN method checks that the actual arguments match the expected formal arguments, and leaves them in the per-thread CallContext (api/gnu/mapping/CallContext.html). Then after the calling function returns, a zero-argument apply() methods evaluates the function body. This new convention has long-term advantages (performance, full continuations), but the most immediate benefit is better handling of generic (otherloaded) functions. There are also improved error messages.

Real numbers, characters, Lisp/Scheme strings (FString (api/gnu/lists/FString.html)) and symbols all now implement the Comparable interface.

In define-class/define-simple-class: [Most of this work was funded by Merced Systems (http://www.mercedsystems.com/).]

- You can specify access: ['private|'protected|'public|'package] to set the Java access permissions of fields and methods.
- Methods can be static by using the access: 'static specifier.

- The reflective routines `invoke` , `field` , `static-field` , `slot-ref` , `slot-set!` can now access non-public methods/fields when appropriate.
- Such classes are no longer initialized when the containing module is loaded.
- The `expr` in `init-form: expr` is now evaluated in the outer scope.
- A new `init: expr` evalues `expr` in the inner scope.
- An option name following `allocation:` can now be a string literal or a quoted symbol. The latter is preferred: `allocation: 'class`.
- Added `'static` as a synonym for `'class` following `allocation:`.
- Initialization of static field (`allocation: 'class init: expr`) now works, and is performed at class initialization time.
- You can use unnamed "dummy fields" to add initialization-time actions not tied to a field:

```
(define-simple-class Foo ()
  (:init (perform-some-action)))
```

Kawa 1.7.90 (2003)

Various fixes and better error messages in number parsing. Some optimizations for the divide function.

New framework for controlling compiler warnings and other features, supporting command-line flags, and the Scheme forms `with-compile-options` and `module-compile-options`. The flag `--warn-undefined-variable` is useful for catching typos. Implementation funded by Merced Systems (`http://www.mercedsystems.com/`).

New `invoke-special` syntax form (implemented by Chris Dean).

New `define-variable` form (similar to Common Lisp's `defvar`).

Kawa 1.7 (June 7, 2003)

`KawaPageServlet` (`api / gnu / kawa / servlet / KawaPageServlet . html`) allows automatic loading and on-the-fly compilation in a servlet engine. See http://www.gnu.org/software/qexo/simple-xquery-webapp.html (`. . / qexo / simple-xquery-webapp.html`).

The default source-base requires various Java 2 features, such as collection. However, `make-select1` will comment out Java2 dependencies, allowing you to build Kawa with an older Java implementation.

The `-f` flag and the load function can take an absolute URL. New Scheme functions `load-relative` and `base-uri`.

Imported implementation of cut and cute from SRFI-26 (`http://srfi.schemers.org/srfi-26/srfi-26.html`) (Notation for Specializing Parameters without Currying).

The way top-level definitions (including Scheme procedures) are mapped into Java fields is changed to use a mostly reversible mapping. (The mapping to method names remains more natural but non-reversible.)

`define-alias` of types can now be exported from a module.

New `--no-inline` and `--inline=none` options.

You can use `define-namespace` to define “namespace aliases”. This is used for the new short-hard syntax for method invocation:

```
(define-namespace Int32 "class:java.lang.Integer")
(Int32:toHexString 255) => "ff"
(Int32:toString (Int32:new "00255")) => "255"
```

Alternatively, you can write:

```
(java.lang.Integer:toHexString 255) => "ff"
```

SRFI-9 (`http://srfi.schemers.org/srfi-9/srfi-9.html`) (define-record-type) has been implemented, and compiled to a `define-class`, with efficient code.

The configure option `--with-collections` is now the default.

Unknowns are no longer automatically static.

If type not specified in a declaration, don’t infer it from it initial value. If no return type is specified for a function, default to `Object`, rather than the return type of the body. (The latter leads to undesirable different behaviour if definitions are re-arranged.)

You can now define and use classes defined using `object`, `define-class`, and `define-simple-class` from the “interpreter”, as well as the compiler. Also, a bug where inherited fields did not get initialized has been fixed.

There are several new procedures useful for servlets.

Numerical comparisions (`<`, `<=`, etc) now generates optimized bytecode if the types of the operands have certain known types. including efficient code for `<int>`, `<long>`, `<double>`, and `<integer>`. Much more code can now (with type declaration) be written just as efficiently in Scheme as in Java.

There have been some internal re-arranging of how Expressions are processed. The Scheme-specific Translator type now inherits from Compilation, which replaces the old Parser class. A Complation is now allocated much earlier, as part of parsing, and includes a SourceMessages object. SourcesMessages now includes (default) line number, which is used by Compilation for the `"current"` line numbers. The ExpWalker class includes a SourceMessages instance (which it gets from the Compilation). CanInline.inline method now takes ExpWalker parameter. Checking of the number or parameters, and mapping known procedures to Java methods are now both done during the inlining pass.

The user-visible effect is that Kawa can now emit error mesages more cleanly more places; the inlining pass can be more agressive, and can emit better error messages, which yields better type information. This gives us better code with fewer warnings about unknown methods.

Changes from Kawa 1.6.98 to 1.6.99.

A new language front-end handles a tiny subset of XSLT. An example is the check-format-users test in gnu/xquery/testsuite/Makefile.

There are now converters between SAX2 and Consumer events, and a basic implementation of XMLReader based on XMLParser.

The function as-xml prints a value in XML format.

Srfi-0 (cond-expand), srfi-8 (receive), and srfi-25 (multi-dimensional arrays) are now implemented. So is srfi-1 (list library), though that requires doing (require ’list-lib).

The JEmacs code is being re-organized, splitting out the Swing-dependent code into a separate gnu.jemacs.swing package. This should make it easier to add JEmacs implementation without Swing.

The class gnu.expr.Interpreter has various new 'eval' methods that are useful for evaluating Scheme/BRL/XQuery/... expressions from Java.

Kawa now uses current versions of autoconf, autoamke, and libtool, allowing the use of automake file inclusion.

The comparisons `<<`, `<=`, `-`, `>`, and `=>` now compile to optimized Java arithmetic if both operands are `<int>` or a literal that fits in `<int>`.

Changes from Kawa 1.6.97 to 1.6.98

Generated HTML and Postscrpt documents are no longer included in the source distribution. Get `kawa-doc-version.tar.gz` instead.

(format #t ...) and (format PORT ...) now returns #!void instead of #t.

Support fluid bindings (fluid-let) for any thread, not just Future and main.

A Unix script header `#!/PROGRAM` is ignored.

You can now take the same Kawa "web" program (written in Scheme, KRL/BRL, or XQuery) and run it as either a servlet or a CGI script.

There are a number of new functions for accessing HTTP requests and generating HTTP responses.

Kawa now supports a new experimental programming KRL (the "Kawa Report Language"). You select this language using –krl on the Kawa command link. It allows Scheme code to be inside template files, like HTML pages, using a syntax based on BRL (brl.sourceforge.net). However, KRL has soem experimental changes to both BRL and standard Scheme. There is also a BRL-compatibile mode, selected using –brl, though that currently only supports a subset of BRL functions.

If language is not explicitly specified and you're running a source file (e.g. "java kawa.repl myscript.xql"), Kawa tried to derive the language from the the filename extension (e.g. "xql"). It still defaults to Scheme if there is no extension or the extension is unrecognized.

New command-line option –output-format alias –format can be used to over-ride the format used to write out top-level (repl, load) values.

XMLPrinter can now print in (non-well-formed-XML) HTML.

Changes from Kawa 1.6.96 to 1.6.97

Changed lots of error messages to use pairs of single quotes rather than starting with a backquote (accent grave): 'name' instead of ‘name'. Many newer fonts make the latter look bad, so it is now discouraged.

The types `<String>` and `<java.lang.String>` new behave differently. The type `<java.lang.String>` now works just like (say) `<java.util.Hashtable>`. Converting an object to a `<java.lang.String>` is done by a simple coercion, so the incoming value must be a java.lang.String reference or null. The special type `<String>` converts any object to a java.string.String by calling toString; it also handles null by specially testing for it.

For convenience (and backwards compatibility) Kawa uses the type `<String>` (rather than `<java.lang.String>`) when it sees the Java type `java.lang`.String, for example in the argument to an `invoke`.

The default behaviour of '[' and '] was changed back to be token (word) constituents, matching R5RS and Common Lisp. However, you can easily change this behaviour using the new setBrackMode method or the defaultBracketMode static field in ReadTable.

You can now build Kawa from source using the Ant build system (from Apache's Jakarta project), as an alternative to using the traditional configure+make system. An advantage of Ant is that it works on most Java systems, without requiring a Unix shell and commands. Specifically, this makes it easy to build Kawa under MS-Windows. Thanks to James White for contributing this support.

Added (current-error-port) which does the obvious.

The new let-values and let-values* macros from srfi-11 provide a more convenient way to use multiple values.

All the abstract apply* and eval* methods now specify 'throws Throwable'. A bunch of code was changed to match. The main visible advantage is that the throw and primitive-throw procedures work for any Throwable without requiring it to be (confusingly) wrapped.

Changes from Kawa 1.6.95 to 1.6.96

A new compilation flag –servlet generates a Servlet which can be deployed in a servlet engin like Tomcat. This is experimental, but it seesm to work for both Scheme source and XQuery source.

The interface gnu.lists.CharSequence was renamed to avoid conflitcs with the (similar) interface java.lang.CharSequence in JDK 1.4beta.

New –help option (contributed by Andreas Schlapbach).

Changed the code generation used when –full-tailcalls. It now is closer to that used by default, in that we don't generate a class for each non-inlined procedure. In both cases calling an unknown procedure involves executing a switch statement to select a method. In addition to generating fewer classes and simplifying one of the more fragile parts of Kawa, it is also a step towards how full continuations will be implemented.

Changed the convention for name "mangling" - i.e. how Scheme names are mapped into Java names. Now, if a Scheme name is a valid Java name it is used as is; otherwise a reversible mangling using "$" characters is used. Thus the Scheme names `'<` and `'$Leq` are both mapped into the same Java name `"$Leq"`. However, other names not containing `"$"` should no longer clash, including pairs like `"char-letter?"` and `"charLetter?"` and `"isCharLetter"` which used to be all mapped to `"isCharLetter"`. Now only names containing `"$"` can be ambiguous.

If the compiler can determine that all the operands of (+ ...) or (- ...) are floating-point, then it will generate optimized code using Java primitive arithmetic.

Guile-style keyword syntax '#:KEYWORD' is recognized. (Note this conflicts with Common Lisp syntax for uninterned symbols.)

New syntax forms define-class and define-simple-class allow you to define classes more easily. define-class supports true multiple inheritance and first class class values, where each Scheme class is compiled to a pair of an inteface and a class. define-simple-class generates more efficient and Java-compatible classes.

Changes from Kawa 1.6.94 to 1.6.95.

A new language "xquery" implements a (so far small subset of) XQuery, the draft XML Query languaage.

Various internal (Java API) changes: Changes to gnu.expr.Interpreter to make it easier to add non-Lisp-like languages; gnu.lists.Consumer now has an endAttribute method that need to be called after each attribute, rather than endAttributes that was called after all of them.

If configured with –with-gcj, Kawa builds and intalls a 'gckawa' script to simlify linking with needed libraries.

The `setter` function is now inlined, and `(set! (field X 'N) V)` and `(set! (static-field <T> "N) V)` are now inlined.

If configured `--with-gcj`, then a `gckawa` helper script is installed, to make it easier to link Kawa+gcj-compiled applications.

Changes from Kawa 1.6.92 to 1.6.94

The JEmacs code now depends on CommonLisp, rather than vice versa, which means Commonlisp no longer depends on Swing, and can be built with GCJ. CommonLisp and JEmacs symbols are now implemented using Binding, not String.

Changes from Kawa 1.6.90 to 1.6.92

Kawa now installs as a .jar file (kawa.jar symlinked to kawa-VERSION.jar), rather than a collection of .class files.

The Kawa manual includes instructions for how to build Kawa using GCJ, and how to compile Scheme code to a native executable using GCJ.

Kawa now has builtin pretty-printer support, using an algorithm from Steel Bank Common Lisp converted from Lisp to Java. The high-level Common Lisp pretty-printing features are mostly not yet implemented, but the low-level support is there. The standard output and error ports default to pretty-printing.

A new formatting framework uses the Consumer interface from gnu.lists. You can associate a format with an output port. Common Lisp and JEmacs finally print using their respective syntaxes.

All output ports (OutPort instances) are now automatically flushed on program exit, using a new WriterManager helper class.

The new commmand-line option –debug-print-expr causes the Expression for each expression to be printed. The option –debug-print-final-expr is similar, but prints Expressions after optimization and just before compilation. They are printed using the new pretty-printer.

Changed calling convention for –full-tailcalls to write results to a Consumer, usually a TreeList or something to be printed. A top-level ModuleBody now uses the same CpsProcedure convention. This is useful for generating xml or html.

New libtool support allows kawa to be built as a shared library.

The new configure flag –with-gcj uses gcj to compile Kawa to both .class files and native code. This is experimental.

Changes from Kawa 1.6.70 to 1.6.90

The reader (for Scheme and Lisp) has been re-written to be table-driven, based on the design of Common Lisp readtables.

The new gnu.lists package has new implementations of sequence-related classes. It replaces most of gnu.kawa.util. See the package.html file.

If the expected type of a non-unary `+` or `-` is `<int>` or `<long>` and the operands are integeral types, then the operands will converted to the primitive integer type and the addition or subtraction done using primitive arithmetic. Similarly if the expected type is `<float>` or `<long>` and the operands have appropriate type. This optimization an make a big performance difference. (We still need to also optimize compare operations like `(< x y)` to really benefit from `<int>` declarations of loop variables.)

The implementation of procedure closures has been changed to basically be the same as top-level procedures (except when –full-tailcalls is specified): Each procedure is now an instance of a ModuleMethod, which each "frame" is an instance of ModuleBody, just like for top-level functions. This sometimes reduces the number of classes generated, but more importantly it simplifies the implementation.

A new `gnu.xml` (`api/gnu/xml/package-summary.html`) package contains XML-related code, currently an XML parser and printer, plus some XPath support. The class `gnu.lists.TreeList` (`api/gnu/lists/TreeList.html`) (alias `<document>`) is useful for compactly representing nested structures, including XML documents. If you `(require 'xml)` you will get Scheme interfaces (`print-as-xml` and `parse-xml-from-url`) to these classes.

New package gnu.kawa.functions, for primitive functions (written in Java).

The map and for-each procedure is now inlined. This is most especially beneficial when it allows the mapped-over procedure to also be inlined, such as when that procedure is a lambda expression.

Added documentation on compiling with Jikes. Renamed some classes to avoid warning when compiling with Jikes.

The reverse! procedure was added.

Internal changes: * If a variable reference is unknown, create a Declaration instance with the IS_UNKNOWN flag to represent an imported binding. * The ExpWalker framework for "tree walking" Expressions had a bit of reorganization. * New package gnu.kawa.functions, for primitive functions (written in Java).

Added a hook for constant-folding and other optimization/inlining at traversal (ExpWalker) time. Optimization of `+` and `-` procedures to use primitive Java operations when the operands are primitive types.

Implementation of SRFI-17. Change the definitions of (set! (f x ...) val) to ((setter f) x ... val), rather then the old ((setter f) val x ...). You can now associate a setter with a procedure, either using make-procedure or set-procedure-property!. Also, (setter f) is now inlined, when possible.

Internally, Syntax (and hence Macro) no longer extend Declaration.

Various Java-level changes, which may be reflected in Scheme later: * gnu.kawa.util.Consumer interface is similar to ObjectOutput and SAX's ContentHandler interfaces. * A gnu.expr.ConsumerTarget is used when evaluating to an implicit Consumer.

* These interfaces will make it easy to write functional-style but efficient code for transforming data streams, including XML. * gnu.kawa.util.FString is now variable-size.

Changes from Kawa 1.6.68 to 1.6.70

The bare beginnings of Common Lisp support, enabled by the –commonlisp (or –clisp) command line option. This is so far little more than a hack of the EmacsLisp support, but with lexical scoping and CL-style format.

Changes from Kawa 1.6.66 to 1.6.68

JEmacs news:

- Define emacs-version as Kawa version but with leading 0 instead of 1. For example, the current value is "0.6.68 JEmacs".
- New testsuite directory.
- Improved autoload framework. Handle ELisp autoload comments.
- Handle escape and meta-key.
- Handle lot more of ELisp.
- Lots more is now done in ELisp, using .el files imported from XEmacs.
- Incomplete support for setting mark, including using selection.
- Basic (but incomplete) implementation of (interactive spec).
- Common Lisp extensions: typep, default arguments.
- A new status.html file to note what works and what doesn't.

You can now specify in `define` and `define-private` the type of a variable. If the variable is module-level, `(define name :: <type> value)` creates a field named "`name`" having the specified type and initial value. (If type is not specified, the default is not `Object`, but rather a `Binding` that *contains* the variable's value.)

You can now define the type of a module-level variable: In (define[-private] :: type expression) New (define-constant name [:: type] expression) definition form.

A procedure can now have arbitrary properties associated with it. Use procedure-property and set-procedure-property! to get and set them.

The new procedure make-procedure creates a generic procedure that may contain one or more methods, as well as specified properties.

New declaration form define-base-unit. Both it and define-unit have been re-implemented to be module-safe. Basically '(define-unit ft 12in)' is sugar for '(define-constant ft$unit (... (* 12 in$unit)))', where ft$unit and in$unit are standard identifiers managed by the module system. Also, the output syntax for units and quantities is cleaner.

The new declaration (module-export name ...) allows control over the names exported from a module. The new declaration (module-static ...) allows control over which definitions are static and which are non-static. This makes it easier to use a module as a Java class.

Procedures names that accidentally clash with inherited method names (such as "run") are now re-named.

Simple aliases (define-aliases defining an alias for a variable name) are implemented more efficiently.

The package hierarchy is getter cleaner, with fewer cyclic dependencies: The gnu.math package no longer has any dependencies on kawa.* or gnu.*. Two classes were moved from gnu.text to other classes, avoiding another cyclic package dependency between gnu.text and gnu.mapping. The new gnu.kawa.lispexpr is for compile-time handling of Lisp-like languages.

Compliation of literals has been re-done. A class that can be used in a literal no longer needs to be declared as Compilable. Instead, you declare it as implementaing java.io.Externalizable, and make sure it has appropriate methods.

All the standard "data" types (i.e. not procedures or ports) now implement java.io.Externalizable, and can thus be serialized. If they appear in literals, they can also be compiled.

Created a new class gnu.kawa.util.AbstractString, with the Scheme alias `<abstract-string>`. The old gnu.kawa.util.FString now extends AbstractString. A new class CharBuffer provides an growable buffer, with markers (automatically-adjusted positions). Many of the Scheme `<string>` procedures now work on `<abstract-string>`. The JEmacs BufferContnat class (contains the characters of a buffer) now extends CharBuffer.

Some JEmacs changes to support a "mode" concept, as well as preliminary support for inferior-process and telnet modes.

New section in manual / web page for projects using Kawa.

The record feasture (make-record-type etc) how handles "funny" type and fields names that need to be "mangled" to Java names.

Re-did implementation of define-alias. For example, you can define type-aliases:

```
(define-alias <marker> <gnu.jemacs.buffer.Marker>)
```

and then use <marker> instead of <gnu.jemacs.buffer.Marker>.

`(field array 'length)` now works.

Changes from Kawa 1.6.64 to 1.6.66

Added documentation to the manual for Homogeneous numeric vector datatypes (SRFI-4).

You can now specify characters using their Unicode value: #\u05d0 is alef.

Kawa now uses a more mnemonic name mangling Scheme. For example, a Scheme function named `<=` would get compiled to method `$Ls$Eq`.

There is now working and useful module support, thought not all features are implemented. The basic idea is that a module can be any class that has a default constructor (or all of whose fields and methods are static); the public fields and methods of such a class are its exported definitions. Compiling a Scheme file produces such a module. Doing:

```
 (require <classname>)
```

will create an anonymous instance of `<classname>` (if needed), and add all its exported definitions to the current environment. Note that if you import a class in a module you are compiling, then an instance of the module will be created at compile-time, and imported definitions are not re-imported. (For now you must compile a module, you cannot just load it.)

The define-private keyword creates a module-local definition.

New syntax to override some properties of the current module:
`(module-name <name>)` overrides the default name for a module.
`(module-extends <class>)` specifies the super-class.
`(module-implements <interface> ...)` specfies the implemented interfaces.

The syntax: (require 'keyword) is syntactic sugar for (require <classname>) where the classname is find is a "module catalog" (currently hard-wired). This provides compatibility with Slib. The Slib "features" gen-write, pretty-print, pprint-file, and printf are now available in Kawa; more will be added, depending on time and demand. See the package directory gnu/kawa/slib for what is available.

Changes from Kawa 1.6.62 to 1.6.64

A lot of improvements to JEmacs (see JEmacs.SourceForge.net).

kawa-compiled-VERSION.zip is replaced by kawa-compiled-VERSION.jar.

You can now use Kawa to generate applets, using the new –applet switch, Check the "Applet compilation" section in the manual. Generating an application using the –main flag should work again. Neither –applet nor –main has Scheme hard-wired any more.

A new macro '(this)' evaluates to the "this object" - the current instance of the current class. The current implementation is incomplete, and buggy, but it will have to do for now.

The command-line argument -f FILENAME will load the same files types as load.

When a source file is compiled, the top-level definitions (procedures, variables, and macros) are compiled to final fields on the resulting class. This are not automatically entered into the current environment; instead that is the responsibility of whoever loads the compiled class. This is a major step towards a module system for Kawa.

There is a new form define-private which is like define, except that the defined name is not exported from the current module.

A procedure that has optional arguments is now typically compiled into multiple methods. If it's a top-level procedure, these will be methods in the modules "ModuleBody" class, with the same (mangled) name. The compiler can in many cases call the appropriate method directly. Usually, each method takes a fixed number of arguments, which means we save the overhead of creating an array for the arguments.

A top-level procedure declared using the form (define (NAME ARS ...) BODY ..) is assumed to be "constant" if it isn't assigned to in the current compilation unit. A call in the same compilation unit will now be implemented as a direct method call. This is not done if the prcedure is declared with the form: (define NAME (lambda (ARGS ,,,) BODY ...)

gnu.expr.Declaration no longer inherits from gnu.bytecode.Variable.

A gnu.mapping.Environment now resolves hash index collisions using "double hashing" and "open addressing" instead of "chaining" through Binding. This allows a Binding to appear in multiple Environments.

The classes Sequence, Pair, PairWithPosition, FString, and Char were moved from kawa.lang to the new package gnu.kawa.util. It seems that these classes (except perhaps Char) belong together. The classes List and Vector were also moved, and at the same time renamed to LList and FVector, respectively, to avoid clashed with classes in java.util.

New data types and procedures for "uniform vectors" of primitive types were implemented. These follow the SRFI-4 specification, which you can find at http://srfi.schemers.org/srfi-4/srfi-4.html .

You can now use the syntax `name :: type` to specify the type of a parameter. For example:
`(define (vector-length x :: <vector>) (invoke x 'length))`
The following also works:
`(define (vector-length (x :: <vector>)) ...)`.

`(define-member-alias name object [fname])` is new syntactic sugar for `(define-alias name (field object fname))`, where the default for `fname` is the mangling of `name`.

Changes from Kawa 1.6.60 to 1.6.62

The new function 'invoke' allows you to call a Java method. All of 'invoke', 'invoke-static' and 'make' now select the bets method. They are also inlined at compile time in many cases. Specifically, if there is a method known to be definitely applicable, based on compile-time types of the argument expressions, the compiler will choose the most specific such method.

The functions slot-ref, slot-set!, field, and static-field are now inlined by the compiler when it can.

Added open-input-string, open-output-string, get-output-string from SRFI-6. See http://srfi.schemers.org/srfi-6/srfi-6.html.

The manual has a new section "Mapping Scheme names to Java names", and a new chapter "Types". The chapters "Extensions", "Objects and Classes", and "Low-level functions" have been extensivley re-organized.

The Kawa license has been simplified. There used to be two licenses: One for the packages gnu.*, and one for the packages kawa.*. There latter has been replaced by the former. The "License" section of the manual was also improved.

Changes from Kawa 1.6.59 to 1.6.60

There is a new package gnu.kawa.reflect. Some classes that used to be in kawa.lang or kawa.standard are now there.

The procedures slot-ref and slot-set! are now available. They are equivalent to the existing 'field', but reading a field 'x' will look for 'getX' method if there is no public 'x' field; writing to a field will look for 'setX'.

The procedure 'make' makes it convenient to create new objects.

There is now a teaser screen snapshot of "JEmacs" at http://www.bothner.com/~per/papers/jemacs.png.

The html version of the manual now has a primitive index. The manual has been slightly re-organized, with a new "Classes and Objects" chapter.

The new functions invoke-static and class-methods allow you to call an arbitary Java method. They both take a class specification and a method name. The result of class-methods is a generic procedure consisting of those methods whose names match. (Instance methods are also matched; they are treated the asme as class methods with an extra initial argument.) The invoke-static function also takes extra arguments, and actually calls the "best"-matching method. An example:

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

Many fewer classes are now generated when compiling a Scheme file. It used to be that each top-level procedure got compiled to its own class; that is no longer the case. The change should lead to faster startup and less resource use, but procedure application will probably be noticably slower (though not so much slower as when reflection is used). The reason for the slowdown is that we in the general case now do an extra method call, plus a not-yet-optimized switch statement. This change is part of the new Kawa module system. That will allow the compiler to substitute direct methods calls in more cases, which I hope will more than make up for the slowdown.

A Scheme procedure is now in general compiled to a Java method whose name is a "mangling" of the Scheme procedure's name. If the procedure takes a variable number of parameters, then "$V" is added to the name; this indicates that the last argument is a Java array containing the rest of the arguments. Conversely, calling a Java method whose name ends in "$V" passes any excess arguments in the last argument, which must be an array type.

Many changes to the "Emacs-emulation" library in gnu.jemacs.buffer: * Implemented commands to read and save files. * We ask for file and buffer names using a dialog pop-up window. * Split windows correctly, so that the windows that are not split keep their sizes, the windows being split gets split as specified, and the frame does not change size. Now also handles horizonal splits. * Fairly good support for buffer-local keymaps and Emacs-style keymap search order. A new class BufferKeymap manages the active keymaps of a buffer. Multi-key key-sequences are handled. Pending prefix keys are remembered on a per-buffer basis (whereas Emacs does it globally).

There is now some low-level support for generic procedures.

The R5RS primitives let-syntax and letrec-syntax for defining local syntax extensions (macros) should now work. Also define-syntax works as an internal definition. All of these should now be properly "hygienic". (There is one known exception: symbols listed among the literals lists are matched as raw symbols, rather that checking that the symbol has the same binding, if any, as at the defining site.) The plan is to support general functions as hygienic rewriters, as in the Chez Scheme "syntax-case" system; as one part of that plan, the syntax-case primitive is available, but so far without any of the supporting machinary to support hygiene.

The read-line procedure was added. This allows you to efficiently read a line from an input port. The interface is the same as scsh and Guile.

Changes from Kawa 1.6.58 to 1.6.59

define-alias now works both top-level and inside a function.

Optimized eqv? so if one of the arguments is constant and not Char or Numeric, inline it the same way eq? is. (This helps case when the labels are symbols, which help the "lattice" benchmark.) ???

The Emacs-related packages are now grouped under a new gnu.jemacs package.

Improved framework for catching errors. This means improved error messages when passing a parameter of the wrong type. Many standard procedures have been improved.

Simplified, documented, and tested (!) procedure for building Kawa from source under Windows (95/98/NT).

New macros trace and untrace for tracing procedures. After executing (trace PROCEDURE), debugging output will be written (to the standard error port) every time PROCEDURE is called, with the parameters and return value. Use (untrace PROCEDURE) to turn tracing off.

New utility functions (system-tmpdir) and (make-temporary-file [format]).

A new (unfinished) framework supports multiple languages. The command-line option –elisp selects Emacs Lisp, while –scheme (the default) selects Scheme. (The only difference so far is the reader syntax; that will change.)

The 'format' function now provides fairly complete functionality for CommonLisp-style formatting. (See the Comon Lisp hyperspec at http://www.harlequin.com/education/books/HyperSpec/Body/ 3.html.) The floating point formatters (~F, ~E, ~G, ~$) now pass the formatst.scm test (from Slib, but with some "fixes"; in the testsuite directory). Also, output ports now track column numbers, so `~T` and `~&` also work correctly.

A new package gnu.emacs provides various building blocks for building an Emacs-like text editor. These classes are only compiled when Kawa is configured with the new –with-swing configuration option. This is a large initial step towards "JEmacs" - an Emacs re-implemented to use Kawa, Java, and Swing, but with full support (using gnu.elisp) for traditional Emacs Lisp. For more imformation see gnu/emacs/overview.html.

A new configuration option –with-swing can be used if Swing is available. It is currently only used in gnu.emacs, but that may change.

Changes from Kawa 1.6.56 to 1.6.58

Kawa is now "properly tail-recursive" if you invoke it with the –full-tail-calls flag. (Exception: the eval procedure does not perform proper tail calls, in violation of R5RS. This will be fixed in a future release.) Code compiled when –full-tail-calls is in effect is also properly tail-recursive. Procedures compiled with –full-tail-calls can call procedures compiled without it, and vice versa (but of course without doing proper tail calls). The default is still –no-full-tail-calls, partly because of performance concerns, partly because that provides better compatibility with Java conventions and tools.

The keywords let (including named let), let*, and letrec support type specifiers for the declared variables For example:

```
(let ((lst :: <list> (foo x))) (reverse lst))
```

Square brackets [...] are allowed as a synonym of parentheses (...).

Changes from Kawa 1.6.55 to 1.6.57

A new command-line flag –server PORT specifies that Kawa should run as a telnet server on the specified PORT, creating a new read-eval-print loop for each connection. This allows you to connect using any telnet client program to a remote "Kawa server".

A new front-end program, written in C, that provides editing of input lines, using the GNU readline library. This is a friendlier interface than the plain "java kawa.repl". However, because kawa.c needs readline and suitable networking library support, it is not built by default, but only when you configure Kawa with the –enable-kawa-frontend flag.

The way Scheme names are mapped ("mangled") into Java identifiers is now more natural. E.g. "foo-bar?" now is mapped to "isFooBar".

New syntax (object (SUPERS ...) FIELD-AND-METHODS ...) for creating a new object instance of an anonymous class. Now fairly powerful.

New procedures field and static-field for more convenient field access.

Syntactic sugar: `(lambda args <type> body) -> (lambda args (as <type> body))`. This is especially useful for declaring methods in classes.

A new synchonized form allows you to synchronize on an arbitrary Java object, and execute some forms while having an exclusive lock on the object. (The syntax matches that used by Skij.)

Changes from Kawa 1.6.53 to 1.6.55

New –debug-dump-zip option writes out a .zip file for compilation. (Useful for debugging Kawa.)

You can now declare parameter types.

Lot of work on more efficient procedure representation and calling convention: Inlining, directly callable statics method, plus some procedures no longer generate a separate Class.

Local functions that are only called from one locations, except for tail-recursion, are now inlined. This inlines do loops, and most "named let" loops.

New representation of closures (closures with captured local variables). We no longer use an array for the closure. Instead we store the captured variables in the Procedure itself. This should be faster (since we can use field accesses rather than array indexing, which requires bounds checking), and avoids a separate environment object.

If the compiler sees a function call whose (non-lexically-bound) name matches an existing (globally-defined) procedure, and that procedure instance has a static method named either "apply" or the mangled procedure name, them the compiler emits a direct call to that method. This can make a very noticable speed difference, though it may violate strict Scheme sementics, and some code may break.

Partial support for first-class "location" variables.

Changes from Kawa 1.6.53 to 1.6.54

Created new packages gnu.mapping and gnu.expr. Many classes were moved from kawa.lang to the new packages. (This is part of the long-term process of splitting Kawa into more manageable chunks, separating the Scheme-specific code from the language-independent code, and moving classes under the gnu hierarchy.)

You can now write keywords with the colon first (e.g. :KEYWORD), which has exactly the same effect and meaning as putting the colon last (e.g. KEYWORD:). The latter is preferred is being more consistent with normal English use of punctuation, but the former is allowed for compatibility with soem other Scheme implementations and Common Lisp.

Changes from Kawa 1.6.52 to 1.6.53

The new package gnu.text contains facilities for reading, formatting, and manipulating text. Some classes in kawa.lang where moved to there.

Added string-upcase!, string-downcase!, string-capitalize!, string-upcase, string-downcase, and string-capitalize; compatible with Slib.

Character constants can now use octal notation (as in Guile). Writing a character uses octal format when that seems best.

A format function, similar to that in Common Lisp (and Slib) has been added.

The default parameter of a #!optional or #!key parameter can now be #!null.

Changes since Kawa 1.6.51

The "record" feature has been changed to that a "record-type descriptor" is now a gnu.bytecode.ClassType (a `<record-type>`), rather than a java.lang.Class. Thus make-record-type now returns a `<record-typee>`, not a Class, and `record-type-descriptor` takes a `<record-typee>`, not a Class.

More robust Eval interfaces.

New Lexer abstract class. New ScmRead class (which extends Lexer) now contains the Scheme reader (moved from Inport). Now read errors are kept in queue, and can be recovered from.

Comparing an exact rational and an inexact real (double) is now done as if by first converting the double to exact, to satisfy R5RS.

Changes since Kawa 1.6.1

The compile virtual method in Expression now takes a Target object, representing the "destination". The special ConditionalTarget is used to evaluate the test of an 'if expression. This allows us to generate much better code for and, or, eq?, not and nested if inside an if.

Added port-line, port-column, and set-port-line! to match Guile.

The Makefiles have been written so all out-of-date .java (or .scm). files in a directory are compiled using a single invocation of javac (or kawa). Building Kawa should now be much faster. (But note that this depends on unreleased recent autoamke changes.)

How the Kawa version number is compiled into Kawa was changed to make it easier for people who want to build from source on non-Unix-like systems.

A new gnu.ecmascript package contains an extremely incomplete implementation of ECMSScript, the ECMA standardized version of JavaScript. It includes an ECMAScript lexer (basically complete), parser (the framework is there but most of the language is missing), incomplete expression evaluation, and a read-eval-print-loop (for testing only).

Changes in Kawa 1.6.1

Improved Kawa home page with extra links, pointer to Java-generated api docs, and homepages for gnu.math and gnu.bytecode.

Implemented system, make-process, and some related procedures.

Added macros for primitive access to object fields, static fields, and Java arrays. Added constant-fold syntax, and used it for the other macros.

The –main flag compiles Scheme code to an application (containing a main method), which can be be invoked directly by a Java interpreter.

Implemented –version (following GNU standards) as kawa.repl command-line flag.

Changes since Kawa 1.5.93

Adding make procedure to create new objects/records.

Extended (set! (f . args) value) to be equivalent to ((setter f) value . args). Implemented setter, as well as (setter car) and (setter cdr).

Can now get and set a record field value using an application: (rec 'fname) gets the value of the field named fname in record rec. (set! (rec 'fname) value) sets the value of the field named fname in rec.

A partial re-write of the implementation of input ports and the Scheme reader, to fix some problems, add some features, and improve performance.

Compiled .class files are now installed in $(datadir)/java, rather than $(prefix)/java. By default, that means they are installed in /usr/local/shared/java, rather than /usr/local/java.

There is now internal infrastructure to support inlining of procedures, and general procedure-specific optimized code generation.

There is better testing that the right number of arguments are passed to a procedure, and better error messages when you don't. If the procedure is inlined, you get a compile-time error message.

The functions created by primitive-constructor, primitive-virtual-method, primitive-static-method, and primitive-interface-method are now first-class procedure values. They use the Java reflection facily, except when the compiler can directly inline them (in which case it generates the same efficient bytecodes as before).

New functions instance? (tests type membership) and as (converts).

The kawa.html is now split into several files, one per chapter. The table of contents is now kawa_toc.html.

The syntactic form try-catch provides low-level exception handler support. It is basically the same as Java's try/catch form, but in Scheme syntax. The new procedure primitive-throw throws an exception object.

The higher-level catch and throw procedures implement exception handling where the handler is specified with a "key" (a symbol). These functions were taken from Guile.

The error function has been generalized to take multiple arguments (as in Guile). It is now a wrapper around (throw 'misc-error ...).

There is a new "friendly" GUI access to the Kawa command-line. If you invoke kawa.repl with the -w flag, a new interaction window is created. This is uses the AWT TextArea class. You can create multiple "consoles". They can either share top-level enevironments, or have separate environments. This window interface has some nice features, including editing. Added a scheme-window procedure, which is another way to create a window.

Changes since Kawa 1.5

The default prompt now shows continuations lines differently.

The copy-file function was added.

The variable port-char-encoding controls how external files are converted to/from internal Unicode characters. It also controls whether CR and CR-LF are converted to LF.

The reader by default no longer down-cases letters in symbols. A new variable symbol-read-case control how case is handled: 'P (the default) preserves case; 'U upper-cases letters; 'D or -" down-cases letters; and 'I inverts case.

The gnu.bytecode package now supports exception handlers. The new syntactic form try-finally supports a cleanup hook that is run after some other code finishes (normally or abnormally). Try-finally is used to implement dynamic-wind and fluid-let.

The environment handling has been improved to support thread-specific environments, a thread-safe fluid-let, and multiple top-levels. (The latter still needs a bit of work.)

The gnu.bytecode package has been extensively changed. There are new classes representing the various standard Attributes, and data associated with an attribute is now stored there.

Added new procedures environment-bound? and scheme-implementation-version.

Scheme symbols are represented as java.lang.String objects. Interned symbols are interned Strings; uninterned symbols are uninterned Strings. Note that Java strings literals are automatically interned in JDK 1.1. This change makes symbols slightly more efficient, and moves Kawa closer to Java.

Ports now use the JDK 1.1 character-based Reader and Writer classes, rather than the byte-oriented InputStream and OutputStream classes. This supports different reading and writing different character encodings [in theory - there is no support yet for other than Ascii or binary files].

An interactive input port now has a prompt function associated with it. It is settable with set-input-port-prompter!. The prompt function takes one argument (the input port), and returns a prompt string. There are also user functions for inquiring about the current line and column number of an input port.

The R4RS procedures transcript-on and transcript-off are implemented.

Standard types can be referred to using syntax similar to RScheme. For example Scheme strings now have the type `<string>` which is preferred to `"kawa.lang.FString"` (which in addition to being longer, is also more suspectible to changes in internal implementation). Though these types are first-class values, this is so far mainly useful for invoking primitive methods.

Changes from Kawa 1.4 to 1.5

Execute a ~/.kawarc.scm file on startup, if it exists.

Add a number of functions for testing, renaming, and deleting files. These are meant to be compatible with scsh, Guile, and MIT Scheme: file-exists?, file-directory?, file-readable?, file-writable?, delete-file, rename-file, create-diretory, and the variable home-directory.

Fixed some small bugs, mainly in gnu.math and in load.

Generalize apply to accept an arbitrary Sequence, or a primitive Java array.

Changes from Kawa 1.2 to 1.4

The codegen package has been renamed gnu.bytecode. The kawa.math package has been moved to gnu.math. Both packages have new license: No restrictions if you use an unmodified release, but GNU General Public License. Let me know if that causes problems. The rest of Kawa still has the old license.

Implement defmacro and gentemp.

Implement make-record-type and related functions to create and use new record types. A record type is implemented as a java.lang.Class object, and this feature depends on the new reflection features of JDK 1.1.

Implement keywords, and extend lambda parameter lists to support #!optional #!rest and #!keyword parameters (following DSSSL).

Added more primitives to call arbitrary interface and constructor methods.

Changes from Kawa 1.0 to 1.2

Added primitives to make it easy to call arbitrary Java methods from Scheme.

Exact rational arithetic is now fully implemented. All integer functions now believed to correctly handle bignums. Logical operations on exact integers have been implemented. These include all the logical functions from Guile.

Complex numbers are implemented (except {,a}{sin,cos,tan}). Quantities (with units) are implemented (as in DSSSL).

Eval is available, as specified for R5RS. Also implemented are scheme-report-environment, null-environment, and interaction-environment.

Internal define is implemented.

Rough support for multiple threads is implemented.

Moved kawa class to kawa/repl. Merged in kawac (compiler) functionality. A 'kawa' shell-script is now created. This is now the preferred interface to both the interactive evaluator and the compiler (on Unix-like systems).

Now builds "without a snag" using Cafe 1.51 under Win95. (Symantec JIT (ver 2.00b19) requires disabling JIT - `JAVA_COMPCMD=disable`.) Compiles under JDK 1.1 beta (with some warnings).

A testsuite (and testing framework) was added.

Documentation moved to doc directory. There is now an internals overview, in doc/kawa-tour.ps.

Changes since 0.4

The numeric classes have been re-written. There is partial support for bignums (infinite-precision integers), but divide (for example) has not been implemented yet. The representation of bignums uses 2's complement, where the "big digits" are laid out so as to be compatible with the mpn functions of the GNU Multi-Precision library (gmp). (The intent is that a future version of Kawa will support an option to use gmp native functions for speed.)

The kawa application takes a number of useful command-line switches.

Basically all of R4RS has been implemented. All the essential forms and functions are implemented. Almost all of the optional forms are implemented. The exceptions are transcript-on, transcript-off, and the functions for complex numbers, and fractions (exact non-integer rationals).

Loading a source file with load now wraps the entire file in a lambda (named "atFileLevel"). This is for better error reporting, and consistency with compile-file.

Changes since 0.3

The hygienic macros described in the appendix to R4RS are now impemented (but only the define-syntax form). They are used to implement the standard "do" form.

The R5RS multiple value functions `values` and `call-with-values` are implemented.

Macros (and primitive syntax) can now be autoloaded as well as procedures.

New kawac application compiles to one or more .class files.

Compile time errors include line numbers. Uncaught exceptions cause a stack trace that includes .scm line numbers. This makes it more practical to debug Kawa with a Java debugger.

Quasiquotation is implemented.

Various minor bug fixes and optimizations.

Changes since 0.2

The biggest single change is that Scheme procedures are now compiled to Java bytecodes. This is mainly for efficiency, but it also allows us to do tail-recursion-elimination in some cases.

The "codegen" library is included. This is a toolkit that handles most of the details needed to generate Java bytecode (.class) files.

The internal structure of Kawa has been extensively re-written, especially how syntax transforms, eval, and apply are done, largely due to the needs for compilation.

Almost all the R4RS procedures are now implemented, except that there are still large gaps in Section 6.5 "Numbers".

2 Features

Runs on the Java platform, with no native code needed.

Extends the Scheme language (`http://en.wikipedia.org/wiki/Scheme_%28programming_language%29`), following the R7RS (`http://r7rs.org/`) specification from 2013. Scheme has many implementations, and is much used in research and teaching.

Programs run fast (`http://per.bothner.com/blog/2010/Kawa-in-shootout/`) - roughly as fast as Java programs, and much faster than other “scripting languages”. This is due to a sophisticated compiler, compile-time transformations, type inference, and optional type declarations.

Full, convenient, and efficient access to the huge set of Java libraries means you can access objects, methods, fields, and classes without run-time overhead.

Start-up times are fast. You don’t have to wait for a lot of initialization. Even if you start with source code, the parser and compiler are fast.

Section 6.2 [Scripts], page 95, are simple Kawa source files that can run as an application or command. These are simple to write, start, and run efficiently, since they’re automatically compiled before execution.

Alternatively, you can embed Kawa as a Section 19.16 [Evaluating Scheme expressions from Java], page 334.

Deployment is easy and flexible. You just need the Kawa jar file.

Section 7.10 [Macros], page 120, and Section 7.11 [Named quasi-literals], page 133, make it easy to extend the syntax and implement Domain-Specific Languages.

Kawa provides the usual Section 6.3 [REPL Console], page 97, as well as batch modes.

Kawa has builtin Section 17.7 [Pretty-printing], page 292, support, and fancy formatting.

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

You can Section 19.10 [Allocating objects], page 324, with a compact "builder" syntax. It works out-of-the-box (with no run-time overhead) on many classes and APIs, but can be customized if need be.

A library for functional Section 21.1 [Composable pictures], page 356, lets you create "picture" objects, display them, transform them, combine them, convert to SVG or images, and more. This can be "printed" directly in the Kawa console (either the DomTerm console or the Swing one).

Section 21.2 [Building JavaFX applications], page 368, is simpler.

You can Section 21.3 [Building for Android], page 369, and there is special handling to make Section 21.4 [Android view construction], page 372, easier.

Flexible shell-like functionality, including [process literals], page 376.

Section 20.5 [Server-side scripts], page 342, are easy to write and install with Section 20.6 [Self-configuring page scripts], page 343, optionally using Section 20.7 [Servlets], page 346, and Section 20.4 [XML literals], page 339.

Section 14.8 [Arrays], page 247, and sequences have a lot of flexibility: Arrays can be multi-dimensional; you can use an array as an index (which generalizes slices and permutations); you can define a lazy array using a function that maps indexes to values; you can re-map the indexes to yield a transformed array.

Many useful features for mathematics and numerics:

- The full "numeric tower" includes infinite-precision rational numbers and complex numbers.
- Compile-time optimization of arithmetic with the use of type declarations and inference.
- A Section 12.5 [Quantities], page 195, is a real number with a unit, such as `3cm`.
- Section 12.4 [Quaternions], page 187, are a 4-dimensional generalization of complex numbers. Unsigned primitive integer types (`ubyte`, `ushort`, `uint`, `ulong`) are implemented efficiently without object allocation.

A Section 8.6 [Lazy evaluation], page 144, wraps an expression which is evaluated only when it is needed.

Kawa provides a Chapter 23 [Framework], page 387, for implementing other programming languages, and comes with incomplete support for CommonLisp, Emacs Lisp, and EcmaScript, and XQuery (`http://www.gnu.org/software/qexo/`).

2.1 Implemented SRFIs

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

- SRFI 0 (`http://srfi.schemers.org/srfi-0/srfi-0.html`): Feature-based conditional expansion construct, using `cond-expand` - see Section 7.9 [Syntax and conditional compilation], page 118.
- SRFI 1 (`http://srfi.schemers.org/srfi-1/srfi-1.html`): List Library, if `(require 'list-lib)` - see [SRFI-1], page 236.
- SRFI 2 (`http://srfi.schemers.org/srfi-2/srfi-2.html`): AND-LET*: an AND with local bindings, a guarded LET* special form.
- SRFI 4 (`http://srfi.schemers.org/srfi-4/srfi-4.html`): Homogeneous numeric vector datatypes - see Section 14.4 [Uniform vectors], page 239.
- SRFI 6 (`http://srfi.schemers.org/srfi-6/srfi-6.html`): Basic String Ports - see Section 17.5 [Ports], page 277.
- SRFI 8 (`http://srfi.schemers.org/srfi-8/srfi-8.html`): `receive`: Binding to multiple values - see Section 9.2 [Multiple values], page 158.
- SRFI 9 (`http://srfi.schemers.org/srfi-9/srfi-9.html`): Defining Record Types, using `define-record-type` - see Section 19.7 [Record types], page 317.
- SRFI 10 (`http://srfi.schemers.org/srfi-10/srfi-10.html`): `#,` external form for special named types. This is deprecated for various reasons, including that it conflicts with syntax-case `unsyntax`. Better to use srfi-108 Section 7.11 [Named quasi-literals], page 133.
- SRFI 11 (`http://srfi.schemers.org/srfi-11/srfi-11.html`): Syntax for receiving multiple values, using `let-values` and `let*-value` - see Section 9.2 [Multiple values], page 158.
- SRFI 13 (`http://srfi.schemers.org/srfi-13/srfi-13.html`): String Library. Needs some polishing.
- SRFI 14 (`http://srfi.schemers.org/srfi-14/srfi-14.html`): Character-set Library - see Section 13.2 [Character sets], page 204.
- SRFI 16 (`http://srfi.schemers.org/srfi-16/srfi-16.html`): Syntax for procedures of variable arity, using `case-lambda` (`http://srfi.schemers.org/srfi-16/srfi-16.html`).
- SRFI 17 (`http://srfi.schemers.org/srfi-17/srfi-17.html`): Generalized `set!` - see Section 15.1 [Locations], page 266.
- SRFI 23 (`http://srfi.schemers.org/srfi-23/srfi-23.html`): Error reporting mechanism, using `error` - see Section 8.9 [Exceptions], page 151.
- SRFI 25 (`http://srfi.schemers.org/srfi-25/srfi-25.html`): Multi-dimensional Array Primitives - see Section 14.8 [Arrays], page 247.
- SRFI 26 (`http://srfi.schemers.org/srfi-26/srfi-26.html`): Notation for Specializing Parameters without Currying - see Chapter 11 [Procedures], page 166.
- SRFI 28 (`http://srfi.schemers.org/srfi-28/srfi-28.html`): Basic Format Strings - see Section 17.6 [Format], page 287.

- SRFI 30 (`http://srfi.schemers.org/srfi-30/srfi-30.html`): Nested Multi-line Comments.
- SRFI 35 (`http://srfi.schemers.org/srfi-35/srfi-35.html`): Conditions.
- SRFI 37 (`http://srfi.schemers.org/srfi-37/srfi-37.html`): `args-fold` - a program argument processor (`http://srfi.schemers.org/srfi-37/srfi-37.html`), if `(require 'args-fold)`.
- SRFI 38 (`http://srfi.schemers.org/srfi-38/srfi-38.html`): External Representation for Data With Shared Structure. The `read-with-shared-structure` is missing, but subsumed by `read`.
- SRFI 39 (`http://srfi.schemers.org/srfi-39/srfi-39.html`): See Section 15.2 [Parameter objects], page 268.
- SRFI 41 (`http://srfi.schemers.org/srfi-41/srfi-41.html`): Streams - see Section 14.7 [Streams], page 247.
- SRFI 45 (`http://srfi.schemers.org/srfi-45/srfi-45.html`): Primitives for Expressing Iterative Lazy Algorithms - see Section 8.6 [Lazy evaluation], page 144.
- SRFI 60 (`http://srfi.schemers.org/srfi-60/srfi-60.html`): Integers as Bits. - see Section 12.6 [Logical Number Operations], page 196.
- SRFI 62 (`http://srfi.schemers.org/srfi-62/srfi-62.html`): S-expression comments.
- SRFI 64 (`http://srfi.schemers.org/srfi-64/srfi-64.html`): A Scheme API for test suites.
- SRFI 69 (`http://srfi.schemers.org/srfi-69/srfi-69.html`): Basic hash tables - see Section 14.9 [Hash tables], page 258.
- SRFI 87 (`http://srfi.schemers.org/srfi-87/srfi-87.html`): `=>` in `case` clauses.
- SRFI 88 (`http://srfi.schemers.org/srfi-88/srfi-88.html`): Keyword objects - see Section 10.3 [Keywords], page 165.
- SRFI 95 (`http://srfi.schemers.org/srfi-95/srfi-95.html`): Sorting and Merging.
- SRFI 97 (`http://srfi.schemers.org/srfi-97/srfi-97.html`): Names for SRFI Libraries.
- SRFI 98 (`http://srfi.schemers.org/srfi-98/srfi-98.html`): An interface to access environment variables
- SRFI 101 (`http://srfi.schemers.org/srfi-101/srfi-101.html`): Purely Functional Random-Access Pairs and Lists - see [SRFI-101], page 236.
- SRFI 107 (`http://srfi.schemers.org/srfi-107/`): XML reader syntax - see Section 20.4 [XML literals], page 339.
- SRFI 108 (`http://srfi.schemers.org/srfi-108/`): Named quasi-literal constructors - see Section 7.11 [Named quasi-literals], page 133.
- SRFI-109 (`http://srfi.schemers.org/srfi-109/srfi-109.html`): Extended string quasi-literals - see [string quasi-literals], page 224.
- SRFI-118 (`http://srfi.schemers.org/srfi-118/srfi-118.html`): Simple adjustable-size strings (`string-append!` and `string-replace!`).

- SRFI-140 (`http://srfi.schemers.org/srfi-140/srfi-140.html`): Immutable Strings.
- SRFI-163 (`http://srfi.schemers.org/srfi-163/srfi-163.html`): Enhanced array literals.
- SRFI-164 (`http://srfi.schemers.org/srfi-164/srfi-164.html`): Enhanced multi-dimensional Arrays

2.2 Compatibility with standards

Kawa implements all the required and optional features of R7RS, with the following exceptions.

The entire “numeric tower" is implemented. However, some transcendental functions only work on reals. Integral functions 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-tailcalls`. (Currently, the `eval` function itself is not fully tail-recursive, in violation of R5RS.) The `--full-tailcalls` 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 compatible with standard Java calling conventions and tools. Code compiled with `--full-tailcalls` can call code compiled without it and vice versa.

Even without `--full-tailcalls`, 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 includes must “obvious” cases of calls to the current function named using `define` or `letrec`, and many cases of mutual tail-recursion (including state-machines using `letrec`).

By default, symbols are case sensitive.

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. 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.

3 The Kawa Community

3.1 Reporting bugs

To report a bug or a feature request use the Issue Tracker (`https://gitlab.com/kashell/Kawa/issues`). This does require a GitLab (`https://gitlab.com/`) account; if this is a problem you can use the Savannah bug tracker.

Older Savannah bug tracker

The older bug tracker for Kawa on Savannah is still available, but we request you use the GitLab Issue Tracker (`https://gitlab.com/kashell/Kawa/issues`) for new issues.

To report a bug or feature request for Kawa (including Qexo or JEmacs) through Savannah, use the bug-submission page (`http://savannah.gnu.org/bugs/?func=additem&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`).

3.2 General Kawa email and discussion

The general Kawa email list is `kawa@sourceware.org`. 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@sourceware.org`. To unsubscribe, send a blank message to `kawa-unsubscribe@sourceware.org`. (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://sourceware.org/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`).

3.3 Acknowledgements and thanks

The author and project leader of Kawa is Per Bothner (`http://per.bothner.com/`) `per@bothner.com`.

Kawa is a re-write of Kawa 0.2, which was a Scheme interpreter written by R. Alexander Milowski `alex@milowski.com`.

Thanks to Cygnus Solutions (now part of Red Hat) for sponsoring the initial development of Kawa, and then transferring their ownership interest to Per.

Financial support

Ean Schuessler and Brainfood (`http://www.brainfood.com/`) provided financial support and encouragement.

Thanks to Chris Dean, Dean Ferreyra, and others at Merced Systems (`http://www.mercedsystems.com/`) for financial support and other contributions.

Google (`http://google.com/`) through their Summer of Code (`http://code.google.com/soc/`) project sponsored Charles Turner during Summer 2011 and 2012, Andrea Bernardini Summer 2014 and 2015, and Tom Bousso Summer 2017.

Thomas Kirk and AT&T provided financial support, and useful bug reports.

Various contributions

Jakub Jankiewicz (`http://jcubic.pl/`) contributed the Kawa logo.

Helmut Eller provided SLIME support, syntaxutils.scm, and many bug reports.

Daniel Bonniot for multiple small improvements to gnu.bytecode and gnu.expr.

Jamison Hope for multiple contributions, including quaternion support, the SRFI-14 implementation, Ant improvements, and Google Summer of Code mentoring.

Jim White for Ant support and other improvements.

Bruce R. Lewis implemented Section 20.11.3 [KRL], page 355, and made other contributions.

Geoff Berry: Handle Exceptions attribute. Other improvements.

Tom Bousso re-implemented `gnu.bytecode` to make use of ASM (`asm.ow2.org`); plus other changes,

Shad Gregory improved JEmacs.

Al Petrofsky improved gnu.math printing and added some IntNum methods.

Marco Vezzoli: SRFI-1 tailoring for Kawa.

Albert Ting - old GuiConsole code.

Christian Surlykke ported JEmacs to use SWT.

Geoff Berry for various gnu.bytecode improvements.

Ivelin Ivanov and Tom Reilly for servlet support.

Anthony Green for Fedora packaging.

Charles Turner for pretty-printer improvements, improvements in the Common Lips support, and other changes.

Andrea Bernardini optimized the implementation of `case`, and implemented full continuation support (in experimental `callcc` branch.

Julien Rousseau and Marius Kjeldahl contributed to Android support.

Peter Lane for many documentation improvements.

William D Clinger for test-cases.

Small fixes and improvements

Patrick Barta; Joseph Bowbeer; Dominique Boucher; Alexander Bunkenburg; Harold Carr; Emmanuel Castro; Álvaro Castro-Castilla; Sudarshan S Chawathe; Heather Downs; Francisco Vides Fernández; Nic Ferrier; Oliver Flasch; Weiqi Gao; Luke Gorrie; Mario Domenech Goulart; Zvi Har'E; Jeff Haynes; Ethan Herdrick; Joerg-Cyril Hoehle; Elliott Hughes; Mike Kenne; Brian Jones; Gerardo Jorvilleur; Simon Josefsson (JEmacs menu); Shiro Kawai; Thomas Kirk; Jay Krell; Timo Myyrä; Edouard Parmelan; Walter C. Pelissero; Rafael Jesus Alcantara Perez; Lynn Quam; Marcus Otto; Terje Pedersen (some XQuery functions); Matthias Radestock; Jim Rees; Ola Rinta-Koski; Andreas Schlapbach; Robert D. Skeels; Benny Tsai; Vladimir Tsichevski; Matthieu Vachon; Vasantha Ganesh; Phil Walker; Knut Wannheden; Chris Wegrzyn; Kay Zheng; Michael Zucchi.

Bug reports and test cases

Seth Alves; Khairul Azhar; Bob Bane; Hans Boehm; Adrián Medraño Calvo; Brian D. Carlstrom; Luis Casillas; Sudarshan S Chawathe; Ken Dickey (format tests); Helge Dietert; Allan Erskine; Marc Feeley (polytype.scm); Margus Freudenthal; Weiqi Gao; Andrea Girotto; Norman Hard; Gerardo Horvilleur; Yaroslav Kavenchuk; Felix S Klock II; Francois Leygues; Mirko Luedde; Leonardo Valeri Manera; Kjetil S. Matheussen; Alex Mitchell; Alex Moiseenko; Marc Nieper-Wißkirchen; Okumura Yuki; Edouard Parmelan; Walter C. Pelissero; Stephen L. Peters; François Pinard; Bill Robinson; Dan Stanger (Eaton Vance); Hallvard Traetteberg; Taylor Venable; Alessandro Vernet; Tony White John Whittaker; Robert Yokota.

Code ported from other packages

Kawa includes Free Software originally written for other purposes, but incorporated into Kawa, perhaps with some porting. A partial list:

Dorai Sitaram wrote pregexp.

The `rationalize` algorithm is by Alan Bawden and Marc Feeley.

Lars T Hansen wrote SRFI-11 (let-values, let*-values macros).

Olin Shivers wrote the SRFI-1 list-processing library, and the SRFI-13 reference impementation.

John David Stone wrote SRFI-8 (receive macro)

Jussi Piitulainen wrote the SRFI-25 specification and tests.

Richard Kelsey and Michael Sperber wrote SRFI-34.

Anthony Carrico wrote the SRFI-37 reference implementation.

Panu Kalliokoski wrote the SRFI-69 reference implementation.

Donovan Kolbly wrote the srfi-64 “meta” testsuite. Alex Shinn improved SRFI-64 portability.

Philip L. Bewig wrote the SRFI-41 (streams) specification and reference implementation.

Simon Tatham wrote listsort.

Aubrey Jaffer wrote much of SLIB, some of which has been imported into gnu.kawa.slib. He also wrote some tests we’re using.

3.4 Technical Support for Kawa

If you have a project that depends on Kawa or one of its component packages, you might do well to 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. Response for support requests received during 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.

3.5 Projects using Kawa

MIT App Inventor (`http://appinventor.mit.edu/`) for Android (formerly Google App Inventor) uses Kawa to translate its visual blocks language.

The HypeDyn (`http://www.narrativeandplay.org/hypedyn/`) hypertext fiction authoring tool is written in Kawa. HypeDyn (pronounced "hyped in") is a procedural hypertext fiction authoring tool for people who want to create text-based interactive stories that adapt to reader choice. HypeDyn is free to download and open source, and runs on Linux, MacOS and Windows. This is a research project carried out at the Department of Communications and New Media, National University of Singapore.

Nü Echo (`http://www.nuecho.com`) develops high-performance speech enabled applications. Nü Echo uses Kawa for the development of innovative speech application development tools, like a complete grammar IDE (`http://www.nuecho.com/en/services/grammar.shtml`).

Merced Systems, Inc. (`http://www.mercedsystems.com/`) uses Kawa extensively in their contact center performance management product Merced Peformance Suite. Kawa Scheme is used for all development and has allowed Merced to realize the large productivity gains that come with using Scheme while still maintaining tight integration with a large number of Java libraries.

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://jemacs.sourceforge.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 an HTML file on a web server.

The SchemeWay Project (`http://schemeway.sourceforge.net`) is a set of Eclipse (`http://www.eclipse.org`) plug-ins for professional Scheme programming. The first plugin released, SchemeScript, is a fully-featured Scheme editor customizable in Scheme. It embeds the Kawa Scheme system and has many features that ease Kawa Scheme programming (like code completion on variable names, class and method names, namespaces, etc).

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 retrieval system as a communication protocol between server and clients, and to do server agents programming.

3.6 Ideas and tasks for contributing to Kawa

Kawa (like other Free Software projects) has no lack of tasks and projects to work on. Here are some ideas.

The ones marked *(GSoC)* are probably most suitable for a Google Summer of Code project, in being a reasonable size, self-contained, and not depending on other tasks.

3.6.1 Recusively initialized data structures

(GSoC)

Kawa has convenient syntax to Section 19.10 [Allocating objects], page 324, but it gets messier it you want to initialize multiple objects that reference each other. Likewise for a single object "tree" which contains links to the root. In this example, we will looks at two vectors, but the feature is more useful for tree structures. Assume:

```
(define-constant list1 [1 2 list2])
(define-constant list2 ['a 'b list1])
```

The compiler translates this to:

```
(define-constant list1
   (let ((t (object[] length: 3))) ;; allocate native Java array
      (set! (t 0) 1)
      (set! (t 1) 2)
      (set! (t 2) list2)
      (FVector:makeConstant t)))
(define-constant list2
   (let ((t (object[] length: 3))) ;; allocate native Java array
      (set! (t 0) 'a)
      (set! (t 1) 'b)
      (set! (t 2) list1)
      (FVector:makeConstant t)))
```

The problem is that `list2` has not been created when we evaluate the initializing expression for `list`.

We can solve the problem by re-writing:

```
(define-private tmp1 (object[] length: 3))
(define-constant list1 (FVector:makeConstant tmp1)
(define-private tmp2 (object[] length: 3))
(define-constant list2 (FVector:makeConstant tmp2)
(set! (tmp1 0) 1)
(set! (tmp1 1) 2)
(set! (tmp1 2) list2)
(set! (tmp2 0) 1)
```

```
(set! (tmp2 1) 2)
(set! (tmp2 2) list1)
```

The complication is that the code for re-writing vector and object constructors is spread out (depending on the result type), and not where we deal with initializing the variables. One solution is to introduce an inlineable helper function `$build$` defined as:

```
(define ($build$ raw-value create init)
  (let ((result (create raw-value))
    (init raw-value result)
    result))
```

Then we can re-write the above code to:

```
(define-constant list1
  ($build$
    (object[] length: 3)
    (lambda (raw) (FVector:makeConstant raw))
    (lambda (raw result)
      ($init-raw-array$ raw 1 2 list2))))
(define-constant list2
  ($build$
    (object[] length: 3)
    (lambda (raw) (FVector:makeConstant raw))
    (lambda (raw result)
      ($init-raw-array$ raw 'a 'b list1))))
```

Note that the call to `$build$`, as well as the generated `lambda` expressions, are all easily inlineable.

Now assume if at the top-level *body* if there is a sequence of `define-constant` definitions initialized with calls to `$build$`. Now it is relatively easy to move all the `init` calls after all `alloc` and `create` expressions. The `$init-raw-array$` calls are expanded after the code has been re-ordered.

The project includes both implementing the above framework, as well as updating type-specific (and default) object creation to use the framework. It would also be good to have compiler warnings if accessing an uninitialized object.

3.6.2 Enhance texinfo-js documentation browser for Kawa documentation

(GSoC)

3.6.3 Run interactive process in separate Java Virtual Machine:

(GSoC)

When developing and testing it is useful for the REPL to support hot-swapping (replacing functions on-the-fly) and debugging. The main goal being able to smoothly reload changed modules (files or functions), and have other modules not break. Debugging (such as setting breakpoints) would not be a priority for this project, but could be a follow-on project. Skills: Should be experienced with Java, and interested in learning about JVM TI (`https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/index.html`) and similar low-level parts of the platform. Difficulty: Challenging, but you can study how Java-9's new jshell (`https://en.wikipedia.org/wiki/Jshell`) uses the JVM TI.

3.6.4 Better dynamic reload

(GSoC - this is related to the previous item)

Kawa does a lot of optimizations and inlining. This conflicts with being able to "reload" a module into an already-running interactive environment.

We could add an option to load a module in "reloadable" mode. Kawa already patches an old function object (a `ModuleMethod`) so existing references to the function get automatically updated. However, there are problems if the "signature" of the function changes - for example if the return type (declared or inferred) becomes more general. In those cases the best thing is to re-compile any code that depends on the modified function.

Reloading a module that defines a class is even trickier, at least if there are existing instances that should work as the updated class. We can handle the special case where only method bodies change: In reloadable mode, each method body is compiled to a separate function, the actual body indirects to the function. We must also recognize when compiling a new version of the same class, which requires a textual comparison between the old and new versions, or a structural comparison between the old class and the new code.

When it comes to top-level variables, an issue is when to re-evaluate the initializing expression. It is reasonable to do so if and only if the expression is modified, which again requires a textual comparison.

3.6.5 Easier Access to Native Libraries using JNA/JNR

(GSoC)

The traditional way to access native (C/C++) functions is using JNI, but it's very awkward. JNA and JNR (`https://github.com/jnr`) are much easier to use (`http://www.oracle.com/technetwork/java/jvmls2013nutter-2013526.pdf`). This project would design and implement an easy-to-use Kawa wrapper for for JNR. You should study existing JNR wrappers, such as that for JRuby. Difficulty: Medium. Need to study existing wrappers and "foreign function interfaces" (in multiple languages) and design one suitable for Kawa. Some Scheme (Kawa) experience would be helpful.

3.6.6 Types for units

(GSoC)

Kawa supports units (such as `cm^2` for square centimeters) and Section 12.5 [Quantities], page 195, (such as `4cm^2`). We would like to integrate these into the type system, both for performance and compile-time type checking.

For syntax we can use a pseudo-parameterized type `quantity`. For example:

```
(define a1 ::quantity[cm^2] 4cm^2)
(* 2.0 a1) ;; ⇒ 8cm^2
(+ 2.0 a1) ;; compile-time error
```

The run-time type of the variable `a1` should be a primitive `double`, without object allocation. Of course when `a1` is converted to an object, we create a `Quantity`, not a `Double`. We can build on Kawa's existing framework for non-standard primitive types such as `character` and `ulong`. Skills: Need good Java experience, and somewhat familiar with the Java Virtual Machine. You will need to become comfortable reading `javap` output. Difficulty: Modest.

3.6.7 Compiler should use class-file reading instead of reflection

The Kawa compiler currently uses reflection to determine properties (such as exported function definitions) from referenced classes. It would be better to read class files. This should not be too difficult, since the `gnu.bytecode` library abstracts over class information read by reflection or class reading.

3.6.8 Mutually dependent Java and Scheme modules

(GSoC - maybe)

We'd like a command for compiling a list of Java and Scheme source files that may have mutual dependencies. A good way to do this is to hook into `javac`, which is quite extensible and pluggable.

One could do something like:

1. Read the “header" of each Kawa source file, to determine the name of the generated main class.
2. Enter these class names into the javac tables as “uncompleted” classes.
3. Start compiling the Java files. When this requires the members of the Kawa classes, switch to the Kawa files. From javac, treat these as pre-compiled .class files. I.e. we treat the Kawa compiler as a black box that produces Symbols in the same way as reading class files. At this point we should only need the initial “scan” phase on Kawa.
4. If necessary, finish compiling remaining Kawa files.

This approach may not immediately provide as robust mixed-language support as is ideal, but it is more amenable to incremental improvement than a standalone stub-generator.

This project is good if you know or want to learn how `javac` works.

3.6.9 Use Java-7 MethodHandles and invokedynamic

Java 7 supports MethodHandles which are meant to provide better performance (ultimately) for dynamic languages. See JSR 292 (`http://jcp.org/en/jsr/detail?id=292`) and the Da Vinci Machine Project (`http://openjdk.java.net/projects/mlvm/`). Kawa makes limited use of MethodHandles, and no use of invokedynamic. There is more to be done. For example, we can start by optimizing arithmetic when the types are unknown at compile-time. They could make implementing generic functions (multimethods) more efficient. At some point we want to compile lambdas in the same way as Java 8 does. This can potentially be more efficient than Kawa's current mechanism.

Remi Forax's vmboiler (`https://github.com/forax/vmboiler`) is a small library on top of ASM that generates optimistically typed bytecodes. It could be useful for ideas.

3.6.10 Parameterized types

(GSoC)

Kawa has some limited support for parameterized types, but it's not used much. Improve type inferencing. Support definition of parameterized classes. Better use of parameterized types for sequence class. Support wildcards. (It might be better to have wild-carding be associated with declarations, as in Scala or proposed for Java (`http://openjdk.java.net/jeps/300`), rather than uses.) See also `http://openjdk.java.net/jeps/8043488`.

3.6.11 Optimized function types and values using MethodHandles

(GSoC)

Kawa doesn't have true function types: Parameter and result types are only handled for "known" functions. The general case with optional and keyword parameter is complicated, but simple fixed-arity procedure types would be very useful.

The following syntax is suggested:

```
procedure[(T1 .. Tn) Tr]
```

T1 through *T1* are types of the parameters, and *Tr* is the type of the result. For example: `procedure[(vector int) string]`. We call this a typed-procedure type (in contrast to plain `procedure`).

If a value has a typed-procedure type then its run-time representation is a just a `MethodHandle`. If such a procedure is called, the generated bytecode is to just call its `invokeExact` method. The argument expressions are converted (and type-checked) the same way as if we were calling a statically-known procedure.

Note that passing an `int` argument of to `procedure[(vector int) string]` value does *not* require allocating an object to "box" the `int`; we can pass a plain `int` as-is. Thus using typed-procedure types can lead to major speed-up. For example the `lib-test.scm` should become much faster.

Converting a known procedure to a typed-procedure type is usually just a matter of creating a `MethodHandle` that references the method implementing the procedure. Some glue code may be needed if the types aren't identical, or if the procedure is a closure.

Converting a type-procedure value `p` to generic value (such as untyped `procedure` or `object`) can be though of as wrapping it in a `lambda`:

```
((lambda (arg1::vector arg2::int)::string (p arg1 arg2))
```

Coercing a generic value or an untyped procedure to a typed-procedure would need to generate a method whose signature matches the typed-procedure type, and in the body of the method use a generic apply.

Coercing from one typed-procedure type to a different typed-procedure type is a combination of the above techniques (as if converting first to object and then to the target type), though some optimizations are worth doing.

Adding varargs support can be done later.

We need a fall-back mechanism for platforms (such as Android) that don't support `MethodHandles`. The easiest is to just treat a typed-procedure type as plain `procedure` at run-time, though we still want the compile-time type-checking,

3.6.12 Full continuations

Currently being worked on.

Add support for full continuations, which is the major feature missing for Kawa to qualify as a "true Scheme". One way to implement continuations is to add a add that converts the abstract syntax tree to continuation-passing-style, and then expand the existing full-tail-call support to manage a stack. There are other ways to solve the problem. This may benefit from [task-faster-tailcalls], page 56.

3.6.13 Faster tailcalls

Make `--full-tailcalls` run faster. This may depend on (or incorporate) [task-TreeList-optimization], page 56.

3.6.14 TreeList-optimization

The TreeList (`http://www.gnu.org/software/kawa/api/gnu/lists/TreeList.html`) class is a data structure for “flattened” trees. It is used for XML-style nodes, for multiple values, and for the full-tail-call API. The basic concept is fine, but it could do with some re-thinking to make make random-access indexing fast. Also, support for updating is insufficient. (This needs someone into designing and hacking on low-level data-structures, along with lots of profiling and testing.)

3.6.15 Asynchronous evaluation

C# recently added `asynch` and `await` keywords for asynchronous programming (`http://msdn.microsoft.com/en-us/vstudio/gg316360`). Kawa’s recently improved support for lazy programming seems like a good framework for equivalent functionality: Instead of an `asynch` method that returns a `Task<T>`, the Kawa programmer would write a function that returns a `lazy[T]`. This involves some design work, and modifying the compiler to rewrite the function body as needed.

This is related to full continuations, as the re-writing is similar.

3.6.16 REPL console and other REPL improvement

Currently being worked on.

Improvements to the read-eval-print console. In addition to a traditional Swing console, it would be useful to support using a web browser as a remote terminal, possibly using web-sockets. (This allows “printing” HTML-expressions, which can be a useful way to learn and experiment with web technologies.) See here (`http://per.bothner.com/blog/2007/ReplPane/`) for an article on the existing Swing REPL, along with some to-do items. Being able to hide and show different parts of the output might be nice. Being able to link from error messages to source might be nice. Better handling of redefinitions is discussed here in the context of JavaXF Script (`http://per.bothner.com/blog/2009/REPL-for-JavaFX/`); this is a general REPL issue, mostly independent of the GUI for it.

An interesting possibility is to use the IPython (`http://ipython.org/`) framework. There are existing ports for Scala: either IScala (`https://github.com/mattpap/IScala`) or Scala Notebook (`https://github.com/Bridgewater/scala-notebook`).

3.6.17 XQuery-3.0 functionality

(GSoC, for some subset)

It would be nice to update the XQuery (Qexo) support to some subset of XQuery 3.0 (`http://www.w3.org/TR/xquery-30/`).

3.6.18 XQuery-updates

It would be nice to support XQuery updates (`http://www.w3.org/TR/xquery-update-10/`). This depends on [task-TreeList-optimization], page 56.

3.6.19 Common Lisp support

Kawa supports a small subset of the Common Lisp language, but it supports a much larger subset of core Common Lisp concepts and data structures, some designed with Common Lisp functionality in mind. Examples include packages, arrays, expanded function declarations, type specifications, and format. A lot could be done to improve the Common Lisp support with modest effort. Some Common Lisp features could also be useful for Scheme: Documentation strings (or markup) as Java annotations, better MOP-like introspection, and generic methods a la defmethod (i.e. with multiple definition statements, possibly in separate files, as opposed to the current make-procedure) all come to mind. Being able to run some existing Common Lisp code bases with at most modest changes should be the goal. One such package to start with might be an existing test framework (`http://aperiodic.net/phil/archives/Geekery/notes-on-lisp-testing-frameworks.html`), perhaps FivaAM (`http://common-lisp.net/project/bese/FiveAM.html`). Full Common Lisp compatibility is nice, but let's walk before we can run.

3.6.20 JEmacs improvements

(GSoC, for some subset)

A lot of work is needed to make JEmacs (`http://jemacs.sourceforge.net/`) useful. One could try to import a useful package and see what works and what fails. Or one may look at basic editing primitives. Enhancements may be needed to core Emacs Lisp language primitives (enhancing [task-common-lisp], page 57, may help), or to the display engine.

Emacs now supports lexical bindings (`http://www.gnu.org/software/emacs/manual/html_node/elisp/Lexical-Binding.html`) - we should do the same.

3.6.21 Improved IDE integration

There is some Kawa support for Eclipse (Schemeway), and possibly other IDEs (NetBeans, IntelliJ). But many improvements are desirable. [task-REPL-improvements], page 56, may be a component of this.

3.6.21.1 Plugin for NetBeans IDE

Kawa-Scheme support for the NetBeans IDE would be useful. One could perhaps build on the Clojure plugin.

3.6.21.2 Plugin for Eclipse IDE

Kawa-Scheme support for the Eclipse IDE would be useful. Probably makes sense to enhance SchemeWay (`http://sourceforge.net/projects/schemeway/`). It may also make sense to build on the Dynamic Languages Toolkit (`http://www.eclipse.org/dltk/`), possibly making use of Schemeide (`http://schemeide.sourceforge.net/`), though DLTk seems more oriented towards interpreted non-JVM-based languages.

3.6.21.3 Improve Emacs integration

SLIME (`http://en.wikipedia.org/wiki/SLIME`) is an Emacs mode that provides IDE-like functionality. It supports Kawa.

JDEE (`http://jdee.sourceforge.net/`) is a Java development environment, so might have better hooks to the JVM and Java debugging architecture.

CEDET (`http://cedet.sourceforge.net/`) is a more general framework of development tools.

3.6.22 Hop-style web programming

Hop (`http://hop.inria.fr/`) is an interesting design for integrating server-side and client-side programming using a Scheme dialect. These ideas seem like they would port quite well to Kawa.

3.6.23 String localization

(GSoC)

Support localization by extending the SRFI_109 (`http://srfi.schemers.org/srfi-109/srfi-109.html`) syntax, in the manner of (and compatible with) GNU gettext (`http://www.gnu.org/software/gettext/`). I.e. optionally specify a localization key (to use as an index in the translation database); if there is no key specified, default to using the literal parts of the string.

3.6.24 Data binding

Implement a “bind” mechanism similar to that of JavaFX Script (`http://docs.oracle.com/javafx/1.3/tutorials/core/dataBinding/`). The idea is that when you initialize a variable or field, instead of initializing it to a fixed value, you bind it to an expression depending on other variables. We install “listeners” on those variables, so when those variables change, we update the bound variable. This feature is useful in many applications, but the initial focus could be GUI programming and perhaps web programming.

3.6.25 Decimal arithmetic and repeated decimals

(GSoC. Possibly a bit light for a full Summer project, but can be extended or combined with other projects.)

Exact decimal arithmetic is a variation of exact rational arithmetic, but may be more user-friendly. In particular, printing using decimals is generally nicer than fractions. It is also sometimes useful to specify an explicit scale, so we can distinguish 0.2 from 0.20. We can use the Java `BigDecimal` class, but run into problems with division - for example `(/ 1.0 3.0)`. We should implement a subclass of `RatNum` that generalizes `BigDecimal` to also handle repeating decimals. We need a lexical syntax for repeating decimals. Possible ideas: `0._81_` or `0.#81`. If a Scheme number literal is specified as exact and has either a decimal point or an exponent (for example `#e1.25`), then it should read as an exact decimal, not a fraction.

3.6.26 Optional strict typing along with an explicit `dynamic` type

(GSoC)

Kawa currently implements “optimistic” typing: The compiler only complains if an expression has no values in common with the target type - for example, if assigning a `string` expression to an `integer` variable. It would be interesting to experiment with a `--strict-typing` option (which would never be the default): Strict typing would only allow “widening” conversions - i.e. that the expression type be a subtype of the target type. For example it would complain if assigning a `number` to an `integer` unless you used an explicit cast.

To make this easier to work with we’d make use of the [dynamic-type], page 297, similar to what `C#` does (`https://msdn.microsoft.com/en-us/library/dd264736.aspx`): Any expression can be converted to or from `dynamic` without the compiler complaining. Similarly, if `x` is `dynamic` then `x:name` is allowed by the compiler regardless of `name`, with all checking being deferred to run-time. If a variable is declared without a type, it should default to `dynamic`. The `dynamic` type is represented in the VM as `object` but with an annotation (like we do with `character`).

The type-checker might need some changes to better distinguish implicit conversions from explicit casts.

4 Getting and installing Kawa

4.1 Getting Kawa

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

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 current release of the Kawa source code is `ftp://ftp.gnu.org/pub/gnu/kawa/kawa-3.1.1.tar.gz`. (To unpack `.tar.gz` files Windows users can use 7-Zip (`http://www.7-zip.org/`), which is Free Software.)

The corresponding pre-compiled release is `ftp://ftp.gnu.org/pub/gnu/kawa/kawa-3.1.1.zip`. The most recent snapshot is `ftp://ftp.gnu.org/pub/gnu/kawa/kawa-latest.zip`. Instructions for using either are Section 4.3 [Binary distribution], page 60.

4.1.1 Getting the development sources using Git

The Kawa sources are managed using a git (`https://gitlab.com/kashell/Kawa`) repository. If you want the very latest version grab a git client (`https://git-scm.com/downloads`), and then check out the source using this command:

```
git clone https://gitlab.com/kashell/Kawa.git
```

After a checkout you will need to run:

```
./autogen.sh
```

before proceding with instructions for Section 4.4 [Source distribution], page 61.

Once you have it checked out, you can keep it up-to-date with `git pull`.

You can also browse the git archive (`https://gitlab.com/kashell/Kawa/tree/master`) online.

4.2 Getting and running Java

Before installing Kawa, you will need a working Java system. The released Kawa jar file assumes Java 8 or newer. You need to build Kawa from source if you have Java 5, Java 6, or are targeting Android. (Older versions of Kawa have been reported to work with JDK from 1.1, Kaffe, Symantec Cafe, J++, and GCJ, but these are no longer supported.)

The discussion below assumes you are using the Java Developer's Kit (JDK) from Oracle. You can download free copies of JDK 8 (`http://www.oracle.com/technetwork/java/javase/downloads/index.html`) for various platforms.

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 `$JAVA_HOME`, and you are using a Bourne-shell compatible shell (/bin/sh, ksh, bash, and some others) you can set `PATH` thus:

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

4.3 Installing and using the binary distribution

The binary release comes as a `.zip` archive that includes Kawa itself (as a `.jar` file `kawa-`*`version`*`.jar`), some third-party helper libraries, `kawa` command scripts (for GNU/Linux/Unix/MacOS or Windows), and documentation (basically this manual).

After downloading (see Section 4.1 [Getting Kawa], page 59), extract the files from the `.zip` archive using a suitable `unzip` program, which will create a directory `kawa-`*`version`*, with `lib`, `bin`, and `doc` sub-directories. In the following, we assume the environment variable `KAWA_HOME` refers to this directory:

```
unzip ~/Downloads/kawa-version.zip
export KAWA_HOME=`pwd`/kawa-version
```

The binary release requires Java 8 or later. If you have an older Java implementation, or build for a mobile environment like Android, then you will need to get the source distribution.

If you want to use Kawa as part of some other application, you just need the `$KAWA_HOME/lib/kawa.jar`.

Running the `kawa` command

To run a Kawa script file or the Kawa read-eval-print-loop run the Kawa application. There are various way to do so.

The recommended way is to execute the `$KAWA_HOME/bin/kawa` Bash shell script. This should work on most Unix-like platforms that have Bash installed, including GNU/Linux, BSD, MacOS, and Cygwin/MingW. (Please report if you have problems.)

The script assumes that either a suitable `java` program is in your `PATH`; or the `JAVA` environment variable names a suitable `java` executable; or that `JAVA_HOME` is set so `$JAVA_HOME/bin/java` is suitable.

If you want to put `kawa` in your search path you can of course do:

```
PATH=$KAWA_HOME/bin:$PATH
```

Alternatively you can create a symbolic link in an already-searched directory. For example:

```
cd /usr/local/bin
ln -s $KAWA_HOME/bin/kawa kawa
```

The `bin/kawa.bat` script works on Windows.

Both scripts add some helper libraries, including support for input editing.

It is also possible to run Kawa using `java` directly:

```
java -jar $KAWA_HOME/lib/kawa.jar
```

or:

```
CLASSPATH=$KAWA_HOME/lib/kawa.jar
export CLASSPATH
java kawa.repl
```

On Windows:

```
set classpath=%KAWA_HOME%\lib\kawa.jar
```

To run Kawa in a fresh window use the -w flag:

```
kawa -w
```

or

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

Reading the documentation

The file `doc/kawa-manual.epub` contains the Kawa documention packaged as an electronic book, which is readable by most e-book readers. Plugins are also available for common browsers, for example EPUBReader (`http://www.epubread.com`) for `firefox`.

Even easier is to invoke [browse-manual-option], page 89, (or on Windows: `bin\kawa.bat --browse-manual`).

An `epub` is essentially a zip archive, which you can unzip:

```
cd $KAWA_HOME/doc
unzip kawa-manual.epub
```

Then you can use a plain browser with the URL `file:$KAWA_HOME/doc/OEBPS/index.html`.

4.4 Installing and using the source distribution

The Kawa release normally comes as a gzip-compressed tar file named '`kawa-3.1.1.tar.gz`'. 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. You can also use this method on Microsoft Windows, with the help of tools from MinGW (`http://www.MinGW.org/`) or Cygwin (`http://www.cygwin.org/`).

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 `Makefiles`, 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.

4.4.1 Build Kawa using `configure` and `make`

(See [building-on-Windows-with-make], page 64, for some notes for building on Microsoft Windows.)

If you have a `tar.gz` file, first unpack that in your build directory:

```
tar xzf kawa-3.1.1.tar.gz
cd kawa-3.1.1
```

If you're building from the Git repository, you need to generate `configure` and some other files. This is easiest done with the `autogen.sh` script:

```
./autogen.sh
```

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

```
./configure
```

The `configure` script takes a number of [configure options], page 62.

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.

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.

It has been reported that parallel make doesn't work, so don't use the `-j2` or above options.

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.

4.4.1.1 Configure options

The `configure` script takes a number of options. The `--help` switch gives you a list of options. The following are some of the more common or important ones.

`--prefix=`*`install-dir`*
`--prefix` *`install-dir`*

By default `make install` will install the compiled `.jar` files info `/usr/local/share/java`, the `kawa` command into `/usr/local/bin`, and so on in `/usr/local`. The `--prefix` option causes the files to be installed under ***install-dir*** instead of `/usr/local`. For example to install the `.jar` in `/opt/kawa/share/java` and otherwise use `/opt/kawa` do:

```
./configure --prefix=/opt/kawa
```

`--with-java-source=`*`version`*

As distributed, the Kawa source code requires Java 8. If you only have Java 7, Java 6, or Java 5, use the `--with-java-source` option:

```
./configure --with-java-source=6
```

Kawa no longer supports older verisons of Java (JDK 1.4 or older). It might be possible to use a tool like Retroweaver (`http://retroweaver.sourceforge.net/`) on the Kawa `.jar` to fix up Java 5 dependencies. Contact the Kawa author if you want to be a tester for this.

`--with-docbook-stylesheets[=`*`path`*`]`

Build the documentation (this manual) as an electronic book (in ebook format) or a website, using the DocBook xslt stylesheets. (You can build the documentation without DocBook, but using it enables nicer-looking and more functional documentation.)

The stylesheets are found using *path*; the file ***path***`/epub3/chunk.xsl` needs to exist. (For example, on Fedora 25 *path* can be `/usr/share/sgml/docbook/xsl-ns-stylesheets`, while on Debian use `/usr/share/xml/docbook/stylesheet/docbook-xsl-ns`.)

`--with-domterm`
`--with-domterm=`*`domterm_home`*

Compile with extra support for the [Using DomTerm], page 100, terminal emulator library, where ***domterm_home*** is such that ***domterm_home***`/lib/domterm.jar` exists. (Some DomTerm support is built-in regardless.)

If you use this option along with `--with-javafx` then creating a new Section 6.3 [REPL Console], page 97, window will create a DomTerm window.

As an optional convenience, you can use the `domterm.jar` in the Kawa binary distribution.

`--with-jline3`
`--with-jline3=`*`jline3.jar`*

Build support for using JLine 3 (`https://github.com/jline/jline3`), which is a library for handling console input, similar to GNU readline. If specified, the *jline3.jar* is added to the classpath of the generated `kawa.sh` or `kawa` shell program.

An advantage of `--with-jline3` (compared to `--enable-kawa-frontend`) is that the former works without native code (on most Unix-like platforms), and it does not require a C wrapper program.

As an optional convenience, you can use the `jline.jar` in the Kawa binary distribution.

`--with-domterm`
`--with-domterm=`*`domterm.jar`*

Compile with extra support for the [Using DomTerm], page 100, terminal emulator library. (Some DomTerm support is built-in regardless.)

If you use this option along with `--with-javafx` then creating a new Section 6.3 [REPL Console], page 97, window will create a DomTerm window.

As an optional convenience, you can use the `domterm.jar` in the Kawa binary distribution.

`--with-servlet`
`--with-servlet=`*`servlet-jar`*

Build support for Section 20.7 [Servlets], page 346, which are used in web servers. This requires the `servlet-api.jar` (available various places including Tomcat (`http://tomcat.apache.org/`) or Glassfish (`https://glassfish.java.net/`)), for `javax.servlet.Servlet` and related classes. If this class isn't in your classpath, specify its location as *`servlet-jar`*. For example:

```
./configure --with-servlet=/path/to/servlet-api.jar
```

`--enable-jemacs`

Build JEmacs (enable Emacs-like text editor) and support (a subset of) the Emacs Lisp language. JEmacs is a proof of concept - not really usable or maintained.

`--with-javafx`
`--with-javafx=`*`javafx-home`*

Set this flag to enable the convenience features for Section 21.2 [Building JavaFX applications], page 368. The JavaFX classes are included in JDK 8 (but not OpenJDK 8), and you don't need to specify *`javafx-home`*. JDK 11 or later does not include JavaFX, so you need to specify the location of the modular OpenJFX SDK as *`javafx-home`*.

`--with-android=`*`android-jar`*

Build for the Android platform. This requires Section 21.3 [Building for Android], page 369.

`--enable-kawa-frontend`

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/`.

Note that using JLine, enabled by `--with-jline3`, is now recommended instead of using the `readline` frontend.

You may need to specify to `make` where to find the `readline` include files (with `READLINE_INCLUDE_PATH`) and the library (with `READINE_LIB_PATH`). For example on OS/X you need to do:

```
make READLINE_INCLUDE_PATH=-I/usr/local/unix/readline/include \
     READLINE_LIB_PATH=-L/usr/local/unix/readline/lib
```

4.4.1.2 Building on Windows using MinGW

The Kawa `configure` and `make` process assumes Unix-like tools, which you can get from the MinGW project (`http://mingw.org`). Download the MingGW Installation Manager, and use it to install at least `mingw-developer-toolkit`. (Also installing `msys-groff` avoids a minor problem building the documentation.)

The `C:\MinGW\msys\1.0\msys.bat` script creates a command window with the `bash` shell and the `PATH` set up as needed. Alternatively, you can use the standard Windows command prompt if you set your `PATH` as described in here (`http://mingw.org/wiki/Getting_Started`).

4.4.1.3 Building on Windows using Cygwin

The free Cygwin (`http://sourceware.org/cygwin/`) environment can be used for building Kawa: The Kawa configure script recognizes Cygwin, and modifies the classpath to use Windows-style path separators.

Beyond the base packages, you probably want to install `autoconf`, `automake`, `git`, `texinfo`, `groff`, `make`, and `diffutils`.

Cygwin (unlike MinGW) has a current version of `makeinfo`, but an undiagnosed bug still prevents building `kawa.info`. You can work around that problem with `touch doc/kawa.info`.

4.4.2 Building the documentation

4.4.2.1 Plain HTML documentation

You can build a plain HTML version of the documentation (using `makeinfo` from the `texinfo` distribution):

```
cd doc && make kawa-html/index.html
```

In this case, point your browser at `file:/`*`kawa_srcdir`*`/doc/kawa-html/index.html`.

4.4.2.2 Fancier HTML documentation

To build the documentation in a nicer form suitable for a web-site you need `makeinfo` *and* the DocBook XSLT tools (and to have run `configure` with the `--with-docbook-stylesheets` option):

```
cd doc && make web/index.html
```

You can then point your browser at `file:/`*`kawa_srcdir`*`/doc/web/index.html`.

4.4.2.3 Using ebook readers or the –browse-manual option

To build an `EPUB` file suitable for ebook readers, as well as enabling support for the [browse-manual-option], page 89, do:

```
cd doc && make kawa-manual.epub
```

This also requires the DocBook XSLT tools.

4.4.2.4 Building a printable PDF file

To build a `pdf` file suitable for printing or online viewing do:

```
cd doc && make kawa.pdf
```

The resulting `kawa.pdf` is somewhat unsatisfactory - when viewed online, links aren't clickable. Furthermore, box drawing characters are missing.

4.4.3 Build Kawa using `ant`

Kawa now includes an Ant buildfile (`build.xml`). Ant (`http://ant.apache.org`) 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 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-3.1.1.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.

There are various "properties" that control what `ant` does. You can override these on the command line or by editing the `build.properties` file in the same directory as `build.xml`. For example, the `build.dir` property 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).

5 Kawa Scheme Tutorial

This is obviously incomplete, but it may be useful, especially if you're starting with Kawa from scratch. If you're new to Scheme you might also check out one of these tutorials: Takafumi Shido's Yet Another Scheme Tutorial (`http://www.shido.info/lisp/idx_scm_e.html`); Dorai Sitaram's Teach Yourself Scheme in Fixnum Days (`http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-1.html`); or Paul Wilson's An Introduction to Scheme and its Implementation (`ftp://ftp.cs.utexas.edu/pub/garbage/cs345/schintro-v14/schintro_toc.html`).

5.1 Introduction

You've heard about all the hot scripting languages – you might even be tired of hearing about them. But Kawa offers you something different than the scripting-language *du-jour* can. You may be interested in one that runs on the Java virtual machine, either because you have to interact with other Java tools, or because you like having access to all the Java packages out there. Or maybe you don't care about Java, but you care about performance. If so, let me tell you about Kawa, which is actually one of the very oldest language implementations running on the Java Virtual Machine, dating back to 1996.

The Kawa language is a dialect/implementation of the Scheme language. (The Kawa project also supports other languages, including XQuery (`http://www.w3.org/XML/Query`) and Emacs Lisp (`http://jemacs.sourceforge.net`), as well as tools for implementing mew programming languages, but we won't cover that in this tutorial.)

Scheme (`http://www.schemers.org/`) is an established language with many implementations (`http://community.schemewiki.org/?scheme-faq-standards#implementations`), a standard (`http://www.schemers.org/Documents/Standards/`) specification (the traditional R5RS (`http://www.schemers.org/Documents/Standards/R5RS/`), R6RS (`http://www.r6rs.org/`) which was ratified in 2007, and R7RS (`http://www.r7rs.org/`) which was ratified in 2013), and is used by universities for both teaching and research. Scheme also has a reputation for being difficult to learn, with a weird parenthesis-heavy syntax, and hard-to-understand concepts like continuations (`http://en.wikipedia.org/wiki/Continuation`). Luckily, you don't need to understand continuations! (Kawa doesn't fully implement them anyway.)

The following assumes that Kawa is already installed on your computer; if not see these Chapter 4 [Installation], page 59. Running the `kawa` command in interactive mode is a good way start learning Kawa:

```
$ kawa
#|kawa:1|#
```

If you don't have `kawa` but you have a Kawa "jar" and you have Java installed you can instead do:

```
$ java -jar kawa-version-number.jar
#|kawa:1|#
```

The prompt string has the form of a Scheme comment, to make it easier to cut-and-paste. Kawa is expecting you type in an expression or command, which it will evaluate, and then print out the result. For example, a quoted string is a simple expression that evaluates to a string value, which will print as itself, before printing the next prompt:

```
#|kawa:1|# "Hello, world!"
Hello, world!
#|kawa:2|#
```

The most noticable difference from most other programming languages is that Scheme uses "prefix" notation for function calls. For example Kawa has a function `max` which returns the largest value of the arguments. Instead of `max(5, 7, 3)` you write `(max 5 7 3)`:

```
(max 5 7 3) ⇒ 7
```

(We use the ⇒ symbol above to indicate that the expression `(max 5 7 3)` evaluates to the value `7`.)

The prefix notation may feel a bit weird, but you quickly get used to it, and it has some advantages. One is consistency: What are special infix operators in most languages are just regular functions in Scheme. For example, addition is just a regular function call, and `+` is just a regular function name:

```
(+ 2.5 1.2) ⇒ 3.7
```

The same prefix notation is used for special operations like assignments:

```
#|kawa:1|# (set! sqrt-of-2 (sqrt 2))
#|kawa:2|# sqrt-of-2
1.4142135623730951
```

5.2 Booleans

Scheme uses the syntax `#t` and `#f` for Boolean true and false value, respectively. For example, the "less-than" function is named `<`. Its result is true if the first argument is less than the second (or, if there are more than two arguments, that they are in increasing order):

```
(< 3 4) ⇒ #t
(< -3 -4) ⇒ #f
(< 2 3 5 7 11)) ⇒ #t
```

The `if` special form takes two or three sub-expressions: It evaluates the first expression. If that is true it evaluates the second expression; otherwise it evaluates the third expression, if provided:

```
(if (< 3 4) (+ 5 5) (+ 5 6)) ⇒ 10
```

We call `if` a special form rather than a function, because for a function all the arguments are evaluated before the function is called, but in a special form that is not neceassarily the case.

In addition to `#t` any value except `#f` (and the Kawa-specific `#!null`) counts as "true" when evaluating the first expression of an `if`. Unlike C or JavaScript both (zero) and `""` (the empty string) are true:

```
(if 0 (+ 5 5) (+ 5 6)) ⇒ 10
```

You can use `and`, `or`, and `not` to create complex boolean expressions. Of these `and` and `or` are special forms that only evaluate as many of the sub-expressions as needed.

```
(if (not (and (>= i 0) (<= i 9)))
    (display "error"))
```

You can use the `cond` form as an alternative to `if`:

```
(cond ((< 3 3) 'greater)
      ((> 3 3) 'less)
      (else ’equal))       ⇒ equal
```

The null value (written as `#!null` in Kawa or `null` in Java) is also considered as false.

5.3 Numbers

Exact integers and fractions

Kawa has the usual syntax for decimal integers. Addition, subtraction, and multiplication are written using the usual `+`, `-`, and `*`, but these are all prefix functions that take a variable number of arguments:

```
(+ 1 2 3) ⇒ 6
(- 10 3 4) ⇒ (- (- 10 3) 4)  ⇒ 3
(* 2 -6)  ⇒ -12
```

Kawa has arbitrary-precision integers.

Let us implement the factorial (`http://en.wikipedia.org/wiki/Factorial`) function. Type in the following (we'll look at the syntax shortly):

```
#|kawa:1|# (define (factorial x)
#|(---:2|#   (if (< x 1) 1
#|(---:3|#     (* x (factorial (- x 1)))))
```

(The prompt changes to indicate a continuation line.) This binds the name `factorial` to a new function, with formal parameter `x`. This new function is immediately compiled to Java bytecodes, and later a JIT compiler may compile it to native code.

A few tests:

```
#|kawa:4|# (list (factorial 3) (factorial 4))
(6 24)
#|kawa:5|# (factorial 30)
265252859812191058636308480000000
```

Floating-point real numbers

Given what was said above about being able to add, subtract and multiply integers, the following may be unexpected:

```
#|kawa:1|# (/ 2 3)
2/3
#|kawa:2|# (+ (/ 1 3) (/ 2 3))
1
```

In many languages, dividing two integers, as 2/3, would result in 0. At best, the result would be a floating point number, similar to 0.666667. Instead, Kawa has a *rational* number type, which holds the results of divisions *exactly*, as a proper fraction. Hence, adding one third to two thirds will always result in exactly one.

Floating-point real numbers are known in Kawa as *inexact* numbers, as they cannot be stored exactly. Consider:

```
#|kawa:3|# (exact? 2/3)
#t
#|kawa:4|# (exact? 0.33333333)
#f
#|kawa:5|# (exact->inexact 2/3)
0.6666666666666666
```

The first two examples check numbers for being `exact?`; there is a corresponding `inexact?` test. The last shows how an exact number can be converted to an inexact form.

Numbers are converted between exact and inexact versions when required within operations or procedures:

```
#|kawa:6|# (+ 0.33333333 2/3)
0.9999999966666666
#|kawa:7|# (inexact? (+ 0.33333333 2/3))
#t
#|kawa:8|# (sin 2/3)
0.618369803069737
```

Complex numbers

A *complex* number is made from two parts: a *real* part and an *imaginary* part. They are written `2+3i`. A complex number can be manipulated just like other numbers:

```
#|kawa:9|# (+ 2+3i 5+2i)
7+5i
#|kawa:10|# (* 2+3i 4-3i)
17+6i
#|kawa:11|# (integer? (+ 2+3i -3i))
#t
```

Notice how in the last example the result is an integer, which Kawa recognises.

Kawa also includes Section 12.4 [Quaternions], page 187, numbers.

Units and dimensions

In many applications, numbers have a *unit*. For example, 5 might be a number of dollar bills, a weight on a scale, or a speed. Kawa enables us to represent numbers as *quantities*:

numbers along with their unit. For example, with weight, we might measure weight in pounds and ounces, where an ounce is 1/16 of a pound.

Using Kawa, we can define units for our weight measurements, and specify the units along with numbers:

```
#|kawa:12|# (define-base-unit pound "Weight")
#|kawa:13|# (define-unit ounce 0.0625pound)
#|kawa:14|# 3pound
3.0pound
#|kawa:15|# (+ 1pound 5ounce)
1.3125pound
```

In this example we define a base unit, the pound, and a unit based on it, the ounce, which is valued at 0.0625 pounds (one sixteenth). Numbers can then be written along with their unit (making them quantities). Arithmetic is possible with quantities, as shown in the last line, and Kawa will do the smart thing when combining units. In this case, 1 pound and 5 ounces is combined to make 1.3125 pounds.

5.4 Functions

To declare a new function use `define`, which has the following form:

```
(define (function-name parameter-names) body)
```

This creates a new function named *function-name*, which takes *parameter-names* as parameters. When the function is called, the *parameter-names* are initialized with the actual arguments. Then *body* is evaluated, and its value becomes the result of the call.

For example, in the `factorial` function we looked at recently, the *function-name* is `factorial`, and the *parameter-names* is `x`:

```
(define (factorial x)
  (if (< x 1) 1
  (* x (factorial (- x 1)))))
```

Anonymous functions

An *anonymous* function is simply a function which does not have a name. We define an anonymous function using a *lambda expression*, which has the following form:

```
(lambda (parameter-names) body)
```

The lambda expression has the *parameter-names* and *body* of a function, but it has no name. What is the point of this?

An important example is creating a function to act on a list, perhaps using `map`. The `map` function takes two parameters: the first is a function which takes a value and returns a value; the second is a list. Here, we want to double every number in the list.

The usual way of doing this is to create a named function, called `double`, and then apply it to a list:

```
#|kawa:1|# (define (double x)
#|.....2|#    (* 2 x))
#|kawa:3|# (map double (list 1 2 3 4 5))
(2 4 6 8 10)
```

Instead, anonymous functions make it easy to create a function to work on a list, without having to define it in advance:

```
#|kawa:4|# (map (lambda (x) (* 2 x)) (list 1 2 3 4 5))
(2 4 6 8 10)
#|kawa:5|# (define y 3)
#|kawa:6|# (map (lambda (x) (* x y)) (list 1 2 3 4 5))
(3 6 9 12 15)
```

The first example shows the double example rewritten as an anonymous function. The second example shows how the anonymous function can be changed to fit the place in which it is used: here, the value of *y* determines the value by which the list values are multiplied.

Notice that we can name our anonymous functions, in just the same way we name any value in Kawa, using `define`:

```
(define double
   (lambda (n)
       (* 2 n)))
```

although more frequently we use the short-hand for defining functions, which we have already met:

```
(define (double n)
  (* 2 n))
```

Anonymous functions are "first-class values" in Kawa, and can be passed to other functions as arguments (like we did with `map`), and they can even be created and returned by functions as results.

Optional, rest and keyword parameters

You can declare a function that takes optional arguments, or a variable number of arguments. You can also use keyword parameters.

The following function illustrates the use of *optional* arguments. The function identifies an optional argument `z`: if the function is called with 3 arguments, `z` will be bound to the third value, otherwise it will be `#f`.

```
(define (addup x y #!optional z)
  (if z
    (+ x y z)
    (+ x y)))
```

The following examples show `addup` applied to 2, 3 and invalid arguments. It is an error to pass just one argument or more than three: `x` and `y` are compulsory, but `z` is optional.

```
#|kawa:12|# (addup 1 2)
3
#|kawa:13|# (addup 1 2 3)
6
#|kawa:14|# (addup 1)
/dev/stdin:14:1: call to 'addup' has too few arguments (1; min=2, max=3)
#|kawa:15|# (addup 1 2 3 4)
/dev/stdin:15:1: call to 'addup' has too many arguments (4; min=2, max=3)
```

In this example, a better way to define the function would be to include a default value for `z`, for when its value is not given by the caller. This is done as follows, with the same behavior as above:

```
(define (addup x y #!optional (z 0))
  (+ x y z))
```

You can include as many optional parameters as you wish, after the `#!optional`.

Rest arguments are an alternative way to pass an undefined number of arguments to a function. Here is `addup` written with rest arguments, notice the variable name after the . (dot):

```
(define (addup x y . args)
  (+ x y (apply + args)))
```

The `args` are simply a list of all remaining values. The following now all work, as the function only requires a minimum of two numbers:

```
#|kawa:4|# (addup 1 2)
3
#|kawa:5|# (addup 1 2 3)
6
#|kawa:6|# (addup 1 2 3 4 5 6 7 8)
36
```

An alternative way to identify the rest args is with `#!rest`:

```
(define (addup x y #!rest args)
  (+ x y (apply + args)))
```

Finally, it can be useful to identify parameters by name and, for this, Kawa provides *keyword* arguments. Consider the following function:

```
#|kawa:38|# (define (vector-3d #!key x y z)
#|.....39|#   (vector x y z))
#|kawa:40|# (vector-3d #:x 2 #:z 3 #:y 4)
#(2 4 3)
```

`vector-3d` is defined with three keyword arguments: `x`, `y`, and `z`. When the function is called, we identify the name for each value by writing `#:` at the start of the name. This allows us to write the arguments in any order. Keyword parameters can also be given default values, as with optional parameters. Keyword parameters with no default value, and no value in the caller, will get the value `#f`.

In the caller, keywords are symbols with `#:` at the front (or `:` at the end): Section 10.3 [Keywords], page 165.

All these extended types of arguments are available both for “named” and for “anonymous” functions. Optional, rest and keyword arguments can be mixed together, along with the usual arguments. For details Section 11.2 [Extended formals], page 169,

5.5 Variables

You can declare a variable using a `!` form. This takes a variable name, and an expression. It declares a new variable with the given name, and gives it the value of the expression.

```
#|kawa:1|# (! binary-kilo 1024)
```

```
#|kawa:2|# (! binary-mega (* binary-kilo binary-kilo))
#|kawa:3|# binary-mega
1048576
```

If you prefer, you can use `define` instead of `!`:

```
#|kawa:1|# (define binary-kilo 1024)
#|kawa:2|# (define binary-mega (* binary-kilo binary-kilo))
#|kawa:3|# binary-mega
1048576
```

The advantage of using `define` is that it is portable to other Scheme implementations. The advantages of using `!` is that it is shorter; it generalizes to patterns (see later); and it guards against accidentally "shadowing" a variable by a nested variable with the same name.

A `!` (or `define`) typed into the command-line defines a top-level variable.

You can also declare local variables, which are variables defined for a given block of code. For example, in the following code `let` is used to set up a local binding of `x` to 3: this does not affect the outer binding of `x` to 5:

```
(define x 5)

(let ((x 3))
  (display x))  ⇒ 3

(display x)     ⇒ 5
```

Alternative forms for defining local variables are `let`, `let*`, or `letrec/letrec*`.

The differences are in the order in which definitions are made. `let` evaluates all its definitions in the environment holding at the start of the `let` statement. In the following example, the local variables are defined using values from the global variables:

```
(define x 5)
(define y 2)

(let ((x (+ 2 y))  ; uses value of global y, i.e. 2
      (y (+ 3 x))) ; uses value of global x, i.e. 5
  (display (list x y)))  ⇒  (4 8)
```

`let*` instead evaluates each definition in the environment holding at the start of the `let*` statement, along with all *previous* local definitions. In the following example, `y` is now defined with the *local* value of `x`:

```
(define x 5)
(define y 2)

(let* ((x (+ 2 y))  ; uses value of global y, i.e. 2
       (y (+ 3 x))) ; uses value of local x, i.e. 4
  (display (list x y)))  ⇒  (4 7)
```

`letrec/letrec*` are similar, but allow the definition of recursive functions:

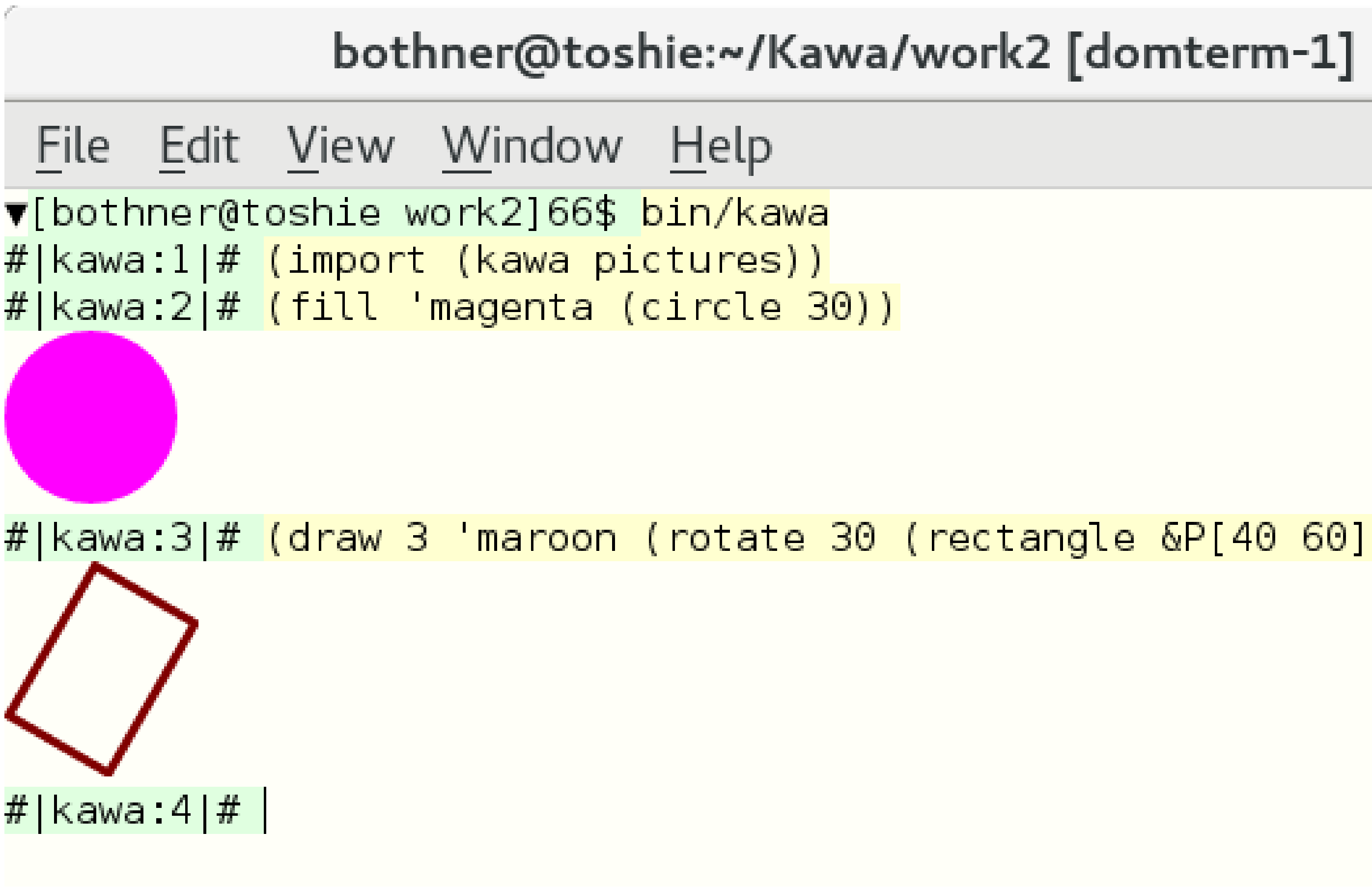
```
(letrec ((is-even? (lambda (n) (and (not (= 1 n))
                                    (or (zero? n)
                                       (is-odd? (- n 1))))))
         (is-odd? (lambda (n) (and (not (zero? n))
                                   (or (= 1 n)
                                       (is-even? (- n 1)))))))
  (display (is-even? 11)))   ⇒  #f
```

5.6 Composable pictures

The `pictures` library lets you create geometric shapes and images, and combine them in interesting ways. You first need to import the library:

```
(import (kawa pictures))
```

The easiest way to use and learn the library is with a suitable REPL, where you can type expressions that evaluate to pictures values, and view the resulting pictures directly on the console. The easiest way is to start the `kawa` command with the `-w` flag. Alternatively, you can use a [Using DomTerm], page 100-based terminal emulator such as `qtdomterm` (which is shown in the image below), and then the `kawa` command.


bothner@toshie:~/Kawa/work2 [domterm-1]
File Edit View Window Help

```
▼[bothner@toshie work2]66$ bin/kawa
#|kawa:1|# (import (kawa pictures))
#|kawa:2|# (fill 'magenta (circle 30))
#|kawa:3|# (draw 3 'maroon (rotate 30 (rectangle &P[40 60]
#|kawa:4|#
```

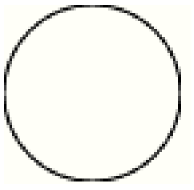
The above image shows two simple examples: a filled circle (radius 30 pixels, color magenta), and a non-filled rotated rectangle (color maroon 3-pixel wide strokes).

See Section 21.1 [Composable pictures], page 356, for details and more examples.

Shapes and coordinates

A *shape* is a geometrical figure consisting of one or more curves and lines. One kind of shape is a circle; you can create one with the `circle` procedure, specifying the radius in "pixels".

```
#|kawa:1|# (import (kawa pictures))
#|kawa:2|# (circle 30)
```



It you print a shape, it will show it as a thin black curve.

A *point* has two real-numbered parts: the point's x-coordinate, and its y-coordinate. The x-coordinate increases as you move right along the page/screen, while the y-coordinate increases as you move *down*. (Unlike traditional mathematics, where the y-coordinate increases as you go up.) The unit distance is one "pixel", which is defined as CSS or HTML. You can create a point with `&P` operator. For example:

```
&P[30 20]
```

is a point 30 pixels right and 20 pixels down from the origin point. To create a circle centered on that point do `(center 30 &P[30 20])`.

The expression `(rectangle &P[10 20] &P[50 40])` creates a rectangle whose upper left corner is (10,20) and whose lower right corner is (50,40).

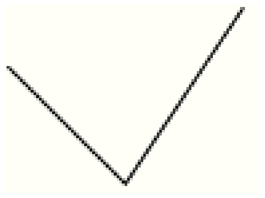
A *dimension* is a pair, a width and height, and is written:

```
&D[width height]
```

In addition to being used for sizes, a dimension is also used for relative offsets. For example, the previous rectangle could also be written `(rectangle &P[10 20] &D[40 20])`.

You can use `line` to create a line. More generally, if you specify *n* points you get a *polyline* of *n-1* line segments:

```
#|kawa:3|# (line &P[10 20] &P[50 60] &P[90 0])
```

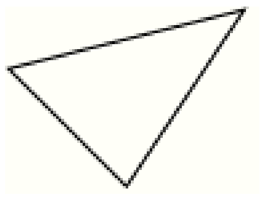


The same line using dimensions for relative offsets:

```
#|kawa:4|# (line &P[10 20] &D[40 20] &D[40 -60])
```

A *closed shape* is one whose end point is the same as its start point. The `polygon` function creates one using straight line segments

```
#|kawa:5|# (polygon &P[10 20] &P[50 60] &P[90 0])
```



Colors and filling

You can override the default color (black) using the `with-paint` procedure, which takes a color and a picture to produce a new picture:

```
#|kawa:6|# (with-paint 'red (circle 32))
```

The first argument can be either one of the standard CSS/HTML5 color names (such as `'red` or `'medium-slate-blue`), or an integer representing an sRGB color, usually written as a hex literal in the form `#xRRGGBB`:

```
#|kawa:7|# (with-paint #x0808FF (circle 32))
```

The name `with-paint` is because the first argument can be not just a color, but a general "paint", such as a gradient or a background image. However, we won't go into that.

If the shape is closed, you can "fill" its inside:

```
(fill (circle 32))
```

You can change the color using `with-paint`:

```
(with-paint 'goldenrod (fill (circle 32)))
```

or as an extra argument to `fill`:

```
(fill 'goldenrod (circle 32))
```

draw TODO

Images

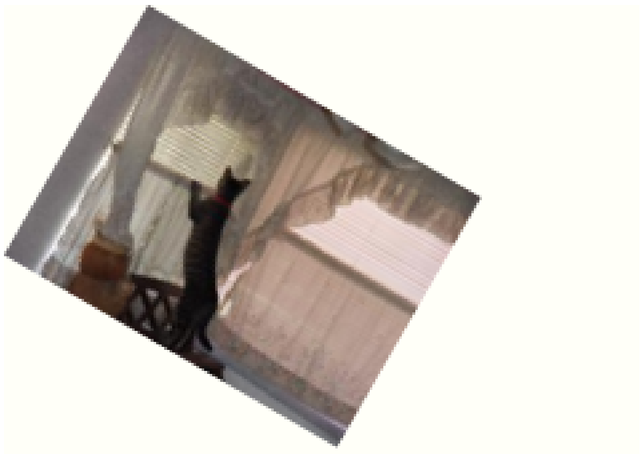
An image is a picture represented as a rectangular grid of color values. It may be a photograph from a camera, or be created by a painting program like Photoshop or gimp. You can use `image-read` to read an image from a file, typically a `.png` or `.jpg` file.

```
#|kawa:10|# (define img1 (image-read "http://pics.bothner.com/2013/Cats/06t.jpg"))
```

```
#|kawa:11|# img1
```

Transforms TODO

```
#|kawa:12|# (scale 0.6 (rotate 30 img1))
```



Combining and adjusting pictures TODO

Using and combining pictures TODO

5.7 Lists and sequences

A *sequence* is a generalized array or list: Zero or more values treated as a compound value. Sequences have certain common operations, including indexing and iteration. (*Technical note:* Sequences generally implement the `java.util.List` interface, but Kawa will also treat strings and native Java arrays as sequences.)

Lists

In traditional Lisp-family languages, the *list* is the most important kind of sequence. (Don't confuse Java's `List` interface with Kawa's use of the word *list*. They're related, in that a Kawa "list" implements the `List` interface, so any *list* is also `List`, but not vice versa.)

A list is implemented as a chain of linked *pairs*. You can create a constant list by quoting a parenthesized list:

```
'(3 4 (10 20 30) "a string")
```

See Section 14.2 [Lists], page 235, for details and operations.

Vectors

A *vector* is a sequence that is implemented by storing the elements side-by-side in memory. A vector uses less space than a list of the same length, and is generally more efficient than a list.

To create a vector you can use a bracketed list:

```
(! vec1 ['A 'B 'C 'D 'E 'F])
```

This creates a vector of 6 symbols and binds it to `vec1`. To select an element you can use the traditional `vector-ref` procedure:

```
(vector-ref vec1 3) ⇒ 'D
```

Alternatively, in Kawa you can use function-call notation:

```
(vec1 3) ⇒ 'D
```

You can also create a vector using the traditional `vector` constructor:

```
(! vec2 (vector 'A 'B 'C 'D 'E 'F))
```

There is one important difference between `vec1` and `vec2`: You can modify `vec2` by changing some or all of its elements. You can't do that for `vec1`. (We say that `vec1` is an *immutable* or *constant* vector, while `vec1` is a *mutable* or *modifiable* vector.) To change an element use either the traditional `vector-set!` procedure, or function-call notation:

```
(vector-set! vec2 2 'Y)
(set! (vec2 4) 'Z)
vec2 ⇒ ['A 'B 'Y 'D 'Z 'F]
(vector-set! vec1 2 'Y) ⇒ throws exception
```

See Section 14.3 [Vectors], page 237, for details and operations.

Java arrays and primitive vectors

See Section 19.14 [Array operations], page 331, for examples.

Indexing of general sequences

You can use function-call notation to index a generalized sequence, whether it is a list, vector, any `java.util.List`, native Java array, or string:

```
((list 'A 'B 'C 'D) 2)  ⇒ 'C
("abcdef" 3)  ⇒  ⇒
(! farr (float[] 1.5 3 4.5))  ;; native Java array
(farr 2) ⇒ 4.5
```

Note that indexing a list with an index *i* will be slow, since it has to step through the list *i* times. (So don't do that!)

Ranges

A *range* is a sequence of numbers in order, spaced uniformly apart. Usually, these are (exact) integers that increase by one. The usual notation is:

```
[start <: end]
```

This is the sequence of integers starting with the integer *start* (inclusive) and ending with the integer *end* (exclusive). For example `[3 <: 7]` is the sequence `[3 4 5 6]`.

The '`<:`' is a keyword; the `<` is a mnemonic for the set of integers that are `<` the end value 6. You can also use `<=:` if you want to include the upper bound: `[4 <=: 8]` is `[4 5 6 7 8]`.

You can use `>=:` or `>:` for a decreasing range. `[5 >=: 1]` or `[5 >: 0]` both evaluate to `[5 4 3 2 1]`. You can also specifify a step value: `[1 by: 2 <=: 9]`, which evaluates to `[1 3 5 7 9]`. (Section 14.6 [Ranges], page 246.)

Using vector and ranges indexes

If an index is a sequence of integers, the result is a new sequence (of the same type) selecting only the elements matching the index values. For example:

```
#|kawa:2|# (vec1 [3 5 2])
#(D F C)
```

In general, `((V1 V2) I)` is `(V1 (V2 I))`.

You can use a range to create a slice - a contiguous subset of a list.

```
#|kawa:3|# (vec1 [2 <: 6])
#(C D E F)
```

A range is different from a vector integer in that you can use a range as the index in the LHS of a set!:

```
#|kawa:4|# (set! (vec1 [2 <: 4]) #(a b c d e))
#|kawa:5|# vec1
#(A B a b c d e E F)
```

Notice how the number of replaced elements can be different then the number of elements in the replacement value. I.e. you can do insertion and deletion this way.

```
#|kawa:7|# (! str1 (string-copy "ABCDEF"))
#|kawa:8|# (set! (str1 [2 <: 5]) "98")
AB98F
```

5.8 Creating and using objects

An *object* is a value that has the following features:

- class - each object is an instance of a specific class, making it part of the class hierarchy, which is an important aspect of the type system;
- properties - various fields and methods, depending on the class;
- identity - it is distinct from all other objects, even if all the properties are the same.

We later discuss Section 5.11 [Tutorial - Classes], page 84. Here we assume you're using an existing class, which could be written in Java or Scheme.

Creating a new object

To create a new object of class `T` you call `T` as if it were a function, passing it the various constructor arguments:

```
(java.io.File "src" "build.xml")
```

If there are keyword arguments they are used to initialize the corresponding named properties:

```
(! button1 (javax.swing.JButton text: "Do it!" tool-tip-text:  "do it"))
```

This create a new `JButton` object (using `JButton`'s default constructor), and sets the `text` and `tool-tip-text` properties (by calling `JButton`'s `setText` and `setToolTipText` methods). If there are constructor arguments, they must come before the keywords.

For objects that have components or elements, you can list these at the end. For example:

```
(java.util.ArrayList 11 22 33)
```

This creates a fresh `java.util.ArrayList` (using the default constructor), and then calls the `add` method 3 times.

If you prefer you can use the `make` procedure, but that only handle simple constructor calls:

```
(make java.io.File "src" "build.xml")
```

See Section 19.10 [Allocating objects], page 324, for details.

Calling instance methods

Given an object *obj* of a class that has a method *meth*, you can call it with argumens *v1* ... *v2* using Section 7.7 [Colon notation], page 115:

```
(obj:meth v1 ... v2)
```

For example:

```
(button1:paintImmediately 10 10 30 20)
```

If you prefer, you can use the `invoke` procedure, normally with a quoted method name:

```
(invoke button1 'paintImmediately 10 10 30 20)
```

You need to use `invoke` (rather than colon notation) if *obj* is a `Class` or a type expression, or its class implements `gnu.mapping.HasNamedParts`.

See Section 19.9 [Method operations], page 320, for details.

Accessing properties

If *obj* has a field or property named *fld* you can also use colon notation:

```
obj:fld
```

You use the same syntax whether *fld* is an actual field in the object, or a *property* (in the Java Beans sense). The latter is implemented using a getter/setter pair: Methods named `getF` and `setF`, respectively. For example:

```
button1:tool-tip-text
```

is equivalent to:

```
(button1:getToolTipText)
```

You can also change a field or property using colon notation:

```
(set! obj:fld value)
```

For example:

```
(set! button1:tool-tip-text "really do it!")
```

This is equivalent to:

```
(button1:setToolTipText "really do it!")
```

Instead of colon notation, you can use the `field` procedure.

See Section 19.11 [Field operations], page 327, for details.

Static fields and methods

Kawa views static properties and methods as properties and methods of the class itself. To call a static method use the syntax:

```
(clas:meth v1 ... vn)
```

For example:

```
(java.math.BigDecimal:valueOf 12345 2) ⇒ 123.45
```

To access a static field do *`clas:fld`*. For example:

```
java.awt.Color:RED
```

You can also use the `static-field` and `invoke-static` procedures.

5.9 Types and declarations

A *type* is a named value for a set of objects with related properties. For example, `vector` is the type for standard Scheme vectors. You can use a type to specify that a variable can only have values of the specified type:

```
#|kawa:5|# (define v ::vector #(3 4 5))
#|kawa:6|# v
#(3 4 5)
#|kawa:7|# (set! v 12)
/dev/stdin:7:1: warning - cannot convert literal (of type gnu.math.IntNum) to vector
Value (12) for variable 'v' has wrong type (gnu.math.IntNum) (gnu.math.IntNum can-
not be cast to gnu.lists.FVector)
at atInteractiveLevel$7.run(stdin:7)
at gnu.expr.ModuleExp.evalModule(ModuleExp.java:302)
at kawa.Shell.run(Shell.java:275)
at kawa.Shell.run(Shell.java:186)
at kawa.Shell.run(Shell.java:167)
at kawa.repl.main(repl.java:870)
Caused by: java.lang.ClassCastException: gnu.math.IntNum cannot be cast to gnu.lists.F
... 6 more
```

Using a type specification catches errors, and makes your programs more readable. It can also allow the Kawa compiler to generate code that runs faster.

You can use a type to check that a value is an instance of the type, using either the `instance?` function:

```
(instance? #(3 4 5) vector) ⇒ #t
```

```
(instance? '(3 4 5) vector) ⇒ #f
```

As a convenience, you can use a type-name followed by a "?":

```
(type? val) == (instance? val type)
```

You can "call" a type as if it were a function, which constructs a new instance of the type. The following example shows how to construct a normal Scheme vector, and a Java array of ints:

```
#|kawa:1|# (vector)
#()
#|kawa:2|# (instance? (vector) vector)
#t
#|kawa:3|# (define x (int[] 1 2 3))
#|kawa:4|# x
[1 2 3]
#|kawa:5|# (instance? x int[])
#t
```

A fully-qualified Java class is a type name. So are the names of Java primitive types. So are Java array types, as shown above.

e.g. a JFrame is constructed by using its class name as a function:

```
#|kawa:6|# (javax.swing.JFrame)
javax.swing.JFrame[frame0,0,25,0x0,invalid,hidden,layout=java.awt.BorderLayout,
title=,resizable,normal,defaultCloseOperation=HIDE_ON_CLOSE,
rootPane=javax.swing.JRootPane[,0,0,0x0,invalid,
layout=javax.swing.JRootPane$RootLayout,alignmentX=0.0,alignmentY=0.0,border=,
flags=16777673,maximumSize=,minimumSize=,preferredSize=],rootPaneCheckingEnabled=true]
```

A type is a true run-time value:

```
(define mytypes (list vector list string))
(instance? #(3 4 5) (car mytypes) ⇒ #t
```

The `define-alias` form is useful for defining shorter names for types, like a generalization of Java's `import` statement:

```
(define-alias jframe javax.swing.JFrame)
```

5.10 Exceptions and errors

Kawa supports the exception framework and forms from R6RS and R7RS. See Section 8.9 [Exceptions], page 151, for details.

Native exception handling

You can also work with native Java exceptions at a low level.

The `primitive-throw` procedure throws a `Throwable` value. It is implemented just like Java's `throw`.

```
(primitive-throw (java.lang.IndexOutOfBoundsException "bad index"))
```

You can catch an exception with the `try-catch` syntax. For example:

```
(try-catch
  (do-a-bunch-of-stuff)
```

```
    (ex java.lang.Throwable
      (format #f "caught ~a~%~!" ex)
      (exit)))
```

A `try-finally` does the obvious:

```
(define (call-with-port port proc)
  (try-finally
   (proc port)
   (close-port port)))
```

Both `try-catch` and `try-finally` are expression forms that can return values, while the corresponding Java forms are statements that cannot return values.

5.11 Classes

See Section 19.1 [Defining new classes], page 298, for the gory details; no tutorial yet.

5.12 Other Java features

Import

The `import` form can be used to avoid having to write fully-qualified class names. For example:

```
(import (class java.util
               Map
               (HashMap HMap)))
```

This defines aliases for two classes in the `java.util` package, one with renaming: `Map` is an alias for `java.util.Map`, and `HMap` is an alias for `java.util.HashMap`.

The `class` keyword is needed because the `import` form is also used for Kawa's module system. See [importing-class-names], page 385, and Section 19.6 [Importing], page 312, for details.

Synchronized blocks

You can use a `synchronized` expression:

```
(synchronized obj form1 ... formn)
```

This waits until it can get an exclusive lock on *obj* and then evaluates *form1* through *formn*. Unlike Java, this is an expression and returns the value of *formn*.

Annotations

You can write annotation declarations - see Section 19.4 [Annotations], page 304, for details.

Kawa does not yet support annotations on types, or declaring new annotation classes.

Reference Documentation

6 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 that `$PREFIX/bin` is in your `$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 Virtual Machine. How you do this depends on the Java implementation. For Oracle's JDK, and some other implementations, you must have the Java evaluator (usually named `java`) 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 4.2 [Running Java], page 60.

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).

6.1 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.)

6.1.1 Argument processing

Kawa processes the command-line arguments in order. Options (which either start with '`-`' or contain a '`=`') may "use up" one or more command arguments. Some of the options ('`-c`', '`-e`', '`-f`', '`-s`', '`-C`', `-w`, '`--`', `--browse-manual`) are *action options*; others set various properties.

When all the command-line arguments have been "used up" and if no action options have been seen, then Kawa enters an interactive read-eval-print loop. (If an action option has been seen, we're done.)

If the next command-line argument is not an option (does not start with '`-`' nor contains a '`=`') then we're done if we've seen an action option (and the last action option wasn't preceded by `--with-arg-count`). (Presumably any remaining arguments were command-line-arguments used by the action option.)

Otherwise, the first remaining argument names either a file that is read and evaluated, or a compiled class. In the former case, the whole file is read and compiled as a module before being loaded (unlike the `-f` flag which reads and evaluates the file command by command.) If the argument is the fully-qualified name of a class, then the class is loaded, an instance allocated, and its `run` method invoked. If the class was compiled from a Kawa Scheme module, then invoking `run` has the effect of evaluating the module body. The `command-line-arguments` vector is set to any remaining arguments after the file/class name. (This can be overridden with the `--with-arg-count` option. Command-line processing continues if there are any further arguments.)

6.1.2 General options

`-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 remaining arguments (if any) are passed to '`command-line-arguments`' and (the `cdr` of) `(command-line)`, 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.

`--script` *`filename-or-url`*
`--script`*`N`* *`filename-or-url`*
: The global variable '`command-line-arguments`' is set to the remaining arguments (if any). Kawa reads and evaluates expressions from the file named by *filename-or-url*. If `script` is followed by an integer *N*, then *N* lines are skipped first.

: Skipping some initial lines is useful if you want to have a non-Kawa preamble before the actual Kawa code. One use for this is for Kawa shell scripts (see Section 6.2 [Scripts], page 95).

`-w`
`-w`*`sub-option`*
: Creates a new top-level window, and runs an interactive read-eval-print in the new window. See [New-Window], page 99. 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.

If Kawa was built with a `.git` repository present, also prints the result of `git describe`.

`--browse-manual`
`--browse-manual=`*`command`*
Browse a local copy of the documentation (this manual).

This creates a mini web-server that reads from `doc/kawa-manual.epub`, which is included in the binary distributions, but not built by default from source.

If no *command* is specified, creates a new mini-browser-window using JavaFX (if the JavaFX modules are available), or creates a new window or tab in your default web browser (otherwise). If *command* is a string containing `%U`, then Kawa replaces `%U` with a URL that references itself, and then executes the resulting command. If *command* does not contain `%U`, then *command* becomes *command*`" %U"`. For example to use the Firefox browser to browse the manual do either of:

```
kawa --browse-manual=firefox
kawa --browse-manual="firefox %U"
```

`--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 start. This option allows you to connect using any Telnet client program to a remote "Kawa server".

`--with-arg-count=`*`argc`*
This option is used before an action option (such as `-f`). The *argc* arguments after the action become the value of the `command-line-arguments` during the action. When the action is finished, command-line-processing resumes after skipping the *argc* arguments.

For example:

```
$ kawa -f a.scm -f b.scm x y
```

When evaluating `a.scm` the `command-line-arguments` by default is *all* the remaining arguments: `["-f" "b.scm" "x" "y"]`. Then `b.scm` is evaluated with `command-line-arguments` set to `["x" "y"]`

```
$ kawa --with-arg-count=0 -f a.scm -f b.scm x y
```

In this case `a.scm` is evaluated with `command-line-arguments` set to the empty vector `[]`, and then `b.scm` is evaluated with `command-line-arguments` set to `["x" "y"]`

```
$ kawa --with-arg-count=4 -f a.scm -f b.scm x y
```

In this case `a.scm` is evaluated with `command-line-arguments` set to `["-f" "b.scm" "x" "y"]`. Since command-line processing skips the arguments specified by `--with-arg-count=4`, in this case `b.scm` is not evaluated.

6.1.3 Options for language selection

`--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.)

`--r5rs`
`--r6rs`
`--r7rs` Provide better compatibility with the specified Scheme standards. (This is a work-in-progress.) For example `--r6rs` aims to disable Kawa extensions that conflict with R6RS. It does not aim to disable all extensions, only incompatible extensions. These extensions disable the colon operator and keyword literals, as well as the use of initial '`@`' as a splicing operator. The "`l`" exponent suffix of a number literal creates a floating-point double, rather than a `BigInteger`. Selecting `--r5rs` makes symbols by default case-insensitive.

`--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 Section 20.11.3 [KRL], page 355.

`--brl` Set the default language to KRL, in BRL-compatibility mode. See Section 20.11.3 [KRL], page 355.

`--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.

`--pedantic`
Try to follow the approprate language specification to the letter, even in corner cases, and even if it means giving up some Kawa convenience features. This flag so far only affects the XQuery parser, but that will hopefully change.

6.1.4 Options for warnings and errors

`--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.) This defaults to on; to turn it off use the `--no-warn-undefined-variable` flag.

`--warn-unknown-member`
: Emit a warning if the code references a named member (field or method) for which there is no match in the compile-time type of the receiver. This defaults to on; to turn it off use the `--no-warn-unknown-member` flag.

`--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 defaults to the value of `--warn-unknown-member`, to turn it off use the `--no-warn-invoke-unknown-method` flag.

`--warn-unused`
: Emit a warning if a variable is unused or code never executed. This defaults to on; to turn it off use the `--no-warn-unused` flag.

`--warn-uninitialized`
: Warn if accessing an uninitialized variable. This defaults to on; to turn it off use the `--no-warn-uninitialized` flag.

`--warn-unreachable`
: Emit a warning if the code can never be executed. This defaults to on; to turn it off use the `--no-warn-unreachable` flag.

`--warn-void-used`
: Emit a warning if an expression depends on an expression that is void (always has zero values), including call to `void` functions and method. Also warn if an expression depends on a conditional (`if`) that has no "else" clause. Examples include using the value of `set-car!` as an argument to a function, or to initialize a variable. This defaults to on; to turn it off use the `--no-warn-void-used` flag.

`--warn-as-error`
: Treat a compilation warning as if it were an error and halt compilation.

`--max-errors=`*`value`*
: Print no more than *value* errors or warnings (at a time). The value `-1` removes the limit. The initial default is 20. (A single error may so confuse Kawa that it prints very many useless error messages.)

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-unknown-member` is the same as `--warn-unknown-member=no`.

These options can also be used in the module source, using `module-compile-options` or `with-compile-options`. (In that case they override the options on the command line.)

6.1.5 Options for setting variables

`name``=`*`value`*
: Set the global variable with the specified *name* to the given *value*. The type of the *value* is currently unspecified; the plan is for it to be like XQuery's *untyped atomic* which can be coerced as needed.

`{namespace-uri}local-name=value`
: Set the global variable with the specified namespace uri and namespace-local name to the given value.

These options are processed when invoking the `kawa` application (i.e. the `kawa.repl` application). If you want a Kawa application compiled with `--main` to process these these assignments, call the `process-command-line-assignments` utility function.

`-Dvariable-name=variable-value`
: Sets the JVM property *variable-name* to *variable-value*, using the `setProperty` method of `java.lang.System`.

6.1.6 Options for the REPL console

`--console`
`--no-console`
: Usually Kawa can detect when the standard input port is a "console" or "terminal", but these are useful for overriding that detection. The `--console` flag is useful when the standard input is a pipe, but you want to direct Kawa to treat it as an interactive terminal. The `--no-console` flag was useful for older pre-Java-6 implementations that did not have the `java.lang.Console` class.

`console:type=`*console-types*
`console:use-jline=`[`yes|no`]
`console:jline-mouse=`[`yes|no`]
: See the Section 6.3 [REPL Console], page 97, section.

`console:prompt1=prompt1`
`console:prompt2=prompt2`
: Initialize [input-prompt1], page 284, respectively.

See also the `--output-format` flag.

6.1.7 Options for controlling output formatting

`--output-format format`
`--format format`
: Change the default output format to that specified by *format*. See Section 17.1 [Named output formats], page 270, for more information and a list.

`out:base=integer`
: The number base (radix) to use by default when printing rational numbers. Must be an integer between 2 and 36, and the default is of course 10. For example the option `out:base=16` produces hexadecimal output. Equivalent to setting the `*print-base*` variable.

`out:radix=no|yes`
: If true, prints an indicator of the radix used when printing rational numbers. The default is `no`. Equivalent to setting the `*print-radix*` variable.

`out:doctype-system=system-identifier`
: If `out:doctype-system` is specified then a `DOCTYPE` declaration is written before writing a top-level XML element, using the specified *system-identifier*.

`out:doctype-public=`*`public-identifier`*
: Ignored unless `out:doctype-system` is also specified, in which case the *public-identifier* is written as the public identifiers of the `DOCTYPE` declaration.

`out:xml-indent=`*`kind`*
: Controls whether extra line breaks and indentation are added when printing XML. If *kind* is `always` or `yes` then newlines and appropriate indentation are added before and after each element. If *kind* is `pretty` then the pretty-printer is used to only add new lines when an element otherwise won't fit on a single line. If *kind* is `no` (the default) then no extra line breaks or indentation are added.

`out:line-length=`*`columns`*
`out:right-margin=`*`columns`*
: Specifies the maximum number of number of columns in a line when the pretty-printer decides where to break a line. (The two options are equivalent.)

6.1.8 Options for compiling and optimizing

`--target` *`version`*
: The *version* can be a JDK or Java specification version: `5`, `6`, or `7`. The JDK versions `1.5` and `1.6` are equivalent to `5` or `6`, respectively. Specify a JVM (classfile) version to target. This is useful if (for example) you use Java 6, but want to create `.class` files that can run on Java 5. In that case specify `--target 5`.

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.

`--no-inline`
: Disable inlining of known functions and methods. The generated code runs slower, but you can more reliably trace procedures. Normally Kawa will assume that a procedure `fn` declared using a `(define (fn args) body)` form is constant, assuming it isn't modified in the current module. However, it is possible some other module might modify the binding of `fn`. You can use the `--no-inline` to disable the assumption that `fn` is constant.

The default is currently `--no-full-tailcalls` because it is usually faster. 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.

These options can also be used in the module source, using `module-compile-options` or `with-compile-options`. (In that case they override the options on the command line.)

The options '`-C`', '`-d`', '`-T`', '`-P`', '`--main`' '`--applet`', and `--servlet` are used to compile a Scheme file; see Section 6.5.1 [Files compilation], page 101. The options '`--module-static`', `--module-nonstatic`, `--no-module-static`, and `--module-static-run` control how a module is mapped to a Java class; see [static-or-non-modules], page 310. The option '`--connect` *`portnum`*' is only used by the '`kawa`' front-end program.

6.1.9 Options for debugging

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

`--debug-dump-zip`
: Normally, when Kawa loads a source 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.

`--debug-syntax-pattern-match`
: Prints logging information to standard error when a `syntax-rules` or `syntax-case` pattern matches.

`--debug-error-prints-stack-trace`
: Prints a stack trace with any error found during compilation.

`--debug-warning-prints-stack-trace`
: Prints a stack trace with any warning found during compilation.

`--langserver`
: Starts Kawa in server mode, responding to requests using the Language Server Protocol (`https://langserver.org`). This is used by editors and IDEs for on-the-fly syntax checking and more. Highly experimental.

6.1.10 Options for web servers

JDK 6 (or later) includes a complete web server library.

`--http-auto-handler` *`context-path appdir`*
: Register a web application handler that uses files in the directory *appdir* to handle HTTP (web) requests containing the given *context-path*. That is it handles requests that start with `http://localhost:`*`portcontext-path`*. (This assumes the *context-path* starts with a `/`.) See Section 20.6 [Self-configuring page scripts], page 343.

`--http-start` *`port`*
: Start the web server, listing on the specified *port*.

6.1.11 Options for the JVM

The `kawa` front-end can pass options to the `java` launcher, using `-J` or `-D` options. These must be given *before* any other arguments. For example:

```
kawa -J-Xms48m -Dkawa.command.name=foo foo.scm
```

is equivalent to (ignoring classpath issues):

```
java -Xms48m -Dkawa.command.name=foo kawa.repl foo.scm
```

You can also pass a `-D` option (but not a `-J` option) after the class name, in which case it is processed by the Kawa command-line processor rather than the `java` launcher. The effect is normally the same.

`-J`*`jvm-option`*
: Passes the *jvm-option* to the `java` command, before the class-name (`kawa.repl`) and Kawa options.

`-D`*`variable-name`*`=`*`variable-value`*
: Sets the JVM property *variable-name* to *variable-value*. Equivalent to `-J-D`*`variable-name`*`=`*`variable-value`*.

6.2 Running Command Scripts

If you write a Kawa application, it is convenient to be able to execute it directly (from the command line or clicking an icon, say), without have to explicitly run `kawa` or `java`. On Unix-like systems the easiest way to do this is to write a small shell script that runs your Kawa application.

For modest-sized applications it is convenient if the shell script and the Kawa code can be in the same file. Unix-like systems support a mechanism where a *script* can specify a program 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.

(Windows has *batch files*, which are similar.)

This 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 '`#`' as a multi-purpose prefix, and Kawa specifically uses '`#!`' as a prefix for various Section 10.4 [Special named constants], page 165, such as `#!optional`.

Kawa does recognize the three-character sequence '`#!/`' at the beginning of a file as special, and ignores it. Here is an example:

```
#!/usr/local/bin/kawa
(format #t "The command-line was:~{ ~w~}~%" (command-line))
```

If you copy this text to a file named `/home/me/bin/scm-echo`, set the execute permission, and make sure it is in your `PATH`, then you can execute it just by naming it on command line:

```
$ chmod +x /home/me/bin/scm-echo
$ PATH=/home/me/bin:$PATH
$ scm-env a b
The command-line was: "/home/me/bin/scm-echo" "a" "b"
```

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 command-line was:~{ ~w~}~%" (command-line))
```

This works the same way, but assumes kawa is in the command PATH.

6.2.1 Setting kawa options in the script

If you need to specify extra arguments to kawa, you can run arbitrary shell command inside Scheme block comments. Here is an example:

```
#!/bin/sh
#|
exec kawa out:base=16 out:radix=yes "$0" "$*"
|#
(format #t "The command-line is:~{ ~w~}.~%" (command-line))
(display "It has ")
(display (apply + (map string-length (command-line))))
(display " characters.")
(newline)
```

The trick is to hide the shell code from Kawa inside a #|...|# block-comment. The start of the block comment is a line starting with a #, so it is treated as a comment by the shell. You can then invoke kawa (or java directly) as you prefer, setting up class-path and jars as needed, and passing whatever arguments you want. (The shell replaces the "$0" by the name of the script, and replaces the "$@" by the remaining arguments passed to the script.) You need to make sure the shell finishes before it reaches the end of the block comment or the Scheme code, which would confuse it. The example uses exec, which tells the shell to *replace* itself by *kawa*; an alternative is to use the shell exit command.

If you copy the above file to /tmp/sch-echo and make that file executable, you can run it directly:

```
$ /tmp/scm-echo "a b" "c d"
The command-line is: "/tmp/scm-echo" "a b c d".
It has #x14 characters.
```

When the Kawa reader sees the initial #/ it sets the command name to the file name, so it can be used by a future call to (command-name). If you want to override this you can use the -Dkawa.command.name=*name* option.

Using comments this way has the advantage that you have the option of running the script "manually" if you prefer:

```
$ kawa /tmp/scm-echo out:base=8 "x y"
The command-line is: "/tmp/scm-echo" "out:base=8" "x y".
It has 26 characters.
```

6.2.2 Other ways to pass options using meta-arg or –script

An argument consisting of just a \ (backslash) causes Kawa to read the *second* line looking for options. (Quotes and backslashes work like in the shell.) These replace the backslash in the command line.

This is a less verbose mechanism, but it requires an absolute path to `kawa`, due to shell limitations.

```
#!/usr/bin/kawa \
  --scheme --full-tailcalls
(format #t "The command-line is:~{ ~w~}.~%" (command-line))
```

In this case the effective command line received by Kawa will be `--scheme`, `--full-tailcalls`, followed by the script filename, followed by other arguments specified when running the script.

The backslash used this way originated in scsh (`http://www.scsh.net`) where it is called the *meta-arg*. (Unlike scsh, Kawa's `#!` is not a block comment, but a rest-of-line, though the backslash causes the following line to also be skipped.)

An alternative method is to use the `--script2` option, which tells Kawa to execute the script after ignoring the initial two lines. For example:

```
#!/bin/sh
exec kawa --commonlisp out:base=16 --script2 "$0" "$@"
(setq xx 20) (display xx) (newline)
```

This is slightly more compact than using block-comments as shown earlier, but it has the disadvantage that you can't explicitly use `kawa` or `java` to run the script unless you make sure to pass it the `--script2` option.

6.2.3 Scripts for compiled code

If you compile your Kawa application to class files (or better: a `jar` file), you probably still want to write a small shell script to set things up. Here is one method:

```
#!/bin/sh
export CLASSPATH=/my/path
exec kawa -Dkawa.command.name="$0" foo "$@"
```

Using the `kawa` front-end is a convenience, since it automatically sets up the paths for the Kawa classes, and (if enabled) it provides readline support for the default input port.

Setting the `kawa.command.name` property to `"$0"` (the filename used to invoke the script) enables `(command-line)` to use the script name as the command name.

You can invoke `java` directly, which is necessary when running a `jar` file:

```
#!/bin/sh
exec java -cp /path/to/kawa -Dkawa.command.name="$0" foo.jar "$@"
```

6.3 The REPL (read-eval-print-loop) console

The read-eval-print-loop (REPL) console is a convenient way to do simple programming, test out things, and experiment. As the name implies, the REPL repeatedly (in a loop) prints out a prompt, reads an input command, evaluates it, then prints the result.

The REPL is started when you invoke the `kawa` command with no arguments. For example:

```
$ kawa
#|kawa:1|# (define pi (* 2 (asin 1)))
#|kawa:2|# (list pi (sqrt pi))
(3.141592653589793 1.7724538509055159)
#|kawa:3|#
```

The colors and styles used for the prompt and the user input depend on user preference and the capabilities of the console device. (If you read this on a color screen you should see pale green for the prompt and pale yellow for the user input; this matches the defaults for the DomTerm console.)

You can [Prompts], page 284, if you want. The default format depends on the (programming) language used; the one shown above is used for Scheme. It has the form of a comment, which can be convenient for copying and pasting lines.

You can Section 17.1 [Named output formats], page 270, with the `--output-format` command-line option.

The basic console has few frills, but should work in any enviroment where you have a console or terminal. It has no dependencies, except the kawa `.jar` file (and Java):

```
$ java kawa-3.1.1.jar
#|kawa:2|#
```

On rare occason you may need to specify the `--console` flag.

6.3.1 Input line editing and history

When typing a command in a console it is helpful to go back and correct mistakes, repeat and edit previous commands, and so on. How well you can do this varies a lot depending on which tools you use. Kawa delegates input editing to an external tool. The recommended and default input-editing tool is the JLine3 library (`https://github.com/jline/jline3`), which is bundled with the Kawa binary distribution.

JLine3 handles the normal editing comands, including arrow keys for moving around in the input, and deleting with backspace or delete. In general, JLine3 uses the same keybindings as GNU readline, which are based on Emacs key-bindings.

You can use the up-arrow to move to previous commands in the input history and down-arrow to go forwards. Control-R ("reverse search" searches backwards in the history for a previous command that contains the search string.

Multi-line commands are treated as a unit by JLine3: If Kawa determines that input is "incomplete" it will ask for continuation lines - and you can go back and edit previous lines in the same command. You can explicitly create a multi-line command with Escape-Space. An entry in the command history may be multiple lines.

Tab-completion works for Kawa-Scheme identifiers: If you type TAB after an identifier, Kawa will present a list of possible completions.

There are multiple alternatives to using JLine3. You can use GNU readline (if you configured with `--enable-kawa-frontend`). You can use a front-end program like `rlfe` or `fep`. You can use Emacs shell or scheme mode. You can also use DomTerm in line-edit mode, where the browser handles the editing.

`console:use-jline=[yes|no]`
: Disable (with `no`) or enable (with `yes`, which is the default) input line editing with JLine.

`console:console:jline-mouse=[yes|no]`
: Enable (with `yes`) mouse click reporting from most xterm-like terminals to JLine, which means you can move the input cursor with the mouse. This is disabled by default because it conflicts with other useful mouse actions (text selection using drag; middle-button paste; right-button context menu; and wheel mouse scrolling). If you enable mouse-reporting, on most terminals you can get the standard behavior when pressing the shift key. E.g. to enable selection, drag with the shift key pressed. (However, mouse-wheel scrolling may not work even with shift pressed.)

6.3.2 Running a Command Interpreter in a new Window

Instead of using an existing terminal window for Kawa's REPL console, you can request a new window. The command-line options `-w` creates a new window. Kawa also creates a new window when it needs to create a REPL (for example if invoked with no options) and it is not running in a console.

You have a number of options for how the window appears and what it supports, controlled by text following `-w`. All except `-wswing` (and `-wconsole`) use DomTerm, so they depend on some kind of web browser technology. All except `-wswing` by default use JLine3 input editing, if available.

`-w`
: Pick the default/preferred console implementation. You can specify your preference with the `console:type=` option, which is followed by one of the options below (without the `"-w"` prefix), It can also be list of options separated by semi-colons, in which case they are tried in order.

: The current default (it may change) is as if you specified:

```
console:type="google-chrome;browser;javafx;swing;console"
```

`-wbrowser`
: Creates a Kawa window or tab in your preferred desktop browser. Kawa starts a builtin HTTP and WebSocket server to communicate with the browser.

`-wbrowser=`*`command`*
: Uses *command* to display the Kawa REPL. The *command* should include the pattern `%U`, which Kawa replaces with a URL that it listens to. (Alternatively, it can use the pattern `%W`, which Kawa replaces with the port number of its WebSocket server. However, this feature may be removed.) If the is no `%` in the *command*, Kawa add `" %U"`. Thus `-wbrowser=firefox` is the same as `-wbrowser="firefox %U"`.

`-wgoogle-chrome`
: Creates a new Google Chrome window in “app mode” - i.e. with no location or menu bar. This is the same as `-wbrowser="google-chrome --app=%U"`.

`-wjavafx`
: Creates a new window using JavaFX WebView, which runs in the same JVM as Kawa. While this doesn't currently have much in the way of Kawa-specific

menus or other features, it has the most potential for adding them in the future. However, it does require JavaFX, which is not always available, and which does not see a lot of love from Oracle. (It uses an old version of WebKit.)

`-wswing` Create a console using the Swing toolkit. This is the old implementation of `-w`. It is deprecated because it only supports the builtin Swing line editing. (I.e. neither DomTerm or JLine3 features are available, though "printing" Section 21.1 [Composable pictures], page 356, does work.)

`-wserve`
`-wserve=`*port*
: Starts up an HTTP server (along with a WebSocket server), but does not automatically create any browser windows. Instead you can use any modern browser to load `http://localhost:`*`port`*`/`. If *port* is not specified, the systems selects it (and prints it out).

`-wconsole`
: Same as `"--"` - i.e. it uses the existing console.

`console:type=`*preference-list*
: Specify the behavior of plain `-w`.

6.3.3 Using DomTerm

DomTerm (`http://domterm.org`) is a family of terminal emulators that use the DomTerm JavaScript library.

You can either have Kawa start DomTerm:

```
$ kawa options -w
```

or start a DomTerm terminal emulator and have it start Kawa:

```
$ domterm kawa options --
```

(You can also start a shell in a `domterm` window, and then start `kawa`.)

Either approach works and both give you the benefits of DomTerm:

- A xterm/ansi-compatible terminal emulator, which means you can use (for example) JLine3 for input editing.
- You can "print" images, Section 21.1 [Composable pictures], page 356, or HTML elements.
- Pretty-printing is handled by the terminal, which means line-breaking is re-computed when window width changes.
- Hide/show buttons allow you to temporarily hide/unhide the output from a specific command.
- You can save a session as an HTML file, which can be viewed later. (Still with dynamic line-breaking and pretty-printing, as well as working hide/show buttons.) The file is actually XHTML, so it can be processed with XML-reading tools.
- Distinct styles for prompts, input, error output and regular output, which can be customized with CSS.

For now it is recommended to use both DomTerm and JLine3.

`domterm-load-stylesheet` *stylesheet* [*name*] [Procedure]

The string *stylesheet* should be a literal CSS stylesheet which is downloaded into the current DomTerm console. The new stylesheet is given the attribute `name=`*`name`*, where *name* defaults to `"Kawa"`. If there is an existing styleshеey whose `name` attribute is *name*, it is replaced. In this example we change the background color to light gray:

```
(domterm-load-stylesheet "div.domterm { background-color: lightgray}")
```

6.4 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`' or '`-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. Typing an end-of-file character (normally ctrl-D under Unix) should also work, but that depends on your operating system and terminal interface.

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 [Exiting the current process], page 382, (with 0 or 1 integer arguments).

6.5 Compiling to byte-code

All Scheme functions and source files are invisibly compiled into internal Java byte-codes. (A traditional interpreter is used for macro-expansion. Kawa used to also interpret "simple" expressions in interactive mode, but always compiling makes things more consistent, and allows for better stack traces on errors.)

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.

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 compiled class files do have to be installed somewhere in the `CLASSPATH`.

6.5.1 Compiling 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 in-
file ...
```

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 6.5.4 [Application compilation], page 103.

`--applet` The resulting class inherits from `java.applet.Applet`, and can be used as an applet. See Section 6.5.5 [Applet compilation], page 104.

`--servlet`
: The resulting class implements `javax.servlet.http.HttpServlet`, and can be used as a 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 `require` syntax or the `load` function to load the code, by specifying the top-level class, either as a file name (relative to *outdirectory*) or as 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:

```
(require my.lib.foo)
```

or:

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

Using `require` is preferred as it imports the definitions from `my.lib.foo` into the compile-time environment, while `load` only imports the definitions into the run-time environment.

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
```

Many of the options Section 6.1 [described earlier], page 87, are relevant when compiling. Commonly used options include language selection, the `--warn-xxx` options, and `--full-tailcalls`.

6.5.2 Compiling to an archive file

`compile-file` *source-file compiled-archive* [Procedure]

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 JVM `.class` files. 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.

6.5.3 Compiling using Ant

Many Java projects use Ant (`http://ant.apache.org`) for building Java projects. Kawa includes a `<kawac>` Ant task that simplifies compiling Kawa source files to classes. See the `build.xml` in the Kawa source distribution for examples. See the `kawac` task documentation (`ant-kawac.html`) for details.

6.5.4 Compiling 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.

For example, assuming your Scheme file is `MyProgram.scm`, you have two ways at your disposal to compile this Scheme program to a standalone application:

1. Compile in the regular way described in the previous section, but add the `--main` option.

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

   The `--main` option will compile all Scheme programs received in arguments to standalone applications.
2. Compile in the regular way decribed in the previous section, but add the `main: #t` module compile option to your module.

   ```
   ;; MyProgram.scm
   (module-name <myprogram>)
   (module-compile-options main: #t)
   ```

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

   This way you can compile multiple Scheme programs at once, and still control which one(s) will compile to standalone application(s).

Both methods will create a `MyProgram.class` which you can either `load` (as described 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`', or the R6RS function '`command-line`'.

If there is no explicit `module-export` in a module compiled with `--main` then no names are exported. (The default otherwise is for all names to be exported.)

6.5.5 Compiling 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 (this)))
    (applet:addMouseListener
     (object (java.awt.event.MouseAdapter)
     ((mousePressed e)
      (set! last-x (e:getX))
      (set! last-y (e:getY)))))
    (applet:addMouseMotionListener
     (object (java.awt.event.MouseMotionAdapter)
     ((mouseDragged e)
      (let ((g (applet:getGraphics))
    (x (e:getX))
    (y (e:getY)))
(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:

```
jar cf scribble.jar scribble*.class
```

Finally, you create an '`.html`' page referencing your applet and its support `jar`s:

```
<html><head><title>Scribble testapp</title></head>
<body><h1>Scribble testapp</h1>
You can scribble here:
<br>
<applet code="scribble.class" archive="scribble.jar, kawa-3.1.1.jar" width=200 height=
Sorry, Java is needed.</applet>
```

```
    </body></html>
```

The problem with using Kawa to write applets is that the Kawa `.jar` file is quite big, and may take a while to download over a network connection. Some possible solutions:

- Try to strip out of the Kawa `.jar` any classes your applet doesn't need.
- Java 2 provides a mechanism to install a download extension (`http://java.sun.com/docs/books/tutorial/ext/basics/download.html`).
- Consider some alternative to applets, such as Java Web Start (`http://java.sun.com/products/javawebstart/`).

6.5.6 Compiling to a native executable

In the past it was possible to compile a Scheme program to native code using GCJ. However, using GCJ with Kawa is no longer supported, as GCJ is no longer being actively maintained.

7 Syntax

7.1 Notation

The formal syntax for Kawa Scheme is written in an extended BNF. Non–terminals are written *like-this*. Case is insignificant for non–terminal names. Literal text (terminals) are written **like this**.

All spaces in the grammar are for legibility.

The following extensions to BNF are used to make the description more concise: *thing** or *thing*... both mean zero or more occurrences of *thing*, and *thing*$^+$ means at least one *thing*.

Some non-terminal names refer to the Unicode scalar values of the same name: *character-tabulation* (U+0009), *linefeed* (U+000A), *carriage-return* (U+000D), *line-tabulation* (U+000B), *form-feed* (U+000C), *space* (U+0020), *next-line* (U+0085), *line-separator* (U+2028), and *paragraph-separator* (U+2029).

7.2 Lexical and datum syntax

The syntax of Scheme code is organized in three levels:

1. the *lexical syntax* that describes how a program text is split into a sequence of lexemes,
2. the *datum syntax*, formulated in terms of the lexical syntax, that structures the lexeme sequence as a sequence of *syntactic data*, where a syntactic datum is a recursively structured entity,
3. the *program syntax* formulated in terms of the datum syntax, imposing further structure and assigning meaning to syntactic data.

Syntactic data (also called *external representations*) double as a notation for objects, and the `read` and `write` procedures can be used for reading and writing syntactic data, converting between their textual representation and the corresponding objects. Each syntactic datum represents a corresponding *datum value*. A syntactic datum can be used in a program to obtain the corresponding datum value using `quote`.

Scheme source code consists of syntactic data and (non–significant) comments. Syntactic data in Scheme source code are called *forms*. (A form nested inside another form is called a *subform*.) Consequently, Scheme's syntax has the property that any sequence of characters that is a form is also a syntactic datum representing some object. This can lead to confusion, since it may not be obvious out of context whether a given sequence of characters is intended to be a representation of objects or the text of a program. It is also a source of power, since it facilitates writing programs such as interpreters or compilers that treat programs as objects (or vice versa).

A datum value may have several different external representations. For example, both `#e28.000` and `#x1c` are syntactic data representing the exact integer object 28, and the syntactic data `(8 13)`, `( 08 13 )`, `(8 . (13 . ()))` all represent a list containing the exact integer objects 8 and 13. Syntactic data that represent equal objects (in the sense of `equal?`) are always equivalent as forms of a program.

Because of the close correspondence between syntactic data and datum values, we sometimes uses the term *datum* for either a syntactic datum or a datum value when the exact meaning is apparent from the context.

7.3 Lexical syntax

The lexical syntax determines how a character sequence is split into a sequence of lexemes, omitting non–significant portions such as comments and whitespace. The character sequence is assumed to be text according to the Unicode standard (`http://unicode.org/`). Some of the lexemes, such as identifiers, representations of number objects, strings etc., of the lexical syntax are syntactic data in the datum syntax, and thus represent objects. Besides the formal account of the syntax, this section also describes what datum values are represented by these syntactic data.

The lexical syntax, in the description of comments, contains a forward reference to *datum*, which is described as part of the datum syntax. Being comments, however, these *datum*s do not play a significant role in the syntax.

Case is significant except in representations of booleans, number objects, and in hexadecimal numbers specifying Unicode scalar values. For example, `#x1A` and `#X1a` are equivalent. The identifier `Foo` is, however, distinct from the identifier `FOO`.

7.3.1 Formal account

Interlexeme-space may occur on either side of any lexeme, but not within a lexeme.

Identifiers, ., *numbers*, *characters*, and *booleans*, must be terminated by a *delimiter* or by the end of the input.

```
lexeme ::= identifier | boolean | number
        | character | string
        | ( | ) | [ | ] | #(
        | fl | ` | , | ,@ | .
        | #fl | #` | #, | #,@
delimiter ::= ( | ) | [ | ] | " | ; | #
        | whitespace
```

((UNFINISHED))

7.3.2 Line endings

Line endings are significant in Scheme in single–line comments and within string literals. In Scheme source code, any of the line endings in *line-ending* marks the end of a line. Moreover, the two–character line endings *carriage-return linefeed* and *carriage-return next-line* each count as a single line ending.

In a string literal, a *line-ending* not preceded by a \ stands for a linefeed character, which is the standard line–ending character of Scheme.

7.3.3 Whitespace and comments

intraline-whitespace `::=` *space* `|` *character-tabulation*
whitespace `::=` *intraline-whitespace*
  `|` *linefeed* `|` *line-tabulation* `|` *form-feed*
  `|` *carriage-return* `|` *next-line*
  `|` *any character whose category is Zs, Zl, or Zp*
line-ending `::=` *linefeed* `|` *carriage return*
  `|` *carriage-return linefeed* `|` *next-line*
  `|` *carriage-return next-line* `|` *line-separator*
comment `::=` **;** all subsequent characters up to a *line-ending*
    or *paragraph-separator*
  `|` *nested-comment*
  `|` **#;** *interlexeme-space datum*
  `|` *shebang-comment*
nested-comment `::=` **#|** *comment-text comment-cont** **|#**
comment-text `::=` character sequence not containing **#|** or **|#**
comment-cont `::=` *nested-comment comment-text*
atmosphere `::=` *whitespace* `|` *comment*
interlexeme-space `::=` atmosphere*

As a special case the characters **#!/** are treated as starting a comment, but only at the beginning of file. These characters are used on Unix systems as an Shebang interpreter directive (`http://en.wikipedia.org/wiki/Shebang_(Unix)`). The Kawa reader skips the entire line. If the last non-whitespace character is \ (backslash) then the following line is also skipped, and so on.

shebang-comment `::=` **#!** *absolute-filename* text up to non-escaped *line-ending*

Whitespace characters are spaces, linefeeds, carriage returns, character tabulations, form feeds, line tabulations, and any other character whose category is Zs, Zl, or Zp. Whitespace is used for improved readability and as necessary to separate lexemes from each other. Whitespace may occur between any two lexemes, but not within a lexeme. Whitespace may also occur inside a string, where it is significant.

The lexical syntax includes several comment forms. In all cases, comments are invisible to Scheme, except that they act as delimiters, so, for example, a comment cannot appear in the middle of an identifier or representation of a number object.

A semicolon (`;`) indicates the start of a line comment. The comment continues to the end of the line on which the semicolon appears.

Another way to indicate a comment is to prefix a *datum* with `#;`, possibly with *interlexeme-space* before the *datum*. The comment consists of the comment prefix `#;` and the *datum* together. This notation is useful for "commenting out" sections of code.

Block comments may be indicated with properly nested `#|` and `|#` pairs.

```
#|
   The FACT procedure computes the factorial of a
   non-negative integer.
|#
(define fact
  (lambda (n)
    ;; base case
    (if (= n 0)
        #;(= n 1)
        1       ; identity of *
        (* n (fact (- n 1))))))
```

7.3.4 Identifiers

identifier ::= *initial subsequent**
        | *peculiar-identifier*
initial ::= *constituent* | *special-initial*
        | *inline-hex-escape*
letter ::= **a** | **b** | **c** | ... | **z**
        | **A** | **B** | **C** | ... | **Z**
constituent ::= *letter*
        | *any character whose Unicode scalar value is greater than 127, and whose category is Lu, Ll, Lt, Lm, Lo, Mn, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co*
special-initial ::= **!** | **$** | **%** | **&** | ***** | **/** | **<** | **=**
        | **>** | **?** | **^** | **_** | **~**
subsequent ::= *initial* | *digit*
        | *any character whose category is Nd, Mc, or Me*
        | *special-subsequent*
digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
oct-digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**
hex-digit ::= *digit*
        | **a** | **A** | **b** | **B** | **c** | **C** | **d** | **D** | **e** | **E** | **f** | **F**
special-subsequent ::= **+** | **-** | **.** | **@**
escape-sequence ::= *inline-hex-escape*
        | “*character-except-x*
        | *multi-escape-sequence*
inline-hex-escape ::= “**x***hex-scalar-value***;**
hex-scalar-value ::= *hex-digit+*
multi-escape-sequence ::= | *symbol-element* *|
symbol-element ::= *any character except* | *or* \
        | *inline-hex-escape* | *mnemonic-escape* | \ |

character-except-x `::=` *any character except* `x`
peculiar-identifier `::=` `+` | `-` | `...` | `->` subsequent*

Most identifiers allowed by other programming languages are also acceptable to Scheme. In general, a sequence of letters, digits, and "extended alphabetic characters" is an identifier when it begins with a character that cannot begin a representation of a number object. In addition, `+`, `-`, and `...` are identifiers, as is a sequence of letters, digits, and extended alphabetic characters that begins with the two–character sequence `->`. Here are some examples of identifiers:

```
lambda         q               soup
list->vector   +               V17a
<=             a34kTMNs        ->-
the-word-recursion-has-many-meanings
```

Extended alphabetic characters may be used within identifiers as if they were letters. The following are extended alphabetic characters:

```
! $ % & * + - . / < = > ? @ ^ _ ~
```

Moreover, all characters whose Unicode scalar values are greater than 127 and whose Unicode category is Lu, Ll, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co can be used within identifiers. In addition, any character can be used within an identifier when specified using an *escape-sequence*. For example, the identifier `H\x65;llo` is the same as the identifier `Hello`.

Kawa supports two additional non-R6RS ways of making identifiers using special characters, both taken from Common Lisp: Any character (except `x`) following a backslash is treated as if it were a *letter*; as is any character between a pair of vertical bars.

Identifiers have two uses within Scheme programs:

- Any identifier may be used as a [variable], page 114, or as a [syntactic keyword], page 120.
- When an identifier appears as or with in [literal], page 113, it is being used to denote a Section 10.1 [symbol], page 160.

In contrast with older versions of Scheme, the syntax distinguishes between upper and lower case in identifiers and in characters specified via their names, but not in numbers, nor in inline hex escapes used in the syntax of identifiers, characters, or strings. The following directives give explicit control over case folding.

`#!fold-case` [Syntax]
`#!no-fold-case` [Syntax]

These directives may appear anywhere comments are permitted and are treated as comments, except that they affect the reading of subsequent data. The `#!fold-case` directive causes the `read` procedure to case-fold (as if by `string-foldcase`) each identifier and character name subsequently read from the same port. The `#!no-fold-case` directive causes the `read` procedure to return to the default, non-folding behavior.

Note that colon `:` is treated specially for Section 7.7 [Colon notation], page 115, in Kawa Scheme, though it is a *special-initial* in standard Scheme (R6RS).

7.3.5 Numbers

((INCOMPLETE))

> *number* `::=` ((TODO))
>   `|` *quantity*
> *decimal* `::=` *digit*`+` *optional-exponent*
>   `|` `.` *digit*`+` *optional-exponent*
>   `|` *digit*`+` `.` *digit*`+` *optional-exponent*
> *optional-exponent* `::=` *empty*
>   `|` *exponent-marker optional-sign digit*`+`
> *exponent-marker* `::=` **e** `|` **s** `|` **f** `|` **d** `|` **l**

The letter used for the exponent in a floating-point literal determines its type:

e Returns a `gnu.math.DFloat` - for example `12e2`. Note this matches the default when there is no *exponent-marker*.

s or **f** Returns a primitive `float` (or `java.lang.Float` when boxed as an object) - for example `12s2` or `12f2`.

d Returns a primitive `double` (or `java.lang.Double` when boxed) - for example `12d2`.

l Returns a `java.math.BigDecimal` - for example `12l2`.

> *optional-sign* `::=` *empty* `|` `+` `|` `-`
> *digit-2* `::=` **0** `|` **1**
> *digit-8* `::=` **0** `|` **1** `|` **2** `|` **3** `|` **4** `|` **5** `|` **6** `|` **7**
> *digit-10* `::=` *digit*
> *digit-16* `::=` *digit-10* `|` **a** `|` **b** `|` **c** `|` **d** `|` **e** `|` **f**

7.4 Datum syntax

The datum syntax describes the syntax of syntactic data in terms of a sequence of *lexeme*s, as defined in the lexical syntax.

The following grammar describes the syntax of syntactic data in terms of various kinds of lexemes defined in the grammar in section "Lexical Syntax":

> *datum* `::=` *defining-datum*
>   `|` *nondefining-datum*
>   `|` *defined-datum*
> *nondefining-datum* `::=` *lexeme-datum*
>   `|` *compound-datum*
>
> *lexeme-datum* `::=` *boolean* `|` *number*
>   `|` *character* `|` *string* `|` *symbol*
> *symbol* `::=` *identifier*
> *compound-datum* `::=` *list* `|` *vector* `|` *uniform-vector* `|` *array-literal* `|` *extended-string-literal* `|` *xml-literal*
> *list* `::=` (*datum**)
>   `|` (*datum* `+` `.` *datum*)
>   `|` *abbreviation*
> *vector* `::=` #(*datum* *)

7.4.1 Datum labels

datum-label `::=` #*indexnum*=
defining-datum `::=` *datum-label* +*nondefining-datum*
defined-datum `::=` #*indexnum*#
indexnum `::=` *digit* +

The lexical syntax `#n=datum` reads the same as *datum*, but also results in *datum* being labelled by *n*, which must a sequence of digits.

The lexical syntax `#n#` serves as a reference to some object labelled by `#n=`; the result is the same object (in the sense of `eq?`) as the `#n=`.

Together, these syntaxes permit the notation of structures with shared or circular substructure.

```
(let ((x (list 'a 'b 'c)))
  (set-cdr! (cddr x) x)
  x)    ⇒ #0=(a b c . #0#)
```

The scope of a datum label is the portion of the outermost datum in which it appears that is to the right of the label. Consequently, a reference `#n#` can occur only after a label `#n=`; it is an error to attempt a forward reference. In addition, it is an error if the reference appears as the labelled object itself (as in `#n=#n#`), because the object labelled by `#n=` is not well defined in this case.

7.4.2 Abbreviations

abbreviation `::=` *r6rs-abbreviation* | *kawa-abbreviation*
r6rs-abbreviation `::=` *abbrev-prefix datum*
abbrev-prefix `::=` **fl** | **`** | **,** | **,@**
        | **#fl** | **#`**
kawa-abbreviation `::=` XXX

The following abbreviations are expanded at read-time:

fl*datum*	means (**quote** *datum*).
`*datum*	means (**quasiquote** *datum*).
,*datum*	means (**unquote** *datum*).
,@*datum*	means (**unquote-splicing** *datum*).
#fl*datum*	means (**syntax** *datum*).
#`*datum*	means (**quasisyntax** *datum*).
#,*datum*	means (**unsyntax** *datum*). This abbreviation is currently only recognized when nested inside an explicit **#`***datum* form, because of a conflict with SRFI-10 named constructors.
#,@*datum*	means (**unsyntax-splicing** *datum*).
*datum1***:***datum2*	means (**$lookup$** *datum1* (**quasiquote** *datum2*)). See Section 7.7 [Colon notation], page 115.

[*expression* ...]
: means **($bracket-list$** *expression* ...**)**.

operator[*expression* ...]
: means **($bracket-apply$** *operator expression* ...**)**.

7.5 Hash-prefixed forms

A number of different special forms are indicated by an initial hash (number) symbols (`#`). Here is a table summarizing them.

Case is ignored for the character followed the `#`. Thus `#x` and `#X` are the same.

#:*keyword*
: Guile-style Section 10.3 [Keywords], page 165, syntax.

#“
: [Character literals], page 202.

#!
: See Section 10.4 [Special named constants], page 165.

#`*datum*
: Equivalent to `(quasisyntax` ***`datum`***`)`. Convenience syntax for syntax-case macros.

#fl*datum*
: Equivalent to `(syntax` ***`datum`***`)`. Convenience syntax for syntax-case macros.

#,*datum*
: Equivalent to `(unsyntax` ***`datum`***`)`. Currently only recognized when inside a `#`` ***`template`*** form. Convenience syntax for syntax-case macros.

#,(*name datum* ...**)**
: Special named constructors. This syntax is deprecated, because it conflicts with `unsyntax`. It is only recognized when *not* in a `#`` ***`template`*** form.

#,@*datum*
: Equivalent to `(unsyntax-splicing` ***`datum`***`)`.

#(
: A vector.

#|
: Start of nested-comment.

#/*regex***/**
: See Section 13.6 [Regular expressions], page 230.

#<
: See Section 20.4 [XML literals], page 339.

#;*datum*
: A datum comment - the *datum* is ignored. (An *interlexeme-space* may appear before the *datum*.)

#*number***=***datum*
: A reference definition, allowing cyclic and shared structure. Equivalent to the *datum*, but also defines an association between the integer *number* and that *datum*, which can be used by a subsequent `#`***`number`***`#` form.

#*number***#**
: A back-reference, allowing cyclic and shared structure.

#*R***a***datum*
: An [array-literals], page 249, for a multi-dimensional array of rank *R*.

#b
: A binary (base-2) number.

#d A decimal (base-10) number.

#e A prefix to treat the following number as exact.

#f
#false The standard boolean false object.

#f*n***(***number* ...**)**
A uniform vector of floating-point numbers. The parameter *n* is a precision, which can be 32 or 64. See Section 14.4 [Uniform vectors], page 239.

#i A prefix to treat the following number as inexact.

#o An octal (base-8) number.

#*base***r** A number in the specified *base* (radix).

#s*n***(***number* ...**)**
A uniform vector of signed integers. The parameter *n* is a precision, which can be 8, 16, 32, or 64. See Section 14.4 [Uniform vectors], page 239.

#t
#true The standard boolean true object.

#u*n***(***number* ...**)**
A uniform vector of unsigned integers. The parameter *n* is a precision, which can be 8, 16, 32, or 64. See Section 14.4 [Uniform vectors], page 239.

#x A hexadecimal (base-16) number.

The follow named constructor forms are supported:

#,(path *path***)**
#,(filepath *path***)**
#,(URI *path***)**
#,(symbol *local-name* [*uri* [*prefix*]]**)**
#,(symbol *local-name namespace***)**
#,(namespace *uri* [*prefix*]**)**
#,(duration *duration***)**

7.6 Primitive expression syntax

expression ::= *literal-expression* | *variable-reference*
  | *procedure-call* | TODO

7.6.1 Literal expressions

literal-expression ::= **(quote** *datum***)**
  | **'** *datum*
  | *constant*
constant ::= *number* | *boolean* | *character* | *string*

`(quote datum)` evaluates to *datum*, which may be any external representation of a Scheme object. This notation is used to include literal constants in Scheme code.

```
(quote a)              ⇒  a
(quote #(a b c))       ⇒  #(a b c)
```

```
(quote (+ 1 2))          ⇒  (+ 1 2)
```

(quote *datum*) may be abbreviated as '*datum*. The two notations are equivalent in all respects.

```
'a                       ⇒  a
'#(a b c)                ⇒  #(a b c)
'()                      ⇒  ()
'(+ 1 2)                 ⇒  (+ 1 2)
'(quote a)               ⇒  (quote a)
''a                      ⇒  (quote a)
```

Numerical constants, string constants, character constants, bytevector constants, and boolean constants evaluate to themselves; they need not be quoted.

```
145932        ⇒  145932
#t            ⇒  #t
"abc"         ⇒  "abc"
```

Note that Section 10.3 [Keywords], page 165, need to be quoted, unlike some other Lisp/Scheme dialect, including Common Lisp, and earlier versions of Kawa. (Kawa currently evaluates a non-quoted keyword as itself, but that will change.)

7.6.2 Variable references

variable-reference ::= *identifier*

An expression consisting of a variable is a variable reference if it is not a macro use (see below). The value of the variable reference is the value stored in the location to which the variable is bound. It is a syntax violation to reference an unbound variable.

The following example assumes the base library has been imported:

```
(define x 28)
x   ⇒  28
```

7.6.3 Procedure calls

procedure-call ::= (*operator* *operand* ...)
operator ::= *expression*
operand ::= *expression*
  | *keyword* *expression*
  | @ *expression*
  | @: *expression*

A procedure call consists of expressions for the procedure to be called and the arguments to be passed to it, with enclosing parentheses. A form in an expression context is a procedure call if *operator* is not an identifier bound as a syntactic keyword.

When a procedure call is evaluated, the operator and operand expressions are evaluated (in an unspecified order) and the resulting procedure is passed the resulting arguments.

```
(+ 3 4)                 ⇒  7
((if #f + *) 3 4)       ⇒  12
```

The syntax *keyword expression* is a *keyword argument*. This is a mechanism for specifying arguments using a name rather than position, and is especially useful for procedures with many optional paramaters. Note that *keyword* must be literal, and cannot be the result from evaluating a non-literal expression. (This is a change from previous versions of Kawa, and is different from Common Lisp and some other Scheme dialects.)

An expression prefixed by `@` or `@:` is a splice argument. The following expression must evaluate to an "argument list" (see Section 11.1 [Application and Arguments Lists], page 166, for details); each element in the argument becomes a separate argument when call the *operator*. (This is very similar to the "spread" operator is EcmaScript 6.)

7.7 Property access using colon notation

The *colon notation* accesses named parts (properties) of a value. It is used to get and set fields, call methods, construct compound symbols, and more. Evaluating the form ***owner**:**property*** evaluates the ***owner*** then it extracts the named ***property*** of the result.

> *property-access-abbreviation* `::=` *property-owner-expression***:***property-name*
> *property-owner-expression* `::=` *expression*
> *property-name* `::=` *identifier* `|` **,***expression*

The *property-name* is usually a literal name, but it can be an unquoted *expression* (i.e. following a `,`), in which case the name is evaluated at run-time. No separators are allowed on either side of the colon.

The input syntax ***owner**:**part*** is translated by the Scheme reader to the internal representation `($lookup$` ***owner*** `(quasiquote` ***part***`))`.

7.7.1 Part lookup rules

Evaluation proceeds as follows. First *property-owner-expression* is evaluated to yield an *owner* object. Evaluating the *property-name* yields a *part* name, which is a simple symbol: Either the literal *identifier*, or the result of evaluating the property-name *expression*. If the *expression* evaluates to a string, it is converted to a symbol, as if using `string->symbol`.

- If the *owner* implements `gnu.mapping.HasNamedParts`, then the result is that of invoking the `get` method of the *owner* with the *part* name as a parameter.

  As a special case of this rule, if *owner* is a `gnu.mapping.Namespace`, then the result is the Section 10.2 [Namespaces], page 161.
- If *owner* is a `java.lang.Class` or a `gnu.bytecode.ObjectType`, the result is the static member named *part* (i.e. a static field, method, or member class).
- If *owner* is a `java.lang.Package` object, we get the member class or sub-package named *part*.
- Otherwise, we look for a named member (instance member or field).

  Note you can't use colon notation to invoke instance methods of a `Class`, because it will match a previous rule. For example if you want to invoke the `getDeclaredMethod` method of the `java.util.List` , you can't write `(java.util.List:getDeclaredMethod` because that will look for a static method in `java.util.List`. Instead, use the `invoke` or `invoke-sttic` method. For example: `(invoke java.util.List 'getDeclaredMethod)`.

If the colon form is on the left-hand-side of an assignment (`set!`), then the named part is modified as appropriate.

7.7.2 Specific cases

Some of these are deprecated; more compact and readable forms are usually preferred.

7.7.2.1 Invoking methods

(*instance*:*method-name arg* ...)
(*class*:*method-name instance arg* ...)
(*class*:*method-name arg* ...)
(***:***method-name instance arg* ...)

For details see Section 19.9 [Method operations], page 320.

7.7.2.2 Accessing fields

class:*field-name*
instance:*field-name*
(*prefix*:.*field-name instance*)

For details see Section 19.11 [Field operations], page 327.

7.7.2.3 Type literal

(*type*:**<>**)

Returns the *type*. Deprecated; usually you can just write:

```
type
```

7.7.2.4 Type cast

(*type*:**@** *expression*)

Performs a cast. Deprecated; usually you can just write:

```
->type
```

7.7.2.5 Type test

(*type*:**instanceof?** *expression*)

Deprecated; usually you can just write:

```
(type? expression)
```

7.7.2.6 New object construction

(*type*:**new** *arg* ...)

Deprecated; usually you can just write:

(*type arg* ...)

7.7.2.7 Getting array length

expression:**length**
(*expression*:**.length**)

7.8 Programs and Bodies

Program units

A *program-unit* consists of a sequence of definitions and expressions.

> *program-unit* `::=` *library-definition* `+` [*statements*]
>   `|` *statements*
> *statements* `::=` *statement* `+`
> *statement* `::=` *definition* `|` *expression* `|` **(begin** *statement* `*` **)**

Typically a *program-unit* corresponds to a single source file (i.e.a named file in the file system). Evaluating a *program-unit* first requires the Kawa processor to analyze the whole *program-unit* to determine which names are defined by the definitions, and then evaluates each *statement* in order in the context of the defined names. The value of an *expression* is normally discarded, but may be printed out instead, depending on the evaluating context.

The read-eval-print-loop (REPL) reads one or more lines until it gets a valid *program-unit*, and evaluates it as above, except that the values of expressions are printed to the console (as if using the `display` function). Then the REPL reads and evaluates another *program-unit*, and so on. A definition in an earlier *program-unit* is remembered and is visible in a later *program-unit* unles it is overridden.

A comment in the first 2 lines of a source file may contain an encoding specification. This can be used to tell the reader what kind of character set encoding is used for the file. This only works for a character encoding that is compatible with ASCII (in the sense that if the high-order bit is clear then it's an ASCII character), and that are no non-ASCI characters in the lines upto and including the encoding specification. A basic example is:

```
;; -*- coding: utf-8 -*-
```

In general any string that matches the following regular expression works:

```
coding[:=]\s*([-a-zA-Z0-9]+)
```

Libraries

A *program-unit* may contain *library-definitions*. In addition, any *statements* in *program-unit* comprise an *implicit library*, in that it can be given a name, and referenced from other libraries. Certain names defined in the *program-unit* can be exported, and then they can be imported by other libraries. For more information see Section 19.5 [Module classes], page 305.

It is recommended but not required that:

- There should be at most one *library-definition* in a *program-unit*.
- The *library-name* of the *library-definition* should match the name of the source file. For example:

  ```
  (define-library (foo bar) ...)
  ```

  should be in a file named `foo/bar.scm`.
- If there is a *library-definition*, there should be no extra *statements* - i.e no implicit library definition. (It is disallowed to `export` any definitions from the implicit library if there is also a *library-definition*.)

Following these recommendations makes it easier to locate and organize libraries. However, having multiple libraries in a single *program-unit* is occasionally useful for source distribution and for testing.

Bodies

The *body* of a `lambda`, `let`, `let*`, `let-values`, `let*-values`, `letrec`, or `letrec*` expression, or that of a definition with a body consists of zero or more definitions or expressions followed by a final expression. (Standard Scheme requires that all definitions precede all expressions.)

body `::=` *statement* *

Each identifier defined by a definition is local to the *body*. That is, the identifier is bound, and the region of the binding is the entire *body*. Example:

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))
⇒ 45
```

When `begin`, `let-syntax`, or `letrec-syntax` forms occur in a body prior to the first expression, they are spliced into the body. Some or all of the body, including portions wrapped in `begin`, `let-syntax`, or `letrec-syntax` forms, may be specified by a macro use.

An expanded *body* containing variable definitions can be converted into an equivalent `letrec*` expression. (If there is a definition following expressions you may need to convert the expressions to dummy definitions.) For example, the `let` expression in the above example is equivalent to

```
(let ((x 5))
  (letrec* ((foo (lambda (y) (bar x y)))
            (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

7.9 Syntax and conditional compilation

Feature testing

`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*)
  | (**library** *library-name*)
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.

The implementation has a set of feature identifiers which are "present", as well as a set of libraries which can be imported. The value of a *feature-requirement* is determined by replacing each *feature-identifier* by `#t` if it is present (and `#f` otherwise); replacing `(library library-name)` by `#t` if *library-name* is importable (and `#f` otherwise); and then evaluating the resulting expression as a Scheme boolean expression under the normal interpretation of `and`, `or`, and `not`.

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.

You can use `java-6`, `java-7`, `java-8`, or `java-9` to check if the underlying Java is a specific version or newer. For example the name `java-7` matches for either Java 7, Java 8, or newer, as reported by `System` property `"java.version"`.

You can use `class-exists:ClassName` to check if `ClassName` exists at compile-time. The identifier `class-exists:org.example.MyClass` is roughly equivalent to the test `(library (org example MyClass))`. (The latter has some special handling for `(srfi ...)` as well as builtin Kawa classes.)

The feature `in-http-server` is defined in a Section 20.6 [Self-configuring page scripts], page 343, and more specifically `in-servlet` in a Section 20.7 [Servlets], page 346.

`features` [Procedure]

Returns a list of feature identifiers which `cond-expand` treats as true. This not a complete list - for example `class-exists:ClassName` feature identifiers are not included. It is an error to modify this list. Here is an example of what `features` might return:

```
(features)  ⇒
(complex exact-complex full-unicode java-7 java-6 kawa
 ratios srfi-0 srfi-4 srfi-6 srfi-8 srfi-9 srfi-11
 srfi-16 srfi-17 srfi-23 srfi-25 srfi-26 srfi-28 srfi-30
 srfi-39 string-normalize-unicode threads)
```

File inclusion

`include` *path*$^+$ [Syntax]
`include-relative` *path*$^+$ [Syntax]

`include-ci` *path*$^+$ [Syntax]

These take one or more path names expressed as string literals, find corresponding files, read the contents of the files in the specified order as if by repeated applications of `read`, and effectively replace the `include` with a `begin` form containing what was read from the files.

You can control the search path used for `include` by setting the `kawa.include.path` property. For example:

```
$ kawa -Dkawa.include.path="|:/opt/kawa-includes"
```

The special `"|"` path element means to search relative to the directory containing the including source file. The default search path is `"|:."` which means to first search the directory containing the including source file, and then search the directory specified by `(current-path)`.

The search path for `include-relative` prepends `"|"` before the search path used by `include`, so it always searches first the directory containing the including source file. Note that if the default search path is used then `include` and `include-relative` are equivalent; there is only a difference if the `kawa.include.path` property changes the default.

Using `include-ci` is like `include`, except that it reads each file as if it began with the `#!fold-case` directive.

7.10 Macros

Libraries and top–level programs can define and use new kinds of derived expressions and definitions called *syntactic abstractions* or *macros*. A syntactic abstraction is created by binding a keyword to a *macro transformer* or, simply, *transformer*.

The transformer determines how a use of the macro (called a *macro use*) is transcribed into a more primitive form.

Most macro uses have the form:

```
(keyword datum ...)
```

where *keyword* is an identifier that uniquely determines the kind of form. This identifier is called the *syntactic keyword*, or simply *keyword*. The number of *datum*s and the syntax of each depends on the syntactic abstraction.

Macro uses can also take the form of improper lists, singleton identifiers, or `set!` forms, where the second subform of the `set!` is the keyword:

```
(keyword datum ... . datum)
keyword
(set! keyword datum)
```

The `define-syntax`, `let-syntax` and `letrec-syntax` forms create bindings for keywords, associate them with macro transformers, and control the scope within which they are visible.

The `syntax-rules` and `identifier-syntax` forms create transformers via a pattern language. Moreover, the `syntax-case` form allows creating transformers via arbitrary Scheme code.

Keywords occupy the same name space as variables. That is, within the same scope, an identifier can be bound as a variable or keyword, or neither, but not both, and local bindings of either kind may shadow other bindings of either kind.

Macros defined using `syntax-rules` and `identifier-syntax` are "hygienic" and "referentially transparent" and thus preserve Scheme's lexical scoping.

- If a macro transformer inserts a binding for an identifier (variable or keyword) not appearing in the macro use, the identifier is in effect renamed throughout its scope to avoid conflicts with other identifiers.
- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

Macros defined using the `syntax-case` facility are also hygienic unless `datum->syntax` is used.

Kawa supports most of the `syntax-case` feature.

Syntax definitions are valid wherever definitions are. They have the following form:

`define-syntax` *keyword* *`transformer-spec`* [Syntax]

The *keyword* is a identifier, and *transformer-spec* is a function that maps syntax forms to syntax forms, usually an instance of `syntax-rules`. If the `define-syntax` occurs at the top level, then the top-level syntactic environment is extended by binding the *keyword* to the specified transformer, but existing references to any top-level binding for *keyword* remain unchanged. Otherwise, it is an *internal syntax definition*, and is local to the *body* in which it is defined.

```
(let ((x 1) (y 2))
   (define-syntax swap!
     (syntax-rules ()
       ((swap! a b)
        (let ((tmp a))
          (set! a b)
          (set! b tmp)))))
   (swap! x y)
   (list x y))  ⇒ (2 1)
```

Macros can expand into definitions in any context that permits them. However, it is an error for a definition to define an identifier whose binding has to be known in order to determine the meaning of the definition itself, or of any preceding definition that belongs to the same group of internal definitions.

`define-syntax-case` *name* (*literals*) (*pattern expr*) ... [Syntax]

A convenience macro to make it easy to define `syntax-case`-style macros. Defines a macro with the given *name* and list of *literals*. Each *pattern* has the form of a `syntax-rules`-style pattern, and it is matched against the macro invocation syntax form. When a match is found, the corresponding *expr* is evaluated. It must evaluate to a syntax form, which replaces the macro invocation.

```
(define-syntax-case macro-name (literals)
  (pat1 result1)
```

```
    (pat2 result2))
```

is equivalent to:

```
(define-syntax macro-name
  (lambda (form)
    (syntax-case form (literals)
      (pat1 result1)
      (pat2 result2))))
```

`define-macro` (*name lambda-list*) *form ...* [Syntax]

This form is deprecated. Functionally equivalent to `defmacro`.

`defmacro` *name lambda-list form ...* [Syntax]

This form is deprecated. Instead of

```
(defmacro (name ...)
  (let ... `(... ,exp ...)))
```

you should probably do:

```
(define-syntax-case name ()
  ((_ ...) (let #`(... #,exp ...))))
```

and instead of

```
(defmacro (name ... var ...) `(... var ...))
```

you should probably do:

```
(define-syntax-case name ()
  ((_ ... var ...) #`(... var ...))
```

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).

`gentemp` [Procedure]

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.)

`expand` *form* [Procedure]

The result of evaluating *form* is treated as a Scheme expression, syntax-expanded to internal form, and then converted back to (roughly) the equivalent expanded Scheme form.

This can be useful for debugging macros.

To access this function, you must first `(require 'syntax-utils)`.

```
(require 'syntax-utils)
(expand '(cond ((> x y) 0) (else 1))) ⇒ (if (> x y) 0 1)
```

7.10.1 Pattern language

A *transformer-spec* is an expression that evaluates to a transformer procedure, which takes an input form and returns a resulting form. You can do general macro-time compilation with such a procedure, commonly using `syntax-case` (which is documented in the R6RS library specification). However, when possible it is better to use the simpler pattern language of `syntax-rules`:

> *transformer-spec* `::=`
>   (**syntax-rules** (*tr-literal* *) *syntax-rule* *)
> | (**syntax-rules** *ellipsis* (*tr-literal* *) *syntax-rule* *)
> | *expression*
> *syntax-rule* `::=` (*list-pattern syntax-template*)
> *tr-literal* `::=` *identifier*
> *ellipsis* `::=` *identifier*

An instance of `syntax-rules` produces a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by `syntax-rules` is matched against the patterns contained in the *syntax-rule*s beginning with the leftmost syntax rule . When a match is found, the macro use is transcribed hygienically according to the template. The optional *ellipsis* species a symbol used to indicate repetition; it defaults to `...` (3 periods).

> *syntax-pattern* `::=`
>   *identifier* | *constant* | *list-pattern* | *vector-pattern*
> *list-pattern* `::=` (*syntax-pattern* *)
> | (*syntax-pattern* *syntax-pattern* * **.** *syntax-pattern*)
> | (*syntax-pattern* * *syntax-pattern ellipsis* *syntax-pattern* *)
> | (*syntax-pattern* * *syntax-pattern ellipsis* *syntax-pattern* * **.** *syntax-pattern*)
> *vector-pattern* `::=` #(*syntax-pattern* *)
> | #(*syntax-pattern* * *syntax-pattern ellipsis* *syntax-pattern* *)

An identifier appearing within a pattern can be an underscore (`_`), a literal identifier listed in the list of *tr-literal*s, or the *ellipsis*. All other identifiers appearing within a pattern are pattern variables.

The outer *syntax-list* of the pattern in a *syntax-rule* must start with an identifier. It is not involved in the matching and is considered neither a pattern variable nor a literal identifier.

Pattern variables match arbitrary input elements and are used to refer to elements of the input in the template. It is an error for the same pattern variable to appear more than once in a *syntax-pattern*.

Underscores also match arbitrary input elements but are not pattern variables and so cannot be used to refer to those elements. If an underscore appears in the literals list, then that takes precedence and underscores in the pattern match as literals. Multiple underscores can appear in a *syntax-pattern*.

Identifiers that appear in `(`*`tr-literal`*`*)` are interpreted as literal identifiers to be matched against corresponding elements of the input. An element in the input matches a literal identifier if and only if it is an identifier and either both its occurrence in the macro expression and its occurrence in the macro definition have the same lexical binding, or the two identifiers are the same and both have no lexical binding.

A subpattern followed by ellipsis can match zero or more elements of the input, unless ellipsis appears in the literals, in which case it is matched as a literal.

More formally, an input expression E matches a pattern P if and only if:

- P is an underscore (`_`); or
- P is a non-literal identifier; or
- P is a literal identifier and E is an identifier with the same binding; or
- P is a list $(P_1 \ldots P_n)$ and E is a list of n elements that match P_1 through P_n, respectively; or
- P is an improper list $(P_1 \ldots P_n \mathbin{.} P_{n+1})$ and E is a list or improper list of n or more elements that match P_1 through P_n, respectively, and whose nth tail matches P_{n+1}; or
- P is of the form $(P_1 \ldots P_k\ P_e\ \mathit{ellipsis}\ P_{k+1} \ldots P_{k+l})$ where E is a proper list of n elements, the first k of which match P_1 through P_k, respectively, whose next n-k-l elements each match P_e, and whose remaining l elements match P_{k+1} through P_{k+l}; or
- P is of the form $(P_1 \ldots P_k\ P_e\ \mathit{ellipsis}\ P_{k+1} \ldots P_{k+l} \mathbin{.} P_x)$ where E is a list or improper list of n elements, the first k of which match P_1 through P_k, whose next n-k-l elements each match P_e, and whose remaining l elements match P_{k+1} through P_{k+l}, and whose nth and final `cdr` matches P_x; or
- P is a vector of the form $\#(P_1 \ldots P_n)$ and E is a vector of n elements that match P_1 through P_n; or
- P is of the form $\#(P_1 \ldots P_k\ P_e\ \mathit{ellipsis}\ P_{k+1} \ldots P_{k+l})$ where E is a vector of n elements the first k of which match P_1 through P_k, whose next n-k-l elements each match P_e, and whose remaining l elements match P_{k+1} through P_{k+l}; or
- P is a constant and E is equal to P in the sense of the `equal?` procedure.

It is an error to use a macro keyword, within the scope of its binding, in an expression that does not match any of the patterns.

> *syntax-template* `::=` *identifier* `|` *constant*
>   `|` (*template-element* *)
>   `|` (*template-element* *template-element* * . *syntax-template*)
>   `|` (*ellipsis* *syntax-template*)
> *template-element* `::=` *syntax-template* [*ellipsis*]

When a macro use is transcribed according to the template of the matching *syntax-rule*, pattern variables that occur in the template are replaced by the elements they match in the input. Pattern variables that occur in subpatterns followed by one or more instances of the identifier *ellipsis* are allowed only in subtemplates that are followed by as many instances of *ellipsis* . They are replaced in the output by all of the elements they match in the input, distributed as indicated. It is an error if the output cannot be built up as specified.

Identifiers that appear in the template but are not pattern variables or the identifier *ellipsis* are inserted into the output as literal identifiers. If a literal identifier is inserted as a free identifier then it refers to the binding of that identifier within whose scope the instance of `syntax-rules` appears. If a literal identifier is inserted as a bound identifier then it is in effect renamed to prevent inadvertent captures of free identifiers.

A template of the form (*ellipsis template*) is identical to *template*, except that *ellipses* within the template have no special meaning. That is, any *ellipses* contained within *tem-*

plate are treated as ordinary identifiers. In particular, the template (*ellipsis ellipsis*) produces a single *ellipsis*. This allows syntactic abstractions to expand into code containing ellipses.

```
(define-syntax be-like-begin
  (syntax-rules ()
    ((be-like-begin name)
     (define-syntax name
       (syntax-rules ()
         ((name expr (... ...))
          (begin expr (... ...))))))))

(be-like-begin sequence)
(sequence 1 2 3 4) ⇒ 4
```

7.10.2 Identifier predicates

identifier? *obj* [Procedure]

Return `#t` if *obj* is an identifier, i.e., a syntax object representing an identifier, and `#f` otherwise.

The `identifier?` procedure is often used within a fender to verify that certain subforms of an input form are identifiers, as in the definition of `rec`, which creates self–contained recursive objects, below.

```
(define-syntax rec
  (lambda (x)
    (syntax-case x ()
      ((_ x e)
       (identifier? #'x)
       #'(letrec ((x e)) x)))))

(map (rec fact
       (lambda (n)
         (if (= n 0)
             1
             (* n (fact (- n 1))))))
     '(1 2 3 4 5))    ⇒ (1 2 6 24 120)

(rec 5 (lambda (x) x))  ⇒ exception
```

The procedures `bound-identifier=?` and `free-identifier=?` each take two identifier arguments and return `#t` if their arguments are equivalent and `#f` otherwise. These predicates are used to compare identifiers according to their *intended use* as free references or bound identifiers in a given context.

bound-identifier=? *id_1 id_2* [Procedure]

id_1 and *id_2* must be identifiers.

The procedure `bound-identifier=?` returns `#t` if a binding for one would capture a reference to the other in the output of the transformer, assuming that the reference appears within the scope of the binding, and `#f` otherwise.

In general, two identifiers are `bound-identifier=?` only if both are present in the original program or both are introduced by the same transformer application (perhaps implicitly, see `datum->syntax`).

The `bound-identifier=?` procedure can be used for detecting duplicate identifiers in a binding construct or for other preprocessing of a binding construct that requires detecting instances of the bound identifiers.

`free-identifier=?` *id_1 id_2* [Procedure]

id_1 and *id_2* must be identifiers.

The `free-identifier=?` procedure returns `#t` if and only if the two identifiers would resolve to the same binding if both were to appear in the output of a transformer outside of any bindings inserted by the transformer. (If neither of two like–named identifiers resolves to a binding, i.e., both are unbound, they are considered to resolve to the same binding.)

Operationally, two identifiers are considered equivalent by `free-identifier=?` if and only the topmost matching substitution for each maps to the same binding or the identifiers have the same name and no matching substitution.

The `syntax-case` and `syntax-rules` forms internally use `free-identifier=?` to compare identifiers listed in the literals list against input identifiers.

```
(let ((fred 17))
  (define-syntax a
    (lambda (x)
      (syntax-case x ()
        ((_ id) #'(b id fred)))))
  (define-syntax b
    (lambda (x)
      (syntax-case x ()
        ((_ id1 id2)
         #`(list
             #,(free-identifier=? #'id1 #'id2)
             #,(bound-identifier=? #'id1 #'id2))))))
  (a fred))
    ⇒ (#t #f)
```

The following definition of unnamed `let` uses `bound-identifier=?` to detect duplicate identifiers.

```
(define-syntax let
  (lambda (x)
    (define unique-ids?
      (lambda (ls)
        (or (null? ls)
            (and (let notmem? ((x (car ls)) (ls (cdr ls)))
                   (or (null? ls)
                       (and (not (bound-identifier=? x (car ls)))
                            (notmem? x (cdr ls)))))
                 (unique-ids? (cdr ls))))))
    (syntax-case x ()
      ((_ ((i v) ...) e1 e2 ...)
       (unique-ids? #'(i ...))
       #'((lambda (i ...) e1 e2 ...) v ...)))))
```

The argument `#'(i ...)` to `unique-ids?` is guaranteed to be a list by the rules given in the description of `syntax` above.

With this definition of `let`:

```
(let ((a 3) (a 4)) (+ a a))     ⇒ syntax error
```

However,

```
(let-syntax
  ((dolet (lambda (x)
            (syntax-case x ()
              ((_ b)
               #'(let ((a 3) (b 4)) (+ a b)))))))
  (dolet a))
⇒ 7
```

since the identifier `a` introduced by `dolet` and the identifier `a` extracted from the input form are not `bound-identifier=?`.

Rather than including `else` in the literals list as before, this version of `case` explicitly tests for `else` using `free-identifier=?`.

```
(define-syntax case
  (lambda (x)
    (syntax-case x ()
      ((_ e0 ((k ...) e1 e2 ...) ...
          (else-key else-e1 else-e2 ...))
       (and (identifier? #'else-key)
            (free-identifier=? #'else-key #'else))
       #'(let ((t e0))
           (cond
            ((memv t '(k ...)) e1 e2 ...)
            ...
            (else else-e1 else-e2 ...))))
      ((_ e0 ((ka ...) e1a e2a ...)
          ((kb ...) e1b e2b ...) ...)
       #'(let ((t e0))
           (cond
            ((memv t '(ka ...)) e1a e2a ...)
            ((memv t '(kb ...)) e1b e2b ...)
            ...))))))
```

With either definition of `case`, `else` is not recognized as an auxiliary keyword if an enclosing lexical binding for `else` exists. For example,

```
(let ((else #f))
  (case 0 (else (write "oops"))))     ⇒ syntax error
```

since `else` is bound lexically and is therefore not the same `else` that appears in the definition of `case`.

7.10.3 Syntax-object and datum conversions

`syntax->datum` *syntax-object* [Procedure]
`syntax-object->datum` *syntax-object* [Deprecated procedure]

Strip all syntactic information from a syntax object and returns the corresponding Scheme datum.

Identifiers stripped in this manner are converted to their symbolic names, which can then be compared with `eq?`. Thus, a predicate `symbolic-identifier=?` might be defined as follows.

```
(define symbolic-identifier=?
  (lambda (x y)
    (eq? (syntax->datum x)
         (syntax->datum y))))
```

`datum->syntax` *template-id datum* [*srcloc*] [Procedure]
`datum->syntax-object` *template-id datum* [Deprecated procedure]

template-id must be a template identifier and *datum* should be a datum value.

The `datum->syntax` procedure returns a syntax-object representation of *datum* that contains the same contextual information as *template-id*, with the effect that the syntax object behaves as if it were introduced into the code when *template-id* was introduced.

If *srcloc* is specified (and neither `#f` or `#!null`), it specifies the file position (including line number) for the result. In that case it should be a syntax object representing a list; otherwise it is currently ignored, though future extensions may support other ways of specifying the position.

The `datum->syntax` procedure allows a transformer to "bend" lexical scoping rules by creating *implicit identifiers* that behave as if they were present in the input form, thus permitting the definition of macros that introduce visible bindings for or references to identifiers that do not appear explicitly in the input form. For example, the following defines a `loop` expression that uses this controlled form of identifier capture to bind the variable `break` to an escape procedure within the loop body. (The derived `with-syntax` form is like `let` but binds pattern variables.)

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      ((k e ...)
       (with-syntax
           ((break (datum->syntax #'k 'break)))
         #'(call-with-current-continuation
             (lambda (break)
               (let f () e ... (f))))))))))

(let ((n 3) (ls '()))
  (loop
    (if (= n 0) (break ls))
    (set! ls (cons 'a ls))
```

```
        (set! n (- n 1))))
    ⇒ (a a a)
```

Were `loop` to be defined as:

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      ((_ e ...)
       #'(call-with-current-continuation
           (lambda (break)
             (let f () e ... (f))))))))
```

the variable `break` would not be visible in `e ...`.

The datum argument *datum* may also represent an arbitrary Scheme form, as demonstrated by the following definition of `include`.

```
(define-syntax include
  (lambda (x)
    (define read-file
      (lambda (fn k)
        (let ((p (open-file-input-port fn)))
          (let f ((x (get-datum p)))
            (if (eof-object? x)
                (begin (close-port p) '())
                (cons (datum->syntax k x)
                      (f (get-datum p))))))))
    (syntax-case x ()
      ((k filename)
       (let ((fn (syntax->datum #'filename)))
         (with-syntax (((exp ...)
                        (read-file fn #'k)))
           #'(begin exp ...)))))))
```

`(include "filename")` expands into a `begin` expression containing the forms found in the file named by `"filename"`. For example, if the file `flib.ss` contains:

```
(define f (lambda (x) (g (* x x))))
```

and the file `glib.ss` contains:

```
(define g (lambda (x) (+ x x)))
```

the expression:

```
(let ()
  (include "flib.ss")
  (include "glib.ss")
  (f 5))
```

evaluates to `50`.

The definition of `include` uses `datum->syntax` to convert the objects read from the file into syntax objects in the proper lexical context, so that identifier references and definitions within those expressions are scoped where the `include` form appears.

Using `datum->syntax`, it is even possible to break hygiene entirely and write macros in the style of old Lisp macros. The `lisp-transformer` procedure defined below creates a transformer that converts its input into a datum, calls the programmer's procedure on this datum, and converts the result back into a syntax object scoped where the original macro use appeared.

```
(define lisp-transformer
  (lambda (p)
    (lambda (x)
      (syntax-case x ()
        ((kwd . rest)
         (datum->syntax #'kwd
           (p (syntax->datum x))))))))
```

7.10.4 Signaling errors in macro transformers

`syntax-error` *message args** [Syntax]

The *message* and *args* are treated similary as for the `error` procedure. However, the error is reported when the `syntax-error` is expanded. This can be used as a `syntax-rules` template for a pattern that is an invalid use of the macro, which can provide more descriptive error messages. The *message* should be a string literal, and the *args* arbitrary (non-evalualted) expressions providing additional information.

```
(define-syntax simple-let
  (syntax-rules ()
    ((_ (head ... ((x . y) val) . tail)
       body1 body2 ...)
     (syntax-error "expected an identifier but got" (x . y)))
    ((_ ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))))
```

`report-syntax-error` *location message* [Procedure]

This is a procedure that can be called at macro-expansion time by a syntax transformer function. (In contrast `syntax-error` is a syntax form used in the expansion result.) The *message* is reported as a compile-time error message. The *location* is used for the source location (file name and line/column numbers): In general it can be a `SourceLocator` value; most commonly it is a syntax object for a sub-list of the input form that is erroneous. The value returned by `report-syntax-error` is an instance of `ErrorExp`, which supresses further compilation.

```
(define-syntax if
  (lambda (x)
    (syntax-case x ()
                 ((_ test then)
                  (make-if-exp #'test #'then #!null))
                 ((_ test then else)
                  (make-if-exp #'test #'then #'else))
                 ((_ e1 e2 e3 . rest)
                  (report-syntax-error #'rest
                   "too many expressions for 'if'"))
                 ((_ . rest)
                  (report-syntax-error #'rest
                   "too few expressions for 'if'")))))
```

In the above example, one could use the source form `x` for the location, but using `#'rest` is more accurate. Note that the following is incorrect, because `e1` might not be a pair, in which case we don't have location information for it (due to a Kawa limitation):

```
          (syntax-case x ()
                            ...
                            ((_ e1)
                             (report-syntax-error
                              #'e1 ;; poor location specifier
                              "too few expressions for 'if'")))))
```

7.10.5 Convenience forms

`with-syntax` ((*pattern expression*) . . .) *body* [Syntax]

The `with-syntax` form is used to bind pattern variables, just as `let` is used to bind variables. This allows a transformer to construct its output in separate pieces, then put the pieces together.

Each *pattern* is identical in form to a `syntax-case` pattern. The value of each *expression* is computed and destructured according to the corresponding *pattern*, and pattern variables within the *pattern* are bound as with `syntax-case` to the corresponding portions of the value within *body*.

The `with-syntax` form may be defined in terms of `syntax-case` as follows.

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      ((_ ((p e0) ...) e1 e2 ...)
       (syntax (syntax-case (list e0 ...) ()
                 ((p ...) (let () e1 e2 ...))))))))
```

The following definition of `cond` demonstrates the use of `with-syntax` to support transformers that employ recursion internally to construct their output. It handles all `cond` clause variations and takes care to produce one-armed `if` expressions where appropriate.

```
(define-syntax cond
  (lambda (x)
    (syntax-case x ()
      ((_ c1 c2 ...)
       (let f ((c1 #'c1) (c2* #'(c2 ...)))
         (syntax-case c2* ()
           (()
            (syntax-case c1 (else =>)
             (((else e1 e2 ...) #'(begin e1 e2 ...))
              ((e0) #'e0)
```

```
                          ((e0 => e1)
                           #'(let ((t e0)) (if t (e1 t))))
                          ((e0 e1 e2 ...)
                           #'(if e0 (begin e1 e2 ...)))))
                      ((c2 c3 ...)
                       (with-syntax ((rest (f #'c2 #'(c3 ...))))
                         (syntax-case c1 (=>)
                           ((e0) #'(let ((t e0)) (if t t rest)))
                           ((e0 => e1)
                            #'(let ((t e0)) (if t (e1 t) rest)))
                           ((e0 e1 e2 ...)
                            #'(if e0
                                  (begin e1 e2 ...)
                                  rest)))))))))))
```

`quasisyntax` *`template`* [Syntax]
`unsyntax` [Auxiliary Syntax]
`unsyntax-splicing` [Auxiliary Syntax]

The `quasisyntax` form is similar to `syntax`, but it allows parts of the quoted text to be evaluated, in a manner similar to the operation of `quasiquote`.

Within a `quasisyntax` *template*, subforms of `unsyntax` and `unsyntax-splicing` forms are evaluated, and everything else is treated as ordinary template material, as with `syntax`.

The value of each `unsyntax` subform is inserted into the output in place of the `unsyntax` form, while the value of each `unsyntax-splicing` subform is spliced into the surrounding list or vector structure. Uses of `unsyntax` and `unsyntax-splicing` are valid only within `quasisyntax` expressions.

A `quasisyntax` expression may be nested, with each `quasisyntax` introducing a new level of syntax quotation and each `unsyntax` or `unsyntax-splicing` taking away a level of quotation. An expression nested within *n* `quasisyntax` expressions must be within *n unsyntax* or `unsyntax-splicing` expressions to be evaluated.

As noted in *abbreviation*, `#`‘*`template`* is equivalent to `(quasisyntax` *`template`*`)`, `#,`*`template`* is equivalent to `(unsyntax` *`template`*`)`, and `#,@`*`template`* is equivalent to `(unsyntax-splicing` *`template`*`)`. *Note* that for backwards compatibility, you should only use `#,`*`template`* inside a literal `#`‘*`template`* form.

The `quasisyntax` keyword can be used in place of `with-syntax` in many cases. For example, the definition of `case` shown under the description of `with-syntax` above can be rewritten using `quasisyntax` as follows.

```
(define-syntax case
  (lambda (x)
    (syntax-case x ()
      ((_ e c1 c2 ...)
       #`(let ((t e))
           #,(let f ((c1 #'c1) (cmore #'(c2 ...)))
               (if (null? cmore)
                   (syntax-case c1 (else)
                     ((else e1 e2 ...)
                      #'(begin e1 e2 ...))
                     (((k ...) e1 e2 ...)
                      #'(if (memv t '(k ...))
                            (begin e1 e2 ...))])
                   (syntax-case c1 ()
                     (((k ...) e1 e2 ...)
                      #`(if (memv t '(k ...))
                            (begin e1 e2 ...)
                            #,(f (car cmore)
                                  (cdr cmore)))))))))))
```

Note: Any `syntax-rules` form can be expressed with `syntax-case` by making the `lambda` expression and `syntax` expressions explicit, and `syntax-rules` may be defined in terms of `syntax-case` as follows.

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (lit ...) ((k . p) t) ...)
       (for-all identifier? #'(lit ... k ...))
       #'(lambda (x)
           (syntax-case x (lit ...)
             ((_ . p) #'t) ...))))))
```

7.11 Named quasi-literals

Traditional Scheme has only a few kinds of values, and thus only a few builtin kinds of literals. Modern Scheme allows defining new types, so it is desirable to have a mechanism for defining literal values for the new types.

Consider the `[URI], page 272` type. You can create a new instance of a `URI` using a constructor function:

```
(URI "http://example.com/")
```

This isn't too bad, though the double-quote characters are an ugly distraction. However, if you need to construct the string it gets messy:

```
(URI (string-append base-uri "icon.png"))
```

Instead use can write:

```
&URI{http://example.com/}
```

or:

```
&URI{&[base-uri]icon.png}
```

This syntax is translated by the Scheme reader to the more familiar but more verbose equivalent forms:

```
($construct$:URI "http://example.com/")
($construct$:URI $<<$ base-uri $>>$ "icon.png")
```

So for this to work there just needs to be a definition of `$construct$:URI`, usually a macro. Normal scope rules apply; typically you'd define `$construct$:URI` in a module.

The names `$<<$` and `$>>$` are bound to unique zero-length strings. They are used to allow the implementation of `$construct$:URI` to determine which arguments are literal and which come from escaped expressions.

If you want to define your own `$construct$:`*`tag`*, or to read motivation and details, see the SRFI 108 (`http://srfi.schemers.org/srfi-108/srfi-108.html`) specification.

extended-datum-literal `::=`
  & *cname* { [*initial-ignored*] *named-literal-part* * }
  | **&** *cname* [*expression* *]{ [*initial-ignored*] *named-literal-part* * }
cname `::=` *identifier*
named-literal-part `::=`
  any character except **&**, { *or* }
  | { *named-literal-part* + }
  | *char-ref*
  | *entity-ref*
  | *special-escape*
  | *enclosed-part*
  | *extended-datum-literal*

8 Program structure

See [program units], page 117, for some notes on structure of an entire source file.

8.1 Boolean values

The standard boolean objects for true and false are written as `#t` and `#f`. Alternatively, they may be written `#true` and `#false`, respectively.

boolean `::=` **#t** | **#f** | **#true** | **#false**

test-expression `::=` *expression*

What really matters, though, are the objects that the Scheme conditional expressions (`if`, `cond`, `and`, `or`, `when`, `unless`, `do`) treat as true or false. The phrase "a true value" (or sometimes just "true") means any object treated as true by the conditional expressions, and the phrase "a false value" (or "false") means any object treated as false by the conditional expressions. In this document, *test-expression* is an expression that is evaluated, but we only care about whether the result is a true or a false value.

Of all the standard Scheme values, only `#f` counts as false in conditional expressions. All other Scheme values, including `#t`, count as true. A *test-expression* is an expression evaluated in this manner for whether it is true or false.

In addition the null value `#!null` (in Java written as `null`) is also considered false. Also, if you for some strange reason create a fresh `java.lang.Boolean` object whose `booleanValue()` returns `false`, that is also considered false.

Note: Unlike some other dialects of Lisp, Scheme distinguishes `#f` and the empty list from each other and from the symbol `nil`.

Boolean constants evaluate to themselves, so they do not need to be quoted in programs.

```
#t        ⇒  #t
```

```
#true    ⇒  #t
#f       ⇒  #f
#false   ⇒  #f
'#f      ⇒  #f
```

`boolean` [Type]

The type of boolean values. As a type conversion, a true value is converted to `#t`, while a false value is converted to `#f`. Represented as a primitive Java `boolean` or `kawa.lang.Boolean` when converted to an object.

`boolean?` *obj* [Procedure]

The `boolean?` predicate returns `#t` if *obj* is either `#t` or `#f`, and returns `#f` otherwise.

```
(boolean? #f)   ⇒  #t
(boolean? 0)    ⇒  #f
(boolean? '())  ⇒  #f
```

`boolean=?` *boolean1 boolean2 boolean3 ...* [Procedure]

Returns `#t` if all the arguments are booleans and all are `#t` or all are `#f`.

8.2 Conditionals

Kawa Scheme has the usual conditional expression forms, such as `if`, `case`, `and`, and `or`:

```
(if (> 3 2) 'yes 'no)          ⇒ yes
```

Kawa also allows you bind variables in the condition, using the '`?`' operator.

```
(if (and (? x ::integer (get-value)) (> x 0))
  (* x 10)
  'invalid)
```

In the above, if `(get-value)` evaluates to an integer, that integer is bound to the variable `x`, which is visible in both following sub-expression of `and`, as well case the true-part of the `if`.

Specifically, the first sub-expression of an `if` is a *test-or-match*, which can be a *test-expression*, or a '`?`' match expression, or a combination using `and`:

test-or-match `::=` *test-expression*
  `|` **(?** *pattern expression* **)**
  `|` **(and** *test-or-match* ***)**

A *test-or-match* is true if every nested *test-expression* is true, and every '`?`' operation succeeds. It produces a set of variable bindings which is the union of the bindings produced by all the *pattern*s. In an `and` form, bindings produced by a *pattern* are visible to all subsequent *test-or-match* sub-expressions.

`?` *`pattern expression`* [Syntax]

The form `(? P V)` informally is true if the value of *V* matches the pattern *P*. Any variables bound in *P* are in scope in the "true" path of the containing conditional.

This has the form of an expression, but it can only be used in places where a *test-or-match* is required. For example it can be used as the first clause of an `if` expression, in which case the scope of the variables bound in the `pattern` includes the second (*consequent*) sub-expression. On the other hand, a '`?`' form may not be used as an argument to a procedure application.

`if` ***test-or-match consequent alternate*** [Syntax]
`if` ***test-or-match consequent*** [Syntax]

consequent `::=` *expression*
alternate `::=` *expression*

An `if` expression is evaluated as follows: first, the *test-or-match* is evaluated. If it it true, then *consequent* is evaluated and its values are returned. Otherwise *alternate* is evaluated and its values are returned. If *test* yields `#f` and no *alternate* is specified, then the result of the expression is void.

```
(if (> 2 3) 'yes 'no)          ⇒ no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                   ⇒ 1
(if #f #f)                     ⇒ #!void
(if (? x::integer 3)
  (+ x 1)
  'invalid)                    ⇒ 4
(if (? x::integer 3.4)
  (+ x 1)
  'invalid)                    ⇒ 'invalid
```

The *consequent* and *alternate* expressions are in tail context if the `if` expression itself is.

`cond` ***cond-clause***$^{+}$ [Syntax]
`cond` ***cond-clause**** **(else *expression*. . .)** [Syntax]

cond-clause `::=` (*test-or-match body*)
  `|` (*test* `=>` *expression*)

A `cond` expression is evaluated by evaluating the *test-or-match*s of successive *cond-clause*s in order until one of them evaluates to a true value. When a *test-or-match* is true value, then the remaining *expression*s in its *cond-clause* are evaluated in order, and the results of the last *expression* in the *cond-clause* are returned as the results of the entire `cond` expression. Variables bound by the *test-or-match* are visible in *body*. If the selected *cond-clause* contains only the *test-or-match* and no *expression*s, then the value of the last *test-expression* is returned as the result. If the selected *cond-clause* uses the `=>` alternate form, then the *expression* is evaluated. Its value must be a procedure. This procedure should accept one argument; it is called on the value of the *test-expression* and the values returned by this procedure are returned by the `cond` expression.

If all *test-or-match*s evaluate to `#f`, and there is no `else` clause, then the conditional expression returns unspecified values; if there is an `else` clause, then its *expression*s are evaluated, and the values of the last one are returned.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))         ⇒ greater

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))           ⇒ equal
```

```
(cond ('(1 2 3) => cadr)
      (else #f))              ⇒ 2
```

For a *cond-clause* of one of the following forms:

```
(test  expression *)
(else expression  expression *)
```

the last *expression* is in tail context if the `cond` form itself is. For a *cond clause* of the form:

```
(test => expression)
```

the (implied) call to the procedure that results from the evaluation of *expression* is in tail context if the `cond` form itself is.

`case` ***`case-key case-clause`***$^{+}$ [Syntax]
`case` ***`case-key case-clause`**** ***`case-else-clause`*** [Syntax]

case-key `::=` *expression*
case-clause `::=` ((*datum* *) *expression* +)
`|` ((*datum* *) => *expression*)
case-else-clause `::=` **(else** *expression* +)
`|` **(else** => *expression*)

Each *datum* is an external representation of some object. Each *datum* in the entire `case` expression should be distinct.

A `case` expression is evaluated as follows.

1. The *case-key* is evaluated and its result is compared using `eqv?` against the data represented by the *datum*s of each *case-clause* in turn, proceeding in order from left to right through the set of clauses.
2. If the result of evaluating *case-key* is equivalent to a datum of a *case-clause*, the corresponding *expression*s are evaluated from left to right and the results of the last expression in the *case-clause* are returned as the results of the `case` expression. Otherwise, the comparison process continues.
3. If the result of evaluating *key* is different from every datum in each set, then if there is an *case-else-clause* its expressions are evaluated and the results of the last are the results of the `case` expression; otherwise the result of `case` expression is unspecified.

If the selected *case-clause* or *case-else-clause* uses the `=>` alternate form, then the *expression* is evaluated. It is an error if its value is not a procedure accepting one argument. This procedure is then called on the value of the *key* and the values returned by this procedure are returned by the `case` expression.

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite))    ⇒ composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                    ⇒ unspecified
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else => (lambda (x) x)))     ⇒ c
```

The last *expression* of a *case clause* is in tail context if the `case` expression itself is.

`match` ***match-key match-clause***$^{+}$ [Syntax]

The `match` form is a generalization of `case` using *pattern*s,

match-key `::=` *expression*
match-clause `::=`
  (*pattern* [*guard*] *body* **)**

The *match-key* is evaluated, Then the *match-clause*s are tried in order. The first *match-clause* whose *pattern* matches (and the *guard*, if any, is true), is selected, and the corresponding *body* evaluated. It is an error if no *match-clause* matches.

```
(match value
  (0 (found-zero))
  (x #!if (> x 0) (found-positive x))
  (x #!if (< x 0) (found-negative x))
  (x::symbol (found-symbol x))
  (_ (found-other)))
```

One `case` feature is not (yet) directly supported by `match`: Matching against a list of values. However, this is easy to simulate using a guard using `memq`, `memv`, or `member`:

```
;; compare similar example under case
(match (car '(c d))
  (x #!if (memv x '(a e i o u)) 'vowel)
  (x #!if (memv x '(w y)) 'semivowel)
  (x x))
```

`and` ***test-or-match**** [Syntax]

If there are no *test-or-match* forms, `#t` is returned.

If the `and` is not in *test-or-match* context, then the last sub-expression (if any) must be a *test-expression*, and not a '`?`' form. In this case the *test-or-match* expressions are evaluated from left to right until either one of them is false (a *test-expression* is false or a '`?`' match fails), or the last *test-expression* is reached. In the former case, the `and` expression returns `#f` without evaluating the remaining expressions. In the latter case, the last expression is evaluated and its values are returned.

If the `and` is in *test-or-match* context, then the last sub-form can be '`?`' form. They are evaluated in order: If one of them is false, the entire `and` is false; otherwise the `and` is true.

Regardless, any bindings made by earlier '`?`' forms are visible in later *test-or-match* forms.

```
(and (= 2 2) (> 2 1))         ⇒  #t
(and (= 2 2) (< 2 1))         ⇒  #f
(and 1 2 'c '(f g))           ⇒  (f g)
(and)                         ⇒  #t
(and (? x ::int 23) (> x 0))  ⇒  #t
```

The `and` keyword could be defined in terms of `if` using `syntax-rules` as follows:

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (if test1 (and test2 ...) #t))))
```

The last *test-expression* is in tail context if the `and` expression itself is.

`or` ***`test-expression`*** *...* [Syntax]

If there are no *test-expressions*, `#f` is returned. Otherwise, the *test-expressions* are evaluated from left to right until a *test-expression* returns a true value *val* or the last *test-expression* is reached. In the former case, the `or` expression returns *val* without evaluating the remaining expressions. In the latter case, the last expression is evaluated and its values are returned.

```
(or (= 2 2) (> 2 1))          ⇒ #t
(or (= 2 2) (< 2 1))          ⇒ #t
(or #f #f #f)                 ⇒ #f
(or '(b c) (/ 3 0))           ⇒ (b c)
```

The `or` keyword could be defined in terms of `if` using `syntax-rules` as follows:

```
(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((x test1))
       (if x x (or test2 ...))))))
```

The last *test-expression* is in tail context if the `or` expression itself is.

`not` *test-expression* [Procedure]

The `not` procedure returns `#t` if *test-expression* is false, and returns `#f` otherwise.

```
(not #t)          ⇒  #f
(not 3)           ⇒  #f
(not (list 3))    ⇒  #f
(not #f)          ⇒  #t
(not ’())         ⇒  #f
(not (list))      ⇒  #f
(not ’nil)        ⇒  #f
(not #!null)      ⇒  #t
```

`when` ***`test-expression`*** *form...* [Syntax]

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

`unless` ***`test-expression`*** *form...* [Syntax]

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

8.3 Variables and Patterns

An identifier can name either a type of syntax or a location where a value can be stored. An identifier that names a type of syntax is called a *syntactic keyword* (informally called a *macro*), and is said to be *bound* to a transformer for that syntax. An identifier that names a location is called a *variable* and is said to be *bound* to that location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value stored in the location to which a variable is bound is called the variable's value. By abuse of terminology, the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Certain expression types are used to create new kinds of syntax and to bind syntactic keywords to those new syntaxes, while other expression types create new locations and bind variables to those locations. These expression types are called *binding constructs*. Those that bind syntactic keywords are discussed in Section 7.10 [Macros], page 120. The most fundamental of the variable binding constructs is the [meta-lambda-expression], page 170, because all other variable binding constructs can be explained in terms of `lambda` expressions. Other binding constructs include the Section 8.4 [Definitions], page 141, and the Section 8.5 [Local binding constructs], page 143.

Scheme is a language with block structure. To each place where an identifier is bound in a program there corresponds a *region* of the program text within which the binding is visible. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a `lambda` expression, for example, then its region is the entire `lambda` expression. Every mention of an identifier refers to the binding of the identifier that established the innermost of the regions containing the use.

If there is no binding of the identifier whose region contains the use, then the use refers to the binding for the variable in the global environment, if any; if there is no binding for the identifier, it is said to be *unbound*.

8.3.1 Patterns

The usual way to bind variables is to match an incoming value against a *pattern*. The pattern contains variables that are bound to some value derived from the value.

```
(! [x::double y::double] (some-expression))
```

In the above example, the pattern `[x::double y::double]` is matched against the incoming value that results from evaluating `(some-expression)`. That value is required to be a two-element sequence. Then the sub-pattern `x::double` is matched against element 0 of the sequence, which means it is coerced to a `double` and then the coerced value is matched against the sub-pattern `x` (which trivially succeeds). Similarly, `y::double` is matched against element 1.

The syntax of patterns is a work-in-progress. (The focus until now has been in designing and implementing how patterns work in general, rather than the details of the pattern syntax.)

pattern `::=` *identifier*
  `|` `_`
  `|` *pattern-literal*
  `|` **ﬂ***datum*

| *pattern* **::** *type*
| [*lpattern* *]
lpattern ::= *pattern*
| **@** *pattern*
| *pattern* **...**
| *guard*
pattern-literal ::=
boolean | number | *character* | *string*
guard ::= **#!if** *expression*

This is how the specific patterns work:

identifier This is the simplest and most common form of pattern. The *identifier* is bound to a new variable that is initialized to the incoming value.

_ This pattern just discards the incoming value. It is equivalent to a unique otherwise-unused *identifier*.

pattern-literal
Matches if the value is `equal?` to the *pattern-literal*.

ﬂ*datum* Matches if the value is `equal?` to the quoted *datum*.

pattern **::** *type*
The incoming value is coerced to a value of the specified *type*, and then the coerced value is matched against the sub-*pattern*. Most commonly the sub-*pattern* is a plain *identifier*, so the latter match is trivial.

[*lpattern**]
The incoming value must be a sequence (a list, vector or similar). In the case where each sub-pattern is a plain *pattern*, then the number of sub-patterns must match the size of the sequence, and each sub-pattern is matched against the corresponding element of the sequence. More generally, each sub-pattern may match zero or more consequtive elements of the incoming sequence.

#!if *expression*
No incoming value is used. Instead the *expression* is evaluated. If the result is true, matching succeeds (so far); otherwise the match fails. This form is called a *guard* (`https://en.wikipedia.org/wiki/Guard_(computer_science)`).

@ *pattern* A *splice pattern* may match multiple (zero or more) elements of a sequence. The *pattern* is matched against the resulting sub-sequence.

```
(! [x @r] [2 3 5 7 11])
```

This binds `x` to 2 and `r` to `[3 5 7 11]`.

pattern **...** Similar to `@pattern` in that it matches multiple elements of a sequence. However, each individual element is matched against the *pattern*, rather than the elements as a sequence. This is a Section 8.7 [Repeat forms], page 149.

8.4 Definitions

A variable definition binds one or more identifiers and specifies an initial value for each of them. The simplest kind of variable definition takes one of the following forms:

`!` ***pattern expression*** [Syntax]

Evaluate *expression*, and match the result against *pattern*. Defining variables in *pattern* becomes bound in the current (surrounding) scope.

This is similar to `define-constant` except generalized to a *pattern*.

```
(! [x y] (vector 3 4))
(format "x is ~w and y is ~w" x y) ⇒ "x is 3 and y is 4"
```

`define` *name* [`::` ***type***] ***expression*** [Syntax]

Evaluate the *expression*, optionally converting it to *type*, and bind the *name* to the result.

`define` (*name* ***formal-arguments***) *(**annotation** | **option-pair**)** ***opt-return-type body*** [Syntax]

`define` (*name* `.` ***rest-arg***) *(**annotation** | **option-pair**)** ***opt-return-type body*** [Syntax]

Bind the *name* to a function definition. The form:

```
(define (name formal-arguments)  option-pair * opt-return-type body)
```

is equivalent to:

```
(define name (lambda formal-arguments) name: name  option-pair * opt-
return-type body))
```

while the form:

```
(define (name . rest-arg)  option-pair * opt-return-type body)
```

is equivalent to:

```
(define name (lambda rest-arg) name: name  option-pair * opt-return-
type body))
```

You can associate Section 19.4 [Annotations], page 304, with *name*. A field annotation will be associated with the generated field; a method annotation will be associated with the generated method(s).

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]

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

Defines *name* to have the given *value*. The value is readonly, and you cannot assign to it. (This is not fully enforced.)

If `define-early-constant` is used *or* the *value* is a compile-time constant, then the compiler will create a `final` field with the given name and type, and evaluate *value* in the module's class initializer (if the definition is static) or constructor (if the definition is non-static), before other definitions and expressions. Otherwise, the *value* is evaluated in the module body where it appears.

If the *value* is a compile-time constant, then the definition defaults to being static.

`define-variable` *name* [:: *type*] [*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 per-thread 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.

For `define-namespace` and `define-private-namespace` see Section 10.2 [Namespaces], page 161.

8.5 Local binding constructs

The binding constructs `let`, `let*`, `letrec`, and `letrec*` give Scheme a block structure, like Algol 60. The syntax of these four constructs is identical, but they differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound; in a `let*` expression, the bindings and evaluations are performed sequentially; while in `letrec` and `letrec*` expressions, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

`let` ((*pattern init*) ...) *body* [Syntax]

Declare new local variables as found in the *pattern*s. Each *pattern* is matched against the corresponding *init*. The *init*s are evaluated in the current environment (in left-to-right onder), the *variable*s in the *patterns*s are bound to fresh locations holding the matched results, the *body* is evaluated in the extended environment, and the values of the last expression of body are returned. Each binding of a variable has *body* as its region.

```
(let ((x 2) (y 3))
  (* x y)) ⇒ 6
(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))   ⇒ 35
```

An example with a non-trivial pattern:

```
(let (([a::double b::integer] (vector 4 5)))
  (cons b a))  ⇒ (5 . 4.0)
```

`let*` ((*pattern init*) ...) *body* [Syntax]

The `let*` binding construct is similar to `let`, but the bindings are performed sequentially from left to right, and the region of a *variable*s in a *pattern* is that part of the `let*` expression to the right of the *pattern*. Thus the second pattern is matched in an environment in which the bindings from the first pattern are visible, and so on.

```
(let ((x 2) (y 3))
(let* ((x 7)
       (z (+ x y)))
  (* z x)))  ⇒ 70
```

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

The *variable*s are bound to fresh locations, each *variable* is assigned in left-to-right order to the result of the corresponding *init*, the *body* is evaluated in the resulting environment, and the values of the last expression in body are returned. Despite the left-to-right evaluation and assignment order, each binding of a *variable* has the entire `letrec` or `letrec*` expression as its region, making it possible to define mutually recursive procedures.

In Kawa `letrec` is defined as the same as `letrec*`. In standard Scheme the order of evaluation of the *init*s is undefined, as is the order of assignments. If the order matters, you should use `letrec*`.

If it is not possible to evaluate each *init* without assigning or referring to the value of the corresponding *variable* or the variables that follow it, it is an error.

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
    ⇒ #t
```

8.6 Lazy evaluation

Lazy evaluation (or call-by-need) delays evaluating an expression until it is actually needed; when it is evaluated, the result is saved so repeated evaluation is not needed. Lazy evaluation (`http://en.wikipedia.org/wiki/Lazy_evaluation`) is a technique that can make some algorithms easier to express compactly or much more efficiently, or both. It is the normal evaluation mechanism for strict functional (side-effect-free) languages such as Haskell (`http://www.haskell.org`). However, automatic lazy evaluation is awkward to combine with side-effects such as input-output. It can also be difficult to implement lazy evaluation efficiently, as it requires more book-keeping.

Kawa, like other Schemes, uses "eager evaluation" - an expression is normally evaluated immediately, unless it is wrapped in a special form. Standard Scheme has some basic building blocks for "manual" lazy evaluation, using an explicit `delay` operator to indicate that an expression is to be evaluated lazily, yielding a *promise*, and a `force` function to force evaluation of a promise. This functionality is enhanced in SRFI 45 (`http://srfi.schemers.org/srfi-45/srfi-45.html`), in R7RS-draft (based on SRFI 45), and SRFI 41 (`http://srfi.schemers.org/srfi-41/srfi-41.html`) (lazy lists aka streams).

Kawa makes lazy evaluation easier to use, by *implicit forcing*: The promise is automatically evaluated (forced) when used in a context that requires a normal value, such as arithmetic needing a number. Kawa enhances lazy evaluation in other ways, including support for safe multi-threaded programming.

8.6.1 Delayed evaluation

`delay` *`expression`* [Syntax]

The `delay` construct is used together with the procedure `force` to implement *lazy evaluation* or *call by need*.

The result of `(delay` *`expression`*`)` is a *promise* which at some point in the future may be asked (by the `force` procedure) to evaluate *expression*, and deliver the resulting value. The effect of *expression* returning multiple values is unspecified.

`delay-force` *`expression`* [Syntax]
`lazy` *`expression`* [Syntax]

The `delay-force` construct is similar to `delay`, but it is expected that its argument evaluates to a promise. (Kawa treats a non-promise value as if it were a forced promise.) The returned promise, when forced, will evaluate to whatever the original promise would have evaluated to if it had been forced.

The expression `(delay-force` *`expression`*`)` is conceptually similar to `(delay (force` *`expression`*`))`, with the difference that forcing the result of `delay-force` will in effect result in a tail call to `(force` *`expression`*`)`, while forcing the result of `(delay (force` *`expression`*`))` might not. Thus iterative lazy algorithms that might result in a long series of chains of `delay` and `force` can be rewritten using delay-force to prevent consuming unbounded space during evaluation.

Using `delay-force` or `lazy` is equivalent. The name `delay-force` is from R7RS; the name `lazy` is from the older SRFI-45.

`eager` *obj* [Procedure]

Returns a promise that when forced will return *obj*. It is similar to `delay`, but does not delay its argument; it is a procedure rather than syntax.

The Kawa implementation just returns *obj* as-is. This is because Kawa treats as equivalent a value and forced promise evaluating to the value.

`force` *promise* [Procedure]

The `force` procedure forces the value of *promise*. As a Kawa extension, if the *promise* is not a promise (a value that does not implement `gnu.mapping.Lazy`) then the argument is returned unchanged. If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or "memoized") so that if it is forced a second time, the previously computed value is returned.

```
(force (delay (+ 1 2)))             ⇒  3

(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))       ⇒  (3 3)
```

```
(define integers
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))
(define head
  (lambda (stream) (car (force stream))))
(define tail
  (lambda (stream) (cdr (force stream))))

(head (tail (tail integers)))          ⇒  2
```

The following example is a mechanical transformation of a lazy stream-filtering algorithm into Scheme. Each call to a constructor is wrapped in `delay`, and each argument passed to a deconstructor is wrapped in `force`. The use of `(lazy ...)` instead of `(delay (force ...))` around the body of the procedure ensures that an ever-growing sequence of pending promises does not exhaust the heap.

```
(define (stream-filter p? s)
  (lazy
   (if (null? (force s))
       (delay '())
       (let ((h (car (force s)))
             (t (cdr (force s))))
         (if (p? h)
             (delay (cons h (stream-filter p? t)))
             (stream-filter p? t))))))

(head (tail (tail (stream-filter odd? integers))))
    ⇒ 5
```

`force*` *promise* [Procedure]

Does `force` as many times as necessary to produce a non-promise. (A non-promise is a value that does not implement `gnu.mapping.Lazy`, or if it does implement `gnu.mapping.Lazy` then forcing the value using the `getValue` method yields the receiver.)

The `force*` function is a Kawa extension. Kawa will add implicit calls to `force*` in most contexts that need it, but you can also call it explicitly.

The following examples are not intended to illustrate good programming style, as `delay`, `lazy`, and `force` are mainly intended for programs written in the functional style. However, they do illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
```

```
(define x 5)
p                  ⇒ a promise
(force p)          ⇒ 6
p                  ⇒ a promise, still
(begin (set! x 10)
       (force p))  ⇒ 6
```

8.6.2 Implicit forcing

If you pass a promise as an argument to a function like `sqrt` if must first be forced to a number. In general, Kawa does this automatically (implicitly) as needed, depending on the context. For example:

```
(+ (delay (* 3 7)) 13)   ⇒ 34
```

Other functions, like `cons` have no problems with promises, and automatic forcing would be undesirable.

Generally, implicit forcing happens for arguments that require a specific type, and does not happen for arguments that work on *any* type (or `Object`).

Implicit forcing happens for:

- arguments to arithmetic functions;
- the sequence and the index in indexing operations, like `string-ref`;
- the operands to `eqv?` and `equal?` are forced, though the operands to `eq?` are not;
- port operands to port functions;
- the value to be emitted by a `display` but *not* the value to be emitted by a `write`;
- the function in an application.

Type membership tests, such as the `instance?` operation, generally do not force their values.

The exact behavior for when implicit forcing happens is a work-in-progress: There are certainly places where implicit forcing doesn't happen while it should; there are also likely to be places where implicit forcing happens while it is undesirable.

Most Scheme implementations are such that a forced promise behaves differently from its forced value, but some Scheme implementions are such that there is no means by which a promise can be operationally distinguished from its forced value. Kawa is a hybrid: Kawa tries to minimize the difference between a forced promise and its forced value, and may freely optimize and replace a forced promise with its value.

8.6.3 Blank promises

A *blank promise* is a promise that doesn't (yet) have a value *or* a rule for calculating the value. Forcing a blank promise will wait forever, until some other thread makes the promise non-blank.

Blank promises are useful as a synchronization mechanism - you can use it to safely pass data from one thread (the producer) to another thread (the consumer). Note that you can only pass one value for a given promise: To pass multiple values, you need multiple promises.

```
(define p (promise))
```

```
(future ;; Consumer thread
  (begin
    (do-stuff)
    (define v (force promise)) ; waits until promise-set-value!
    (do-stuff-with v)))
;; Producer thread
... do stuff ...
(promise-set-value! p (calculate-value))
```

`promise` [Constructor]

Calling `promise` as a zero-argument constructor creates a new blank promise.

This calls the constructor for `gnu.mapping.Promise`. You can also create a non-blank promise, by setting one of the `value`, `alias`, `thunk`, or `exception` properties. Doing so is equivalent to calling `promise-set-value!`, `promise-set-alias!`, `promise-set-thunk!`, or `promise-set-exception!` on the resulting promise. For example: `(delay exp)` is equivalent to:

```
(promise thunk: (lambda() exp))
```

The following four procedures require that their first arguments be blank promises. When the procedure returns, the promise is no longer blank, and cannot be changed. This is because a promise is conceptually a placeholder for a single “not-yet-known” value; it is not a location that can be assigned multiple times. The former enables clean and safe (“declarative") use of multiple threads; the latter is much trickier.

`promise-set-value!` *promise value* [Procedure]

Sets the value of the *promise* to *value*, which makes the *promise* forced.

`promise-set-exception!` *promise exception* [Procedure]

Associate *exception* with the *promise*. When the *promise* is forced the *exception* gets thrown.

`promise-set-alias!` *promise other* [Procedure]

Bind the *promise* to be an alias of *other*. Forcing *promise* will cause *other* to be forced.

`promise-set-thunk!` *promise thunk* [Procedure]

Associate *thunk* (a zero-argument procedure) with the *promise*. The first time the *promise* is forced will causes the *thunk* to be called, with the result (a value or an exception) saved for future calls.

`make-promise` *obj* [Procedure]

The `make-promise` procedure returns a promise which, when forced, will return *obj*. It is similar to `delay`, but does not delay its argument: it is a procedure rather than syntax. If *obj* is already a promise, it is returned.

Because of Kawa's implicit forcing, there is seldom a need to use `make-promise`, except for portability.

8.6.4 Lazy and eager types

`promise[T]` [Type]

This parameterized type is the type of promises that evaluate to an value of type `T`. It is equivalent to the Java interface `gnu.mapping.Lazy<T>`. The implementation class for promises is usually `gnu.mapping.Promise`, though there are other classes that implement `Lazy`, most notably `gnu.mapping.Future`, used for futures, which are promises evaluated in a separate thread.

Note the distinction between the types `integer` (the type of actual (eager) integer values), and `promise[integer]` (the type of (lazy) promises that evaluate to integer). The two are compatible: if a `promise[integer]` value is provided in a context requiring an `integer` then it is automatically evaluated (forced). If an `integer` value is provided in context requiring a `promise[integer]`, that conversion is basically a no-op (though the compiler may wrap the `integer` in a pre-forced promise).

In a fully-lazy language there would be no distinction, or at least the promise type would be the default. However, Kawa is a mostly-eager language, so the eager type is the default. This makes efficient code-generation easier: If an expression has an eager type, then the compiler can generate code that works on its values directly, without having to check for laziness.

8.7 Repeat patterns and expressions

Many programming languages have some variant of list comprehension syntax (`https://en.wikipedia.org/wiki/List_comprehension`). Kawa splits this into two separate forms, that can be in separate parts of the program:

- A *repeat pattern* as you might guess repeats a pattern by matching the pattern once for each element of a sequence. For example, assume `A` is a some sequence-valued expression. Then:

  ```
  #|kawa:3|# (! [a::integer ...] A)
  ```

  Here '`a::integer ...`' is a *repeat pattern* that matches all the elements pf `A`. We call '`a::integer`' the *repeated pattern* - it matches an individual element of `A`. Any variable defined in a repeated pattern is a *repeat variable*. In the example, that would be `a`.

- A *repeat expression* creates a sequence by repeating an expression for each element of the result.

  ```
  #|kawa:4|# [(* 2 a) ...]
  [4 6 10 14 22]
  ```

  In this case '`(* 2 a) ...`' is the repeat expression. The *repeated expression* is '`(* 2 a)`'. The repeated expression is evaluated once for each element of any contained repeat variable. If there is more than one repeat variable, they are repeated in parallel, as many times as the "shortest" repeat variable, similar to the `map` procedure. (If there is no repeat variable, the repeated expression is potentially evaluated infinitely many times, which is not allowed. A planned extension will allow it for lazy repeated expression.)

The use of '`...`' for repeat patterns and expressions mirrors exactly their use in `syntax-rules` patterns and templates.

It is an error to use a repeat variable outside of repeat context:

```
#|kawa:5|# a
/dev/stdin:2:1: using repeat variable 'a' while not in repeat context
```

The repeat form feature is not yet complete. It is missing functionality such as selecting only some elements from a repeat sequence, lazy sequences, and it could be optimized more.

A repeat variable can be used multiple times in the same repeat expressions, or different repeat expressions:

```
#|kawa:7|# [a ... a ...]
[2 3 5 7 11 2 3 5 7 11]
#|kawa:8|# [(* a a) ...]
[4 9 25 49 121]
```

Repeat expressions are useful not just in sequence literals, but in the argument list of a procedure call, where the resulting sequence is spliced into the argument list. This is especially useful for functions that take a variable number of arguments, because that enables a convenient way to do fold/accumulate/reduce (`https://en.wikipedia.org/wiki/Fold_(higher-order_function)`) operations. For example:

```
#|kawa:9|# (+ a ...)
28
```

because 28 is the result of `(+ 2 3 5 7 11)`.

An elegant way to implement dot product (`https://en.wikipedia.org/wiki/Dot_product`):

```
(define (dot-product [x ...] [y ...])
  (+ (* x y) ...))
```

When an ellipse expression references two or more distinct repeat variables then they are processed "in parallel". That does not (necessarily) imply muliple threads, but that the first element of the repeat result is evaluated using the first element of all the repeat sequences, the second element of the result uses the second element of all the repeat sequences, and so on.

Sub-patterns in repeat patterns

While the repeated pattern before the '`...`' is commonly in identifier, it may be a more complex pattern. We showed earlier the repeated pattern with a type specifier, which applies to each element:

```
#|kawa:11|# (define (isum [x::integer ...]) (+ x ...))
#|kawa:12|# (isum [4 5 6])
15
#|kawa:12|# (isum [4 5.1 6])
Argument #1 (null) to 'isum' has wrong type
at gnu.mapping.CallContext.matchError(CallContext.java:189)
at atInteractiveLevel-6.isum$check(stdin:11)
...
```

(The stack trace line number `stdin:11` is that of the `isum` definition.)

You can nest repeat patterns, allowing matching against sequences whose elements are sequences.

```
#|kawa:31|# (define (fun2 [[x ...] ...] [y ...])
#|.....32|#   [[(+ x y) ...] ...])
#|kawa:33|# (fun2 [[1 2 3] [10 11 12]] [100 200])
[[101 102 103] [210 211 212]]
```

Note that `x` is double-nested, while `y` is singly-nested.

Here each element is constrained to be a pair (a -element sequence):

```
#|kawa:1|# (! [[x y] ...] [[11 12] [21 22] [31 32]])
#|kawa:2|# [(+ x y) ...]
#(23 43 63)
#|kawa:3|# [[x ...] [y ...]]
#(#(11 21 31) #(12 22 32))
```

8.8 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.

`future` *expression* [Syntax]

Creates a new thread that evaluates *expression*.

(The result extends `java.lang.Thread` and implements `gnu.mapping.Lazy`.)

`force` *thread* [Procedure]

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

`runnable` *function* [Procedure]

Creates a new `Runnable` instance from a function. Useful for passing to Java code that expects a `Runnable`. You can get the result (a value or a thrown exception) using the `getResult` method.

`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.

8.9 Exception handling

An *exception* is an object used to signal an error or other exceptional situation. The program or run-time system can *throw* the exception when an error is discovered. An exception handler is a program construct that registers an action to handle exceptions when the handler is active.

If an exception is thrown and not handled then the read-eval-print-loop will print a stack trace, and bring you back to the top level prompt. When not running interactively, an unhandled exception will normally cause Kawa to be exited.

In the Scheme exception model (as of R6RS and R7RS), exception handlers are one-argument procedures that determine the action the program takes when an exceptional situation is signaled. The system implicitly maintains a current exception handler in the dynamic environment. The program raises an exception by invoking the current exception handler, passing it an object encapsulating information about the exception. Any procedure accepting one argument can serve as an exception handler and any object can be used to represent an exception.

The Scheme exception model is implemented on top of the Java VM's native exception model where the only objects that can be thrown are instances of `java.lang.Throwable`. Kawa also provides direct access to this native model, as well as older Scheme exception models.

`with-exception-handler` *handler thunk* [Procedure]

It is an error if *handler* does not accept one argument. It is also an error if *thunk* does not accept zero arguments. The `with-exception-handler` procedure returns the results of invoking *thunk*. The *handler* is installed as the current exception handler in the dynamic environment used for the invocation of *thunk*.

```
(call-with-current-continuation
  (lambda (k)
   (with-exception-handler
    (lambda (x)
     (display "condition: ")
     (write x)
     (newline)
     (k 'exception))
    (lambda ()
     (+ 1 (raise ’an-error))))))
       ⇒ exception
       and prints condition: an-error
```

```
(with-exception-handler
 (lambda (x)
  (display "something went wrong\n"))
 (lambda ()
  (+ 1 (raise ’an-error))))
    prints something went wrong
```

After printing, the second example then raises another exception.

Performance note: The *thunk* is inlined if it is a lambda expression. However, the *handler* cannot be inlined even if it is a lambda expression, because it could be called by `raise-continuable`. Using the `guard` form is usually more efficient.

`raise` *obj* [Procedure]

Raises an exception by invoking the current exception handler on *obj*. The handler is called with the same dynamic environment as that of the call to raise, except that the current exception handler is the one that was in place when the handler being called was installed. If the handler returns, then *obj* is re-raised in the same dynamic environment as the handler.

If *obj* is an instance of `java.lang.Throwable`, then `raise` has the same effect as `primitive-throw`.

`raise-continuable` *obj* [Procedure]

Raises an exception by invoking the current exception handler on *obj*. The handler is called with the same dynamic environment as the call to `raise-continuable`, except that: (1) the current exception handler is the one that was in place when the handler being called was installed, and (2) if the handler being called returns, then it will again become the current exception handler. If the handler returns, the values it returns become the values returned by the call to `raise-continuable`.

```
(with-exception-handler
  (lambda (con)
    (cond
      ((string? con)
       (display con))
      (else
       (display "a warning has been issued")))
    42)
  (lambda ()
    (+ (raise-continuable "should be a number")
       23)))
      prints: should be a number
      ⇒ 65
```

`guard` *`variable cond-clause`*$^+$ *`body`* [Syntax]

The *body* is evaluated with an exception handler that binds the raised object to *variable* and, within the scope of that binding, evaluates the clauses as if they were the clauses of a `cond` expression. That implicit `cond` expression is evaluated with the continuation and dynamic environment of the `guard` expression. If every cond-clause's test evaluates to `#f` and there is no `else` clause, then `raise-continuable` is invoked on the raised object within the dynamic environment of the original call to `raise` or `raise-continuable`, except that the current exception handler is that of the `guard` expression.

```
(guard (condition
         ((assq 'a condition) => cdr)
         ((assq 'b condition)))
  (raise (list (cons 'a 42))))
      ⇒ 42

(guard (condition
         ((assq 'a condition) => cdr)
         ((assq 'b condition)))
  (raise (list (cons 'b 23))))
      ⇒ (b . 23)
```

Performance note: Using `guard` is moderately efficient: there is some overhead compared to using native exception handling, but both the *body* and the handlers in the *cond-clause* are inlined.

`dynamic-wind` *in-guard thunk out-guard* [Procedure]

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.

`read-error?` *obj* [Procedure]

Returns #t if *obj* is an object raised by the `read` procedure. (That is if *obj* is a `gnu.text.SyntaxException`.)

`file-error?` *obj* [Procedure]

Returns #t if *obj* is an object raised by inability to open an input or output port on a file. (This includes `java.io.FileNotFoundException` as well as certain other exceptions.)

8.9.1 Simple error objects

`error` *message obj ...* [Procedure]

Raises an exception as if by calling `raise` on a newly allocated *simple error object*, which encapsulates the information provided by *message* (which should a string), as well as any *obj* arguments, known as the irritants.

The string representation of a simple error object is as if calling `(format "#<ERROR ~a~{ ~w~}>"` *`message irritants`*`)`. (That is the *message* is formatted as if with `display` while each irritant *obj* is formatted as if with `write`.)

This procedure is part of SRFI-23, and R7RS. It differs from (and is incompatible with) R6RS's `error` procedure.

`error-object?` *obj* [Procedure]

Returns `#t` if *obj* is a simple error object. Specifically, that *obj* is an instance of `kawa.lang.NamedException`. Otherwise, it returns `#f`.

`error-object-message` *error-object* [Procedure]

Returns the message encapsulated by error-object, which must be a simple error object.

`error-object-irritants` *error-object* [Procedure]

Returns a list of the irritants (other arguments) encapsulated by error-object, which must be a simple error object.

8.9.2 Named exceptions

These functions associate a symbol with exceptions and handlers: A handler catches an exception if the symbol matches.

`catch` *key thunk handler* [Procedure]

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 arg ...* [Procedure]

Invoke the catch form matching *key*, passing the *arg*s to the current *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.

8.9.3 Native exception handling

`primitive-throw` *exception* [Procedure]

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`. However, the current implementation does not duplicate the *handler*.)

`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`.)

9 Control features

9.1 Mapping functions

The procedures `string-for-each` and `string-map` are documented under Section 13.3 [Strings], page 205.

The procedure `string-cursor-for-each` is documented under [String Cursor API], page 221.

`map` *proc sequence₁ sequence₂ ...* [Procedure]
`for-each` *proc sequence₁ sequence₂ ...* [Procedure]

The `map` procedure applies *proc* element-wise to the elements of the *sequence*s and returns a list of the results, in order. The dynamic order in which *proc* is applied to the elements of the *sequence*s is unspecified.

The `for-each` procedure does the same, but is executed for the side-effects of *proc*, whose result (if any) is discarded. Unlike `map`, `for-each` is guaranteed to call *proc* on the elements of the *sequences*s in order from the first element(s) to the last. The value returned by `for-each` is the void value.

Each *sequence* must be a generalized sequence. (Traditionally, these arguments were restricted to lists, but Kawa allows sequences, including vectors, Java arrays, and strings.) If more than one *sequence* is given and not all *sequence*s have the same length, the procedure terminates when the shortest *sequence* runs out. The *sequence*s can be infinite (for example circular lists), but it is an error if all of them are infinite.

The *proc* must be a procedure that accepts as many arguments as there are *sequence* arguments. It is an error for *proc* to mutate any of the *sequence*s. In the case of `map`, *proc* must return a single value.

```
(map cadr '((a b) (d e) (g h)))
    ⇒ (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
    ⇒ (1 4 27 256 3125)

(map + ’(1 2 3) ’(4 5 6 7))  ⇒ (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b)))
    ⇒ (1 2) or (2 1)
```

The result of `map` is a list, even if the arguments are non-lists:

```
(map +
     #(3 4 5)
     (float[] 0.5 1.5))
```

```
          ⇒ (3.5 5.5)
```

To get a vector result, use `vector-map`.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)
    ⇒  #(0 1 4 9 16)
```

A string is considered a sequence of `character` values (not 16-bit `char` values):

```
(let ((v (make-vector 10 #\-)))
  (for-each (lambda (i ch)
              (vector-set! v i ch))
            [0 <: ]
            "Smile !")
   v)
    ⇒ #(#\S #\m #\i #\l #\e #\space #\x1f603 #\! #\- #\-)
```

Performance note: These procedures are pretty well optimized. For each *sequence* the compiler will by default create an iterator. However, if the type of the *sequence* is known, the compiler will inline the iteration code.

`vector-map` ***proc sequence*** $_1$ ***sequence*** $_2$ ... [Procedure]

Same as the `map` procedure, except the result is a vector. (Traditionally, these arguments were restricted to vectors, but Kawa allows sequences, including lists, Java arrays, and strings.)

```
(vector-map cadr '#((a b) (d e) (g h)))
    ⇒ #(b e h)

(vector-map (lambda (n) (expt n n))
            '#(1 2 3 4 5))
    ⇒ #(1 4 27 256 3125)

(vector-map + '#(1 2 3) ’#(4 5 6 7))
    ⇒ #(5 7 9)

(let ((count 0))
  (vector-map
    (lambda (ignored)
      (set! count (+ count 1))
      count)
    '#(a b)))
    ⇒ #(1 2) or #(2 1)
```

`vector-for-each` ***proc vector*** $_1$ ***vector*** $_2$ ... [Procedure]

Mostly the same as `for-each`, however the arguments should be generalized vectors. Specifically, they should implement `java.util.List` (which both regular vectors and uniform vectors do). The *vectors* should also be efficiently indexable.

(Traditionally, these arguments were restricted to vectors, but Kawa allows sequences, including lists, Java arrays, and strings.)

```
(let ((v (make-list 5)))
  (vector-for-each
    (lambda (i) (list-set! v i (* i i)))
    '#(0 1 2 3 4))
  v)
    ⇒ (0 1 4 9 16)
```

9.2 Multiple values

The multiple-value feature was added in R5RS.

`values` *object* ... [Procedure]

Delivers all of its arguments to its continuation.

`call-with-values` *producer consumer* [Procedure]

Calls its *producer* argument with no arguments and a continuation that, when passed some values, calls the *consumer* procedure with those values as arguments.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
                        ⇒ 5

(call-with-values * -)   ⇒ -1
```

Performance note: If either the *producer* or *consumer* is a fixed-arity lambda expression, it is inlined.

`define-values` *`formals`* *`expression`* [Syntax]

It is an error if a variable appears more than once in the set of *formals*.

The *expression* is evaluated, and the *formals* are bound to the return values in the same way that the *formals* in a `lambda` expression are matched to the arguments in a procedure call.

```
(define-values (x y) (integer-sqrt 17))
(list x y)    ⇒ (4 1)
(let ()
  (define-values (x y) (values 1 2))
  (+ x y))
              ⇒  3
```

`let-values` ((*`formals`* *`expression`*) ...) *`body`* [Syntax]

Each *formals* should be a formal arguments list, as for a `lambda`.

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.

`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]

This convenience form (from SRFI-8 (`http://srfi.schemers.org/srfi-8/srfi-8.html`)) is equivalent to:

```
(let-values ((formals expression)) body)
```

For example:

```
(receive a (values 1 2 3 4)
  (reverse a)) ⇒ (4 3 2 1)

(receive (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))
  (receive (a b) (values x y)
    (receive (x y) (values a b)
      (list a b x y))))    ⇒ (x y x y)
```

`values-append` *arg1 ...* [Procedure]

The values resulting from evaluating each argument are appended together.

10 Symbols and namespaces

An identifier is a name that appears in a program.

A symbol is an object representing a string that cannot be modified. This string is called the symbol's name. Unlike strings, two symbols whose names are spelled the same way are indistinguishable. A symbol is immutable (unmodifiable) and normally viewed as atomic. Symbols are useful for many applications; for instance, they may be used the way enumerated values are used in other languages.

In addition to the simple symbols of standard Scheme, Kawa also has compound (two-part) symbols.

10.1 Simple symbols

Simple symbols have no properties other than their name, an immutable string. They have the useful property that two simple symbols are identical (in the sense of `eq?`, `eqv?` and `equal?`) if and only if their names are spelled the same way. A symbol literal is formed using `quote`.

`symbol?` *obj* [Procedure]

Return `#t` if *obj* is a symbol, `#f` otherwise.

```
(symbol? 'foo)          ⇒ #t
(symbol? (car '(a b)))  ⇒ #t
(symbol? "bar")         ⇒ #f
(symbol? 'nil)          ⇒ #t
(symbol? '())           ⇒ #f
(symbol? #f)            ⇒ #f
```

`symbol->string` *symbol* [Procedure]

Return the name of *symbol* as an immutable string.

```
(symbol->string 'flying-fish)                     ⇒  "flying-fish"
(symbol->string 'Martin)                          ⇒  "Martin"
(symbol->string (string->symbol "Malvina"))       ⇒  "Malvina"
```

`string->symbol` *string* [Procedure]

Return the symbol whose name is *string*.

```
(eq? 'mISSISSIppi 'mississippi)
⇒ #f

(string->symbol "mISSISSIppi")
⇒ the symbol with name "mISSISSIppi"

(eq? 'bitBlt (string->symbol "bitBlt"))
⇒ #t

(eq? 'JollyWog (string->symbol (symbol->string 'JollyWog)))
⇒ #t

(string=? "K. Harper, M.D."
          (symbol->string (string->symbol "K. Harper, M.D.")))
⇒ #t
```

10.2 Namespaces and compound symbols

Different applications may want to use the same symbol to mean different things. To avoid such *name clashes* we can use *compound symbols*, which have two string parts: a *local name* and a *namespace URI*. The namespace-uri can be any string, but it is recommended that it have the form of an absolute URI (`http://en.wikipedia.org/wiki/Uniform_Resource_Identifier`). It would be too verbose to write the full URI all the time, so one usually uses a *namespace prefix* (namespace alias) as a short local alias to refer to a namespace URI.

Compound symbols are usually written using the infix colon operator:

`prefix``:`*`local-name`*

where *prefix* is a namespace alias bound to some (lexically-known) namespace URI.

Compound symbols are used for namespace-aware XML processing.

10.2.1 Namespace objects

A *namespace* is a mapping from strings to symbols. The string is the local-name of the resulting symbol. A namespace is similar to a Common Lisp *package*.

A namespace has a namespace-uri, which a string; it is recommended that it have the form of an absolute URI. A namespace may optionally have a prefix, which is a string used when printing out symbols belonging to the namespace. (If you want "equivalent symbols" (i.e. those that have the same local-name and same uri) to be the identical symbol object, then you should use namespaces whose prefix is the empty string.)

`namespace` *name* [*prefix*] [Constructor]

Return a namespace with the given *name* and *prefix*. If no such namespace exists, create it. The *namespace-name* is commonly a URI, especially when working with XML, in which case it is called a *namespace-uri*. However, any non-empty string is allowed. The prefix can be a string or a simple symbol. (If a symbol is used, then the symbol's local-name is used.) The default for *prefix* is the empty string. Multiple calls with the same arguments will yield the same namespace object.

The reader macro `#,namespace` is equivalent to the `namespace` function, but it is invoked at read-time:

```
#,(namespace "http://www.w3.org/1999/XSL/Transform" xsl)
(eq? #,(namespace "foo") (namespace "foo")) ⇒ #t
```

The form `(,#namespace "" "")` returns the default *empty namespace*, which is used for simple symbols.

`namespace-uri` *namespace* [Procedure]

Return the namespace-uri of the argument *namespace*, as a string.

`namespace-prefix` *namespace* [Procedure]

Return the namespace prefix of the argument *namespace*, as a string.

10.2.2 Compound symbols

A compound symbol is one that belongs to a namespace other than the default empty namespace, and (normally) has a non-empty namespace uri. (It is possible for a symbol to belong to a non-default namespace and have an empty namespace uri, but that is not recommended.)

`symbol` *local-name namespace-spec* [Constructor]
`symbol` *local-name* [*uri* [*prefix*]] [Constructor]

Construct a symbol with the given *local-name* and namespace. If *namespace-spec* is a namespace object, then find (or, if needed, construct) a symbol with the given *local-name* belonging to the namespace. Multiple calls to `symbol` with the same namespace and *local-name* will yield the same symbol object.

If uri is a string (optionally followed by a prefix), then:

```
(symbol lname uri [prefix])
```

is equivalent to:

```
(symbol lname (namespace uri [prefix]))
```

Using `#t` for the *namespace-spec* is equivalent to using the empty namespace `#,(namespace "")`.

Using `#!null` or `#f` for the *namespace-spec* creates an *uninterned* symbol, which does not belong to any namespace.

`symbol-local-name` *symbol* [Procedure]

Return the local name of the argument symbol, as an immutable string. (The string is interned, except in the case of an uninterned symbol.)

`symbol-prefix` *symbol* [Procedure]

Return the prefix of the argument symbol, as an immutable (and interned) string.

`symbol-namespace-uri` *symbol* [Procedure]

Return the namespace uri of the argument symbol, as an immutable (and interned) string.

`symbol-namespace` *symbol* [Procedure]

Return the namespace object (if any) of the argument symbol. Returns `#!null` if the symbol is uninterned.

`symbol=?` *`symbol`*$_1$ *`symbol`*$_2$ *`symbol`*$_3$ *`...`* [Procedure]

Return `#t` if the symbols are equivalent as symbols, i.e., if their local-names and namespace-uris are the same. They may have different values of `symbol-prefix` and `symbol-namespace`. If a symbol is uninterned (or is `#!null`) then `symbol=?` returns the same result as `eq?`.

Two symbols are `equal?` or `eqv?` if they're `symbol=?`.

10.2.3 Namespace aliases

A namespace is usually referenced using a shorter *namespace alias*, which is is a lexical definition that binds a namespace prefix to a namespace object (and thus a namespace uri). This allows using compound symbols as identifiers in Scheme programs.

`define-namespace` *name namespace-name* [Syntax]

Defines *name* as a *namespace prefix* - a lexically scoped "nickname" for the namespace whose full name is *namespace-name*, which should be a non-empty string literal. It is customary for the string have syntactic form of an absolute URI (`http://en.wikipedia.org/wiki/Uniform_Resource_Identifier`), but any non-empty string is acceptable and is used without further interpretation.

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-name*.

Has mostly the same effect as:

```
(define-constant name #,(namespace namespace-name)
```

However, using `define-namespace` (rather than `define-constant`) is recommended if you want to use compound symbols as names of variables, especially local variables, or if you want to quote compound symbols.

Note that the prefix is only visible lexically: it is not part of the namespace, or thus indirectly the symbols, and so is not available when printing the symbol. You might consider using `define-xml-namespace` as an alternative.

A namespace is similar to a Common Lisp package, and the *namespace-name* is like the name of the package. However, a namespace alias belongs to the lexical scope, while a Common Lisp package nickname is global and belongs to the package itself.

If the namespace-name starts with the string `"class:"`, then the *name* can be used for invoking Java methods (see Section 19.9 [Method operations], page 320) and accessing fields (see Section 19.11 [Field operations], page 327).

You can use a namespace as an abbreviation or renaming of a class name, but as a matter of style `define-alias` is preferred.

`define-private-namespace` *name namespace-name* [Syntax]

Same as `define-namespace`, but the prefix *name* is local to the current module.

For example, you might have a set of a geometry definitions defined under the namespace-uri `"http://foo.org/lib/geometry"`:

```
(define-namespace geom "http://foo.org/lib/geometry")
(define (geom:translate x y)
  (java.awt.geom.AffineTransform:getTranslateInstance x y))
(define geom:zero (geom:translate 0 0))
geom:zero
  ⇒ AffineTransform[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]]
```

You could have some other definitions for complex math:

```
(define-namespace complex "http://foo.org/lib/math/complex")
(define complex:zero +0+0i)
```

You can use a namespace-value directly in a compound name:

```
(namespace "http://foo.org/lib/geometry"):zero
  ⇒ AffineTransform[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]]
```

The variation `define-xml-namespace` is used for Section 20.3 [Creating XML nodes], page 338.

`define-xml-namespace` *prefix* "*namespace-uri*" [Syntax]

Defines a namespace with prefix *prefix* and URI *namespace-uri*. This is similar to `define-namespace` but with two important differences:

- Every symbol in the namespace automatically maps to an element-constructor-type, as with the `html` namespace.
- The *prefix* is a component of the namespace object, and hence indirectly of any symbols belongining to the namespace.

Thus the definition is roughly equivalent to:

```
(define-constant name #,(namespace namespace-name name)
```

along with an infinite set of definitions, for every possible *tag*:

```
(define (name:tag . rest) (apply make-element 'name:tag rest))
```

```
$ kawa --output-format xml
#|kawa:1|# (define-xml-namespace na "Namespace1")
#|kawa:2|# (define-xml-namespace nb "Namespace1")
#|kawa:3|# (define xa (na:em "Info"))
#|kawa:4|# xa
<na:em xmlns:na="Namespace1">Info</na:em>
#|kawa:5|# (define xb (nb:em "Info"))
#|kawa:6|# xa
<nb:em xmlns:nb="Namespace1">Info</nb:em>
```

Note that the prefix is part of the qualified name (it is actually part of the namespace object), and it is used when printing the tag. Two qualified names (symbols) that have the same local-name and the same namespace-name are considered equal, even if they have different prefix. You can think of the prefix as annotation used when printing, but not otherwise part of the "meaning" of a compound symbol. They are the same object if they also have the same prefix. This is an important difference from traditional Lisp/Scheme symbols, but it is how XML QNames work.

```
#|kawa:7|# (instance? xb na:em)
true
#|kawa:8|# (eq? 'na:em 'nb:em)
false
#|kawa:9|# (equal? 'na:em 'nb:em)
true
#|kawa:10|# (eqv? 'na:em 'nb:em)
true
```

(Note that `#t` is printed as `true` when using XML formatting.)

The predefined `html` prefix could be defined thus:

```
(define-xml-namespace html "http://www.w3.org/1999/xhtml")
```

10.3 Keywords

Keywords are similar to symbols. They are used mainly for specifying keyword arguments.

Historically keywords have been self-evaluating (you did not need to quote them). This has changed: you must quote a keyword if you want a literal keyword value, and not quote it if it is used as a keyword argument.

> *keyword* `::=` *identifier*`:`
> `|` `#:`*identifier*

The two syntaxes have the same meaning: The former is nicer-looking; the latter is more portable (and required if you use the `--r7rs` command-line flag).

> *Details:* In r7rs and other Scheme standards the colon character does not have any special meaning, so `foo:` or `foo:bar` are just regular identifiers. Therefore some other Scheme variants that have keywords (including Guile and Racket) use the `#:` syntax. Kawa has some hacks so that *most* standard Scheme programs that have colons in identifiers will work. However, for best compatibility, use the `--r7rs` command-line flag (which turns colon into a regular character in a symbol), and the `#:` syntax.

A keyword is a single token; therefore no whitespace is allowed between the *identifier* and the colon or after the `#:`; these characters are not considered part of the name of the keyword.

`keyword?` *obj* [Procedure]

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

`keyword->string` *keyword* [Procedure]

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

`string->keyword` *string* [Procedure]

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

10.4 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.

11 Procedures

11.1 Application and Arguments Lists

When a procedure is called, the actual argument expressions are evaluated, and the resulting values becomes the actual argument list. This is then matched against the formal parameter list in the procedure definition, and (assuming they match) the procedure body is called.

11.1.1 Arguments lists

An argument list has three parts:

- Zero or more *prefix arguments*, each of which is a value. These typically get bound to named required or optional formal parameters, but can also get bound to patterns.
- Zero or more *keyword arguments*, each of which is a keyword (an identifier specified with keyword syntax) combined with a value. These are bound to either named keyword formal parameters, or bundled in with a rest parameter.
- Zero or more *postfix arguments*, each of which is a value. These are usually bound to a “rest” formal parameter, which receives any remaining arguments.

  If there are no keyword arguments, then it ambiguous where prefix arguments end and where postfix arguments start. This is normally not a problem: the called procedure can split them up however it wishes.

Note that all keyword arguments have to be grouped together: It is not allowed to have a keyword argument followed by a plain argument followed by a keyword argument.

The argument list is constructed by evaluating each *operand* of the *procedure-call* in order:

expression The *expression* is evaluated, yielding a single value that becomes a prefix or postfix argument.

keyword expression
The *expression* is evaluated. The resulting value combined with the *keyword* becomes a keyword argument.

@*expression*
The *expression* is evaluated. The result must be a sequence - a list, vector, or primitive array. The values of the sequence are appended to the resulting argument list. Keyword arguments are not allowed.

@:*expression*

The *expression* is evaluted. The result can be a sequence; a hash table (viewed as a collection of (keyword,value) pairs); or an *explicit argument list* object, which is a sequence of values *or* keyword arguments. The values and keyword arguments are appended to the resulting argument list, though subject to the restriction that keyword arguments must be adjacent in the resulting argument list.

11.1.2 Explicit argument list objects

Sometimes it is useful to create an argument list out of pieces, take argument lists apart, iterate over them, and generally treat an argument list as an actual first-class value.

Explicit argument list objects can take multiple forms. The simplest is a sequence: a list, vector, or primitive array. Each element of the list becomes a value in the resulting argument list.

```
(define v1 '(a b c))
(define v2 (int[] 10 11 12 13))
(list "X" @v1 "Y" @v2 "Z")
  ⇒ ("X" a b c "Y" 10 11 12 13 "Z")
```

Things get more complicated once keywords are involved. An explicit argument list with keywords is only allowed when using the `@:` splicing form, not the `@` form. It can be either a hash table, or the types `arglist` or `argvector`.

> *Design note:* An argument list with keywords is straightforward in Common Lisp and some Scheme implementations (including order versions of Kawa): It's just a list some of whose `car` cells are keyword objects. The problem with this model is neither a human or the compiler can reliably tell when an argument is a keyword, since any variable might have been assigned a keyword. This limits performance and error checking.

A hash table (anything the implements `java.util.Map`) whose keys are strings or keyword objects is interpreted as a sequence of keyword arguments, using the hash-table keys and values.

`argvector` [Type]
`argvector` *operand** [Constructor]

List of arguments represented as an immutable vector. A keyword argument takes two elements in this vector: A keyword object, followed by the value.

```
(define v1 (argvector 1 2 k1: 10 k2: 11 98 99))
(v1 4) ⇒ 'k2
(v1 5) ⇒ 11
```

When `v1` is viewed as a vector it is equivalent to `(vector 1 2 'k1: 10 'k2: 11 98 99)`. (Note in this case the keywords need to be quoted, since the `vector` constructor does not take keyword arguments.) However, the `argvector` "knows" which arguments are actually keyword arguments, and can be examined using the `(kawa arglist)` library discussed below:

```
(arglist-key-count (argvector 1 x: 2 3)) ⇒ 1
(arglist-key-count (argvector 1 'x: 2 3)) ⇒ 0
```

```
(arglist-key-count (vector 1 'x: 2 3)) ⇒ 0
```

In this case:

```
(fun 'a @:v1)
```

is equivalent to:

```
(fun 'a 1 2 k1: 10 k2: 11 98 99)
```

`arglist` [Type]
`arglist` *operand** [Constructor]

Similar to `argvector`, but compatible with `list`. If there are no keyword arguments, returns a plain list. If there is at least one keyword argument creates a special `gnu.mapping.ArgListPair` object that implements the usual `list` properties but internally wraps a `argvector`.

11.1.3 Argument list library

```
(import (kawa arglist))
```

In the following, *args* is an `arglist` or `argvector` (or in general any object that implement `gnu.mapping.ArgList`). Also, *args* can be any sequence, in which case it behaves like an `argvector` that has no keyword arguments.

`arglist-walk` *args proc* [Procedure]

Call *proc* once, in order, for each argument in *args*. The *proc* is called with two arguments, corresponding to `(arglist-key-ref args i)` and `(arglist-arg-ref args i)` for each *i* from 0 up to `(arglist-arg-count args)` (exclusive). I.e. the first argument is either `#!null` or the keyword (as a string); the second argument is the corresponding argument value.

```
(define (print-arguments args #!optional (out (current-output-port)))
  (arglist-walk args
                (lambda (key value)
                  (if key (format out "key: ~a value: ~w~%" key value)
                      (format out "value: ~w~%" value)))))
```

`arglist-key-count` *args* [Procedure]

Return the number of keyword arguments.

`arglist-key-start` *args* [Procedure]

Number of prefix arguments, which is the number of arguments before the first keyword argument.

`arglist-arg-count` *args* [Procedure]

Return the number of arguments. The count includes the number of keyword arguments, but not the actual keywords.

```
(arglist-arg-count (arglist 10 11 k1: -1 19)) ⇒ 4
```

`arglist-arg-ref` *args index* [Procedure]

Get the *index*'th argument value. The *index* counts keyword argument values, but not the keywords themselves.

```
(arglist-arg-ref (arglist 10 11 k1: -1 19) 2) ⇒ -1
(arglist-arg-ref (arglist 10 11 k1: -1 19) 3) ⇒ 19
```

`arglist-key-ref` *args index* [Procedure]

The *index* counts arguments like `arglist-arg-ref` does. If this is a keyword argument, return the corresponding keyword (as a string); otherwise, return `#!null` (which counts as false).

```
(arglist-key-ref (argvector 10 11 k1: -1 k2: -2 19) 3) ⇒ "k2"
(arglist-key-ref (argvector 10 11 k1: -1 k2: -2 19) 4) ⇒ #!null
```

`arglist-key-index` *args key* [Procedure]

Search for a keyword matching *key* (which must be an interned string). If there is no such keyword, return -1. Otherwise return the keyword's index as an argument to `arglist-key-ref`.

`arglist-key-value` *args key default* [Procedure]

Search for a keyword matching *key* (which must be an interned string). If there is no such keyword, return the *default*. Otherwise return the corresponding keyword argument's value.

11.1.4 Apply procedures

`apply` *proc argi* argrest* [Procedure]

Argrest 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 *argi*... values plus all the elements of *argrest*.

Equivalent to: `(`*proc argi** `@`*argrest*`)`.

`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.

11.2 Lambda Expressions and Formal Parameters

A `lambda` expression evaluates to a procedure. The environment in effect when the `lambda` expression was evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the `lambda` expression was evaluated will be extended by binding the variables in the formal argument list to fresh locations, and the corresponding actual argument values will be stored in those locations. (A *fresh location* is one that is distinct from every previously existing location.) Next, the expressions in the body of the lambda expression will be evaluated sequentially in the extended environment. The results of the last expression in the body will be returned as the results of the procedure call.

```
(lambda (x) (+ x x))   ⇒ a procedure
```

```
((lambda (x) (+ x x)) 4)  ⇒ 8

(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10) ⇒ 3

(define add4
  (let ((x 4))
   (lambda (y) (+ x y))))
(add4 6) ⇒ 10
```

The formal arguments list of a lambda expression has some extensions over standard Scheme: Kawa borrows the extended formal argument list of DSSSL, and allows you to declare the type of the parameter. More generally, you can use Section 8.3 [Variables and Patterns], page 140.

lambda-expression `::=` **(lambda** *formals option-pair * opt-return-type body***)**
opt-return-type `::=` [**::** *type*]
formals `::=` **(***formal-arguments***)** `|` *rest-arg*

An *opt-return-type* specifies the return type of the procedure: The result of evaluating the *body* is coerced to the specified *type*.

Deprecated: If 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 that is another way to specify the function's return type.

See Section 11.3 [Procedure properties], page 172, for how to set and use an *option-pair*.

The Section 8.4 [Definitions], page 141, form has a short-hand that combines a lambda definition with binding the lambda to a variable:

(define (`name formal-arguments`**)** `opt-return-type body`**)**

formal-arguments `::=` *required-or-guard ** [**#!optional** *optional-arg* ...] *rest-key-args*
rest-key-args `::=` [**#!rest** *rest-arg*] [**#!key** *key-arg* ...] [*guard*]
`|` [**#!key** *key-arg* ...] [*rest-parameter*] [*guard*]
`|` **.** *rest-arg*

When the procedure is applied to an [argument list], page 166, the latter is matched against formal parameters. This may involve some complex rules and pattern matching.

Required parameters

required-or-guard `::=` *required-arg* `|` *guard*
required-arg `::=` *pattern*
`|` **(** *pattern* **::** *type***)**

The *required-arg*s are matched against the actual (pre-keyword) arguments in order, starting with the first actual argument. It is an error if there are fewer pre-keyword arguments then there are *required-arg*s. While a *pattern* is most commonly an identifier, more complicated patterns are possible, thus more (or fewer) variables may be bound than there are arguments.

Note a *pattern* may include an *opt-type-specifier*. For example:

```
(define (isquare x::integer)
    (* x x))
```

In this case the actual argument is coerced to an `integer` and then the result matched against the pattern `x`. This is how parameter types are specified.

The *pattern* may be enclosed in parentheses for clarify (just like for optional parameters), but in that case the type specifier is required to avoid ambiguity.

Optional parameters

optional-arg `::=` *variable opt-type-specifier*
  `|` `(` *pattern opt-type-specifier* [*initializer* [*supplied-var*]]`)`
supplied-var `::=` *variable*

Next the *optional-arg*s are bound to remaining pre-keyword arguments. If there are fewer remaining pre-keyword arguments than there are *optional-arg*s, then the remaining *variable*s are bound to the corresponding *initializer*. If no *initializer* was specified, it defaults to `#f`. (TODO: If a *type* is specified the default for *initializer* is the default value of the *type*.) The *initializer* is evaluated in an environment in which all the previous formal parameters have been bound. If a *supplied-var* is specified, it has type boolean, and is set to true if there was an actual corresponding argument, and false if the initializer was evaluated.

Keyword parameters

key-arg `::=` *variable opt-type-specifier*
   `|` `(` *variable opt-type-specifier* [*initializer* [*supplied-var*]] `)`

Keyword parameters follow `#!key`. For each *variable* if there is an actual keyword parameter whose keyword matches *variable*, then *variable* is bound to the corresponding value. If there is no matching artual argument, then the *initializer* is evaluated and bound to the argument. If *initializer* is not specified, it defaults to `#f`. The *initializer* is evaluated in an environment in which all the previous formal parameters have been bound.

```
(define (fun x #!key (foo 1) (bar 2) (baz 3))
  (list x foo bar baz))
(fun 9 baz: 10 foo: 11) ⇒ (9 11 2 10)
```

The following cause a match failure, *unless* there is a rest parameter:

- There may not be extra non-keyword arguments (prefix or postfix) beyond those matched by required and optional parameters.
- There may not be any duplicated keyword arguments.
- All keyowrds in the actual argument list must match one of the keyword formal parameters.

It is not recommended to use both keyword parameters and a rest parameter that can match keyword arguments. Currently, the rest parameter will include any arguments that match the explicit keyword parameters, as well any that don't, though this may change.

On the other hand, it is fine to have both keyword parameters and a rest parameter does not accept keywords. In that case the rest parameter will match any "postfix" arguments:

```
#|kawa:8|# (define (fun x #!key k1 k2 #!rest r)
  (format "x:~w k1:~w k2:~w r:~w" x k1 k2 r))
(fun 3 k2: 12 100 101) ⇒ x:3 k1:#f k2:12 r:(100 101)
```

The *supplied-var* argument is as for optional arguments.

Performance note: Keyword parameters are implemented very efficiently and compactly when explicit in the code. The parameters are sorted by the compiler, and the actual keyword arguemnts at the call state are also sorted at compile-time. So keyword matching just requires a fast linear scan comparing the two sorted lists. This implementation is also very compact, compared to say a hash table.

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.

Rest parameters

A “rest parameter” matches any arguments not matched by other parameters. You can write it using any of the following ways:

> *rest-parameter* `::=`
>   **#!rest** *rest-arg* [**::** *type*]
>   `|` **@***rest-arg*
>   `|` **@:***rest-arg*
> *rest-arg* `::=` *variable*

In addition, if *formals* is just a *rest-arg* identifier, or a *formal-arguments* ends with `.` *`rest-arg`* (i.e. is a dotted list) that is equivalent to using `#!rest`.

These forms are similar but differ in the type of the *rest-arg* and whether keywords are allowed (as part of the *rest-arg*):

- If **#!rest** *rest-arg* is used with no *type* specifier (or a *type* specifier of `list`) then *rest-arg* is a list. Keywords are not allowed if `#!key` has been seen. (For backward compatibility, it is allowed to have extra keywords if `#!rest` is followed by `!key`.) If there are any keywords, then *rest-arg* is more specifically an `arglist`.
- If **#!rest** *rest-arg* is used with *type* specifier that is a Java array (for example `#!rest r::string[]` then *rest-arg* has that type. Each argument must be compatible with the element type of the array. Keywords are not allowed (even if *type* is `object[]`).

  The generated method will be compiled like a Java varargs methods if possible (i.e. no non-trivial patterns or keyword paremeters).
- Using **@***rest-arg* is equivalent to `#!rest` *`rest-arg`*`::object[]`: Keywords are not allowed; the type of *rest-arg* is a Java array; the method is compiled like a Java varargs method.
- For **@:***rest-arg* then *rest-arg* is a vector, specifically an `argvector`. Keywords are allowed.

Guards (conditional expressions)

A *guard* is evaluated when it appears in the formal parameter list. If it evaluates to false, then matching fails. Guards can appears before or after required arguments, or at the very end, after all other formal parameters.

11.3 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 preferred way to set a property is using an *option-pair* in a *lambda-expression*. For example, to set the `setter` property of a procedure to `my-set-car` do the following:

```
(define my-car
  (lambda (arg) setter: my-set-car (primitive-car arg)))
```

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` procedure.

`procedure-property` *proc key* [*default*] [Procedure]

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* [Procedure]

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)))
```

You can also specify properties in the lambda body:

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

11.3.1 Standard properties

`name`	The name of a procedure (as a symbol), which is used when the procedure is printed.
`setter`	Set the setter procedure associated with the procedure.

`validate-apply`
`validate-xapply`
Used during the validation phase of the compiler.

`compile-apply`
Used during the bytecode-generation phase of the compiler: If we see a call to a known function with this property, we can emit custom bytecode for the call.

11.4 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.

> **Warning:** 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, selecting between different Scheme procedures based on parameter types should be considered experimental. The main problem is we can't determine the most specific method, so Kawa just tries the methods in order.

`make-procedure` [*keyword: value*]... *method...* [Procedure]

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

The *keywords* 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)))
```

11.5 Partial application

`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.

12 Quantities and Numbers

Kawa supports the full Scheme set of number operations with some extensions.

Kawa converts between Scheme number types and Java number types as appropriate.

12.1 Numerical types

Mathematically, numbers are arranged into a tower of subtypes in which each level is a subset of the level before it: number; complex number; real number; rational number; integer.

For example, `3` is an integer. Therefore `3` is also a rational, a real, and a complex number. The same is true of the Scheme numbers that model 3. For Scheme numbers, these types are defined by the predicates `number?`, `complex?`, `real?`, `rational?`, and `integer?`.

There is no simple relationship between a number's type and its representation inside a computer. Although most implementations of Scheme will offer at least two different representations of 3, these different representations denote the same integer.

Scheme's numerical operations treat numbers as abstract data, as independent of their representation as possible. Although an implementation of Scheme may use multiple internal representations of numbers, this ought not to be apparent to a casual programmer writing simple programs.

`number` [Type]

The type of Scheme numbers.

`quantity` [Type]

The type of quantities optionally with units. This is a sub-type of `number`.

`complex` [Type]

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

`real` [Type]

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

`rational` [Type]

The type of exact rational numbers. This is a sub-type of `real`.

`integer` [Type]

The type of exact Scheme integers. This is a sub-type of `rational`.

Kawa allows working with expressions of “primitive” types, which are supported by the JVM without object allocation, and using builtin arithmetic. Using these types may be much faster, assuming the compiler is able to infer that the variable or expression has primitive type.

`long` [Type]
`int` [Type]
`short` [Type]
`byte` [Type]

These are fixed-sized primitive signed exact integer types, of respectively 64, 32, 18, and 8 bits. If a value of one of these types needs to be converted to an object, the standard classes `java.lang.Long`, `java.lang.Integer`, `java.lang.Short`, or `java.lang.Byte`, respectively, are used.

`ulong` [Type]
`uint` [Type]
`ushort` [Type]
`ubyte` [Type]

These are fixed-sized primitive unsigned exact integer types, of respectively 64, 32, 18, and 8 bits. These are presented at runtime using the corresponding signed types (`long`, `int`, `short`, or `byte`). However, for arithmetic the Kawa compiler generates code to perform the “mathematically correct” result, truncated to an unsigned result rather than signed. If a value of one of these types needs to be converted to an object, the classes `gnu.math.ULong`, `gnu.math.UInt`, `gnu.math.UShort`, or `gnu.math.UByte` is used.

`double` [Type]
`float` [Type]

These are fixed-size primitive inexact floating-point real types, using the standard 64-bit or 32-bit IEEE representation. If a value of one of these types needs to be converted to an object, the standard classes `java.lang.Double`, or `java.lang.Float` is used.

12.1.1 Exactness

It is useful to distinguish between numbers that are represented exactly and those that might not be. For example, indexes into data structures must be known exactly, as must some polynomial coefficients in a symbolic algebra system. On the other hand, the results of measurements are inherently inexact, and irrational numbers may be approximated by rational and therefore inexact approximations. In order to catch uses of inexact numbers where exact numbers are required, Scheme explicitly distinguishes exact from inexact numbers. This distinction is orthogonal to the dimension of type.

A Scheme number is *exact* if it was written as an exact constant or was derived from exact numbers using only exact operations. A number is *inexact* if it was written as an inexact constant, if it was derived using inexact ingredients, or if it was derived using inexact operations. Thus inexactness is a contagious property of a number. In particular, an *exact complex number* has an exact real part and an exact imaginary part; all other complex numbers are *inexact complex numbers*.

If two implementations produce exact results for a computation that did not involve inexact intermediate results, the two ultimate results will be mathematically equal. This is generally not true of computations involving inexact numbers since approximate methods such as floating-point arithmetic may be used, but it is the duty of the implementation to make the result as close as practical to the mathematically ideal result.

Rational operations such as `+` should always produce exact results when given exact arguments. If the operation is unable to produce an exact result, then it may either report the violation of an implementation restriction or it may silently coerce its result to an inexact value.

Except for `exact`, the operations described in this section must generally return inexact results when given any inexact arguments. An operation may, however, return an exact result if it can prove that the value of the result is unaffected by the inexactness of its arguments. For example, multiplication of any number by an exact zero may produce an exact zero result, even if the other argument is inexact.

Specifically, the expression `(* 0 +inf.0)` may return `0`, or `+nan.0`, or report that inexact numbers are not supported, or report that non-rational real numbers are not supported, or fail silently or noisily in other implementation-specific ways.

The procedures listed below will always return exact integer results provided all their arguments are exact integers and the mathematically expected results are representable as exact integers within the implementation: `-`, `*`, `+`, `abs`, `ceiling`, `denominator`, `exact-integer-sqrt`, `expt`, `floor`, `floor/`, `floor-quotient`, `floor-remainder`, `gcd`, `lcm`, `max`, `min`, `modulo`, `numerator`, `quotient`, `rationalize`, `remainder`, `square`, `truncate`, `truncate/`, `truncate-quotient`, `truncate-remainder`.

12.1.2 Numerical promotion and conversion

When combining two values of different numeric types, the values are converted to the first line in the following that subsumes (follows) both types. The computation is done using values of that type, and so is the result. For example adding a `long` and a `float` converts the former to the latter, yielding a `float`.

Note that `short`, `byte`, `ushort`, `ubyte` are converted to `int` regardless, even in the case of a single-operand operation, such as unary negation. Another exception is trancendental functions (such as `cos`), where integer operands are converted to `double`.

- `int` subsumes `short`, `byte`, `ushort`, `ubyte`.
- `uint`
- `long`
- `ulong`
- `java.lang.BigInteger`
- `integer` (i.e. `gnu.math.IntNum`)
- `rational` (i.e. `gnu.math.RatNum`)
- `float`
- `double`
- `gnu.math.FloNum`
- `real` (i.e. `gnu.math.RealNum`)
- `number`
- `complex`
- `quantity`

When comparing a primitive signed integer value with a primitive unsigned integer (for example `<` applied to a `int` and a `ulong`) the mathematically correct result is computed, as it converting both operands to `integer`.

12.2 Arithmetic operations

real-valued? *obj* [Procedure]
rational-valued? *obj* [Procedure]
integer-valued? *obj* [Procedure]

These numerical type predicates can be applied to any kind of argument. The `real-valued?` procedure returns `#t` if the object is a number object and is equal in the sense of `=` to some real number object, or if the object is a NaN, or a complex number object whose real part is a NaN and whose imaginary part is zero in the sense of `zero?`. The `rational-valued?` and `integer-valued?` procedures return `#t` if the object is a number object and is equal in the sense of `=` to some object of the named type, and otherwise they return `#f`.

```
(real-valued? +nan.0)             ⇒ #t
(real-valued? +nan.0+0i)          ⇒ #t
(real-valued? -inf.0)             ⇒ #t
(real-valued? 3)                  ⇒ #t
(real-valued? -2.5+0.0i)          ⇒ #t

(real-valued? -2.5+0i)            ⇒ #t
(real-valued? -2.5)               ⇒ #t
(real-valued? #e1e10)             ⇒ #t
```

```
(rational-valued? +nan.0)           ⇒ #f
(rational-valued? -inf.0)           ⇒ #f
(rational-valued? 6/10)             ⇒ #t
(rational-valued? 6/10+0.0i)        ⇒ #t
(rational-valued? 6/10+0i)          ⇒ #t
(rational-valued? 6/3)              ⇒ #t

(integer-valued? 3+0i)              ⇒ #t
(integer-valued? 3+0.0i)            ⇒ #t
(integer-valued? 3.0)               ⇒ #t
(integer-valued? 3.0+0.0i)          ⇒ #t
(integer-valued? 8/4)               ⇒ #t
```

Note: These procedures test whether a given number object can be coerced to the specified type without loss of numerical accuracy. Specifically, the behavior of these predicates differs from the behavior of `real?`, `rational?`, and `integer?` on complex number objects whose imaginary part is inexact zero.

Note: The behavior of these type predicates on inexact number objects is unreliable, because any inaccuracy may affect the result.

exact-integer? *z* [Procedure]

Returns `#t` if *z* is both exact and an integer; otherwise returns `#f`.

```
(exact-integer? 32)                 ⇒ #t
(exact-integer? 32.0)               ⇒ #t
(exact-integer? 32/5)               ⇒ #f
```

finite? *z* [Procedure]

Returns `#t` if *z* is finite real number (i.e. an infinity and not a NaN), or if *z* is a complex number whose real and imaginary parts are both finite.

```
(finite? 3)             ⇒ #t
(finite? +inf.0)        ⇒ #f
(finite? 3.0+inf.0i)    ⇒ #f
```

infinite? *z* [Procedure]

Return `#t` if *z* is an infinite real number (`+int.0` or `-inf.0`), or if *z* is a complex number where either real or imaginary parts or both are infinite.

```
(infinite? 5.0)         ⇒ #f
(infinite? +inf.0)      ⇒ #t
(infinite? +nan.0)      ⇒ #f
(infinite? 3.0+inf.0i)  ⇒ #t
```

nan? *z* [Procedure]

For a real numer returns whether its is a NaN; for a complex number if the real or imaginary parts or both is a NaN.

```
(nan? +nan.0)           ⇒ #t
(nan? 32)               ⇒ #f
(nan? +nan.0+5.0i)      ⇒ #t
(nan? 1+2i)             ⇒ #f
```

`+` *z* ... [Procedure]
`*` *z* ... [Procedure]

These procedures return the sum or product of their arguments.

```
(+ 3 4)                          ⇒  7
(+ 3)                            ⇒  3
(+)                              ⇒  0
(+ +inf.0 +inf.0)                ⇒  +inf.0
(+ +inf.0 -inf.0)                ⇒  +nan.0

(* 4)                            ⇒  4
(*)                              ⇒  1
(* 5 +inf.0)                     ⇒  +inf.0
(* -5 +inf.0)                    ⇒  -inf.0
(* +inf.0 +inf.0)                ⇒  +inf.0
(* +inf.0 -inf.0)                ⇒  -inf.0
(* 0 +inf.0)                     ⇒  +nan.0
(* 0 +nan.0)                     ⇒  +nan.0
(* 1.0 0)                        ⇒  0.0
```

For any real number object *x* that is neither infinite nor NaN:

```
(+ +inf.0 x)                   ⇒  +inf.0
(+ -inf.0 x)                   ⇒  -inf.0
```

For any real number object *x*:

```
(+ +nan.0 x)                   ⇒  +nan.0
```

For any real number object *x* that is not an exact 0:

```
(* +nan.0 x)                   ⇒  +nan.0
```

The behavior of `-0.0` is illustrated by the following examples:

```
(+  0.0 -0.0)  ⇒  0.0
(+ -0.0  0.0)  ⇒  0.0
(+  0.0  0.0)  ⇒  0.0
(+ -0.0 -0.0)  ⇒ -0.0
```

`-` *z* [Procedure]
`-` z_1 z_2 z_3 ... [Procedure]

With two or more arguments, this procedures returns the difference of its arguments, associating to the left. With one argument, however, it returns the negation (additive inverse) of its argument.

```
(- 3 4)                              ⇒  -1
(- 3 4 5)                            ⇒  -6
(- 3)                                ⇒  -3
(- +inf.0 +inf.0)                    ⇒  +nan.0
```

The behavior of `-0.0` is illustrated by the following examples:

```
(-  0.0)       ⇒ -0.0
(- -0.0)       ⇒  0.0
(-  0.0 -0.0)  ⇒  0.0
(- -0.0  0.0)  ⇒ -0.0
(-  0.0  0.0)  ⇒  0.0
(- -0.0 -0.0)  ⇒  0.0
```

`/` *z* [Procedure]
`/` z_1 z_2 z_3 ... [Procedure]

If all of the arguments are exact, then the divisors must all be nonzero. With two or more arguments, this procedure returns the quotient of its arguments, associating to the left. With one argument, however, it returns the multiplicative inverse of its argument.

```
(/ 3 4 5)                  ⇒  3/20
(/ 3)                      ⇒  1/3
(/ 0.0)                    ⇒  +inf.0
(/ 1.0 0)                  ⇒  +inf.0
(/ -1 0.0)                 ⇒  -inf.0
(/ +inf.0)                 ⇒  0.0
(/ 0 0)                    ⇒  exception &assertion
(/ 3 0)                    ⇒  exception &assertion
(/ 0 3.5)                  ⇒  0.0
(/ 0 0.0)                  ⇒  +nan.0
(/ 0.0 0)                  ⇒  +nan.0
(/ 0.0 0.0)                ⇒  +nan.0
```

If this procedure is applied to mixed non–rational real and non–real complex arguments, it either raises an exception with condition type `&implementation-restriction` or returns an unspecified number object.

`floor/` *x y* [Procedure]
`truncate/` *x y* [Procedure]
`div-and-mod` *x y* [Procedure]
`div0-and-mod0` *x y* [Procedure]

These procedures implement number–theoretic integer division. They accept two real numbers *x* and *y* as operands, where *y* must be nonzero. In all cases the result is two values *q* (an integer) and *r* (a real) that satisfy the equations:

```
x = q * y + r
q = rounding-op(x/y)
```

The result is inexact if either argument is inexact.

For `floor/` the *rounding-op* is the `floor` function (below).

```
(floor/ 123 10)         ⇒  12 3
(floor/ 123 -10)        ⇒  -13 -7
(floor/ -123 10)        ⇒  -13 7
(floor/ -123 -10)       ⇒  12 -3
```

For `truncate/` the *rounding-op* is the `truncate` function.

```
(truncate/ 123 10)      ⇒  12 3
(truncate/ 123 -10)     ⇒  -12 3
(truncate/ -123 10)     ⇒  -12 -3
(truncate/ -123 -10)    ⇒  12 -3
```

For `div-and-mod` the *rounding-op* is either `floor` (if y is positive) or `ceiling` (if y is negative). We have:

```
0  <= r < |y|
```

```
(div-and-mod 123 10)     ⇒  12 3
(div-and-mod 123 -10)    ⇒  -12 3
(div-and-mod -123 10)    ⇒  -13 7
(div-and-mod -123 -10)   ⇒  13 7
```

For `div0-and-mod0` the *rounding-op* is the `round` function, and `r` lies within a half–open interval centered on zero.

```
-|y/2| <= r < |y/2|
```

```
(div0-and-mod0 123 10)    ⇒  12 3
(div0-and-mod0 123 -10)   ⇒  -12 3
(div0-and-mod0 -123 10)   ⇒  -12 -3
(div0-and-mod0 -123 -10)  ⇒  12 -3
(div0-and-mod0 127 10)    ⇒  13 -3
(div0-and-mod0 127 -10)   ⇒  -13 -3
(div0-and-mod0 -127 10)   ⇒  -13 3
(div0-and-mod0 -127 -10)  ⇒  13 3
```

The inconsistent naming is for historical reasons: `div-and-mod` and `div0-and-mod0` are from R6RS, while `floor/` and `truncate/` are from R7RS.

`floor-quotient` *x y* [Procedure]
`truncate-quotient` *x y* [Procedure]
`div` *x y* [Procedure]
`div0` *x y* [Procedure]

These procedures return the quotient part (first value) of respectively `floor/`, `truncate/`, `div-and-mod`, and `div0-and-mod0`.

`floor-remainder` *x y* [Procedure]
`truncate-remainder` *x y* [Procedure]
`mod` *x y* [Procedure]
`mod0` *x y* [Procedure]

These procedures return the remainder part (second value) of respectively `floor/`, `truncate/`, `div-and-mod`, and `div0-and-mod0`.

As a Kawa extension y may be zero, in which case the result is x:

```
(mod 123 0)      ⇒  123 ;; Kawa extension
```

`quotient` *x y* [Procedure]
`remainder` *x y* [Procedure]
`modulo` *x y* [Procedure]

These are equivalent to `truncate-quotient`, `truncate-remainder`, and `floor-remainder`, respectively. These are provided for backward compatibility.

```
(remainder 13 4)     ⇒ 1
(remainder -13 4)    ⇒ -1
(remainder 13 -4)    ⇒ 1
(remainder -13 -4)   ⇒ -1
```

```
(remainder -13 -4.0) ⇒ -1.0
(modulo 13 4)    ⇒ 1
(modulo -13 4)   ⇒ 3
(modulo 13 -4)   ⇒ -4
(modulo -13 -4)  ⇒ -1
```

abs *x* [Procedure]

Returns the absolute value of its argument.

```
(abs -7)                         ⇒  7
(abs -inf.0)                     ⇒  +inf.0
```

gcd *n_1 . . .* [Procedure]

lcm *n_1 . . .* [Procedure]

These procedures return the greatest common divisor or least common multiple of their arguments. The result is always non–negative. The arguments must be integers; if an argument is inexact, so is the result.

```
(gcd 32 -36)                     ⇒  4
(gcd)                            ⇒  0
(lcm 32 -36)                     ⇒  288
(lcm 32.0 -36)                   ⇒  288.0 ; inexact
(lcm)                            ⇒  1
```

numerator *q* [Procedure]

denominator *q* [Procedure]

These procedures return the numerator or denominator of their argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of `0` is defined to be `1`. The arguments must be integers; if an argument is inexact, so is the result.

```
(numerator   (/ 6 4))            ⇒  3
(denominator (/ 6 4))            ⇒  2
(denominator (inexact (/ 6 4)))        ⇒  2.0
```

floor *x* [Procedure]

ceiling *x* [Procedure]

truncate *x* [Procedure]

round *x* [Procedure]

These procedures return inexact integer objects for inexact arguments that are not infinities or NaNs, and exact integer objects for exact rational arguments.

`floor` Returns the largest integer object not larger than *x*.

`ceiling` Returns the smallest integer object not smaller than *x*.

`truncate` Returns the integer object closest to *x* whose absolute value is not larger than the absolute value of *x*.

`round` Returns the closest integer object to *x*, rounding to even when *x* represents a number halfway between two integers.

If the argument to one of these procedures is inexact, then the result is also inexact. If an exact value is needed, the result should be passed to the `exact` procedure.

Although infinities and NaNs are not integer objects, these procedures return an infinity when given an infinity as an argument, and a NaN when given a NaN.

```
(floor -4.3)                 ⇒  -5.0
(ceiling -4.3)               ⇒  -4.0
(truncate -4.3)              ⇒  -4.0
(round -4.3)                 ⇒  -4.0

(floor 3.5)                  ⇒  3.0
(ceiling 3.5)                ⇒  4.0
(truncate 3.5)               ⇒  3.0
(round 3.5)                  ⇒  4.0

(round 7/2)                  ⇒  4
(round 7)                    ⇒  7

(floor +inf.0)               ⇒  +inf.0
(ceiling -inf.0)             ⇒  -inf.0
(round +nan.0)               ⇒  +nan.0
```

rationalize x_1 x_2 [Procedure]

The `rationalize` procedure returns a number object representing the *simplest* rational number differing from x_1 by no more than x_2.

A rational number *r_1* is *simpler* than another rational number *r_2* if `r_1 = p_1/q_1` and `r_2 = p_2/q_2` (in lowest terms) and `|p_1| <= |p_2|` and `|q_1| <= |q_2|`. Thus `3/5` is simpler than `4/7`.

Although not all rationals are comparable in this ordering (consider `2/7` and `3/5`) any interval contains a rational number that is simpler than every other rational number in that interval (the simpler `2/5` lies between `2/7` and `3/5`).

Note that `0 = 0/1` is the simplest rational of all.

```
(rationalize (exact .3) 1/10)       ⇒ 1/3
(rationalize .3 1/10)               ⇒ #i1/3  ; approximately

(rationalize +inf.0 3)              ⇒  +inf.0
(rationalize +inf.0 +inf.0)         ⇒  +nan.0
```

The first two examples hold only in implementations whose inexact real number objects have sufficient precision.

exp z [Procedure]
log z [Procedure]
log z_1 z_2 [Procedure]
sin z [Procedure]
cos z [Procedure]
tan z [Procedure]
asin z [Procedure]

`acos` *z* [Procedure]
`atan` *z* [Procedure]
`atan` *x1 x2* [Procedure]

These procedures compute the usual transcendental functions.

The `exp` procedure computes the base–*e* exponential of *z*. The `log` procedure with a single argument computes the natural logarithm of *z* (**not** the base–10 logarithm); `(log` z_1 z_2`)` computes the base–z_2 logarithm of z_1.

The `asin`, `acos`, and `atan` procedures compute arcsine, arccosine, and arctangent, respectively. The two–argument variant of `atan` computes:

```
(angle (make-rectangular x2 x1))
```

These procedures may return inexact results even when given exact arguments.

```
(exp +inf.0)     ⇒ +inf.0
(exp -inf.0)     ⇒ 0.0
(log +inf.0)     ⇒ +inf.0
(log 0.0)        ⇒ -inf.0
(log 0)          ⇒ exception &assertion
(log -inf.0)     ⇒ +inf.0+3.141592653589793i    ; approximately
(atan -inf.0)    ⇒ -1.5707963267948965          ; approximately
(atan +inf.0)    ⇒ 1.5707963267948965           ; approximately
(log -1.0+0.0i)  ⇒ 0.0+3.141592653589793i       ; approximately
(log -1.0-0.0i)  ⇒ 0.0-3.141592653589793i       ; approximately
                                                ; if -0.0 is distinguished
```

`sinh` *z* [Procedure]
`cosh` *z* [Procedure]
`tanh` *z* [Procedure]
`asinh` *z* [Procedure]
`acosh` *z* [Procedure]
`atanh` *z* [Procedure]

The hyperbolic functions.

`square` *z* [Procedure]

Returns the square of *z*. This is equivalent to `(* z z)`.

```
(square 42)     ⇒ 1764
(square 2.0)    ⇒ 4.0
```

`sqrt` *z* [Procedure]

Returns the principal square root of *z*. For rational *z*, the result has either positive real part, or zero real part and non–negative imaginary part. The value of `(sqrt z)` could be expressed as:

```
e^((log z)/2)
```

The `sqrt` procedure may return an inexact result even when given an exact argument.

```
(sqrt -5)              ⇒  0.0+2.23606797749979i ; approximately
(sqrt +inf.0)          ⇒  +inf.0
(sqrt -inf.0)          ⇒  +inf.0i
```

Note that if the argument is a primitive number (such as `double`) or an instance of the corresponding boxed class (such as `java.lang.Double`) then we use the real-number version of `sqrt`:

```
(sqrt (->double -5))        ⇒  NaN
```

That is, we get different a result for `java.lang.Double` and `gnu.math.DFloNum`, even for arguments that are numerically equal in the sense of `=`. This is so that the compiler can use the `java.lang.Math.sqrt` method without object allocation when the argument is a `double` (and because we want `double` and `java.lang.Double` to behave consistently).

`exact-integer-sqrt` *k* [Procedure]

The `exact-integer-sqrt` procedure returns two non–negative exact integer objects *s* and *r* where `k = s^2 + r` and `k < (s+1)^2`.

```
(exact-integer-sqrt 4)  ⇒ 2 0 ; two return values
(exact-integer-sqrt 5)  ⇒ 2 1 ; two return values
```

`expt` *z1 z2* [Procedure]

Returns z_1 raised to the power z_2. For nonzero z_1, this is ${z_1}^{z_2} = e^{z_2 \log z_1}$. The value of 0^z is 1 if `(zero? z)`, 0 if `(real-part z)` is positive, and an error otherwise. Similarly for 0.0^z, with inexact results.

12.3 Numerical input and output

`number->string` *z* [*radix*] [Procedure]

The procedure `number->string` takes a number and a radix and returns as a string an external representation of the given number in the given radix such that

```
(let ((number number)
      (radix radix))
  (eqv? number
        (string->number (number->string number radix)
                        radix)))
```

is true. It is an error if no possible result makes this expression true.

If present, *radix* must be an exact integer in the range 2 to 36, inclusive. If omitted, *radix* defaults to 10.

If *z* is inexact, the *radix* is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression; otherwise the format of the result is unspecified.

The result returned by `number->string` never contains an explicit radix prefix.

Note: The error case can occur only when *z* is not a complex number or is a complex number with a non-rational real or imaginary part.

Rationale: If *z* is an inexact number and the *radix* is 10, then the above expression is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and unusual representations.

string->number *string* [*radix*] [Procedure]

Returns a number of the maximally precise representation expressed by the given *string*. It is an error if *radix* is not an exact integer in the range 2 to 26, inclusive.

If supplied, *radix* is a default radix that will be overridden if an explicit radix prefix is present in the string (e.g. `"#o177"`). If *radix* is not supplied, then the default *radix* is 10. If *string* is not a syntactically valid notation for a number, or would result in a number that the implementation cannot represent, then `string->number` returns `#f`. An error is never signaled due to the content of *string*.

```
(string->number "100")       ⇒  100
(string->number "100" 16)    ⇒  256
(string->number "1e2")       ⇒  100.0
(string->number "#x100" 10) ⇒  256
```

12.4 Quaternions

Kawa extends the Scheme numeric tower to include quaternions (`http://en.wikipedia.org/wiki/Quaternion`) as a proper superset of the complex numbers. Quaternions provide a convenient notation to represent rotations in three-dimensional space (`http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation`), and are therefore commonly found in applications such as computer graphics, robotics, and spacecraft engineering. The Kawa quaternion API is modeled after this (`http://www.ccs.neu.edu/home/dorai/squat/squat.html`) with some additions.

A quaternion is a number that can be expressed in the form '`w+xi+yj+zk`', where `w`, `x`, `y`, and `z` are real, and `i`, `j`, and `k` are imaginary units satisfying $i^2 = j^2 = k^2 = ijk = -1$. The magnitude of a quaternion is defined to be its Euclidean norm when viewed as a point in R^4.

The real–part of a quaternion is also called its '`scalar`', while the i–part, j–part, and k–part taken together are also called its '`vector`'. A quaternion with zero j–part and k–part is an ordinary complex number. (If the i–part is also zero, then it is a real). A quaternion with zero real–part is called a '`vector quaternion`'.

The reader syntax for number literals has been extended to support both rectangular and polar (hyperspherical) notation for quaternions. The rectangular notation is as above, i.e. `w+xi+yj+zk`. The polar notation takes the form `r@t%u&v`, where `r` is the magnitude, `t` is the first angle, and `u` and `v` are two other angles called the "colatitude" and "longitude".

The rectangular coordinates and polar coordinates are related by the equations:

```
w = r * cos t
x = r * sin t * cos u
y = r * sin t * sin u * cos v
z = r * sin t * sin u * sin v
```

With either notation, zero elements may be omitted.

make-rectangular *w x* [Procedure]
make-rectangular *w x y z* [Procedure]

These procedures construct quaternions from Cartesian coordinates.

`make-polar` *r t* [Procedure]
`make-polar` *r t u v* [Procedure]

These procedures construct quaternions from polar coordinates.

`+` *q* ... [Procedure]
`-` *q* ... [Procedure]
`*` *q* ... [Procedure]
`/` *q* [Procedure]
`/` q_1 q_2 q_3 ... [Procedure]
`expt` q_1 q_2 [Procedure]
`exp` *q* [Procedure]
`log` *q* [Procedure]
`sqrt` *q* [Procedure]
`sin` *q* [Procedure]
`cos` *q* [Procedure]
`tan` *q* [Procedure]
`asin` *q* [Procedure]
`acos` *q* [Procedure]
`atan` *q* [Procedure]

All of the arithmetic and transcendental functions defined for complex arguments have been extended to support quaternions.

Quaternion multiplication is not commutative, so there are two possible interpretations of `(/ q1 q2)` which would yield different results: either `(* q1 (/ q2))`, or `(* (/ q2) q1)`. Division in this implementation has been defined such that `(/ q1 q2 ...)` is equivalent to `(* q1 (/ q2) ...)`, but it is recommended to use reciprocals (unary `/`) and multiplication.

`real-part` *q* [Procedure]

Return the real–part of *q*.

```
(real-part 0)          ⇒  0
(real-part -i)         ⇒  0
(real-part 1+2i-3j+4k) ⇒  1
```

`imag-part` *q* [Procedure]

Return the i–part of *q*.

```
(imag-part 0)          ⇒  0
(imag-part -i)         ⇒  -1
(imag-part 1+2i-3j+4k) ⇒  2
```

`magnitude` *q* [Procedure]

Return the Euclidean norm of *q*. If *q* is `a+bi+cj+dk`, then `(magnitude q)` is `(sqrt (apply + (map square (list a b c d))))`

`angle` *q* [Procedure]

Return the angle of *q*.

12.4.1 The `(kawa quaternions)` module

The following additional functionality is made available by doing one of:

```
(require 'quaternions) ;; or
(import (kawa quaternions))
```

`quaternion` [Alias]

An alias for `gnu.math.Quaternion`, useful for type declarations.

`quaternion?` *x* [Procedure]

Return `#t` if *x* is a quaternion, i.e. an ordinary number, and `#f` otherwise.

```
(quaternion? 0)           ⇒  #t
(quaternion? -i)          ⇒  #t
(quaternion? 1+2i-3j+4k)  ⇒  #t
(quaternion? 10.0m)       ⇒  #f
(quaternion? "x")         ⇒  #f
```

`jmag-part` *q* [Procedure]

Return the j–part of *q*.

```
(jmag-part 0)           ⇒  0
(jmag-part -i)          ⇒  0
(jmag-part 1+2i-3j+4k)  ⇒  -3
```

`kmag-part` *q* [Procedure]

```
(kmag-part 0)           ⇒  0
(kmag-part -i)          ⇒  0
(kmag-part 1+2i-3j+4k)  ⇒  4
```

`complex-part` *q* [Procedure]

Return the projection of *q* into the complex plane: `(+ (real-part q) (* +i (imag-part q)))`

```
(complex-part 0)           ⇒  0
(complex-part -i)          ⇒  -1i
(complex-part 1+2i-3j+4k)  ⇒  1+2i
```

`vector-part` *q* [Procedure]

Return the vector–part of *q*.

```
(vector-part 0)           ⇒  0
(vector-part -i)          ⇒  -1i
(vector-part 1+2i-3j+4k)  ⇒  +2i-3j+4k
```

`unit-quaternion` *q* [Procedure]

Return a quaternion of unit magnitude with the same direction as *q*. If *q* is zero, return zero. This is like a 4D version of a signum function.

```
(unit-quaternion 0)           ⇒  0
(unit-quaternion -i)          ⇒  -1i
(unit-quaternion 1+2i-3j+4k)  ⇒  0.18257418583505536+0.3651483716701107i-0.547722
```

`unit-vector` *q* [Procedure]

Return the vector–part of *q*, scaled to have magnitude 1. If the vector–part is zero, then return zero.

```
(unit-vector 0)          ⇒  0
(unit-vector -i)         ⇒  -1i
(unit-vector 1+2i-3j+4k) ⇒  +0.3713906763541037i-0.5570860145311556j+0.742781352
```

`colatitude` *q* [Procedure]

Return the colatitude of *q*.

`longitude` *q* [Procedure]

Return the longitude of *q*.

`vector-quaternion?` *obj* [Procedure]

Return `#t` if *obj* is a vector quaternion, i.e. a quaternion with zero real–part.

`make-vector-quaternion` *x y z* [Procedure]

Construct vector quaternion `xi+yj+zk`. This is equivalent to `(make-rectangular 0 x y z)`.

`vector-quaternion->list` *vq* [Procedure]

Return a newly allocated list of the x, y, and z components of *vq*. This is equivalent to `(list (imag-part vq) (jmag-part vq) (kmag-part vq))`.

`dot-product` q_1 q_2 [Procedure]

For two vector quaternions q_1 = `ai+bj+ck` and q_2 = `di+ej+fk`, return `ad + be + cf`. This is equal to the R^3 dot product for vectors (a, b, c) and (d, e, f), and is also equal to `(- (real-part (* q1 q2)))`. It is an error if either q_1 or q_2 has a non-zero real–part.

`cross-product` q_1 q_2 [Procedure]

For two vector quaternions q_1 = `ai+bj+ck` and q_2 = `di+ej+fk`, return the R^3 cross product for vectors (a, b, c) and (d, e, f), which is equal to `(vector-part (* q1 q2))`. It is an error if either q_1 or q_2 has a non-zero real–part.

`conjugate` *q* [Procedure]

Return `(+ (real-part q) (* -1 (vector-part q)))`.

```
(conjugate 0)          ⇒  0
(conjugate -i)         ⇒  +1i
(conjugate 1+2i-3j+4k) ⇒  1-2i+3j-4k
```

12.4.2 The `(kawa rotations)` module

The `(kawa rotations)` library provides a set of functions which use unit quaternions to represent 3D spatial rotations. To use these functions, the library must be imported:

```
(import (kawa rotations))
```

These functions normalize their quaternion inputs as needed to be of length 1.

12.4.2.1 Rotation Representation Conversions

Conversions to and from several alternate representations of rotations are supported.

The set of unit quaternions provides a double covering of all possible 3D rotations: `q` and `-q` represent the same rotation. Most other representations also have multiple numerical values which map to the same rotation (for example, the rotation about `axis-vec` by `angle` is the same as the rotation about `-axis-vec` by `-angle+2pi`). Therefore, these functions do not necessarily act as inverses in the sense of `equal?`. Furthermore, rotations involve trigonometric functions, so there will typically be some floating point error: `(acos (cos 0.1))` returns 0.09999999999999945, which is very close to 0.1 but not exact.

Rotation Matrices

`quaternion->rotation-matrix` *q* [Procedure]
`rotation-matrix->quaternion` *m* [Procedure]

The `quaternion->rotation-matrix` procedure returns a 3x3 rotation matrix representing the same rotation as *q*. The rotation matrix is instantiated as a Section 14.8 [Arrays], page 247, backed by an Section 14.4 [Uniform vectors], page 239.

The `rotation-matrix->quaternion` procedure performs the reverse operation, producing an equivalent unit quaternion for the rotation matrix (multi-dimensional array) *m*.

```
(rotation-matrix->quaternion (quaternion->rotation-matrix -1)) ⇒ 1.0
```

Axis-Angle Representation

`rotation-axis` *q* [Procedure]
`rotation-angle` *q* [Procedure]
`rotation-axis/angle` *q* [Procedure]

The `rotation-axis` procedure returns the axis of rotation of the quaternion *q* as a unit-length vector quaternion. If the axis of rotation is not well-defined (the angle of rotation is 0), then `+i` is arbitrarily chosen as the axis.

The `rotation-angle` procedure returns the corresponding angle of rotation. Note that this is not the same as the result of the `angle` procedure.

The `rotation-axis/angle` procedure returns the rotation axis and angle as multiple values.

```
(let* ((q 1/2+1/2i+1/2j+1/2k)
       (ar (rotation-angle q))
       (ad (java.lang.Math:toDegrees ar))
       (exact-ad (exact ad)))
  (rationalize exact-ad 1/10)) ⇒ 120
```

`make-axis/angle` *axis-vec angle* [Procedure]
`make-axis/angle` *axis-x axis-y axis-z angle* [Procedure]

The `make-axis/angle` procedure returns a quaternion representing the given axis/angle rotation. The axis is specified as either a single vector quaternion argument *axis-vec*, or as three reals *axis-x*, *axis-y*, and *axis-z*.

`rotx` *angle* [Procedure]
`roty` *angle* [Procedure]
`rotz` *angle* [Procedure]

The procedures `rotx`, `roty`, and `rotz` return quaternions representing rotations about the X-, Y-, and Z-axes.

Intrinsic Angle Sets

The intrinsic angle sets represent arbitrary rotations as a sequence of three rotations about coordinate frame axes attached to the rotating body (i.e. the axes rotate with the body).

There are twelve possible angle sets which neatly divide into two groups of six. The six with same first and third axes are also known as "Euler angles". The six with different first and third axes are also known as "Tait-Bryan angles".

`intrinsic-xyx` *q* [Procedure]
`intrinsic-xzx` *q* [Procedure]
`intrinsic-yxy` *q* [Procedure]
`intrinsic-yzy` *q* [Procedure]
`intrinsic-zxz` *q* [Procedure]
`intrinsic-zyz` *q* [Procedure]

These functions decompose the rotation represented by *q* into Euler angles of the given set (XYX, XZX, YXY, YZY, ZXZ, or ZYZ) and returns the three angles as multiple values. The middle angle will be in the range [0,pi]. If it is on the edges of that range (within 1.0E-12 of 0 or pi), such that the first and third axes are colinear, then the first angle will be set to 0.

```
(intrinsic-zyz (* (rotz 0.3) (roty 0.8) (rotz -0.6))) ⇒ 0.3000000000000001 0.799
```

`euler-xyx` [Alias]
`euler-xzx` [Alias]
`euler-yxy` [Alias]
`euler-yzy` [Alias]
`euler-zxz` [Alias]
`euler-zyz` [Alias]

Aliases for the corresponding `intrinsic-` procedures.

`intrinsic-xyz` *q* [Procedure]
`intrinsic-xzy` *q* [Procedure]
`intrinsic-yxz` *q* [Procedure]
`intrinsic-yzx` *q* [Procedure]
`intrinsic-zxy` *q* [Procedure]
`intrinsic-zyx` *q* [Procedure]

These functions decompose the rotation represented by *q* into Tait-Bryan angles of the given set (XYZ, XZY, YXZ, YZX, ZXY, or ZYX) and returns the three angles as multiple values. The middle angle will be in the range [-pi/2,pi/2]. If it is on the edges of that range, such that the first and third axes are colinear, then the first angle will be set to 0.

`tait-bryan-xyz` [Alias]
`tait-bryan-xzy` [Alias]

`tait-bryan-yxz` [Alias]
`tait-bryan-yzx` [Alias]
`tait-bryan-zxy` [Alias]
`tait-bryan-zyx` [Alias]

Aliases for the corresponding `intrinsic-` procedures.

`make-intrinsic-xyx` *alpha beta gamma* [Procedure]
`make-intrinsic-xzx` *alpha beta gamma* [Procedure]
`make-intrinsic-yxy` *alpha beta gamma* [Procedure]
`make-intrinsic-yzy` *alpha beta gamma* [Procedure]
`make-intrinsic-zxz` *alpha beta gamma* [Procedure]
`make-intrinsic-zyz` *alpha beta gamma* [Procedure]

These functions return quaternions representing the given Euler angle rotations.

`make-euler-xyx` [Alias]
`make-euler-xzx` [Alias]
`make-euler-yxy` [Alias]
`make-euler-yzy` [Alias]
`make-euler-zxz` [Alias]
`make-euler-zyz` [Alias]

Aliases for the corresponding `make-intrinsic-` procedures.

```
(let-values (((a b c) (euler-xyx (make-euler-xyx 1.0 0.0 2.0))))
  (list a b c)) ⇒ (0.0 0.0 3.0)
```

`make-intrinsic-xyz` *alpha beta gamma* [Procedure]
`make-intrinsic-xzy` *alpha beta gamma* [Procedure]
`make-intrinsic-yxz` *alpha beta gamma* [Procedure]
`make-intrinsic-yzx` *alpha beta gamma* [Procedure]
`make-intrinsic-zxy` *alpha beta gamma* [Procedure]
`make-intrinsic-zyx` *alpha beta gamma* [Procedure]

These functions return quaternions representing the given Tait-Bryan angle rotations.

`make-tait-bryan-xyz` [Alias]
`make-tait-bryan-xzy` [Alias]
`make-tait-bryan-yxz` [Alias]
`make-tait-bryan-yzx` [Alias]
`make-tait-bryan-zxy` [Alias]
`make-tait-bryan-zyx` [Alias]

Aliases for the corresponding `make-intrinsic-` procedures.

Extrinsic Angle Sets

The extrinsic angle sets represent arbitrary rotations as a sequence of three rotations about fixed-frame axes (i.e. the axes do not rotate with the body).

There are twelve possible extrinsic angle sets, and each is the dual of an intrinsic set. The extrinsic rotation about axes `A`, `B`, and `C` by angles `a`, `b`, and `c` is the same as the intrinsic rotation about axes `C`, `B`, and `A` by angles `c`, `b`, and `a`, with the order of the three axes reversed.

`extrinsic-xyx` *q* [Procedure]
`extrinsic-xyz` *q* [Procedure]
`extrinsic-xzx` *q* [Procedure]
`extrinsic-zxy` *q* [Procedure]
`extrinsic-yxy` *q* [Procedure]
`extrinsic-yxz` *q* [Procedure]
`extrinsic-yzx` *q* [Procedure]
`extrinsic-yzy` *q* [Procedure]
`extrinsic-zxy` *q* [Procedure]
`extrinsic-zxz` *q* [Procedure]
`extrinsic-zyx` *q* [Procedure]
`extrinsic-zyz` *q* [Procedure]

These functions decompose the rotation represented by *q* into extrinsic angles of the given set and returns the three angles as multiple values.

`make-extrinsic-xyx` *gamma beta alpha* [Procedure]
`make-extrinsic-xyz` *gamma beta alpha* [Procedure]
`make-extrinsic-xzx` *gamma beta alpha* [Procedure]
`make-extrinsic-xzy` *gamma beta alpha* [Procedure]
`make-extrinsic-yxy` *gamma beta alpha* [Procedure]
`make-extrinsic-yxz` *gamma beta alpha* [Procedure]
`make-extrinsic-yzx` *gamma beta alpha* [Procedure]
`make-extrinsic-yzy` *gamma beta alpha* [Procedure]
`make-extrinsic-zxy` *gamma beta alpha* [Procedure]
`make-extrinsic-zxz` *gamma beta alpha* [Procedure]
`make-extrinsic-zyx` *gamma beta alpha* [Procedure]
`make-extrinsic-zyz` *gamma beta alpha* [Procedure]

These functions return quaternions representing the given extrinsic angle rotations.

`rpy` [Alias]
`make-rpy` [Alias]

Aliases for `extrinsic-xyz` and `make-extrinsic-xyz`.

```
(let ((r (make-rpy 0.12 -0.23 0.34)))
  (let-values (((a b c) (tait-bryan-zyx r)))
    (list a b c))) ⇒ (0.3400000000000001 -0.2300000000000001 0.12000000000000002
```

12.4.2.2 Rotation Operations

`rotate-vector` *rq vq* [Procedure]

Applies the rotation represented by quaternion *rq* to the vector represented by vector quaternion *vq*, and returns the rotated vector. This is equivalent to `(* rq vq (conjugate rq))` for normalized *rq*.

```
(rotate-vector +k +2i)                      ⇒ -2i
(rotate-vector 1/2+1/2i+1/2j+1/2k +i+2j+3k) ⇒ +3.0i+1.0j+2.0k
```

`make-rotation-procedure` *rq* [Procedure]

A partial application of `rotate-vector`. Returns a single-argument procedure which will take a vector quaternion argument and rotate it by *rq*. The returned procedure closes over both *rq* and its conjugate, so this will likely be more efficient than `rotate-vector` at rotating many vectors by the same rotation.

12.5 Quantities and Units

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.

Quantity literals have the following syntax:

> *quantity* `::=` *optional-sign decimal unit-term* [`*` *unit-term*]... [`/` *unit-term*]
> *unit-term* `::=` *unit-name* [`^` *digit*+]
> *unit-name* `::=` *letter*+

Some examples are `10pt` (10 points), `5s` (5 seconds), and `4cm^2` (4 square centimeters).

Note the *quantity* syntax is not recognized by the reader. Instead these are read as symbols. Assuming there is no lexical binding the for the symbol, it will be rewritten at compile-time into an expression. For example `4cm^2` is transformed into:

```
(* 4.0 (expt unit:cm 2))
```

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

`quantity->number` *q* [Procedure]
: 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* [Procedure]
: Returns the unit of the quantity *q*. If *q* is a number, returns the empty unit.

`make-quantity` *x unit* [Procedure]
: 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)
```

The *unit-name* is declared in the `unit` namespace, so the above is equivalent to:

```
(define-constant unit:cent (* 0.01 unit:dollar))
```

Angles

The following angle units are dimensionless, with no base unit.

Some procedures treat a unit-less real number as if it were in radians (which mathematicians prefer); some procedures (such as `rotate`) treat a unit-less real number as if it were in degrees (which is common in Web and other standards).

`rad` [Unit]
: A unit for angles specified in radians. A full circle is 2*pi radians. Note that `(= 1.5 1.5rad)` is true, while `(eqv? 1.5 1.5rad)` is false.

`deg` [Unit]
: A unit for angles specified in degrees. A full circle is 360 degrees.

`grad` [Unit]
: A unit for angles specified in gradians. A full circle is 400 gradians.

12.6 Logical Number Operations

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

`bitwise-not` *i* [Procedure]
: Returns the bit-wise logical inverse of the argument. More formally, returns the exact integer whose two's complement representation is the one's complement of the two's complement representation of *i*.

`bitwise-and` *i ...* [Procedure]
`bitwise-ior` *i ...* [Procedure]
`bitwise-xor` *i ...* [Procedure]
: These procedures return the exact integer that is the bit-wise "and", "inclusive or", or "exclusive or" of the two's complement representations of their arguments. If they are passed only one argument, they return that argument. If they are passed no arguments, they return the integer that acts as identity for the operation: -1, 0, or 0, respectively.

`bitwise-if` *i1 i2 i3* [Procedure]
: Returns the exact integer that is the bit-wise "if" of the twos complement representations of its arguments, i.e. for each bit, if it is 1 in i1, the corresponding bit in i2 becomes the value of the corresponding bit in the result, and if it is 0, the corresponding bit in i3 becomes the corresponding bit in the value of the result. This is equivaent to the following computation:

```
(bitwise-ior (bitwise-and i1 i2)
             (bitwise-and (bitwise-not i1) i3))
```

`bitwise-bit-count` *i* [Procedure]
: If i is non-negative, returns the number of 1 bits in the twos complement representation of i. Otherwise it returns the result of the following computation:

```
(bitwise-not (bitwise-bit-count (bitwise-not i)))
```

`bitwise-length` *i* [Procedure]

Returns the number of bits needed to represent i if it is positive, and the number of bits needed to represent `(bitwise-not i)` if it is negative, which is the exact integer that is the result of the following computation:

```
(do ((result 0 (+ result 1))
     (bits (if (negative? i)
               (bitwise-not i)
               ei)
           (bitwise-arithmetic-shift bits -1)))
    ((zero? bits)
     result))
```

This is the number of bits needed to represent *i* in an unsigned field.

`bitwise-first-bit-set` *i* [Procedure]

Returns the index of the least significant 1 bit in the twos complement representation of i. If i is 0, then - 1 is returned.

```
(bitwise-first-bit-set 0) ⇒ -1
(bitwise-first-bit-set 1) ⇒ 0
(bitwise-first-bit-set -4) ⇒ 2
```

`bitwise-bit-set?` *i1 i2* [Procedure]

Returns `#t` if the i2'th bit (where *i2* must be non-negative) is 1 in the two's complement representation of *i1*, and `#f` otherwise. This is the result of the following computation:

```
(not (zero?
       (bitwise-and
         (bitwise-arithmetic-shift-left 1 i2)
         i1)))
```

`bitwise-copy-bit` *i bitno replacement-bit* [Procedure]

Returns the result of replacing the *bitno*'th bit of *i* by *replacement-bit*, where *bitno* must be non-negative, and *replacement-bit* must be either 0 or 1. This is the result of the following computation:

```
(let* ((mask (bitwise-arithmetic-shift-left 1 bitno)))
  (bitwise-if mask
           (bitwise-arithmetic-shift-left replacement-bit bitno)
           i))
```

`bitwise-bit-field` *n start end* [Procedure]

Returns the integer formed from the (unsigned) bit-field starting at *start* and ending just before *end*. Same as:

```
(let ((mask
       (bitwise-not
        (bitwise-arithmetic-shift-left -1 end))))
  (bitwise-arithmetic-shift-right
    (bitwise-and n mask)
    start))
```

`bitwise-copy-bit-field` *to start end from* [Procedure]

Returns the result of replacing in *to* the bits at positions from *start* (inclusive) to *end* (exclusive) by the bits in *from* from position 0 (inclusive) to position *end* - *start* (exclusive). Both *start* and *start* must be non-negative, and *start* must be less than or equal to *start*.

This is the result of the following computation:

```
(let* ((mask1
         (bitwise-arithmetic-shift-left -1 start))
       (mask2
         (bitwise-not
           (bitwise-arithmetic-shift-left -1 end)))
       (mask (bitwise-and mask1 mask2)))
  (bitwise-if mask
              (bitwise-arithmetic-shift-left from
                                             start)
              to))
```

`bitwise-arithmetic-shift` *i j* [Procedure]

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)))`.

Examples:

```
(bitwise-arithmetic-shift -6 -1) ⇒-3
(bitwise-arithmetic-shift -5 -1) ⇒ -3
(bitwise-arithmetic-shift -4 -1) ⇒ -2
(bitwise-arithmetic-shift -3 -1) ⇒ -2
(bitwise-arithmetic-shift -2 -1) ⇒ -1
(bitwise-arithmetic-shift -1 -1) ⇒ -1
```

`bitwise-arithmetic-shift-left` *i amount* [Procedure]
`bitwise-arithmetic-shift-right` *i amount* [Procedure]

The *amount* must be non-negative The `bitwise-arithmetic-shift-left` procedure returns the same result as `bitwise-arithmetic-shift`, and `(bitwise-arithmetic-shift-right i amount)` returns the same result as `(bitwise-arithmetic-shift i (- amount))`.

If *i* is a primitive integer type, then *amount* must be less than the number of bits in the promoted type of *i* (32 or 64). If the type is unsigned, an unsigned (logic) shift is done for `bitwise-arithmetic-shift-right`, rather than a signed (arithmetic) shift.

`bitwise-rotate-bit-field` *n start end count* [Procedure]

Returns the result of cyclically permuting in *n* the bits at positions from *start* (inclusive) to *end* (exclusive) by *count* bits towards the more significant bits, *start* and *end* must be non-negative, and *start* must be less than or equal to *end*. This is the result of the following computation:

```
(let* ((n     ei1)
       (width (- end start)))
  (if (positive? width)
      (let* ((count (mod count width))
             (field0
               (bitwise-bit-field n start end))
             (field1 (bitwise-arithmetic-shift-left
                       field0 count))
             (field2 (bitwise-arithmetic-shift-right
                       field0
                       (- width count)))
             (field (bitwise-ior field1 field2)))
        (bitwise-copy-bit-field n start end field))
      n))
```

`bitwise-reverse-bit-field` *i start end* [Procedure]

Returns the result obtained from *i* by reversing the order of the bits at positions from *start* (inclusive) to *end* (exclusive), where *start* and *end* must be non-negative, and *start* must be less than or equal to *end*.

```
(bitwise-reverse-bit-field #b1010010 1 4) ⇒  88 ; #b1011000
```

`logop` *op x y* [Procedure]

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

`logtest` *i j* [Procedure]

Returns true if the arguments have any bits in common. Same as `(not (zero? (bitwise-and i j)))`, but is more efficient.

12.6.1 SRFI-60 Logical Number Operations

Kawa supports SRFI-60 “Integers as Bits” as well, although we generally recommend using the R6RS-compatible functions instead when possible. Unless noted as being a builtin function, to use these you must first `(require 'srfi-60)` or `(import (srfi :60))` (or `(import (srfi :60 integer-bits))`).

`logand` *i ...* [Procedure]

Equivalent to `(bitwise-and i ...)`. Builtin.

`logior` *i ...* [Procedure]

Equivalent to `(bitwise-ior i ...)`. Builtin.

`logxor` *i ...* [Procedure]

Equivalent to `(bitwise-xor i ...)`. Builtin.

`lognot` *i* [Procedure]

Equivalent to `(bitwise-not i)`. Builtin.

`bitwise-merge` *mask i j* [Procedure]

Equivalent to `(bitwise-if mask i j)`.

`any-bits-set?` *i j* [Procedure]

Equivalent to `(logtest i j)`.

`logcount` *i* [Procedure]
`bit-count` *i* [Procedure]

Count the number of 1-bits in *i*, if it is non-negative. If *i* is negative, count number of 0-bits. Same as `(bitwise-bit-count i)` if *i* is non-negative. Builtin as `logcount`.

`integer-length` *i* [Procedure]

Equivalent to `(bitwise-length i)`. Builtin.

`log2-binary-factors` *i* [Procedure]
`first-set-bit` *i* [Procedure]

Equivalent to `(bitwise-first-bit-set i)`.

`logbit?` *pos i* [Procedure]
`bit-set?` *pos i* [Procedure]

Equivalent to `(bitwise-bit-set? i pos)`.

`copy-bit` *bitno i bool* [Procedure]

Equivalent to `(bitwise-copy-bit i bitno (if bool 1 0))`.

`bit-field` *n start end* [Procedure]

Equivalent to `(bitwise-bit-field n start end)`.

`copy-bit-field` *to from start end* [Procedure]

Equivalent to `(bitwise-copy-bit-field to start end from)`.

`arithmetic-shift` *i j* [Procedure]

Equivalent to `(bitwise-arithmetic-shift i j)`. Builtin.

`ash` *i j* [Procedure]

Alias for `arithmetic-shift`. Builtin.

`rotate-bit-field` *n count start end* [Procedure]

Equivalent to `(bitwise-rotate-bit-field n start end count)`.

`reverse-bit-field` *i start end* [Procedure]

Equivalent to `(bitwise-reverse-bit-field i start end)`.

`integer->list` *k* [*length*] [Procedure]
`list->integer` *list* [Procedure]

The `integer->list` procedure returns a list of *length* booleans corresponding to the bits of the non-negative integer *k*, with `#t` for `1` and `#f` for `0`. *length* defaults to `(bitwise-length k)`. The list will be in order from MSB to LSB, with the value of `(odd? k)` in the last car.

The `list->integer` procedure returns the integer corresponding to the booleans in the list *list*. The `integer->list` and `list->integer` procedures are inverses so far as `equal?` is concerned.

`booleans->integer` *bool1 ...* [Procedure]

Returns the integer coded by the *bool1 ...* arguments. Equivalent to `(list->integer (list bool1 ...))`.

12.6.2 Deprecated Logical Number Operations

This older function is still available, but we recommend using the R6RS-compatible function.

`bit-extract` *n start end* [Procedure]

Equivalent to `(bitwise-bit-field` *`n start end`*`)`.

12.7 Performance of numeric operations

Kawa can generally do a pretty good job of generating efficient code for numeric operations, at least when it knows or can figure out the types of the operands.

The basic operations `+`, `-`, and `*` are compiled to single-instruction bytecode if both operands are `int` or `long`. Likewise, if both operands are floating-point (or one is floating-point and the other is rational), then single-instruction `double` or `float` instructions are emitted.

A binary operation involving an infinite-precision `integer` and a fixed-size `int` or `long` is normally evaluated by expanding the latter to `integer` and using `integer` arithmetic. An exception is an integer literal whose value fits in an `int` or `long` - in that case the operation is done using `int` or `long` arithmetic.

In general, integer literals have amorphous type. When used to infer the type of a variable, they have `integer` type:

```
(let ((v1 0))
  ... v1 has type integer ... )
```

However, a literal whose value fits in the `int` or `long` range is implicitly viewed `int` or `long` in certain contexts, primarily method overload resolution and binary arithmetic (as mentioned above).

The comparison functions `<`, `<=`, `=`, `>`, and `=>` are also optimized to single instriction operations if the operands have appropriate type. However, the functions `zero?`, `positive?`, and `negative?` have not yet been optimized. Instead of `(positive? x)` write `(> x 0)`.

There are a number of integer division and modulo operations. If the operands are `int` or `long`, it is faster to use `quotient` and `remainder` rather than `div` and `mod` (or `modulo`). If you know the first operand is non-negative and the second is positive, then use `quotient` and `remainder`. (If an operand is an arbitrary-precision `integer`, then it dosn't really matter.)

The logical operations `bitwise-and`, `bitwise-ior`, `bitwise-xor`, `bitwise-not`, `bitwise-arithmetic-shift-left`, `bitwise-arithmetic-shift-right` are compiled to single bitcode instructions if the operands are `int` or `long`. Avoid `bitwise-arithmetic-shift` if the sign of the shift is known. If the operands are arbitrary-precision `integer`, a library call is needed, but run-time type dispatch is avoided.

13 Characters and text

13.1 Characters

Characters are objects that represent human-readable characters such as letters and digits. More precisely, a character represents a Unicode scalar value (`http://www.unicode.org/glossary/#unicode_scalar_value`). Each character has an integer value in the range `0` to `#x10FFFF` (excluding the range `#xD800` to `#xDFFF` used for Surrogate Code Points (`http://www.unicode.org/glossary/#surrogate_code_point`)).

> *Note:* Unicode distinguishes between glyphs, which are printed for humans to read, and characters, which are abstract entities that map to glyphs (sometimes in a way that's sensitive to surrounding characters). Furthermore, different sequences of scalar values sometimes correspond to the same character. The relationships among scalar, characters, and glyphs are subtle and complex.
>
> Despite this complexity, most things that a literate human would call a "character" can be represented by a single Unicode scalar value (although several sequences of Unicode scalar values may represent that same character). For example, Roman letters, Cyrillic letters, Hebrew consonants, and most Chinese characters fall into this category.
>
> Unicode scalar values exclude the range `#xD800` to `#xDFFF`, which are part of the range of Unicode *code points*. However, the Unicode code points in this range, the so-called *surrogates*, are an artifact of the UTF-16 encoding, and can only appear in specific Unicode encodings, and even then only in pairs that encode scalar values. Consequently, all characters represent code points, but the surrogate code points do not have representations as characters.

`character` [Type]

A Unicode code point - normally a Unicode scalar value, but could be a surrogate. This is implemented using a 32-bit `int`. When an object is needed (i.e. the *boxed* representation), it is implemented an instance of `gnu.text.Char`.

`character-or-eof` [Type]

A `character` or the specical `#!eof` value (used to indicate end-of-file when reading from a port). This is implemented using a 32-bit `int`, where the value -1 indicates end-of-file. When an object is needed, it is implemented an instance of `gnu.text.Char` or the special `#!eof` object.

`char` [Type]

A UTF-16 code unit. Same as Java primitive `char` type. Considered to be a sub-type of `character`. When an object is needed, it is implemented as an instance of `java.lang.Character`. Note the unfortunate inconsistency (for historical reasons) of `char` boxed as `Character` vs `character` boxed as `Char`.

Characters are written using the notation `#\`*character* (which stands for the given *character*; `#\x`*hex-scalar-value* (the character whose scalar value is the given hex integer); or `#\`*character-name* (a character with a given name):

character `::=` #“*any-character*
 `|` #“ *character-name*
 `|` #“**x** *hex-scalar-value*
 `|` #“**X** *hex-scalar-value*

The following *character-name* forms are recognized:

#“alarm `#\x0007` - the alarm (bell) character

#“backspace
`#\x0008`

#“delete

#“del

#“rubout `#\x007f` - the delete or rubout character

#“escape

#“esc `#\x001b`

#“newline
#“linefeed
`#\x001a` - the linefeed character

#“null
#“nul `#\x0000` - the null character

#“page `#\000c` - the formfeed character

#“return `#\000d` - the carriage return character

#“space `#\x0020` - the preferred way to write a space

#“tab `#\x0009` - the tab character

#“vtab `#\x000b` - the vertical tabulation character

#“ignorable-char
A special `character` value, but it is not a Unicode code point. It is a special value returned when an index refers to the second `char` (code point) of a surrogate pair, and which should be ignored. (When writing a `character` to a string or file, it will be written as one or two `char` values. The exception is `#\ignorable-char`, for which zero `char` values are written.)

`char?` *`obj`* [Procedure]

Return `#t` if *obj* is a character, `#f` otherwise. (The *obj* can be any character, not just a 16-bit `char`.)

`char->integer` *`char`* [Procedure]
`integer->char` *`sv`* [Procedure]

sv should be a Unicode scalar value, i.e., a non–negative exact integer object in `[0, #xD7FF] union [#xE000, #x10FFFF]`. (Kawa also allows values in the surrogate range.)

Given a character, `char->integer` returns its Unicode scalar value as an exact integer object. For a Unicode scalar value *sv*, `integer->char` returns its associated character.

```
(integer->char 32)                    ⇒ #\space
(char->integer (integer->char 5000))  ⇒ 5000
(integer->char #\xD800)               ⇒ throws ClassCastException
```

Performance note: A call to `char->integer` is compiled as casting the argument to a `character`, and then re-interpreting that value as an `int`. A call to `integer->char` is compiled as casting the argument to an `int`, and then re-interpreting that value as an `character`. If the argument is the right type, no code is emitted: the value is just re-interpreted as the result type.

`char=?` ***char₁ char₂ char₃ ...*** [Procedure]
`char<?` ***char₁ char₂ char₃ ...*** [Procedure]
`char>?` ***char₁ char₂ char₃ ...*** [Procedure]
`char<=?` ***char₁ char₂ char₃ ...*** [Procedure]
`char>=?` ***char₁ char₂ char₃ ...*** [Procedure]

These procedures impose a total ordering on the set of characters according to their Unicode scalar values.

```
(char<? #\z #\ß)      ⇒ #t
(char<? #\z #\Z)      ⇒ #f
```

Performance note: This is compiled as if converting each argument using `char->integer` (which requires no code) and the using the corresponing `int` comparison.

`digit-value` *char* [Procedure]

This procedure returns the numeric value (0 to 9) of its argument if it is a numeric digit (that is, if `char-numeric?` returns `#t`), or `#f` on any other character.

```
(digit-value #\3)        ⇒ 3
(digit-value #\x0664)    ⇒ 4
(digit-value #\x0AE6)    ⇒ 0
(digit-value #\x0EA6)    ⇒ #f
```

13.2 Character sets

Sets of characters are useful for text-processing code, including parsing, lexing, and pattern-matching. SRFI 14 (`http://srfi.schemers.org/srfi-14/srfi-14.html`) specifies a `char-set` type for such uses. Some examples:

```
(import (srfi :14 char-sets))
(define vowel (char-set #\a #\e #\i #\o #\u))
(define vowely (char-set-adjoin vowel #\y))
(char-set-contains? vowel #\y) ⇒  #f
(char-set-contains? vowely #\y) ⇒  #t
```

See the SRFI 14 specification (`http://srfi.schemers.org/srfi-14/srfi-14.html`) for details.

`char-set` [Type]

The type of character sets. In Kawa `char-set` is a type that can be used in type specifiers:

```
(define vowely ::char-set (char-set-adjoin vowel #\y))
```

Kawa uses inversion lists (`https://en.wikipedia.org/wiki/Inversion_list`) for an efficient implementation, using Java `int` arrays to represents character ranges (inversions).

The `char-set-contains?` function uses binary search, so it takes time proportional to the logarithm of the number of inversions. Other operations may take time proportional to the number of inversions.

13.3 Strings

Strings are sequences of characters. The *length* of a string is the number of characters that it contains, as an exact non-negative integer. The *valid indices* of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

Strings are *implemented* as a sequence of 16-bit `char` values, even though they're semantically a sequence of 32-bit Unicode code points. A character whose value is greater than `#xffff` is represented using two surrogate characters. The implementation allows for natural interoperability with Java APIs. However it does make certain operations (indexing or counting based on character counts) difficult to implement efficiently. Luckily one rarely needs to index or count based on character counts; alternatives are discussed below.

There are different kinds of strings:

- An *istring* is *immutable*: It is fixed, and cannot be modified. On the other hand, indexing (e.g. `string-ref`) is efficient (constant-time), while indexing of other string implementations takes time proportional to the index.

  String literals are istrings, as are the return values of most of the procedures in this chapter.

  An *istring* is an instance of the `gnu.lists.IString` class.
- An *mstring* is *mutable*: You can replace individual characters (using `string-set!`). You can also change the *mstring*'s length by inserting or removing characters (using `string-append!` or `string-replace!`).

  An *mstring* is an instance of the `gnu.lists.FString` class.
- Any other object that implements the `java.lang.CharSequence` interface is also a string. This includes standard Java `java.lang.String` and `java.lang.StringBuilder` objects.

Some of the procedures that operate on strings ignore the difference between upper and lower case. The names of the versions that ignore case end with "`-ci`" (for "case insensitive").

Compatibility: Many of the following procedures (for example `string-append`) return an immutable istring in Kawa, but return a "freshly allocated" mutable string in standard Scheme (include R7RS) as well as most Scheme implementations (including previous versions of Kawa). To get the "compatibility mode" versions of those procedures (which return mstrings), invoke Kawa with one the `--r5rs`, `--r6rs`, or `--r7rs` options, or you can `import` a standard library like `(scheme base)`.

`string` [Type]

> The type of string objects. The underlying type is the interface `java.lang.CharSequence`. Immultable strings are `gnu.lists.IString` or `java.lang.String`, while mutable strings are `gnu.lists.FString`.

13.3.1 Basic string procedures

`string?` *obj* [Procedure]
: Return `#t` if *obj* is a string, `#f` otherwise.

`istring?` *obj* [Procedure]
: Return `#t` if *obj* is a istring (a immutable, constant-time-indexable string); `#f` otherwise.

`string` *char* ... [Constructor]
: Return a string composed of the arguments. This is analogous to *list*.

Compatibility: The result is an istring, except in compatibility mode, when it is a new allocated mstring.

`string-length` *string* [Procedure]
: Return the number of characters in the given *string* as an exact integer object.

Performance note: If the *string* is not an istring, the calling `string-length` may take time proportional to the length of the *string*, because of the need to scan for surrogate pairs.

`string-ref` *string k* [Procedure]
: *k* must be a valid index of *string*. The `string-ref` procedure returns character *k* of *string* using zero–origin indexing.

Performance note: If the *string* is not an istring, then calling `string-ref` may take time proportional to *k* because of the need to check for surrogate pairs. An alternative is to use `string-cursor-ref`. If iterating through a string, use `string-for-each`.

`string-null?` *string* [Procedure]
: Is *string* the empty string? Same result as `(= (string-length` *`string`*`) 0)` but executes in O(1) time.

`string-every` *pred string* [*start end*]) [Procedure]
`string-any` *pred string* [*start end*]) [Procedure]
: Checks to see if every/any character in *string* satisfies *pred*, proceeding from left (index *start*) to right (index *end*). These procedures are short-circuiting: if *pred* returns false, `string-every` does not call *pred* on subsequent characters; if *pred* returns true, `string-any` does not call *pred* on subsequent characters. Both procedures are "witness-generating":

- If `string-every` is given an empty interval (with *start* = *end*), it returns `#t`.
- If `string-every` returns true for a non-empty interval (with *start* `<` *end*), the returned true value is the one returned by the final call to the predicate on `(string-ref` *`string`* `(-` *`end`* `1))`.
- If `string-any` returns true, the returned true value is the one returned by the predicate.

Note: The names of these procedures do not end with a question mark. This indicates a general value is returned instead of a simple boolean (`#t` or `#f`).

13.3.2 Immutable String Constructors

`string-tabulate` *proc len* [Procedure]

Constructs a string of size *len* by calling *proc* on each value from 0 (inclusive) to *len* (exclusive) to produce the corresponding element of the string. The procedure *proc* accepts an exact integer as its argument and returns a character. The order in which *proc* is called on those indexes is not specifified.

Rationale: Although `string-unfold` is more general, `string-tabulate` is likely to run faster for the common special case it implements.

`string-unfold` *stop? mapper successor seed* [*base make-final*] [Procedure]
`string-unfold-right` *stop? mapper successor seed* [*base make-final*] [Procedure]

This is a fundamental and powerful constructor for strings.

- *successor* is used to generate a series of "seed" values from the initial seed: *seed*, (*successor seed*), (*successor*^{2} *seed*), (*successor*^{3} *seed*), ...
- *stop?* tells us when to stop — when it returns true when applied to one of these seed values.
- *mapper* maps each seed value to the corresponding character(s) in the result string, which are assembled into that string in left-to-right order. It is an error for *mapper* to return anything other than a character or string.
- *base* is the optional initial/leftmost portion of the constructed string, which defaults to the empty string `""`. It is an error if *base* is anything other than a character or string.
- *make-final* is applied to the terminal seed value (on which *stop?* returns true) to produce the final/rightmost portion of the constructed string. It defaults to `(lambda (x) "")`. It is an error for *make-final* to return anything other than a character or string.

`string-unfold-right` is the same as `string-unfold` except the results of *mapper* are assembled into the string in right-to-left order, *base* is the optional rightmost portion of the constructed string, and *make-final* produces the leftmost portion of the constructed string.

You can use it `string-unfold` to convert a list to a string, read a port into a string, reverse a string, copy a string, and so forth. Examples:

```
(define (port->string p)
  (string-unfold eof-object? values
                 (lambda (x) (read-char p))
                 (read-char p)))

(define (list->string lis)
  (string-unfold null? car cdr lis))

(define (string-tabulate f size)
  (string-unfold (lambda (i) (= i size)) f add1 0))
```

To map *f* over a list *lis*, producing a string:

```
(string-unfold null? (compose f car) cdr lis)
```

Interested functional programmers may enjoy noting that `string-fold-right` and `string-unfold` are in some sense inverses. That is, given operations *knull?*, *kar*, *kdr*, *kons*, and *knil* satisfying

```
(kons (kar x) (kdr x)) = x  and  (knull? knil) = #t
```

then

```
(string-fold-right kons knil (string-unfold knull? kar kdr x)) = x
```

and

```
(string-unfold knull? kar kdr (string-fold-right kons knil string)) = string.
```

This combinator pattern is sometimes called an “anamorphism.”

13.3.3 Selection

`substring` *string start end* [Procedure]

string must be a string, and *start* and *end* must be exact integer objects satisfying:

```
0 <= start <= end <= (string-length string)
```

The `substring` procedure returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with index *end* (exclusive).

`string-take` *string nchars* [Procedure]
`string-drop` *string nchars* [Procedure]
`string-take-right` *string nchars* [Procedure]
`string-drop-right` *string nchars* [Procedure]

`string-take` returns an immutable string containing the first *nchars* of *string*; `string-drop` returns a string containing all but the first *nchars* of *string*. `string-take-right` returns a string containing the last *nchars* of *string*; `string-drop-right` returns a string containing all but the last *nchars* of *string*.

```
(string-take "Pete Szilagyi" 6) ⇒ "Pete S"
(string-drop "Pete Szilagyi" 6) ⇒ "zilagyi"

(string-take-right "Beta rules" 5) ⇒ "rules"
(string-drop-right "Beta rules" 5) ⇒ "Beta "
```

It is an error to take or drop more characters than are in the string:

```
(string-take "foo" 37) ⇒ error
```

`string-pad` *string len* [*char start end*] [Procedure]
`string-pad-right` *string len* [*char start end*] [Procedure]

Returns an istring of length *len* comprised of the characters drawn from the given subrange of *string*, padded on the left (right) by as many occurrences of the character *char* as needed. If *string* has more than *len* chars, it is truncated on the left (right) to length *len*. The *char* defaults to `#\space`

```
(string-pad     "325" 5) ⇒ "  325"
(string-pad   "71325" 5) ⇒ "71325"
(string-pad "8871325" 5) ⇒ "71325"
```

`string-trim` *string* [*pred start end*] [Procedure]
`string-trim-right` *string* [*pred start end*] [Procedure]
`string-trim-both` *string* [*pred start end*] [Procedure]

Returns an istring obtained from the given subrange of *string* by skipping over all characters on the left / on the right / on both sides that satisfy the second argument *pred*: *pred* defaults to `char-whitespace?`.

```
(string-trim-both "  The outlook wasn't brilliant,  \n\r")
    ⇒ "The outlook wasn't brilliant,"
```

13.3.4 String Comparisons

`string=?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]

Return `#t` if the strings are the same length and contain the same characters in the same positions. Otherwise, the `string=?` procedure returns `#f`.

```
(string=? "Straße" "Strasse")    ⇒ #f
```

`string<?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]
`string>?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]
`string<=?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]
`string>=?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]

These procedures return `#t` if their arguments are (respectively): monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically nonincreasing. These predicates are required to be transitive.

These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, `string<?` is the lexicographic ordering on strings induced by the ordering `char<?` on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

```
(string<? "z" "ß")      ⇒ #t
(string<? "z" "zz")     ⇒ #t
(string<? "z" "Z")      ⇒ #f
```

`string-ci=?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]
`string-ci<?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]
`string-ci>?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]
`string-ci<=?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]
`string-ci>=?` *string*$_1$ *string*$_2$ *string*$_3$ ... [Procedure]

These procedures are similar to `string=?`, etc., but behave as if they applied `string-foldcase` to their arguments before invoking the corresponding procedures without `-ci`.

```
(string-ci<? "z" "Z")                 ⇒ #f
(string-ci=? "z" "Z")                 ⇒ #t
(string-ci=? "Straße" "Strasse")      ⇒ #t
(string-ci=? "Straße" "STRASSE")      ⇒ #t
(string-ci=? "ΧΑΟΣ" "χαοσ")              ⇒ #t
```

13.3.5 Conversions

`list->string` *list* [Procedure]

The `list->string` procedure returns an istring formed from the characters in *list*, in order. It is an error if any element of *list* is not a character.

Compatibility: The result is an istring, except in compatibility mode, when it is an mstring.

`reverse-list->string` *list* [Procedure]

An efficient implementation of `(compose list->text reverse)`:

```
(reverse-list->text '(#\a #\B #\c))  ⇒ "cBa"
```

This is a common idiom in the epilogue of string-processing loops that accumulate their result using a list in reverse order. (See also `string-concatenate-reverse` for the "chunked" variant.)

`string->list` *string* [*start* [*end*]] [Procedure]

The `string->list` procedure returns a newly allocated list of the characters of *string* between *start* and *end*, in order. The `string->list` and `list->string` procedures are inverses so far as `equal?` is concerned.

`vector->string` *vector* [*start* [*end*]] [Procedure]

The `vector->string` procedure returns a newly allocated string of the objects contained in the elements of *vector* between *start* and *end*. It is an error if any element of *vector* between *start* and *end* is not a character, or is a character forbidden in strings.

```
(vector->string #(#\1 #\2 #\3))             ⇒ "123"
(vector->string #(#\1 #\2 #\3 #\4 #\5) 2 4) ⇒ "34"
```

`string->vector` *string* [*start* [*end*]] [Procedure]

The `string->vector` procedure returns a newly created vector initialized to the elements of the string *string* between *start* and *end*.

```
(string->vector "ABC")         ⇒ #(#\A #\B #\C)
(string->vector "ABCDE" 1 3)   ⇒ #(#\B #\C)
```

`string-upcase` *string* [Procedure]
`string-downcase` *string* [Procedure]
`string-titlecase` *string* [Procedure]
`string-foldcase` *string* [Procedure]

These procedures take a string argument and return a string result. They are defined in terms of Unicode's locale–independent case mappings from Unicode scalar–value sequences to scalar–value sequences. In particular, the length of the result string can be different from the length of the input string. When the specified result is equal in the sense of `string=?` to the argument, these procedures may return the argument instead of a newly allocated string.

The `string-upcase` procedure converts a string to upper case; `string-downcase` converts a string to lower case. The `string-foldcase` procedure converts the string to its case–folded counterpart, using the full case–folding mapping, but without the special mappings for Turkic languages. The `string-titlecase` procedure converts the first cased character of each word, and downcases all other cased characters.

```
(string-upcase "Hi")              ⇒ "HI"
(string-downcase "Hi")            ⇒ "hi"
(string-foldcase "Hi")            ⇒ "hi"

(string-upcase "Straße")          ⇒ "STRASSE"
(string-downcase "Straße")        ⇒ "straße"
(string-foldcase "Straße")        ⇒ "strasse"
(string-downcase "STRASSE")       ⇒ "strasse"

(string-downcase "Σ")             ⇒ "σ"
; Chi Alpha Omicron Sigma:
(string-upcase "ΧΑΟΣ")            ⇒ "ΧΑΟΣ"
(string-downcase "ΧΑΟΣ")          ⇒ "χαος"
(string-downcase "ΧΑΟΣΣ")         ⇒ "χαοσς"
(string-downcase "ΧΑΟΣ Σ")        ⇒ "χαος σ"
(string-foldcase "ΧΑΟΣΣ")         ⇒ "χαοσσ"
(string-upcase "χαος")            ⇒ "ΧΑΟΣ"
(string-upcase "χαοσ")            ⇒ "ΧΑΟΣ"

(string-titlecase "kNock KNoCK")  ⇒ "Knock Knock"
(string-titlecase "who's there?") ⇒ "Who's There?"
(string-titlecase "r6rs")         ⇒ "R6rs"
(string-titlecase "R6RS")         ⇒ "R6rs"
```

Since these procedures are locale–independent, they may not be appropriate for some locales.

Kawa Note: The implementation of `string-titlecase` does not correctly handle the case where an initial character needs to be converted to multiple characters, such as “LATIN SMALL LIGATURE FL” which should be converted to the two letters `"Fl"`.

Compatibility: The result is an istring, except in compatibility mode, when it is an mstring.

`string-normalize-nfd` ***`string`*** [Procedure]
`string-normalize-nfkd` ***`string`*** [Procedure]
`string-normalize-nfc` ***`string`*** [Procedure]
`string-normalize-nfkc` ***`string`*** [Procedure]

These procedures take a string argument and return a string result, which is the input string normalized to Unicode normalization form D, KD, C, or KC, respectively. When the specified result is equal in the sense of `string=?` to the argument, these procedures may return the argument instead of a newly allocated string.

```
(string-normalize-nfd "\xE9;")          ⇒ "\x65;\x301;"
(string-normalize-nfc "\xE9;")          ⇒ "\xE9;"
(string-normalize-nfd "\x65;\x301;")    ⇒ "\x65;\x301;"
(string-normalize-nfc "\x65;\x301;")    ⇒ "\xE9;"
```

13.3.6 Searching and matching

`string-prefix-length` *string*$_1$ *string*$_2$ [*start*$_1$ *end*$_1$ *start*$_2$ *end*$_2$] [Procedure]
`string-suffix-length` *string*$_1$ *string*$_2$ [*start*$_1$ *end*$_1$ *start*$_2$ *end*$_2$] [Procedure]

Return the length of the longest common prefix/suffix of *string*$_1$ and *string*$_2$. For prefixes, this is equivalent to their "mismatch index" (relative to the start indexes).

The optional *start*/*end* indexes restrict the comparison to the indicated substrings of *string*$_1$ and *string*$_2$.

`string-prefix?` *string*$_1$ *string*$_2$ [*start*$_1$ *end*$_1$ *start*$_2$ *end*$_2$] [Procedure]
`string-suffix?` *string*$_1$ *string*$_2$ [*start*$_1$ *end*$_1$ *start*$_2$ *end*$_2$] [Procedure]

Is *string*$_1$ a prefix/suffix of *string*$_2$?

The optional *start*/*end* indexes restrict the comparison to the indicated substrings of *string*$_1$ and *string*$_2$.

`string-index` *string pred* [*start end*] [Procedure]
`string-index-right` *string pred* [*start end*] [Procedure]
`string-skip` *string pred* [*start end*] [Procedure]
`string-skip-right` *string pred* [*start end*] [Procedure]

`string-index` searches through the given substring from the left, returning the index of the leftmost character satisfying the predicate *pred*. `string-index-right` searches from the right, returning the index of the rightmost character satisfying the predicate *pred*. If no match is found, these procedures return `#f`.

The *start* and *end* arguments specify the beginning and end of the search; the valid indexes relevant to the search include *start* but exclude *end*. Beware of "fencepost"" errors: when searching right-to-left, the first index considered is `(- end 1)`, whereas when searching left-to-right, the first index considered is *start*. That is, the start/end indexes describe the same half-open interval `[start,end)` in these procedures that they do in other string procedures.

The `-skip` functions are similar, but use the complement of the criterion: they search for the first char that *doesn't* satisfy *pred*. To skip over initial whitespace, for example, say

```
(substring string
           (or (string-skip string char-whitespace?)
               (string-length string))
           (string-length string))
```

These functions can be trivially composed with `string-take` and `string-drop` to produce `take-while`, `drop-while`, `span`, and `break` procedures without loss of efficiency.

`string-contains` *string*$_1$ *string*$_2$ [*start*$_1$ *end*$_1$ *start*$_2$ *end*$_2$] [Procedure]
`string-contains-right` *string*$_1$ *string*$_2$ [*start*$_1$ *end*$_1$ *start*$_2$ *end*$_2$] [Procedure]

Does the substring of *string*$_1$ specified by *start*$_1$ and *end*$_1$ contain the sequence of characters given by the substring of *string*$_2$ specified by *start*$_2$ and *end*$_2$?

Returns `#f` if there is no match. If *start*$_2$ = *end*$_2$, `string-contains` returns *start*$_1$ but `string-contains-right` returns *end*$_1$. Otherwise returns the index in *string*$_1$ for the first character of the first/last match; that index lies within the half-open interval [$start_1$,end_1), and the match lies entirely within the [$start_1$,end_1) range of $string_1$.

```
(string-contains "eek -- what a geek." "ee" 12 18) ; Searches "a geek"
    ⇒ 15
```

Note: The names of these procedures do not end with a question mark. This indicates a useful value is returned when there is a match.

13.3.7 Concatenation and replacing

`string-append` *string* . . . [Procedure]

Returns a string whose characters form the concatenation of the given strings.

Compatibility: The result is an istring, except in compatibility mode, when it is an mstring.

`string-concatenate` *string-list* [Procedure]

Concatenates the elements of *string-list* together into a single istring.

Rationale: Some implementations of Scheme limit the number of arguments that may be passed to an n-ary procedure, so the `(apply string-append` ***`string-list`***`)` idiom, which is otherwise equivalent to using this procedure, is not as portable.

`string-concatenate-reverse` *string-list* [*final-string* [*end*]]) [Procedure]

With no optional arguments, calling this procedure is equivalent to `(string-concatenate (reverse` ***`string-list`***`))`. If the optional argument *final-string* is specified, it is effectively consed onto the beginning of *string-list* before performing the list-reverse and string-concatenate operations.

If the optional argument *end* is given, only the characters up to but not including *end* in *final-string* are added to the result, thus producing

```
(string-concatenate
  (reverse (cons (substring final-string 0 end)
                 string-list)))
```

For example:

```
(string-concatenate-reverse '(" must be" "Hello, I") " going.XXXX" 7)
  ⇒ "Hello, I must be going."
```

Rationale: This procedure is useful when constructing procedures that accumulate character data into lists of string buffers, and wish to convert the accumulated data into a single string when done. The optional end argument accommodates that use case when *final-string* is a bob-full mutable string, and is allowed (for uniformity) when *final-string* is an immutable string.

`string-join` *string-list* [*delimiter* [*grammar*]] [Procedure]

This procedure is a simple unparser; it pastes strings together using the *delimiter* string, returning an istring.

The *string-list* is a list of strings. The *delimiter* is the string used to delimit elements; it defaults to a single space `" "`.

The *grammar* argument is a symbol that determines how the *delimiter* is used, and defaults to `'infix`. It is an error for *grammar* to be any symbol other than these four:

`'infix` An infix or separator grammar: insert the delimiter between list elements. An empty list will produce an empty string.

`'strict-infix`
Means the same as `'infix` if the string-list is non-empty, but will signal an error if given an empty list. (This avoids an ambiguity shown in the examples below.)

`'suffix` Means a suffix or terminator grammar: insert the *delimiter* after every list element.

`'prefix` Means a prefix grammar: insert the *delimiter* before every list element.

```
(string-join '("foo" "bar" "baz"))
        ⇒ "foo bar baz"
(string-join '("foo" "bar" "baz") "")
        ⇒ "foobarbaz"
(string-join '("foo" "bar" "baz") ":")
        ⇒ "foo:bar:baz"
(string-join '("foo" "bar" "baz") ":" 'suffix)
        ⇒ "foo:bar:baz:"

;; Infix grammar is ambiguous wrt empty list vs. empty string:
(string-join '()   ":") ⇒ ""
(string-join '("") ":") ⇒ ""

;; Suffix and prefix grammars are not:
(string-join '()   ":" 'suffix)) ⇒ ""
(string-join '("") ":" 'suffix)) ⇒ ":"
```

string-replace *string$_1$ string$_2$ start$_1$ end$_1$* [*start$_2$ end$_2$*] [Procedure]

Returns

```
(string-append (substring string1 0 start1)
               (substring string2 start2 end2)
               (substring string1 end1 (string-length string1)))
```

That is, the segment of characters in *string$_1$* from *start$_1$* to *end$_1$* is replaced by the segment of characters in *string$_2$* from *start$_2$* to *end$_2$*. If *start$_1$*=*end$_1$*, this simply splices the characters drawn from *string$_2$* into *string$_1$* at that position.

Examples:

```
(string-replace "The TCL programmer endured daily ridicule."
                "another miserable perl drone" 4 7 8 22)
    ⇒ "The miserable perl programmer endured daily ridicule."

(string-replace "It's easy to code it up in Scheme." "lots of fun" 5 9)
    ⇒ "It's lots of fun to code it up in Scheme."
```

```
(define (string-insert s i t) (string-replace s t i i))

(string-insert "It's easy to code it up in Scheme." 5 "really ")
    ⇒ "It's really easy to code it up in Scheme."

(define (string-set s i c) (string-replace s (string c) i (+ i 1)))

(string-set "String-ref runs in O(n) time." 19 #\1)
    ⇒ "String-ref runs in O(1) time."
```

Also see `string-append!` and `string-replace!` for destructive changes to a mutable string.

13.3.8 Mapping and folding

`string-fold` *kons knil string* [*start end*] [Procedure]
`string-fold-right` *kons knil string* [*start end*] [Procedure]

These are the fundamental iterators for strings.

The `string-fold` procedure maps the *kons* procedure across the given *string* from left to right:

```
(... (kons string_2 (kons string_1 (kons string_0 knil))))
```

In other words, string-fold obeys the (tail) recursion

```
  (string-fold kons knil string start end)
= (string-fold kons (kons string_start knil) start+1 end)
```

The `string-fold-right` procedure maps *kons* across the given string *string* from right to left:

```
(kons string_0
      (... (kons string_end-3
                 (kons string_end-2
                       (kons string_end-1
                             knil)))))
```

obeying the (tail) recursion

```
  (string-fold-right kons knil string start end)
= (string-fold-right kons (kons string_end-1 knil) start end-1)
```

Examples:

```
;;; Convert a string or string to a list of chars.
(string-fold-right cons '() string)

;;; Count the number of lower-case characters in a string or string.
(string-fold (lambda (c count)
               (if (char-lower-case? c)
                   (+ count 1)
                   count))
             0
```

```
                    string)
```

The string-fold-right combinator is sometimes called a "catamorphism."

string-for-each *proc* *string*$_1$ *string*$_2$ ... [Procedure]
string-for-each *proc* *string*$_1$ [*start* [*end*]] [Procedure]

The *strings* must all have the same length. *proc* should accept as many arguments as there are *strings*.

The *start-end* variant is provided for compatibility with the SRFI-13 version. (In that case *start* and *end* count code Unicode scalar values (`character` values), not Java 16-bit `char` values.)

The `string-for-each` procedure applies *proc* element–wise to the characters of the *strings* for its side effects, in order from the first characters to the last. *proc* is always called in the same dynamic environment as `string-for-each` itself.

Analogous to `for-each`.

```
(let ((v '()))
  (string-for-each
    (lambda (c) (set! v (cons (char->integer c) v)))
    "abcde")
   v)
  ⇒ (101 100 99 98 97)
```

Performance note: The compiler generates efficient code for `string-for-each`. If *proc* is a lambda expression, it is inlined.

string-map *proc* *string*$_1$ *string*$_2$ ... [Procedure]

The `string-map` procedure applies *proc* element-wise to the elements of the strings and returns a string of the results, in order. It is an error if *proc* does not accept as many arguments as there are strings, or return other than a single character or a string. If more than one string is given and not all strings have the same length, `string-map` terminates when the shortest string runs out. The dynamic order in which *proc* is applied to the elements of the strings is unspecified.

```
(string-map char-foldcase "AbdEgH")  ⇒ "abdegh"
(string-map
  (lambda (c) (integer->char (+ 1 (char->integer c))))
  "HAL")
        ⇒ "IBM"
(string-map
  (lambda (c k)
    ((if (eqv? k #\u) char-upcase char-downcase) c))
  "studlycaps xxx"
  "ululululul")
        ⇒ "StUdLyCaPs"
```

Traditionally the result of *proc* had to be a character, but Kawa (and SRFI-140) allows the result to be a string.

Performance note: The `string-map` procedure has not been optimized (mainly because it is not very useful): The characters are boxed, and the *proc* is not inlined even if it is a lambda expression.

`string-map-index` *proc string* [*start end*] [Procedure]

Calls *proc* on each valid index of the specified substring, converts the results of those calls into strings, and returns the concatenation of those strings. It is an error for *proc* to return anything other than a character or string. The dynamic order in which proc is called on the indexes is unspecified, as is the dynamic order in which the coercions are performed. If any strings returned by *proc* are mutated after they have been returned and before the call to `string-map-index` has returned, then `string-map-index` returns a string with unspecified contents; the *string-map-index* procedure itself does not mutate those strings.

`string-for-each-index` *proc string* [*start end*] [Procedure]

Calls *proc* on each valid index of the specified substring, in increasing order, discarding the results of those calls. This is simply a safe and correct way to loop over a substring.

Example:

```
(let ((txt (string->string "abcde"))
      (v '()))
  (string-for-each-index
    (lambda (cur) (set! v (cons (char->integer (string-ref txt cur)) v)))
    txt)
  v) ⇒ (101 100 99 98 97)
```

`string-count` *string pred* [*start end*] [Procedure]

Returns a count of the number of characters in the specified substring of *string* that satisfy the predicate *pred*.

`string-filter` *pred string* [*start end*] [Procedure]
`string-remove` *pred string* [*start end*] [Procedure]

Return an immutable string consisting of only selected characters, in order: `string-filter` selects only the characters that satisfy *pred*; `string-remove` selects only the characters that *not* satisfy *pred*

13.3.9 Replication & splitting

`string-repeat` *string-or-character len* [Procedure]

Create an istring by repeating the first argument *len* times. If the first argument is a character, it is as if it were wrapped with the `string` constructor. We can define string-repeat in terms of the more general `xsubstring` procedure:

```
(define (string-repeat S N)
   (let ((T (if (char? S) (string S) S)))
     (xsubstring T 0 (* N (string-length T)))))
```

`xsubstring` *string* [*from to* [*start end*]] [Procedure]

This is an extended substring procedure that implements replicated copying of a substring. The *string* is a string; *start* and *end* are optional arguments that specify a substring of *string*, defaulting to 0 and the length of *string*. This substring is conceptually replicated both up and down the index space, in both the positive and negative directions. For example, if *string* is `"abcdefg"`, *start* is 3, and *end* is 6, then we have the conceptual bidirectionally-infinite string

```
... d e f d e f d e f d e f d e f d e f d ...
```

```
-9 -8 -7 -6 -5 -4 -3 -2 -1  0 +1 +2 +3 +4 +5 +6 +7 +8 +9
```

`xsubstring` returns the substring of the *string* beginning at index *from*, and ending at *to*. It is an error if *from* is greater than *to*.

If *from* and *to* are missing they default to 0 and *from*+(*end*-*start*), respectively. This variant is a generalization of using `substring`, but unlike `substring` never shares substructures that would retain characters or sequences of characters that are substructures of its first argument or previously allocated objects.

You can use `xsubstring` to perform a variety of tasks:

- To rotate a string left: `(xsubstring "abcdef" 2 8)` ⇒ `"cdefab"`
- To rotate a string right: `(xsubstring "abcdef" -2 4)` ⇒ `"efabcd"`
- To replicate a string: `(xsubstring "abc" 0 7)` ⇒ `"abcabca"`

Note that

- The *from*/*to* arguments give a half-open range containing the characters from index *from* up to, but not including, index *to*.
- The *from*/*to* indexes are not expressed in the index space of *string*. They refer instead to the replicated index space of the substring defined by *string*, *start*, and *end*.

It is an error if *start*=*end*, unless *from*=*to*, which is allowed as a special case.

`string-split` *string delimiter* [*grammar limit start end*] [Procedure]

Returns a list of strings representing the words contained in the substring of *string* from *start* (inclusive) to *end* (exclusive). The *delimiter* is a string to be used as the word separator. This will often be a single character, but multiple characters are allowed for use cases such as splitting on `"\r\n"`. The returned list will have one more item than the number of non-overlapping occurrences of the *delimiter* in the string. If *delimiter* is an empty string, then the returned list contains a list of strings, each of which contains a single character.

The *grammar* is a symbol with the same meaning as in the `string-join` procedure. If it is `infix`, which is the default, processing is done as described above, except an empty string produces the empty list; if grammar is `strict-infix`, then an empty string signals an error. The values `prefix` and `suffix` cause a leading/trailing empty string in the result to be suppressed.

If *limit* is a non-negative exact integer, at most that many splits occur, and the remainder of string is returned as the final element of the list (so the result will have at most limit+1 elements). If limit is not specified or is #f, then as many splits as possible are made. It is an error if limit is any other value.

To split on a regular expression, you can use SRFI 115's `regexp-split` procedure.

13.3.10 String mutation

The following procedures create a mutable string, i.e. one that you can modify.

`make-string` [*k* [*char*]] [Procedure]

Return a newly allocated mstring of *k* characters, where *k* defaults to 0. If *char* is given, then all elements of the string are initialized to *char*, otherwise the contents of the *string* are unspecified.

The 1-argument version is deprecated as poor style, except when k is 0.

Rationale: In many languags the most common pattern for mutable strings is to allocate an empty string and incrementally append to it. It seems natural to initialize the string with `(make-string)`, rather than `(make-string 0)`.

To return an immutable string that repeats *k* times a character *char* use `string-repeat`.

This is as R7RS, except the result is variable-size and we allow leaving out *k* when it is zero.

`string-copy` *string* [*start* [*end*]] [Procedure]

Returns a newly allocated mutable (mstring) copy of the part of the given *string* between *start* and *end*.

The following procedures modify a mutable string.

`string-set!` *string k char* [Procedure]

This procedure stores *char* in element *k* of *string*.

```
(define s1 (make-string 3 #\*))
(define s2 "***")
(string-set! s1 0 #\?) ⇒ void
s1 ⇒ "?**"
(string-set! s2 0 #\?) ⇒ error
(string-set! (symbol->string 'immutable) 0 #\?) ⇒ error
```

Performance note: Calling `string-set!` may take time proportional to the length of the string: First it must scan for the right position, like `string-ref` does. Then if the new character requires using a surrogate pair (and the old one doesn't) then we have to make room in the string, possibly re-allocating a new `char` array. Alternatively, if the old character requires using a surrogate pair (and the new one doesn't) then following characters need to be moved.

The function `string-set!` is deprecated: It is inefficient, and it very seldom does the correct thing. Instead, you can construct a string with `string-append!`.

`string-append!` *string value ...* [Procedure]

The *string* must be a mutable string, such as one returned by `make-string` or `string-copy`. The `string-append!` procedure extends *string* by appending each *value* (in order) to the end of *string*. Each `value` should be a character or a string.

Performance note: The compiler converts a call with multiple *value*s to multiple `string-append!` calls. If a *value* is known to be a `character`, then no boxing (object-allocation) is needed.

The following example shows how to efficiently process a string using `string-for-each` and incrementally "build" a result string using `string-append!`.

```
(define (translate-space-to-newline str::string)::string
  (let ((result (make-string 0)))
    (string-for-each
     (lambda (ch)
       (string-append! result
                       (if (char=? ch #\Space) #\Newline ch)))
     str)
    result))
```

`string-copy!` *to at from* [*start* [*end*]] [Procedure]

Copies the characters of the string *from* that are between *start* end *end* into the string *to*, starting at index *at*. The order in which characters are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary string and then into the destination. (This is achieved without allocating storage by making sure to copy in the correct direction in such circumstances.)

This is equivalent to (and implemented as):

```
(string-replace! to at (+ at (- end start)) from start end))
```

```
(define a "12345")
(define b (string-copy "abcde"))
(string-copy! b 1 a 0 2)
b  ⇒  "a12de"
```

`string-replace!` *dst dst-start dst-end src* [*src-start* [*src-end*]] [Procedure]

Replaces the characters of string *dst* (between *dst-start* and *dst-end*) with the characters of *src* (between *src-start* and *src-end*). The number of characters from *src* may be different than the number replaced in *dst*, so the string may grow or contract. The special case where *dst-start* is equal to *dst-end* corresponds to insertion; the case where *src-start* is equal to *src-end* corresponds to deletion. The order in which characters are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary string and then into the destination. (This is achieved without allocating storage by making sure to copy in the correct direction in such circumstances.)

`string-fill!` *string fill* [*start* [*end*]] [Procedure]

The `string-fill!` procedure stores *fill* in the elements of *string* between *start* and *end*. It is an error if *fill* is not a character or is forbidden in strings.

13.3.11 Strings as sequences

13.3.11.1 Indexing a string

Using function-call syntax with strings is convenient and efficient. However, it has some “gotchas”.

We will use the following example string:

```
(! str1 "Smile \x1f603;!")
```

or if you’re brave:

```
(! str1 "Smile !")
```

This is `"Smile "` followed by an emoticon (“smiling face with open mouth”) followed by `"!"`. The emoticon has scalar value `\x1f603` - it is not in the 16-bit Basic Multi-language Plane, and so it must be encoded by a surrogate pair (`#\xd83d` followed by `#\xde03`).

The number of scalar values (`character`s) is 8, while the number of 16-bits code units (`char`s) is 9. The `java.lang.CharSequence:length` method counts `char`s. Both the `length` and the `string-length` procedures count `character`s. Thus:

```
(length str1)          ⇒ 8
(string-length str1)   ⇒ 8
(str1:length)          ⇒ 9
```

Counting `char`s is a constant-time operation (since it is stored in the data structure). Counting `character`s depends on the representation used: In geneeral it may take time proportional to the length of the string, since it has to subtract one for each surrogate pair; however the *istring* type (`gnu.lists.IString` class) uses a extra structure so it can count characters in constant-time.

Similarly we can can index the string in 3 ways:

```
(str1 1)              ⇒ #\m :: character
(string-ref str1 1)   ⇒ #\m :: character
(str1:charAt 1)       ⇒ #\m :: char
```

Using function-call syntax when the "function" is a string and a single integer argument is the same as using `string-ref`.

Things become interesting when we reach the emoticon:

```
(str1 6)              ⇒ #\ :: character
(str1:charAt 6)       ⇒ #\d83d :: char
```

Both `string-ref` and the function-call syntax return the real character, while the `charAt` methods returns a partial character.

```
(str1 7)              ⇒ #\! :: character
(str1:charAt 7)       ⇒ #\de03 :: char
(str1 8)              ⇒ throws StringIndexOutOfBoundsException
(str1:charAt 8)       ⇒ #\! :: char
```

13.3.11.2 Indexing with a sequence

You can index a string with a list of integer indexes, most commonly a range:

```
(str [i ...])
```

is basically the same as:

```
(string (str i) ...)
```

Generally when working with strings it is best to work with substrings rather than individual characters:

```
(str [start <: end])
```

This is equivalent to invoking the `substring` procedure:

```
(substring str start end)
```

13.3.12 String Cursor API

Indexing into a string (using for example `string-ref`) is inefficient because of the possible presence of surrogate pairs. Hence given an index *i* access normally requires linearly scanning the string until we have seen *i* characters.

The string-cursor API is defined in terms of abstract "cursor values", which point to a position in the string. This avoids the linear scan.

Typical usage is:

```
(let* ((str whatever)
       (end (string-cursor-end str)))
  (do ((sc::string-cursor (string-cursor-start str)
                          (string-cursor-next str sc)))
    ((string-cursor>=? sc end))
    (let ((ch (string-cursor-ref str sc)))
      (do-something-with ch))))
```

Alternatively, the following may be marginally faster:

```
(let* ((str whatever)
       (end (string-cursor-end str)))
  (do ((sc::string-cursor (string-cursor-start str)
                          (string-cursor-next-quick sc)))
    ((string-cursor>=? sc end))
    (let ((ch (string-cursor-ref str sc)))
      (if (not (char=? ch #\ignorable-char))
        (do-something-with ch)))))
```

The API is non-standard, but is based on that in Chibi Scheme.

`string-cursor` [Type]

An abstract position (index) in a string. Implemented as a primitive `int` which counts the number of preceding code units (16-bit `char` values).

`string-cursor-start` *str* [Procedure]

Returns a cursor for the start of the string. The result is always 0, cast to a `string-cursor`.

`string-cursor-end` *str* [Procedure]

Returns a cursor for the end of the string - one past the last valid character. Implemented as `(as string-cursor (invoke str 'length))`.

`string-cursor-ref` *str cursor* [Procedure]

Return the `character` at the *cursor*. If the *cursor* points to the second `char` of a surrogate pair, returns `#\ignorable-char`.

`string-cursor-next` *string cursor* [*count*] [Procedure]

Return the cursor position *count* (default 1) character positions forwards beyond *cursor*. For each *count* this may add either 1 or 2 (if pointing at a surrogate pair) to the *cursor*.

`string-cursor-next-quiet` *cursor* [Procedure]

Increment cursor by one raw `char` position, even if *cursor* points to the start of a surrogate pair. (In that case the next `string-cursor-ref` will return `#\ignorable-char`.) Same as `(+ cursor 1)` but with the `string-cursor` type.

`string-cursor-prev` *string cursor* [*count*] [Procedure]

Return the cursor position *count* (default 1) character positions backwards before *cursor*.

`substring-cursor` *string* [*start* [*end*]] [Procedure]

Create a substring of the section of *string* between the cursors *start* and *end*.

`string-cursor<?` *cursor1 cursor2* [Procedure]
`string-cursor<=?` *cursor1 cursor2* [Procedure]
`string-cursor=?` *cursor1 cursor2* [Procedure]
`string-cursor>=?` *cursor1 cursor2* [Procedure]
`string-cursor>?` *cursor1 cursor2* [Procedure]

Is the position of *cursor1* respectively before, before or same, same, after, or after or same, as *cursor2*.

Performance note: Implemented as the corresponding `int` comparison.

`string-cursor-for-each` *proc string* [*start* [*end*]] [Procedure]

Apply the procedure *proc* to each character position in *string* between the cursors *start* and *end*.

13.4 String literals

Kaw support two syntaxes of string literals: The traditional, portable, qdouble-quoted-delimited literals like `"this"`; and extended SRFI-109 quasi-literals like `&{this}`.

13.4.1 Simple string literals

string `::=` `"` *string-element* *`"`
string-element `::=` *any character other than* `"` *or* `\`
  `|` *mnemonic-escape* `|` `\` `"` `|` `\` `\`
  `|` `\` *intraline-whitespace* **line-ending* *intraline-whitespace* *
  `|` *inline-hex-escape*
mnemonic-escape `::=` `\` **a** `|` `\` **b** `|` `\` **t** `|` `\` **n** `|` `\` **r** `|` ... *(see below)*

A string is written as a sequence of characters enclosed within quotation marks ("). Within a string literal, various escape sequence represent characters other than themselves. Escape sequences always start with a backslash (\):

\a Alarm (bell), `#\x0007`.

\b Backspace, `#\x0008`.

\e Escape, `#\x001B`.

\f Form feed, `#\x000C`.

\n Linefeed (newline), `#\x000A`.

\r Return, `#\x000D`.

\t Character tabulation, `#\x0009`.

\v Vertical tab, `#\x000B`.

\C-*x*
\^*x* Returns the scalar value of *x* masked (anded) with `#x9F`. An alternative way to write the Ascii control characters: For example `"\C-m"` or `"\^m"` is the same as `"#\x000D"` (which the same as `"\r"`). As a special case `\^?` is rubout (delete) (`\x7f;`).

\x *hex-scalar-value***;**
\X *hex-scalar-value***;**
A hex encoding that gives the scalar value of a character.

\\ *oct-digit*$^+$
At most three octal digits that give the scalar value of a character. (Historical, for C compatibility.)

\u *hex-digit*$^+$
Exactly four hex digits that give the scalar value of a character. (Historical, for Java compatibility.)

\M-*x* (Historical, for Emacs Lisp.) Set the meta-bit (high-bit of single byte) of the following character *x*.

\| Vertical line, `#\x007c`. (Not useful for string literals, but useful for symbols.)

\" Double quote, `#\x0022`.

\\ Backslah, `#\005C`.

\intraline-whitespace* *line-ending intraline-whitespace**
Nothing (ignored). Allows you to split up a long string over multiple lines; ignoring initial whitespace on the continuation lines allows you to indent them.

Except for a line ending, any character outside of an escape sequence stands for itself in the string literal. A line ending which is preceded by \intraline-whitespace* expands to nothing (along with any trailing *intraline-whitespace*), and can be used to indent strings for improved legibility. Any other line ending has the same effect as inserting a **\n** character into the string.

Examples:

```
"The word \"recursion\" has many meanings."
"Another example:\ntwo lines of text"
"Here’s text \
containing just one line"
"\x03B1; is named GREEK SMALL LETTER ALPHA."
```

13.4.2 String templates

The following syntax is a *string template* (also called a string quasi-literal or “here document (`http://en.wikipedia.org/wiki/Here_document`)”):

```
&{Hello &[name]!}
```

Assuming the variable `name` evaluates to `"John"` then the example evaluates to `"Hello John!"`.

The Kawa reader converts the above example to:

```
($string$ "Hello " $<<$ name $>>$ "!")
```

See the SRFI-109 (`http://srfi.schemers.org/srfi-109/srfi-109.html`) specification for details.

> *extended-string-literal* `::=` **&{** [*initial-ignored*] *string-literal-part* * }
> *string-literal-part* `::=` *any character except* **&**, { *or* }
> | { *string-literal-part* * }
> | *char-ref*
> | *entity-ref*
> | *special-escape*
> | *enclosed-part*

You can use the plain "*string*" syntax for longer multiline strings, but `&{`*string*`}` has various advantages. The syntax is less error-prone because the start-delimiter is different from the end-delimiter. Also note that nested braces are allowed: a right brace `}` is only an end-delimiter if it is unbalanced, so you would seldom need to escape it:

```
&{This has a {braced} section.}
  ⇒ "This has a {braced} section."
```

The escape character used for special characters is **&**. This is compatible with XML syntax and Section 20.4 [XML literals], page 339.

13.4.2.1 Special characters

> *char-ref* `::=`
>   **&#** *digit* + **;**
> | **&#x** *hex-digit* + **;**
> *entity-ref* `::=`
>   **&** *char-or-entity-name* **;**
> *char-or-entity-name* `::=` *tagname*

You can the standard XML syntax for character references, using either decimal or hexadecimal values. The following string has two instances of the Ascii escape character, as either decimal 27 or hex 1B:

```
&{&#27;&#x1B;} ⇒ "\e\e"
```

You can also use the pre-defined XML entity names:

```
&{&amp; &lt; &gt; &quot; &apos;} ⇒ "& < > \" '"
```

In addition, `&lbrace;` `&rbrace;` can be used for left and right curly brace, though you don't need them for balanced parentheses:

```
&{ &rbrace;_&lbrace; / {_} }  ⇒ " }_{ / {_} "
```

You can use the standard XML entity names (`http://www.w3.org/2003/entities/2007/w3centities-f.ent`). For example:

```
&{L&aelig;rdals&oslash;yri}
  ⇒ "Lærdalsøyri"
```

You can also use the standard R7RS character names `null`, `alarm`, `backspace`, `tab`, `newline`, `return`, `escape`, `space`, and `delete`. For example:

```
&{&escape;&space;}
```

The syntax `&`*`name`*`;` is actually syntactic sugar (specifically reader syntax) to the variable reference `$entity$:`*`name`*. Hence you can also define your own entity names:

```
(define $entity$:crnl "\r\n")
```

```
&{&crnl;}  "\r\n"
```

13.4.2.2 Multiline string literals

```
initial-ignored ::=
     intraline-whitespace * line-ending  intraline-whitespace * &|
special-escape ::=
     intraline-whitespace * &|
  | & nested-comment
  | &-  intraline-whitespace * line-ending
```

A line-ending directly in the text is becomes a newline, as in a simple string literal:

```
(string-capitalize &{one two three
uno dos tres
}) ⇒ "One Two Three\nUno Dos Tres\n"
```

However, you have extra control over layout. If the string is in a nested expression, it is confusing (and ugly) if the string cannot be indented to match the surrounding context. The indentation marker `&|` is used to mark the end of insignificant initial whitespace. The `&|` characters and all the preceding whitespace are removed. In addition, it also suppresses an initial newline. Specifically, when the initial left-brace is followed by optional (invisible) intraline-whitespace, then a newline, then optional intraline-whitespace (the indentation), and finally the indentation marker `&|` - all of which is removed from the output. Otherwise the `&|` only removes initial intraline-whitespace on the same line (and itself).

```
(write (string-capitalize &{
     &|one two three
     &|uno dos tres
}) out)
     ⇒ prints "One Two Three\nUno Dos Tres\n"
```

As a matter of style, all of the indentation lines should line up. It is an error if there are any non-whitespace characters between the previous newline and the indentation marker. It is also an error to write an indentation marker before the first newline in the literal.

The line-continuation marker `&-` is used to suppress a newline:

```
&{abc&-
  def} ⇒ "abc  def"
```

You can write a `#|...|#`-style comment following a `&`. This could be useful for annotation, or line numbers:

```
&{&#|line 1|#one two
  &#|line 2|# three
  &#|line 3|#uno dos tres
} ⇒ "one two\n three\nuno dos tres\n"
```

13.4.2.3 Embedded expressions

```
enclosed-part ::=
    &  enclosed-modifier * [  expression * ]
  | &  enclosed-modifier * (  expression + )
```

An embedded expression has the form `&[expression]`. It is evaluated, the result converted to a string (as by `display`), and the result added in the result string. (If there are multiple expressions, they are all evaluated and the corresponding strings inserted in the result.)

```
&{Hello &[(string-capitalize name)]!}
```

You can leave out the square brackets when the expression is a parenthesized expression:

```
&{Hello &(string-capitalize name)!}
```

13.4.2.4 Formatting

```
enclosed-modifier ::=
  ~ format-specifier-after-tilde
```

Using Section 17.6 [Format], page 287, allows finer-grained control over the output, but a problem is that the association between format specifiers and data expressions is positional, which is hard-to-read and error-prone. A better solution places the specifier adjacant to the data expression:

```
&{The response was &~,2f(* 100.0 (/ responses total))%.}
```

The following escape forms are equivalent to the corresponding forms withput the ~*fmt-spec*, except the expression(s) are formatted using `format`:

&~*fmt-spec*[*expression* *]

Again using parentheses like this:

&~*fmt-spec*(*expression* +)

is equivalent to:

&~*fmt-spec*[(*expression* +)]

The syntax of `format` specifications is arcane, but it allows you to do some pretty neat things in a compact space. For example to include "_" between each element of the array `arr` you can use the `~{...~}` format speciers:

```
(define arr [5 6 7])
&{&~{&[arr]&~^_&~}} ⇒ "5_6_7"
```

If no format is specified for an enclosed expression, the that is equivalent to a `~a` format specifier, so this is equivalent to:

```
&{&~{&~a[arr]&~^_&~}} ⇒ "5_6_7"
```

which is in turn equivalent to:

```
(format #f "~{~a~^_~}" arr)
```

The fine print that makes this work: If there are multiple expressions in a `&[...]` with no format specifier then there is an implicit `~a` for each expression. On the other hand, if there is an explicit format specifier, it is not repeated for each enclosed expression: it appears exactly once in the effective format string, whether there are zero, one, or many expressions.

13.5 Unicode character classes and conversions

Some of the procedures that operate on characters or strings ignore the difference between upper case and lower case. These procedures have `-ci` (for “case insensitive”) embedded in their names.

13.5.1 Characters

`char-upcase` *`char`* [Procedure]
`char-downcase` *`char`* [Procedure]
`char-titlecase` *`char`* [Procedure]
`char-foldcase` *`char`* [Procedure]

These procedures take a character argument and return a character result.

If the argument is an upper–case or title–case character, and if there is a single character that is its lower–case form, then `char-downcase` returns that character.

If the argument is a lower–case or title–case character, and there is a single character that is its upper–case form, then `char-upcase` returns that character.

If the argument is a lower–case or upper–case character, and there is a single character that is its title–case form, then `char-titlecase` returns that character.

If the argument is not a title–case character and there is no single character that is its title–case form, then `char-titlecase` returns the upper–case form of the argument.

Finally, if the character has a case–folded character, then `char-foldcase` returns that character. Otherwise the character returned is the same as the argument.

For Turkic characters `#\x130` and `#\x131`, `char-foldcase` behaves as the identity function; otherwise `char-foldcase` is the same as `char-downcase` composed with `char-upcase`.

```
(char-upcase #\i)          ⇒  #\I
(char-downcase #\i)        ⇒  #\i
(char-titlecase #\i)       ⇒  #\I
(char-foldcase #\i)        ⇒  #\i

(char-upcase #\ß)          ⇒  #\ß
(char-downcase #\ß)        ⇒  #\ß
(char-titlecase #\ß)       ⇒  #\ß
(char-foldcase #\ß)        ⇒  #\ß

(char-upcase #\Σ)          ⇒  #\Σ
(char-downcase #\Σ)        ⇒  #\σ
(char-titlecase #\Σ)       ⇒  #\Σ
(char-foldcase #\Σ)        ⇒  #\σ

(char-upcase #\ς)          ⇒  #\Σ
(char-downcase #\ς)        ⇒  #\ς
(char-titlecase #\ς)       ⇒  #\Σ
(char-foldcase #\ς)        ⇒  #\σ
```

Note: `char-titlecase` does not always return a title–case character.

Note: These procedures are consistent with Unicode's locale–independent mappings from scalar values to scalar values for upcase, downcase, titlecase, and case–folding operations. These mappings can be extracted from `UnicodeData.txt` and `CaseFolding.txt` from the Unicode Consortium, ignoring Turkic mappings in the latter.

Note that these character–based procedures are an incomplete approximation to case conversion, even ignoring the user's locale. In general, case mappings require the context of a string, both in arguments and in result. The `string-upcase`, `string-downcase`, `string-titlecase`, and `string-foldcase` procedures perform more general case conversion.

`char-ci=?` *char₁ char₂ char₃ . . .* [Procedure]
`char-ci<?` *char₁ char₂ char₃ . . .* [Procedure]
`char-ci>?` *char₁ char₂ char₃ . . .* [Procedure]
`char-ci<=?` *char₁ char₂ char₃ . . .* [Procedure]
`char-ci>=?` *char₁ char₂ char₃ . . .* [Procedure]

These procedures are similar to `char=?`, etc., but operate on the case–folded versions of the characters.

```
(char-ci<? #\z #\Z)          ⇒ #f
(char-ci=? #\z #\Z)          ⇒ #f
(char-ci=? #\ς #\σ)          ⇒ #t
```

`char-alphabetic?` *char* [Procedure]
`char-numeric?` *char* [Procedure]
`char-whitespace?` *char* [Procedure]
`char-upper-case?` *char* [Procedure]
`char-lower-case?` *char* [Procedure]
`char-title-case?` *char* [Procedure]

These procedures return `#t` if their arguments are alphabetic, numeric, whitespace, upper–case, lower–case, or title–case characters, respectively; otherwise they return `#f`.

A character is alphabetic if it has the Unicode "Alphabetic" property. A character is numeric if it has the Unicode "Numeric" property. A character is whitespace if has the Unicode "White_Space" property. A character is upper case if it has the Unicode "Uppercase" property, lower case if it has the "Lowercase" property, and title case if it is in the Lt general category.

```
(char-alphabetic? #\a)          ⇒  #t
(char-numeric? #\1)             ⇒  #t
(char-whitespace? #\space)      ⇒  #t
(char-whitespace? #\x00A0)      ⇒  #t
(char-upper-case? #\Σ)          ⇒  #t
(char-lower-case? #\σ)          ⇒  #t
(char-lower-case? #\x00AA)      ⇒  #t
(char-title-case? #\I)          ⇒  #f
(char-title-case? #\x01C5)      ⇒  #t
```

`char-general-category` *char* [Procedure]

Return a symbol representing the Unicode general category of *char*, one of `Lu`, `Ll`, `Lt`, `Lm`, `Lo`, `Mn`, `Mc`, `Me`, `Nd`, `Nl`, `No`, `Ps`, `Pe`, `Pi`, `Pf`, `Pd`, `Pc`, `Po`, `Sc`, `Sm`, `Sk`, `So`, `Zs`, `Zp`, `Zl`, `Cc`, `Cf`, `Cs`, `Co`, or `Cn`.

```
(char-general-category #\a)         ⇒ Ll
(char-general-category #\space)     ⇒ Zs
(char-general-category #\x10FFFF)   ⇒ Cn
```

13.5.2 Deprecated in-place case modification

The following functions are deprecated; they really don't and cannot do the right thing, because in some languages upper and lower case can use different number of characters.

`string-upcase!` *str* [Procedure]

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

`string-downcase!` *str* [Procedure]

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

`string-capitalize!` *str* [Procedure]

Deprecated: 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.

13.6 Regular expressions

Kawa provides *regular expressions*, which is a convenient mechanism for matching a string against a *pattern* and maybe replacing matching parts.

A regexp is a string that describes a pattern. A regexp matcher tries to match this pattern against (a portion of) another string, which we will call the text string. The text string is treated as raw text and not as a pattern.

Most of the characters in a regexp pattern are meant to match occurrences of themselves in the text string. Thus, the pattern "`abc`" matches a string that contains the characters "`a`", "`b`", "`c`" in succession.

In the regexp pattern, some characters act as *metacharacters*, and some character sequences act as *metasequences*. That is, they specify something other than their literal selves. For example, in the pattern "`a.c`", the characters "`a`" and "`c`" do stand for themselves but the metacharacter "`.`" can match any character (other than newline). Therefore, the pattern "`a.c`" matches an "`a`", followed by any character, followed by a "`c`".

If we needed to match the character "`.`" itself, we *escape* it, ie, precede it with a backslash "`\`". The character sequence "`\.`" is thus a metasequence, since it doesn't match itself but rather just "`.`". So, to match "`a`" followed by a literal "`.`" followed by "`c`" we use the regexp pattern "`a\.c`". To write this as a Scheme string literal, you need to quote the backslash, so you need to write `"a\\.c"`. Kawa also allows the literal syntax `#/a\.c/`, which avoids the need to double the backslashes.

You can choose between two similar styles of regular expressions. The two differ slightly in terms of which characters act as metacharacters, and what those metacharacters mean:

- Functions starting with `regex-` are implemented using the `java.util.regex` package. This is likely to be more efficient, has better Unicode support and some other minor extra features, and literal syntax `#/a\.c/` mentioned above.
- Functions starting with `pregexp-` are implemented in pure Scheme using Dorai Sitaram's "Portable Regular Expressions for Scheme" library. These will be portable to more Scheme implementations, including BRL, and is available on older Java versions.

13.6.1 Java regular expressions

The syntax for regular expressions is documented here (`http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html`).

`regex` [Type]

A compiled regular expression, implemented as `java.util.regex.Pattern`.

`regex` *arg* [Constructor]

Given a regular expression pattern (as a string), compiles it to a `regex` object.

```
(regex "a\\.c")
```

This compiles into a pattern that matches an "`a`", followed by any character, followed by a "`c`".

The Scheme reader recognizes "`#/`" as the start of a regular expression *pattern literal*, which ends with the next un-escaped "`/`". This has the big advantage that you don't need to double the backslashes:

```
#/a\.c/
```

This is equivalent to `(regex "a\\.c")`, except it is compiled at read-time. If you need a literal "`/`" in a pattern, just escape it with a backslash: "`#/a\/c/`" matches a "`a`", followed by a "`/`", followed by a "`c`".

You can add single-letter *modifiers* following the pattern literal. The following modifiers are allowed:

`i` The modifier "`i`" cause the matching to ignore case. For example the following pattern matches "`a`" or "`A`".

```
#/a/i
```

`m` Enables "metaline" mode. Normally metacharacters "`^`" and "`$`' match at the start end end of the entire input string. In metaline mode "`^`" and "`$`" also match just before or after a line terminator.

Multiline mode can also be enabled by the metasequence "`(?m)`".

`s` Enable "singleline" (aka "dot-all") mode. In this mode the matacharacter "`.` matches any character, including a line breaks. This mode be enabled by the metasequence "`(?s)`".

The following functions accept a regex either as a pattern string or a compiled `regex` pattern. I.e. the following are all equivalent:

```
(regex-match "b\\.c" "ab.cd")
(regex-match #/b\.c/ "ab.cd")
(regex-match (regex "b\\.c") "ab.cd")
(regex-match (java.util.regex.Pattern:compile "b\\.c") "ab.cd")
```

These all evaluate to the list `("b.c")`.

The following functions must be imported by doing one of:

```
(require 'regex) ;; or
(import (kawa regex))
```

`regex-match-positions` *regex string* [*start* [*end*]] [Procedure]

The procedure `regexmatchposition` takes pattern and a text *string*, and returns a match if the regex matches (some part of) the text string.

Returns `#f` if the regexp did not match the string; and a list of index pairs if it did match.

```
(regex-match-positions "brain" "bird") ⇒ #f
(regex-match-positions "needle" "hay needle stack")
  ⇒ ((4 . 10))
```

In the second example, the integers 4 and 10 identify the substring that was matched. 4 is the starting (inclusive) index and 10 the ending (exclusive) index of the matching substring.

```
(substring "hay needle stack" 4 10) ⇒ "needle"
```

In this case the return list contains only one index pair, and that pair represents the entire substring matched by the regexp. When we discuss subpatterns later, we will see how a single match operation can yield a list of submatches.

`regexmatchpositions` takes optional third and fourth arguments that specify the indices of the text string within which the matching should take place.

```
(regex-match-positions "needle"
  "his hay needle stack -- my hay needle stack -- her hay needle stack"
  24 43)
  ⇒ ((31 . 37))
```

Note that the returned indices are still reckoned relative to the full text string.

`regex-match` *regex string* [*start* [*end*]] [Procedure]

The procedure `regexmatch` is called like `regexmatchpositions` but instead of returning index pairs it returns the matching substrings:

```
(regex-match "brain" "bird") ⇒ #f
(regex-match "needle" "hay needle stack")
  ⇒ ("needle")
```

`regexmatch` also takes optional third and fourth arguments, with the same meaning as does `regexmatchpositions`.

`regex-split` *regex string* [Procedure]

Takes two arguments, a *regex* pattern and a text *string*, and returns a list of substrings of the text string, where the pattern identifies the delimiter separating the substrings.

```
(regex-split ":" "/bin:/usr/bin:/usr/bin/X11:/usr/local/bin")
  ⇒ ("/bin" "/usr/bin" "/usr/bin/X11" "/usr/local/bin")

(regex-split " " "pea soup")
  ⇒ ("pea" "soup")
```

If the first argument can match an empty string, then the list of all the single-character substrings is returned, plus we get a empty strings at each end.

```
(regex-split "" "smithereens")
  ⇒ ("" "s" "m" "i" "t" "h" "e" "r" "e" "e" "n" "s" "")
```

(Note: This behavior is different from `pregexp-split`.)

To identify one-or-more spaces as the delimiter, take care to use the regexp “ `+`”, not “ `*`”.

```
(regex-split " +" "split pea     soup")
  ⇒ ("split" "pea" "soup")
(regex-split " *" "split pea     soup")
  ⇒ ("" "s" "p" "l" "i" "t" "" "p" "e" "a" "" "s" "o" "u" "p" "")
```

`regexreplace` *regex string replacement* [Procedure]

Replaces the matched portion of the text *string* by another a *replacdement* string.

```
(regex-replace "te" "liberte" "ty")
  ⇒ "liberty"
```

Submatches can be used in the replacement string argument. The replacement string can use “`$n`” as a *backreference* to refer back to the *n*th submatch, ie, the substring that matched the *n*th subpattern. “`$0`” refers to the entire match.

```
(regex-replace #/_(.+?)_/
               "the _nina_, the _pinta_, and the _santa maria_"
"*$1*"))
  ⇒ "the *nina*, the _pinta_, and the _santa maria_"
```

`regexreplace*` *regex string replacement* [Procedure]

Replaces all matches in the text *string* by the *replacement* string:

```
(regex-replace* "te" "liberte egalite fraternite" "ty")
  ⇒ "liberty egality fratyrnity"
(regex-replace* #/_(.+?)_/
                "the _nina_, the _pinta_, and the _santa maria_"
                "*$1*")
  ⇒ "the *nina*, the *pinta*, and the *santa maria*"
```

`regex-quote` *pattern* [Procedure]

Takes an arbitrary string and returns a pattern string that precisely matches it. In particular, characters in the input string that could serve as regex metacharacters are escaped as needed.

```
(regex-quote "cons")
  ⇒ "\Qcons\E"
```

`regexquote` is useful when building a composite regex from a mix of regex strings and verbatim strings.

13.6.2 Portable Scheme regular expressions

This provides the procedures `pregexp`, `pregexpmatchpositions`, `pregexpmatch`, `pregexpsplit`, `pregexpreplace`, `pregexpreplace*`, and `pregexpquote`.

Before using them, you must require them:

```
(require 'pregexp)
```

These procedures have the same interface as the corresponding `regex-` versions, but take slightly different pattern syntax. The replace commands use “`\`” instead of “`$`” to indicate substitutions. Also, `pregexpsplit` behaves differently from `regexsplit` if the pattern can match an empty string.

See here for details (`http://www.ccs.neu.edu/home/dorai/pregexp/index.html`).

14 Data structures

14.1 Sequences

A *sequence* is a generalized list, consisting of zero or more values. You can choose between a number of different kinds of sequence implementations. Scheme traditionally has Section 14.2 [Lists], page 235, and Section 14.3 [Vectors], page 237. Any Java class that implements `java.util.List` is a sequence type. Raw Java arrays can also be viewerd as a sequence, and strings can be viewed a sequence (or vector) of characters. Kawa also provides Section 14.4 [Uniform vectors], page 239.

Sequence types differ in their API, but given a sequence type *stype* you can construct instances of that type using the syntax:

```
(stype v0 v1 .... vn)
```

For example:

```
(bytevector 9 8 7 6)  ⇒ #u8(9 8 7 6)
```

For a raw Java class name *jname* you may need to use the empty keyword `||:` to separate constructor parameters (if any) from sequence elements, as in:

```
(gnu.lists.U8Vector ||: 9 8 7 6)  ⇒ #u8(9 8 7 6)
```

This syntax works with any type with a default constructor and a 1-argument `add` method; see Section 19.10 [Allocating objects], page 324, for details. You can use the same syntax for allocating arrays, though array creation supports [Creating-new-Java-arrays], page 331.

To extract an element from Scheme sequence of type *stype* there is usually a function *stype*`-ref`. For example:

```
(define vec1 (vector 5 6 7 8))
(vector-ref vec1 2) ⇒ 7
```

More concisely, you can use (Kawa-specific) function call syntax:

```
(vec1 3) ⇒ 8
```

The index can be another sequence, which creates a new sequence of the selected indexes:

```
(vec1 [3 0 2 1]) ⇒ #(8 5 7 6)
```

It is convenient to use a Section 14.6 [Ranges], page 246, to select a sub-sequence:

```
(vec1 [1 <=: 3]) ⇒ #(6 7 8)
(vec1 [2 <:]) ⇒ #(7 8)
```

The same function call syntax also works for raw Java arrays (though the index is restricted to an integer, not a sequence or array):

```
(define arr1 (long[] 4 5 6 7))
(arr1 3) ⇒ 7
```

To assign to (replace) an element from a sequence of Scheme type *stype* there is usually a function `stype-set!`:

```
(vector-set! vec1 1 9)
vec1 ⇒ #(5 9 7 8)
```

Again, you can use the function call syntax:

```
(set! (vec1 2) 'x)
vec1 ⇒ #(5 9 x 8)
```

`length` *seq* [Procedure]

Returns the number of elements of the *seq*.

```
(length '(a b c))             ⇒  3
(length '(a (b) (c d e)))     ⇒  3
(length '())                  ⇒  0
(length [3 4 [] 12])          ⇒  4
(length (vector))             ⇒  0
(length (int[] 7 6))          ⇒  2
```

The length of a string is the number of characters (Unicode code points). In contrast, the `length` *method* (of the `CharSequence` interface) returns the number of 16-bit code points:

```
(length "Hello")              ⇒  5
(define str1 "Hello \x1f603;!")
(invoke str1 'length)         ⇒  9
(length str1)                 ⇒  8 ; Used to return 9 in Kawa 2.x.
(string-length str1)          ⇒  8
```

14.2 Lists

A pair (sometimes called a *dotted pair*) is a record structure with two fields called the car and cdr fields (for historical reasons). Pairs are created by the procedure `cons`. The car and cdr fields are accessed by the procedures `car` and `cdr`. The car and cdr fields are assigned by the procedures `set-car!` and `set-cdr!`.

Pairs are used primarily to represent lists. A *list* can be defined recursively as either the empty list or a pair whose cdr is a list. More precisely, the set of lists is defined as the smallest set X such that:

- The empty list is in X.
- If *list* is in X, then any pair whose cdr field contains *list* is also in X.

The objects in the car fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type. It is not a pair, it has no elements, and its length is zero.

Note: The above definitions imply that all lists have finite length and are terminated by the empty list.

The most general notation (external representation) for Scheme pairs is the "dotted" notation `(`*`c1`*` . `*`c2`*` )` where *c1* is the value of the car field and *c2* is the value of the cdr field. For example `(4 . 5)` is a pair whose car is 4 and whose cdr is 5. Note that `(4 . 5)` is the external representation of a pair, not an expression that evaluates to a pair.

A more streamlined notation can be used for lists: the elements of the list are simply enclosed in parentheses and separated by spaces. The empty list is written `()`. For example,

```
(a b c d e)
```

and

```
(a . (b . (c . (d . (e . ())))))
```

are equivalent notations for a list of symbols.

A chain of pairs not ending in the empty list is called an *improper list*. Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists:

```
(a b c . d)
```

is equivalent to

```
(a . (b . (c . d)))
```

Needs to finish merging from R7RS!

`make-list` *k* [*fill*] [Procedure]

Returns a newly allocated list of *k* elements. If a second argument is given, the each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

```
(make-list 2 3)   ⇒ (3 3)
```

14.2.1 SRFI-1 list library

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* [Procedure]

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.

14.2.2 SRFI-101 Purely Functional Random-Access Pairs and Lists

SRFI-101 (`http://srfi.schemers.org/srfi-101/srfi-101.html`) specifies immutable (read-only) lists with fast (logarithmic) indexing and functional update (i.e. return a modified list). These are implemented by a `RAPair` class which extends the generic `pair` type, which means that most code that expects a standard list will work on these lists as well.

14.3 Vectors

Vectors are heterogeneous structures whose elements are indexed by integers. A vector typically occupies less space than a list of the same length, and the average time needed to access a randomly chosen element is typically less for the vector than for the list.

The *length* of a vector is the number of elements that it contains. This number is a non–negative integer that is fixed when the vector is created. The *valid indices* of a vector are the exact non–negative integer objects less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Vectors are written using the notation `#(obj ...)`. For example, a vector of length 3 3 containing the number zero in element 0, the list `(2 2 2 2)` in element 1, and the string `"Anna"` in element 2 can be written as following:

```
#(0 (2 2 2 2) "Anna")
```

Note that this is the external representation of a vector. In Kawa, a vector datum is self-evaluating, but for style (and compatibility with R7RS) is is suggested you quote a vector constant:

```
'#(0 (2 2 2 2) "Anna")  ⇒ #(0 (2 2 2 2) "Anna")
```

Compare these different ways of creating a vector:

`(vector a b c)`
: In this case `a`, `b`, and `c` are expressions evaluated at run-time and the results used to initialize a newly-allocated 3-element vector.

`[a b c]`
: Same as using vector, but more concise, and results in an immutable (non-modifiable) vector.

`#(a b c)`
: This is reader syntax and creates a vector literal, at read-time, early in compile-time. The symbols `a`, `b`, and `c` are not evaluated but instead used literally.

``#(,a ,b ,c)`
: This is reader-syntax, using quasi-quotation, so `a`, `b`, and `c` are expressions evaluated at run-time. This is equivalent to `[a b c]` in that it results in an immutable vector.

`vector` [Type]

The type of vector objects.

`vector` *`obj ...`* [Constructor]

Return a newly allocated vector whose elements contain the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c)              ⇒  #(a b c)
```

Alternatively, you can use square-bracket syntax, which results in an immutable vector:

```
['a 'b 'c]               ⇒  #(a b c)
```

`make-vector` *k* [Procedure]
`make-vector` *k fill* [Procedure]

Return a newly allocated vector of *k* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is `#!null`.

`vector?` *obj* [Procedure]

Return `#t` if *obj* is a vector, `#f` otherwise.

`vector-length` *vector* [Procedure]

Return the number of elements in *vector* as an exact integer.

`vector-ref` *vector k* [Procedure]

It is an error if *k* is not a valid index of *vector*. The `vector-ref` procedure returns the contents of element *k* of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21) 5)     ⇒  8
(vector-ref '#(1 1 2 3 5 8 13 21)
  (inexact->exact (round (* 2 (acos -1)))))
⇒ 13
```

`vector-set!` *vector k obj* [Procedure]

It is an error if *k* is not a valid index of *vector*. The `vector-set!` procedure stores *obj* in element *k* of *vector*, and returns no values.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
  ⇒  #(0 ("Sue" "Sue") "Anna")

(vector-set! '#(0 1 2) 1 "doe")
  ⇒  error    ;; constant vector
```

A concise alternative to `vector-ref` and `vector-set!` is to use function call syntax. For example:

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (set! (vec 1) '("Sue" "Sue"))
  (list (vec 2) (vec 1)))
  ⇒  ("Anna" ("Sue" "Sue"))
```

`vector->list` *vector* [*start* [*end*]] [Procedure]

The `vector->list` procedure returns a newly allocated list of the objects contained in the elements of *vector* between *start* and *end*.

```
(vector->list '#(dah dah didah))       ⇒  (dah dah didah)
(vector->list '#(dah dah didah) 1 2)   ⇒  (dah)
```

`list->vector` *list* [Procedure]

The `list->vector` procedure returns a newly created vector initialized to the elements of the list *list*, in order.

```
(list->vector '(dididit dah))          ⇒  #(dididit dah)
```

`vector-copy` *vector* [*start* [*end*]] [Procedure]

Returns a newly allocated copy of the elements of the given *vector* between *start* and *end* . The elements of the new vector are the same (in the sense of `eqv?`) as the elements of the old.

```
(define a #(1 8 2 8)) ; a may be immutable
(define b (vector-copy a))
(vector-set! b 0 3)   ; b is mutable
b                     ⇒      #(3 8 2 8)
(define c (vector-copy b 1 3))
c                     ⇒ #(8 2)
```

`vector-copy!` *to at from* [*start* [*end*]] [Procedure]

Copies the elements of vector from between start and end to vector to, starting at at. The order in which elements are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary vector and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

It is an error if *at* is less than zero or greater than the length of *to*. It is also an error if `(- (vector-length` *`to`*`) at)` is less than `(-` *`end start`*`)`.

```
(define a (vector 1 2 3 4 5))
(define b (vector 10 20 30 40 50))
(vector-copy! b 1 a 0 2)
b    ⇒ #(10 1 2 40 50)
```

`vector-append` *`arg...`* [Procedure]

Creates a newly allocated vector whose elements are the concatenation of the elements of the given arguments. Each *arg* may be a vector or a list.

```
(vector-append #(a b c) #(d e f))
    ⇒ #(a b c d e f)
```

`vector-fill!` *`vector fill`* [*start* [*end*]] [Procedure]

Stores *fill* in in the elements of *vector* between *start* and *end*.

```
(define a (vector 1 2 3 4 5))
(vector-fill! a 'smash 2 4)
a  ⇒ #(1 2 smash smash 5)
```

The procedures `vector-map` and `vector-for-each` are documented in Section 9.1 [Mapping functions], page 156.

14.4 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`). The type names (such as `s8vector`) are a Kawa extension.

uniform-vector `::=` # *uniform-tag list*
uniform-tag `::=` **f32** `|` **f64**
    `|` **s8** `|` **s16** `|` **s32** `|` **s64**

| u8 | u16 | u32 | u64

This example is a literal for a 5-element vector of unsigned short (`ushort`) values:

```
(define uvec1 #u16(64000 3200 160 8 0))
```

Since a uniform vector is a sequence, you can use function-call notation to index one. For example:

```
(uvec1 1) ⇒ 3200
```

In this case the result is a primitive unsigned short (`ushort`), which is converted to a `gnu.math.UShort` if an object is needed.

`s8vector` [Type]

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

`u8vector` [Type]

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.

This type is a synonym for `bytevector`, which has Section 14.5 [Bytevectors], page 243.

`s16vector` [Type]

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

`u16vector` [Type]

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` [Type]

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

`u32vector` [Type]

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` [Type]

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

`u64vector` [Type]

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` [Type]

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

`f64vector` [Type]

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

`s8vector?` *value* [Procedure]
`u8vector?` *value* [Procedure]
`s16vector?` *value* [Procedure]
`u16vector?` *value* [Procedure]
`s32vector?` *value* [Procedure]
`u32vector?` *value* [Procedure]
`s64vector?` *value* [Procedure]
`u64vector?` *value* [Procedure]
`f32vector?` *value* [Procedure]
`f64vector?` *value* [Procedure]

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

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

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 ...* [Constructor]
`u8vector` *value ...* [Constructor]
`s16vector` *value ..* [Constructor]
`u16vector` *value ...* [Constructor]
`s32vector` *value ...* [Constructor]
`u32vector` *value ...* [Constructor]
`s64vector` *value ...* [Constructor]
`u64vector` *value ...* [Constructor]
`f32vector` *value ...* [Constructor]
`f64vector` *value ...* [Constructor]

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* [Procedure]
`u8vector-length` *v* [Procedure]
`s16vector-length` *v* [Procedure]
`u16vector-length` *v* [Procedure]
`s32vector-length` *v* [Procedure]
`u32vector-length` *v* [Procedure]
`s64vector-length` *v* [Procedure]
`u64vector-length` *v* [Procedure]
`f32vector-length` *v* [Procedure]

`f64vector-length` *v* [Procedure]

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

`s8vector-ref` *v i* [Procedure]
`u8vector-ref` *v i* [Procedure]
`s16vector-ref` *v i* [Procedure]
`u16vector-ref` *v i* [Procedure]
`s32vector-ref` *v i* [Procedure]
`u32vector-ref` *v i* [Procedure]
`s64vector-ref` *v i* [Procedure]
`u64vector-ref` *v i* [Procedure]
`f32vector-ref` *v i* [Procedure]
`f64vector-ref` *v i* [Procedure]

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

`s8vector-set!` *v i x* [Procedure]
`u8vector-set!` *v i x* [Procedure]
`s16vector-set!` *v i x* [Procedure]
`u16vector-set!` *v i x* [Procedure]
`s32vector-set!` *v i x* [Procedure]
`u32vector-set!` *v i x* [Procedure]
`s64vector-set!` *v i x* [Procedure]
`u64vector-set!` *v i x* [Procedure]
`f32vector-set!` *v i x* [Procedure]
`f64vector-set!` *v i x* [Procedure]

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* [Procedure]
`u8vector->list` *v* [Procedure]
`s16vector->list` *v* [Procedure]
`u16vector->list` *v* [Procedure]
`s32vector->list` *v* [Procedure]
`u32vector->list` *v* [Procedure]
`s64vector->list` *v* [Procedure]
`u64vector->list` *v* [Procedure]
`f32vector->list` *v* [Procedure]
`f64vector->list` *v* [Procedure]

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

`list->s8vector` *l* [Procedure]
`list->u8vector` *l* [Procedure]
`list->s16vector` *l* [Procedure]
`list->u16vector` *l* [Procedure]
`list->s32vector` *l* [Procedure]
`list->u32vector` *l* [Procedure]
`list->s64vector` *l* [Procedure]
`list->u64vector` *l* [Procedure]

`list->f32vector` *l* [Procedure]
`list->f64vector` *l* [Procedure]

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.

14.4.1 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`.

14.5 Bytevectors

Bytevectors represent blocks of binary data. They are fixed-length sequences of bytes, where a *byte* is an exact integer in the range [0, 255]. A bytevector is typically more space-efficient than a vector containing the same values.

The length of a bytevector is the number of elements that it contains. This number is a non-negative integer that is fixed when the bytevector is created. The valid indexes of a bytevector are the exact non-negative integers less than the length of the bytevector, starting at index zero as with vectors.

The `bytevector` type is equivalent to the `u8vector` Section 14.4 [Uniform vectors], page 239, type, but is specified by the R7RS standard.

Bytevectors are written using the notation `#u8(byte . . . )`. For example, a bytevector of length 3 containing the byte 0 in element 0, the byte 10 in element 1, and the byte 5 in element 2 can be written as following:

```
#u8(0 10 5)
```

Bytevector constants are self-evaluating, so they do not need to be quoted in programs.

`bytevector` [Type]

The type of bytevector objects.

`bytevector` *byte . . .* [Constructor]

Return a newly allocated bytevector whose elements contain the given arguments. Analogous to `vector`.

```
(bytevector 1 3 5 1 3 5)  ⇒  #u8(1 3 5 1 3 5)
(bytevector)  ⇒  #u8()
```

`bytevector?` *obj* [Procedure]

Return `#t` if *obj* is a bytevector, `#f` otherwise.

`make-bytevector` *k* [Procedure]
`make-bytevector` *k byte* [Procedure]

The `make-bytevector` procedure returns a newly allocated bytevector of length *k*. If *byte* is given, then all elements of the bytevector are initialized to *byte*, otherwise the contents of each element are unspecified.

```
(make-bytevector 2 12) ⇒ #u8(12 12)
```

`bytevector-length` *bytevector* [Procedure]

Returns the length of *bytevector* in bytes as an exact integer.

`bytevector-u8-ref` *bytevector k* [Procedure]

It is an error if *k* is not a valid index of *bytevector*. Returns the *k*th byte of *bytevector*.

```
(bytevector-u8-ref '#u8(1 1 2 3 5 8 13 21) 5)
  ⇒ 8
```

`bytevector-u8-set!` *bytevector k byte* [Procedure]

It is an error if *k* is not a valid index of *bytevector*. Stores *byte* as the *k*th byte of *bytevector*.

```
(let ((bv (bytevector 1 2 3 4)
  (bytevector-u8-set! bv 1 3)
  bv)
  ⇒ #u8(1 3 3 4)
```

`bytevector-copy` *bytevector* [*start* [*end*]] [Procedure]

Returns a newly allocated bytevector containing the bytes in *bytevector* between *start* and *end*.

```
(define a #u8(1 2 3 4 5))
(bytevector-copy a 2 4))
    ⇒ #u8(3 4)
```

`bytevector-copy!` *to at from* [*start* [*end*]] [Procedure]

Copies the bytes of bytevector*from* between *start* and *end* to bytevector *to*, starting at *at*. The order in which bytes are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary bytevector and then into the destination. This is achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

It is an error if *at* is less than zero or greater than the length of *to*. It is also an error if `(- (bytevector-length to) at)` is less than `(- end start)`.

```
(define a (bytevector 1 2 3 4 5))
(define b (bytevector 10 20 30 40 50))
(bytevector-copy! b 1 a 0 2)
b        ⇒ #u8(10 1 2 40 50)
```

`bytevector-append` *bytevector...* [Procedure]

Returns a newly allocated bytevector whose elements are the concatenation of the elements in the given bytevectors.

```
(bytevector-append #u8(0 1 2) #u8(3 4 5))
        ⇒  #u8(0 1 2 3 4 5)
```

14.5.1 Converting to or from strings

`utf8->string` *bytevector* [*start* [*end*]] [Procedure]

This procedure decodes the bytes of a bytevector between *start* and *end*, interpreting as a UTF-8-encoded string, and returns the corresponding string. It is an error for *bytevector* to contain invalid UTF-8 byte sequences.

```
(utf8->string #u8(#x41))  ⇒ "A"
```

`utf16->string` *bytevector* [*start* [*end*]] [Procedure]
`utf16be->string` *bytevector* [*start* [*end*]] [Procedure]
`utf16le->string` *bytevector* [*start* [*end*]] [Procedure]

These procedures interpret their `<var>bytevector</var>` argument as a UTF-16 encoding of a sequence of characters, and return an istring containing that sequence.

The bytevector subrange given to `utf16->string` may begin with a byte order mark (BOM); if so, that BOM determines whether the rest of the subrange is to be interpreted as big-endian or little-endian; in either case, the BOM will not become a character in the returned string. If the subrange does not begin with a BOM, it is decoded using the same implementation-dependent endianness used by `string->utf16`.

The `utf16be->string` and `utf16le->string` procedures interpret their inputs as big-endian or little-endian, respectively. If a BOM is present, it is treated as a normal character and will become part of the result.

It is an error if `(- end start)` is odd, or if the bytevector subrange contains invalid UTF-16 byte sequences.

`string->utf8` *string* [*start* [*end*]] [Procedure]

This procedure encodes the characters of a string between *start* and *end* and returns the corresponding bytevector, in UTF-8 encoding.

```
(string->utf8 "λ")      ⇒ " #u8(#xCE #xBB)
```

`string->utf16` *string* [*start* [*end*]] [Procedure]
`string->utf16be` *string* [*start* [*end*]] [Procedure]
`string->utf16le` *string* [*start* [*end*]] [Procedure]

These procedures return a newly allocated (unless empty) bytevector containing a UTF-16 encoding of the given substring.

The bytevectors returned by `string->utf16be` and `string->utf16le` do not contain a byte-order mark (BOM); `string->utf16be>` returns a big-endian encoding, while `string->utf16le` returns a little-endian encoding.

The bytevectors returned by `string->utf16` begin with a BOM that declares an implementation-dependent endianness, and the bytevector elements following that BOM encode the given substring using that endianness.

Rationale: These procedures are consistent with the Unicode standard. Unicode suggests UTF-16 should default to big-endian, but Microsoft prefers little-endian.

14.6 Ranges

A *range* is an immutable sequence of values that increase “linearly” - i.e. by a fixed amount. Most commonly it’s a sequence of consequtive integers. An example of the syntax is `[3 <: 7]` which evaluates to the sequence `[3 4 5 6]`. You can specify an explicit increment with a `by:` option. There are multiple ways to specify when the sequence stops. For example `[3 by 2 <=: 7]` is the even numbers from 3 up to 7 (inclusive, because of the `<=`).

Ranges are very useful for loop indexes, or selecting a sub-sequence. If you have a sequence *q* and a range *r*, and you use the syntax `(q r)` to “apply” *q* with the argument *r*, is result is to select elements of *q* with indexes in *r*.

```
("ABCDEFG" [1 by: 2 <: 7])  ⇒ "BDF"
```

A range can be *unbounded*, or non-finite, if you leave off the end value. For example `[3 by: 2]` is the odd integers starting at 3.

unbounded-range `::=`
  [*start-expression* **by:** *step-expression*]
  `|` [*start-expression* **<:**]

The expression `[start by: step]` evaluates to an infinite sequence of values, starting with *start*, and followed by `(+ start step)`, `(+ start (* 2 step))`, and so on.

The syntax `[start-expression <:]` is shorthand for `[start-expression by: 1]`.

bounded-range `::=` [*start-expression* [**by:** *step-expression*] *range-end*]
range-end `::=` **<:** *end-expression*
  `|` **<=:** *end-expression*
  `|` **>:** *end-expression*
  `|` **>=:** *end-expression*
  `|` **size:** *size-expression*

A bounded range takes an initial subsequence of the unbounded range specified by the *start-expression* and optional *step-expression*. The different *end-expression* variants provide different ways to specify the initial subsequence.

If `size: size` is specified, then the resulting range is the first *size* elements of unbounded sequence.

In the `<: end` or `<=: end` cases then the sequence counts up: The *step* must be positive, and defaults to 1. The resulting values are those *x* such that `(< x end)`, or `(<= x end)`, respectively.

In the `>: end` or `>=: end` cases then the sequence counts down: The *step* must be negative, and defaults to -1. The resulting values are those *x* such that `(> x end)`, or `(>= x end)`, respectively.

The *start-expression*, *step-expression*, and *size-expression* must evaluate to real numbers, not necessarily integers. For example: `[1 by: 0.5 <=: 3.0]` is `[1.0 1.5 2.0 2.5 3.0]`.

The two pseudo-ranges `[<:]` and `[>:]` are useful as array indexes. They mean “all of the valid indexes” of the array being indexed. For increasing index values use `[<:]`; for decreasing index values (i.e. reversing) use `[>:]`.

14.7 Streams - lazy lists

Streams, sometimes called lazy lists, are a sequential data structure containing elements computed only on demand. A stream is either null or is a pair with a stream in its cdr. Since elements of a stream are computed only when accessed, streams can be infinite. Once computed, the value of a stream element is cached in case it is needed again.

Note: These are not the same as Java 8 streams.

```
(require 'srfi-41)
(define fibs
  (stream-cons 1
    (stream-cons 1
      (stream-map +
        fibs
        (stream-cdr fibs)))))
(stream->list 8 fibs) ⇒ (1 1 2 3 5 8 13 21)
```

See the SRFI 41 specification (`http://srfi.schemers.org/srfi-41/srfi-41.html`) for details.

The Kawa implementations builds on Section 8.6 [promises], page 144. The `stream-null` value is a promise that evaluates to the empty list. The result of `stream-cons` is an eager immutable pair whose `car` and `cdr` properties return promises.

14.8 Multi-dimensional Arrays

Arrays are heterogeneous data structures that generaize vectors to multiple indexes or dimensions. Instead of a single integer index, there are multiple indexes: An index is a vector of integers; the length of a valid index sequence is the rank or the number of dimensions of an array.

Kawa multi-dimensional arrays follows the by SRFI-25 specification (`http://srfi.schemers.org/srfi-25/srfi-25.html`), with additions from Racket's math.array (`https://docs.racket-lang.org/math/array.html`) package and other sources.

An array whose rank is 1, and where the (single) lower bound is 0 is a sequence. Furthermore, if such an array is simple (not created by `share-array`) it will be implemented using a `<vector>`. Uniform vectors and strings are also arrays in Kawa.

A rank-0 array has a single value. It is essentially a box for that value. Functions that require arrays may treat non-arrays as a rank-0 array containing that value.

An array of rank 2 is frequently called a *matrix*.

Note that Kawa arrays are distinct from Java (native) arrays. The latter is a simpler one-dimensional vector-like data structure, which is used to implement Kawa arrays and vectors.

`array?` *obj* [Procedure]

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

14.8.1 Array shape

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 *i* that satisfies both `(<= b i)` and `(< i 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.

There is no separate data type for a shape. The canonical representation for a shape (a *canonical shape*) is a rank-2 array where the first index is the dimension (zero-based), and the second index is 0 or 1: Elements (*i* 0) and (*i* 1) are respectively the lower bound and upper bound of dimension *i*.

For convenience, the procedures that require a shape can accept a *shape-specifier*, as if converted by the procedure `->shape`. For example `(array-reshape array shape)` is equivalent to `(array-reshape array (->shape shape))`.

`->shape` *specifier* [Procedure]

Convert the shape specifier *specifier* to a canonical shape. The *specifier* must be either a canonical shape, or vector with one element for each dimension, as described below. We use as examples a 2*3 array with lower bounds 0 and a 3*4 array with lower bounds 1.

- A vector of simple integers. Each integer *e* is an upper bound, with a zero lower bound. Equivalent to the range `[0 <: e]`.

  A specifier for the first examples is `#(2 3)`, and the second is not expressible.
- A vector of lists of length 2. The first element of each list is the lower bound, and the second is the upper bound.

  Examples: `#((0 2) (0 3))` and `#((1 3) (1 4))`.
- A vector of simple Section 14.6 [Ranges], page 246, one for each dimension, all of who are bounded (finite), consist of integer values, and have a *step* of 1. Each range, which is usually written as `[b <: e]`, expresses the bounds of the corresponding dimension For the first example you can use `[[0 <: 2] [0 <=: 2]]`; for the second you can use `[[1 <: 3] [1 size: 4]]`.
- A vector consisting of a mix of integers, length-2 lists, and ranges.

  Examples: `#(2 (0 3))` and `['(1 3) [1 size: 4]]`.
- A canonical shape: A rank-2 array *S* whose own shape is `[r 2]`. For each dimension *k* (where `(<= k 0)` and `(< k r)`), the lower bound b_k is `(S k 0)`, and the upper bound e_k is `(S k 1)`.

  Examples: `#2a((0 2) (0 3))` and `#2a((1 3) (1 4))`.

`shape` *bound ...* [Procedure]

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)`.

`(shape @bounds)` is equivalent to: `(array [2 (/ (length bounds) 2)] @bounds)` `array-shape` *array* [Procedure]

Return the shape of *array* in the canonical (r 2) form. It is an error to attempt to modify the shape array.

`array-rank` *array* [Procedure]

Returns the number of dimensions of *array*.

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

Returns 2.

`array-start` *array k* [Procedure]

Returns the lower bound (inclusive) for the index along dimension *k*. This is most commonly 0.

`array-end` *array k* [Procedure]

Returns the upper bound for the index along dimension *k*. The bound is exclusive - i.e. the first integer higher than the last legal index.

`array-size` *array* [Procedure]

Return the total number of elements of *array*. This is the product of `(- (array-end` *`array k`*`) (array-start` *`array k`*`))` for every valid *k*.

14.8.2 Array types

`array` [Type]
`array`*`N`* [Type]
`array[`*`etype`*`]` [Type]
`array`*`N`*`[`*`etype`*`]` [Type]

The type `array` matches all array values. The type `array`*`N`*, where *N* is an integer matches array of rank *N*. For example `array2` matches rank-2 array - i.e. matrixes.

You can optionally specify the element type *etype*. This can be a primitive type. For example a `array2[double]` is a rank-2 array whose elements are `double` values.

14.8.3 Array literals and printing

An array literal starts with `#` followed by its rank, followed by a tag that describes the underlying vector (by default `a`), optionally followed by information about its shape, and finally followed by the cells, organized into dimensions using parentheses.

For example, `#2a((11 12 13) (21 22 23))` is a rank-2 array (a matrix) whose shape is `[2 3]` or equivalently `[[0 <: 2] [0 <: 3]]`. It is pretty-printed as:

```
#2a:2:3
111213

212223
```

array-literal `::=` *array-literal-header datum*
array-literal-header `::=` # *rank vectag array-bound **
array-bound `::=` [`@`*lower*]`:`*length* `|` `@`*lower*

vectag `::=` **a** | *uniform-tag*

The *vectag* specifies the type of the elements of the array.

Following the *vectag* you can optionally include information about the shape: For each dimension you can optionally specify the lower bounds (after the character `"@"`), followed by the length of the dimension (after the character `":"`). The shape information is required if a lower bound is non-zero, or any length is zero.

The *datum* contains the elements in a nested-list format: a rank-1 array (i.e. vector) uses a single list, a rank-2 array uses a list-of-lists, and so on. The elements are in lexicographic order.

A uniform u32 array of rank 2 with index ranges 2..3 and 3..4:

```
#2u32@2@3((1 2) (2 3))
```

This syntax follows Common Lisp with Guile extensions (`https://www.gnu.org/software/guile/manual/html_node/Array-Syntax.html`). (Note that Guile prints rank-0 arrays with an extra set of parentheses. Kawa follows Common Lisp in not doing so.)

When an array is printed with the `write` function, the result is an `array-literal`. Printing with `display` formats the array in a rectangular grid using the `format-array` procedure. (Note that `format-array` is only used when the output is in column 0, because Kawa has very limited support for printing rectangles.)

`format-array` *value* [*port*] [*element-format*] [Procedure]

Produce a nice "pretty" display for *value*, which is usually an array.

If *port* is an output port, the formatted output is written into that port. Otherwise, *port* must be a boolean (one of `#t` or `#f`). If the port is `#t`, output is to the `(current-output-port)`. If the port is `#f` or no port is specified, the output is returned as a string. If the port is specified and is `#t` or an output-port, the result of the `format-array` procedure is unspecified. (This convention matches that of the `format` procedure.)

The top line includes an `array-literal-header`. The lower bound are only printed if non-zero. The dimension lengths are printed if there is room, or if one of them is zero.

```
#|kawa:34|# (! arr (array [[1 <=: 2] [1 <=: 3]]
#|.....35|#   #2a((1 2) (3 4)) 9 #2a((3 4) (5 6))
#|.....36|#   [42 43] #2a:1:3((8 7 6)) #2a((90 91) (100 101))))
#|kawa:37|# arr
#2a@1:2@1:3
#2a        9#2a
12         34

34         56


#1a:2#2a:1:3#2a:2:2
4243876 90 91

                100101
```

If *element-format* is specified, it is a format string used for format each non-array:

```
#|kawa:38|# (format-array arr "~4,2f")
#2a@1:2@1:3
#2a:2:2              9.00#2a:2:2
1.002.00                  3.004.00

3.004.00                  5.006.00



#1a:2#2a:1:3#2a:2:2
42.0043.008.007.006.00 90.00 91.00

                             100.00101.00
```

If the rank is more than 2, then each “layer” is printed separated by double lines.

```
#|kawa:42|# (array-reshape [1 <=: 24] [3 2 4])
#3a:3:2:4
 1 2 3 4

 5 6 7 8

 9101112

13141516

17181920

21222324
```

14.8.4 Array construction

See also `array-reshape`

`array` *shape obj ...* [Procedure]

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*.

`make-array` *shape* [Procedure]
`make-array` *shape value...* [Procedure]

Returns a newly allocated array whose shape is given by *shape*. If *value* is provided, then each element is initialized to it. If there is more than one *value*, they are used in order, starting over when the *value*s are exhausted. If there is no *value*, the initial contents of each element is unspecified. (Actually, it is the `#!null`.) The array does not retain a reference to *shape*.

```
#|kawa:16|# (make-array [2 4] 1 2 3 4 5)
#2a:2:4
1234

5123
```

Compatibility: Guile has an incompatible `make-array` procedure.

`build-array` *shape getter* [*setter*] [Procedure]

Construct a "virtual array" of the given *shape*, which uses no storage for the elements. Instead, elements are calculated on-demand by calling *getter*, which takes a single argument, an index vector.

There is no caching or memoization.

```
#|kawa:1|# (build-array [[10 <: 12] 3]
#|....:2|#   (lambda (ind)
#|....:3|#     (let ((x (ind 0)) (y (ind 1)))
#|....:4|#       (- x y))))
#2a@10:2:3
10 98

11109
```

The resulting array is mutable if a *setter* is provided. The *setter* takes two arguments: An index vector, and the new value for the specified element. Below is a simple and space-efficient (but slow) implementation of sparse arrays: Most element have a default initial value, but you can override specific elements.

```
(define (make-sparse-array shape default-value)
  (let ((vals '())) ;; association list of (INDEX-VECTOR . VALUE)
    (build-array shape
                 (lambda (I)
                   (let ((v (assoc I vals)))
                     (if v (cdr v)
                         default-value)))
                 (lambda (I newval)
                   (let ((v (assoc I vals)))
                     (if v
                         (set-cdr! v newval)
                         (set! vals (cons (cons I newval) vals))))))))
```

`index-array` *shape* [Procedure]

Return a new immutable array of the specified *shape* where each element is the corresponding row-major index. Same as `(array-reshape [0 <: size] shape)` where *size* is the `array-size` of the resulting array.

```
#|kawa:1|# (index-array [[1 <: 3] [2 <: 6]])
```

```
#2a@1:2@2:4
0123

4567
```

14.8.5 Array indexing

If you “call” an array as it it were a function, it is equivalent to using `array-index-ref`, which is generalization of `array-ref`. For example, given a rank-2 array *arr* with integer indexes *i* and *j*, the following all get the element of *arr* at index `[i j]`.

```
(arr i j)
(array-index-ref arr i j)
(array-ref arr i j)
(array-ref arr [i j])
```

Using function-call notation or `array-index-ref` (but not plain `array-ref`) you can do generalized APL-style slicing and indirect indexing. See under `array-index-ref` for examples.

`array-ref` *array k ...* [Procedure]
`array-ref` *array index* [Procedure]

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 (a 0-based 1-dimensional array) containing *k* ....

```
(array-ref (array [2 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-index-ref` *array index ...* [Procedure]

Generalized APL-style array indexing, where each *index* can be either an array or an integer.

If each *index* is an integer, then the result is the same as `array-ref`.

Otherwise, the result is an immutable array whose rank is the sum of the ranks of each *index*. An integer is treated as rank-0 array.

If *marr* is the result of `(array-index-ref arr` M_1 M_2 `...)` then:

```
(marr i_11 i_12 ... i_21 i_22 ...)
```

is defined as:

```
(arr (M_1 i_11 i_12 ...) (M_2 i_21 i_22 ...) ...)
```

Each M_k gets as many indexes as its rank. If M_k is an integer, then it we use it directly without any indexing.

Here are some examples, starting with simple indexing.

```
#|kawa:1|# (define arr (array #2a((1 4) (0 4))
#|.....2|#                    10 11 12 13 20 21 22 23 30 31 32 33))
#|kawa:3|# arr
#2a@1:3:4
10111213

20212223

30313233

#|kawa:4|# (arr 2 3)
23
```

If one index is a vector and the rest are scalar integers, then the result is a vector:

```
#|kawa:5|# (arr 2 [3 1])
#(23 21)
```

You can select a “sub-matrix” when all indexes are vectors:

```
#|kawa:6|# (arr [2 1] [3 1 3])
#2a:2:3
232123

131113
```

Using ranges for index vectors selects a rectangular sub-matrix.

```
#|kawa:7|# (arr [1 <: 3] [1 <: 4])
#2a:2:3
111213

212223
```

You can add new dimensions:

```
#|kawa:8|# (arr [2 1] #2a((3 1) (3 2)))
#3a
2321

2322

1311

1312
```

The pseudo-range `[<:]` can be used to select all the indexes along a dimension. To select row 2 (1-origin):

```
#|kawa:9|# (arr 2 [<:])
#(20 21 22 23)
```

To reverse the order use `[>:]`:

```
#|kawa:10|# (arr 2 [>:])
#(23 22 21 20)
```

To select column 3:

```
#|kawa:11|# (arr [<:] 3)
#(13 23 33)
```

If you actually want a column matrix (i.e. with shape `[3 1]` you can place the index in a single-element vector:

```
#|kawa:12|# (arr [<:] [3])
#2a
13

23

33
```

To expand that column to 5 colums you can repeat the column index:

```
#|kawa:13|# [3 by: 0 size: 5]
#(3 3 3 3 3)
#|kawa:14|# (arr [<:] [3 by: 0 size: 5])
#2a:3:5
1313131313

2323232323

3333333333
```

14.8.6 Modifying arrays

You can use `set!` to modify one or multiple elements. To modify a single element:

```
(set! (arr index ...) new-value)
```

is equivalent to

```
(array-set! arr index ... new-value)
```

You can set a slice (or all of the elements). In that case:

```
(set! (arr index ...) new-array)
```

is equivalent to:

```
(array-copy! (array-index-share arr index ...) new-array)
```

array-set! *array k ... obj* [Procedure]
array-set! *array index obj* [Procedure]

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"`.

Compatibility: SRFI-47, Guile and Scheme-48 have `array-set!` with a different argument order.

`array-copy!` *dst src* [Procedure]

Compatibility: Guile has a `array-copy!` with the reversed argument order.

`array-fill!` *array value* [Procedure]

Set all the values *array* to *value*.

14.8.7 Transformations and views

A view or transform of an array is an array a_2 whose elements come from some other array a_1, given some transform function T that maps a_2 indexes to a_1 indexes. Specifically `(array-ref` a_2 *indexes*`)` is `(array-ref` a_1 `(`*T indexes*`))`. Modifying a_2 causes a_1 to be modified; modifying a_1 may modify a_2 (depending on the transform function). The shape of a_2 is in generally different than that of a_1.

`array-transform` *array shape transform* [Procedure]

This is a general mechanism for creating a view. The result is a new array with the given *shape*. Accessing this new array is implemented by calling the *transform* function on the index vector, which must return a new index vector valid for indexing the original *array*. Here is an example (using the same `arr` as in the `array-index-ref` example):

```
#|kawa:1|# (define arr (array #2a((1 4) (0 4))
#|.....2|#                      10 11 12 13 20 21 22 23 30 31 32 33))
#|kawa:14|# (array-transform arr #2a((0 3) (1 3) (0 2))
#|.....15|#   (lambda (ix) (let ((i (ix 0)) (j (ix 1)) (k (ix 2)))
#|.....16|#                  [(+ i 1)
#|.....17|#                   (+ (* 2 (- j 1)) k)])))
#3a:3@1:2:2
1011

1213

2021

2223

3031

3233
```

The `array-transform` is generalization of `share-array`, in that it does not require the *transform* to be affine. Also note the different calling conversions for the *tranform*: `array-transform` takes a single argument (a vector of indexes), and returns a single result (a vector of indexes); `share-array` takes one argument for each index, and returns one value for each index. The difference is historical.

`array-index-share` *array index ...* [Procedure]

This does the same generalized APL-style indexing as `array-index-ref`. However, the resulting array is a modifiable view into the argument *array*.

`array-reshape` *array shape* [Procedure]

Creates a new array *narray* of the given *shape*, such that `(array->vector` *array*`)` and `(array->vector` *narray*`)` are equivalent. I.e. the *i*'th element in row-major-order of *narray* is the *i*'th element in row-major-order of *array*. Hence `(array-size` *narray*`)` (as specied from the *shape*) must be equal to `(array-size` *array*`)`. The resulting *narray* is a view such that modifying *array* also modifies *narray* and vice versa.

`share-array` *array shape proc* [Procedure]

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.

Here is an example where the *array* is a uniform vector:

```
(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)))
        ⇒  #2f64((1.0 2.0 3.0) (4.0 5.0 6.0))
```

`array-flatten` *array* [Procedure]
`array->vector` *array* [Procedure]

Return a vector consisting of the elements of the *array* in row-major-order.

The result of `array-flatten` is fresh (mutable) copy, not a view. The result of `array->vector` is a view: If *array* is mutable, then modifying *array* changes the flattened result and vice versa.

If *array* is "simple", `array->vector` returns the original vector. Specifically, if *vec* is a vector then:

```
(eq? vec (array->vector (array-reshape vec shape)))
```

14.8.8 Miscellaneous

`format-array` *value* [*element-format*] [Procedure]

14.9 Hash tables

A *hashtable* is a data structure that associates keys with values. The hashtable has no intrinsic order for the (key, value) associations it contains, and supports in-place modification as the primary means of setting the contents of a hash table. Any object can be used as a key, provided a *hash function* and a suitable *equivalence function* is available. A hash function is a procedure that maps keys to exact integer objects.

The hashtable provides key lookup and destructive update in amortised constant time, provided that a good hash function is used. A hash function *h* is acceptable for an equivalence predicate *e* iff `(e obj1 obj2)` implies `(= (h obj1) (h obj2))`. A hash function *h* is good for a equivalence predicate *e* if it distributes the resulting hash values for non-equal objects (by *e*) as uniformly as possible over the range of hash values, especially in the case when some (non-equal) objects resemble each other by e.g. having common subsequences. This definition is vague but should be enough to assert that e.g. a constant function is not a good hash function.

Kawa provides two complete sets of functions for hashtables:

- The functions specified by R6RS have names starting with `hashtable-`
- The functions specified by the older SRFI-69 (`http://srfi.schemers.org/srfi-69/srfi-69.html`) specifiation have names starting with `hash-table-`

Both interfaces use the same underlying datatype, so it is possible to mix and match from both sets. That datatype implements `java.util.Map`. Freshly-written code should probably use the R6RS functions.

14.9.1 R6RS hash tables

To use these hash table functions in your Kawa program you must first:

```
(import (rnrs hashtables))
```

This section uses the *hashtable* parameter name for arguments that must be hashtables, and the *key* parameter name for arguments that must be hashtable keys.

`make-eq-hashtable` [Procedure]
`make-eq-hashtable` *`k`* [Procedure]

Return a newly allocated mutable hashtable that accepts arbitrary objects as keys, and compares those keys with `eq?`. If an argument is given, the initial capacity of the hashtable is set to approximately *k* elements.

`make-eqv-hashtable` [Procedure]
`make-eqv-hashtable` *`k`* [Procedure]

Return a newly allocated mutable hashtable that accepts arbitrary objects as keys, and compares those keys with `eqv?`. If an argument is given, the initial capacity of the hashtable is set to approximately *k* elements.

`make-hashtable` *`hash-function equiv`* [Procedure]
`make-hashtable` *`hash-function equiv k`* [Procedure]

hash-function and *equiv* must be procedures. *hash-function* should accept a key as an argument and should return a non–negative exact integer object. *equiv* should accept two keys as arguments and return a single value. Neither procedure should mutate the hashtable returned by `make-hashtable`.

The `make-hashtable` procedure returns a newly allocated mutable hashtable using *hash-function* as the hash function and *equiv* as the equivalence function used to compare keys. If a third argument is given, the initial capacity of the hashtable is set to approximately *k* elements.

Both *hash-function* and *equiv* should behave like pure functions on the domain of keys. For example, the `string-hash` and `string=?` procedures are permissible only if all keys are strings and the contents of those strings are never changed so long as any of them continues to serve as a key in the hashtable. Furthermore, any pair of keys for which *equiv* returns true should be hashed to the same exact integer objects by *hash-function*.

> *Note:* Hashtables are allowed to cache the results of calling the hash function and equivalence function, so programs cannot rely on the hash function being called for every lookup or update. Furthermore any hashtable operation may call the hash function more than once.

14.9.1.1 Procedures

`hashtable?` *`obj`* [Procedure]

Return `#t` if *obj* is a hashtable, `#f` otherwise.

`hashtable-size` *`hashtable`* [Procedure]

Return the number of keys contained in *hashtable* as an exact integer object.

`hashtable-ref` *`hashtable key default`* [Procedure]

Return the value in *hashtable* associated with *key*. If *hashtable* does not contain an association for *key*, *default* is returned.

`hashtable-set!` *`hashtable key obj`* [Procedure]

Change *hashtable* to associate *key* with *obj*, adding a new association or replacing any existing association for *key*, and returns unspecified values.

`hashtable-delete!` *`hashtable key`* [Procedure]

Remove any association for *key* within *hashtable* and returns unspecified values.

`hashtable-contains?` *`hashtable key`* [Procedure]

Return `#t` if *hashtable* contains an association for *key*, `#f` otherwise.

`hashtable-update!` *`hashtable key proc default`* [Procedure]

proc should accept one argument, should return a single value, and should not mutate *hashtable*.

The `hashtable-update!` procedure applies *proc* to the value in *hashtable* associated with *key*, or to *default* if *hashtable* does not contain an association for *key*. The *hashtable* is then changed to associate *key* with the value returned by *proc*.

The behavior of `hashtable-update!` is equivalent to the following code, but is may be (and is in Kawa) implemented more efficiently in cases where the implementation can avoid multiple lookups of the same key:

```
(hashtable-set!
  hashtable key
  (proc (hashtable-ref
         hashtable key default)))
```

`hashtable-copy` *`hashtable`* [Procedure]
`hashtable-copy` *`hashtable mutable`* [Procedure]

Return a copy of *hashtable*. If the *mutable* argument is provided and is true, the returned hashtable is mutable; otherwise it is immutable.

`hashtable-clear!` *`hashtable`* [Procedure]
`hashtable-clear!` *`hashtable k`* [Procedure]

Remove all associations from *hashtable* and returns unspecified values.

If a second argument is given, the current capacity of the hashtable is reset to approximately *k* elements.

`hashtable-keys` *`hashtable`* [Procedure]

Return a vector of all keys in *hashtable*. The order of the vector is unspecified.

`hashtable-entries` *`hashtable`* [Procedure]

Return two values, a vector of the keys in *hashtable*, and a vector of the corresponding values.

Example:

```
(let ((h (make-eqv-hashtable)))
  (hashtable-set! h 1 'one)
  (hashtable-set! h 2 'two)
  (hashtable-set! h 3 'three)
  (hashtable-entries h))
⇒ #(1 2 3) #(one two three) ; two return values
```

the order of the entries in the result vectors is not known.

14.9.1.2 Inspection

`hashtable-equivalence-function` *hashtable* [Procedure]

Return the equivalence function used by *hashtable* to compare keys. For hashtables created with `make-eq-hashtable` and `make-eqv-hashtable`, returns `eq?` and `eqv?` respectively.

`hashtable-hash-function` *hashtable* [Procedure]

Return the hash function used by *hashtable*. For hashtables created by `make-eq-hashtable` or `make-eqv-hashtable`, `#f` is returned.

`hashtable-mutable?` *hashtable* [Procedure]

Return `#t` if *hashtable* is mutable, otherwise `#f`.

14.9.1.3 Hash functions

The `equal-hash`, `string-hash`, and `string-ci-hash` procedures of this section are acceptable as the hash functions of a hashtable only if the keys on which they are called are not mutated while they remain in use as keys in the hashtable.

`equal-hash` *obj* [Procedure]

Return an integer hash value for *obj*, based on its structure and current contents. This hash function is suitable for use with `equal?` as an equivalence function.

> *Note:* Like `equal?`, the `equal-hash` procedure must always terminate, even if its arguments contain cycles.

`string-hash` *string* [Procedure]

Return an integer hash value for *string*, based on its current contents. This hash function is suitable for use with `string=?` as an equivalence function.

`string-ci-hash` *string* [Procedure]

Return an integer hash value for *string* based on its current contents, ignoring case. This hash function is suitable for use with `string-ci=?` as an equivalence function.

`symbol-hash` *symbol* [Procedure]

Return an integer hash value for *symbol*.

14.9.2 SRFI-69 hash tables

To use these hash table functions in your Kawa program you must first:

```
(require 'srfi-69)
```

or

```
(require 'hash-table)
```

or

```
(import (srfi 69 basic-hash-tables))
```

14.9.2.1 Type constructors and predicate

`make-hash-table` [*equal?* [*hash* [*size-hint*]]] ↦ *hash-table* [Procedure]

Create a new hash table with no associations. The *equal?* parameter is a predicate that should accept two keys and return a boolean telling whether they denote the same key value; it defaults to the `equal?` function.

The *hash* parameter is a hash function, and defaults to an appropriate hash function for the given *equal?* predicate (see the Hashing section). However, an acceptable default is not guaranteed to be given for any equivalence predicate coarser than `equal?`, except for `string-ci=?`. (The function `hash` is acceptable for `equal?`, so if you use coarser equivalence than `equal?` other than `string-ci=?`, you must always provide the function hash yourself.) (An equivalence predicate *c1* is coarser than a equivalence predicate *c2* iff there exist values *x* and *y* such that `(and (`*c1 x y*`) (not (`*c2 x y*`))).`)

The *size-hint* parameter can be used to suggested an approriate initial size. This option is not part of the SRFI-69 specification (though it is handled by the reference implementation), so specifying that option might be unportable.

`hash-table?` *obj* ↦ *boolean* [Procedure]

A predicate to test whether a given object *obj* is a hash table.

`alist->hash-table` *alist* [*equal?* [*hash* [*size-hint*]]] ↦ *hash-table* [Procedure]

Takes an association list *alist* and creates a hash table *hash-table* which maps the `car` of every element in *alist* to the `cdr` of corresponding elements in *alist*. The *equal?*, *hash*, and *size-hint* parameters are interpreted as in `make-hash-table`. If some key occurs multiple times in *alist*, the value in the first association will take precedence over later ones. (Note: the choice of using `cdr` (instead of `cadr`) for values tries to strike balance between the two approaches: using *cadr* would render this procedure unusable for `cdr` alists, but not vice versa.)

14.9.2.2 Reflective queries

`hash-table-equivalence-function` *hash-table* [Procedure]

Returns the equivalence predicate used for keys of *hash-table*.

`hash-table-hash-function` *hash-table* [Procedure]

Returns the hash function used for keys of *hash-table*.

14.9.2.3 Dealing with single elements

`hash-table-ref` *hash-table key* [*thunk*] ↦ *value* [Procedure]

This procedure returns the value associated to *key* in *hash-table*. If no value is associated to *key* and *thunk* is given, it is called with no arguments and its value is returned; if *thunk* is not given, an error is signalled. Given a good hash function, this operation should have an (amortised) complexity of O(1) with respect to the number of associations in *hash-table*.

`hash-table-ref/default` *hash-table key default* ↦ *value* [Procedure]

Evaluates to the same value as `(hash-table-ref` ***`hash-table key`*** `(lambda ()` ***`default`***`))`. Given a good hash function, this operation should have an (amortised) complexity of O(1) with respect to the number of associations in hash-table.

`hash-table-set!` *hash-table key value* ↦ *void* [Procedure]

This procedure sets the value associated to *key* in *hash-table*. The previous association (if any) is removed. Given a good hash function, this operation should have an (amortised) complexity of O(1) with respect to the number of associations in hash-table.

`hash-table-delete!` *hash-table key* ↦ *void* [Procedure]

This procedure removes any association to *key* in *hash-table*. It is not an error if no association for the *key* exists; in this case, nothing is done. Given a good hash function, this operation should have an (amortised) complexity of O(1) with respect to the number of associations in hash-table.

`hash-table-exists?` *hash-table key* ↦ *boolean* [Procedure]

This predicate tells whether there is any association to *key* in *hash-table*. Given a good hash function, this operation should have an (amortised) complexity of O(1) with respect to the number of associations in hash-table.

`hash-table-update!` *hash-table key function* [*thunk*] ↦ *void* [Procedure]

Semantically equivalent to, but may be implemented more efficiently than, the following code:

```
(hash-table-set! hash-table key
                 (function (hash-table-ref hash-table key thunk)))
```

`hash-table-update!/default` *hash-table key function default* ↦ *void* [Procedure]

Behaves as if it evaluates to `(hash-table-update!` ***`hash-table key function`*** `(lambda ()` ***`default`***`))`.

14.9.2.4 Dealing with the whole contents

`hash-table-size` *hash-table* ↦ *integer* [Procedure]

Returns the number of associations in *hash-table*. This operation takes constant time.

`hash-table-keys` *hash-table* ↦ *list* [Procedure]

Returns a list of keys in *hash-table*. The order of the keys is unspecified.

`hash-table-values` *hash-table* ↦ *list* [Procedure]

Returns a list of values in *hash-table*. The order of the values is unspecified, and is not guaranteed to match the order of keys in the result of `hash-table-keys`.

`hash-table-walk` *hash-table proc* ↦ *void* [Procedure]

proc should be a function taking two arguments, a key and a value. This procedure calls *proc* for each association in *hash-table*, giving the key of the association as key and the value of the association as value. The results of *proc* are discarded. The order in which *proc* is called for the different associations is unspecified.

`hash-table-fold` *hash-table f init-value* ↦ *final-value* [Procedure]

This procedure calls *f* for every association in *hash-table* with three arguments: the key of the association key, the value of the association value, and an accumulated value, *val*. The *val* is *init-value* for the first invocation of *f*, and for subsequent invocations of *f*, the return value of the previous invocation of *f*. The value *final-value* returned by `hash-table-fold` is the return value of the last invocation of *f*. The order in which *f* is called for different associations is unspecified.

`hash-table->alist` *hash-table* ↦ *alist* [Procedure]

Returns an association list such that the `car` of each element in *alist* is a key in *hash-table* and the corresponding `cdr` of each element in *alist* is the value associated to the key in *hash-table*. The order of the elements is unspecified.

The following should always produce a hash table with the same mappings as a hash table *h*:

```
(alist->hash-table (hash-table->alist h)
                   (hash-table-equivalence-function h)
                   (hash-table-hash-function h))
```

`hash-table-copy` *hash-table* ↦ *hash-table* [Procedure]

Returns a new hash table with the same equivalence predicate, hash function and mappings as in *hash-table*.

`hash-table-merge!` *hash-table1 hash-table2* ↦ *hash-table* [Procedure]

Adds all mappings in *hash-table2* into *hash-table1* and returns the resulting hash table. This function may modify *hash-table1* destructively.

14.9.2.5 Hash functions

The Kawa implementation always calls these hash functions with a single parameter, and expects the result to be within the entire (32-bit signed) `int` range, for compatibility with standard `hashCode` methods.

`hash` *object* [*bound*] ↦ *integer* [Procedure]

Produces a hash value for object in the range from 0 (inclusive) tp to *bound* (exclusive).

If *bound* is not given, the Kawa implementation returns a value within the range `(- (expt 2 32))` (inclusive) to `(- (expt 2 32) 1)` (inclusive). It does this by calling the standard `hashCode` method, and returning the result as is. (If the *object* is the Java `null` value, 0 is returned.) This hash function is acceptable for `equal?`.

`string-hash` *string* [*bound*] ↦ *integer* [Procedure]

The same as `hash`, except that the argument string must be a string. (The Kawa implementation returns the same as the `hash` function.)

`string-ci-hash` *string* [*bound*] ↦ *integer* [Procedure]

The same as `string-hash`, except that the case of characters in string does not affect the hash value produced. (The Kawa implementation returns the same the `hash` function applied to the lower-cased *string*.)

`hash-by-identity` *object* [*bound*] ↦ *integer* [Procedure]

The same as `hash`, except that this function is only guaranteed to be acceptable for `eq?`. Kawa uses the `identityHashCode` method of `java.lang.System`.

15 Eval and Environments

`environment` *list** [Procedure]

This procedure returns a specifier for the environment that results by starting with an empty environment and then importing each *list*, considered as an *import-set*, into it. The bindings of the environment represented by the specifier are immutable, as is the environment itself. See the `eval` function for examples.

`null-environment` *version* [Procedure]

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* [Procedure]

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` [Procedure]

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

`environment-bound?` *environment symbol* [Procedure]

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

`environment-fold` *environment proc init* [Procedure]

Call *proc* for each key in the *environment*, which may be any argument to `eval`, such as `(interaction-environment)` or a call to the `environment` procedure. The *proc* is called with two arguments: The binding's key, and an accumulator value. The *init* is the initial accumulator value; the result returned by *proc* is used for subsequent accumulator values. The value returned by `environment-fold` is the final acumulator value.

A key is normally a symbol, but can also be a `KeyPair` object (a pair of a symbol and a property symbol used to implement Common Lisp-style property lists).

```
(environment-fold (environment '(scheme write)) cons '())
  ⇒ (display write-shared write write-simple)
```

To get all the predefined bindings use `(environment '(kawa base))`.

`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*] [Procedure]

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.

`eval` *expression* [*environment*] [Procedure]

This procedure evaluates *expression* in the environment indicated by *environment*. The default for *environment* is the result of `(interaction-environment)`.

```
(eval '(* 7 3) (environment '(scheme base)))
           ⇒ 21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))
           ⇒ 20

(eval '(define foo 32) (environment '(scheme base)))
           ⇒ error is signaled
```

`load` *path* [*environment*] [Procedure]
`load-relative` *path* [*environment*] [Procedure]

The *path* can be an (absolute) URL or a filename of a source file, which is read and evaluated line-by-line. The *path* can also be a fully-qualified class name. (Mostly acts like the `-f` command-line option, but with different error handling.) Since `load` is a run-time function it doesn't know about the enclosing lexical environment, and the latter can't know about definitions introduced by `load`. For those reasons it is highly recommended that you use instead use `[require], page 312` or `[include], page 119`.

Evaluation is done in the specified *environment*, which defauls to result of `(interaction-environment)`.

The `load-relative` procedure is like `load`, except that *path* is a URI that is relative to the context's current base URI.

15.1 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* [Procedure]

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.

`define-private-alias` *variable lvalue* [Syntax]

Same as `define-alias`, but the *variable* is local to the current module.

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.

You can use `define-alias` to define a shorter type synonym, similar to Java's `import TypeName` (single-type-import) declaration:

```
(define-alias StrBuf java.lang.StringBuffer)
```

15.2 Parameter objects

A parameter object is a procedure that is bound to a location, and may optionally have a conversion procedure. The procedure accepts zero or one argument. When the procedure is called with zero arguments, the content of the location is returned. On a call with one argument the content of the location is updated with the result of applying the parameter object's conversion procedure to the argument.

Parameter objects are created with the `make-parameter` procedure which takes one or two arguments. The second argument is a one argument conversion procedure. If only one argument is passed to make-parameter the identity function is used as a conversion procedure. A new location is created and asociated with the parameter object. The initial content of the location is the result of applying the conversion procedure to the first argument of make-parameter.

Note that the conversion procedure can be used for guaranteeing the type of the parameter object's binding and/or to perform some conversion of the value.

The `parameterize` special form, when given a parameter object and a value, binds the parameter object to a new location for the dynamic extent of its body. The initial content of the location is the result of applying the parameter object's conversion procedure to the value. The `parameterize` special form behaves analogously to `let` when binding more than one parameter object (that is the order of evaluation is unspecified and the new bindings are only visible in the body of the parameterize special form).

When a new thread is created using `future` or `runnable` then the child thread inherits initial values from its parent. Once the child is running, changing the value in the child does not affect the value in the parent or vice versa. (In the past this was not the case: The child would share a location with the parent except within a `parameterize`. This was changed to avoid unsafe and inefficient coupling between threads.)

Note that `parameterize` and `fluid-let` have similar binding and sharing behavior. The difference is that `fluid-let` modifies locations accessed by name, while `make-parameter` and `parameterize` create anonymous locations accessed by calling a parameter procedure.

The R5RS procedures `current-input-port` and `current-output-port` are parameter objects.

`make-parameter` *init* [*converter*] [Procedure]

Returns a new parameter object which is bound in the global dynamic environment to a location containing the value returned by the call `(`*`converter init`*`)`. If the conversion procedure converter is not specified the identity function is used instead.

The parameter object is a procedure which accepts zero or one argument. When it is called with no argument, the content of the location bound to this parameter object in the current dynamic environment is returned. When it is called with one argument, the content of the location is set to the result of the call `(converter arg)`, where *arg* is the argument passed to the parameter object, and an unspecified value is returned.

```
(define radix
  (make-parameter 10))

(define write-shared
  (make-parameter
    #f
    (lambda (x)
      (if (boolean? x)
          x
          (error "only booleans are accepted by write-shared")))))

(radix)           ⇒  10
(radix 2)
(radix)           ⇒  2
(write-shared 0)  gives an error

(define prompt
  (make-parameter
    123
    (lambda (x)
      (if (string? x)
          x
          (with-output-to-string (lambda () (write x)))))))

(prompt)       ⇒  "123"
(prompt ">")
(prompt)       ⇒  ">"
```

`parameterize` ((*expr1 expr2*) ...) *body* [Syntax]

The expressions *expr1* and *expr2* are evaluated in an unspecified order. The value of the *expr1* expressions must be parameter objects. For each *expr1* expression and in an unspecified order, the local dynamic environment is extended with a binding of the parameter object *expr1* to a new location whose content is the result of the call `(converter val)`, where *val* is the value of *expr2* and *converter* is the conversion procedure of the parameter object. The resulting dynamic environment is then used for the evaluation of *body* (which refers to the R5RS grammar nonterminal of that name). The result(s) of the parameterize form are the result(s) of the *body*.

```
(radix)                                          ⇒  2
(parameterize ((radix 16)) (radix))              ⇒  16
(radix)                                          ⇒  2

(define (f n) (number->string n (radix)))

(f 10)                                           ⇒  "1010"
```

```
(parameterize ((radix 8)) (f 10))                     ⇒  "12"
(parameterize ((radix 8) (prompt (f 10))) (prompt))  ⇒  "1010"
```

16 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.

Note that Kawa will normally assume that a procedure defined with the procedure-defining variant of `define` is constant, and so it might be inlined:

```
(define (ff x) (list x x))
(trace ff) ;; probably won't work
(ff 3)     ;; not traced
```

It works if you specify the `--no-inline` flag to Kawa. Alternatively, you can use the variable-defining variant of `define`:

```
#|kawa:1|# (define ff (lambda (x) name: 'ff (list x x)))
#|kawa:2|# (trace ff) ;; works
#|kawa:3|# (ff 3)
call to ff (3)
return from ff => (3 3)
(3 3)
```

Note the use of the `name:` procedure property to give the anonymous `lambda` a name.

`untrace` *procedure* [Syntax]

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

`disassemble` *procedure* [Procedure]

Returns a string representation of the disassembled bytecode for *procedure*, when known.

17 Input, output, and file handling

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

A programmable reader.

A powerful pretty-printer.

17.1 Named output formats

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 just a script name). (It also affects applications compiled with the `--main` flag.) It currently affects 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`
`xhtml`
`html`
: Values are printed in XML, XHTML, or HTML format. This is discussed in more detail in Section 20.1 [Formatting XML], page 336.

`cgi`
: The output should 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-type` depending on the following value.

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

17.2 Paths - file name, URLs, and URIs

A *Path* is the name of a file or some other *resource*. The path mechanism provides a layer of abstraction, so you can use the same functions on either a filename or a URL/URI. Functions that in standard Scheme take a filename have been generalized to take a path or a path string, as if using the `path` function below. For example:

```
(open-input-file "http://www.gnu.org/index.html")
```

```
(open-input-file (URI "ftp://ftp.gnu.org/README"))
```

`path` [Type]

A general path, which can be a `filename` or a `URI`. It can be either a `filename` or a `URI`. Represented using the abstract Java class `gnu.kawa.io.Path`.

Coercing a value to a `Path` is equivalent to calling the `path` constructor documented below.

`path` *arg* [Constructor]

Coerces the *arg* to a `path`. If *arg* is already a `path`, it is returned unchanged. If *arg* is a `java.net.URI`, or a `java.net.URL` then a `URI` value is returned. If *arg* is a `java.io.File`, a `filepath` value is returned. Otherwise, *arg* can be a string. A `URI` value is returned if the string starts with a URI scheme (such as `"http:"`), and a `filepath` value is returned otherwise.

`path?` *arg* [Predicate]

True if *arg* is a `path` - i.e. an instance of a `gnu.kawa.io.Path`.

`current-path` [*new-value*] [Procedure]

With no arguments, returns the default directory of the current thread as a `path`. This is used as the base directory for relative pathnames. The initial value is that of the `user.dir` property as returned by `(java.lang.System:getProperty "user.dir")`.

If a *new-value* argument is given, sets the default directory:

```
(current-path "/opt/myApp/")
```

A string value is automatically converted to a `path`, normally a `filepath`.

Alternatively, you can change the default using a setter:

```
(set! (current-path) "/opt/myApp/")
```

Since `current-path` is a Section 15.2 [Parameter objects], page 268, you can locally change the value using [parameterize-syntax], page 269.

`filepath` [Type]

The name of a local file. Represented using the Java class `gnu.kawa.io.FilePath`, which is a wrapper around `java.io.File`.

`filepath?` *arg* [Predicate]

True if *arg* is a `filepath` - i.e. an instance of a `gnu.kawa.io.FilePath`.

`URI` [Type]

A Uniform Resource Indicator, which is a generalization of the more familiar URL. The general format is specified by RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax (`http://www.ietf.org/rfc/rfc2396.txt`). Represented using the Java class `gnu.kawa.io.URIPath`, which is a wrapper around `java.net.URI`. A URI can be a URL, or it be a relative URI.

`URI?` *arg* [Predicate]

True if *arg* is a `URI` - i.e. an instance of a `gnu.kawa.io.URIPath`.

`URL` [Type]

A Uniform Resource Locator - a subtype of `URI`. Represented using the Java class `gnu.kawa.io.URLPath`, which is a wrapper around a `java.net.URL`, in addition to extending `gnu.kawa.io.URIPath`.

17.2.1 Extracting Path components

`path-scheme` *arg* [Procedure]

Returns the "URI scheme" of *arg* (coerced to a `path`) if it is defined, or `#f` otherwise. The URI scheme of a `filepath` is `"file"` if the `filepath` is absolute, and `#f` otherwise.

```
(path-scheme "http://gnu.org/") ⇒ "http"
```

`path-authority` *arg* [Procedure]

Returns the authority part of *arg* (coerced to a `path`) if it is defined, or `#f` otherwise. The "authority" is usually the hostname, but may also include user-info or a port-number.

```
(path-authority "http://me@localhost:8000/home") ⇒ "me@localhost:8000"
```

`path-host` *arg* [Procedure]

Returns the name name part of *arg* (coerced to a `path`) if it is defined, or `#f` otherwise.

```
(path-host "http://me@localhost:8000/home") ⇒ "localhost"
```

`path-user-info` *arg* [Procedure]

Returns the "user info" of *arg* (coerced to a `path`) if it is specified, or `#f` otherwise.

```
(path-host "http://me@localhost:8000/home") ⇒ "me"
```

`path-port` *arg* [Procedure]

Returns the port number of *arg* (coerced to a `path`) if it is specified, or `-1` otherwise. Even if there is a default port associated with a URI scheme (such as 80 for `http`), the value -1 is returned unless the port number is *explictly* specified.

```
(path-host "http://me@localhost:8000/home") ⇒ 8000
(path-host "http://me@localhost/home") ⇒ -1
```

`path-file` *arg* [Procedure]

Returns the "path component" of the *arg* (coerced to a `path`). (The name `path-path` might be more logical, but it is obviously a bit awkward.) The path component of a file name is the file name itself. For a URI, it is the main hierarchical part of the URI, without schema, authority, query, or fragment.

```
(path-file "http://gnu.org/home/me.html?add-bug#body") ⇒ "/home/me.html"
```

`path-directory` *arg* [Procedure]

If *arg* (coerced to a `path`) is directory, return *arg*; otherwise return the "parent" path, without the final component.

```
(path-directory "http://gnu.org/home/me/index.html#body")
  ⇒ (path "http://gnu.org/home/me/")
(path-directory "http://gnu.org/home/me/")
  ⇒ (path "http://gnu.org/home/me/")
```

`(path-directory "./dir")` ⇒ `(path "./dir")` if `dir` is a directory, and `(path ".")` otherwise.

`path-parent` *arg* [Procedure]
Returns the "parent directory" of *arg* (coerced to a `path`). If *arg* is not a directory, same as `path-directory` *arg*.

```
(path-parent "a/b/c") ⇒ (path "a/b")
(path-parent "file:/a/b/c") ⇒ (path "file:/a/b/c")
(path-parent "file:/a/b/c/") ⇒ (path "file:/a/b/")
```

`path-last` *arg* [Procedure]
The last component of path component of *arg* (coerced to a `path`). Returns a substring of `(path-file` *arg*`)`. If that string ends with '`/`' or the path separator, that last character is ignored. Returns the tail of the path-string, following the last (non-final) '`/`' or path separator.

```
(path-last "http:/a/b/c") ⇒ "c"
(path-last "http:/a/b/c/") ⇒ "c"
(path-last "a/b/c") ⇒ "c"
```

`path-extension` *arg* [Procedure]
Returns the "extension" of the *arg* (coerced to a `path`).

```
(path-extension "http://gnu.org/home/me.html?add-bug#body") ⇒ "html"
(path-extension "/home/.init") ⇒ #f
```

`path-query` *arg* [Procedure]
Returns the query part of *arg* (coerced to a `path`) if it is defined, or `#f` otherwise. The query part of a URI is the part after '`?`'.

```
(path-query "http://gnu.org/home?add-bug") ⇒ "add-bug"
```

`path-fragment` *arg* [Procedure]
Returns the fragment part of *arg* (coerced to a `path`) if it is defined, or `#f` otherwise. The fragment of a URI is the part of after '`#`'.

```
(path-query "http://gnu.org/home#top") ⇒ "top"
```

`resolve-uri` *uri base* [Procedure]
Returns a *uri* unchanged if it is an absolute URI. Otherwise resolves it against a base URI *base*, which is normally (though not always) absolute.

This uses the algorithm specifyed by RFC-3986 (assuming *base* is absolute), unlike the obsolete RFC-2396 algorithm used by `java.net.URI.resolve`.

17.3 File System Interface

`file-exists?` *filename* [Procedure]
Returns true iff the file named *filename* actually exists. This function is defined on arbitrary `path` values: for URI values we open a `URLConnection` and invoke `getLastModified()`.

`file-directory?` *filename* [Procedure]
: Returns true iff the file named *filename* actually exists and is a directory. This function is defined on arbitrary `path` values; the default implementation for non-file objects is to return `#t` iff the path string ends with the character '`/`'.

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

`file-writable?` *filename* [Procedure]
: 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* [Procedure]
: Delete the file named *filename*. On failure, throws an exception.

`rename-file` *oldname newname* [Procedure]
: Renames the file named *oldname* to *newname*.

`copy-file` *oldname newname-from path-to* [Procedure]
: Copy the file named *oldname* to *newname*. The return value is unspecified.

`create-directory` *dirname* [Procedure]
: 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` [Procedure]
: Return the name of the default directory for temporary files.

`make-temporary-file` [*format*] [Procedure]
: 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 implementation is safe from race conditions, and the returned filename will not be reused by this JVM.

17.4 Reading and writing whole files

The following procedures and syntax allow you to read and write the entire contents of a file, without iterating using a port.

17.4.1 Reading a file

For reading the contents of a file in a single operation, you can use the following syntax:

&<{ *named-literal-part*+}

This is equivalent to using the `path-data` function (defined below):

(path-data &{ *named-literal-part*+} **)**

For example:

```
(define dir "/home/me/")
(define help-message &<{&[dir]HELP})
```

This binds `help-message` to the contents of the file named `HELP` in the `dir` directory.

17.4.2 Blobs

The content of a file is, in general, a sequence of uninterpreted bytes. Often these bytes represent text in a locale-dependent encoding, but we don't always know this. Sometimes they're images, or videos, or word-processor documents. A filename extension or a "magic number" in the file can give you hints, but not certainty as to the type of the data.

A *blob* (`http://en.wikipedia.org/wiki/Binary_large_object`) is a raw uninterpreted sequence of bytes. It is a `bytevector` that can be automatically converted to other types as needed, specifically to a string or a bytevector.

The `&<{..}` returns a blob. For example, assume the file `README` contains (bytes representing) the text `"Check doc directory.\n"`. Then:

```
#|kawa:1|# (define readme &<{README}))
|kawa:2|# readme:class
class gnu.lists.Blob
#|kawa:3|# (write (->string readme))
"Check doc directory.\n"
#|kawa:4|# (write (->bytevector readme))
#u8(67 104 101 99 107 32 100 111 99 32 100 105 114 101 99 116 111 114 121 46 10)
#|kawa:5|# (->bytevector readme):class
class gnu.lists.U8Vector
```

17.4.3 Writing to a file

The `&<{..}` syntax can be used with `set!` to replace the contents of a file:

```
(set! &<{README} "Check example.com\n")
```

The new contents must be blob-compatible - i.e. a bytevector or a string.

If you dislike using `<` as an output operator, you can instead using the `&>{..}` operation, which evaluates to a function whose single argument is the new value:

```
(&>{README} "Check example.com\n")
```

In general:

&>{ *named-literal-part+*}

is equivalent to:

```
(lambda (new-contents)
  (set! &<{ named-literal-part+} new-contents))
```

You can use `&>>` to append more data to a file:

```
(&>>{README} "or check example2.com\n")
```

17.4.4 Functions

`path-data` *path* [Procedure]

Reads the contents of the file specified by *path*, where *path* can be a Section 17.2 [Paths], page 271, object, or anything that can be converted to a `Path`, including a filename string or a URL. returning the result as a blob. The result is a *blob*, which is a kind of bytevector than can be auto-converted to a string or bytevecor as required.

The function `path-data` has a setter, which replaces the contents with new contents:

```
(set! &<{file-name} new-contents)
```

`path-bytes` *path* [Procedure]

Reads the contents of the file specified by *path*, as with the `path-data` function, but the result is a plain bytevector, rather than a blob. This functtion also has a setter, which you can use to replace the file-contents by new bytevector-valued data.

17.5 Ports

Ports represent input and output devices. An input port is a Scheme object that can deliver data upon command, while an output port is a Scheme object that can accept data.

Different *port types* operate on different data:

- A *textual port* supports reading or writing of individual characters from or to a backing store containing characters using `read-char` and `write-char` below, and it supports operations defined in terms of characters, such as `read` and `write`.
- A *binary port* supports reading or writing of individual bytes from or to a backing store containing bytes using `read-u8` and `write-u8` below, as well as operations defined in terms of bytes (integers in the range 0 to 255).

  All Kawa binary ports created by procedures documented here are also textual ports. Thus you can either read/write bytes as described above, or read/write characters whose scalar value is in the range 0 to 255 (i.e. the Latin-1 character set), using `read-char` and `write-char`.

  A native binary port is a `java.io.InputStream` or `java.io.OutputStream` instance. These are not textual ports. You can use methods `read-u8` and `write-u8`, but not `read-char` and `write-char` on native binary ports. (The functions `input-port?`, `output-port?`, `binary-port?`, and `port?` all currently return false on native binary ports, but that may change.)

`call-with-port` *port proc* [Procedure]

The `call-with-port` procedure calls *proc* with port as an argument. If *proc* returns, then the port is closed automatically and the values yielded by the proc are returned.

If *proc* does not return, then the port must not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

As a Kawa extension, *port* may be any object that implements `java.io.Closeable`. It is an error if *proc* does not accept one argument.

`call-with-input-file` *path proc* [Procedure]
`call-with-output-file` *path proc* [Procedure]

These procedures obtain a textual port obtained by opening the named file for input or output as if by `open-input-file` or `open-output-file`. The port and *proc* are then passed to a procedure equivalent to `call-with-port`.

It is an error if *proc* does not accept one argument.

`input-port?` *obj* [Procedure]
`output-port?` *obj* [Procedure]
`textual-port?` *obj* [Procedure]
`binary-port?` *obj* [Procedure]

`port?` *obj* [Procedure]

These procedures return `#t` if obj is an input port, output port, textual port, binary port, or any kind of port, respectively. Otherwise they return `#f`.

These procedures currently return `#f` on a native Java streams (`java.io.InputStream` or `java.io.OutputStream`), a native reader (a `java.io.Reader` that is not an `gnu.mapping.Inport`), or a native writer (a `java.io.Writer` that is not an `gnu.mapping.Outport`). This may change if conversions between native ports and Scheme ports becomes more seamless.

`input-port-open?` *port* [Procedure]
`output-port-open?` *port* [Procedure]

Returns `#t` if *port* is still open and capable of performing input or output, respectively, and `#f` otherwise. (Not supported for native binary ports - i.e. `java.io.InputStteam` or `java.io.OutputStream`.)

`current-input-port` [Procedure]
`current-output-port` [Procedure]
`current-error-port` [Procedure]

Returns the current default input port, output port, or error port (an output port), respectively. (The error port is the port to which errors and warnings should be sent - the *standard error* in Unix and C terminology.) These procedures are Section 15.2 [Parameter objects], page 268, which can be overridden with [parameterize-syntax], page 269.

The initial bindings for `(current-output-port)` and `(current-error-port)` are hybrid textual/binary ports that wrap the values of the corresponding `java.lang.System` fields `out`, and `err`. The latter, in turn are bound to the standard output and error streams of the JVM process. This means you can write binary data to standard output using `write-bytevector` and `write-u8`.

The initial value `(current-input-port)` similarly is a textual port that wraps the `java.lang.System` field `in`, which is bound to the standard input stream of the JVM process. It is a *hybrid* textual/binary port only if there is no console (as determined by `(java.lang.System:console)` returning `#!null`) - i.e. if standard input is not a tty.

Here is an example that copies standard input to standard output:

```
(let* ((in (current-input-port))
       (out (current-output-port))
       (blen ::int 2048)
       (buf (make-bytevector blen)))
  (let loop ()
    (define n (read-bytevector! buf in))
    (cond ((not (eof-object? n))
           (write-bytevector buf out 0 n)
           (loop)))))
```

`with-input-from-file` *path thunk* [Procedure]
`with-output-to-file` *path thunk* [Procedure]

The file is opened for input or output as if by `open-input-file` or `open-output-file`, and the new port is made to be the value returned by `current-input-port` or `current-output-port` (as used by `(read)`, `(write obj)`, and so forth). The thunk is then called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. It is an error if *thunk* does not accept zero arguments. Both procedures return the values yielded by *thunk*. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input or output port had been bound dynamically with `parameterize`.

`open-input-file` *path* [Procedure]
`open-binary-input-file` *path* [Procedure]

Takes a *path* naming an existing file and returns a textual input port or binary input port that is capable of delivering data from the file.

The procedure `open-input-file` checks the fluid variable [port-char-encoding], page 286, to determine how bytes are decoded into characters. The procedure `open-binary-input-file` is equivalent to calling `open-input-file` with `port-char-encoding` set to `#f`.

`open-output-file` *path* [Procedure]
`open-binary-output-file` *path* [Procedure]

Takes a *path* naming an output file to be created and returns respectively a textual output port or binary output port that is capable of writing data to a new file by that name. If a file with the given name already exists, the effect is unspecified.

The procedure `open-output-file` checks the fluid variable [port-char-encoding], page 286, to determine how characters are encoded as bytes. The procedure `open-binary-output-file` is equivalent to calling `open-output-file` with `port-char-encoding` set to `#f`.

`close-port` *port* [Procedure]
`close-input-port` *port* [Procedure]
`close-output-port` *port* [Procedure]

Closes the resource associated with *port*, rendering the port incapable of delivering or accepting data. It is an error to apply the last two procedures to a port which is not an input or output port, respectively. (Specifically, `close-input-port` requires a `java.io.Reader`, while `close-output-port` requires a `java.io.Writer`. In contrast `close-port` accepts any object whose class implements `java.io.Closeable`.)

These routines have no effect if the port has already been closed.

17.5.1 String and bytevector ports

`open-input-string` *string* [Procedure]

Takes a string and returns a text 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` [Procedure]

Returns an textual 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* [Procedure]

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 in the order they were output. If the result string is modified, the effect is unspecified.

```
(parameterize
    ((current-output-port (open-output-string)))
    (display "piece")
    (display " by piece ")
    (display "by piece.")
    (newline)
    (get-output-string (current-output-port)))
        ⇒ "piece by piece by piece.\n"
```

`call-with-input-string` *string proc* [Procedure]

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* [Procedure]

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.

`open-input-bytevector` *bytevector* [Procedure]

Takes a bytevector and returns a binary input port that delivers bytes from the bytevector.

`open-output-bytevector` [Procedure]

Returns a binary output port that will accumulate bytes for retrieval by `get-output-bytevector`.

`get-output-bytevector` *port* [Procedure]

Returns a bytevector consisting of the bytes that have been output to the port so far in the order they were output. It is an error if *port* was not created with `open-output-bytevector`.

17.5.2 Input

If *port* is omitted from any input procedure, it defaults to the value returned by `(current-input-port)`. It is an error to attempt an input operation on a closed port.

`read` [*port*] [Procedure]

The `read` procedure converts external representations of Scheme objects into the objects themselves. That is, it is a parser for the non-terminal *datum*. It returns the next object parsable from the given textual input port, updating port to point to the first character past the end of the external representation of the object.

If an end of file is encountered in the input before any characters are found that can begin an object, then an end-of-file object is returned. The port remains open, and further attempts to read will also return an end-of-file object. If an end of file is encountered after the beginning of an object's external representation, but the external representation is incomplete and therefore not parsable, an error that satisfies `read-error?` is signaled.

`read-char` [*port*] [Procedure]

Returns the next character available from the textual input *port*, updating the port to point to the following character. If no more characters are available, an end-of-file value is returned.

The result type is `character-or-eof`.

`peek-char` [*port*] [Procedure]

Returns the next character available from the textual input *port*, but *without* updating the port to point to the following character. If no more characters are available, an end-of-file value is returned.

The result type is `character-or-eof`.

Note: The value returned by a call to `peek-char` is the same as the value that would have been returned by a call to `read-char` with the same *port*. The only difference is that the very next call to `read-char` or `peek-char` on that *port* will return the value returned by the preceding call to `peek-char`. In particular, a call to `peek-char` on an interactive port will hang waiting for input whenever a call to `read-char` would have hung.

`read-line` [*port* [*handle-newline*]] [Procedure]

Reads a line of input from the textual input *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. If an end of file is encountered before any end of line is read, but some characters have been read, a string containing those characters is returned. (In this case, `'trim`, `'peek`, and `'concat` have the same result and effect. The `'split` case returns two values: The characters read, and the delimiter is an empty string.) If an end of file is encountered before any characters are read, an end-of-file object is returned. For the purpose of this procedure, an end of line consists of either a linefeed character, a carriage return character, or a sequence of a carriage return character followed by a linefeed character.

`eof-object?` *obj* [Procedure]

Returns `#t` if *obj* is an end-of-file object, otherwise returns `#f`.

`Performance note`: If *obj* has type `character-or-eof`, this is compiled as an `int` comparison with -1.

`eof-object` [Procedure]

Returns an end-of-file object.

`char-ready?` [*port*] [Procedure]

Returns `#t` if a character is ready on the textual input *port* and returns `#f` otherwise. If char-ready returns `#t` then the next `read-char` operation on the given *port* is guaranteed not to hang. If the port is at end of file then `char-ready?` returns `#t`.

Rationale: The `char-ready?` procedure exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors as- sociated with such ports must ensure that characters whose existence has been asserted by `char-ready?` cannot be removed from the input. If `char-ready?` were to return `#f` at end of file, a port at end-of-file would be indistinguishable from an interactive port that has no ready characters.

`read-string` *k* [*port*] [Procedure]

Reads the next *k* characters, or as many as are available before the end of file, from the textual input *port* into a newly allocated string in left-to-right order and returns the string. If no characters are available before the end of file, an end-of-file object is returned.

`read-u8` [*port*] [Procedure]

Returns the next byte available from the binary input *port*, updating the *port* to point to the following byte. If no more bytes are available, an end-of-file object is returned.

`peek-u8` [*port*] [Procedure]

Returns the next byte available from the binary input *port*, but *without* updating the *port* to point to the following byte. If no more bytes are available, an end-of-file object is returned.

`u8-ready?` [*port*] [Procedure]

Returns `#t` if a byte is ready on the binary input *port* and returns `#f` otherwise. If `u8-ready?` returns `#t` then the next `read-u8` operation on the given port is guaranteed not to hang. If the port is at end of file then `u8-ready?` returns `#t`.

`read-bytevector` *k* [*port*] [Procedure]

Reads the next *k* bytes, or as many as are available before the end of file, from the binary input *port* into a newly allocated bytevector in left-to-right order and returns the bytevector. If no bytes are available before the end of file, an end-of-file object is returned.

`read-bytevector!` *bytevector* [*port* [*start* [*end*]]] [Procedure]

Reads the next *end* − *start* bytes, or as many as are available before the end of file, from the binary input *port* into *bytevector* in left-to-right order beginning at the *start* position. If *end* is not supplied, reads until the end of *bytevector* has been reached. If *start* is not supplied, reads beginning at position 0. Returns the number of bytes read. If no bytes are available, an end-of-file object is returned.

17.5.3 Output

If *port* is omitted from any output procedure, it defaults to the value returned by `(current-output-port)`. It is an error to attempt an output operation on a closed port.

The return type of these methods is `void`.

`write` *obj* [*port*] [Procedure]

Writes a representation of *obj* to the given textual output *port*. Strings that appear in the written representation are enclosed in quotation marks, and within those strings backslash and quotation mark characters are escaped by backslashes. Symbols that contain non-ASCII characters are escaped with vertical lines. Character objects are written using the `#\` notation.

If *obj* contains cycles which would cause an infinite loop using the normal written representation, then at least the objects that form part of the cycle must be represented using [datum labels], page 110. Datum labels must not be used if there are no cycles.

`write-shared` *obj* [*port*] [Procedure]

The `write-shared` procedure is the same as `write`, except that shared structure must be represented using datum labels for all pairs and vectors that appear more than once in the output.

`write-simple` *obj* [*port*] [Procedure]

The `write-simple` procedure is the same as `write`, except that shared structure is never represented using datum labels. This can cause write-simple not to terminate if *obj* contains circular structure.

`display` *obj* [*port*] [Procedure]

Writes a representation of *obj* to the given textual output port. Strings that appear in the written representation are output as if by `write-string` instead of by `write`. Symbols are not escaped. Character objects appear in the representation as if written by `write-char` instead of by `write`. The `display` representation of other objects is unspecified.

`newline` [*port*] [Procedure]

Writes an end of line to textual output *port*. This is done using the `println` method of the Java class `java.io.PrintWriter`.

`write-char` *char* [*port*] [Procedure]

Writes the character *char* (not an external representation of the character) to the given textual output *port*.

`write-string` *string* [*port* [*start* [*end*]]] [Procedure]

Writes the characters of *string* from *start* to *end* in left-to-right order to the textual output *port*.

`write-u8` *byte* [*port*] [Procedure]

Writes the *byte* to the given binary output port.

`write-bytevector` *bytevector* [*port* [*start* [*end*]]] [Procedure]

Writes the bytes of *bytevector* from *start* to *end* in left-to-right order to the binary output *port*.

`flush-output-port` [*port*] [Procedure]
`force-output` [*port*] [Procedure]

Forces any pending output on *port* to be delivered to the output file or device and returns an unspecified value. If the *port* argument is omitted it defaults to the value returned by `(current-output-port)`. (The name `force-output` is older, while R6RS added `flush-output-port`. They have the same effect.)

17.5.4 Prompts for interactive consoles (REPLs)

When an interactive input port is used for a read-eval-print-loop (REPL or console) it is traditional for the REPL to print a short *prompt* string to signal that the user is expected to type an expression. These prompt strings can be customized.

`input-prompt1` [Variable]
`input-prompt2` [Variable]

These are fluid variable whose values are string templates with placeholders similar to `printf`-style format. The placeholders are expanded (depending on the current state), and the resulting string printed in front of the input line.

The `input-prompt1` is used normally. For multi-line input commands (for example if the first line is incomplete), `input-prompt1` is used for the first line of each command, while `input-prompt2` is used for subsequent "continuation" lines.

The following placeholders are handled:

%% A literal '%'.

%N The current line number. This is `(+ 1 (port-line` *`port`*`))`.

%*n*P*c* Insert padding at this possion, repeating the following character *`c`* as needed to bring the total number of columns of the prompt to that specified by the digits *`n`*.

%P*c* Same as `%`*`n`*`P`*`c`*, but *`n`* defaults to the number of columns in the initial prompt from the expansion of `input-prompt1`. This is only meaningful when expanding `input-prompt2` for continuation lines.

%{*hidden*%}
Same as *hidden*, but the characters of *hidden* are assumed to have zero visible width. Can be used for ANSI escape sequences (`https://en.wikipedia.org/wiki/ANSI_escape_code`) to change color or style:

```
(set! input-prompt1 "%{\e[48;5;51m%}{Kawa:%N} %{\e[0m%}")
```

The above changes both the text and the background color (to a pale blue).

%H*cd* If running under DomTerm, use the characters *c* and *d* as a clickable mini-button to hide/show (fold) the command and its output. (When output is visible *c* is displayed; clicking on it hides the output. When output is hidden *d* is displayed; clicking on it shows the output.) Ignored if not running under DomTerm.

%M Insert a "message" string. Not normally used by Kawa, but supported by JLine.

These variables can be initialized by the command-line arguments `console:prompt1=`*`prompt1`* and `console:prompt2=`*`prompt2`*, respectively. If these are not specified, languages-specific defaults are used. For example for Scheme the default value of `input-prompt1` is `"#|%Hkawa:%N|# "` and `input-prompt2` is `"#|%P.%N| "`. These have the form of Scheme comments, to make it easier to cut-and-paste.

If `input-prompt1` (respectively `input-prompt2`) does not contain an escape sequence (either `"%{` or the escape character `"\e"`) then ANSI escape sequences are added to to highlight the prompt. (Under DomTerm this sets the `prompt` style, which can be customised with CSS but defaults to a light green background; if using JLine the background is set to light green.)

For greater flexibility, you can also set a prompter procedure.

`set-input-port-prompter!` *port prompter* [Procedure]

Set the prompt procedure associated with *port* to *prompter*, which must be a one-argument procedure taking an input port, and returning a string. The procedure is called before reading the first line of a command; its return value is used as the first-line prompt.

The prompt procedure can have side effects. In Bash shell terms: It combines the features of `PROMPT_COMMAND` and `PS1`.

The initial *prompter* is `default-prompter`, which returns the expansion of `input-prompt1`.

`input-port-prompter` *port* [Procedure]

Get the prompt procedure associated with *port*.

`default-prompter` *port* [Procedure]

The default prompt procedure. Normally (i.e. when `input-port-read-state` is a space) returns `input-prompt1` after expanding the `%`-placeholders. Can also expand `input-prompt2` when `input-port-read-state` is not whitespace.

17.5.5 Line numbers and other input port properties

`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* [Procedure]
: Set (0-origin) line number of the current line of *port* to *num*.

`input-port-line-number` *port* [Procedure]
: 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* [Procedure]
: 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* [Procedure]
: 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* [Procedure]
: 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.) The value of `symbol-read-case` only checked when a reader is created, not each time a symbol is read.

17.5.6 Miscellaneous

`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)))
```

`*print-base*` [Variable]

The number base (radix) to use by default when printing rational numbers. Must be an integer between 2 and 36, and the default is of course 10. For example setting `*print-base*` to 16 produces hexadecimal output.

`*print-radix*` [Variable]

If true, prints an indicator of the radix used when printing rational numbers. If `*print-base*` is respectively 2, 8, or 16, then `#b`, `#o` or `#x` is written before the number; otherwise `#Nr` is written, where `N` is the base. An exception is when `*print-base*` is 10, in which case a period is written *after* the number, to match Common Lisp; this may be inappropriate for Scheme, so is likely to change.

`*print-right-margin*` [Variable]

The right margin (or line width) to use when pretty-printing.

`*print-miser-width*` [Variable]

If this an integer, and the available width is less or equal to this value, then the pretty printer switch to the more *miser* compact style.

`*print-xml-indent*` [Variable]

When writing to XML, controls pretty-printing and indentation. If the value is `'always` or `'yes` force each element to start on a new suitably-indented line. If the value is `'pretty` only force new lines for elements that won't fit completely on a line. The the value is `'no` or unset, don't add extra whitespace.

17.6 Formatted Output (Common-Lisp-style)

`format` *destination fmt . arguments* [Procedure]

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* [Procedure]

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`!

17.6.1 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).

17.6.2 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,commawidthB` padding.

`~nR` Radix *n*.

- `~n,mincol,padchar,commachar,commawidthR` 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.

17.6.3 Formatting real numbers

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

- `~width,digits,scale,overflowchar,padcharF`
- `~@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,exponentcharE`
- `~@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,exponentcharG`
- `~@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.

17.6.4 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,colincT`
  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*

`~<`*mincol* Start justification: spacing is evenly distributed between text segments with a width of *mincol*. This enables an even right margin.

`~>` End of segments to justify.

17.6.5 Unimplemented CL Format Control Directives

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

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

`~:^`

17.6.6 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.

17.7 Pretty-printing

Pretty-printing is displaying a data structure as text, by adding line-breaks and indenttaion so that the visual structure of the output corresponds to the logical structure of data structure. This makes it easier to read and understand. Pretty-printing takes into account the column width of the output so as to avoid using more lines than needed.

Pretty-printing of standard sequences types such as lists and vectors is done by default. For example:

```
#|kawa:11|# (set! *print-right-margin* 50)
#|kawa:12|# '(ABCDEF (aa bb cc dd) (x123456789
#|.....13|# y123456789 z123456789) ABCDEFG HIJKL)
(ABCDEF (aa bb cc dd)
 (x123456789 y123456789 z123456789) ABCDEFG HIJK)
```

Setting `*print-right-margin*` to 50 causes output to be limited to 50 columns. Notice the top-level list has to be split, but sub-lists `(aa bb cc dd)` and `(x123456789 y123456789 z123456789)` don't need to be split.

When outputting to a DomTerm REPL, then `*print-right-margin*` is ignored, and the line-breaking is actually handled by DomTerm. If you change the window width, DomTerm will dynamically re-calculate the line-breaks of previous pretten output. This works even in the case of a session saved to an HTML file, as long as JavaScript is enabled.

The concepts and terminology are based on those of Common Lisp (`https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node253.html`).

17.7.1 Pretty-printing Scheme forms

Scheme and Lisp code is traditionally pretty-printed slightly differently than plain lists. The `pprint` procedure assumes the argument is a Scheme form, and prints its accordingly. For example the special form `(let ...)` is printed differently from a regular function call `(list ...)`.

`pprint` *obj* [*out*] [Procedure]

Assume *obj* is a Scheme form, and pretty-print it in traditional Scheme format. For example:

```
#|kawa:1|# (import (kawa pprint))
#|kawa:2|# (define fib-form
#|.....3|#   '(define (fibonacci n)
#|.....4|#      (let loop ((i0 0) (i1 1) (n n))
#|.....5|#        (if (<= n 0) i0
#|.....6|#            (loop i1 (+ i0 i1) (- n 1))))))
#|kawa:7|# (set! *print-right-margin* 80)
#|kawa:8|# (pprint fib-form)
```

```
(define (fibonacci n)
  (let loop ((i0 0) (i1 1) (n n)) (if (<= n 0) i0 (loop i1 (+ i0 i1) (-
n 1)))))
#|kawa:9|# (set! *print-right-margin* 40)
#|kawa:10|# (pprint fib-form)
(define (fibonacci n)
  (let loop ((i0 0) (i1 1) (n n))
    (if (<= n 0)
        i0
        (loop i1 (+ i0 i1) (- n 1)))))
```

The `pprint` special-cases forms that start with `define`, `if`, `lambda`, `let`, and a few more, and formats them with "traditional" indentation. However, it is not as complete or polished as it should be. (It should also use a programmable dispatch table, rather than having these special cases hard-wired. That is an improvemet for another day.)

17.7.2 Generic pretty-printing functions

The following procedures are used to indicate logical blocks, and optional newlines.

To access them do:

```
(import (kawa pprint))
```

In the following, *out* is the output port, which defaults to `(current-output-port)`.

`pprint-logical-block` *options statement** [Syntax]

Evaluate the *statement*s within the context of a new "logical block".

The *options* are one or more of the following:

prefix: *prefix*
per-line: *per-line-prefix*
: Emit *prefix* or *per-line-prefix* (only one of them can be specified) before the start of the logical block. If *per-line-prefix* is provided, it is also print for each line within the logical block, indented the same. These are strings and default to `""`.

suffix: *suffix*
: Emit *suffix* after the end of the logical block.

out: *out* The output file.

For example to print a list you might do:

```
(pprint-logical-block prefix: "(" suffix: ")"
   print contents of list)
```

This macro is equivalent to:

```
(pprint-start-logical-block prefix is-per-line suffix out)
(try-finally
  (begin  statement*)
  (pprint-end-logical-block suffix out))
```

`pprint-start-logical-block` *prefix is-per-line suffix out* [Procedure]

Start a logical block. The *is-per-line* argument is a boolean to specifiy of *prefix* is a per-line-prefix or a plain prefix.

`pprint-end-logical-block` *suffix out* [Procedure]
: End a logical block.

`pprint-newline` *kind* [*out*] [Procedure]
: Print a conditional newline, where *kind* is one of the symbols `'fill`, `'linear`, `'mandatory`, or `'miser`. Usually follows printing of a space, as nothing is printed if the line is not broken here.

`pprint-ident` *mode amount* [*out*] [Procedure]
: Change how much following lines are indented (with the current logical block). The *amount* is the size of the indentation, in characters. The *mode* is either `'current` (if the *amount* is relative to the current position), or `'block` (if the *amount* is relative to the start (after any *prefix*) of the current logical block).

17.8 Resources

A resource is a file or other fixed data that an application may access. Resources are part of the application and are shipped with it, but are stored in external files. Examples are images, sounds, and translation (localization) of messages. In the Java world a resource is commonly bundled in the same jar file as the application itself.

`resource-url` *resource-name* [Syntax]
: Returns a `URLPath` you can use as a `URL`, or you can pass to it `open-input-file` to read the resource data. The *resource-name* is a string which is passed to the `ClassLoader` of the containing module. If the module class is in a jar file, things will magically work if the resource is in the same jar file, and *resource-name* is a filename relative to the module class in the jar. If the module is immediately evaluated, the *resource-name* is resolved against the location of the module source file.

`module-uri` [Syntax]
: Evaluates to a special URI that can be used to access resources relative to the class of the containing module. The URI has the form `"class-resource://`*`CurrentClass`*`/"` in compiled code, to allow moving the classes/jars. The current `ClassLoader` is associated with the URI, so accessing resources using the URI will use that `ClassLoader`. Therefore you should not create a `"class-resource:"` URI except by using this function or `resolve-uri`, since that might try to use the wrong `ClassLoader`.

: The macro `resource-url` works by using `module-uri` and resolving that to a normal `URL`.

`module-class` [Syntax]
: Evaluates to the containing module class, as a `java.lang.Class` instance.

18 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. The most used types correspond to Java classes and primitive types, but Kawa also has other non-Java types.

Type specifiers have a *type expressions*, and a type expression is conceptually an expression that 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.

type `::=` *expression*
opt-type-specifier `::=` [**::** *type*]

Various Kawa constructs require or allow a type to be specified. You can use a type specifier most places where you Section 8.3 [Variables and Patterns], page 140. Types specifiers can appear in other placess, such as procedure return type specifiers. For example in this procedure definition, `::vector` is an argument type specifier (and `vec::vector` is a pattern), while `::boolean` is a return type specifier.

```
(define (vector-even? vec::vector)::boolean
  (not (odd? (vector-length vec))))
(vector-even? #(3 4 5)) ⇒ #f
(vector-even? (list 3 4 5 6)) ⇒ error
```

18.1 Standard Types

These types are predefined with the following names.

Instead of plain ***typename*** you can also use the syntax `<`***typename***`>` with angle brackets, but that syntax is no longer recommended, because it doesn't "fit" as well with some ways type names are used.

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`.

The numeric types (`number`, `quantity`, `complex`, `real`, `rational`, `integer`, `long`, `int`, `short`, `byte ulong`, `uint`, `ushort`, `ubyte`, `double`, `float`) are discussed in Section 12.1 [Numerical types], page 175.

The types `character` and `char` are discussed in Section 13.1 [Characters], page 202.

`Object` [Variable]

An arbitrary Scheme value - and hence an arbitrary Java object.

`symbol` [Variable]

The type of Scheme symbols. (Implemented using the Java class `gnu.mapping.Symbol`.) (*Compatibility:* Previous versions of Kawa implemented a simple Scheme symbol using an interned `java.lang.String`.)

`keyword` [Variable]

The type of keyword values. See Section 10.3 [Keywords], page 165.

`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 Scheme strings. (Implemented using `gnu.lists.IString` or `java.lang.String` for immutable strings, and `gnu.lists.FString` for mutable strings. These all implement the interface `java.lang.CharSequence`. (*Compatibility:* Previous versions of Kawa always used `gnu.lists.FString`.)

`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`. 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 method, any parameter whose type is `java.lang.String` is converted as if it was declared as a `String`.

`parameter` [Variable]

A parameter object, as created by `make-parameter`. This type can take a type parameter (sic):

```
(define-constant client ::parameter[Client] (make-parameter #!null))
```

This lets Kawa know that reading the parameter (as in `(client)`) returns a value of the specified type (in this case `Client`).

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.

`dynamic` [Variable]

Used to specify that the type is unknown, and is likely to change at run-time. Warnings about unknown member names are supressed (a run-time name lookup is formed). An expression of type `dynamic` is (statically) compatible with any type.

18.2 Parameterized Types

Kawa has some basic support for parameterized (generic) types. The syntax:

```
Type[Arg1 Arg2 ... ArgN]
```

is more-or-less equivalent to Java's:

```
Type<Arg1, Arg2, ..., ArgN>
```

This is a work-in-progress. You can use this syntax with fully-qualified class names, and also type aliases:

```
(define v1 ::gnu.lists.FVector[gnu.math.IntNum] [4 5 6])
(define-alias fv gnu.lists.FVector)
(define v2 ::fv[integer] [5 6 7])
(define-alias fvi fv[integer])
(define v3 ::fvi [6 7 8])
```

18.3 Type tests and conversions

Scheme defines a number of standard type testing predicates. For example `(vector? x)` is `#t` if and only if `x` is a vector.

Kawa generalizes this to arbitrary type names: If *T* is a type-name (that is in scope at compile-time), then *T*? is a one-argument function that returns `#t` if the argument is an instance of the type *T*, and `#f` otherwise:

```
(gnu.lists.FVector? #(123)) ⇒ #t
(let ((iarr (int[] 10))) (int[]? iarr)) ⇒ #t
```

To convert (coerce) the result of an expression *value* to a type *T* use the syntax: `(->T value)`.

```
(->float 12) ⇒ 12.0f0
```

In general:

```
(T? x) ⇒ (instance? x T)
(->T x) ⇒ (as T x)
```

`instance?` *value type* [Procedure]

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

`as` *type value* [Procedure]

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`.

19 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 classes).

19.1 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` *`class-name`* **(***supers ...***)** *(`annotation`|`option-pair`)** *`field-or-method-decl`* ... [Syntax]

`define-simple-class` *`class-name`* **(***supers ...***)** *(`annotation`|`option-pair`)** *`field-or-method-decl`* ... [Syntax]

Defines a new class named *class-name*. If `define-simple-class` is used, creates a normal Java class named *class-name* in the current package. (If *class-name* has the form `<xyz>` the Java implementation type is named `xyz`.) For `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.

class-name `::=` *identifier*
option-pair `::=` *option-keyword option-value*
field-or-method-decl `::=` *field-decl* `|` *method-decl*

19.1.1 General class properties

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 optionally 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`.

interface: *make-interface*
: Specifies whether Kawa generates a Java class, interface, or both. If *make-interface* is `#t`, then a Java interface is generated. In that case all the supertypes must be interfaces, and all the declared methods must be abstract. If *make-interface* is `#f`, then a Java class is generated. If `interface:` is unspecified, the default is `#f` for `define-simple-class`. For `define-class` the default is to generate an interface, and in addition (if needed) a helper class that implements the interface. (In that case any non-abstract methods are compiled to static methods. The methods that implement the interface are just wrapper methods that call the real static methods. This allows Kawa to implement true multiple inheritance.)

access: *kind*
: Specifies the Java access permission on the class. Can be one of `'public` (which is the default in Kawa), `'package` (which the default "unnamed" permission in Java code), `'protected`, `'private`, `'volatile`, or `'transient`. Can also be used to specify `final`, `abstract`, or `enum`, as in Java. (You don't need to explicitly specify the class is `abstract` if any *method-body* is `#!abstract`, or you specify `interface: #t`.) The *kind* can also be a list, as for example:

```
access: '(protected volatile)
```

class-name: "*cname*"
: Specifies the Java name of the created class. The *name* specified after `define-class` or `define-simple-class` is the *Scheme name*, i.e. the name of a Scheme variable that is bound to the class. The Java name is by default derived from the Scheme name, but you can override the default with a `class-name:` specifier. If the *cname* has no periods, then it is a name in the package of the main (module) class. If the *cname* starts with a period, then you get a class nested within the module class. In this case the actual class name is *moduleClass*`$`*rname*, where *rname* is *cname* without the initial period. To force a class in the top-level (unnamed) package (something not recommended) write a period at the end of the *cname*.

19.1.2 Declaring fields

field-decl `::=` **(***field-name* (*annotation* `|` *opt-type-specifier* `|` *field-option*)***)**
field-name `::=` *identifier*
field-option `::=` *keyword expression*

As a matter of style the following order is suggested, though this not enforced:

(*field-name annotation* opt-type-specifier field-option****)**

Each *field-decl* declares a instance "slot" (field) with the given *field-name*. By default it is publicly visible, but you can specify a different visiblity with the `access:` specifier. The following *field-option keywords* are implemented:

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

allocation: *kind*
If *kind* is `'class` or `'static` a single slot is 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.

If *kind* is `'instance` then 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.

access: *kind*
Specifies the Java access permission on the field. Can be one of `'private`, `'protected`, `'public` (which is the default in Kawa), or `'package` (which the default "unnamed" permission in Java code). Can also be used to specify `volatile`, `transient`, `enum`, or `final`, as in Java, or a quoted list with these symbols.

init: *expr* An expression used to initialize the slot. The expression is evaluated in a scope that includes the field and method names of the current class.

init-form: *expr*
An expression used to initialize the slot. The lexical environment of the *expr* is that of the `define-class`; it does *not* include the field and method names of the current class. or `define-simple-class`.

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 *field-name*.

The *field-name* can be left out. That indicates a "dummy slot", which is useful for initialization not tied to a specific field. In Java terms this is an instance or static initializer, i.e., a block of code executed when a new instance is created or the class is loaded.

In this example, `x` is the only actual field. It is first initialized to 10, but if `(some-condition)` is true then its value is doubled.

```
(define-simple-class <my-class> ()
  (allocation: 'class
   init: (perform-actions-when-the-class-is-initizalized))
  (x init: 10)
  (init: (if (some-condition) (set! x (* x 2)))))
```

19.1.3 Declaring methods

method-decl `::=` ((*method-name formal-arguments*)
    method-option * [*deprecated-return-specifier*] *method-body*)
method-name `::=` *identifier*
method-option `::=` *annotation* `|` *opt-return-type* `|` *option-pair*
method-body `::=` *body* `|` **#!abstract** `|` **#!native**
deprecated-return-specifier `::=` *identifier*

Each *method-decl* declares a method, which is by default public and non-static, and 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 a *opt-return-type*, *deprecated-return-specifier*, 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.)

A *method-decl* in a `define-simple-class` can have the following *option-keyword*s:

access: *kind*
: Specifies the Java access permission on the method. Can be one of `'private`, `'protected`, `'public`, or `'package`. Can also be `'synchronized`, `'final`, `'strictfp`, or a quoted list.

allocation: *kind*
: If *kind* is `'class` or `'static` creates a static method.

throws: (*exception-class-name* ...)
: Specifies a list of checked exception that the method may throw. Equivalent to a `throws` specification in Java code. For example:

```
(define-simple-class T
  (prefix)
  ((lookup name) throws: (java.io.FileNotFoundException)
   (make java.io.FileReader (string-append prefix name))))
```

The scope of the *body* of a method includes the *field-decl*s and *method-decl*s of the class, including those inherited from superclasses and implemented interfaces.

If the *method-body* is the special form `#!abstract`, then the method is abstract. This means the method must be overridden in a subclass, and you're not allowed to create an instance of the enclosing class.

```
(define-simple-class Searchable () interface: #t
  ((search value) :: boolean #!abstract))
```

If the *method-body* is the special form `#!native`, then the method is native, implemented using JNI (`http://en.wikipedia.org/wiki/Java_Native_Interface`).

The special *method-name* '`*init*`' can be used to name a non-default constructor (only if *make-interface* discussed above is `#f`). It can be used to initialize a freshly-allocated instance using passed-in parameters. You can call a superclass or a sibling constructor using the `invoke-special` special function. (This is general but admittedly a bit verbose; a more compact form may be added in the future.) See the example below.

19.1.4 Example

In the following example we define a simple class `2d-vector` and a class `3d-vector` that extends it. (This is for illustration only - defining 3-dimensional points as an extension of 2-dimensional points does not really make sense.)

```
(define-simple-class 2d-vector ()
  (x ::double init-keyword: x:)
  ;; Alternative type-specification syntax.
  (y type: double init-keyword: y:)
  (zero-2d :: 2d-vector allocation: 'static
   init-value: (2d-vector 0))
  ;; An object initializer (constructor) method.
  ((*init* (x0 ::double) (y0 ::double))
   (set! x x0)
   (set! y y0))
  ((*init* (xy0 ::double))
   ;; Call above 2-argument constructor.
   (invoke-special 2d-vector (this) '*init* xy0 xy0))
  ;; Need a default constructor as well.
  ((*init*) #!void)
  ((add (other ::2d-vector)) ::2d-vector
   ;; Kawa compiles this using primitive Java types!
   (2d-vector
     x: (+ x other:x)
     y: (+ y other:y)))
  ((scale (factor ::double)) ::2d-vector
   (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:)
  ;; A constructor which calls the superclass constructor.
  ((*init* (x0 ::double) (y0 ::double) (z0 ::double))
   (invoke-special 2d-vector (this) '*init* x0 y0)
   (set! z z0))
  ;; Need a default constructor.
  ((*init*) #!void)
  ((scale (factor ::double)) ::2d-vector
   ;; Note we cannot override the return type to 3d-vector
   ;; because Kawa doesn't yet support covariant return types.
   (3d-vector
     x: (* factor x)
     y: (* factor (this):y) ;; Alternative syntax.
     z: (* factor z))))
```

Note we define both explicit non-default constructor methods, and we associate fields with keywords, so they can be named when allocating an object. Using keywords requires a default constructor, and since having non-default constructors suppresses the implicit default constructor we have to explicitly define it. Using both styles of constructors is rather redundant, though.

19.2 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`.

object-field-or-method-decl `::=` *object-field-decl* `|` *method-decl*
object-field-decl `::=` **(***field-name* (*annotation* `|` *opt-type-specifier* `|` *field-option*)* [*object-init*] **)**
object-init `::=` *expression*

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 *object-finit* is given, it is an expression whose value becomes the initial value of the field. The *object-init* is evaluated at the same time as the `object` expression is evaluated, in a scope where all the *field-names* are visible.

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

19.2.1 Lambda as shorthand for anonymous class

An anonymous class is commonly used in the Java platform where a function language would use a lambda expression. Examples are call-back handlers, events handlers, and `run` methods. In these cases Kawa lets you use a lambda expression as a short-hand for an anonymous class. For example:

```
(button:addActionListener
  (lambda (e) (do-something)))
```

is equivalent to:

```
(button:addActionListener
  (object (java.awt.event.ActionListener)
    ((actionPerformed (e ::java.awt.event.ActionEvent))::void
     (do-something))))
```

This is possible when the required type is an interface or abstract class with a Single (exactly one) Abstract Methods. Such a class is sometimes called a *SAM-type*, and the conversion from a lambda expression to an anonymous class is sometimes called *SAM-conversion*.

Note that Kawa can also infer the parameter and return types of a method that overrides a method in a super-class.

19.3 Enumeration types

An enumeration type is a set of named atomic enumeration values that are distinct from other values. You define the type using `define-enum`, and you reference enumeration values using colon notation:

```
(define-enum colors (red blue green))
(define favorite-color colors:green)
```

Displaying an enum just prints the enum name, but readable output using `write` (or the `~s format` specifier) prepends the type name:

```
(format "~a" favorite-color) ⇒ "green"
(format "~s" favorite-color) ⇒ "colors:green"
```

The static `values` method returns a Java array of the enumeration values, in declaration order, while `ordinal` yields the index of an enumeration value:

```
(colors:values) ⇒ [red blue green]
((colors:values) 1) ⇒ blue
(favorite-color:ordinal) ⇒ 2
```

If you invoke the enumeration type as a function, it will map the name (as a string) to the corresponding value. (This uses the `valueOf` method.)

```
(colors "red") ⇒ red
(colors "RED") ⇒ throws IllegalArgumentException
(eq? favorite-color (colors:valueOf "green")) ⇒ #t
```

Kawa enumerations are based on Java enumerations. Thus the above is similar to a Java5 `enum` declaration, and the type `colors` above extends `java.lang.Enum`.

`define-enum` *enum-type-name* ***option-pair...*** (*enum-value-name ...*) ***field-or-method-decl...*** [Syntax]

This declares a new enumeration type *enum-type-name*, whose enumerations values are the *enum-value-name* list. You can specify extra options and members using *option-pair* and *field-or-method-decl*, which are as in `define-simple-class`. (The *define-enum* syntax is similar to a `define-simple-class` that extends `java.lang.Enum`.)

(Note that R6RS has a separate Enumerations library `(rnrs enum)`. Unfortunately, this is not compatible with standard Java enums. R6RS enums are simple symbols, which means you cannot distinguish two enum values from different enumeration types if they have the same value, nor from a vanilla symbol. That makes them less useful.)

19.4 Annotations of declarations

The Java platform lets you associate with each declaration zero or more annotations (`http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`). They provide an extensible mechanism to associate properties with declarations. Kawa support for annotations is not complete (the most important functionality missing is being able to declare annotation types), but is fairly functional. Here is a simple example illustrating use of JAXB annotations (`http://jcp.org/en/jsr/detail?id=222`): an `XmlRootElement` annotation on a class, and an `XmlElement` annotation on a field:

```
(import (class javax.xml.bind.annotation XmlRootElement XmlElement))
```

```
(define-simple-class Bib ( ) (@XmlRootElement name: "bib")
  (books (@XmlElement name: "book" type: Book) ::java.util.ArrayList))
(define-simple-class Book () ...)
```

This tutorial (`http://per.bothner.com/blog/2011/Using-JAXB-annotations`) explains the JAXB example in depth.

Here is the syntax:

annotation `::=` `(@`*annotation-typename annotations-element-values*`)`
annotations-element-values `::=` *annotation-element-value*
  `|` *annotation-element-pair ...*
annotation-element-pair `::=` *keyword annotation-element-value*
annotation-element-value `::=` *expression*
annotation-typename `::=` *expression*

An *annotations-element-values* consisting of just a single *annotation-element-value* is equivalent to an *annotation-element-pair* with a `value:` keyword.

Each *keyword* must correspond to the name of an element (a zero-argument method) in the annotation type. The corresponding *annotation-element-value* must be compatible with the element type (return type of the method) of the annotation type.

Allowed element types are of the following kinds:

- Primitive types, where the *annotation-element-value* must be number or boolean coercible to the element type.
- Strings, where the *annotation-element-value* is normally a string literal.
- Classes, where the *annotation-element-value* is normally a classname.
- Enumeration types. The value usually has the form *`ClassName`*`:`*`enumFieldname`*.
- Nested annotation types, where the *annotation-element-value* must be a compatible *annotation* value.
- An array of one of the allowable types. An array constructor expression works, but using the square bracket syntax is recommended.

Annotations are usually used in declarations, where they are required to be "constant-folded" to compile-time constant annotation values. This is so they can be written to class files. However, in other contexts an annotation can be used as an expression with general sub-expressions evaluated at run-time:

```
(define bk-name "book")
(define be (@XmlElement name: bk-name type: Book))
(be:name) ⇒ "book"
```

(This may have limited usefulness: There are some bugs, including lack of support for default values for annotation elements. These bugs can be fixed if someone reports a need for runtime construction of annotation values.)

19.5 Modules and how they are compiled to classes

Modules provide a way to organize Scheme into reusable parts with explicitly defined interfaces to the rest of the program. 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.

19.5.1 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`. (If compiling with the `--main` flag, then by default no identifiers are exported.) 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` (or `export`) declaration in the module, then only those names listed are exported. There can be more than one `module-export`, and they can be anywhere in the Scheme file. The recommended style has a single `module-export` near the beginning of the file.

`module-export` *`export-spec`** [Syntax]
`export` *`export-spec`** [Syntax]

The forms `export` and `module-export` are equivalent. (The older Kawa name is `module-export`; `export` comes from R7RS.) Either form specifies a list of identifiers which can be made visible to other libraries or programs.

export-spec `::=` *identifier*
`|` **(rename** *identifier*$_1$ *identifier*$_2$**)**

In the former variant, an *identifier* names a single binding defined within or imported into the library, where the external name for the export is the same as the name of the binding within the library. A `rename` spec exports the binding defined within or imported into the library and named by *identifier*$_1$, using *identifier*$_2$ as the external name.

Note that it is an error if there is no definition for *identifier* (or *identifier*$_1$) in the current module, or if it is defined using `define-private`.

As a matter of style, `export` or `module-export` should appear after `module-name` but *before* other commands (including `import` or `require`). (This is a requirement if there are any cycles.)

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) ...)
```

19.5.2 R7RS explicit library modules

A R7RS `define-library` form is another way to create a module. The R7RS term *library* is roughly the same as a Kawa module. In Kawa, each source file is a [implicit library], page 117, which may contain zero or more explicit sub-modules (in the form of `define-library`) optionally followed by the definitions and expressions of the implicit (file-level) module.

`define-library` ***`library-name library-declaration`**** [Syntax]

library-name `::=` `(` *library-name-parts* `)`
library-name-parts `::=` identifier+

A *library-name* is a list whose members are identifiers and exact non-negative integers. It is used to identify the library uniquely when importing from other programs or libraries. Libraries whose first identifier is `scheme` are reserved for use by the R7RS report and future versions of that report. Libraries whose first identifier is `srfi` are reserved for libraries implementing Scheme Requests for Implementation (`http://srfi.schemer.org/`). It is inadvisable, but not an error, for identifiers in library names to contain any of the characters `| \ ? * < " : > + [ ] / .` or control characters after escapes are expanded.

See [module-name], page 308, for how a *library-name* is mapped to a class name.

library-declaration `::=`
  export-declaration
  `|` *import-declaration*
  `|` **(begin** *statement* *)
  `|` **(include** *filename* +)
  `|` **(include-ci** *filename* +)
  `|` **(include-library-declarations** *filename* +)
  `|` **(cond-expand** *cond-expand-clause* * [(**else** command-or-definition*)])
  `|` *statement*

The `begin`, `include`, and `include-ci` declarations are used to specify the body of the library. They have the same syntax and semantics as the corresponding expression types. This form of `begin` is analogous to, but not the same as regular `begin`. A plain *statement* (which is allowed as a Kawa extension) is also part of the body of the library, as if it were wrapped in a `begin`).

The `include-library-declarations` declaration is similar to `include` except that the contents of the file are spliced directly into the current library definition. This can be used, for example, to share the same `export` declaration among multiple libraries as a simple form of library interface.

The `cond-expand` declaration has the same syntax and semantics as the `cond-expand` expression type, except that it expands to spliced-in library declarations rather than expressions enclosed in `begin`.

When a library is loaded, its expressions are executed in textual order. If a library's definitions are referenced in the expanded form of a program or library body, then that library must be loaded before the expanded program or library body is evaluated. This rule applies transitively. If a library is imported by more than one program or library, it may possibly be loaded additional times.

Similarly, during the expansion of a library `(foo)`, if any syntax keywords imported from another library `(bar)` are needed to expand the library, then the library `(bar)` must be expanded and its syntax definitions evaluated before the expansion of `(foo)`.

Regardless of the number of times that a library is loaded, each program or library that imports bindings from a library must do so from a single loading of that library, regardless of the number of import declarations in which it appears. That is, `(import (only (foo) a))` followed by `(import (only (foo) b))` has the same effect as `(import (only (foo) a b))`.

19.5.3 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`, 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 stripped 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]
`module-name <`*name*`>` [Syntax]
`module-name` *`library-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.

If the form *library-name* is used, then the class name is the result of taking each *identifier* in the *library-name-parts*, Section 19.12 [Mangling], page 330, if needed, and concatenating them separated by periods. For example `(org example doc-utils)` becomes `org.example.doc-utils`. (You can't reference the class name `doc-utils` directly in Java, but the JVM has no problems with it. In Java you can use reflection to access classes with such names.)

As a matter of style, `module-name` should be the first command in a file (after possible comments). It must appear before a `require` or `import`, in case of cycles.

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.Location` object, and the value Scheme variable is defined to be the value stored in the `Location`. However, if you specify an explicit type, then the field will have the specified type, instead of being a `Location`. The indirection using `Location` 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 6.5.5 [Applet compilation], page 104) 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.

Unfortunately, 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 19.12 [Mangling], page 330) 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-generated field types implement the `Named` interface.

19.5.4 Same class for module and defined class

You can declare a class using `define-simple-class` with the same name as the module class, for example the following in a file named `foo.scm`:

```
(define-simple-class foo ...)
```

In this case the defined class will serve dual-purpose as the module class.

To avoid confusion, in this case you must not specify `module-extends`, `module-implements`, or `(module-static #t)`. Also, the defined class should not have public static members. In that case it works out pretty well: public static members represent bindings exported by the module; other non-private members "belong" to the defined class.

In this case `(module-static 'init-run)` is implied.

19.5.5 Static vs non-static modules

There are two kinds of module class: A *static module* is a class (or gets compiled to a class) all of whose public fields are 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.

You can control whether a module is compiled to a static or a non-static class using either a command-line flag to the compiler, or using the `module-static` special form.

`--module-static`
: Generate a static module (as if `(module-static #t)` were specified). This is (now) the default.

`--module-nonstatic`
`--no-module-static`
: Generate a non-static module (as if `(module-static #f)` were specified). This used to be the default.

`--module-static-run`
: Generate a static module (as if `(module-static 'init-run)` were specified).

`module-static` *name ...* [Syntax]
`module-static #t` [Syntax]
`module-static #f` [Syntax]
`module-static 'init-run` [Syntax]

Control whether the generated fields and methods are static. If `#t` or `'init-run` is specified, then the module will be a static module, *all* definitions will be static. If `'init-run` is specified, in addition the module body is evaluated in the class's static initializer. (Otherwise, it is run the first time it is `require`'d.) Otherwise, the module is an instance module. If there is a non-empty list of *name*s then the module is an instance module, but 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:

1. If there is a `module-extends` or `module-implements` declaration, or one of the `--applet` or `--servlet` command-line flags was specified, then `(module-static #f)` is implied.

2. If one of the command-line flags `--no-module-static`, `--module-nonstatic`, `--module-static`, or `--module-static-run` was specified, then the default is `#f`, `#f`, `#t`, or `'init-run`, respectively.
3. If the module class is [dual-purpose-class], page 309, then `(module-static 'init-run)` is implied.
4. Otherwise the default is `(module-static #t)`. (It used to be `(module-static #f)` in older Kawa versions.)

The default is `(module-static #t)`. It usually produces more efficient code, and is recommended if a module contains only procedure or macro definitions. 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.

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.)

19.5.6 Module options

Certain compilation options can be be specified *either* on the command-line when compiling, or in the module itself.

`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 supported option names (The ending colon makes it a Kawa keyword). Valid option keys are:

- **main:** - Generate an application, with a main method.
- **full-tailcalls:** - Use a calling convention that supports proper tail recursion.
- **warn-undefined-variable:** - Warn if no compiler-visible binding for a variable.
- **warn-unknown-member:** - Warn if referencing an unknown method or field.
- **warn-invoke-unknown-method:** - Warn if invoke calls an unknown method (subsumed by warn-unknown-member).
- **warn-unused:** - Warn if a variable is usused or code never executed.
- **warn-uninitialized:** - Warn if accessing an uninitialized variable.
- **warn-unreachable:** - Warn if this code can never be executed.
- **warn-void-used:** - Warn if an expression depends on the value of a void sub-expression (one that never returns a value).
- **warn-as-error:** - Treat a compilation warning as if it were an error.

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*.

The module option key `main:` has no effect when applied to a particular body via the `with-compile-options` syntax.

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

19.6 Importing from a library

You can import a module into the current namespace with `import` or `require`. This adds the exported bindings (or a subset of them) to the current lexical scope. It follows that these bindings (which are said to be imported) are determined at compile-time.

`import` *`import-set`** [Syntax]

An `import` declaration provides a way to import identifiers exported by a library (module). Each *import-set* names a set of bindings from a library and possibly specifies local names for the imported bindings.

import-set `::=`
  classname
 | *library-reference*
 | (**library** *library-reference*)
 | (**class** *class-prefix* import-only-name*)
 | (**only** *import-set* import-only-name*)
 | (**except** *import-set* identifier*)
 | (**prefix** *import-set identifier*)
 | (**rename** *import-set* *rename-pair* $*$)
library-reference `::=` (*library-name-parts* [*explicit-source-name*])
import-only-name `::=` *identifier* | *rename-pair*
explicit-source-name `::=` *string*
rename-pair `::=` (*identifier*$_1$ *identifier*$_2$)

A *library-reference* is mapped to a class name by concatenating all the identifiers, separated by dots. For example:

```
(import (gnu kawa slib srfi37))
```

is equivalent to:

```
(import gnu.kawa.slib.srfi37)
```

as well as to:

```
(require gnu.kawa.slib.srfi37)
```

By default, all of an imported library's exported bindings are made visible within an importing library using the names given to the bindings by the imported library. The precise set of bindings to be imported and the names of those bindings can be adjusted with the `only`, `except`, `prefix`, and `rename` forms as described below.

- An `only` form produces a subset of the bindings from another *import-set*, including only the listed *identifier*s. The included *identifier*s must be in the original *import-set*. If a *rename-pair* is used, then the `identifier1` must be in the original *import-set*, and is renamed to `identifier2`. For example:

```
(import (only (kawa example) A (B1 B2) C (D1 D2)))
```

is equivalent to:

```
(import (rename (only (kawa example) A B1 C D1)
                (B1 B2) (D1 D2)))
```

The names `A`, `B1`, `C`, and `D1` must exist in the library `(kawa example)`. The bindings are accessible using the names `A`, `B2`, `C`, and `D2`.

- An `except` form produces a subset of the bindings from another *import-set*, including all but the listed *identifier*s. All of the excluded *identifier*s must be in the original *import-set*.
- A `prefix` form adds the *identifier* prefix to each name from another *import-set*.
- A `rename` form:

  ```
  (rename (identifier1 identifier2) ...)
  ```

  removes the bindings for `identifier1 ...` to form an intermediate *import-set*, then adds the bindings back for the corresponding `identifier2 ...` to form the final *import-set*. Each `identifier1` must be in the original *import-set*, each *identifier*$_2$ must not be in the intermediate *import-set*, and the *identifier*$_2$s must be distinct.

A `class` form is a convenient way to define abbreviations for class names; it may be more convenient than `define-alias`. The *class-prefix* is concatenated with each *identifier* (with a period in between) to produce a classname. Each *identifier* becomes an alias for the class. For example:

```
(import (class java.util Map (HashMap HMap)))
```

This defines `Map` as an alias for `java.util.Map`, and `HMap` as an alias for `java.util.HashMap`. (You can think of the `class` form as similar to a `only` form, where the *class-prefix* names a special kind of library represented of a Java package, and whose exported bindings are the classes in the package.)

You can combine the `class` form with `only`, `except`, `rename`, and `prefix`, though only `prefix` is likely to be useful. For example:

```
(import (prefix (class java.lang Long Short) jl-))
```

is equivalent to

```
(import (class java.lang (Long jl-Long) (Short jl-Short)))
```

which is equivalent to:

```
(define-private-alias jl-Short java.lang.Short)
(define-private-alias jl-Long java.lang.Long)
```

`require` **fl***featureName* [Syntax]
`require` *classname* [*`explicit-source-name`*] [Syntax]
`require` *`explicit-source-name`*] [Syntax]

Search for a matching module (class), and add the names exported by that module to the current set of visible names. Normally, the module is specified using *classname*.

The form `require` has similar functionality as `import`, but with a different syntax, and without options like `rename`.

If a `"`*`sourcepath`*`"` is specified then that is used to locate the source file for the module, and if necessary, compile it.

If a `'`*`featurename`* is specified then the *featurename* is looked up (at compile time) in the "feature table" which yields the implementing *classname*.

`provide` fl*featurename* [Syntax]

Declare that `'`*`featurename`* is available. A following `cond-expand` in this scope will match *featurename*.

Using `require` and `provide` with *featurename*s is similar to the same-named macros in SLib, Emacs, and Common Lisp. However, in Kawa these are not functions, but instead they are syntax forms that are processed at compile time. That is why only quoted *featurename*s are supported. This is consistent with Kawa emphasis on compilation and static binding.

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

19.6.1 Searching for modules

When Kawa sees a `import` or `require` it searches for either a matching source file or a previously-compiled class with a matching name.

For `import` we generate a classname by converting it in the same way `module-name` does: taking each identifier in the *library-name-parts*, mangling if needed, and concatenating the parts separated by periods.

If there is a matching module in any *program-unit* that is in the process of being compiled, we use that. This may be a file requested to be compiled with the `-C` command-line switch, or an extra *library-definition* in a file already parsed. Kawa will attempt to finish compiling the module and load the class, but if there are circular dependencies it will use the uncompiled definitions.

Next Kawa looks for a matching class in the context classpath. (There is special handling if the library-name starts with `srfi`, and certain builtin classes will have `kawa.lib.` prepended.)

Kawa also searches for a matching source file, described below. It uses the implicit source name (formed by concatenating the library-name parts, separated by `"/"`), as well as any *explicit-source-name*. The source file is parsed as a *program-unit*. It is an error if the *program-unit* does not declare a library (explicit or implicit) with the matching name.

If Kawa finds both a matching source file and a class, it will pick one based on which is newer.

19.6.2 Searching for source files

The Java property `kawa.import.path` controls how `import` and `require` search for a suitable source file. Example usage:

```
$ kawa -Dkawa.import.path=".:<foo fo>/opt/fo-libs/*.scm:/usr/local/kawa"
```

The value of the `kawa.import.path` property is a list of path elements, separated by `":"`. Each path element is combined with either the explicit source name or the implicit source name to produce a filename. If a matching file exists, then we have found a source file.

If a path element contains a `"*"` then the `"*"` is replaced by the implicit source name (without an extension). (Any explicit source name is ignored in this case.) For example, for `(import (foo bar))` or `(require foo.bar)` the implicit source name is `"foo/bar"`. If the path element is `"/opt/kawa/*.sc"` then the resulting filename is `"/opt/kawa/foo/bar.sc"`.

If there is no `"*"` in the path element, and there is an explicit source, then it is appended to the path element (or replaces the path element if the explicit source is absolute). Otherwise we use the implicit source, followed by the default file extension. (The default file extension is that of the current source if that is a named file; otherwise the default for the current language, which is `".scm"` for Scheme.)

A path element that starts with a selector of the form `"<`*`library-name-parts`*`>"` is only applicable if a prefix of the requested module name matches the *library-name-parts*. If there is `"*"` in the path element, that is replaced by the corresponding rest of the implicit source name. For example if importing `(fee fo foo fum)` and the path element is `"<fee fo>/opt/fo-libs/*.scm"` then the resulting filename is `"/opt/fo-libs/foo/fum.scm"`. If there is a selector but no `"*"`, then the rest of the path element following the selector is combined with the explicit or implicit source as if there were no selector (assuming of course that the selector matches).

If the resulting filename is relative, then it is resolved relative to the *current root*. For example the source to a library with the name `(x y)` that compiles to a class `x.y` might be a file named `/a/b/x/y.scm`. Then the current root would be `/a/b/` - that is the directory that results from removing the library name suffix from the file name.

More generally: assume the current module has N name components. For example the name `(x y)` (with the class name `x.y`) has 2 components. The current root is what you get when you take the current file name (say `"/a/b/c/d.scm"`), and remove everything after the N'th slash (`"/"`) from the end (say `"c/d.scm"`; what remains (e.g. `"/a/b/"` is the current root. (If the current input source is not a named file, use the value of `(current-path)` with a `"/"` appended.)

The default search path is `"."` - i.e. just search relative to the current root.

19.6.3 Builtin libraries

The following libraries are bundled with Kawa:

`(scheme base)`
`(scheme case-lambda)`
`(scheme char)`
`(scheme complex)`
`(scheme cxr)`
`(scheme cxr)`
`(scheme eval)`
`(scheme inexact)`
`(scheme lazy)`
`(scheme load)`
`(scheme process-context)`
`(scheme read)`
`(scheme repl)`
`(scheme time)`
`(scheme write)`
`(scheme r5rs)`

The above are standard libraries as defined by R7RS.

`(rnrs arithmetic bitwise)`
`(rnrs hashtables)`
`(rnrs lists)`
`(rnrs programs)`
`(rnrs sorting)`
`(rnrs unicode)`

The above are standard libraries as defined by R6RS.

`(kawa reflect)`

Defines procedures and syntax for acessing Java objects and members: `as` `field` `instance?` `invoke` `invoke-static` `invoke-special` `make` `primitive-throw` `set-field!` `set-static-field!` `static-field`

`(kawa expressions)`
`(kawa hashtable)`
`(kawa quaternions)`
`(kawa rotations)`
`(kawa regex)`
`(kawa string-cursors)`

Various Kawa libraries *add details*.

`(kawa base)`

All the bindings by default available to the kawa top-level.

19.6.4 Importing a SRFI library

Importing a supported SRFI numbered *N* is conventionally doing using a `(import (srfi N))` or the older R6RS syntax `(import (srfi :N))` (with a colon, for historical reasons). You can also give it a name, as specified by SRFI 95 (`http://srfi.schemers.org/srfi-95/srfi-95.html`). For example, any of these work:

```
(import (srfi 95))
(import (srfi 95 sorting-and-merging))
```

```
(import (srfi :95))
(import (srfi :95 sorting-and-merging))
```

You can also use `(require 'srfi-N)`:

```
(require 'srfi-95)
```

19.6.5 Importing from a plain class

Note you can import from many classes, even if they weren't compiled from a library-definition. The set of `public` fields in a class are considered as the set of exported definitions, with the names demangled as needed.

The module can be static module (all public fields must be static), or an instance module (it has a public default constructor).

If an imported definition is a non-static field 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 imported fields of the module class are then incorporated in the current set of local visible names in the current module. (This is for both instance and static modules.) 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).

19.7 Record types

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-9.html`), 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
```

Kawa compiles the record type into a nested class. If the `define-record-type` appears at module level, the result is a class that is a member of the module class. For example if the above `pare` class is define in a module `parelib`, then the result is a class named `pare` with the internal JVM name `parelib$pare`. The `define-record-type` can appear inside a procedure, in which case the result is an inner class.

The nested class has 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 resulting functions, without needing to use run-time reflection.

19.8 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 Section 19.7 [Record types], page 317).

`make-record-type` *type-name field-names* [Procedure]

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*] [Procedure]

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* [Procedure]

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* [Procedure]

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* [Procedure]

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* [Procedure]

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

`record-type-descriptor` *record* [Procedure]

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* [Procedure]

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* [Procedure]

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.

19.9 Calling Java methods from Scheme

You can call a Java method as if it were a Scheme procedure using various mechanisms.

19.9.1 Calling static methods using colon notation

The easiest way to invoke a static method is to use Section 7.7 [Colon notation], page 115, specifically:

(*class-expression*:*method-name argument* ...)

The *class-expression* can be a class in the current lexical scope, such as a class defined using `define-simple-class`:

```
(define-simple-class MyClass ()
  ((add2 x y) allocation: 'static (+ x y)))
(MyClass:add2 3 4) ⇒ 7
```

Often *class-expression* is a fully-qualified class name:

```
(java.lang.Math:sqrt 9.0) ⇒ 3.0
```

This is only allowed when the name is of a class that exists and is accessible both at compile-time and run-time, and the name is not otherwise lexically bound.

You can also use a defined alias:

```
(define-alias jlMath java.lang.Math)
(jlMath:sqrt 16.0) ⇒ 4.0
```

You can even evaluate *class-expression* at run-time (in which case Kawa may have to use slower reflection):

```
(let ((math java.lang.Math)) (math:sqrt 9.0)) ⇒ 3.0
```

Here `java.lang.Math` evaluates to a `java.lang.Class` instance for the named class (like Java's `java.lang.Class.class`, again assuming the class exists and is accessible both at compile-time and run-time, and the name is not otherwise lexically bound.

19.9.2 Calling instance methods using colon notation

The syntax is:

(*instance*:*method-name argument* ...)

This invokes the method named *method-name* with the evaluated *instance* as the target object and the evaluated *arguments* as the method arguments.

For example:

```
((list 9 8 7):toString) ⇒ "(9 8 7)"
([5 6 7]:get 2) ⇒ 7
```

This older syntax is also available:

(*:*method-name instance argument* ...**)**

For example:

```
(*:toString (list 9 8 7))
```

You can also name the class explicitly:

(*class-expression***:***method-name instance argument* ...**)**

For example:

```
(java.util.List:get [5 6 7] 2) ⇒ 7
```

Using an explicit class is like coercing the *instance*:

(*:*method-name* **(as** *class-expression instance* **)***argument* ...**)**

Note that for some special values, including `java.lang.Class` instances, you can't use the compact form of Section 7.7 [Colon notation], page 115, where the *instance* is before the comma:

```
(java.lang.Integer:getDeclaredField "MAX_VALUE") ⇒ error
```

This is because in this case we look for a static member of `java.lang.Integer` (at least as currently defined and implemented), while we want an instance member of `java.lang.Class`. In those cases you can use one of these alternative forms, which all return the same `java.lang.reflect.Field` result:

```
(*:getDeclaredField java.lang.Integer "MAX_VALUE")
(java.lang.Class:getDeclaredField java.lang.Integer "MAX_VALUE")
(invoke java.lang.Integer 'getDeclaredField "MAX_VALUE")
```

19.9.3 Method names

The method to invoke is selected using the specified method name and argments. If specified name is not a Java name, it is "mangled" (see Section 19.12 [Mangling], page 330) into a valid Java name. All accessible methods whose names match are considered. Methods that match after appending `$V` or `$X` or `$V$X` are also considered. A `$V` suffix matches a variable number of arguments: any excess arguments are collect into an `gnu.lists.LList` or a Java array (depending on the final parameter type). A `$X` specifies that the method expects an extra implicit `CallContext` parameter. In that case the method's result is written to the `CallContext`, so the method result type must be `void`.

(Kawa may compile a procedure with a `#!rest` or keyword args whose name is *`fn`* to a method named *`fn`*`$V`. It adds an implicit parameter for the extra arguments. By default this extra extra parameter is a Scheme list. You can specify a Java array type instead, in which case the method is named *`fn`* without the `$V`, and instead it is marked as a Java-5 varargs method. The array element type must be compatible with all the extra arguments.)

19.9.4 Invoking a method with the `invoke` function

If you prefer, you can instead use the following functions. (There is also an older deprecated lower-level interface (see [Low-level Method invocation], page 384.)

`invoke-static` *class name args ...* [Procedure]

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.

Any accessible methods (static or instance) in the specified *class* (or its super-classes) 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.

An example:

```
(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 ...* [Procedure]

The *name* can be `<symbol>` or `<string>` that names one or more methods in the Java class.

Any accessible methods (static or instance) in the specified *class* (or its super-classes) 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.

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 ...* [Procedure]

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.

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* [Procedure]

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.

19.9.5 Using a namespace prefix

This way of invoking a method is deprecated.

You can 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.

Instead of a *vamespace-uri* you can use a variable that names a class, usually of the form `<classname>`. The following is equivalent to the above:

```
(define-namespace Int32 <java.lang.Integer>)
```

However, there is one important difference: The `<classname>` is first searched in the lexical scope. It may resolve to a class defined in the current compilation unit (perhaps defined using `define-simple-class`), or imported from another module, or an alias (such as from `define-alias`). Only if `<classname>` is *not* found in the current scope is it tried as the class name *classname*.

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 String `"ff"`.

The general syntax is

```
(prefix:method-name arg ...)
```

This invokes the method named *method-name* in the class corresponding to *prefix*, and the *arg*s are the method arguments.

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")
```

You can also call instance methods using a namespace prefix:

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

This returns the `double` value `255.0`.

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 a known class, then Kawa will automatically 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.

As a final shorthand you can use an identifier in handle brackets, such as an existing type alias like `<list>`. The following are all equivalent:

```
(<list>:list3 'a 'b 'c)
```

This is equivalent to:

```
(define-namespace prefix <list>)
(prefix:list3 'a 'b 'c)
```

for some otherwise-unused *prefix*.

19.10 Allocating objects

The recommended way to create an instance of a type *T* is to “call” *T* as if it were a function, with the arguments used to initialize the object. If `T` is a class and `T` has a matching constructor, then the arguments will used for constructor arguments:

```
(java.util.StringTokenizer "this/is/a/test" "/")
```

(You can think of the type *T* as being coerced to an instance-constructor function.)

If `T` is a container or collection type, then typically the arguments will be used to specify the child or component values. Many standard Scheme procedures fit this convention. For example in Kawa `list` and `vector` evaluate to types, rather than procedures as in standard Scheme, but because types can be used as constructor functions it just works:

```
(list 'a (+ 3 4) 'c) ⇒ (a 7 c)
(vector 'a 'b 'c) ⇒ #(a b c)
```

Any class `T` that has a default constructor and an `add` method can be initialized this way. Examples are `java.util` collection classes, and `jawa.awt` and `javax.swing` containers.

```
(java.util.ArrayList 11 22 33) ⇒ [11, 22, 333]
```

The above expression is equivalent to:

```
(let ((tmp (java.util.ArrayList)))
  (tmp:add 11)
  (tmp:add 22)
  (tmp:add 33)
  tmp)
```

Allocating Java arrays (see [Creating-new-Java-arrays], page 331) uses a similar pattern:

```
(int[] 2 3 5 7 11)
```

Sometimes you want to set some named property to an initial value. You can do that using a keyword argument. For example:

```
(javax.swing.JButton text: "Do it!" tool-tip-text: "do it")
```

This is equivalent to using *setter methods*:

```
(let ((tmp (javax.swing.JButton)))
  (tmp:setText "Do it!")
  (tmp:setToolTipText "do it")
  tmp)
```

A keyword argument ***key-name*:** can can translated to either a **set*KeyName*:** or a **add*KeyName*:** method. The latter makes it convenient to add listeners:

```
(javax.swing.JButton
  text: "Do it!"
  action-listener:
   (object (java.awt.event.ActionListener)
     ((actionPerformed e) (do-the-action))))
```

This is equivalent to:

```
(let ((tmp (javax.swing.JButton)))
  (tmp:setText "Do it!")
  (tmp:addActionListener
    (object (java.awt.event.ActionListener)
      ((actionPerformed e) (do-the-action))))
  tmp)
```

Making use of so-called “SAM-conversion” (see [SAM-conversion], page 303) makes it even more convenient:

```
(javax.swing.JButton
  text: "Do it!"
  action-listener:
   (lambda (e) (do-the-action)))
```

The general case allows for a mix of constructor arguments, property keywords, and child values:

class-type constructor-value... property-initializer... child-value...
constructor-value `::=` *expression*
property-initializer `::=` *keyword expression*
child-value `::=` *expression*

First an object is constructed with the *constructor-value* arguments (if any) passed to the object constructor; then named properties (if any) are used to initialize named properties; and then remaining arguments are used to add child values.

There is an ambiguity if there is no *property-initializer* - we can't distinguish between a *constructor-value* and a *child-value*. In that case, if there is a matching constructor method, then all of the arguments are constructor arguments; otherwise, there must a default constructor, and all of the arguments are *child-value* arguments.

There is a trick you can you if you need both *constructor-value* and *child-value* arguments: separate them with an "empty keyword" `||:`. This matches a method named `add`, which means that the next argument effectively a *child-value* - as do all the remaining arguments. Example:

```
(let ((vec #(1 2 3)))
  (java.util.ArrayList vec ||: 4 5 6))
  ⇒ [1, 2, 3, 4, 5, 6]
```

The compiler rewrites these allocations expression to generated efficient bytecode, assuming that the "function" being applied is a type known by the compiler. Most of the above expressions also work if the type is applied at run-time, in which case Kawa has to use slower reflection:

```
(define iarr int[])
(apply iarr (list 3 4 5)) ⇒ [3 4 5]
```

However `add`*`Xxx`* methods and SAM-conversion are currently only recognized in the case of a class known at compile-time, not at run-time.

Here is a working Swing demo illustrating many of these techniques:

```
(import (class javax.swing
               JButton Box JFrame))
(define-simple-class HBox (Box)
  ((*init*) (invoke-special Box (this) '*init* 0)))

(define value 0)

(define txt
  (javax.swing.JLabel
   text: "0"))

(define (set-value i)
  (set! value i)
  (set! txt:text (number->string i)))

(define fr
  (JFrame
     title: "Hello!"
     (Box 1#|VERTICAL|# ||:
      (javax.swing.Box:createGlue)
      txt
      (javax.swing.Box:createGlue)
```

```
          (HBox
           (JButton ;; uses 1-argument constructor
    "Decrement" ;; constructor argument
    tool-tip-text: "decrement"
    action-listener: (lambda (e) (set-value (- value 1))))
           (javax.swing.Box:createGlue)
           (JButton ;; uses 0-argument constructor
    text: "Increment"
    tool-tip-text: "increment"
    action-listener: (lambda (e) (set-value (+ value 1))))))))
    (fr:setSize 200 100)
    (set! fr:visible #t)
```

If you prefer, you can use the older `make` special function:

`make` *type args ...* [Procedure]

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

```
type args ...
```

Another (semi-deprecated) function is to use the colon notation with the `new` pseudo-function. The following three are all equivalent:

```
(java.awt.Point:new x: 4 y: 3)
(make java.awt.Point: x: 4 y: 3)
(java.awt.Point x: 4 y: 3)
```

19.11 Accessing object fields

19.11.1 Accessing static fields and properties

The recommmended way to access fields uses the Section 7.7 [Colon notation], page 115. For static fields and properties the following is recommended:

*class-expression***:***field-name*

For example:

```
java.lang.Integer:MAX_VALUE
```

A property with a `get` method is equivalent to a field. The following are all equivalent:

```
java.util.Currency:available-currencies
java.util.Currency:availableCurrencies
(java.util.Currency:getAvailableCurrencies)
```

Just like for a method call, the *class-expression* can be a class in the current lexical scope, a fully-qualified class name, or more generally an expression that evaluates to a class.

19.11.2 Accessing instance fields and properties

The syntax is:

*instance***:***field-name*

The *field-name* can of course be the name of an actual object field, but it can also be the name of a property with a zero-argument `get` method. For example, if `cal` is a `java.util-Calendar` instance, then the following are all equivalent:

```
cal:time-zone
cal:timeZone
(cal:getTimeZone)
(cal:get-time-zone)
```

You can use colon notation to assign to a field:

```
(set! cal:time-zone TimeZone:default)
```

which is equivalent to:

```
(cal:setTimeZone (TimeZone:getDefault))
```

A Java array only has the `length` field, plus the `class` property:

```
(int[] 4 5 6):length ⇒ 3
(int[] 4 5 6):class:name ⇒ "int[]"
```

19.11.3 Using field and static-field methods

The following methods are useful in cases where colon notation is ambiguous, for example where there are both fields and methods with the same name. You might also prefer as a matter of style, to emphasise that a field is being accessed.

`field` *object fieldname* [Procedure]

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

The field name is "mangled" (see Section 19.12 [Mangling], page 330) 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 `"getFieldname"` (or `"isFieldname"` for a boolean property).

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* [Procedure]

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

If the *fieldname* is the special name `class`, then it returns the `java.lang.Class` object corresponding to *class* (which is usually a `gnu.bytecode.ClassType` object).

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* [Procedure]

A synonym for `(field object fieldname)`.

`slot-set!` *object fieldname value* [Procedure]

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

19.11.4 Older colon-dot notation

There is older syntax where following the colon there is field name a following the colon *and* a period.

To access an static field named *field-name* use this syntax

```
(prefix:.field-name instance)
```

The *prefix* can be as discussed in See Section 19.9 [Method operations], page 320. Here are 5 equivalent ways:

```
(java.lang.Integer:.MAX_VALUE)
(<java.lang.Integer>:.MAX_VALUE)
(define-namespace Int32 <java.lang.Integer>)
(Int32:.MAX_VALUE)
(define-namespace Integer "class:java.lang.Integer")
(Integer:.MAX_VALUE)
(define-alias j.l.Integer java.lang.Integer)
(j.l.Integer:.MAX_VALUE)
```

You can set a static field using this syntax:

```
(set! (prefix:.field-name) new-value)
```

The special field name `class` can be used to extract the `java.lang.Class` object for a class-type. For example:

```
(java.util.Vector:.class) ⇒ class java.util.Vector
```

To access a instance field named *field-name* use the following syntax. Note the period before the *field-name*.

```
(*:.field-name instance)
```

This syntax works with `set!` - to set the field use this syntax:

```
(set! (*:.field-name instance) new-value)
```

Here is an example:

```
(define p (list 3 4 5))
(*:.cdr p) ⇒ (4 5)
(set! (*:.cdr p) (list 6 7))
p ⇒ (3 6 7)
```

You can specify an explicit class:

```
(prefix:.field-name instance)
```

If *prefix* is bound to `<class>`, then the above is equivalent to:

```
(*:.field-name (as <class> instance))
```

19.12 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. Traditionally, 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. Names in the Java language can contain upper- and lower-case letters, digits, and the special symbols '`_`' and '`$`'. The Java virtual machine (JVM) allows most characters, but still has some limitations.

Kawa translates class names, package names, field names, and local variable names using the "symbolic" convention (`https://blogs.oracle.com/jrose/entry/symbolic_freedom_in_the_vm`), so most characters are unchanged. For example the Scheme function '`file-exists?`' becomes the field '`file-exists?`', but `dotted.name` becomes '`dotted\,name`'. Such names may not be valid Java name, so to access them from a Java program you might have to use reflection.

When translating procedure names to method names, Kawa uses a different translation, in order to achieve more "Java-like" names. This means translating a Scheme-style name like '`make-temporary-file`' to "mixed-case" words, such as '`makeTemporaryFile`'. 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 '`$Ls`'. See the source code to the `mangleName` method in the `gnu.expr.Mangling` class for the full list. Characters that do not have a mnemonic abbreviation are mangled as '`$`' 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.

19.13 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`.

19.14 Using Java Arrays

19.14.1 Creating new Java arrays

To allocate a Java array you can use the array type specifier as a constructor function. For example, to allocate an array with room for 10 elements each of each is a primitive `int`:

```
(int[] length: 10)
```

You can specify the initial elements instead of the length:

```
(object[] 31 32 33 34)
```

This creates a 4-length array, initialized to the given values.

Note this is a variation of the generation object-allocation (see Section 19.10 [Allocating objects], page 324) pattern. You can explicitly use the `make` function, if you prefer:

```
(make object[] 31 32 33 34)
```

If you specify a length, you can also specify initial values for selected elements. If you specify an index, in the form of a literal integer-valued keyword, then following elements are placed starting at that position.

```
(int[] length: 100 10 12 80: 15 16 50: 13 14)
```

This creates an array with 100 elements. Most of them are initialized to the default value of zero, but elements with indexes 0, 1, 50, 51, 80, 81 are initialized to the values 10, 12, 13, 14, 15, 16, respectively.

19.14.2 Accessing Java array elements

You can access the elements of a Java array by treating it as a one-argument function, where the argument is the index:

```
(define primes (integer[] 2 3 5 7 11 13))
(primes 0) ⇒ 2
(primes 5) ⇒ 13
```

You can set an element by treating the array as a function with a `setter`:

```
(set! (primes 0) -2)
(set! (primes 3) -7)
primes ⇒ [-2 3 5 -7 11 13]
```

To get the number of elements of an array, you can treat it as having a `length` field:

```
primes:length ⇒ 6
```

Here is a longer example. This is the actual definition of the standard `gcd` function. Note the `args` variable receives all the arguments on the form of an `integer` array. (This uses the Java5 varargs feature.)

```
(define (gcd #!rest (args ::integer[])) ::integer
  (let ((n ::int args:length))
    (if (= n 0)
0
(let ((result ::integer (args 0)))
  (do ((i ::int 1 (+ i 1)))
      ((>= i n) result)
    (set! result (gnu.math.IntNum:gcd result (args i))))))))
```

The above example generates good code, thanks to judicious use of casts and type specifications. In general, if Kawa knows that a “function” is an array then it will generate efficient bytecode instructions for array operations.

19.14.3 Old low-level array macros

The deprecated [Low-level array macros], page 385, are also supported.

19.15 Loading Java functions into Scheme

When `kawa -C` compiles (see Section 6.5.1 [Files compilation], page 101) 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);
}
```

19.16 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 do 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);
```

19.16.1 Using `javax.script` portable Java scripting

Kawa also supports the standard `javax.script` (`http://docs.oracle.com/javase/8/docs/api/javax/script/package-summary.html`) API. The main advantage of this API is if you want your users to be able to choose between multiple scripting languages. That way you can support Kawa without Kawa-specific programming.

For example the standard JDK tool jrunscript (`http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jrunscript.html`) provides a read-eval-print-loop for any language that implements the `javax.script` API. It knows nothing about Kawa but can still use it:

```
$ jrunscript -cp kawa.jar -l scheme
scheme> (cadr '(3 4 5))
4
```

(Of course the `jrunscript` REPL isn't as nice as the one that Kawa provides. For example the latter can handle multi-line inputs.)

20 Working with XML and HTML

Kawa has a number of features for working with XML, HTML, and generated web pages.

In Kawa you don't write XML or HTML directly. Instead you write expressions that evaluate to "node objects" corresponding to elements, attributes, and text. You then write these node objects using either an XML or HTML format.

Many web-page-generating tools require you to work directly with raw HTML, as for example:

```
(display "<p>Don't use the <code>&lt;blink&gt;</code> tag.</p>")
```

In Kawa you would instead do:

```
(display (html:p "Don't use the " (html:code "<blink>") " tag."))
```

The conversion from node objects to XML or HTML is handled by the formatter (or serializer). Some advantages of doing it this way are:

- You don't have to worry about quoting special characters. Missing or incorrect quoting is a common source of bugs and security problems on systems that work directly with text, such as PHP.
- Some errors, such as mismatched element tags, are automatically avoided.
- The generated XML can be validated as it is generated, or even using compile-time type-checking. (Kawa doesn't yet do either.)
- In an application that also reads XML, you can treat XML that is read in and XML that is generated using the same functions.

20.1 Formatting XML

The easiest way to generate HTML or XML output is to run Kawa with the appropriate Section 17.1 [`--output-format` option], page 270.

The intentation is that these output modes should be compatible with XSLT 2.0 and XQuery 1.0 Serialization (`http://www.w3.org/TR/2006/PR-xslt-xquery-serialization-20061121/`). (However, that specifies many options, most of which have not yet been implemented.

`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 '`&lt;`').
`xhtml`	Same as `xml`, but follows the xhtml compatibility guidelines.
`html`	Values are printed in HTML format. Mostly same as `xml` format, but certain elements 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.

To illustrate:

```
$ kawa --output-format html
#|kawa:1|# (html:img src:"img.jpg")
```

```
<img src="img.jpg">
$ kawa --output-format xhtml
#|kawa:1|# (html:img src:"img.jpg")
<img xmlns="http://www.w3.org/1999/xhtml" src="img.jpg" />
$ kawa --output-format xml
#|kawa:1|# (html:img src:"img.jpg")
<img xmlns="http://www.w3.org/1999/xhtml" src="img.jpg"></img>
```

And here is the default `scheme` formatting:

```
$ kawa
#|kawa:1|# (html:img src:"img.jpg")
({http://www.w3.org/1999/xhtml}img src: img.jpg )
```

`as-xml` *value* [Procedure]

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>`.

`unescaped-data` *data* [Procedure]

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 ‘`&lt;?xml?&gt;`’, but `(unescaped-data "<?xml?>")` prints as ‘`<?xml?>`’.

20.2 Creating HTML nodes

The `html` prefix names a special namespace (see Section 10.2 [Namespaces], page 161) of functions to create HTML element nodes. For example, `html:em` is a constructor that when called creates a element node whose tag is `em`. If this element node is formatted as HTML, the result has an `<em>` tag.

`html:`*`tag`* *attributes ... content ...* [Syntax]

Creates an element node whose tag is *tag*. The parameters are first zero or more attributes, followed by zero of more child values. An attribute is either an attribute value (possibly created using `make-attribute`), or a pair of arguments: A keyword followed by the attribute value. Child values are usually either strings (text content), or nested element nodes, but can also be comment or processing-instruction nodes.

```
(html:a href: "http://gnu.org/" "the "(html:i "GNU")" homepage")
```

The compound identifier `html:`*`tag`* is actually a type: When you call it as a function you’re using Kawa’s standard coercion of a type to its constructor function. This means you can do type tests:

```
(define some-node ...)
(if (instance? some-node html:blink)
  (error "blinking not allowed!"))
```

Object identity is currently not fully specified. Specifically, it is undefined if a nested (child) element node is copied “by value” or “by reference”. This is related to whether nodes have a parent reference. In the XPath/XQuery data model nodes do have a parent reference, and child nodes are conceptually copied. (In the actual implemention copying is commonly avoided.) Kawa/Scheme currently followed the XQuery copying semantics, which may not be the most appropriate for Scheme.

20.3 Creating XML nodes

The XML data model is similar to HTML, with one important addition: XML tags may be *qualified names*, which are similar to Section 10.2 [compound symbols], page 161.

You must do this to use the following types and functions:

```
(require 'xml)
```

The following types and functions assume:

```
(require 'xml)
```

`make-element` *tag* [*attribute ...*] *child ...* [Procedure]

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>)))
```

`element-name` *element* [Procedure]

Returns the name (tag) of the element node, as a symbol (QName).

`make-attribute` *name value...* [Procedure]

Create an "attribute", which is a name-value pair. For now, *name* should be a symbol.

`attribute-name` *element* [Procedure]

Returns the name of the attribute node, as a symbol (QName).

`comment` [Type]

Instances of this type represent comment values, specifically including comments in XML files. Comment nodes are currently ignored when printing using Scheme formatting, though that may change.

`comment` *comment-text* [Constructor]

Create a comment object with the specified *comment-text*.

`processing-instruction` [Type]

Instances of this type represent “processing instructions”, such as may appear in XML files. Processing-instruction nodes are currently ignored when printing using Scheme formatting, though that may change.

`processing-instruction` *target contents* [Constructor]

> Crreate a processing-instruction object with the specified *target* (a simple symbol) and *contents* (a string).

20.4 XML literals

You can write XML literals directly in Scheme code, following a `#`. Notice that the outermost element needs to be prefixed by `#`, but nested elements do not (and must not).

```
#<p>The result is <b>final</b>!</p>
```

Actually, these are not really literals since they can contain enclosed expressions:

```
#<em>The result is &{result}.</em>
```

The value of *result* is substituted into the output, in a similar way to quasi-quotation. (If you try to quote one of these "XML literals", what you get is unspecified and is subject to change.)

An *xml-literal* is usually an element constructor, but there some rarely used forms (processing-instructions, comments, and CDATA section) we'll cover later.

xml-literal `::=` #*xml-constructor*
xml-constructor `::=` *xml-element-constructor*
  `|` *xml-PI-constructor*
  `|` *xml-comment-constructor*
  `|` *xml-CDATA-constructor*

20.4.1 Element constructors

xml-element-constructor `::=`
  `<`*QName xml-attribute**`>`*xml-element-datum...*`</`*QName* `>`
  `|` `<`*xml-name-form xml-attribute**`>`*xml-element-datum...*`</>`
  `|` `<`*xml-name-form xml-attribute**`/>`
xml-name-form `::=` *QName*
  `|` *xml-enclosed-expression*
xml-enclosed-expression `::=`
  { *expression*}
  `|` (*expression...*)

The first *xml-element-constructor* variant uses a literal *QName*, and looks like standard non-empty XML element, where the starting *QName* and the ending *QName* must match exactly:

```
#<a href="next.html">Next</a>
```

As a convenience, you can leave out the ending tag(s):

```
This is a paragraph in <emphasis>DocBook</> syntax.</>
```

You can use an expression to compute the element tag at runtime - in that case you *must* leave out the ending tag:

```
#<p>This is <(if be-bold 'strong 'em)>important</>!</p>
```

You can use arbitrary *expression* inside curly braces, as long as it evaluates to a symbol. You can leave out the curly braces if the *expression* is a simple parenthesised compound expression. The previous example is equivalent to:

```
#<p>This is <{(if be-bold 'strong 'em)}>important</>!</p>
```

The third *xml-element-constructor* variant above is an XML “empty element”; it is equivalent to the second variant when there are no *xml-element-datum* items.

(Note that every well-formed XML element, as defined in the XML specifications, is a valid *xml-element-constructor*, but not vice versa.)

20.4.2 Elements contents (children)

The “contents” (children) of an element are a sequence of character (text) data, and nested nodes. The characters `&`, `<`, and `>` are special, and need to be escaped.

```
xml-element-datum ::=
    any character except &, or <.
  | xml-constructor
  | xml-escaped
xml-escaped ::=
    &xml-enclosed-expression
  | &xml-entity-name;
  | xml-character-reference
xml-character-reference ::=
    &#digit+;
  | &#xhex-digit+;
```

Here is an example shows both hex and decimal character references:

```
#<p>A&#66;C&#x44;E</p>  ⇒  <p>ABCDE</p>
```

xml-entity-name `::=` *identifier*

Currently, the only supported values for *xml-entity-name* are the builtin XML names `lt`, `gt`, `amp`, `quot`, and `apos`, which stand for the characters `<`, `>`, `&`, `"`, and `'`, respectively. The following two expressions are equivalent:

```
#<p>&lt; &gt; &amp; &quot; &apos;</p>
#<p>&{"< > & \" '"}</p>
```

20.4.3 Attributes

```
xml-attribute ::=
    xml-name-form=xml-attribute-value
xml-attribute-value ::=
    "quot-attribute-datum*"
  | ﬂapos-attribute-datum*ﬂ
quot-attribute-datum ::=
    any character except ", &, or <.
  | xml-escaped
apos-attribute-datum ::=
    any character except ', &, or <.
  | xml-escaped
```

If the *xml-name-form* is either `xmlns` or a compound named with the prefix `xmlns`, then technically we have a namespace declaration, rather than an attribute.

20.4.4 QNames and namespaces

The names of elements and attributes are *qualified names* (QNames), which are represented using compound symbols (see Section 10.2 [Namespaces], page 161). The lexical syntax for a QName is either a simple identifier, or a (prefix,local-name) pair:

> *QName* `::=` *xml-local-part*
>   `|` *xml-prefix*:*xml-local-part*
> *xml-local-part* `::=` *identifier*
> *xml-prefix* `::=` *identifier*

An *xml-prefix* is an alias for a namespace-uri, and the mapping between them is defined by a namespace-declaration. You can either use a `define-namespace` form, or you can use a *namespace declaration attribute*:

> *xml-namespace-declaration-attribute* `::=`
>   **xmlns:***xml-prefix*=*xml-attribute-value*
> `|` **xmlns**=*xml-attribute-value*

The former declares *xml-prefix* as a namespace alias for the namespace-uri specified by *xml-attribute-value* (which must be a compile-time constant). The second declares that *xml-attribute-value* is the default namespace for simple (unprefixed) element tags. (A default namespace declaration is ignored for attribute names.)

```
(let ((qn (element-name #<gnu:b xmlns:gnu="http://gnu.org/"/>)))
  (list (symbol-local-name qn)
        (symbol-prefix qn)
        (symbol-namespace-uri qn)))
⇒ ("b" "gnu" "http://gnu.org/")
```

20.4.5 Other XML types

20.4.5.1 Processing instructions

An *xml-PI-constructor* can be used to create an XML *processing instruction*, which can be used to pass instructions or annotations to an XML processor (or tool). (Alternatively, you can use the `processing-instruction` type constructor.)

> *xml-PI-constructor* `::=` **<?***xml-PI-target xml-PI-content***?>**
> *xml-PI-target* `::=` *NCname* (i.e. a simple (non-compound) identifier)
> *xml-PI-content* `::=` any characters, not containing `?>`.

For example, the DocBook XSLT stylesheets can use the `dbhtml` instructions to specify that a specific chapter should be written to a named HTML file:

```
#<chapter><?dbhtml filename="intro.html" ?>
<title>Introduction</title>
...
</chapter>
```

20.4.5.2 XML comments

You can cause XML comments to be emitted in the XML output document. Such comments can be useful for humans reading the XML document, but are usually ignored by programs. (Alternatively, you can use the `comment` type constructor.)

xml-comment-constructor `::=` **<!--***xml-comment-content***-->**
xml-comment-content `::=` any characters, not containing `--`.

20.4.5.3 CDATA sections

A `CDATA` section can be used to avoid excessive use of *xml-entity-ref* such as `&amp;` in element content.

xml-CDATA-constructor `::=` **<![CDATA[***xml-CDATA-content***]]>**
xml-CDATA-content `::=` any characters, not containing `]]>`.

The following are equivalent:

```
#<p>Specal characters <![CDATA[< > & ' "]]> here.</p>
#<p>Specal characters &lt; &gt; &amp; &quot; &apos; here.</p>
```

Kawa remembers that you used a `CDATA` section in the *xml-element-constructor* and will write it out using a `CDATA` constructor.

20.5 Web page scripts

A Kawa *web page script* is a Kawa program that is invoked by a web server because the server received an HTTP request. The result of evaluating the top-level expressions becomes the HTTP response that the servlet sends back to the client, usually a browser.

A web page script may be as simple as:

```
(format "The time is <~s>." (java.util.Date))
```

This returns a response of consisting of a formatted string giving the current time. The string would interpreted as `text/plain` content: The angle brackets are regular characters, and not HTML tag markers.

The script can alternatively evaluate to XML/HTML node values, for example those created by Section 20.4 [XML literals], page 339:

```
#<p>Hello, <b>&(request-remote-host)</b>!</p>
```

In this case the response would be `text/html` or similar content: The angle brackets should be interpreted by the browser as HTML tag markers. The function `request-remote-host` is available (automatically) to web page scripts; it returns the host that made the HTTP request, which is then interpolated into the response.

Following sections will go into more details about how to write web page scripts. You can do so in any supported Kawa language, including Scheme, BRL, KRL, or XQuery.

A web server will use a URL mapping to map a request URL to a specific web page script. This can be done in a number of different ways:

- The easiest to manage is to use Kawa's mechanism for Section 20.6 [Self-configuring page scripts], page 343. Ths is especially easy if you the web server built in to JDK 6, since no configuration files are needed. You can also use a “servlet engine” like Tomcat or Glassfish.
- You can explicitly compile the web page script to a servlet, in the same way Java servlets are compiled. This can then be installed ("deployed") in a servlet-supporting web server, such a Tomcat or Glassfish. See Section 20.7 [Servlets], page 346.
- You can run the servlet as a Section 20.8 [CGI scripts], page 349.

For details on how to extract information from the request see Section 20.9 [HTTP requests], page 350. For details on how the response is created see Section 20.10 [HTTP response], page 354. If the response is HTML or XML, you may want to read Section 20.2 [Creating HTML nodes], page 337, or Section 20.3 [Creating XML nodes], page 338, or Section 20.4 [XML literals], page 339.

Here are some examples, starting with a simple `hello.scm`:

```
(response-content-type 'text/html) ; Optional
(html: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.")))
```

This returns two `<p>` (paragraph) elements: One using `make-element` and one using the `html:p` constructor. Or you may prefer to use Section 20.4 [XML literals], page 339.

The same program using KRL:

```
<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>
```

The `+default+` script in the `doc` directory is useful for reading the Kawa documentation using a browser. The script uses the `jar:` URL scheme to automatically extract and uncompress the pages from `doc/kawa-manual.epub`, which is in EPUB3 format. Read the script for usage instructions.

20.6 Self-configuring web page scripts

Kawa makes it easy to set up a web site without configuration files. Instead, the mapping from request URL to web page script matches the layout of files in the application directory.

Many web servers make it easy to execute a script using a script processor which is selected depending on the extension of the requested URL. That is why you see lots of URLs that end in `.cgi`, `.php`, or `.jsp`. This is bad, because it exposes the server-side implementation to the user: Not only are such URLs ugly, but they make it difficult to change the server without breaking people's bookmarks and search engines. A server will usually provide a mechanism to use prettier URLs, but doing so requires extra effort, so many web-masters don't.

If you want a script to be executed in response to a URL `http://host/app/foo/bar` you give the script the name `app/foo/bar`, in the appropriate server “application” directory (as explained below). You get to pick the name `bar`. Or you can use the name `bar.html`, even though the file named `bar.html` isn’t actually an html file - rather it produces html when evaluated. Or better: just use a name without an extension at all. Kawa figures out what kind of script it is based on the content of the file, rather than the file name. Once Kawa has found a script, it looks at the first line to see if it can recognize the kind (language) of the script. Normally this would be a comment that contains the name of a programming language that Kawa knows about. For example:

```
;; Hello world page script written in -*- scheme -*-
#<p>Hello, <b>&(request-remote-host)</b>!</p>
```

(Using the funny-looking string `-*- scheme -*-` has the bonus is that it recognized by the Emacs text editor.)

A script named `+default+` is run if there isn’t a matching script. For example assume the following is a file named `+default`.

```
;; This is -*- scheme -*-
(make-element 'p "servlet-path: " (request-servlet-path))
```

This becomes the default script for HTTP requests that aren’t handled by a more specific script. The `request-servlet-path` function returns the "servlet path", which is the part of the requested URL that is relative to the current web application. Thus a request for `http://host:port/app/this/is/a/test` will return:

```
servlet-path: /this/is/a/test
```

You can use the feature variable `in-http-server` in a `cond-expand` to test if the code is executing in a web server.

20.6.1 Using the OpenJDK built-in web server

The easiest way to run a Kawa web server is to use the web server built in to JDK 6 or later.

```
kawa --http-auto-handler context-path appdir --http-start port
```

This starts a web server that listens on the given *port*, using the files in directory *appdir* to handle requests that start with the given *context-path*. The *context-path* must start with a `"/"` (one is added if needed), and it is recommended that it also end with a `"/"` (otherwise you might get some surprising behavior).

You can specify multiple `--http-auto-handler` options.

For example use the files in the current directory to handle all requests on the standard port 80 do:

```
kawa --http-auto-handler / . --http-start 80
```

There are some examples in the `testsuite/webtest` directory the Kawa source distribution. You can start the server thus:

```
bin/kawa --http-auto-handler / testsuite/webtest/ --http-start 8888
```

and then for example browse to `http://localhost:8888/adder.scm`.

For lots of information about the HTTP request, browse to `http://localhost:8888/info/`*`anything`*.

20.6.2 Using a servlet container

You can also can use a “servlet container” such as Tomcat or Glassfish with self-configuring script. See Section 20.7 [Servlets], page 346, for information on how to install these servers, and the concept of web applications. Once you have these server installed, you create a web application with the following in the *appdir*/WEB-INF/web.xml configuration file:

```
<web-app>
  <display-name>Kawa auto-servlet</display-name>
  <servlet>
    <servlet-name>KawaPageServlet</servlet-name>
    <servlet-class>gnu.kawa.servlet.KawaPageServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>KawaPageServlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

This creates a web application where all URLs are handled by the `gnu.kawa.servlet.KawaPageServlet` servlet class, which is included in the Kawa jar file. The `KawaPageServlet` class handles the searching and compiling described in this page.

20.6.3 Finding a matching script

When Kawa receives a request for:

```
http://host:port/appname/a/b/anything
```

it will look for a file:

```
appdir/a/b/anything
```

If such a file exists, the script will be executed, as described below. If not, it will look for a file name `+default+` in the same directory. If that desn’t exist either, it will look for `+default+` in the parent directory, then the grand-parent directory, and so on until it gets to the appname web application root directory. So the default script is this: *appdir*/`+default`.

If that doesn’t exist then Kawa returns a 404 "page not found" error.

20.6.4 Determining script language

Once Kawa has found a script file corresponding to a request URL, it needs to determine if this is a data file or a web page script, and in the latter case, what language it is written in.

Kawa recognizes the following "magic strings" in the first line of a script:

`kawa:scheme`
: The Scheme language.

`kawa:xquery`
: The XQuery language.

`kawa:`*`language`*
: Some other language known to Kawa.

Kawa also recognizes Emacs-style "mode specifiers":

`-*- scheme -*-`
: The Scheme language.

`-*- xquery -*-`
: The XQuery language (though Emacs doesn't know about XQuery).

`-*- emacs-lisp -*-`
`-*- elisp -*-`
: The Emacs Lisp extension language.

`-*- common-lisp -*-`
`-*- lisp -*-`
: The Common Lisp language.

Also, it also recognizes comments in the first two columns of the line:

`;;` A Scheme or Lisp comment - assumed to be in the Scheme language.

`(:` Start of an XQuery comment, so assumed to be in the XQuery language.

If Kawa doesn't recognize the language of a script (and it isn't named +default+) then it assumes the file is a data file. It asks the servlet engine to figure out the content type (using the getMimeType method of ServletContext), and just copies the file into the response.

20.6.5 Compilation and caching

Kawa automatically compiles a script into a class. The class is internal to the server, and is not written out to disk. (There is an unsupported option to write the compiled file to a class file, but there is no support to use previously-compiled classes.) The server then creates a module instance to handle the actual request, and runs the body (the `run` method) of the script class. On subsequence requests for the same script, the same class and instance are reused; only the `run` is re-executed.

If the script is changed, then it is re-compiled and a new module instance created. This makes it very easy to develop and modify a script. (Kawa for performance reasons doesn't check more than once a second whether a script has been modified.)

20.7 Installing web page scripts as Servlets

You can compile a Kawa program to a Servlet (`http://en.wikipedia.org/wiki/Java_Servlet`), and run it in a servlet engine (a Servlet-aware web server). One or more servlets are installed together as a web application. This section includes specific information for the Tomcat and Glassfish web servers.

20.7.1 Creating a web application

A *web application* is a group of data, servlets, and configuration files to handle a related set of URLs. The servlet specification (`http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html`) specifies the directory structure of a web application.

Assume the web application is called `myapp`, and lives in a directory with the same name. The application normally handles requests for URLs that start with `http://example.com/myapp`. Most files in the application directory are used to handle requests with corresponding URL. For example, a file `myapp/list/help.html` would be the response to the request `http://example.com/myapp/list/help.html`.

The directory `WEB-INF` is special. It contains configuration files, library code, and other server data.

So to create the `myapp` application, start with:

```
mkdir myapp
cd myapp
mkdir WEB-INF WEB-INF/lib WEB-INF/classes
```

Copy the Kawa jar from the `lib` direcory. (You can also use a "hard" link, but symbolic links may not work, for security systems.)

```
cp kawa-home/kawa-3.1.1.jar WEB-INF/lib/kawa.jar
```

If you build Kawa from source, you must specify the `--with-servlet` [configure options], page 62.

You should also create the file `WEB-INF/web.xml`. For now, this is is just a place-holder:

```
<web-app>
  <display-name>My Application</display-name>
</web-app>
```

20.7.2 Compiling a web page script to a servlet

Assume for simplicity that the source files are in the `WEB-INF/classes` directory, and make that the current directory:

```
cd .../myapp/WEB-INF/classes
```

Depending on the source language, you compile your script sing the `--servlet` switch:

```
kawa --servlet -C hello.scm
```

or:

```
kawa --servlet --krl -C hello.krl
```

or:

```
kawa --servlet --xquery -C hello.xql
```

This lets the web-application find the compiled servlets. Finally, you just need to add the new servlet to the `WEB-INF/web.xml` file:

```
<web-app>
  <display-name>My Application</display-name>

  <servlet>
    <servlet-name>MyHello</servlet-name>
    <servlet-class>hello</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>MyHello</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

The `<servlet>` clause says that the servlet named `MyHello` is implemented by the Java class `hello`. The `<servlet-mapping>` clause says that a request URL `/hello` should be handled by the servlet named `MyHello`. The URL is relative to the application context path, so the actual URL would be `http://example.com/myapp/hello`.

20.7.3 Installing a servlet under Tomcat

Apache's Tomcat (`http://tomcat.apache.org/`) is an open-source implementation of the servlet specifications. After you download it (`http://tomcat.apache.org/download-60.cgi`), uncompress it in some convenient location, which is commonly referred to as `$CATALINA_HOME`.

To install your web application, copy/move its directory to be in the `$CATALINA_HOME/webapps` directory. Thus for the example above you would have a `$CATALINA_HOME/webapps/myapp` directory.

To start or stop Tomcat use the scripts in `$CATALINA_HOME/bin`. For example to start Tomcat on a GNU/Linux system run `$CATALINA_HOME/bin/startup.sh`. This will start a web server that listens on the default port of 8080, so you can browse the above example at `http://localhost:8080/myapp/hello`.

If you're running Fedora GNU/Linux, you can use the `tomcat6` package:

```
# yum install tomcat6
# export CATALINA_HOME=/usr/share/tomcat6
```

You can the manage Tomcat like other system services. You can install webapps under `$CATALINA_HOME/webapps`.

20.7.4 Installing a servlet under Glassfish

Glassfish (`https://glassfish.dev.java.net/`) from Oracle/Sun is a open-source "application server" that implements Java EE 6, including the 3.0 servlet specification. After you download it (`https://glassfish.dev.java.net/downloads/3.0.1-final.html`), uncompress it in some convenient location. This location is called *as-install-parent* in the Quick Start Guide (`http://docs.sun.com/app/docs/doc/820-7689/aboaa?a=view`). The commands you will use is most in *`as-install`*`/bin`, where *as-install* is *`as-install`*`/glassfish`.

To start the server, do:

```
as-install/bin/startserv
```

or under under Windows:

```
as-install\bin\startserv.bat
```

The default post to listen to is `8080`; you can the port (and lots of other properties) using the adminstration console at port `4848`.

A web application does not need to be any particular location, instead you just install it with this command:

```
as-install/bin/adadmin deploy appdir
```

where *appdir* is the application directory - `myapp` in the example. (Use `asadmin.bat` under Windows.)

20.7.5 Servlet-specific script functions

The following functions only work within a servlet container. To use these functions, first do:

```
(require 'servlets)
```

You can conditionalize your code to check at compile-time for servlets, like this:

```
(cond-expand
 (in-servlet
   (require 'servlets)
   (format "[servlet-context: ~s]" (current-servlet-context)))
 (else
   "[Not in a servlet]"))
```

For a run-time check you can test if `(current-servlet)` is non-`#!null`.

`current-servlet` [Procedure]

When called from a Kawa servlet handler, returns the actual `javax.servlet.http.HttpServlet` instance. Returns `#!null` if the current context is not that of `KawaServlet`. (Hence this function also returns `#!null` if you compile a servlet "by hand" rather that using the `--servet` option.)

`current-servlet-context` [Procedure]

Returns the context of the currently executing servlet, as an instance of `javax.servlet.ServletContext`.

`current-servlet-config` [Procedure]

Returns the `ServletConfig` of the currently executing servlet.

`get-request` [Procedure]

Return the current servlet request, as an instance of `javax.servlet.http.HttpServletRequest`.

`get-response` [Procedure]

Return the current servlet response, as an instance of `javax.servlet.http.HttpServletResponse`.

`request-servlet-path` [Procedure]

Get the servlet path of the current request. Similar to `request-script-path`, but not always the same, depending on configuration, and does *not* end with a `"/"`.

`request-path-info` [Procedure]

Get the path info of the current request. Corresponds to the CGI variable `PATH_INFO`.

`servlet-context-realpath` [*path*] [Procedure]

Returns the file path of the current servlet's `"Web application"`.

20.8 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 20.5 [Server-side scripts], page 342, and then use Kawa's supplied CGI-to-servlet bridge.

First, compile your program to one or more class files as explained in Section 20.5 [Server-side scripts], page 342. 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.

20.9 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.

The examples below assume the request `http://example.com:8080/myapp/foo/bar?val1=xyz&val2=abc`, where `myapp` is the application context. We also assume that this is handled by a script `foo/+default+`.

The file `testsuite/webtest/info/+default+` in the Kawa source distribution calls most of these functions. You can try it as described in Section 20.6 [Self-configuring page scripts], page 343.

20.9.1 Request URL components

`request-URI` [Procedure]

Returns the URI of the request, as a value of type `URI`. This excludes the server specification, but includes the query string. (It is the combination of CGI variables `SCRIPT_NAME`, `PATH_INFO`, and `QUERY_STRING`. Using servlets terminology, it is the combination of Context Path, Servlet Path, PathInfo, and Query String.)

```
(request-URI) ⇒ "/myapp/foo/bar?val1=xyz&val2=abc"
```

`request-path` [Procedure]

Returns the URI of the request, as a value of type `URI`. This excludes the server specification and the query string. Equivalent to `(path-file (request-URI))`. (It is the combination of CGI variables `SCRIPT_NAME`, and `PATH_INFO`. Same as the concatenation of `(request-context-path)`, `(request-script-path)`, and `(request-local-path)`. Using servlets terminology, it is the combination of Context Path, Servlet Path, and PathInfo.)

```
(request-path) ⇒ "/myapp/foo/bar"
```

`request-uri` [Procedure]

This function is deprecated, because of possible confusion with `request-URI`. Use `request-path` instead.

`request-url` [Procedure]

Returns the complete URL of the request, except the query string. The result is a `java.lang.StringBuffer`.

```
(request-url) ⇒ "http://example.com:8080/myapp/foo/bar"
```

`request-context-path` [Procedure]

Returns the context path, relative to the server root. This is an initial substring of the `(request-path)`. Similar to the Context Path of a servlet request, except that it ends with a `"/"`.

```
(request-context-path) ⇒ "/myapp/"
```

`request-script-path` [Procedure]

Returns the path of the script, relative to the context. This is either an empty string, or a string that ends with `"/"`, but does not start with one. (The reason for this is to produce URIs that work better with operations like `resolve-uri`.) This is conceptually similar to `request-servlet-path`, though not always the same, and the `"/"` conventions differ.

```
(request-script-path) ⇒ "foo/"
```

`request-local-path` [Procedure]

Returns the remainder of the `request-path`, relative to the `request-script-path`.

```
(request-local-path) ⇒ "bar"
```

`request-query-string` [Procedure]

Returns the query string from an HTTP request. The query string is the part of the request URL after a question mark. Returns false if there was no query string. Corresponds to the CGI variable `QUERY_STRING`.

```
(request-query-string) ⇒ "val1=xyz&val2=abc"
```

20.9.2 Request parameters

Request parameters are used for data returned from forms, and for other uses. They may be encoded in the query string or in the request body.

`request-parameter` *name* [*default*] [Procedure]

If there is a parameter with the given name (a string), return the (first) corresponding value, as a string. Otherwise, return the *default* value, or `#!null` if there is no *default*.

```
(request-parameter "val1") ⇒ "xyz"
(request-parameter "val9" "(missing)") ⇒ "(missing)"
```

`request-parameters` *name* [Procedure]

If there is are one or more parameter with the given name (a string), return them all (as multiple values). Otherwise, return no values (i.e. `(values)`).

```
(request-parameters "val1") ⇒ "xyz"
(request-parameters "val9") ⇒ #!void
```

`request-parameter-map` [Procedure]

Request a map of all the parameters. This is a map from strings to a sequence of strings. (Specifically, a `java.util.Map<String,java.util.List<String>>`.)

20.9.3 Request headers

The request headers are a set of (keyword, string)-pairs transmitted as part of the HTTP request, before the request body.

`request-header` *name* [Procedure]

If there is a header with the given *name* (a string), return the corresponding value string. Otherwise, return `#!null`.

```
(request-header "accept-language") ⇒ "en-us,en;q=0.5"
```

`request-header-map` [Procedure]

Request a map of all the headers. This is a map from strings to a sequence of strings. (Specifically, a `java.util.Map<String,java.util.List<String>>`.)

20.9.4 Request body

`request-input-port` [Procedure]

Return a textual input port for reading the request body, as a sequence of characters.

`request-input-stream` [Procedure]

Return a binary input stream for reading the request body, as a sequence of bytes.

`request-body-string` [Procedure]

Return the entire request body as a string

20.9.5 Request IP addresses and ports

Information about the interface and port on which the request was received.

`request-local-socket-address` [Procedure]

The local address on which the request was received. This is the combination of `(request-local-host)` and `(request-local-port)`, as an instance of `java.net.InetSocketAddress`.

`request-local-host` [Procedure]
Get the IP address of the interface on which request was received, as an `java.net.InetAddress`.

`request-local-IP-address` [Procedure]
Get the IP address of the interface on which request was received, a string in numeric form:

```
(request-local-host) ⇒ "127.0.0.1"
```

`request-local-port` [Procedure]
Get the port this request was received on.

```
(request-local-port) ⇒ 8080
```

Information about the interface and port of the remote client that invoked the request.

`request-remote-socket-address` [Procedure]
The address of the remote client (usually a web browser) which invoked the request. This is the combination of `(request-remove-host)` and `(request-remote-port)`, as an instance of `java.net.InetSocketAddress`.

`request-remote-host` [Procedure]
Get the IP address of the remote client which invoked the request, as an `java.net.InetAddress`.

`request-remote-IP-address` [Procedure]
Get the IP address of the remote client which invoked the request, as a string in numeric form.

```
(request-remote-host) ⇒ "123.45.6.7"
```

`request-remote-port` [Procedure]
The port used by the remote client.

20.9.6 Miscellaneous request properties

`request-path-translated` [Procedure]
Map the request-path to a file name (a string) in the server application directory. Corresponds to the CGI variable `PATH_TRANSLATED`.

`request-method` [Procedure]
Returns the method of the HTTP request, usually `"GET"` or `"POST"`. Corresponds to the CGI variable `REQUEST_METHOD`.

`request-scheme` [Procedure]
Returns the scheme (protocol) of the request. Usually `"http"`, or `"https"`.

20.10 Generating HTTP responses

The result of evaluating the top-level expressions of a web page script 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. Before the main part of the response there 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 a `"Content-Type"` response value is special since it controls the formatting of the following non-response-header values.

`response-header` *key value* [Procedure]

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* [Procedure]

Species the content-type of the result - for example `"text/plain"`. Convenience function for `(response-header "Content-Type"` *`type`*`)`.

`error-response` *code* [*message*] [Procedure]

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*] [Procedure]

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`.)

20.11 Using non-Scheme languages for XML/HTML

20.11.1 XQuery language

Bundled with Kawa is a fairly complete implementation of W3C's new XML Query language (`http://www.w3c.org/XML/Query`). 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.

20.11.2 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.

20.11.3 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`.

20.11.4 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 `&lt;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.

21 Miscellaneous topics

`scheme-implementation-version` [Procedure]

Returns the Kawa version number as a string.

`scheme-window` [*shared*] [Procedure]

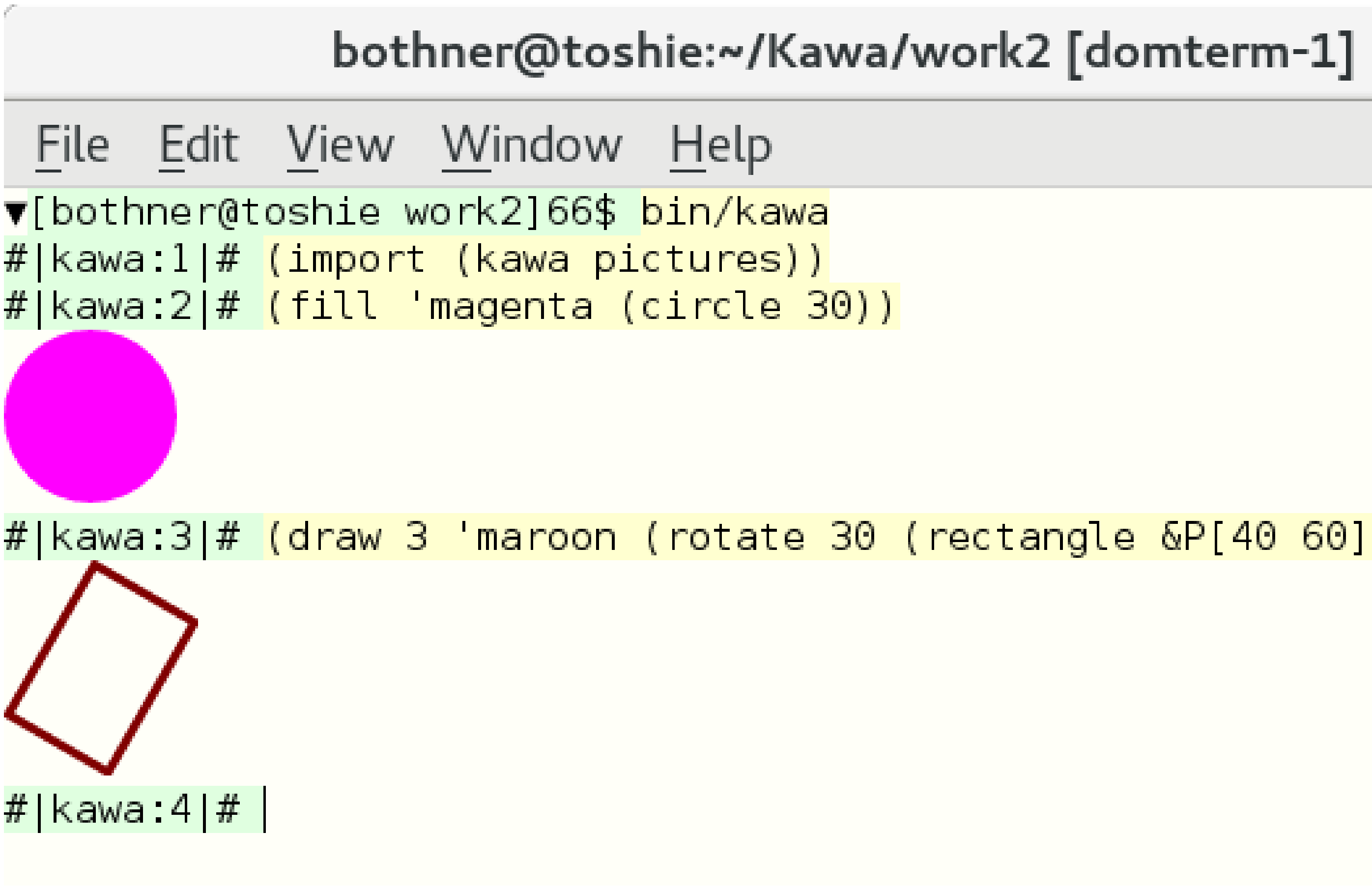
Create a read-eval-print-loop in a new top-level window. If *shared* is true, it uses the same environment as the current `(interaction-environment)`; if not (the default), a new top-level environment is created.

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

21.1 Composable pictures

The `(kawa pictures)` library lets you create geometric shapes and images, and combine them in interesting ways. The Section 5.6 [Tutorial - Pictures], page 75, gives an introduction.

The easiest way to use and learn the `pictures` library is with a suitable REPL. You can use the old Swing-based console or any [Using DomTerm], page 100-based terminal emulator. You can create a suitable window either by starting `kawa` with the `-w` flag, or by running the `kawa` command inside an existing DomTerm-based terminal emulator. The screenshot below is of the latter, using the `qtdomterm` terminal emulator.


bothner@toshie:~/Kawa/work2 [domterm-1]
File Edit View Window Help

```
▼[bothner@toshie work2]66$ bin/kawa
#|kawa:1|# (import (kawa pictures))
#|kawa:2|# (fill 'magenta (circle 30))
#|kawa:3|# (draw 3 'maroon (rotate 30 (rectangle &P[40 60]
#|kawa:4|# |
```

After `(import (kawa swing))` you can use `show-picture` to display a picture in a Swing window.

A *picture* is an object that can be displayed on a screen, web-page, or printed page, and combined with other pictures.

A picture has a method printing itself in a graphical context. It also has various properties.

An important property of a picture is its *bounding box*. This is a rectangle (with edges parallel to the axes) that surrounds the contents of the picture. Usually the entire visible part of the picture is inside the bounding box, but in some cases part of the picture may stick outside the bounding box. For example when a circle is drawn (stroked) with a “pen”, the bounding box is that of the infinitely-thin mathematical circle, so “ink” from the pen that is outside the circle may be outside the bounding box.

A picture has an origin point corresponding to the (0 0) cordinates. The origin is commonly but not always inside the bounding box. Certain operations (for example `hbox`) combine pictures by putting them “next to” each other, where “next to” is defined in terms of the bounding box and origin point.

21.1.1 Coordinates - points and dimensions

The library works with a two-dimensional grid, where each position has an x cordinate and y coordinate. Normally, x values increase as you move right on the screen/page, while y values increase as you move *down*. This convention matches that used by Java 2D, SVG, and many other graphics libraries. However, note that this is different from the traditional mathematical convention of y values increasing as you go up.

By default, one unit is one “pixel”. (More precisely, it is the `px` unit in the CSS specification.)

`Point` [Type]

A point is a pair consisting of an x and a y coordinate.

`&P[` *x y* `]` [Literal]

Construct a point value with the specified *x* and *y* values. Both *x* and *y* are expressions that evaluate to real numbers:

```
&P[(+ old-right 10) baseline]
```

`Dimension` [Type]

A dimension value is a pair of a width and a height. It is used for the size of pictures in the two dimensions.

In a context that expects a point, a dimension is treated as an offset relative to some other point.

`&D[` *`width height`* `]` [Literal]

Construct a dimension value with the specified *width* and *height* values, which are both expressions that evaluate to real numbers.

21.1.2 Shapes

A shape is a collection of lines and curves. Examples include lines, circles, and polygons. A shape can be *stroked*, which you can think of as being drawn by a very fancy calligraphic pen that follows the lines and curves of the shape.

A *closed shape* is a shape that is continuous and ends up where it started. This includes circles and polygons. A closed shape can be filled, which means the entire “interior” is painted with some color or texture.

A shape is represented by the Java `java.awt.Shape` interface. The `picture` library only provides relatively simple shapes, but you can use any methods that create a `java.awt.Shape` object.

Shape is effectively a sub-type of *picture*, though they’re represented using using disjoint classes: If you use a shape where a picture is required, the shape is automatically converted to a picture, as if using the `draw` procedure.

`line` *p1* [*p2* ...] [Procedure]

In the simple case two points are specified, and the result is a line that goes from point *p1* to *p2*. If *n* points are specied, the result is a *polyline*: a path consisting of *n-1* line segments, connecting adjacent arguments. (If only a single point is specified, the result is degenerate single-point shape.)

All of the points except the first can optionally be specified using a dimension, which is treated an an offset from the previous point:

```
(line &P[30 40] &D[10 5] &D[10 -10])
```

is the same as:

```
(line &P[30 40] &P[40 45] &P[50 35])
```

`polygon` *p1* [*p2* ...] [Procedure]

Constructs a closed shape from line segments. This is the same as calling `line` with the same arguments, with the addition of a final line segment from the last point back to the first point.

`rectangle` *p1* [*p2*] [Procedure]

A rectangle is closed polygon of 4 line segments that are alternatively parallel to the x-axis and the y-axis. I.e. if you rotate a rectangle it is no longer a rectangle by this definition, though of course it still has a rectangular shape. If *p2* is not specified, constructs a rectangle whose upper-left corner is `&P[0 0]` and whose lower-right corner is *p1*. If *p2* is specified, constructs a rectangle whose upper-left corner is *p1* and whose lower-right corner is *p2*. If *p2* is a dimension it is considered a relative offset from *p1*, just like for `polygon`.

`circle` *radius* [*center*] [Procedure]

Creates a circle with the specified *radius*. If the *center* is not specified, it defaults to `&P[0 0]`.

21.1.3 Colors and paints

A *paint* is a color pattern used to fill part of the canvas. A paint can be a color, a texture (a replicated pattern), or a gradient (a color that gradually fades to some other color).

A *color* is defined by red, green, and blue values. It may also have an alpha component, which specifies transparency.

`->paint` *value* [Procedure]

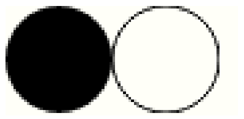
Converts *value* to a color - or more general a paint. Specificlly, the return type is `java.awt.Paint`. The *value* can be any one of:

- A `java.awt.Paint`, commonly a `java.awt.Color`.
- A 24-bit integer value is converted to a color. Assume the integer is equal to a hexadecimal literal of the form `#xRRGGBB`. Then `RR` (bits 16-23) is the intensity of the red component; `GG` (bits 8-15) is the intensity of the green component; and `RR` (bits 0-7) is the intensity of the red component.
- One of the standard HTML/CSS/SVG color names, as a string or symbol. See the table in `gnu/kawa/models/StandardColor.java` source file. Case is ignored, and you can optionally use hyphens to separate words. For example `'hot-pink`, `'hotpink`, and `'hotPink` are all the same sRGB color `#xFF69B4`.

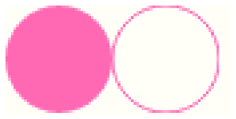
`with-paint` *paint picture* [Procedure]

Create a new picture that is the "same" as *picture* but use *paint* as the default paint. For *paint* you can use any valid argument to `->paint`. The default paint (which is the color black if none is specified) is used for both `fill` (paint interior) and `draw` (stroke outline).

```
#|kawa:1|# (! circ1 (circle 20 &P[20 20]))
#|kawa:2|# (hbox (fill circ1) (draw circ1))
```

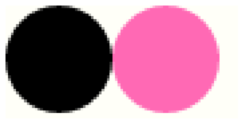


```
#|kawa:3|# (with-paint 'hot-pink (hbox (fill circ1) (draw circ1)))
```

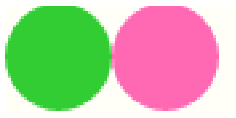


Above we use `with-paint` to create a cloned picture, which is the same as the original `hbox`, except for the default paint, in this case the color `hot-pink`.

```
#|kawa:4|# (! circ2 (hbox (fill circ1) (with-paint 'hot-pink (fill circ1))))
#|kawa:5|# circ2
```



```
#|kawa:6|# (with-paint 'lime-green circ2)
```



Here `circ2` is an `hbox` of two filled circles, one that has unspecified paint, and one that is `hot-pink`. Printing `circ2` directly uses black for the circle with unspecified color, but if we wrap `circ2` in another `with-paint` that provides a default that is used for the first circle.

21.1.4 Filling a shape with a color

`fill` *shape* [Procedure]
`fill` *paint shape* [Procedure]

Fill the "inside" of *shape*. If no *paint* is specified, uses the current default paint. Otherwise, `(fill` *`paint shape`*`)` is the same `(with-paint` *`paint`* `(fill` *`shape`*`))`.

21.1.5 Stroking (outlining) a shape

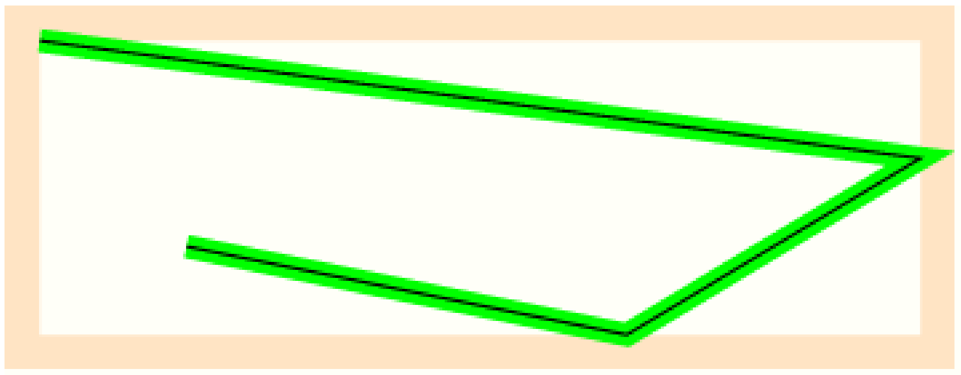
`draw` *option** *shape*+ [Procedure]

Returns a picture that draws the outline of the *shape*. This is called *stroking*. An *option* may be one of:

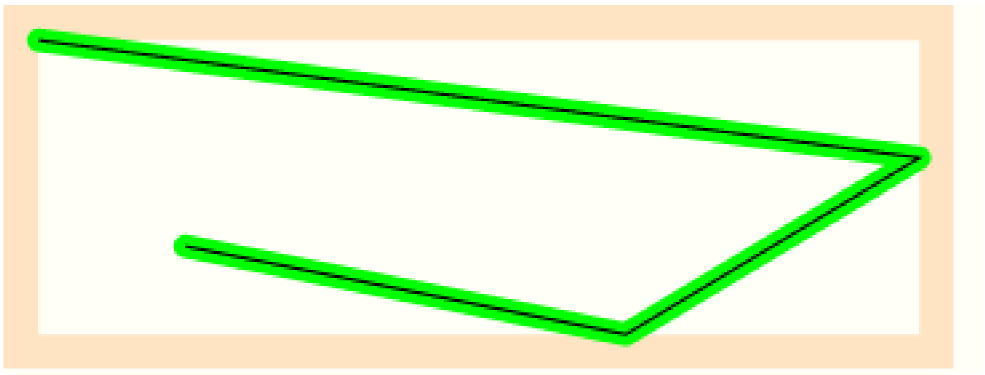
- A `Paint` or `Color` object, which is used to draw the shape.
- A standard color name, such as `'red` or `'medium-slate-blue`. This is mapped to a `Color`.
- A join-specifier, which is a symbol specifying how each curve of the shape is connected to the next one. The options are `'miter-join`, `'round-join`, and `'bevel-join`. The default if none is specified is `'miter-join`.
- A end-cap-specifier, which is a symbol specifying how each end of the shape is terminated. The options are `'square-cap`, `'round-cap`, or `'butt-cap`. The default is `'butt-cap`. (This follows SVG and HTML Canvas. The default in plain Java AWT is a square cap.)
- A real number specifies the thickness of the stroke.
- A `java.awt.Stroke` object. This combines join-specifier, end-cap-specifier, thickness, and more in a single object. The `BasicStroke` class can specify dashes, though that is not yet supported for SVG output; only AWT or image output.

Let us illustrate with a sample line `lin` and a helper macro `show-draw`, which adds a border around a shape, then draws the given shape with some options, and finally re-draws the shape in plain form.

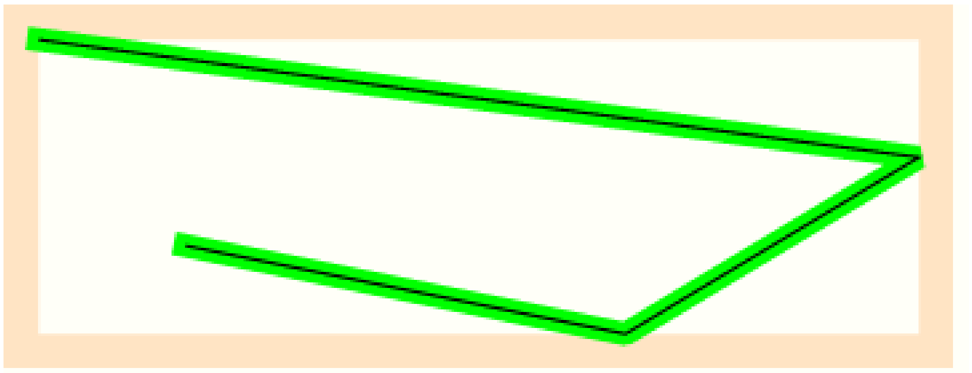
```
#|kawa:10|# (define lin (line &P[0 0] &P[300 40] &P[200 100] &P[50 70]))
#|kawa:11|# (define-syntax show-draw
#|....:12|#   (syntax-rules ()
#|....:13|#     ((_ options ... shape)
#|....:14|#      (border 12 'bisque (zbox (draw options ... shape) shape)))))
#|....:15|# (show-draw 8 'lime lin)
```



```
#|....:16|# (show-draw 8 'lime 'round-cap 'round-join lin)
```



```
#|....:17|# (show-draw 8 'lime 'square-cap 'bevel-join lin)
```



Notice how the different cap and join styles change the drawing. Also note how the stroke (color lime) extends beyond its bounding box, into the surrounding border (color bisque).

21.1.6 Affine transforms

A 2D affine transform is a linear mapping from coordinates to coordinates. It generalizes translation, scaling, flipping, shearing, and their composition. An affine transform maps straight parallel lines into other straight parallel lines, so it is only a subset of possible mappings - but a very useful subset.

affine-transform *xx xy yx yy x0 y0* [Procedure]

affine-transform *px py p0* [Procedure]

Creates a new affine transform. The result of applying `(affine-transform` x_x x_y y_x y_y x_0 y_0`)` to the point `&P[x y]` is the transformed point

```
&P[(+ (* x xx) (* y yx) x0)
   (+ (* x xy) (* y yy) y0)]
```

If using point arguments, `(affine-transform &P[`x_x x_y`] &P[`y_x y_y`] &P[`x_0 y_0`])` is equivalent to: `(affine-transform` x_x x_y y_x y_y x_0 y_0`)`.

`with-transform` *transform picture* [Procedure]
`with-transform` *transform shape* [Procedure]

Creates a transformed picture.

If the argument is a *shape*, then the result is also a shape.

`with-transform` *transform point* [Procedure]

Apply a transform to a single point, yielding a new point.

`with-transform` *transform1 transform2* [Procedure]

Combine two transforms, yielding the composed transform.

`rotate` *angle* [Procedure]
`rotate` *angle picture* [Procedure]

The one-argument variant creates a new affine transform that rotates a picture about the origin by the specified *angle*. A positive *angle* yields a clockwise rotation. The *angle* can be either a quantity (with a unit of either `rad` radians, `deg` (degrees), or `grad` (gradians)), or it can be a unit-less real number (which is treated as degrees).

The two-argument variant applies the resulting transform to the specified picture. It is equivalent to:

```
(with-transform (rotate angle) picture)
```

`scale` *factor* [Procedure]
`scale` *factor picture* [Procedure]

Scales the *picture* by the given *factor*. The *factor* can be a real number. The *factor* can also be a point or a dimension, in which case the two cordinates are scaled by a different amount.

The two-argument variant applies the resulting transform to the specified picture. It is equivalent to:

```
(with-transform (scale factor) picture)
```

`translate` *offset* [Procedure]
`translate` *offset picture* [Procedure]

The single-argument variant creates a transform that adds the *offset* to each point. The *offset* can be either a point or a dimension (which are treated quivalently).

The two-argument variant applies the resulting transform to the specified picture. It is equivalent to:

```
(with-transform (translate offset) picture)
```

21.1.7 Combining pictures

`hbox` [*spacing*] *picture ...* [Procedure]
`vbox` [*spacing*] *picture ...* [Procedure]
`zbox` *picture ...* [Procedure]

Make a combined picture from multiple sub-pictures drawn either "next to" or "on top of" each other.

The case of `zbox` is simplest: The sub-pictures are drawn in argument order at the same position (origin). The "`z`" refers to the idea that the pictures are stacked on top of each other along the "Z-axis" (the one perpendicular to the screen).

The `hbox` and `vbox` instead place the sub-pictures next to each other, in a row or column. If *spacing* is specified, if must be a real number. That much extra spacing is added between each sub-picture.

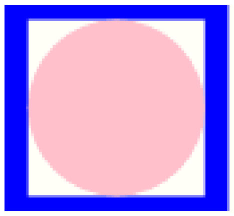
More precisely: `hbox` shifts each sub-picture except the first so its left-origin control-point (see discussion at `re-center`) has the same position as the right-origin control point of the previous picture *plus* the amount of *spacing*. Similarly, `vbox` shifts each sub-picture except the first so its top-origin control point has the same position as the bottom-origin point of the previous picture, plus *spacing*.

The bounding box of the result is the smallest rectangle that includes the bounding boxes of the (shifted) sub-pictures. The origin of the result is that of the first picture.

`border` [*size* [*paint*]] *picture* [Procedure]

Return a picture that combines *picture* with a rectangular border (frame) around *picture*'s bounding box. The *size* specifies the thickness of the border: it can be real number, in which it is the thickness on all four sides; it can be a Dimension, in which case the width is the left and right thickness, while the height is the top and bottom thickness; or it can be a Rectangle, in which case it is the new bounding box. If *paint* is specified it is used for the border; otherwise the default paint is used. The border is painted before (below) the *picture* painted. The bounding box of the result is that of the border, while the origin point is that of the original *picture*.

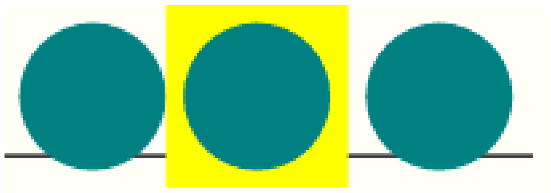
```
#|kawa:2|# (with-paint 'blue (border &D[8 5] (fill 'pink (circle 30))))
```



`padding` *width* [*background*] *picture* [Procedure]

This is similar to `border`, but it just adds extra space around *picture*, without painting it. The *size* is specified the same way. If *background* is specified, it becomes the background paint for the entire padded rectangle (both *picture* and the extra padding).

```
#|kawa:3|# (define circ1 (fill 'teal (circle 25)))
#|kawa:4|# (zbox (line &P[-30 20] &P[150 20])
#|kawa:5|#   (hbox circ1 (padding 6 'yellow circ1) (padding 6 circ1)))
```



This shows a circle drawn three ways: as-is; with padding and a background color; with padding and a transparent background. A line is drawn before (below) the circles to contrast the yellow vs transparent backgrounds.

`re-center` [*xpos*] [*ypos*] *picture* [Procedure]

Translate the *picture* such that the point specified by *xpos* and *ypos* is the new origin point, adjusting the bounding box to match. If the *picture* is a shape, so is the result.

The *xpos* can have four possible values, all of which are symbols: `'left` (move the origin to the left edge of the bounding box); `'right` (move the origin to the right edge of the bounding box); `'center` (or `'centre`) (move the origin to halfway between the left and right edges); or `'origin` (don't change the location along the x-axis). The *ypos* can likewise have four possible values: `'top` (move the origin to the top edge of the bounding box); `'bottom` (move the origin to the bottom edge of the bounding box); `'center` (or `'centre`) (move the origin to halfway between the top and bottom edges); or `'origin` (don't change the location along the y-axis).

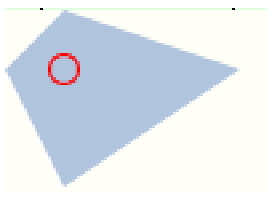
A single `'center` argument is the same as specifying `'center` for both axis; this is the default. A single `'origin` argument is the same as specifying `'origin` for both axis; this is the same as just *picture*.

The 16 control points are shown below, relative to a picture's bounding box and the X- and Y-axes. The abbreviations have the obvious values, for example `LC` means `'left 'center`.

```
LT     OT   CT       RT



LC    OC   C        RC
LOOCORO



LB     OB   CB        RB
```

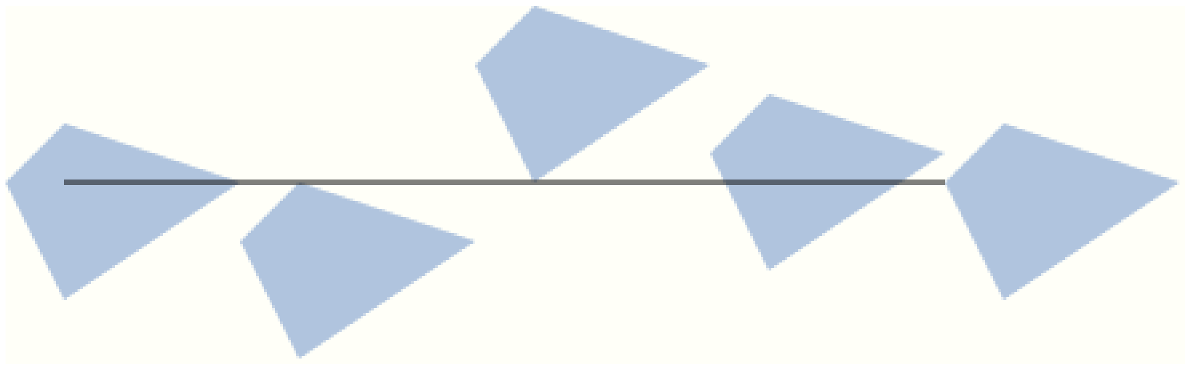
The result of (for example) `(re-center 'left 'center P)` is *P* translated so the origin is at control point `LC`.

```
#|kawa:1|# (define D (fill 'light-steel-blue (polygon &P[-20 0] &P[0 -
20] &P[60 0] &P[0 40])))
#|kawa:2|# (zbox D (draw 'red (circle 5)))
```



Above we defined `D` as a vaguely diamond-shaped quadrilateral. A small red circle is added to show the origin point. Below we display 5 versions of `D` in a line (an `hbox`), starting with the original `D` and 4 calls to `re-center`.

```
#|kawa:3|# (zbox (hbox D (re-center 'top D) (re-center 'bottom D)
#|....:4|#              (re-center 'center D) (re-center 'origin D))
#|....:5|#   (line &P[0 0] &P[300 0]))
```



The line at *y=0* shows the effects of `re-center`.

21.1.8 Images

An image is a picture represented as a rectangular grid of color values. An image file is some encoding (usually compressed) of an image, and mostly commonly has the extensions `png`, `gif`, or `jpg`/`jpeg`.

A "native image" is an instance of `java.awt.image.BufferedImage`, while a "picture image" is an instance of `gnu.kawa.models.DrawImage`. (Both classes implement the `java.awt.image.RenderedImage` interface.) A `BufferedImage` is automatically converted to a `DrawImage` when needed.

`image` *bimage* [Procedure]
`image` *picture* [Procedure]
`image` *src: path* [Procedure]

Creates a picture image, using either an existing native image *bimage*, or an image file specified by *path*.

Writing `(image src: `*`path`*`)` is roughly the same as `(image (read-image `*`path`*`))` except that the former has the *path* associated with the resulting picture image. This can make a difference when the image is used or displayed.

If the argument is a *picture*, it is converted to an image as if by `->image`.

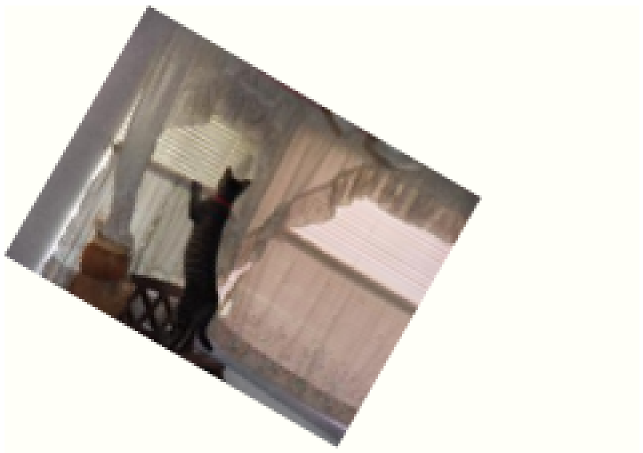
`image-read` *path* [Procedure]

Read an image file from the specified *path*, and returns a native image object (a `BufferedImage`).

```
#|kawa:10|# (define img1 (image-read "http://pics.bothner.com/2013/Cats/06t.jpg"))
```

```
#|kawa:11|# img1
```

```
#|kawa:12|# (scale 0.6 (rotate 30 img1))
```



Note that while `img1` above is a (native) image, the scaled rotated image is not an image object. It is a picture - a more complex value that *contains* an image.

`image-write` *picture path* [Procedure]

The *picture* is converted to an image (as if by using `->image`) and then it is written to the specified *path*. The format used depends on (the lower-cased string value of) the path: A JPG file if the name ends with `".jpg"` or `".jpeg"`; a GIF file if the name ends with `".gif"`; a PNG file if the name ends with `".png"`. (Otherwise, the defalt is PNG, but that might change.)

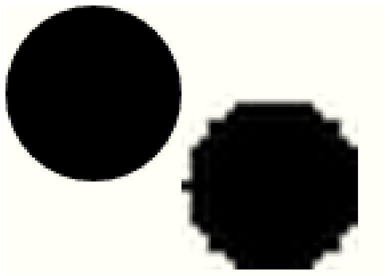
`image-width` *image* [Procedure]
`image-height` *image* [Procedure]

Return the width or height of the given *image*, in pixels.

`->image` *picture* [Procedure]

Convert *picture* to an image (a `RenderedImage`). If the *picture* is an image, return as-is. Otherwise, create an empty image (a `BufferedImage` whose size is the *picture*'s bounding box), and "paint" the *picture* into it.

```
#|kawa:1|# (define c (fill (circle 10)))
#|kawa:2|# (scale 3 (hbox c (->image c)))
```



Here we take a circle `c`, and convert it to an image. Note how when the image is scaled, the pixel artifacts are very noticable. Also note how the origin of the image is the top-level corner, while the origin of the original circle is its center.

21.1.9 Compositing - Controlling how pictures are combined

`with-composite` [[*compose-op*] *picture*] ... [Procedure]

Limited support - SVG and DomTerm output has not been implemented.

21.1.10 Displaying and exporting pictures

21.1.10.1 Export to SVG

`picture-write-svg` *picture path* [*headers*] [Procedure]

Writes the *picture* to the file specified by *path*, in SVG (Structered Vector Graphics) format. If *headers* is true (which is the default) first write out the XML and DOCTYPE declarations that should be in a well-formed standaline SVG file. Otherwise, just write of the `<svg>` element. (Modern browers should be able to display a file consisting of just the `<svg>` element, as long as it has extension `.svg`, `.xml`, or `.html`; the latter may add some extra padding.)

`picture->svg-node` *picture* [Procedure]

Returns a SVG representation of `picture`, in the form of an `<svg>` element, similar to those discussed in Section 20.3 [Creating XML nodes], page 338. If you convert the `<svg>` element to a string, you get formatted XML; if you `write` the `<svg>` element you get an Section 20.4 [XML literals], page 339, of form `"#<svg>...</svg>"`. If you `display` the `<svg>` element in a DomTerm terminal you get the picture (as a picture). This works because when you display an element in DomTerm it gets inserted into the display.

21.1.10.2 Display in Swing

These procedures require `(import (kawa swing))` in addition to `(import (kawa pictures))`.

The convenience function `show-picture` is useful for displaying a picture in a new (Swing) window.

`show-picture` *picture* [Procedure]

If this is the first call to `show-pictures`, displays *picture* in a new top-level window (using the Swing toolkit). Sequenent calls to `show-picture` will reuse the window.

```
#|kawa:1|# (import (kawa swing) (kawa pictures))
#|kawa:2|# (show-picture some-picture)
#|kawa:3|# (set-frame-size! &D[200 200]) ; Adjust window size
#|kawa:4|# (show-picture some-other-picture)
```

`picture->jpanel` *picture* [Procedure]

Return a `JPanel` that displays *picture*. You can change the displayed picture by:

```
(set! panel:picture some-other-picture)
```

`set-frame-size!` *size* [*frame*] [Procedure]

If *frame* is specified, set its size. Otherwise, remember the size for future `show-picture` calls; if there is already a `show-picture` window, adjust its size.

21.1.10.3 Convert to image

You can convert a picture to an image using the `->image` procedure, or write it to a file using the `image-write` procedure.

21.2 Building JavaFX applications

Kawa makes it easy to build "rich client" (i.e. GUI) applications using JavaFX (`http://www.oracle.com/technetwork/java/javafx/overview/index.html`). For example the following program will print up a window with a button; clicking on the button will print a message on the console output about the event.

```
(require 'javafx-defs)
(javafx-application)

(javafx-scene
 title: "Hello Button"
 width: 600 height: 450
 (Button
  text: "Click Me"
  layout-x: 25
  layout-y: 40
  on-action: (lambda (e) (format #t "Event: ~s~%~!" e))))
```

JavaFX support is builtin to the pre-built `kawa-3.1.1.jar`. It is easiest to use JDK 8; see below if you're using JDK 7. If you build Kawa from source, specify `--with-javafx` on the `configure` command line (assuming you're using JDK 8).

Assume the above file is `HelloButton1.scm`, you can run it like this:

```
$ kawa HelloButton1.scm
```

For more information and examples read this (slightly older) introduction (`http://per.bothner.com/blog/2011/JavaFX-using-Kawa-intro/`), and this on animation (`http://localhost/per/blog/2011/JavaFX-using-Kawa-animation/`).

The `browse-kawa-manual` script in the `doc` directory (source only) uses JavaFX WebView to create a window for browsing the Kawa documentation.

21.2.1 Using JavaFX with JDK 11+

Starting with JDK 11, JavaFX has been moved to a separate project and is no longer included in the JDK. The separate project OpenJFX provides an SDK that includes modular jar files which can be added to the `CLASSPATH` or via the `--module-path` parameter to `javac` and `java`.

To run the previous `HelloButton1.scm` you can do:

```
$ java -cp $KAVA_HOME/kawa.jar --module-path $JAVAFX_HOME/lib \
  --add-modules javafx.web HelloButton1.scm
```

If you build Kawa from source you must use an appropriate JDK version and enable the modular OpenJFX SDK:

```
$ ./configure --with-javafx=path-to-sdk ...other-args...
```

The resulting Kawa binary sets up the module path and the boostrap module so you just need to do:

```
$ kawa HelloButton1.scm
```

21.3 Building for Android

Google's phone/tablet operating system Android (`https://developers.google.com/android/`) is based on a custom virtual machine on top of a Linux kernel. Even though Android isn't strictly (or legally) speaking Java, you can build Android applications using Kawa.

Below is "Hello world" written in Kawa Scheme. A slightly more interesting example is in Section 21.4 [Android view construction], page 372.

```
(require 'android-defs)
(activity hello
  (on-create-view
   (android.widget.TextView (this)
    text: "Hello, Android from Kawa Scheme!")))
```

The following instructions have been tested on GNU/Linux, specifically Fedora 17. This link (`http://asieno.com/blog/index.php/post/2012/08/16/Setting-up-the-environment-Android-Kawa`) may be helpful if you're building on Windows.

21.3.1 Downloading and setting up the Android SDK

First download the Android SDK (`http://code.google.com/android/download.html`). Unzip in a suitable location, which we'll refer to as `ANDROID_HOME`.

```
export ANDROID_HOME=/path/to/android-sdk-linux
```

```
PATH=$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools:$PATH
```

Next you have to get the appropriate platform SDK:

```
$ android update sdk
```

You need to select an Android "platform". Platform (API) 16 corresponds to Android 4.1.2 (Jelly Bean). Select that or whatever you prefer, and click `Install`. (You can install multiple platforms, but each project is built for a specific platform.)

```
ANDROID_PLATFORM=android-16
```

21.3.2 Building Kawa for Android

Set `JAVA_HOME` to where your JDK tree is. You should use JDK 6; JDK 7 does not work at time of writing.

```
$ export JAVA_HOME=/opt/jdk1.6
```

First Section 4.1 [Getting Kawa], page 59.

If using Ant (as is recommended on Windows):

```
$ ant -Denable-android=true
```

Alternatively, you can use `configure` and `make`:

```
$ KAWA_DIR=path_to_Kawa_sources
$ cd $KAWA_DIR
$ ./configure --with-android=$ANDROID_HOME/platforms/$ANDROID_PLATFORM/android.jar --d
$ make
```

21.3.3 Creating the application

Next, we need to create a project or "activity". This tutorial assumes you want to create the project in the target directory `KawaHello`, with the main activity being a class named `hello` in a package `kawa.android`:

```
PROJECT_DIR=KawaHello
PROJECT_CLASS=hello
PROJECT_PACKAGE=kawa.android
PROJECT_PACKAGE_PATH=kawa/android
```

To create the project use the following command:

```
$ android create project --target $ANDROID_PLATFORM --name $PROJECT_DIR --activity $PR
```

Replace the skeleton `hello.java` by the Scheme code at the top of this note, placing in a file named `hello.scm`:

```
$ cd $PROJECT_DIR
$ HELLO_APP_DIR=`pwd`
$ cd $HELLO_APP_DIR/src/$PROJECT_PACKAGE_PATH
$ rm $PROJECT_CLASS.java
$ create $PROJECT_CLASS.scm
```

We need to copy/link the Kawa jar file so the Android SDK can find it:

```
$ cd $HELLO_APP_DIR
$ ln -s $KAWA_DIR/kawa-3.1.1.jar libs/kawa.jar
```

Optionally, you can use kawart-3.1.1.jar, which is slightly smaller, but does not support eval, and does not get built by the Ant build:

```
$ ln -s $KAWA_DIR/kawart-3.1.1.jar libs/kawa.jar
```

Copy or link `custom_rules.xml` from the Kawa sources:

```
ln -s $KAWA_DIR/gnu/kawa/android/custom_rules.xml .
```

Finally to build the application just do:

```
$ ant debug
```

21.3.4 Running the application on the Android emulator

First you need to create an Android Virtual Device (avd) (`http://developer.android.com/tools/devices`). Start:

```
android
```

Then from menu `Tools` select `Manage AVDs...`. In the new window click `New....` Pick a `Name` (we use `avd16` in the following), a `Target` (to match `$ANDROID_PLATFORM`), and optionally change the other properties, before clicking `Create AVD`.

Now you can start up the Android emulator:

```
$ emulator -avd avd16 &
```

Wait until Android has finished booting (you will see the Android home screen), click the menu and home buttons. Now install our new application:

```
adb install bin/KawaHello-debug.apk
```

21.3.5 Running the application on your device

If the emulator is running, kill it:

```
$ kill %emulator
```

On your phone or other Android devude, enable USB debugging. (This is settable from the `Settings` application, under `Applications / Development`.)

Connect the phone to your computer with the USB cable. Verify that the phone is accessible to `adb`:

```
$ adb devices
List of devices attached
0A3A560F0C015024 device
```

If you don't see a device listed, it may be permission problem. You can figure out which device corresponds to the phone by doing:

```
$ ls -l /dev/bus/usb/*
/dev/bus/usb/001:
total 0
...
crw-rw-rw- 1 root wheel 189, 5 2010-10-18 16:52 006
...
```

The timestamp corresponds to when you connected the phone. Make the USB connection readable:

```
$ sudo chmod a+w /dev/bus/usb/001/006
```

Obviously if you spend time developing for an Androd phone you'll want to automate this process; this link (`https://sites.google.com/site/siteofhx/Home/android/drivers/udc`) or this link (`https://groups.google.com/forum/?fromgroups=#!topic/android-developers/nTfhhPktGfM`) may be helpful.

Anyway, once `adb` can talk to the phone, you install in the same way as before:

```
adb install bin/KawaHello-debug.apk
```

21.3.6 Some debugging notes

You will find a copy of the SDK documentation in `$ANDROID_HOME/docs/index.html`.

If the emulator complains that your application has stopped unexpectedly, do:

```
$ adb logcat
```

This shows log messages, stack traces, output from the `Log.i` logging method, and other useful information. (You can alternatively start `ddms` (Dalvik Debug Monitor Service), click on the `kawa.android line` in the top-left sub-window to select it, then from the `Device` menu select `Run logcat....`).

To uninstall your application, do:

```
$ adb uninstall kawa.android
```

21.3.7 Other resources

(A more interesting text-to-speech (`http://androidscheme.blogspot.com/2010/10/text-to-speech-app.html`) example app is on Santosh Rajan's Android-Scheme blog (`http://androidscheme.blogspot.com/`).)

`https://github.com/ecraven/SchemeAndroidOGL`

21.4 Android view construction

An Android user interface is constructed from `View` objects. The following is an example that illustrates some features of Kawa to help write views hierarchies, The example is self-contained, and can be built and run as described in Section 21.3 [Building for Android], page 369.

```
(require 'android-defs)
(activity hello
  (on-create-view
   (define counter ::integer 0)
   (define counter-view
     (TextView text: "Not clicked yet."))
   (LinearLayout orientation: LinearLayout:VERTICAL
    (TextView text: "Hello, Android from Kawa Scheme!")
    (Button
     text: "Click here!"
     on-click-listener: (lambda (e)
                          (set! counter (+ counter 1))
                          (counter-view:setText
                           (format "Clicked ~d times." counter))))
    counter-view)))
```

The first `import` form imports various useful definitions from the Kawa Android library. Using these is not required for writing a Kawa application, but makes it more convenient.

The names `LinearLayout`, `TextView`, and `Button` are just aliases for standard Android `View` sub-classes. A few are prefined by `(require 'android-defs)`, or you can define them yourself using `define-alias`.

An Android application consists of one or more *activities*, each of which is an instance of the `android.app.Activity` class. You can use the `activity` macro to define your `Activity` class. The first macro argument (in this case `hello`) is the class name, and the others are members of the class, in the syntax of a *field-or-method-decl*. The sub-form `on-create-view` is an abbreviation for declaring an `onCreate` method (which is called when the `Activity` starts up followed by a `setContentView`: The body of the `on-create-view` is evaluated. The result should be a `View` expression, which is passed to `setContentView`.

`current-activity` [*new-value*] [Procedure]

With no arguments, returns the current `Activity`. If a *new-value* argument is given, sets the current activity. It is set automatically by the `on-create` and `on-create-view` methods of the `activity` macro.

Since `current-activity` is a Section 15.2 [Parameter objects], page 268, you can locally change the value using [parameterize-syntax], page 269.

21.4.1 View object allocation

To create an instance of a `View` class you "call" the class as if it were a function, as described in Section 19.10 [Allocating objects], page 324. For example:

```
(TextView (this) text: "Hello, Android from Kawa Scheme!")
```

If you `(require 'android-defs)` that defines some special handling for `View` classes. You can leave out the `(this)` argument, which refers to the enclosing `Activity`:

```
(TextView text: "Hello, Android from Kawa Scheme!")
```

21.4.2 Event handlers

You can register event listeners on Android `View` objects using methods typically named `setOn`*`EVENT`*`Listener`. For example `setOnClickListener`. When allocating an object you can leave out the `set`, and you can optionally use Scheme-style names: `on-click-listener`. The argument is an object of a special nested listener class, for example `View$OnClickListener`. These are single-method classes, so you can use a lambda expression and [SAM-conversion], page 303, will automatically create the needed listener class.

21.5 System inquiry

`home-directory` [Variable]

A string containing the home directory of the user.

`command-line` [Procedure]

Returns a nonempty list of immutable strings. The first element is an implementation-specific name for the running top-level program. The remaining elements are the command-line arguments, as passed to the `main` method (except for those flags processed by Kawa itself).

The first element will depend on how the Kawa module was invoked. Kawa uses the following rules to determine the command name:

1. If the property `kawa.command.name` is set, that is used. This variable can be set on the `kawa` command line, for example from a script:

   ```
   kawa -Dkawa.command.name="$0" foo "$@"
   ```

   This variable is also set implicitly by the meta-arg option. FIXME.
2. If we're reading a source file that starts with the Unix command-file prefix '`#!/`' then we use the name of the source file. The assumption is that such a file is an executable script.
3. If the Java property `kawa.command.line` is set, then we use that (after stripping off text that duplicates the remaining arguments). The `kawa` program sets this property to the command line used to invoke it (specifically the contents of the entire `argv` array), before invoking the `java` program.
4. If the Java property `sun.java.command` is set, then we use that (after stripping off text that duplicates the remaining arguments), and then prepending the string `"java "`. The OpenJDK `java` program sets this property.
5. If all else fails, the command name is `"kawa"`.

`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.

`process-command-line-assignments` [Procedure]

Process any initial command-line options that set variables. These have the form `name=value`. Any such command-line options (at the start of the command-line) are processed and removed from the command-line.

```
$ java kawa.repl -- abc=123 def
#|kawa:1|# (write (command-line))
("java kawa.repl --" "abc=123" "def")
#|kawa:2|# (process-command-line-assignments)
#|kawa:3|# (write (command-line))
("java kawa.repl -- abc=123" "def")
#|kawa:4|# abc
123
```

This function is mostly useful for Kawa applications compiled with the `--main` option. (It is used to set XQuery `external` variables.)

`get-environment-variable` *name* [Procedure]

Many operating systems provide each running process with an environment conisting of environment variables. (This environment is not to be confused with the Scheme environments that can be passed to `eval`.) Both the name and value of an environment variable are strings. The procedure `get-environment-variable` returns the value of the environment variable *name*, or `#f` if the environment variable is not found. (This uses the `java.lang.System:getenv` method.) It is an error to mutate the resulting string.

```
(get-environment-variable "PATH")
    ⇒ "/usr/local/bin:/usr/bin:/bin"
```

`get-environment-variables` [Procedure]

Returns the names and values of all the environment variables as an alist, where the car of each entry is the name of an environment variable, and the cdr is its value, both as strings. It is an error to mutate any of the strings or the alist itself.

```
(get-environment-variables)
  ⇒ (("USER" . "root") ("HOME" . "/"))
```

21.6 Processes

A *process* is a native (operating-system-level) application or program that runs separately from the current virtual machine.

Many programming languages have facilities to allow access to system processes (commands). (For example Java has `java.lang.Process` and `java.lang.ProcessBuilder`.) These facilities let you send data to the standard input, extract the resulting output, look at the return code, and sometimes even pipe commands together. However, this is rarely as easy as it is using the old Bourne shell; for example command substitution is awkward. Kawa's solution is based on these two ideas:

- A "process expression" (typically a function call) evaluates to a `LProcess` value, which provides access to a Unix-style (or Windows) process.
- In a context requiring a string (or a bytevector), an `LProcess` is automatically converted to a string (or bytevector) comprising the standard output from the process.

21.6.1 Creating a process

The most flexible way to start a process is with either the `run-process` procedure or the ``&`{command}`` syntax for [process literals], page 376.

`run-process` *process-keyword-argument** *command* [Procedure]

Creates a process object, specifically a `gnu.kawa.functions.LProcess` object. A *process-keyword-argument* can be used to set various options, as discussed below.

The *command* is the process command-line (name and arguments). It can be an array of strings, in which case those are used as the command arguments directly:

```
(run-process ["ls" "-l"])
```

The *command* can also be a single string, which is split (tokenized) into command arguments separated by whitespace. Quotation groups words together just like traditional shells:

```
(run-process "cmd a\"b 'c\"d k'l m\"n'o")
   ⇒ (run-process ["cmd"   "ab 'cd"   "k'l m\"no"])
```

The syntax shorthand ``&`{command}`` or `&sh{command}` (discussed below) is usually more convenient.

```
process-keyword-argument ::=
    process-redirect-argument
  | process-environment-argument
  | process-misc-argument
```

We discuss *process-redirect-argument* and *process-environment-argument* later. The *process-misc-argument* options are just the following:

shell: *shell* Currently, *shell* must be one of `#f` (which is ignored) or `#t`. The latter means to use an external shell to tokenize the *command*. I.e. the following are equivalent:

```
(run-process shell: #t "command")
(run-process ["/bin/sh" "-c" "command"])
```

directory: *dir*
Change the working directory of the new process to *dir*.

21.6.2 Process literals

A simple *process literal* is a kind of Section 7.11 [Named quasi-literals], page 133, that uses the backtick character (`` ` ``) as the *cname*. For example:

```
&`{date --utc}
```

This is equivalent to:

```
(run-process "date --utc")
```

In general the following are roughly equivalent (using [string quasi-literals], page 224):

```
&`[args...]{command}
(run-process args... &{command})
```

The reason for the "roughly" is if *command* contains escaped sub-expressions; in that case ``&` `` may process the resulting values differently from plain string-substitution, as discussed below.

If you use `&sh` instead of ``&` `` then a shell is used:

```
&sh{rm *.class}
```

which is equivalent to:

```
&`{/bin/sh -c "rm *.class"}
```

In general, the following are equivalent:

```
&sh[args...]{command}
&`[shell: #t args...]{command}
```

21.6.3 Process values and process output

The value returned from a call to `run-process` or a process literal is an instance of `gnu.kawa.functions.LProcess`. This class extends `java.lang.Process`, so you can treat it as any other `Process` object.

```
#|kawa:1|# (define p1 &`{date --utc})
#|kawa:2|# (p1:toString)
gnu.kawa.functions.LProcess@377dca04
#|kawa:3|# (write p1)
gnu.kawa.functions.LProcess@377dca04
```

What makes an `LProcess` interesting is that it is also a [Blobs], page 275, which is automatically converted to a string (or bytevector) in a context that requires it. The contents of the blob comes from the standard output of the process. The blob is evaluated Section 8.6 [Lazy evaluation], page 144, so data it is only collected when requested.

```
#|kawa:4|# (define s1 ::string p1)
#|kawa:5|# (write s1)
"Wed Jan  1 01:18:21 UTC 2014\n"
#|kawa:6|# (define b1 ::bytevector p1)
(write b1)
#u8(87 101 100 32 74 97 110 ... 52 10)
```

The `display` procedure prints it in "human" form, as a string:

```
#|kawa:7|# (display p1)
Wed Jan  1 01:18:21 UTC 2014
```

This is also the default REPL formatting:

```
#|kawa:8|# &`{date --utc}
Wed Jan  1 01:18:22 UTC 2014
```

When you type a command to a shell, its output goes to the console, Similarly, in a REPL the output from the process is copied to the console output - which can sometimes by optimized by letting the process inherit its standard output from the Kawa process.

21.6.4 Substitution and tokenization

To substitute the variable or the result of an expression in the command line use the usual syntax for quasi literals:

```
(define filename (make-temporary-file))
&sh{run-experiment >&[filename]}
```

Since a process is convertible a string, we need no special syntax for command substitution:

```
`{echo The directory is: &[&`{pwd}]}
```

or equivalently:

```
`{echo The directory is: &`{pwd}}
```

Things get more interesting when considering the interaction between substitution and tokenization. This is not simple string interpolation. For example, if an interpolated value contains a quote character, we want to treat it as a literal quote, rather than a token delimiter. This matches the behavior of traditional shells. There are multiple cases, depending on whether the interpolation result is a string or a vector/list, and depending on whether the interpolation is inside quotes.

- If the value is a string, and we're not inside quotes, then all non-whitespace characters (including quotes) are literal, but whitespace still separates tokens:

  ```
  (define v1 "a b'c ")
  &`{cmd x y&[v1]z}   ⇒  (run-process ["cmd" "x" "ya" "b'c" "z"])
  ```

- If the value is a string, and we are inside single quotes, all characters (including whitespace) are literal.

  ```
  &`{cmd 'x y&[v1]z'}   ⇒  (run-process ["cmd" "x ya b'c z"])
  ```

Double quotes work the same except that newline is an argument separator. This is useful when you have one filename per line, and the filenames may contain spaces, as in the output from `find`:

```
&`{ls -l "&`{find . -name '*.pdf'}"}
```

This solves a problem that is quite painful with traditional shells.

- If the value is a vector or list (of strings), and we're not inside quotes, then each element of the array becomes its own argument, as-is:

  ```
  (define v2 ["a b" "c\"d"])
  &`{cmd &[v2]}  ⇒  (run-process ["cmd" "a b" "c\"d"])
  ```

  However, if the enclosed expression is adjacent to non-space non-quote characters, those are prepended to the first element, or appended to the last element, respectively.

  ```
  &`{cmd x&[v2]y}   ⇒  (run-process ["cmd" "xa b" "c\"dy"])
  &`{cmd x&[[]]y}   ⇒  (run-process ["cmd" "xy"])
  ```

  This behavior is similar to how shells handle `"$@"` (or `"${name[@]}"` for general arrays), though in Kawa you would leave off the quotes.

  Note the equivalence:

  ```
  &`{&[array]}   ⇒  (run-process array)
  ```

- If the value is a vector or list (of strings), and we *are* inside quotes, it is equivalent to interpolating a single string resulting from concatenating the elements separated by a space:

  ```
  &`{cmd "&[v2]"}
   ⇒  (run-process ["cmd" "a b c\"d"])
  ```

  This behavior is similar to how shells handle `"$*"` (or `"${name[*]}"` for general arrays).

- If the value is the result of a call to `unescaped-data` then it is parsed as if it were literal. For example a quote in the unescaped data may match a quote in the literal:

  ```
  (define vu (unescaped-data "b ' c d '"))
  &`{cmd 'a &[vu]z'}   ⇒  (run-process ["cmd" "a b " "c" "d" "z"])
  ```

- If we're using a shell to tokenize the command, then we add quotes or backslashes as needed so that the shell will tokenize as described above:

  ```
  (define authors ["O'Conner" "de Beauvoir"])
  &sh{list-books &[authors]}
  ```

  The command passed to the shell is:

  ```
  list-books 'O'\''Conner' 'de Beauvoir
  ```

  Having quoting be handled by the `$construct$:sh` implementation automatically eliminates common code injection problems.

Smart tokenization only happens when using the quasi-literal forms such as `&`{command}`. You can of course use string templates with `run-process`:

```
(run-process &{echo The directory is: &`{pwd}})
```

However, in that case there is no smart tokenization: The template is evaluated to a string, and then the resulting string is tokenized, with no knowledge of where expressions were substituted.

21.6.5 Input/output redirection

You can use various keyword arguments to specify standard input, output, and error streams. For example to lower-case the text in `in.txt`, writing the result to `out.txt`, you can do:

```
&`[in-from: "in.txt" out-to: "out.txt"]{tr A-Z a-z}
```

or:

```
(run-process in-from: "in.txt" out-to: "out.txt" "tr A-Z a-z")
```

A *process-redirect-argument* can be one of the following:

in: *value*
The *value* is evaluated, converted to a string (as if using `display`), and copied to the input file of the process. The following are equivalent:

```
&`[in: "text\n"]{command}
&`[in: &`{echo "text"}]{command}
```

You can pipe the output from `command1` to the input of `command2` as follows:

```
&`[in: &`{command1}]{command2}
```

in-from: *path*
The process reads its input from the specified *path*, which can be any value coercible to a `filepath`.

out-to: *path*
The process writes its output to the specified *path*.

err-to: *path*
Similarly for the error stream.

out-append-to: *path*
err-append-to: *path*
Similar to `out-to` and `err-to`, but append to the file specified by *path*, instead of replacing it.

in-from: flpipe
out-to: flpipe
err-to: flpipe
Does not set up redirection. Instead, the specified stream is available using the methods `getOutputStream`, `getInputStream`, or `getErrorStream`, respectively, on the resulting `Process` object, just like Java's `ProcessBuilder.Redirect.PIPE`.

in-from: flinherit
out-to: flinherit
err-to: flinherit
Inherits the standard input, output, or error stream from the current JVM process.

out-to: *port*
err-to: *port*
Redirects the standard output or error of the process to the specified *port*.

out-to: flcurrent
err-to: flcurrent

Same as `out-to: (current-output-port)`, or `err-to: (current-error-port)`, respectively.

in-from: *port*
in-from: flcurrent

Re-directs standard input to read from the *port* (or `(current-input-port)`). It is unspecified how much is read from the *port*. (The implementation is to use a thread that reads from the port, and sends it to the process, so it might read to the end of the port, even if the process doesn't read it all.)

err-to: flout

Redirect the standard error of the process to be merged with the standard output.

The default for the error stream (if neither `err-to` or `err-append-to` is specified) is equivalent to `err-to: 'current`.

Note: Writing to a port is implemented by copying the output or error stream of the process. This is done in a thread, which means we don't have any guarantees when the copying is finished. (In the future we might change `process-exit-wait` (discussed later) wait for not only the process to finish, but also for these helper threads to finish.)

A here document (`https://en.wikipedia.org/wiki/Here_document`) is a form a literal string, typically multi-line, and commonly used in shells for the standard input of a process. You can use string literals or [string quasi-literals], page 224, for this. For example, this passes the string `"line1\nline2\nline3\n"` to the standard input of `command`:

```
(run-process [in: &{
    &|line1
    &|line2
    &|line3
    }] "command")
```

Note the use of `&|` to mark the end of ignored indentation.

21.6.6 Pipe-lines

Piping the output of one process as the input of another is in principle easy - just use the `in:` process argument. However, writing a multi-stage pipe-line quickly gets ugly:

```
&`[in: &`[in: "My text\n"]{tr a-z A-Z}]{wc}
```

The convenience macro `pipe-process` makes this much nicer:

```
(pipe-process
  "My text\n"
  &`{tr a-z A-Z}
  &`{wc})
```

`pipe-process` *input process** [Syntax]

All of the *process* expressions must be `run-process` forms, or equivalent ``&`{command}`` forms. The result of evaluating *input* becomes the input to the first *process*; the output from the first *process* becomes the input to the second *process*, and so on. The result of whole `pipe-process` expression is that of the last *process*.

Copying the output of one process to the input of the next is optimized: it uses a copying loop in a separate thread. Thus you can safely pipe long-running processes that produce huge output. This isn't quite as efficient as using an operating system pipe, but is portable and works pretty well.

21.6.7 Setting the process environment

By default the new process inherits the system environment of the current (JVM) process as returned by `System.getenv()`, but you can override it. A *process-environment-argument* can be one of the following:

env-*name***:** *value*

In the process environment, set the `"name"` to the specified *value*. For example:

```
&`[env-CLASSPATH: ".:classes"]{java MyClass}
```

*NAME***:** *value*

Same as using the `env-NAME` option above, but only if the `NAME` is uppercase (i.e. if uppercasing `NAME` yields the same string). For example the previous example could be written:

```
&`[CLASSPATH: ".:classes"]{java MyClass}
```

environment: *env*

The *env* is evaluated and must yield a `HashMap`. This map is used as the system environment of the process.

21.6.8 Waiting for process exit

When a process finishes, it returns an integer exit code. The code is traditionally 0 on successful completion, while a non-zero code indicates some kind of failure or error.

`process-exit-wait` *process* [Procedure]

The *process* expression must evaluate to a process (any `java.lang.Process` object). This procedure waits for the process to finish, and then returns the exit code as an `int`.

```
(process-exit-wait (run-process "echo foo")) ⇒ 0
```

`process-exit-ok?` *process* [Procedure]

Calls `process-exit-wait`, and then returns `#false` if the process exited it 0, and returns `#true` otherwise.

This is useful for emulating the way traditional shell do logic control flow operations based on the exit code. For example in `sh` you might write:

```
if grep Version Makefile >/dev/null
then echo found Version
else echo no Version
fi
```

The equivalent in Kawa:

```
(if (process-exit-ok? &`{grep Version Makefile})
  &`{echo found}
  &`{echo not found})
```

Strictly speaking these are not quite the same, since the Kawa version silently throws away the output from `grep` (because no-one has asked for it). To match the output from the `sh`, you can use `out-to: 'inherit`:

```
(if (process-exit-ok? &`[out-to: 'inherit]{grep Version Makefile})
  &`{echo found}
  &`{echo not found})
```

21.6.9 Exiting the current process

`exit` [*code*] [Procedure]

Exits the Kawa interpreter, and ends the Java session. Returns the value of *code* to the operating system: The *code* must be integer, or the special values `#f` (equivalent to -1), or `#t` (equivalent to 0). If *code* is not specified, zero is returned. The *code* is a status code; by convention a non-zero value indicates a non-standard (error) return.

Before exiting, finally-handlers (as in `try-finally`, or the *after* procedure of `dynamic-wind`) are executed, but only in the current thread, and only if the current thread was started normally. (Specifically if we're inside an `ExitCalled` block with non-zero nesting - see `gnu.kawa.util.ExitCalled`.) Also, JVM shutdown hooks are executed - which includes flushing buffers of output ports. (Specifically `Writer` objects registered with the `WriterManager`.)

`emergency-exit` [*code*] [Procedure]

Exits the Kawa interpreter, and ends the Java session. Communicates an exit value in the same manner as `exit`. Unlike `exit`, neither finally-handlers nor shutdown hooks are executed.

21.6.10 Deprecated functions

`make-process` *command envp* [Procedure]

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.

Except for the representation of *envp*, this is similar to:

```
(run-process environment: envp command)
```

`system` *command* [Procedure]

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`.

Equivalent to:

```
(process-exit-wait (make-process command #!null))
```

`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* [Procedure]

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.)

21.7 Time-related functions

`current-second` [Procedure]

Returns an inexact number represent the current time on the International Atomic Time (TAI) (`http://en.wikipedia.org/wiki/International_Atomic_Time`) scale. The value 0.0 represents midnight on January 1, 1070 TAI (equivalent to 10 seconds before midnight Universal Time), and the value 1.0 represents on TAI second later. Neither high acuracy nor high precision are required; in particular returning Coordinated Universal Time plus a suitable constant might be the best an implementation cat do. The Kawa implementation just multiplies by 0.001 the result of calling the method `currentTimeMillis` in class `java.lang.System`.

`current-jiffy` [Procedure]

Returns the number of *jiffies* as an exact integer that have elapses since an arbitrary implementation-defined epoch (instant). A jiffy is an implementation-defined fraction of a second which is defined by the return value of the `jiffies-per-second` procedure. The starting epoch (instant 0) is guaranteed to be constant during a run of the program, but may vary between runs. (At the time of writing, Kawa's jiffy is one nano-second.)

Rationale: Jiffies are allowed to be implementation-dependent so that `current-jiffy` can execute with minimal overhead. It should be very likely that a compactly represented integer will suffice as the return value. Any particular jiffy size will be inappropriate some some implementations: a microsecond is too long for a very fast machine, while a much smaller unit would force many implementations to return integers which have to allocated for most calls, rendering `current-jiffy` less useful for accurate timing measurements.

`jiffies-per-second` [Procedure]

Returns an exact integer representing the number of jiffies per SI second. This value is an implementation-specified constant. (At the time of writing, the value in Kawa is 1,000,000,000.)

`sleep` *time* [Procedure]

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`).

21.8 Deprecated low-level functions

These sections document older and less convenient ways to call Java methods, access Java fields, and use Java arrays.

21.8.1 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 19.9 [Method operations], page 320) 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.

21.8.2 Low-level field operations

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 19.11 [Field operations], page 327) are easier to use, more powerful, and just as efficient. However, the high-level functions currently do not provide access to non-public 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.

21.8.3 Old low-level array macros

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)`.)

22 Frequently Asked Questions

What is the equivalent of Java import?

To provide a short name for a class instead of the complete fully-qualified name use either `define-alias` (or `define-private-alias`) or the `import-class` combination. For example, to be able to write `ArrayList` instead of `java.util.ArrayList` do either:

```
(import (class java.util ArrayList))
```

or

```
(define-alias ArrayList java.util.ArrayList)
```

Using `import` is recommended: It handles errors better, and it allows you to define multiple aliases conveniently:

```
(import (class java.util Map HashMap))
```

Both forms allow renaming. For example if you want to refer to `java.lang.StringBuilder` as `StrBuf` do:

```
(import (class java.lang (StringBuilder StrBuf)))
```

or:

```
(define-alias StrBuf java.lang.StringBuilder)
```

The name(s) defined by `import` are by default private. A name defined using `define-alias` is by default exported; to avoid that use `define-private-alias` instead.

You can also use `define-namespace` to introduce an abbreviation or renaming of a class name, but as a matter of style `define-alias` is preferred.

There is no direct equivalent to Java's `import PackageOrTypeName.*` (type-import-on-demand) declaration, but you can alias a package:

```
(define-alias jutil java.util)
(define mylist :: jutil:List (jutil:ArrayList))
```

To import a static member, giving it a shortened name (like Java's static-import-on-demand declaration), you can use `define-alias`. For example:

```
(define-alias console java.lang.System:console)
```

For static fields only (not methods or member classes) you can use an `import` form, either:

```
(import (only (java lang System) out))
```

or:

```
(import (only java.lang.System out))
```

This works because Kawa can treat any class as a “library”; in which case it considers all public static fields as exported bindings.

How do I refer to a Java member (nested) class?

Consider the Java SE member class `javax.swing.text.AbstractDocument.Content`. Using the Java syntax doesn't work in Kawa. Inside you should use Kawa's colon operator:

```
javax.swing.text.AbstractDocument:Content
```

Alternatively, you can use the internal JVM class name:

```
javax.swing.text.AbstractDocument$Content
```

Why does Kawa's REPL use display rather than write?

The read-eval-print-loop of most Scheme implementations prints the evaluation result using `write`, while Kawa uses `display` by default.

First note that it is easy to override the default with the `--output-format` command-line option:

```
$kawa --output-format readable-scheme
#|kawa:1|# "abc"
"abc"
```

The reason `display` is the default is because of a vision of the REPL console as more than just printing out Scheme objects in textual form for use by a programmer. Some examples:

- A math program can display equations and graphs as the output of an expression.
- An expression can evaluate to a"Section 21.1 [Composable pictures], page 356" which would be displayed inline.
- An HTML/XML obj can be insert into the output in visual form if the console understands HTML. (There is a prototype for this that works by using the JavaFX WebView as the display.)
- The plan for "Kawa-shell" functionality is to have expressions that evaluate to process objects, which would be lazy strings. This string would be the data from standard output. Thus the effect of displaying a process object would be to print out the standard output - just like a regular shell. Users would find it confusing/annoying if shell output used quotes.

This "repl-as-pad" model doesn't work as well if the repl uses `write` rather than `display`.

23 The Kawa language framework

Kawa is a framework written in Java for implementing high-level and dynamic languages, compiling them into Java bytecodes.

The Kawa distributions includes of other programming languages besides Scheme, including XQuery (Qexo) (`../qexo/index.html`) and Emacs Lisp (JEmacs) (`http://JEmacs.sourceforge.net/`).

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

This article (`gnu.bytecode/compiling-regexps.html`) explains how to use `gnu.bytecode` to compile regular expressions to bytecode.

24 License

24.1 License for the Kawa software

The license for the Kawa software (except the optional JEmacs and BRL features - see below) is the X11/MIT license (`http://opensource.org/licenses/mit-license.php`) which is quoted below.

In the past the Kawa license was a “modified GNU GPL (General Public License)". If you find any files that contain the old license or otherwise seem to contradict the new license, please report that as a bug.

Some of the JEmacs files are based on Emacs and have a GPL license, which is incompatible with non-Free (proprietary) products. For that reason, the `gnu.jemacs.*` packages are not included any more in the standard `.jar`, or by default when building from source, to avoid surprises. To build JEmacs you have to specify the `configure` flag `--enable-jemacs` or the `ant` flag `-Denable-jemacs=true`.

Some code in `gnu/brl` and `gnu/kawa/brl` is copyright Bruce R. Lewis and Eaton Vance Management, with a modified-GPL license: no restrictions if used unmodified, but otherwise the GPL applies. These packages are no longer included by default in Kawa builds, but have to be selected with the `configure` flag `--enable-brl` or the `ant` flag `-Denable-brl=true`.

Kawa uses some math routines from fdlib's libf77, which have a AT&T Bell Laboratories and Bellcore copyright. See the source file `gnu/math/DComplex.java`.

The sorting routine in `gnu.xquery.util.OrderedTuples` is a re-implementatiomn in Java of code copyrighted by Simon Tatham.

Some of the Scheme code in `kawa/lib` and `gnu/kawa/slib` are copyright other parties, and may have slightly different license wording, but I believe none of then contradicts the main Kawa license or impose extra restrictions. Search for the word `copyright` in these directories.

Some code has been converted from other languages, with permission. This includes the `rationalize` method in `gnu/math/RatNum.java`, based on an algorithm of Alan Bawden, as expressed by Marc Feeley in C-Gambit. The concepts and algorithm of `gnu/text/PrettyWriter.java` are converted from SBCL, which is in the public domain.

24.2 License for the Kawa manual

Here is the copyright license for this manual:

Parts of this manual are copied from the R6RS (`http://www.r6rs.org/`) or R7RS (`http://www.r7rs.org/`), which both state:

> We intend this report to belong to the entire Scheme community, and so we grant permission to copy it in whole or in part without fee. In particular, we encourage implementors of Scheme to use this report as a starting point for manuals and other documentation, modifying it as necessary.

Parts of this manual were derived from the SLIB manual, copyright © 1993-1998 Todd R. Eigenschink and Aubrey Jaffer.

Parts of this manual were derived from ISO/EIC 10179:1996(E) (Document Style and Specifical Language) - unknown copyright.

This manual has quoted from SRFI-6 (Basic String Ports), which is Copyright (C) William D Clinger (1999). All Rights Reserved.

This manual has quoted from SRFI-8 (receive: Binding to multiple values), which is Copyright (C) John David Stone (1999). All Rights Reserved.

This manual has quoted from SRFI-9 (Defining Record Types) which is Copyright (C) Richard Kelsey (1999). All Rights Reserved.

This manual has quoted from SRFI-11 (Syntax for receiving multiple values), which is Copyright (C) Lars T. Hansen (1999). All Rights Reserved.

This manual has quoted from SRFI-25 (Multi-dimensional Array Primitives), which is Copyright (C) Jussi Piitulainen (2001). All Rights Reserved.

This manual has quoted from SRFI-26 (Notation for Specializing Parameters without Currying), which is Copyright (C) Sebastian Egner (2002). All Rights Reserved.

This manual has quoted from SRFI-39 (Parameter objects), which is Copyright (C) Marc Feeley (2002). All Rights Reserved.

The following notice applies to SRFI-6, SRFI-8, SRFI-9, SRFI-11, SRFI-25, SRFI-26, and SRFI-39, which are quoted in this manual, but it does not apply to the manual as a whole:

Appendix A Index