Guile Library

version 0.1.5, updated 24 September 2007

Andy Wingo (wingo at pobox.com)
Richard Todd (richardt at vzavenue.net)

Short Contents

1 (apicheck) 1
2 (config load) 2
3 (container async-queue) 3
4 (container nodal-tree) 4
5 (container delay-tree) 5
6 (debugging assert) 6
7 (debugging time) 7
8 (graph topological-sort) 8
9 (htmlprag) 9
10 (io string) 11
11 (logging logger) 12
12 (logging port-log) 17
13 (logging rotating-log) 18
14 (match-bind) 19
15 (math minima) 21
16 (math primes) 22
17 (math rationalize) 23
18 (os process) 24
19 (scheme documentation) 28
20 (scheme kwargs) 29
21 (scheme session) 31
22 (search basic) 33
23 (statprof) 34
24 (string completion) 37
25 (string soundex) 39
26 (string transform) 40
27 (string wrap) 42
28 (sxml apply-templates) 44
29 (sxml fold) 45
30 (sxml simple) 47
31 (sxml ssax) 48
32 (sxml ssax input-parse) 51
33 (sxml transform) 52
34 (sxml xpath) 54
35 (term ansi-color) 56

36 (texinfo) 57
37 (texinfo docbook) 60
38 (texinfo html) 62
39 (texinfo indexing) 63
40 (texinfo nodal-tree) 64
41 (texinfo plain-text) 65
42 (texinfo serialize) 66
43 (texinfo reflection) 67
44 (text parse-lalr) 69
45 (unit-test) 72
A Copying This Manual 74
Concept Index 82
Function Index 83

1 (apicheck)

1.1 Overview

`(apicheck)` exports two routines. `apicheck-generate` produces a description of the Scheme API exported by a set of modules as an S-expression. `apicheck-validate` verifies that the API exported by a set of modules is compatible with an API description generated by `apicheck-generate`.

It would be nice to have Makefile.am fragments here, but for now, see the Guile-Library source distribution for information on how to integrate apicheck with your module's unit test suite.

1.2 Usage

`apicheck-generate` *module-names* [Function]

Generate a description of the API exported by the set of modules *module-names*.

`apicheck-validate` *api module-names* [Function]

Validate that the API exported by the set of modules *module-names* is compatible with the recorded API description *api*. Raises an exception if the interface is incompatible.

2 (config load)

2.1 Overview

```
This module needs to be documented.
```

2.2 Usage

`<configuration>` [Class]

`load-config!` [Generic]

`load-config!` (*cfg* `<configuration>`) (*commands* `<list>`) (*file-name* `<string>`) [Method]

`&config-error` [Class]

`config-error-arguments` *condition* [Function]

3 (container async-queue)

3.1 Overview

A asynchronous queue can be used to safely send messages from one thread to another.

3.2 Usage

`make-async-queue` [Function]

Create a new asynchronous queue.

`async-enqueue!` *q elt* [Function]

Enqueue *elt* into *q*.

`async-dequeue!` *q* [Function]

Dequeue a single element from *q*. If the queue is empty, the calling thread is blocked until an element is enqueued by another thread.

4 (container nodal-tree)

4.1 Overview

A nodal tree is a tree composed of nodes, each of which may have children. Nodes are represented as alists. The only alist entry that is specified is `children`, which must hold a list of child nodes. Other entries are intentionally left unspecified, so as to allow for extensibility.

4.2 Usage

`nodal-tree?` *x* [Function]

> Predicate to determine if *x* is a nodal tree. Not particularly efficient: intended for debugging purposes.

`make-node` . *attributes* [Function]

`node-ref` *node name* [Function]

`node-set!` *node name val* [Function]

`node-children` *node* [Function]

5 (container delay-tree)

5.1 Overview

A delay tree is a superset of a nodal tree (see (container nodal-tree)). It extends nodal trees to allow any entry of the node to be a promise created with the `delay` operator.

5.2 Usage

`force-ref` *node field* [Function]

Access a field in a node of a delay tree. If the value of the field is a promise, the promise will be forced, and the value will be replaced with the forced value.

6 (debugging assert)

6.1 Overview

Defines an `assert` macro, and the `cout` and `cerr` utility functions.

6.2 Usage

`assert` *expr . others* [Special Form]

Assert the truth of an expression (or of a sequence of expressions).

syntax: `assert ?expr ?expr ... [report: ?r-exp ?r-exp ...]`

If `(and ?expr ?expr ...)` evaluates to anything but `#f`, the result is the value of that expression. Otherwise, an error is reported.

The error message will show the failed expressions, as well as the values of selected variables (or expressions, in general). The user may explicitly specify the expressions whose values are to be printed upon assertion failure – as *?r-exp* that follow the identifier `report:`.

Typically, *?r-exp* is either a variable or a string constant. If the user specified no *?r-exp*, the values of variables that are referenced in *?expr* will be printed upon the assertion failure.

`cout` *. args* [Function]

Similar to `cout << arguments << args`, where *argument* can be any Scheme object. If it's a procedure (e.g. `newline`), it's called without args rather than printed.

`cerr` *. args* [Function]

Similar to `cerr << arguments << args`, where *argument* can be any Scheme object. If it's a procedure (e.g. `newline`), it's called without args rather than printed.

7 (debugging time)

7.1 Overview

Defines a macro to time execution of a body of expressions. Each element is timed individually.

7.2 Usage

`time` *expr . others* [Special Form]

syntax: `(time` *`expr1 expr2`*`...)`

Times the execution of a list of expressions, in milliseconds. The resolution is limited to guile's `internal-time-units-per-second`. Disregards the expressions' return value(s) (FIXME).

8 (graph topological-sort)

8.1 Overview

```
The algorithm is inspired by Cormen, Leiserson and Rivest (1990)
‘‘Introduction to Algorithms’’, chapter 23.
```

8.2 Usage

`topological-sort` *dag* [Function]

Returns a list of the objects in the directed acyclic graph, *dag*, topologically sorted. Objects are compared using `equal?`. The graph has the form:

```
(list (obj1 . (dependents-of-obj1))
      (obj2 . (dependents-of-obj2)) ...)
```

...specifying, for example, that `obj1` must come before all the objects in `(dependents-of-obj1)` in the sort.

`topological-sortq` *dag* [Function]

Returns a list of the objects in the directed acyclic graph, *dag*, topologically sorted. Objects are compared using `eq?`. The graph has the form:

```
(list (obj1 . (dependents-of-obj1))
      (obj2 . (dependents-of-obj2)) ...)
```

...specifying, for example, that `obj1` must come before all the objects in `(dependents-of-obj1)` in the sort.

`topological-sortv` *dag* [Function]

Returns a list of the objects in the directed acyclic graph, *dag*, topologically sorted. Objects are compared using `eqv?`. The graph has the form:

```
(list (obj1 . (dependents-of-obj1))
      (obj2 . (dependents-of-obj2)) ...)
```

...specifying, for example, that `obj1` must come before all the objects in `(dependents-of-obj1)` in the sort.

9 (htmlprag)

9.1 Overview

HtmlPrag provides permissive HTML parsing capability to Scheme programs, which is useful for software agent extraction of information from Web pages, for programmatically transforming HTML files, and for implementing interactive Web browsers. HtmlPrag emits "SHTML," which is an encoding of HTML in [SXML], so that conventional HTML may be processed with XML tools such as [SXPath] and [SXML-Tools]. Like [SSAX-HTML], HtmlPrag provides a permissive tokenizer, but also attempts to recover structure. HtmlPrag also includes procedures for encoding SHTML in HTML syntax.

The HtmlPrag parsing behavior is permissive in that it accepts erroneous HTML, handling several classes of HTML syntax errors gracefully, without yielding a parse error. This is crucial for parsing arbitrary real-world Web pages, since many pages actually contain syntax errors that would defeat a strict or validating parser. HtmlPrag's handling of errors is intended to generally emulate popular Web browsers' interpretation of the structure of erroneous HTML. We euphemistically term this kind of parse "pragmatic."

HtmlPrag also has some support for [XHTML], although XML namespace qualifiers [XML-Names] are currently accepted but stripped from the resulting SHTML. Note that valid XHTML input is of course better handled by a validating XML parser like [SSAX].

To receive notification of new versions of HtmlPrag, and to be polled for input on changes to HtmlPrag being considered, ask the author to add you to the moderated, announce-only email list, `htmlprag-announce`.

Thanks to Oleg Kiselyov and Kirill Lisovsky for their help with SXML.

9.2 Usage

`shtml-comment-symbol` [Variable]

`shtml-decl-symbol` [Variable]

`shtml-empty-symbol` [Variable]

`shtml-end-symbol` [Variable]

`shtml-entity-symbol` [Variable]

`shtml-named-char-id` [Variable]

`shtml-numeric-char-id` [Variable]

`shtml-pi-symbol` [Variable]

`shtml-start-symbol` [Variable]

`shtml-text-symbol` [Variable]

`shtml-top-symbol` [Variable]

`html->shtml` *input* [Function]

`html->sxml` *input* [Function]

`html->sxml-0nf` *input* [Function]

`html->sxml-1nf` *input* [Function]

`html->sxml-2nf` *input* [Function]

`make-html-tokenizer` *in normalized?* [Function]

`parse-html/tokenizer` *tokenizer normalized?* [Function]

`shtml->html` *shtml* [Function]

`shtml-entity-value` *entity* [Function]

`shtml-token-kind` *token* [Function]

`sxml->html` *shtml* [Function]

`test-htmlprag` [Function]

`tokenize-html` *in normalized?* [Function]

`write-shtml-as-html` *shtml out* [Function]

`write-sxml-html` *shtml out* [Function]

10 (io string)

10.1 Overview

Procedures that do io with strings.

10.2 Usage

`find-string-from-port?` *str* <*input-port*> . *max-no-char* [Function]

Looks for *str* in <*input-port*>, optionally within the first *max-no-char* characters.

11 (logging logger)

11.1 Overview

This is a logging subsystem similar to the one in the python standard library. There are two main concepts to understand when working with the logging modules. These are loggers and log handlers.

Loggers

Loggers are the front end interfaces for program logging. They can be registered by name so that no part of a program needs to be concerned with passing around loggers. In addition, a default logger can be designated so that, for most applications, the program does not need to be concerned with logger instances at all beyond the initial setup.

Log messages all flow through a logger. Messages carry with them a level (for example: 'WARNING, 'ERROR, 'CRITICAL), and loggers can filter out messages on a level basis at runtime. This way, the amount of logging can be turned up during development and bug investigation, but turned back down on stable releases.

Loggers depend on Log Handlers to actually get text to the log's destination (for example, a disk file). A single Logger can send messages through multiple Log Handlers, effectively multicasting logs to multiple destinations.

Log Handlers

Log Handlers actually route text to a destination. One or more handlers must be attached to a logger for any text to actually appear in a log.

Handlers apply a configurable transformation to the text so that it is formatted properly for the destination (for instance: syslogs, or a text file). Like the loggers, they can filter out messages based on log levels. By using filters on both the Logger and the Handlers, precise controls can be put on which log messages go where, even within a single logger.

11.2 Example use of logger

Here is an example program that sets up a logger with two handlers. One handler sends the log messages to a text log that rotates its logs. The other handler sends logs to standard error, and has its levels set so that INFO and WARN-level logs don't get through.

```
(use-modules (logging logger)
             (logging rotating-log)
             (logging port-log)
             (scheme documentation)
             (oop goops))

 ----------------------------------------------------------------------
 Support functions
 ----------------------------------------------------------------------
(define (setup-logging)
  (let ((lgr       (make <logger>))
```

```
          (rotating  (make <rotating-log>
                       #:num-files 3
                       #:size-limit 1024
                       #:file-name "test-log-file"))
          (err       (make <port-log> #:port (current-error-port))))

      ;; don't want to see warnings or info on the screen!!
      (disable-log-level! err 'WARN)
      (disable-log-level! err 'INFO)

      ;; add the handlers to our logger
      (add-handler! lgr rotating)
      (add-handler! lgr err)

      ;; make this the application's default logger
      (set-default-logger! lgr)
      (open-log! lgr)))


(define (shutdown-logging)
  (flush-log)   ;; since no args, it uses the default
  (close-log!)  ;; since no args, it uses the default
  (set-default-logger! #f))

 ----------------------------------------------------------------------
 Main code
 ----------------------------------------------------------------------
(setup-logging)

 Due to log levels, this will get to file,
 but not to stderr
(log-msg 'WARN "This is a warning.")

 This will get to file AND stderr
(log-msg 'CRITICAL "ERROR message!!!")

(shutdown-logging)
```

11.3 Usage

`<log-handler>` [Class]

This is the base class for all of the log handlers, and encompasses the basic functionality that all handlers are expected to have. Keyword arguments recognized by the `<log-handler>` at creation time are:

`#:formatter`

This optional parameter must be a function that takes three arguments: the log level, the time (as from `current-time`), and the log string itself. The function must return a string representing the formatted log.

Here is an example invokation of the default formatter, and what it's output looks like:

```
(default-log-formatter 'CRITICAL
                       (current-time)
                       "The servers are melting!")
==> "2003/12/29 14:53:02 (CRITICAL): The servers are melting!"
```

`emit-log` [Generic]

`emit-log handler str`. This method should be implemented for all the handlers. This sends a string to their output media. All level checking and formatting has already been done by `accept-log`.

`accept-log` [Generic]

`accept-log handler lvl time str`. If *lvl* is enabled for *handler*, then *str* will be formatted and sent to the log via the `emit-log` method. Formatting is done via the formatting function given at *handler*'s creation time, or by the default if none was given.

This method should not normally need to be overridden by subclasses. This method should not normally be called by users of the logging system. It is only exported so that writers of log handlers can override this behavior.

`accept-log` (*self* `<log-handler>`) (*level* `<top>`) (*time* `<top>`) (*str* `<top>`) [Method]

`<logger>` [Class]

This is the class that aggregates and manages log handlers. It also maintains the global information about which levels of log messages are enabled, and which have been suppressed. Keyword arguments accepted on creation are:

`#:handlers`

This optional parameter must be a list of objects derived from `<log-handler>`. Handlers can always be added later via `add-handler!` calls.

`add-handler!` [Generic]

`add-handler! lgr handler`. Adds *handler* to *lgr*'s list of handlers. All subsequent logs will be sent through the new handler, as well as any previously registered handlers.

`add-handler!` (*lgr* `<logger>`) (*handler* `<log-handler>`) [Method]

`log-msg` [Generic]

`log-msg [lgr] lvl arg1 arg2 ...`. Send a log message made up of the `display`'ed representation of the given arguments. The log is generated at level *lvl*, which should be a symbol. If the *lvl* is disabled, the log message is not generated. Generated log messages are sent through each of *lgr*'s handlers.

If the *lgr* parameter is omitted, then the default logger is used, if one is set.

As the args are `display`'ed, a large string is built up. Then, the string is split at newlines and sent through the log handlers as independent log messages. The reason for this behavior is to make output nicer for log handlers that prepend information like pid and timestamps to log statements.

```
;; logging to default logger, level of WARN
 (log-msg 'WARN "Warning! " x " is bigger than " y "!!!")

;; looking up a logger and logging to it
 (let ((l (lookup-logger "main")))
     (log-msg l 'CRITICAL "FAILURE TO COMMUNICATE!")
     (log-msg l 'CRITICAL "ABORTING NOW"))
```

log-msg (*lgr* <logger>) (*lvl* <top>) (*objs* <top>)... [Method]

log-msg (*lvl* <symbol>) (*objs* <top>)... [Method]

set-default-logger! *lgr* [Function]

Sets the given logger, *lgr*, as the default for logging methods where a logger is not given. *lgr* can be an instance of `<logger>`, a string that has been registered via `register-logger!`, or `#f` to remove the default logger.

With this mechanism, most applications will never need to worry about logger registration or lookup.

```
;; example 1
 (set-default-logger! "main")  ;; look up "main" logger and make it the default

;; example 2
 (define lgr (make  <logger>))
 (add-handler! lgr
              (make <port-handler>
                    #:port (current-error-port)))
 (set-default-logger! lgr)
 (log-msg 'CRITICAL "This is a message to the default logger!!!")
 (log-msg lgr 'CRITICAL "This is a message to a specific logger!!!")
```

register-logger! *str lgr* [Function]

Makes *lgr* accessible from other parts of the program by a name given in *str*. *str* should be a string, and *lgr* should be an instance of class `<logger>`.

```
 (define main-log  (make <logger>))
 (define corba-log (make <logger>))
 (register-logger! "main" main-log)
 (register-logger! "corba" corba-log)

;; in a completely different part of the program....
 (log-msg (lookup-logger "corba") 'WARNING "This is a corba warning.")
```

lookup-logger *str* [Function]

Looks up an instance of class `<logger>` by the name given in *str*. The string should have already been registered via a call to `register-logger!`.

`enable-log-level!` *lgr lvl* [Function]

Enables a specific logging level given by the symbol *lvl*, such that messages at that level will be sent to the log handlers. *lgr* can be of type `<logger>` or `<log-handler>`.

Note that any levels that are neither enabled or disabled are treated as enabled by the logging system. This is so that misspelt level names do not cause a logging blackout.

`disable-log-level!` *lgr lvl* [Function]

Disables a specific logging level, such that messages at that level will not be sent to the log handlers. *lgr* can be of type `<logger>` or `<log-handler>`.

Note that any levels that are neither enabled or disabled are treated as enabled by the logging system. This is so that misspelt level names do not cause a logging blackout.

`flush-log` [Generic]

`flush-log handler`. Tells the `handler` to output any log statements it may have buffered up. Handlers for which a flush operation doesn't make sense can choose not to implement this method. The default implementation just returns `#t`.

`flush-log (`*`lgr`*` <logger>)` [Method]

`flush-log` [Method]

`flush-log (`*`lh`*` <log-handler>)` [Method]

`open-log!` [Generic]

`open-log! handler`. Tells the `handler` to open its log. Handlers for which an open operation doesn't make sense can choose not to implement this method. The default implementation just returns `#t`.

`open-log!` [Method]

`open-log! (`*`lgr`*` <logger>)` [Method]

`open-log! (`*`lh`*` <log-handler>)` [Method]

`close-log!` [Generic]

`open-log! handler`. Tells the `handler` to close its log. Handlers for which a close operation doesn't make sense can choose not to implement this method. The default implementation just returns `#t`.

`close-log!` [Method]

`close-log! (`*`lgr`*` <logger>)` [Method]

`close-log! (`*`lh`*` <log-handler>)` [Method]

12 (logging port-log)

12.1 Overview

This module defines a log handler that writes to an arbitrary port of the user's choice. Uses of this handler could include:

- Sending logs across a socket to a network log collector.
- Sending logs to the screen
- Sending logs to a file
- Collecting logs in memory in a string port for later use

12.2 Usage

`<port-log>` [Class]

This is a log handler which writes logs to a user-provided port.

Keywords recognized by `<port-log>` on creation are:

`#:port` This is the port to which the log handler will write.

`#:formatter`
Allows the user to provide a function to use as the log formatter for this handler. See [logging logger <log-handler>], page 13, for details.

Example of creating a `<port-log>`:

```
(make <port-log> #:port (current-error-port))
```

13 (logging rotating-log)

13.1 Overview

This module defines a log handler for text logs that rotate when they get to be a user-defined size. This is similar to the behavior of many UNIX standard log files. See Chapter 11 [logging logger], page 12, for more information in general on log handlers.

13.2 Usage

`<rotating-log>` [Class]

This is a log handler which writes text logs that rotate when they reach a configurable size limit.

Keywords recognized by `<rotating-log>` on creation are:

`#:num-files`
: This is the number of log files you want the logger to use. Default is 4.

`#:size-limit`
: This is the size, in bytes, a log file must get before the logs get rotated. Default is 1MB (104876 bytes).

`#:file-name`
: This is the base of the log file name. Default is “logfile”. Numbers will be appended to the file name representing the log number. The newest log file is always “*NAME*.1”.

`#:formatter`
: Allows the user to provide a function to use as the log formatter for this handler. See [logging logger <log-handler>], page 13, for details.

Example of creating a `<rotating-log>`:

```
(make <rotating-log>
     #:num-files 3
     #:size-limit 1024
     #:file-name "test-log-file"))
```

14 (match-bind)

14.1 Overview

Utility functions and syntax constructs for dealing with regular expressions in a concise manner. Will be submitted to Guile for inclusion.

14.2 Usage

`match-bind` *regex str vars consequent . alternate* [Special Form]

Match a string against a regular expression, binding lexical variables to the various parts of the match.

vars is a list of names to which to bind the parts of the match. The first variable of the list will be bound to the entire match, so the number of variables needed will be equal to the number of open parentheses (‘(’) in the pattern, plus one for the whole match.

consequent is executed if the given expression *str* matches *regex*. If the string does not match, *alternate* will be executed if present. If *alternate* is not present, the result of `match-bind` is unspecified.

Here is a short example:

```
(define (star-indent line)
  "Returns the number of spaces until the first
   star (‘*’) in the input, or #f if the first
   non-space character is not a star."
  (match-bind "^( *)\*.*$" line (_ spaces)
              (string-length spaces)
              #f))
```

`match-bind` compiles the regular expression *regex* at macro expansion time. For this reason, *regex* must be a string literal, not an arbitrary expression.

`s///` *pat subst* [Function]

Make a procedure that performs perl-like regular expression search-and-replace on an input string.

The regular expression pattern *pat* is in the standard regular expression syntax accepted by `make-regexp`. The substitution string is very similar to perl's `s///` operator. Backreferences are indicated with a dollar sign (‘$’), and characters can be escaped with the backslash.

`s///` returns a procedure of one argument, the input string to be matched. If the string matches the pattern, it will be returned with the first matching segment replaced as per the substitution string. Otherwise the string will be returned unmodified.

Here are some examples:

```
((s/// "foo" "bar") "foo bar baz qux foo")
   ⇒ "bar bar baz qux foo"
```

```
((s/// "zag" "bar") "foo bar baz qux foo")
   ⇒ "foo bar baz qux foo"

((s/// "(f(o+)) (zag)?" "$1 $2 $3")
 "foo bar baz qux foo")
   ⇒ "foo oo bar baz qux foo"
```

`s///g` *pat subst* [Function]

Make a procedure that performs perl-like global search-and-replace on an input string.

The *pat* and *subst* arguments are as in the non-global `s///`. See [s///], page 19, for more information.

`s///g` differs from `s///` in that it does a global search and replace, not stopping at the first match.

15 (math minima)

15.1 Overview

This module contains functions for computing the minimum values of mathematical expressions on an interval.

15.2 Usage

`golden-section-search` *f x0 x1 prec* [Function]

The Golden Section Search algorithm finds minima of functions which are expensive to compute or for which derivatives are not available. Although optimum for the general case, convergence is slow, requiring nearly 100 iterations for the example (x^3-2x-5).

If the derivative is available, Newton-Raphson is probably a better choice. If the function is inexpensive to compute, consider approximating the derivative.

x0 and *x1* are real numbers. The (single argument) procedure *func* is unimodal over the open interval (*x0*, *x1*). That is, there is exactly one point in the interval for which the derivative of *func* is zero.

It returns a pair (*x* . *func*(*x*)) where *func*(*x*) is the minimum. The *prec* parameter is the stop criterion. If *prec* is a positive number, then the iteration continues until *x* is within *prec* from the true value. If *prec* is a negative integer, then the procedure will iterate -*prec* times or until convergence. If *prec* is a procedure of seven arguments, *x0*, *x1*, *a*, *b*, *fa*, *fb*, and *count*, then the iterations will stop when the procedure returns `#t`.

Analytically, the minimum of x^3-2x-5 is 0.816497.

```
(define func (lambda (x) (+ (* x (+ (* x x) -2)) -5)))
(golden-section-search func 0 1 (/ 10000))
     ==> (816.4883855245578e-3 . -6.0886621077391165)
(golden-section-search func 0 1 -5)
     ==> (819.6601125010515e-3 . -6.088637561916407)
(golden-section-search func 0 1
                       (lambda (a b c d e f g ) (= g 500)))
     ==> (816.4965933140557e-3 . -6.088662107903635)
```

16 (math primes)

16.1 Overview

This module defines functions related to prime numbers, and prime factorization.

16.2 Usage

`prime:trials` [Variable]

This is the maximum number of iterations of Solovay-Strassen that will be done to test a number for primality. The chance of error (a composite being labelled prime) is `(expt 2 (- prime:trials))`.

`prime?` *n* [Function]

Returns `#f` if *n* is composite, and `t` if it is prime. There is a slight chance, `(expt 2 (- prime:trials))`, that a composite will return `#t`.

`prime>` *start* [Function]

Return the first prime number greater than *start*. It doesn't matter if *start* is prime or composite.

`primes>` *start count* [Function]

Returns a list of the first *count* prime numbers greater than *start*.

`prime<` *start* [Function]

Return the first prime number less than *start*. It doesn't matter if *start* is prime or composite. If no primes are less than *start*, `#f` will be returned.

`primes<` *start count* [Function]

Returns a list of the first *count* prime numbers less than *start*. If there are fewer than *count* prime numbers less than *start*, then the returned list will have fewer than *start* elements.

`factor` *k* [Function]

Returns a list of the prime factors of *k*. The order of the factors is unspecified. In order to obtain a sorted list do `(sort! (factor k) <)`.

17 (math rationalize)

17.1 Overview

`Functions for rationalizing numbers, and finding simple ratios.`

17.2 Usage

`rationalize` *x e* [Function]

Returns an exact number that is within *e* of *x*. Computes the correct result for exact arguments (provided the implementation supports exact rational numbers of unlimited precision); and produces a reasonable answer for inexact arguments when inexact arithmetic is implemented using floating-point.

`find-ratio` *x e* [Function]

Returns the list of the *simplest* numerator and denominator whose quotient differs from *x* by no more than *e*.

```
(find-ratio 3/97 .0001)  ⇒ (3 97)
(find-ratio 3/97 .001)   ⇒ (1 32)
```

18 (os process)

18.1 Overview

This is a library for execution of other programs from Guile. It also allows communication using pipes (or a pseudo terminal device, but that's not currently implemented). This code originates in the `(goosh)` modules, which itself was part of goonix in one of Guile's past lives.

The following will hold when starting programs:

1. If the name of the program does not contain a `/` then the directories listed in the current `PATH` environment variable are searched to locate the program.
2. Unlike for the corresponding primitive exec procedures, e.g., `execlp`, the name of the program can not be set independently of the path to execute: the zeroth and first members of the argument vector are combined into one.

All symbols exported with the prefix `os:process:` are there in support of macros that use them. They should be ignored by users of this module.

18.2 Usage

`os:process:pipe-fork-child` *expr in-conns out-conns pipes* [Special Form]

`run+` *expr . connections* [Special Form]

Evaluate an expression in a new foreground process and wait for its completion. If no connection terms are specified, then all ports except `current-input-port`, `current-output-port` and `current-error-port` will be closed in the new process. The file descriptors underlying these ports will not be changed.

The value returned is the exit status from the new process as returned by the `waitpid` procedure.

The *keywords* and *connections* arguments are optional: see `run-concurrently+`, which is documented below. The `#:foreground` keyword is implied.

```
(run+ (begin (write (+ 2 2)) (newline) (quit 0)))
(run+ (tail-call-program "cat" "/etc/passwd"))
```

`run-concurrently+` *proc . connections* [Special Form]

Evaluate an expression in a new background process. If no connection terms are specified, then all ports except `current-input-port`, `current-output-port` and `current-error-port` will be closed in the new process. The file descriptors underlying these ports will not be changed.

The value returned in the parent is the pid of the new process.

When the process terminates its exit status can be collected using the `waitpid` procedure.

Keywords can be specified before the connection list:

`#:slave` causes the new process to be put into a new session. If `current-input-port` (after redirections) is a tty it will be assigned as the controlling terminal. This option is used when controlling a process via a pty.

`#:no-auto-close` prevents the usual closing of ports which occurs by default.

`#:foreground` makes the new process the foreground job of the controlling terminal, if the current process is using job control. (not currently implemented). The default is to place it into the background

The optional connection list can take several forms:

`(port)` usually specifies that a given port not be closed. However if `#:no-auto-close` is present it specifies instead a port which should be closed.

`(port 0)` specifies that a port be moved to a given file descriptor (e.g., 0) in the new process. The order of the two components is not significant, but one must be a number and the other must evaluate to a port. If the file descriptor is one of the standard set `(0, 1, 2)` then the corresponding standard port (e.g., `current-input-port`) will be set to the specified port.

Example:

```
(let ((p (open-input-file "/etc/passwd")))
  (run-concurrently+ (tail-call-program "cat") (p 0)))
```

`tail-call-pipeline` *. args* [Special Form]

Replace the current process image with a pipeline of connected processes.

The expressions in the pipeline are run in new background processes. The foreground process waits for them all to terminate. The exit status is derived from the status of the process at the tail of the pipeline: its exit status if it terminates normally, otherwise 128 plus the number of the signal that caused it to terminate.

The signal handlers will be reset and file descriptors set up as for `tail-call-program`. Like `tail-call-program` it does not close open ports or flush buffers.

Example:

```
(tail-call-pipeline ("ls" "/etc") ("grep" "passwd"))
```

`tail-call-pipeline+` *. args* [Special Form]

Replace the current process image with a pipeline of connected processes.

Each process is specified by an expression and each pair of processes has a connection list with pairs of file descriptors. E.g., `((1 0) (2 0))` specifies that file descriptors 1 and 2 are to be connected to file descriptor 0. This may also be written as `((1 2 0))`.

The expressions in the pipeline are run in new background processes. The foreground process waits for them all to terminate. The exit status is derived from the status of the process at the tail of the pipeline: its exit status if it terminates normally, otherwise 128 plus the number of the signal that caused it to terminate.

The signal handlers will be reset and file descriptors set up as for `tail-call-program`. Like `tail-call-program` it does not close open ports or flush buffers.

Example:

```
(tail-call-pipeline+ (tail-call-program "ls" "/etc") ((1 0))
                     (tail-call-program "grep" "passwd"))
```

`os:process:new-comm-pipes` *old-pipes out-conns* [Function]

`os:process:pipe-make-commands` *fdes port portvar* [Function]

`os:process:pipe-make-redir-commands` *connections portvar* [Function] `os:process:setup-redirected-port` *port fdes* [Function]

`run` *prog . args* [Function]

Execute *prog* in a new foreground process and wait for its completion. The value returned is the exit status of the new process as returned by the `waitpid` procedure.

Example:

```
(run "cat" "/etc/passwd")
```

`run-concurrently` *. args* [Function]

Start a program running in a new background process. The value returned is the pid of the new process.

When the process terminates its exit status can be collected using the `waitpid` procedure.

Example:

```
(run-concurrently "cat" "/etc/passwd")
```

`run-with-pipe` *mode prog . args* [Function]

Start *prog* running in a new background process. The value returned is a pair: the CAR is the pid of the new process and the CDR is either a port or a pair of ports (with the CAR containing the input port and the CDR the output port). The port(s) can be used to read from the standard output of the process and/or write to its standard input, depending on the *mode* setting. The value of *mode* should be one of `"r"`, `"w"` or `"r+"`.

When the process terminates its exit status can be collected using the `waitpid` procedure.

Example:

```
(use-modules (ice-9 rdelim)) ; needed by read-line
(define catport (cdr (run-with-pipe "r" "cat" "/etc/passwd")))
(read-line catport)
```

`tail-call-program` *prog . args* [Function]

Replace the current process image by executing *prog* with the supplied list of arguments, *args*.

This procedure will reset the signal handlers and attempt to set up file descriptors as follows:

1. File descriptor 0 is set from (current-input-port).
2. File descriptor 1 is set from (current-output-port).
3. File descriptor 2 is set from (current-error-port).

If a port can not be used (e.g., because it's closed or it's a string port) then the file descriptor is opened on the file specified by `*null-device*` instead.

Note that this procedure does not close any ports or flush output buffers. Successfully executing *prog* will prevent the normal flushing of buffers that occurs when Guile terminates. Doing otherwise would be incorrect after forking a child process, since the buffers would be flushed in both parent and child.

Examples:

```
(tail-call-program "cat" "/etc/passwd")
(with-input-from-file "/etc/passwd"
 (lambda ()
   (tail-call-program "cat")))
```

19 (scheme documentation)

19.1 Overview

Defines some macros to help in documenting macros, variables, generic functions, and classes.

19.2 Usage

`define-macro-with-docs` *name-and-args docs . body* [Special Form]

Define a macro with documentation.

`define-with-docs` *sym docs val* [Special Form]

Define a variable with documentation.

`define-generic-with-docs` *name documentation* [Special Form]

Define a generic function with documentation.

`define-class-with-docs` *name supers docs . slots* [Special Form]

Define a class with documentation.

20 (scheme kwargs)

20.1 Overview

Support for defining functions that take python-like keyword arguments. In one of his early talks, Paul Graham wrote about a large system called "Rtml":

> Most of the operators in Rtml were designed to take keyword parameters, and what a help that turned out to be. If I wanted to add another dimension to the behavior of one of the operators, I could just add a new keyword parameter, and everyone's existing templates would continue to work. A few of the Rtml operators didn't take keyword parameters, because I didn't think I'd ever need to change them, and almost every one I ended up kicking myself about later. If I could go back and start over from scratch, one of the things I'd change would be that I'd make every Rtml operator take keyword parameters.

See [lambda/kwargs], page 29, for documentation and examples.

See Section "Optional Arguments" in *Guile Reference Manual*, for more information on Guile's standard support for optional and keyword arguments. Quote taken from `http://lib.store.yahoo.net/lib/paulgraham/bbnexcerpts.txt`.

20.2 Usage

`define/kwargs` *what . body* [Special Form]

Defines a function that takes kwargs. See [scheme kwargs lambda/kwargs], page 29, for more information.

`lambda/kwargs` *BINDINGS . BODY* [Special Form]

Defines a function that takes keyword arguments.

bindings is a list of bindings, each of which may either be a symbol or a two-element symbol-and-default-value list. Symbols without specified default values will default to `#f`.

For example:

```
(define frobulate (lambda/kwargs (foo (bar 13) (baz 42))
                    (list foo bar baz)))
(frobulate) ⇒ (#f 13 42)
(frobulate #:baz 3) ⇒ (#f 13 3)
(frobulate #:foo 3) ⇒ (3 13 42)
(frobulate 3 4) ⇒ (3 4 42)
(frobulate 1 2 3) ⇒ (1 2 3)
(frobulate #:baz 2 #:bar 1) ⇒ (#f 1 2)
(frobulate 10 20 #:foo 3) ⇒ (3 20 42)
```

This function differs from the standard `lambda*` provided by Guile in that invoking the function will accept positional arguments. As an example, the `lambda/kwargs` behaves more intuitively in the following case:

```
((lambda* (#:optional (bar 42) #:key (baz 73))
   (list bar baz))
```

```
 1 2) ⇒ (1 73)
((lambda/kwargs ((bar 42) (baz 73))
   (list bar baz))
 1 2) ⇒ (1 2)
```

The fact that `lambda*` accepts the extra '`2`' argument is probably just a bug. In any case, `lambda/kwargs` does the right thing.

21 (scheme session)

21.1 Overview

The same thing as guile 1.6's `(ice-9 session)`, except with hooks that introduce extensibility to the `help` macro. The added functions are `add-value-help-handler!` and `remove-value-help-handler!`.

21.2 Usage

help [Special Form]

(help [NAME]) Prints useful information. Try '(help)'.

system-module [Special Form]

add-name-help-handler! *proc* [Function]

Adds a handler for performing 'help' on a name.

'proc' will be called with the unevaluated name as its argument. That is to say, when the user calls '(help FOO)', the name is FOO, exactly as the user types it.

The return value of 'proc' is as specified in 'add-value-help-handler!'.

add-value-help-handler! *proc* [Function]

Adds a handler for performing 'help' on a value.

'proc' will be called as (PROC NAME VALUE). 'proc' should return #t to indicate that it has performed help, a string to override the default object documentation, or #f to try the other handlers, potentially falling back on the normal behavior for 'help'.

apropos *rgx . options* [Function]

Search for bindings: apropos regexp {options= 'full 'shadow 'value}

apropos-fold *proc init rgx folder* [Function]

```
Folds PROCEDURE over bindings matching third arg REGEXP.

Result is

  (PROCEDURE MODULE1 NAME1 VALUE1
    (PROCEDURE MODULE2 NAME2 VALUE2
      ...
      (PROCEDURE MODULEn NAMEn VALUEn INIT)))

where INIT is the second arg to 'apropos-fold'.

Fourth arg FOLDER is one of

  (apropos-fold-accessible MODULE) ;fold over bindings accessible in MODULE
  apropos-fold-exported            ;fold over all exported bindings
  apropos-fold-all                 ;fold over all bindings
```

`apropos-fold-accessible` *module* [Function]

`apropos-fold-all` *fold-module init* [Function]

`apropos-fold-exported` *fold-module init* [Function]

`apropos-internal` *rgx* [Function]

Return a list of accessible variable names.

`arity` *obj* [Function]

`module-commentary` *name* [Function]

`remove-name-help-handler!` *proc* [Function]

Removes a handler for performing 'help' on a name.

See the documentation for 'add-name-help-handler' for more information.

`remove-value-help-handler!` *proc* [Function]

Removes a handler for performing 'help' on a value.

See the documentation for 'add-value-help-handler' for more information.

`source` *obj* [Function]

22 (search basic)

22.1 Overview

`This module has the classic search functions in it.`

22.2 Usage

`depth-first-search` *init done? expander* [Function]

Performs a depth-first search from initial state *init*. It will return the first state it sees for which predicate *done?* returns `#t`. It will use function *expander* to get a list of all states reacheable from a given state.

init can take any form the user wishes. This function treats it as opaque data to pass to *done?* and *expander*.

done? takes one argument, of the same type as *init*, and returns either `#t` or `#f`.

expander takes one argument, of the same type as *init*, and returns a list of states that can be reached from there.

`breadth-first-search` *init done? expander* [Function]

Performs a breadth-first search from initial state *init*. It will return the first state it sees for which predicate *done?* returns `#t`. It will use function *expander* to get a list of all states reacheable from a given state.

init can take any form the user wishes. This function treats it as opaque data to pass to *done?* and *expander*.

done? takes one argument, of the same type as *init*, and returns either `#t` or `#f`.

expander takes one argument, of the same type as *init*, and returns a list of states that can be reached from there.

`binary-search-sorted-vector` *vec target* [*cmp* = -] [*default* = #f] [Function]

Searches a sorted vector *vec* for item *target*. A binary search is employed which should find an item in O(log n) time if it is present. If *target* is found, the index into *vec* is returned.

As part of the search, the function *cmp* is applied to determine whether a vector item is less than, greater than, or equal to the *target*. If *target* cannot be found in the vector, then *default* is returned.

cmp defaults to `-`, which gives a correct comparison for vectors of numbers. *default* will be `#f` if another value is not given.

```
(binary-search-sorted-vector #(10 20 30) 20) ⇒ 1
```

23 (statprof)

23.1 Overview

`(statprof)` is intended to be a fairly simple statistical profiler for guile. It is in the early stages yet, so consider its output still suspect, and please report any bugs to `guile-devel at gnu.org`, or to me directly at `rlb at defaultvalue.org`.

A simple use of statprof would look like this:

```
(statprof-reset 0 50000 #t)
(statprof-start)
(do-something)
(statprof-stop)
(statprof-display)
```

This would reset statprof, clearing all accumulated statistics, then start profiling, run some code, stop profiling, and finally display a gprof flat-style table of statistics which will look something like this:

```
  %   cumulative     self              self    total
 time    seconds   seconds    calls  ms/call  ms/call  name
 35.29      0.23      0.23     2002     0.11     0.11  -
 23.53      0.15      0.15     2001     0.08     0.08  positive?
 23.53      0.15      0.15     2000     0.08     0.08  +
 11.76      0.23      0.08     2000     0.04     0.11  do-nothing
  5.88      0.64      0.04     2001     0.02     0.32  loop
  0.00      0.15      0.00        1     0.00   150.59  do-something
 ...
```

All of the numerical data with the exception of the calls column is statistically approximate. In the following column descriptions, and in all of statprof, "time" refers to execution time (both user and system), not wall clock time.

% time
: The percent of the time spent inside the procedure itself (not counting children).

cumulative seconds
: The total number of seconds spent in the procedure, including children.

self seconds
: The total number of seconds spent in the procedure itself (not counting children).

calls
: The total number of times the procedure was called.

self ms/call
: The average time taken by the procedure itself on each call, in ms.

total ms/call
: The average time taken by each call to the procedure, including time spent in child functions.

name
: The name of the procedure.

The profiler uses `eq?` and the procedure object itself to identify the procedures, so it won't confuse different procedures with the same name. They will show up as two different rows in the output.

Right now the profiler is quite simplistic. I cannot provide call-graphs or other higher level information. What you see in the table is pretty much all there is. Patches are welcome :-)

23.2 Implementation notes

The profiler works by setting the unix profiling signal `ITIMER_PROF` to go off after the interval you define in the call to `statprof-reset`. When the signal fires, a sampling routine is run which looks at the current procedure that's executing, and then crawls up the stack, and for each procedure encountered, increments that procedure's sample count. Note that if a procedure is encountered multiple times on a given stack, it is only counted once. After the sampling is complete, the profiler resets profiling timer to fire again after the appropriate interval.

Meanwhile, the profiler keeps track, via `get-internal-run-time`, how much CPU time (system and user – which is also what `ITIMER_PROF` tracks), has elapsed while code has been executing within a statprof-start/stop block.

The profiler also tries to avoid counting or timing its own code as much as possible.

23.3 Usage

`statprof-active?` [Function]

Returns `#t` if `statprof-start` has been called more times than `statprof-stop`, `#f` otherwise.

`statprof-start` [Function]

Start the profiler.

`statprof-stop` [Function]

Stop the profiler.

`statprof-reset` *sample-seconds sample-microseconds count-calls?* [Function]

Reset the statprof sampler interval to *sample-seconds* and *sample-microseconds*. If *count-calls?* is true, arrange to instrument procedure calls as well as collecting statistical profiling data.

Enables traps and debugging as necessary.

`statprof-accumulated-time` [Function]

Returns the time accumulated during the last statprof run.

`statprof-sample-count` [Function]

Returns the number of samples taken during the last statprof run.

`statprof-fold-call-data` *proc init* [Function]

Fold *proc* over the call-data accumulated by statprof. Cannot be called while statprof is active. *proc* should take two arguments, `(call-data prior-result)`.

Note that a given proc-name may appear multiple times, but if it does, it represents different functions with the same name.

`statprof-proc-call-data` *proc* [Function]

Returns the call-data associated with *proc*, or `#f` if none is available.

`statprof-call-data-name` *cd* [Function]

`statprof-call-data-calls` *cd* [Function]

`statprof-call-data-cum-samples` *cd* [Function]

`statprof-call-data-self-samples` *cd* [Function]

`statprof-call-data->stats` *call-data* [Function]

Returns an object of type `statprof-stats`.

`statprof-stats-proc-name` *stats* [Function]

`statprof-stats-%-time-in-proc` *stats* [Function]

`statprof-stats-cum-secs-in-proc` *stats* [Function]

`statprof-stats-self-secs-in-proc` *stats* [Function]

`statprof-stats-calls` *stats* [Function]

`statprof-stats-self-secs-per-call` *stats* [Function]

`statprof-stats-cum-secs-per-call` *stats* [Function]

`statprof-display` *. port* [Function]

Displays a gprof-like summary of the statistics collected. Unless an optional *port* argument is passed, uses the current output port.

`statprof-display-anomolies` [Function]

A sanity check that attempts to detect anomolies in statprof's statistics.

`with-statprof` *. args* [Special Form]

Profiles the expressions in its body.

Keyword arguments:

`#:loop` Execute the body *loop* number of times, or `#f` for no looping

default: `#f`

`#:hz` Sampling rate

default: `20`

`#:count-calls?`

Whether to instrument each function call (expensive)

default: `#f`

24 (string completion)

24.1 Overview

This module provides a facility that can be used to implement features such as TAB-completion in programs. A class `<string-completer>` tracks all the potential complete strings. Here is an example usage.

```
(use-modules (string completion)
             (oop goops)
             (srfi srfi-11))      ;; for the (let-values)

(define c (make <string-completer>))
(add-strings! c "you your yourself yourselves")

(let-values (((completions expansion exact? unique?) (complete c "yours")))
            (display completions)(newline)
            (display expansion) (newline)
            (display exact?)(newline)
            (display unique?)(newline))

==> ("yourself" "yourselves")
    "yoursel"
    #f
    #f
```

There are several more options for usage, which are detailed in the class and method documentation.

24.2 Usage

`<string-completer>` [Class]

This is the class that knows what the possible expansions are, and can determine the completions of given partial strings. The following are the recognized keywords on the call to `make`:

`#:strings`
: This gives the completer an initial set of strings. It is optional, and the `add-strings!` method can add strings to the completer later, whether these initial strings were given or not. The strings that follow this keyword can take any form that the `add-strings!` method can take (see below).

`#:case-sensitive?`
: This is a boolean that directs the completer to do its comparisons in a case sensiteve way or not. The default value is `#t`, for case-sensitive behavior.

`case-sensitive-completion?` [Generic]

`case-sensitive-completion? completer`. Returns `#t` if the completer is case-sensitive, and `#f` otherwise.

`case-sensitive-completion? (o <string-completer>)` [Method]

`add-strings!` [Generic]

`add-strings! completer strings`. Adds the given strings to the set of possible comletions known to *completer*. *strings* can either be a list of strings, or a single string of words separated by spaces. The order of the words given is not important.

`add-strings! (sc <string-completer>) (strings <top>)` [Method]

`all-completions` *completer str* [Function]

Returns a list of all possible completions for the given string *str*. The returned list will be in alphabetical order.

Note that users wanting to customize the completion algorithm can subclass `<string-completer>` and override this method.

`complete` [Generic]

`complete completer str`. Accepts a string, *str*, and returns four values via a `values` call. These are:

completions
: This is the same list that would be returned from a call to `all-completions`.

expansion
: This is the longest string that would have returned identical results. In other words, this is what most programs replace your string with when you press TAB once. This value will be equal to *str* if there were no known completionss.

```
 ("wonders" "wonderment" "wondering")
completed against "won" yields an expansion
of "wonder"
```

exact?
: This will be `#t` if the returned *expansion* is an exact match of one of the possible completions.

unique?
: This will be #t if there is only one possible completion. Note that when *unique?* is `#t`, then *exact?* will also be `#t`.

`complete (sc <string-completer>) (str <top>)` [Method]

25 (string soundex)

25.1 Overview

Soundex algorithm, taken from Knuth, Vol. 3 "Sorting and searching", pp 391–2

25.2 Usage

`soundex` *name* [Function]

Performs the original soundex algorithm on the input *name*. Returns the encoded string. The idea is for similar sounding sames to end up with the same encoding.

```
 (soundex "Aiza")
=> "A200"
 (soundex "Aisa")
=> "A200"
 (soundex "Aesha")
=> "A200"
```

26 (string transform)

26.1 Overview

Module '`(string transform)`' provides functions for modifying strings beyond that which is provided in the guile core and '`(srfi srfi-13)`'.

26.2 Usage

`escape-special-chars` *str special-chars escape-char* [Function]

Returns a copy of *str* with all given special characters preceded by the given *escape-char*.

special-chars can either be a single character, or a string consisting of all the special characters.

```
;; make a string regexp-safe...
 (escape-special-chars "***(Example String)***"
                      "[]()/*."
                      #\\)
=> "\\*\\*\\*\\(Example String\\)\\*\\*\\*"

;; also can escape a singe char...
 (escape-special-chars "richardt@vzavenue.net"
                      #\@
                      #\@)
=> "richardt@@vzavenue.net"
```

`transform-string` *str match? replace* [*start* = *#f*] [*end* = *#f*] [Function]

Uses *match?* against each character in *str*, and performs a replacement on each character for which matches are found.

match? may either be a function, a character, a string, or `#t`. If *match?* is a function, then it takes a single character as input, and should return '`#t`' for matches. *match?* is a character, it is compared to each string character using `char=?`. If *match?* is a string, then any character in that string will be considered a match. `#t` will cause every character to be a match.

If *replace* is a function, it is called with the matched character as an argument, and the returned value is sent to the output string via '`display`'. If *replace* is anything else, it is sent through the output string via '`display`'.

Note that te replacement for the matched characters does not need to be a single character. That is what differentiates this function from '`string-map`', and what makes it useful for applications such as converting '`#\&`' to '`"&"`' in web page text. Some other functions in this module are just wrappers around common uses of '`transform-string`'. Transformations not possible with this function should probably be done with regular expressions.

If *start* and *end* are given, they control which portion of the string undergoes transformation. The entire input string is still output, though. So, if *start* is '`5`', then the first five characters of *str* will still appear in the returned string.

```
; these two are equivalent...
 (transform-string str #\space #\-) ; change all spaces to -'s
 (transform-string str (lambda (c) (char=? #\space c)) #\-)
```

`expand-tabs` *str* [*tab-size = 8*] [Function]

Returns a copy of *str* with all tabs expanded to spaces. *tab-size* defaults to 8.

Assuming tab size of 8, this is equivalent to:

```
 (transform-string str #\tab "        ")
```

`center-string` *str* [*width = 80*] [*chr = #\space*] [*rchr = #f*] [Function]

Returns a copy of *str* centered in a field of *width* characters. Any needed padding is done by character *chr*, which defaults to '`#\space`'. If *rchr* is provided, then the padding to the right will use it instead. See the examples below. left and *rchr* on the right. The default *width* is 80. The default *lchr* and *rchr* is '`#\space`'. The string is never truncated.

```
 (center-string "Richard Todd" 24)
=> "      Richard Todd      "

 (center-string " Richard Todd " 24 #\=)
=> "===== Richard Todd ====="

 (center-string " Richard Todd " 24 #\< #\>)
=> "<<<<< Richard Todd >>>>>"
```

`left-justify-string` *str* [*width = 80*] [*chr = #\space*] [Function]

`left-justify-string str [width chr]`. Returns a copy of *str* padded with *chr* such that it is left justified in a field of *width* characters. The default *width* is 80. Unlike '`string-pad`' from srfi-13, the string is never truncated.

`right-justify-string` *str* [*width = 80*] [*chr = #\space*] [Function]

Returns a copy of *str* padded with *chr* such that it is right justified in a field of *width* characters. The default *width* is 80. The default *chr* is '`#\space`'. Unlike '`string-pad`' from srfi-13, the string is never truncated.

`collapse-repeated-chars` *str* [*chr = #\space*] [*num = 1*] [Function]

Returns a copy of *str* with all repeated instances of *chr* collapsed down to at most *num* instances. The default value for *chr* is '`#\space`', and the default value for *num* is 1.

```
 (collapse-repeated-chars "H  e  l  l  o")
=> "H e l l o"
 (collapse-repeated-chars "H--e--l--l--o" #\-)
=> "H-e-l-l-o"
 (collapse-repeated-chars "H-e--l---l----o" #\- 2)
=> "H-e--l--l--o"
```

27 (string wrap)

27.1 Overview

Module '`(string wrap)`' provides functions for formatting text strings such that they fill a given width field. A class, `<text-wrapper>`, does the work, but two convenience methods create instances of it for one-shot use, and in the process make for a more "schemey" interface. If many strings will be formatted with the same parameters, it might be better performance-wise to create and use a single `<text-wrapper>`.

27.2 Usage

`<text-wrapper>` [Class]

This class encapsulates the parameters needing to be fed to the text wrapping algorithm. The following are the recognized keywords on the call to `make`:

`#:line-width`
: This is the target length used when deciding where to wrap lines. Default is 80.

`#:expand-tabs?`
: Boolean describing whether tabs in the input should be expanded. Default is #t.

`#:tab-width`
: If tabs are expanded, this will be the number of spaces to which they expand. Default is 8.

`#:collapse-whitespace?`
: Boolean describing whether the whitespace inside the existing text should be removed or not. Default is #t.

  If text is already well-formatted, and is just being wrapped to fit in a different width, then setting this to '`#f`'. This way, many common text conventions (such as two spaces between sentences) can be preserved if in the original text. If the input text spacing cannot be trusted, then leave this setting at the default, and all repeated whitespace will be collapsed down to a single space.

`#:initial-indent`
: Defines a string that will be put in front of the first line of wrapped text. Default is the empty string, "".

`#:subsequent-indent`
: Defines a string that will be put in front of all lines of wrapped text, except the first one. Default is the empty string, "".

`#:break-long-words?`
: If a single word is too big to fit on a line, this setting tells the wrapper what to do. Defaults to #t, which will break up long words. When set to #f, the line will be allowed, even though it is longer than the defined `#:line-width`.

Here's an example of creating a `<text-wrapper>`:

```
(make <text-wrapper> #:line-width 48 #:break-long-words? #f)
```

`fill-string` [Generic]

`fill-string str keywds ...`. Wraps the text given in string *str* according to the parameters provided in *keywds*, or the default setting if they are not given. Returns a single string with the wrapped text. Valid keyword arguments are discussed with the `<text-wrapper>` class.

`fill-string tw str`. fills *str* using the instance of `<text-wrapper>` given as *tw*.

`fill-string` (*tw* `<text-wrapper>`) (*str* `<top>`) [Method]

`fill-string` (*str* `<top>`) (*keywds* `<top>`)... [Method]

`string->wrapped-lines` [Generic]

`string->wrapped-lines str keywds ...`. Wraps the text given in string *str* according to the parameters provided in *keywds*, or the default setting if they are not given. Returns a list of strings representing the formatted lines. Valid keyword arguments are discussed with the `<text-wrapper>` class.

`string->wrapped-lines tw str`. Wraps the text given in string *str* according to the given `<text-wrapper>` *tw*. Returns a list of strings representing the formatted lines. Valid keyword arguments are discussed with the `<text-wrapper>` class.

`string->wrapped-lines` (*tw* `<text-wrapper>`) (*str* `<top>`) [Method]

`string->wrapped-lines` (*str* `<top>`) (*keywds* `<top>`)... [Method]

28 (sxml apply-templates)

28.1 Overview

Pre-order traversal of a tree and creation of a new tree:

```
apply-templates:: tree x <templates> -> <new-tree>
```

where

```
<templates> ::= (<template> ...)
<template>  ::= (<node-test> <node-test> ... <node-test> . <handler>)
<node-test> ::= an argument to node-typeof? above
<handler>   ::= <tree> -> <new-tree>
```

This procedure does a *normal*, pre-order traversal of an SXML tree. It walks the tree, checking at each node against the list of matching templates.

If the match is found (which must be unique, i.e., unambiguous), the corresponding handler is invoked and given the current node as an argument. The result from the handler, which must be a `<tree>`, takes place of the current node in the resulting tree. The name of the function is not accidental: it resembles rather closely an `apply-templates` function of XSLT.

28.2 Usage

`apply-templates` *tree templates* [Function]

29 (sxml fold)

29.1 Overview

`(sxml fold)` defines a number of variants of the *fold* algorithm for use in transforming SXML trees. Additionally it defines the layout operator, `fold-layout`, which might be described as a context-passing variant of SSAX's `pre-post-order`.

29.2 Usage

`foldt` *fup fhere tree* [Function]

The standard multithreaded tree fold.

fup is of type [a] -> a. *fhere* is of type object -> a.

`fold` *proc seed list* [Function]

The standard list fold.

proc is of type a -> b -> b. *seed* is of type b. *list* is of type [a].

`foldts` *fdown fup fhere seed tree* [Function]

The single-threaded tree fold originally defined in SSAX. See Chapter 31 [(sxml ssax)], page 48, for more information.

`foldts*` *fdown fup fhere seed tree* [Function]

A variant of [foldts], page 45 that allows pre-order tree rewrites. Originally defined in Andy Wingo's 2007 paper, *Applications of fold to XML transformation*.

`fold-values` *proc list . seeds* [Function]

A variant of [fold], page 45 that allows multi-valued seeds. Note that the order of the arguments differs from that of `fold`.

`foldts*-values` *fdown fup fhere tree . seeds* [Function]

A variant of [foldts*], page 45 that allows multi-valued seeds. Originally defined in Andy Wingo's 2007 paper, *Applications of fold to XML transformation*.

`fold-layout` *tree bindings params layout stylesheet* [Function]

A traversal combinator in the spirit of SSAX's [pre-post-order], page 53.

`fold-layout` was originally presented in Andy Wingo's 2007 paper, *Applications of fold to XML transformation*.

```
bindings := (<binding>...)
binding  := (<tag> <bandler-pair>...)
          | (*default* . <post-handler>)
          | (*text* . <text-handler>)
tag      := <symbol>
handler-pair := (pre-layout . <pre-layout-handler>)
          | (post . <post-handler>)
          | (bindings . <bindings>)
          | (pre . <pre-handler>)
          | (macro . <macro-handler>)
```

pre-layout-handler

A function of three arguments:

kids the kids of the current node, before traversal

params the params of the current node

layout the layout coming into this node

pre-layout-handler is expected to use this information to return a layout to pass to the kids. The default implementation returns the layout given in the arguments.

post-handler

A function of five arguments:

tag the current tag being processed

params the params of the current node

layout the layout coming into the current node, before any kids were processed

klayout the layout after processing all of the children

kids the already-processed child nodes

post-handler should return two values, the layout to pass to the next node and the final tree.

text-handler

text-handler is a function of three arguments:

text the string

params the current params

layout the current layout

text-handler should return two values, the layout to pass to the next node and the value to which the string should transform.

30 (sxml simple)

30.1 Overview

A simple interface to XML parsing and serialization.

30.2 Usage

`xml->sxml` [*port* = (*current-input-port*)] [Function]

Use SSAX to parse an XML document into SXML. Takes one optional argument, *port*, which defaults to the current input port.

`sxml->xml` *tree* [*port* = (*current-output-port*)] [Function]

Serialize the sxml tree *tree* as XML. The output will be written to the current output port, unless the optional argument *port* is present.

`sxml->string` *sxml* [Function]

Detag an sxml tree *sxml* into a string. Does not perform any formatting.

`universal-sxslt-rules` [Variable]

A set of `pre-post-order` rules that transform any SXML tree into a form suitable for XML serialization by `(sxml transform)`'s `SRV:send-reply`. Used internally by `sxml->xml`.

31 (sxml ssax)

31.1 Overview

Functional XML parsing framework

SAX/DOM and SXML parsers with support for XML Namespaces and validation

This is a package of low-to-high level lexing and parsing procedures that can be combined to yield a SAX, a DOM, a validating parser, or a parser intended for a particular document type. The procedures in the package can be used separately to tokenize or parse various pieces of XML documents. The package supports XML Namespaces, internal and external parsed entities, user-controlled handling of whitespace, and validation. This module therefore is intended to be a framework, a set of "Lego blocks" you can use to build a parser following any discipline and performing validation to any degree. As an example of the parser construction, this file includes a semi-validating SXML parser.

The present XML framework has a "sequential" feel of SAX yet a "functional style" of DOM. Like a SAX parser, the framework scans the document only once and permits incremental processing. An application that handles document elements in order can run as efficiently as possible. *Unlike* a SAX parser, the framework does not require an application register stateful callbacks and surrender control to the parser. Rather, it is the application that can drive the framework – calling its functions to get the current lexical or syntax element. These functions do not maintain or mutate any state save the input port. Therefore, the framework permits parsing of XML in a pure functional style, with the input port being a monad (or a linear, read-once parameter).

Besides the *port*, there is another monad – *seed*. Most of the middle- and high-level parsers are single-threaded through the *seed*. The functions of this framework do not process or affect the *seed* in any way: they simply pass it around as an instance of an opaque datatype. User functions, on the other hand, can use the seed to maintain user's state, to accumulate parsing results, etc. A user can freely mix his own functions with those of the framework. On the other hand, the user may wish to instantiate a high-level parser: `SSAX:make-elem-parser` or `SSAX:make-parser`. In the latter case, the user must provide functions of specific signatures, which are called at predictable moments during the parsing: to handle character data, element data, or processing instructions (PI). The functions are always given the *seed*, among other parameters, and must return the new *seed*.

From a functional point of view, XML parsing is a combined pre-post-order traversal of a "tree" that is the XML document itself. This down-and-up traversal tells the user about an element when its start tag is encountered. The user is notified about the element once more, after all element's children have been handled. The process of XML parsing therefore is a fold over the raw XML document. Unlike a fold over trees defined in [1], the parser is necessarily single-threaded – obviously as elements in a text XML document are laid down sequentially. The parser therefore is a tree fold that has been transformed to accept an accumulating parameter [1,2].

Formally, the denotational semantics of the parser can be expressed as

```
parser:: (Start-tag -> Seed -> Seed) ->
  (Start-tag -> Seed -> Seed -> Seed) ->
  (Char-Data -> Seed -> Seed) ->
  XML-text-fragment -> Seed -> Seed
parser fdown fup fchar "<elem attrs> content </elem>" seed
 = fup "<elem attrs>" seed
(parser fdown fup fchar "content" (fdown "<elem attrs>" seed))

parser fdown fup fchar "char-data content" seed
 = parser fdown fup fchar "content" (fchar "char-data" seed)

parser fdown fup fchar "elem-content content" seed
 = parser fdown fup fchar "content" (
parser fdown fup fchar "elem-content" seed)
```

Compare the last two equations with the left fold

```
fold-left kons elem:list seed = fold-left kons list (kons elem seed)
```

The real parser created by `SSAX:make-parser` is slightly more complicated, to account for processing instructions, entity references, namespaces, processing of document type declaration, etc.

The XML standard document referred to in this module is`http://www.w3.org/TR/1998/REC-xml-1998021(`

The present file also defines a procedure that parses the text of an XML document or of a separate element into SXML, an S-expression-based model of an XML Information Set. SXML is also an Abstract Syntax Tree of an XML document. SXML is similar but not identical to DOM; SXML is particularly suitable for Scheme-based XML/HTML authoring, SXPath queries, and tree transformations. See SXML.html for more details. SXML is a term implementation of evaluation of the XML document [3]. The other implementation is context-passing.

The present frameworks fully supports the XML Namespaces Recommendation:`http://www.w3.org/TR/REC-xml-names/` Other links:

[1] Jeremy Gibbons, Geraint Jones, "The Under-appreciated Unfold," Proc. ICFP'98, 1998, pp. 273-279.

[2] Richard S. Bird, The promotion and accumulation strategies in transformational programming, ACM Trans. Progr. Lang. Systems, 6(4):487-504, October 1984.

[3] Ralf Hinze, "Deriving Backtracking Monad Transformers," Functional Pearl. Proc ICFP'00, pp. 186-197.

31.2 Usage

`xml-token?` [Function]

```
-- Scheme Procedure: pair? x
    Return '#t' if X is a pair; otherwise return '#f'.
```

`xml-token-kind` [Special Form]

`xml-token-head` [Special Form]

`make-empty-attlist` [Function]

attlist-add *syntmp-attlist-277 syntmp-name-value-278* [Function]

attlist-null? [Function]

```
-- Scheme Procedure: null? x
    Return '#t' iff X is the empty list, else '#f'.
```

attlist-remove-top *syntmp-attlist-280* [Function]

attlist->alist *syntmp-attlist-281* [Function]

attlist-fold *syntmp-kons-214 syntmp-knil-215 syntmp-lis1-216* [Function]

ssax:uri-string->symbol *syntmp-uri-str-312* [Function]

ssax:skip-internal-dtd *syntmp-port-246* [Function]

ssax:read-pi-body-as-string *syntmp-port-243* [Function]

ssax:reverse-collect-str-drop-ws *syntmp-fragments-494* [Function]

ssax:read-markup-token *syntmp-port-238* [Function]

ssax:read-cdata-body *syntmp-port-248 syntmp-str-handler-249 syntmp-seed-250* [Function]

ssax:read-char-ref *syntmp-port-260* [Function]

ssax:read-attributes *syntmp-port-301 syntmp-entities-302* [Function]

ssax:complete-start-tag *syntmp-tag-head-355 syntmp-port-356 syntmp-elems-357 syntmp-entities-358 syntmp-namespaces-359* [Function]

ssax:read-external-id *syntmp-port-370* [Function]

ssax:read-char-data *syntmp-port-387 syntmp-expect-eof?-388 syntmp-str-handler-389 syntmp-seed-390* [Function]

ssax:xml->sxml *syntmp-port-500 syntmp-namespace-prefix-assig-501* [Function]

ssax:make-elem-parser [Special Form]

ssax:make-parser [Special Form]

ssax:make-pi-parser [Special Form]

32 (sxml ssax input-parse)

32.1 Overview

A simple lexer.

The procedures in this module surprisingly often suffice to parse an input stream. They either skip, or build and return tokens, according to inclusion or delimiting semantics. The list of characters to expect, include, or to break at may vary from one invocation of a function to another. This allows the functions to easily parse even context-sensitive languages.

EOF is generally frowned on, and thrown up upon if encountered. Exceptions are mentioned specifically. The list of expected characters (characters to skip until, or break-characters) may include an EOF "character", which is to be coded as the symbol, `*eof*`.

The input stream to parse is specified as a *port*, which is usually the last (and optional) argument. It defaults to the current input port if omitted.

If the parser encounters an error, it will throw an exception to the key `parser-error`. The arguments will be of the form `(`*port message specialising-msg*`*)`.

The first argument is a port, which typically points to the offending character or its neighborhood. You can then use `port-column` and `port-line` to query the current position. *message* is the description of the error. Other arguments supply more details about the problem.

32.2 Usage

`peek-next-char` [*port* = (*current-input-port*)] [Function]

`assert-curr-char` *expected-chars comment* [*port* = (*current-input-port*)] [Function]

`skip-until` *arg* [*port* = (*current-input-port*)] [Function]

`skip-while` *skip-chars* [*port* = (*current-input-port*)] [Function]

`next-token` *prefix-skipped-chars break-chars* [*comment* = `""`] [*port* = (*current-input-port*)] [Function]

`next-token-of` *incl-list/pred* [*port* = (*current-input-port*)] [Function]

`read-text-line` [*port* = (*current-input-port*)] [Function]

`read-string` *n* [*port* = (*current-input-port*)] [Function]

33 (sxml transform)

33.1 Overview

SXML expression tree transformers

Pre-Post-order traversal of a tree and creation of a new tree

```
pre-post-order:: <tree> x <bindings> -> <new-tree>
```

where

```
 <bindings> ::= (<binding> ...)
 <binding> ::= (<trigger-symbol> *preorder* . <handler>) |
               (<trigger-symbol> *macro* . <handler>) |
(<trigger-symbol> <new-bindings> . <handler>) |
(<trigger-symbol> . <handler>)
 <trigger-symbol> ::= XMLname | *text* | *default*
 <handler> :: <trigger-symbol> x [<tree>] -> <new-tree>
```

The pre-post-order function visits the nodes and nodelists pre-post-order (depth-first). For each <Node> of the form (*name* <Node> ...), it looks up an association with the given *name* among its <*bindings*>. If failed, pre-post-order tries to locate a *default* binding. It's an error if the latter attempt fails as well. Having found a binding, the pre-post-order function first checks to see if the binding is of the form

```
(<trigger-symbol> *preorder* . <handler>)
```

If it is, the handler is 'applied' to the current node. Otherwise, the pre-post-order function first calls itself recursively for each child of the current node, with <*new-bindings*> prepended to the <*bindings*> in effect. The result of these calls is passed to the <*handler*> (along with the head of the current <*Node*>). To be more precise, the handler is _applied_ to the head of the current node and its processed children. The result of the handler, which should also be a <tree>, replaces the current <*Node*>. If the current <*Node*> is a text string or other atom, a special binding with a symbol *text* is looked up.

A binding can also be of a form

```
(<trigger-symbol> *macro* . <handler>)
```

This is equivalent to *preorder* described above. However, the result is re-processed again, with the current stylesheet.

33.2 Usage

SRV:send-reply . *fragments* [Function]

Output the *fragments* to the current output port.

The fragments are a list of strings, characters, numbers, thunks, #f, #t – and other fragments. The function traverses the tree depth-first, writes out strings and characters, executes thunks, and ignores #f and '(). The function returns #t if anything was written at all; otherwise the result is #f If #t occurs among the fragments, it is not written out but causes the result of SRV:send-reply to be #t.

foldts *fdown fup fhere seed tree* [Function]

post-order *tree bindings* [Function]

pre-post-order *tree bindings* [Function]

replace-range *beg-pred end-pred forest* [Function]

34 (sxml xpath)

34.1 Overview

SXPath: SXML Query Language

SXPath is a query language for SXML, an instance of XML Information set (Infoset) in the form of s-expressions. See `(sxml ssax)` for the definition of SXML and more details. SXPath is also a translation into Scheme of an XML Path Language, XPath. XPath and SXPath describe means of selecting a set of Infoset's items or their properties.

To facilitate queries, XPath maps the XML Infoset into an explicit tree, and introduces important notions of a location path and a current, context node. A location path denotes a selection of a set of nodes relative to a context node. Any XPath tree has a distinguished, root node – which serves as the context node for absolute location paths. Location path is recursively defined as a location step joined with a location path. A location step is a simple query of the database relative to a context node. A step may include expressions that further filter the selected set. Each node in the resulting set is used as a context node for the adjoining location path. The result of the step is a union of the sets returned by the latter location paths.

The SXML representation of the XML Infoset (see SSAX.scm) is rather suitable for querying as it is. Bowing to the XPath specification, we will refer to SXML information items as 'Nodes':

```
<Node> ::= <Element> | <attributes-coll> | <attrib>
   | "text string" | <PI>
```

This production can also be described as

```
<Node> ::= (name . <Nodeset>) | "text string"
```

An (ordered) set of nodes is just a list of the constituent nodes:

```
<Nodeset> ::= (<Node> ...)
```

Nodesets, and Nodes other than text strings are both lists. A `<Nodeset>` however is either an empty list, or a list whose head is not a symbol. A symbol at the head of a node is either an XML name (in which case it's a tag of an XML element), or an administrative name such as '`@`'. This uniform list representation makes processing rather simple and elegant, while avoiding confusion. The multi-branch tree structure formed by the mutually-recursive datatypes `<Node>` and `<Nodeset>` lends itself well to processing by functional languages.

A location path is in fact a composite query over an XPath tree or its branch. A singe step is a combination of a projection, selection or a transitive closure. Multiple steps are combined via join and union operations. This insight allows us to *elegantly* implement XPath as a sequence of projection and filtering primitives – converters – joined by *combinators*. Each converter takes a node and returns a nodeset which is the result of the corresponding query relative to that node. A converter can also be called on a set of nodes. In that case it returns a union of the corresponding queries over each node in the set. The union is easily implemented as a list append operation as all nodes in a SXML tree are considered distinct, by XPath conventions. We also preserve the order of the members in the union. Query combinators are high-order functions: they take converter(s) (which is a

Node|Nodeset -> Nodeset function) and compose or otherwise combine them. We will be concerned with only relative location paths [XPath]: an absolute location path is a relative path applied to the root node.

Similarly to XPath, SXPath defines full and abbreviated notations for location paths. In both cases, the abbreviated notation can be mechanically expanded into the full form by simple rewriting rules. In case of SXPath the corresponding rules are given as comments to a sxpath function, below. The regression test suite at the end of this file shows a representative sample of SXPaths in both notations, juxtaposed with the corresponding XPath expressions. Most of the samples are borrowed literally from the XPath specification, while the others are adjusted for our running example, tree1.

34.2 Usage

`nodeset?` *x* [Function]

`node-typeof?` *crit* [Function]

`node-eq?` *other* [Function]

`node-equal?` *other* [Function]

`node-pos` *n* [Function]

`filter` *pred?* [Function]

`take-until` *pred?* [Function]

`take-after` *pred?* [Function]

`map-union` *proc lst* [Function]

`node-reverse` *node-or-nodeset* [Function]

`node-trace` *title* [Function]

`select-kids` *test-pred?* [Function]

`node-self` *pred?* [Function]

`node-join` *. selectors* [Function]

`node-reduce` *. converters* [Function]

`node-or` *. converters* [Function]

`node-closure` *test-pred?* [Function]

`node-parent` *rootnode* [Function]

`sxpath` *path* [Function]

35 (term ansi-color)

35.1 Overview

The '(term ansi-color)' module generates ANSI escape sequences for colors. Here is an example of the module's use:

```
 method one: safer, since you know the colors
 will get reset
(display (colorize-string "Hello!\n" 'RED 'BOLD 'ON-BLUE))

 method two: insert the colors by hand
(for-each display
          (list (color 'RED 'BOLD 'ON-BLUE)
                "Hello!"
                 (color 'RESET)))
```

35.2 Usage

`color` . *lst* [Function]

Returns a string containing the ANSI escape sequence for producing the requested set of attributes.

The allowed values for the attributes are listed below. Unknown attributes are ignored.

Reset Attributes
: 'CLEAR' and 'RESET' are allowed and equivalent.

Non-Color Attributes
: 'BOLD' makes text bold, and 'DARK' reverses this. 'UNDERLINE' and 'UNDERSCORE' are equivalent. 'BLINK' makes the text blink. 'REVERSE' invokes reverse video. 'CONCEALED' hides output (as for getting passwords, etc.).

Foregrond Color Attributes
: 'BLACK', 'RED', 'GREEN', 'YELLOW', 'BLUE', 'MAGENTA', 'CYAN', 'WHITE'

Background Color Attributes
: 'ON-BLACK', 'ON-RED', 'ON-GREEN', 'ON-YELLOW', 'ON-BLUE', 'ON-MAGENTA', 'ON-CYAN', 'ON-WHITE'

`colorize-string` *str* . *color-list* [Function]

Returns a copy of *str* colorized using ANSI escape sequences according to the attributes specified in *color-list*. At the end of the returned string, the color attributes will be reset such that subsequent output will not have any colors in effect.

The allowed values for the attributes are listed in the documentation for the `color` function.

36 (texinfo)

36.1 Overview

Texinfo processing in scheme

This module parses texinfo into SXML. TeX will always be the processor of choice for print output, of course. However, although `makeinfo` works well for info, its output in other formats is not very customizable, and the program is not extensible as a whole. This module aims to provide an extensible framework for texinfo processing that integrates texinfo into the constellation of SXML processing tools.

Notes on the SXML vocabulary

Consider the following texinfo fragment:

```
@deffn Primitive set-car! pair value
This function...
@end deffn
```

Logically, the category (Primitive), name (set-car!), and arguments (pair value) are "attributes" of the deffn, with the description as the content. However, texinfo allows for `@`-commands within the arguments to an environment, like `@deffn`, which means that texinfo "attributes" are PCDATA. XML attributes, on the other hand, are CDATA. For this reason, "attributes" of texinfo `@`-commands are called "arguments", and are grouped under the special element, '%'.

Because '%' is not a valid NCName, stexinfo is a superset of SXML. In the interests of interoperability, this module provides a conversion function to replace the '%' with 'texinfo-arguments'.

36.2 Usage

`call-with-file-and-dir` *filename proc* [Function]

Call the one-argument procedure *proc* with an input port that reads from *filename*. During the dynamic extent of *proc*'s execution, the current directory will be `(dirname filename)`. This is useful for parsing documents that can include files by relative path name.

`texi-command-specs` [Variable]

A list of (*name content-model . args*)

name The name of an `@`-command, as a symbol.

content-model
A symbol indicating the syntactic type of the `@`-command:

`EMPTY-COMMAND`
No content, and no `@end` is coming

`EOL-ARGS` Unparsed arguments until end of line

`EOL-TEXT` Parsed arguments until end of line

`INLINE-ARGS`
: Unparsed arguments ending with `#\}`

`INLINE-TEXT`
: Parsed arguments ending with `#\}`

`ENVIRON`
: The tag is an environment tag, expect `@end foo`.

`TABLE-ENVIRON`
: Like ENVIRON, but with special parsing rules for its arguments.

`FRAGMENT`
: For `*fragment*`, the command used for parsing fragments of texinfo documents.

`INLINE-TEXT` commands will receive their arguments within their bodies, whereas the `-ARGS` commands will receive them in their attribute list.

`EOF-TEXT` receives its arguments in its body.

`ENVIRON` commands have both: parsed arguments until the end of line, received through their attribute list, and parsed text until the `@end`, received in their bodies.

`EOF-TEXT-ARGS` receives its arguments in its attribute list, as in `ENVIRON`.

There are four `@`-commands that are treated specially. `@include` is a low-level token that will not be seen by higher-level parsers, so it has no content-model. `@para` is the paragraph command, which is only implicit in the texinfo source. `@item` has special syntax, as noted above, and `@entry` is how this parser treats `@item` commands within `@table`, `@ftable`, and `@vtable`.

Also, indexing commands (`@cindex`, etc.) are treated specially. Their arguments are parsed, but they are needed before entering the element so that an anchor can be inserted into the text before the index entry.

args
: Named arguments to the command, in the same format as the formals for a lambda. Only present for `INLINE-ARGS`, `EOL-ARGS`, `ENVIRON`, `TABLE-ENVIRON` commands.

`texi-command-depth` *command max-depth* [Function]

Given the texinfo command *command*, return its nesting level, or `#f` if it nests too deep for *max-depth*.

Examples:

```
(texi-command-depth 'chapter 4)        ⇒ 1
(texi-command-depth 'top 4)            ⇒ 0
(texi-command-depth 'subsection 4)     ⇒ 3
(texi-command-depth 'appendixsubsec 4) ⇒ 3
(texi-command-depth 'subsection 2)     ⇒ #f
```

`texi-fragment->stexi` *string-or-port* [Function]

Parse the texinfo commands in *string-or-port*, and return the resultant stexi tree. The head of the tree will be the special command, `*fragment*`.

`texi->stexi` *port* [Function]

Read a full texinfo document from *port* and return the parsed stexi tree. The parsing will start at the `@settitle` and end at `@bye` or EOF.

`stexi->sxml` *tree* [Function]

Transform the stexi tree *tree* into sxml. This involves replacing the `%` element that keeps the texinfo arguments with an element for each argument.

FIXME: right now it just changes % to `texinfo-arguments` – that doesn't hang with the idea of making a dtd at some point

37 (texinfo docbook)

37.1 Overview

This module exports procedures for transforming a limited subset of the SXML representation of docbook into stexi. It is not complete by any means. The intention is to gather a number of routines and stylesheets so that external modules can parse specific subsets of docbook, for example that set generated by certain tools.

37.2 Usage

`*sdocbook->stexi-rules*` [Variable]

A stylesheet for use with SSAX's `pre-post-order`, which defines a number of generic rules for transforming docbook into texinfo.

`*sdocbook-block-commands*` [Variable]

The set of sdocbook element tags that should not be nested inside each other. See [sdocbook-flatten], page 60, for more information.

`filter-empty-elements` *sdocbook* [Function]

Filters out empty elements in an sdocbook nodeset. Mostly useful after running `sdocbook-flatten`.

`replace-titles` *sdocbook-fragment* [Function]

Iterate over the sdocbook nodeset *sdocbook-fragment*, transforming contiguous `refsect` and `title` elements into the appropriate texinfo sectioning command. Most useful after having run `sdocbook-flatten`.

For example:

```
(replace-titles '((refsect1) (title "Foo") (para "Bar.")))
   ⇒ '((chapter "Foo") (para "Bar."))
```

`sdocbook-flatten` *sdocbook* [Function]

"Flatten" a fragment of sdocbook so that block elements do not nest inside each other.

Docbook is a nested format, where e.g. a `refsect2` normally appears inside a `refsect1`. Logical divisions in the document are represented via the tree topology; a `refsect2` element *contains* all of the elements in its section.

On the contrary, texinfo is a flat format, in which sections are marked off by standalone section headers like `@chapter`, and block elements do not nest inside each other.

This function takes a nested sdocbook fragment *sdocbook* and flattens all of the sections, such that e.g.

```
(refsect1 (refsect2 (para "Hello")))
```

becomes

```
((refsect1) (refsect2) (para "Hello"))
```

Oftentimes (always?) sectioning elements have `<title>` as their first element child; users interested in processing the `refsect*` elements into proper sectioning elements like `chapter` might be interested in `replace-titles` and `filter-empty-elements`. See [replace-titles], page 60, and [filter-empty-elements], page 60.

Returns a nodeset, as described in Chapter 34 [sxml xpath], page 54. That is to say, this function returns an untagged list of stexi elements.

38 (texinfo html)

38.1 Overview

This module implements transformation from `stexi` to HTML. Note that the output of `stexi->shtml` is actually SXML with the HTML vocabulary. This means that the output can be further processed, and that it must eventually be serialized by [sxml simple sxml->xml], page 47. References (i.e., the `@ref` family of commands) are resolved by a *ref-resolver*. See [texinfo html add-ref-resolver!], page 62, for more information.

38.2 Usage

add-ref-resolver! *proc* [Function]

Add *proc* to the head of the list of ref-resolvers. *proc* will be expected to take the name of a node and the name of a manual and return the URL of the referent, or `#f` to pass control to the next ref-resolver in the list.

The default ref-resolver will return the concatenation of the manual name, `#`, and the node name.

stexi->shtml *tree* [Function]

Transform the stexi *tree* into shtml, resolving references via ref-resolvers. See the module commentary for more details.

urlify *str* [Function]

39 (texinfo indexing)

39.1 Overview

Given a piece of stexi, return an index of a specified variety.

Note that currently, `stexi-extract-index` doesn't differentiate between different kinds of index entries. That's a bug ;)

39.2 Usage

`stexi-extract-index` *tree manual-name kind* [Function]

Given an stexi tree *tree*, index all of the entries of type *kind*. *kind* can be one of the predefined texinfo indices (`concept`, `variable`, `function`, `key`, `program`, `type`) or one of the special symbols `auto` or `all`. `auto` will scan the stext for a `(printindex)` statement, and `all` will generate an index from all entries, regardless of type.

The returned index is a list of pairs, the CAR of which is the entry (a string) and the CDR of which is a node name (a string).

40 (texinfo nodal-tree)

40.1 Overview

This module exports a procedure to chunk a stexi doument into pieces, delimited by sectioning commands (`@chapter`, `@appendixsec`, etc.). Note that the sectioning commands must be preceded by a `@node`, a condition that the output of `(sxml texinfo)` respects.

The output is a nodal tree (see (container nodal-tree)), with the following fields defined for each node:

40.2 Usage

`stexi->nodal-tree` *stexi max-depth* [Function]

Break *stexi* into a nodal tree. Only break until sectioning identifiers of depth *max-depth*. The following fields are defined for each node:

`name` The name of the section.

`value` The content of the section, as `stexi`. The containing element is `texinfo`.

`parent` A reference to the parent node.

`children` A list of subnodes, corresponding to the subsections of the current section.

41 (texinfo plain-text)

41.1 Overview

Transformation from stexi to plain-text. Strives to re-create the output from `info`; comes pretty damn close.

41.2 Usage

`stexi->plain-text` *tree* [Function]

Transform *tree* into plain text. Returns a string.

42 (texinfo serialize)

42.1 Overview

Serialization of `stexi` to plain texinfo.

42.2 Usage

`stexi->texi` *tree* [Function]

Serialize the stexi *tree* into plain texinfo.

43 (texinfo reflection)

43.1 Overview

Routines to generare `stexi` documentation for objects and modules.

Note that in this context, an *object* is just a value associated with a location. It has nothing to do with GOOPS.

43.2 Usage

module-stexi-documentation [*#:sym-name = #f*] [*#:docs-resolver = (lambda (name def) def)*] [Function]

Return documentation for the module named *sym-name*. The documentation will be formatted as `stexi` (see Chapter 36 [texinfo], page 57).

object-stexi-documentation [*#:object = #f*] [*#:name = "[unknown]"*] [*#:force = #f*] [Function]

package-stexi-standard-copying *name version updated years copyright-holder permissions* [Function]

Create a standard texinfo `copying` section.

years is a list of years (as integers) in which the modules being documented were released. All other arguments are strings.

package-stexi-standard-titlepage *name version updated authors* [Function]

Create a standard GNU title page.

authors is a list of `(name . email)` pairs. All other arguments are strings.

Here is an example of the usage of this procedure:

```
(package-stexi-standard-titlepage
 "Foolib"
 "3.2"
 "26 September 2006"
 '(("Alyssa P Hacker" . "alyssa@example.com"))
 '(2004 2005 2006)
 "Free Software Foundation, Inc."
 "Standard GPL permissions blurb goes here")
```

package-stexi-standard-menu *name modules module-descriptions extra-entries* [Function]

Create a standard top node and menu, suitable for processing by makeinfo.

package-stexi-standard-prologue *name filename category description copying titlepage menu* [Function]

Create a standard prologue, suitable for later serialization to texinfo and .info creation with makeinfo.

Returns a list of stexinfo forms suitable for passing to `package-stexi-documentation` as the prologue. See [texinfo reflection package-stexi-documentation], page 68, [texinfo reflection package-stexi-standard-titlepage], page 67, [texinfo reflection package-stexi-standard-copying], page 67, and [texinfo reflection package-stexi-standard-menu], page 67.

`package-stexi-documentation` [*#:modules = #f*] [*#:name = #f*] [*#:filename = #f*] [*#:prologue = #f*] [*#:epilogue = #f*] [*#:module-stexi-documentation-args = (quote ())*] [Function]

Create stexi documentation for a *package*, where a package is a set of modules that is released together.

modules is expected to be a list of module names, where a module name is a list of symbols. The stexi that is returned will be titled *name* and a texinfo filename of *filename*.

prologue and *epilogue* are lists of stexi forms that will be spliced into the output document before and after the generated modules documentation, respectively. See [texinfo reflection package-stexi-standard-prologue], page 67, to create a conventional GNU texinfo prologue.

module-stexi-documentation-args is an optional argument that, if given, will be added to the argument list when `module-texi-documentation` is called. For example, it might be useful to define a `#:docs-resolver` argument.

44 (text parse-lalr)

44.1 Overview

This file contains yet another LALR(1) parser generator written in Scheme. In contrast to other such parser generators, this one implements a more efficient algorithm for computing the lookahead sets. The algorithm is the same as used in Bison (GNU yacc) and is described in the following paper:

"Efficient Computation of LALR(1) Look-Ahead Set", F. DeRemer and T. Pennello, TOPLAS, vol. 4, no. 4, october 1982.

As a consequence, it is not written in a fully functional style. In fact, much of the code is a direct translation from C to Scheme of the Bison sources.

44.2 Defining a parser

The module `(text parse-lalr)` declares a macro called `lalr-parser`:

```
(lalr-parser tokens rules ...)
```

This macro, when given appropriate arguments, generates an LALR(1) syntax analyzer. The macro accepts at least two arguments. The first is a list of symbols which represent the terminal symbols of the grammar. The remaining arguments are the grammar production rules.

44.3 Running the parser

The parser generated by the `lalr-parser` macro is a function that takes two parameters. The first parameter is a lexical analyzer while the second is an error procedure. The lexical analyzer is zero-argument function (a thunk) invoked each time the parser needs to look-ahead in the token stream. A token is usually a pair whose `car` is the symbol corresponding to the token (the same symbol as used in the grammar definition). The `cdr` of the pair is the semantic value associated with the token. For example, a string token would have the `car` set to `'string` while the `cdr` is set to the string value `"hello"`. Once the end of file is encountered, the lexical analyzer must always return the symbol `'*eoi*` each time it is invoked. The error procedure must be a function that accepts at least two parameters.

44.4 The grammar format

The grammar is specified by first giving the list of terminals and the list of non-terminal definitions. Each non-terminal definition is a list where the first element is the non-terminal and the other elements are the right-hand sides (lists of grammar symbols). In addition to this, each rhs can be followed by a semantic action. For example, consider the following (yacc) grammar for a very simple expression language:

```
e : e '+' t
  | e '-' t
  | t
  ;
t : t '*' f
  : t '/' f
```

```
      | f
      ;
    f : ID
      ;
```

The same grammar, written for the scheme parser generator, would look like this (with semantic actions)

```
(define expr-parser
  (lalr-parser
   ; Terminal symbols
   (ID + - * /)
   ; Productions
   (e (e + t)    : (+ $1 $3)
      (e - t)    : (- $1 $3)
      (t)        : $1)
   (t (t * f)    : (* $1 $3)
      (t / f)    : (/ $1 $3)
      (f)        : $1)
   (f (ID)       : $1)))
```

In semantic actions, the symbol `$n` refers to the synthesized attribute value of the nth symbol in the production. The value associated with the non-terminal on the left is the result of evaluating the semantic action (it defaults to `#f`). The above grammar implicitly handles operator precedences. It is also possible to explicitly assign precedences and associativity to terminal symbols and productions a la Yacc. Here is a modified (and augmented) version of the grammar:

```
(define expr-parser
 (lalr-parser
  ; Terminal symbols
  (ID
   (left: + -)
   (left: * /)
   (nonassoc: uminus))
  (e (e + e)              : (+ $1 $3)
     (e - e)              : (- $1 $3)
     (e * e)              : (* $1 $3)
     (e / e)              : (/ $1 $3)
     (- e (prec: uminus)) : (- $2)
     (ID)                 : $1)))
```

The `left:` directive is used to specify a set of left-associative operators of the same precedence level, the `right:` directive for right-associative operators, and `nonassoc:` for operators that are not associative. Note the use of the (apparently) useless terminal `uminus`. It is only defined in order to assign to the penultimate rule a precedence level higher than that of `*` and `/`. The `prec:` directive can only appear as the last element of a rule. Finally, note that precedence levels are incremented from left to right, i.e. the precedence level of `+` and `-` is less than the precedence level of `*` and `/` since the formers appear first in the list of terminal symbols (token definitions).

44.5 A final note on conflict resolution

Conflicts in the grammar are handled in a conventional way. In the absence of precedence directives, Shift/Reduce conflicts are resolved by shifting, and Reduce/Reduce conflicts are resolved by choosing the rule listed first in the grammar definition. You can print the states of the generated parser by evaluating `(print-states)`. The format of the output is similar to the one produced by bison when given the -v command-line option.

44.6 Usage

`lalr-parser` *tokens . rules* [Special Form]

The grammar declaration special form. *tokens* is the list of token symbols, and *rules* are the grammar rules. See the module documentation for more details.

`print-states` [Function]

Print the states of a generated parser.

45 (unit-test)

45.1 Overview

45.2 Usage

`assert-equal` *expected got* [Function]

`assert-true` *got* [Function]

`assert-numeric-=` *expected got precision* [Function]

`<test-result>` [Class]

`tests-run` [Generic]

`tests-run` (*o* `<test-result>`) [Method]

`tests-failed` [Generic]

`tests-failed` (*o* `<test-result>`) [Method]

`tests-log` [Generic]

`tests-log` (*o* `<test-result>`) [Method]

`failure-messages` [Generic]

`failure-messages` (*o* `<test-result>`) [Method]

`test-started` [Generic]

`test-started` (*self* `<test-result>`) (*description* `<string>`) [Method]

`test-failed` [Generic]

`test-failed` (*self* `<test-result>`) (*description* `<string>`) [Method]

`summary` [Generic]

`summary` (*self* `<test-result>`) [Method]

`<test-case>` [Class]

`name` [Generic]

`name` (*o* `<test-suite>`) [Method]

`name` (*o* `<test-case>`) [Method]

`set-up-test` [Generic]

`set-up-test` (*self* `<test-case>`) [Method]

`tear-down-test` [Generic]

`tear-down-test` (*self* `<test-case>`) [Method]

`run` [Generic]

`run` (*self* `<test-suite>`) (*result* `<test-result>`) [Method]

`run` (*self* `<test-case>`) (*result* `<test-result>`) [Method]

`<test-suite>` [Class]

`tests` [Generic]

`tests` (*o* `<test-suite>`) [Method]

`add` [Generic]

`add` (*self* `<test-suite>`) (*suite* `<test-suite>`) [Method]

`add` (*self* `<test-suite>`) (*test* `<test-case>`) [Method]

`run-all-defined-test-cases` [Function]

`exit-with-summary` *result* [Function]

`assert-exception` *expression* [Special Form]

Appendix A Copying This Manual

This manual is covered under the GNU Free Documentation License. A copy of the FDL is provided here.

A.1 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties:

any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

A

ANSI color codes 56

C

color codes, ANSI 56

F

factors, prime 22
FDL, GNU Free Documentation License 74

G

golden section 21
Goosh module 24

H

handlers, relationship with loggers 12

L

log levels 12
loggers, relationship with handlers 12
logging 12
logs, rotating 18
logs, through ports 17

M

minimum, of a mathematical function 21

N

numbers, prime 22
numbers, prime factors of 22

P

pipeline, process 24
ports, for logging 17
prime factors 22
prime number 22
process chain 24
process, Operating System 24

S

section, golden 21

T

terminals, ANSI color codes for 56

Function Index

A

accept-log 14
add 73
add-handler! 14
add-name-help-handler! 31
add-ref-resolver! 62
add-strings! 38
add-value-help-handler! 31
all-completions 38
apicheck-generate 1
apicheck-validate 1
apply-templates 44
apropos 31
apropos-fold 31
apropos-fold-accessible 32
apropos-fold-all 32
apropos-fold-exported 32
apropos-internal 32
arity 32
assert 6
assert-curr-char 51
assert-equal 72
assert-exception 73
assert-numeric-= 72
assert-true 72
async-dequeue! 3
async-enqueue! 3
attlist->alist 50
attlist-add 50
attlist-fold 50
attlist-null? 50
attlist-remove-top 50

B

binary-search-sorted-vector 33
breadth-first-search 33

C

call-with-file-and-dir 57
case-sensitive-completion? 37, 38
center-string 41
cerr 6
close-log! 16
collapse-repeated-chars 41
color 56
colorize-string 56
complete 38
config-error-arguments 2
cout 6

D

define-class-with-docs 28
define-generic-with-docs 28
define-macro-with-docs 28
define-with-docs 28
define/kwargs 29
depth-first-search 33
disable-log-level! 16

E

emit-log 14
enable-log-level! 16
escape-special-chars 40
exit-with-summary 73
expand-tabs 41

F

factor 22
failure-messages 72
fill-string 43
filter 55
filter-empty-elements 60
find-ratio 23
find-string-from-port? 11
flush-log 16
fold 45
fold-layout 45
fold-values 45
foldt 45
foldts 45, 53
foldts* 45
foldts*-values 45
force-ref 5

G

golden-section-search 21

H

help 31
html->shtml 9
html->sxml 9
html->sxml-0nf 9
html->sxml-1nf 10
html->sxml-2nf 10

L

lalr-parser 71
lambda/kwargs 29
left-justify-string 41

load-config! 2
log-msg 14, 15
lookup-logger 15

M

make-async-queue 3
make-empty-attlist 49
make-html-tokenizer 10
make-node 4
map-union 55
match-bind 19
module-commentary 32
module-stexi-documentation 67

N

name 72
next-token 51
next-token-of 51
nodal-tree? 4
node-children 4
node-closure 55
node-eq? 55
node-equal? 55
node-join 55
node-or 55
node-parent 55
node-pos 55
node-reduce 55
node-ref 4
node-reverse 55
node-self 55
node-set! 4
node-trace 55
node-typeof? 55
nodeset? 55

O

object-stexi-documentation 67
open-log! 16
os:process:new-comm-pipes 25
os:process:pipe-fork-child 24
os:process:pipe-make-commands 25
os:process:pipe-make-redir-commands 25
os:process:setup-redirected-port 26

P

package-stexi-documentation 68
package-stexi-standard-copying 67
package-stexi-standard-menu 67
package-stexi-standard-prologue 67
package-stexi-standard-titlepage 67
parse-html/tokenizer 10
peek-next-char 51
post-order 53
pre-post-order 53
prime< 22
prime> 22
prime? 22
primes< 22
primes> 22
print-states 71

R

rationalize 23
read-string 51
read-text-line 51
register-logger! 15
remove-name-help-handler! 32
remove-value-help-handler! 32
replace-range 53
replace-titles 60
right-justify-string 41
run 26, 72
run+ 24
run-all-defined-test-cases 73
run-concurrently 26
run-concurrently+ 24
run-with-pipe 26

S

s/// 19
s///g 20
sdocbook-flatten 60
select-kids 55
set-default-logger! 15
set-up-test 72
shtml->html 10
shtml-entity-value 10
shtml-token-kind 10
skip-until 51
skip-while 51
soundex 39
source 32
SRV:send-reply 52
ssax:complete-start-tag 50
ssax:make-elem-parser 50
ssax:make-parser 50
ssax:make-pi-parser 50
ssax:read-attributes 50
ssax:read-cdata-body 50
ssax:read-char-data 50
ssax:read-char-ref 50
ssax:read-external-id 50
ssax:read-markup-token 50
ssax:read-pi-body-as-string 50
ssax:reverse-collect-str-drop-ws 50
ssax:skip-internal-dtd 50
ssax:uri-string->symbol 50
ssax:xml->sxml 50
statprof-accumulated-time 35

statprof-active? 35
statprof-call-data->stats 36
statprof-call-data-calls 36
statprof-call-data-cum-samples 36
statprof-call-data-name 36
statprof-call-data-self-samples 36
statprof-display 36
statprof-display-anomolies 36
statprof-fold-call-data 35
statprof-proc-call-data 36
statprof-reset 35
statprof-sample-count 35
statprof-start 35
statprof-stats-%-time-in-proc 36
statprof-stats-calls 36
statprof-stats-cum-secs-in-proc 36
statprof-stats-cum-secs-per-call 36
statprof-stats-proc-name 36
statprof-stats-self-secs-in-proc 36
statprof-stats-self-secs-per-call 36
statprof-stop 35
stexi->nodal-tree 64
stexi->plain-text 65
stexi->shtml 62
stexi->sxml 59
stexi->texi 66
stexi-extract-index 63
string->wrapped-lines 43
summary 72
sxml->html 10
sxml->string 47
sxml->xml 47
sxpath 55
system-module 31

T

tail-call-pipeline 25
tail-call-pipeline+ 25
tail-call-program 26
take-after 55
take-until 55
tear-down-test 72
test-failed 72
test-htmlprag 10
test-started 72
tests 73
tests-failed 72
tests-log 72
tests-run 72
texi->stexi 59
texi-command-depth 58
texi-fragment->stexi 58
time 7
tokenize-html 10
topological-sort 8
topological-sortq 8
topological-sortv 8
transform-string 40

U

urlify 62

W

with-statprof 36
write-shtml-as-html 10
write-sxml-html 10

X

xml->sxml 47
xml-token-head 49
xml-token-kind 49
xml-token? 49