Web: PLT Web Applications

Version 4.2.1

Jay McCarthy

July 30, 2009

This manual describes the PLT libraries for building Web applications.

§1 “Running Web Servlets” describes how to run the servlets you’ve written.

§2 “Stateful Servlets” and §3 “Stateless Servlets” describe two ways to write Web applications. §2 “Stateful Servlets” use the entire PLT Scheme language, but their continuations are stored in the Web server’s memory. §3 “Stateless Servlets” use a slightly restricted PLT Scheme language, but their continuation can be stored by the Web client or on a Web server’s disk. If you can, you want to use §3 “Stateless Servlets” for the improved scalability.

The §4 “HTTP: Hypertext Transfer Protocol” section describes the common library function for manipulating HTTP requests and creating HTTP responses. In particular, this section covers cookies, authentication, and request bindings.

The final three sections (§5 “URL-Based Dispatch”, §6 “Formlets: Functional Form Abstraction”, and §7 “Templates: Separation of View”) cover utility libraries that ease the creation of typical Web applications.

This manual closes with a frequently asked questions section: §8 “Troubleshooting and Tips”.

1 Running Web Servlets

There are a number of ways to run Web servlets.

1.1 Instant Servlets

```
#lang web-server/insta
```

The fastest way to get a servlet running in the Web server is to use the ”Insta” language in DrScheme. Enter the following into DrScheme:

```
#lang web-server/insta

(define (start request)
  `(html (head (title "Hello world!"))
         (body (p "Hey out there!"))))
```

And press Run. A Web browser will open up showing your new servlet. This servlet will only be accessible from your local machine.

Behind the scenes, DrScheme has used `serve/servlet` to start a new server that uses your `start` function as the servlet. You are given the entire `web-server/servlet` API.

The following API is provided to customize the server instance:

`(no-web-browser)` → `void`

Calling this will instruct DrScheme to *not* start a Web browser when you press Run.

`(static-files-path` *`path`*`)` → `void`
  `path` : `path-string?`

This instructs the Web server to serve static files, such as stylesheet and images, from *`path`*.

1.2 Simple Single Servlet Servers

```
(require web-server/servlet-env)
```

The Web Server provides a way to quickly configure and start a servlet.

Here’s a simple example:

```
#lang scheme
(require web-server/servlet
         web-server/servlet-env)

(define (my-app request)
  `(html (head (title "Hello world!"))
         (body (p "Hey out there!"))))

(serve/servlet my-app)
```

Suppose you'd like to change the port to something else, change the last line to:

```
(serve/servlet my-app
               #:port 8080)
```

Suppose you want to accept connections from external machines:

```
(serve/servlet my-app
               #:listen-ip #f)
```

By default the URL for your servlet is `"http://localhost:8000/servlets/standalone.ss"`, suppose you wanted it to be `"http://localhost:8000/hello.ss"`:

```
(serve/servlet my-app
               #:servlet-path "/hello.ss")
```

Suppose you wanted it to capture top-level requests:

```
(serve/servlet my-app
               #:servlet-path "/")
```

Or, perhaps just some nice top-level name:

```
(serve/servlet my-app
               #:servlet-path "/main")
```

Suppose you wanted to use a style-sheet (`"style.css"`) found on your Desktop (`"/Users/jay/Desktop/"`):

```
(serve/servlet my-app
               #:extra-files-paths
               (list
                (build-path "/Users/jay/Desktop")))
```

These files are served *in addition* to those from the `#:server-root-path` `"htdocs"` directory. You may pass any number of extra paths.

If you want to use `serve/servlet` in a start up script for a Web application, and don't want

a browser opened or the DrScheme banner printed, then you can write:

```
(serve/servlet my-app
               #:command-line? #t)
```

Suppose you would like to start a server for a stateless Web servlet `"servlet.ss"` that provides `start`:

```
#lang scheme
(require "servlet.ss"
         web-server/servlet-env)

(serve/servlet start #:stateless? #t)
```

Warning: If you put the call to `serve/servlet` in a `web-server` module directly it will not work correctly. Consider the following module:

```
#lang web-server
(require web-server/servlet-env)

(define (start req)
  (start
   (send/suspend
    (lambda (k-url)
      `(html (body (a ([href ,k-url]) "Hello world!")))))))

(serve/servlet start #:stateless? #t)
```

First, if this module is not saved in a file (e.g., `"servlet.ss"`), then the serialization layer cannot locate the definitions of the serialized continuations. Second, due to an unfortunately subtle bug that we have not yet corrected, every time the continuation link is clicked, `serve/servlet` will run and attempt to start a Web server instance and open a browser window. These problems do not occur if your servlet is saved in a file and if `serve/servlet` is run in another module.

```
(serve/servlet
 start
 [#:command-line? command-line?
  #:launch-browser? launch-browser?
  #:quit? quit?
  #:banner? banner?
  #:listen-ip listen-ip
  #:port port
  #:servlet-path servlet-path
  #:servlet-regexp servlet-regexp
  #:stateless? stateless?
  #:stuffer stuffer
  #:manager manager
  #:servlet-namespace servlet-namespace
  #:server-root-path server-root-path
  #:extra-files-paths extra-files-paths
  #:servlets-root servlets-root
  #:servlet-current-directory servlet-current-directory
  #:file-not-found-responder file-not-found-responder
  #:mime-types-path mime-types-path
  #:ssl? ssl?
  #:ssl-cert ssl-cert
  #:ssl-key ssl-key
  #:log-file log-file
  #:log-format log-format])
→ void
 start : (request? . -> . response/c)
 command-line? : boolean? = #f
 launch-browser? : boolean? = (not command-line?)
 quit? : boolean? = (not command-line?)
 banner? : boolean? = (not command-line?)
 listen-ip : (or/c false/c string?) = "127.0.0.1"
 port : number? = 8000
 servlet-path : string? = "/servlets/standalone.ss"
 servlet-regexp : regexp? = (regexp
                             (format
                              "^~a$"
                              (regexp-quote servlet-path)))
 stateless? : boolean? = #f
 stuffer : (stuffer/c serializable? bytes?) = default-stuffer
 manager : manager?
         = (make-threshold-LRU-manager #f (* 128 1024 1024))
 servlet-namespace : (listof module-path?) = empty
 server-root-path : path-string? = default-server-root-path
 extra-files-paths : (listof path-string?)
                   = (list (build-path server-root-path "htdocs"))
 servlets-root : path-string?
               = (build-path server-root-path "htdocs")
 servlet-current-directory : path-string? = servlets-root
 file-not-found-responder : (request? . -> . response/c)
                          = (gen-file-not-found-responder
                             (build-path
                              server-root-path
                              "conf"
                              "not-found.html"))
```

This sets up and starts a fairly default server instance.

start is loaded as a servlet and responds to requests that match *servlet-regexp*. The current directory of servlet execution is *servlet-current-directory*.

If *launch-browser?* is true, then a web browser is opened to "http://localhost:<port><servlet-path>".

If *quit?* is true, then the URL "/quit" ends the server.

If *stateless?* is true, then the servlet is run as a stateless

```
#lang web-server
```

module and *stuffer* is used as the stuffer.

Advanced users may need the following options:

The server listens on *listen-ip* and port *port*. If *listen-ip* is #f, then the server accepts connections to all of the listening machine's addresses. Otherwise, the server accepts connections only at the interface(s) associated with the given string. For example, providing "127.0.0.1" (the default) as *listen-ip* creates a server that accepts only connections to "127.0.0.1" (the loopback interface) from the local machine.

If *ssl-cert* and *ssl-key* are not false, then the server runs in HTTPS mode with *ssl-cert* and *ssl-key* as the certificates and private keys.

The servlet is loaded with *manager* as its continuation manager. (The default manager limits the amount of memory to 64 MB and deals with memory pressure as discussed in the make-threshold-LRU-manager documentation.)

The modules specified by *servlet-namespace* are shared with other servlets.

The server files are rooted at *server-root-path* (which is defaultly the distribution root.) File paths, in addition to the "htdocs" directory under *server-root-path* may be provided with *extra-files-paths*. These paths are checked first, in the order they appear in the list.

Other servlets are served from *servlets-root*.

If a file cannot be found, *file-not-found-responder* is used to generate an error response.

If *banner?* is true, then an informative banner is printed. You may want to use this when running from the command line, in which case the *command-line?* option controls similar options.

MIME types are looked up at *mime-types-path*. By default the "mime.types" file in the

server-root-path is used, but if that file does not exist, then the file that ships with the Web Server is used instead. Of course, if a path is given, then it overrides this behavior.

If *log-file* is given, then it used to log requests using *log-format* as the format. Allowable formats are those allowed by log-format->format.

1.3 Command-line Tools

One command-line utility is provided with the Web Server:

```
plt-web-server [-f <file-name> -p <port> -a <ip-address> --ssl]
```

The optional file-name argument specifies the path to a configuration-table S-expression (see configuration-table->sexpr for the syntax documentation.) If this is not provided, the default configuration shipped with the server is used. The optional port and ip-address arguments override the corresponding portions of the configuration-table. If the SSL option is provided, then the server uses HTTPS with "server-cert.pem" and "private-key.pem" in the current directory, with 443 as the default port. (See the openssl module for details on the SSL implementation.)

The configuration-table is given to configuration-table->web-config@ and used to construct a web-config^ unit, and is linked with the web-server@ unit. The resulting unit is invoked, and the server runs until the process is killed.

2 Stateful Servlets

```
(require web-server/servlet)
```

2.1 Example

A stateful servlet should `provide` the following exports:

`interface-version` : `(one-of/c 'v2)`

This indicates that the servlet is a version two servlet.

`manager` : `manager?`

The manager for the continuations of this servlet. See §2.5 "Continuation Managers" for options.

`(start` *initial-request*`)` → `response/c`
  initial-request : `request?`

This function is called when an instance of this servlet is started. The argument is the HTTP request that initiated the instance.

An example version 2 module:

```
#lang scheme
(require web-server/managers/none)
(provide interface-version manager start)

(define interface-version 'v2)
(define manager
  (create-none-manager
   (lambda (req)
     `(html (head (title "No Continuations Here!"))
            (body (h1 "No Continuations Here!"))))))
(define (start req)
  `(html (head (title "Hello World!"))
         (body (h1 "Hi Mom!"))))
```

These servlets have an extensive API available to them: `net/url`, `web-server/http`, `web-server/http/bindings`, `web-server/servlet/servlet-structs`, `web-server/servlet/web`, `web-server/servlet/web-cells`, and `web-`

server/dispatch. Some of these are documented in the subsections that follow.

2.2 Common Contracts

```
(require web-server/servlet/servlet-structs)
```

This module provides a number of contracts for use in servlets.

---

k-url? : contract?

Equivalent to string?.

Example: "http://localhost:8080/servlets;1*1*20131636/examples/add.ss"

---

response-generator/c : contract?

Equivalent to (-> k-url? response/c).

Example:

```
(lambda (k-url)
  `(html
    (body
     (a ([href ,k-url])
        "Click Me to Invoke the Continuation!"))))
```

---

expiration-handler/c : contract?

Equivalent to (or/c false/c (-> request? response/c)).

Example:

```
(lambda (req)
  `(html (head (title "Expired"))
         (body (h1 "Expired")
               (p "This URL has expired. "
                  "Please return to the home page."))))
```

---

embed/url/c : contract?

Equivalent to (-> (-> request? any) string?).

This is what send/suspend/dispatch gives to its function argument.

2.3 Web Interaction

```
(require web-server/servlet/web)
```

The web-server/servlet/web library provides the primary functions of interest for the servlet developer.

(send/back *response*) → void?
  response : response/c

Sends *response* to the client. No continuation is captured, so the servlet is done.

Example:

```
(send/back
 `(html
   (body
    (h1 "The sum is: "
        ,(+ first-number
            second-number)))))
```

(send/suspend *make-response*) → request?
  make-response : (string? . -> . response/c)

Captures the current continuation, stores it with exp as the expiration handler, and binds it to a URL. *make-response* is called with this URL and is expected to generate a response/c, which is sent to the client. If the continuation URL is invoked, the captured continuation is invoked and the request is returned from this call to send/suspend.

Example:

```
(send/suspend
 (lambda (k-url)
   `(html (head (title "Enter a number"))
          (body
           (form ([action ,k-url])
                 "Enter a number: "
                 (input ([name "number"]))
                 (input ([type "submit"])))))))
```

When this form is submitted by the browser, the request will be sent to the URL generated by send/suspend. Thus, the request will be "returned" from send/suspend to the

continuation of this call.

```
(send/suspend/url make-response) → request?
  make-response : (url? . -> . response/c)
```

Like send/suspend but with a URL struct.

```
(send/suspend/dispatch make-response) → any
  make-response : (((request? . -> . any) . -> . string?) . -> . response/c)
```

Calls *make-response* with a function (embed/url) that, when called with a procedure from request? to any/c will generate a URL, that when invoked will call the function with the request? object and return the result to the caller of send/suspend/dispatch. Therefore, if you pass embed/url the identity function, send/suspend/dispatch devolves into send/suspend:

```
(define (send/suspend response-generator)
  (send/suspend/dispatch
   (lambda (embed/url)
     (response-generator (embed/url (lambda (x) x))))))
```

Use send/suspend/dispatch when there are multiple 'logical' continuations of a page. For example, we could either add to a number or subtract from it:

```
(define (count-dot-com i)
  (count-dot-com
   (send/suspend/dispatch
    (lambda (embed/url)
      `(html
        (head (title "Count!"))
        (body
         (h2 (a ([href
                  ,(embed/url
                    (lambda (req)
                      (sub1 i)))])
                "-"))
         (h1 ,(number->string i))
         (h2 (a ([href
                  ,(embed/url
                    (lambda (req)
                      (add1 i)))])
                "+")))))))))
```

Notice that in this example the result of the handlers are returned to the continuation of send/suspend/dispatch. However, it is very common that the return value of

send/suspend/dispatch is irrevelant in your application and you may think of it as "embedding" value-less callbacks. Here is the same example in this style:

```
(define (count-dot-com i)
  (send/suspend/dispatch
   (lambda (embed/url)
     `(html
       (head (title "Count!"))
       (body
        (h2 (a ([href
                 ,(embed/url
                   (lambda (req)
                     (count-dot-com (sub1 i))))])
               "-"))
        (h1 ,(number->string i))
        (h2 (a ([href
                 ,(embed/url
                   (lambda (req)
                     (count-dot-com (add1 i))))])
               "+")))))))
```

(send/suspend/url/dispatch *make-response*) → any
  make-response : (((request? . -> . any) . -> . url?) . -> . response/c)

Like send/suspend/dispatch, but with a URL struct.

(send/forward *make-response*) → request?
  make-response : (string? . -> . response/c)

Calls clear-continuation-table!, then send/suspend.

Use this if the user can logically go 'forward' in your application, but cannot go backward.

(send/finish *response*) → void?
  response : response/c

Calls clear-continuation-table!, then send/back.

Use this if the user is truly 'done' with your application. For example, it may be used to display the post-logout page:

```
(send/finish
 `(html (head (title "Logged out"))
        (body (p "Thank you for using the services "
                 "of the Add Two Numbers, Inc."))))
```

(redirect/get) → request?

Calls send/suspend with redirect-to.

This implements the Post-Redirect-Get pattern. Use this to prevent the Refresh button from duplicating effects, such as adding items to a database.

(redirect/get/forget) → request?

Calls send/forward with redirect-to.

---

current-servlet-continuation-expiration-handler : (parameter/c expiration-handler/c)

Holds the expiration-handler/c to be used when a continuation captured in this context is expired, then looked up.

Example:

```
(parameterize
    ([current-servlet-continuation-expiration-handler
      (lambda (req)
        `(html (head (title "Custom Expiration!"))))])
  (send/suspend
   ....))
```

(clear-continuation-table!) → void?

Calls the servlet's manager's clear-continuation-table! function. Normally, this deletes all the previously captured continuations.

```
(with-errors-to-browser send/finish-or-back
                        thunk)              → any
  send/finish-or-back : (response/c . -> . request?)
  thunk : (-> any)
```

Calls *thunk* with an exception handler that generates an HTML error page and calls *send/finish-or-back*.

Example:

```
(with-errors-to-browser
 send/back
 (lambda ()
   (/ 1 (get-number (request-number)))))
```

```
(adjust-timeout! t) → void?
  t : number?
```

Calls the servlet's manager's `adjust-timeout!` function.

Warning: This is deprecated and will be removed in a future release.

```
(continuation-url? u)
 → (or/c false/c (list/c number? number? number?))
  u : url?
```

Checks if `u` is a URL that refers to a continuation, if so returns the instance id, continuation id, and nonce.

2.4 Web Cells

```
(require web-server/servlet/web-cells)
```

The `web-server/servlet/web-cells` library provides the interface to Web cells.

A Web cell is a kind of state defined relative to the *frame tree*. The frame-tree is a mirror of the user's browsing session. Every time a continuation is invoked, a new frame (called the *current frame*) is created as a child of the current frame when the continuation was captured.

You should use Web cells if you want an effect to be encapsulated in all interactions linked from (in a transitive sense) the HTTP response being generated. For more information on their semantics, consult the paper "Interaction-Safe State for the Web".

```
(web-cell? v) → boolean?
  v : any/c
```

Determines if `v` is a web-cell.

```
(make-web-cell v) → web-cell?
  v : any/c
```

Creates a web-cell with a default value of `v`.

```
(web-cell-ref wc) → any/c
  wc : web-cell?
```

Looks up the value of `wc` found in the nearest frame.

```
(web-cell-shadow wc v) → void
  wc : web-cell?
  v : any/c
```

Binds `wc` to `v` in the current frame, shadowing any other bindings to `wc` in the current frame.

Below is an extended example that demonstrates how Web cells allow the creation of reusable Web abstractions without requiring global transformations of the program into continuation or store passing style.

```
#lang web-server/insta

(define (start initial-request)
 (define counter1 (make-counter))
 (define counter2 (make-counter))
 (define include1 (include-counter counter1))
 (define include2 (include-counter counter2))
 (send/suspend/dispatch
  (lambda (embed/url)
    `(html
      (body (h2 "Double Counters")
            (div (h3 "First")
                 ,(include1 embed/url))
            (div (h3 "Second")
                 ,(include2 embed/url)))))))

(define (make-counter)
 (make-web-cell 0))

(define (include-counter a-counter)
 (let/cc k
   (let loop ()
     (k
      (lambda (embed/url)
        `(div (h3 ,(number->string (web-cell-ref a-counter)))
              (a ([href
                   ,(embed/url
                     (lambda _
                       ; A new frame has been created
                       (define last (web-cell-ref a-counter))
                       ; We can inspect the value at the parent
                       (web-cell-shadow a-counter (add1 last))
                       ; The new frame has been modified
                       (loop)))])
                 "+")))))))
```

2.5 Continuation Managers

Since Scheme servlets store their continuations on the server, they take up memory on the server. Furthermore, garbage collection can not be used to free this memory, because there are roots outside the system: users' browsers, bookmarks, brains, and notebooks. Therefore, some other strategy must be used if memory usage is to be controlled. This functionality is pluggable through the manager interface.

2.5.1 General

```
(require web-server/managers/manager)
```

This module defines the manager interface. It is required by the users and implementers of managers.

```
(struct manager (create-instance
                 adjust-timeout!
                 clear-continuations!
                 continuation-store!
                 continuation-lookup))
  create-instance : ((-> void) . -> . number?)
  adjust-timeout! : (number? number? . -> . void)
  clear-continuations! : (number? . -> . void)
  continuation-store! : (number? any/c expiration-handler/c . -> . (list/c number? number?))
  continuation-lookup : (number? number? number? . -> . any/c)
```

`create-instance` is called to initialize a instance, to hold the continuations of one servlet session. It is passed a function to call when the instance is expired. It runs the id of the instance.

`adjust-timeout!` is a to-be-deprecated function that takes an instance-id and a number. It is specific to the timeout-based manager and will be removed.

`clear-continuations!` expires all the continuations of an instance.

`continuation-store!` is given an instance-id, a continuation value, and a function to include in the exception thrown if the continuation is looked up and has been expired. The two numbers returned are a continuation-id and a nonce.

`continuation-lookup` finds the continuation value associated with the instance-id, continuation-id, and nonce triple it is given.

```
(struct (exn:fail:servlet-manager:no-instance exn:fail) (expiration-handler
  expiration-handler : expiration-handler/c
```

This exception should be thrown by a manager when an instance is looked up that does not exist.

```
(struct (exn:fail:servlet-manager:no-continuation exn:fail) (expiration-handler
  expiration-handler : expiration-handler/c
```

This exception should be thrown by a manager when a continuation is looked up that does

not exist.

2.5.2 No Continuations

```
(require web-server/managers/none)
```

This module defines a manager constructor:

```
(create-none-manager instance-expiration-handler) → manager?
  instance-expiration-handler : expiration-handler/c
```

This manager does not actually store any continuation or instance data. You could use it if you know your servlet does not use the continuation capturing functions and want the server to not allocate meta-data structures for each instance.

If you *do* use a continuation capturing function, the continuation is simply not stored. If the URL is visited, the *instance-expiration-handler* is called with the request.

If you are considering using this manager, also consider using the Web Language. (See §3 “Stateless Servlets”.)

2.5.3 Timeouts

```
(require web-server/managers/timeouts)
```

This module defines a manager constructor:

```
(create-timeout-manager instance-exp-handler
                        instance-timeout
                        continuation-timeout) → manager?
  instance-exp-handler : expiration-handler/c
  instance-timeout : number?
  continuation-timeout : number?
```

Instances managed by this manager will be expired *instance-timeout* seconds after the last time it is accessed. If an expired instance is looked up, the exn:fail:servlet-manager:no-instance exception is thrown with *instance-exp-handler* as the expiration handler.

Continuations managed by this manager will be expired *continuation-timeout* seconds after the last time it is accessed. If an expired continuation is looked up, the exn:fail:servlet-manager:no-continuation exception is thrown with *instance-*

exp-handler as the expiration handler, if no expiration-handler was passed to `continuation-store!`.

`adjust-timeout!` corresponds to `reset-timer!` on the timer responsible for the servlet instance.

This manager has been found to be... problematic... in large-scale deployments of the Web Server .

2.5.4 LRU

```
(require web-server/managers/lru)
```

This module defines a manager constructor:

```
(create-LRU-manager  instance-expiration-handler
                     check-interval
                     collect-interval
                     collect?
                    [#:initial-count initial-count
                     #:inform-p inform-p])          → manager?
  instance-expiration-handler : expiration-handler/c
  check-interval : integer?
  collect-interval : integer?
  collect? : (-> boolean?)
  initial-count : integer? = 1
  inform-p : (integer? . -> . void) = (lambda _ (void))
```

Instances managed by this manager will be expired if there are no continuations associated with them, after the instance is unlocked. If an expired instance is looked up, the `exn:fail:servlet-manager:no-instance` exception is thrown with `instance-exp-handler` as the expiration handler.

Continuations managed by this manager are given a "Life Count" of *initial-count* initially. If an expired continuation is looked up, the `exn:fail:servlet-manager:no-continuation` exception is thrown with `instance-exp-handler` as the expiration handler, if no expiration-handler was passed to `continuation-store!`.

Every *check-interval* seconds *collect?* is called to determine if the collection routine should be run. Every *collect-interval* seconds the collection routine is run.

Every time the collection routine runs, the "Life Count" of every continuation is decremented by `1`. If a continuation's count reaches `0`, it is expired. The *inform-p* function is called if any continuations are expired, with the number of continuations expired.

The recommended usage of this manager is codified as the following function:

```
(make-threshold-LRU-manager instance-expiration-handler
                            memory-threshold)
 → manager?
  instance-expiration-handler : expiration-handler/c
  memory-threshold : number?
```

This creates an LRU manager with the following behavior: The memory limit is set to *memory-threshold* bytes. Continuations start with 24 life points. Life points are deducted at the rate of one every 10 minutes, or one every 5 seconds when the memory limit is exceeded. Hence the maximum life time for a continuation is 4 hours, and the minimum is 2 minutes.

If the load on the server spikes—as indicated by memory usage—the server will quickly expire continuations, until the memory is back under control. If the load stays low, it will still efficiently expire old continuations.

3 Stateless Servlets

```
#lang web-server
```

3.1 Example

A stateless servlet should `provide` the following exports:

`interface-version : (one-of/c 'stateless)`

This indicates that the servlet is a stateless servlet.

`stuffer : (stuffer/c serializable? bytes?)`

This is the stuffer that will be used for the servlet.

If it is not provided, it defaults to `default-stuffer`.

`manager : manager?`

This is the manager that will be used for the servlet.

If it is not provided, it defaults to `(create-none-manager #f)`.

`(start initial-request) → response/c`
`  initial-request : request?`

This function is called when an instance of this servlet is started. The argument is the HTTP request that initiated the instance.

An example `'stateless` servlet module:

```
#lang web-server
(provide interface-version stuffer start)
(define interface-version 'stateless)
(define stuffer
 (stuffer-chain
  serialize-stuffer
  (md5-stuffer (build-path (find-system-path 'home-dir) ".urls"))))
(define (start req)
  `(html (body (h2 "Look ma, no state!"))))
```

These servlets have an extensive API available to them: net/url, web-server/http, web-server/http/bindings, web-server/lang/abort-resume, web-server/lang/web, web-server/lang/native, web-server/lang/web-param, web-server/lang/web-cells, web-server/lang/file-box, web-server/lang/soft, web-server/dispatch, and web-server/stuffers. Some of these are documented in the subsections that follow.

3.2 Serializable Continuations

```
(require web-server/lang/abort-resume)
```

The main purpose of the stateless language is to provide serializable continuations to your servlet.

(call-with-serializable-current-continuation *response-generator*)
→ any
response-generator : (continuation? . -> . any)

Captures the current continuation in a serializable way and calls *response-generator* with it, returning the result.

This potentially uses resources of the current servlet's manager if serial->native and native->serial were used to capture an untransformable context.

(serial->native *expr*)

serial->native informs the serializing runtime that *expr* is potentially a call to an untransformed context. This sets up the necessary information for native->serial to signal to call-with-serializable-current-continuation to capture the native (and thus unserializable) section of the context and store it on the server.

(native->serial *expr*)

native->serial informs the serializing runtime that *expr* marks first expression after returning from an untransformed context. This captures the untransformed context such that call-with-serializable-current-continuation can store it on the server and reference it from serializable continuations.

For example,

```
(build-list
 3
 (lambda (i)
   (call-with-serializable-current-continuation
    (lambda (k) (serialize k)))))
```

will fail at runtime because `build-list` is not transformed. However,

```
(serial->native
 (build-list
  3
  (lambda (i)
    (native->serial
     (call-with-serializable-current-continuation
      (lambda (k) (serialize k)))))))
```

will succeed and `k` will reference a cell in the current servlet's `manager` that stores the part of the continuation in `build-list`.

3.3 Native Interfaces

```
(require web-server/lang/native)
```

It is sometimes inconvenient to use `serial->native` and `native->serial` throughout your program. This module provides a macro for creating wrappers.

`(define-native (`*native arg-spec ...*`) `*original*`)`

arg-spec : ho
arg-spec : _

Builds an interface around *original* named *native* such that calls to *native* are wrapped in `serial->native` and all arguments marked with `ho` in *arg-spec* are assumed to procedures and are wrapped in `native->serial`.

For example,

```
(define-native (build-list/native _ ho) build-list)
```

is equivalent to

```
(define (build-list/native fst snd)
  (serial->native
   (build-list
    fst
    (lambda args
      (native->serial
       (apply snd args))))))
```

3.4 Stateless Web Interaction

```
(require web-server/lang/web)
```

(send/suspend/url *response-generator*) → request?
  response-generator : (url? . -> . response/c)

Captures the current continuation. Serializes it and stuffs it into a URL. Calls *response-generator* with this URL and delivers the response to the client. If the URL is invoked the request is returned to this continuation.

(send/suspend *response-generator*) → request?
  response-generator : (string? . -> . response/c)

Like send/suspend/url but with a string URL representation.

(send/suspend/hidden *response-generator*) → request?
  response-generator : (url? xexpr/c . -> . response/c)

Captures the current continuation. Serializes it and stuffs it into a hidden INPUT form element. Calls *response-generator* with this URL and form field and delivers the response to the client. If the URL is invoked with form data containing the hidden form, the request is returned to this continuation.

(send/suspend/url/dispatch *make-response*) → any
  make-response : (((request? . -> . any) . -> . url?) . -> . response/c)

Calls *make-response* with a function that, when called with a procedure from request? to any/c will generate a URL, that when invoked will call the function with the request? object and return the result to the caller of send/suspend/dispatch.

(send/suspend/dispatch *make-response*) → request?
  make-response : (((request? . -> . any) . -> . string?) . -> . response/c)

Like send/suspend/url/dispatch but with a string URL representation.

(redirect/get) → request?

See web-server/servlet/web.

3.5 Stateless Web Cells

```
(require web-server/lang/web-cells)
```

The web-server/lang/web-cells library provides the same API as web-server/servlet/web-cells, but in a way compatible with the Web Language. The one difference is that make-web-cell is syntax, rather than a function.

```
(web-cell? v) → boolean?
  v : any/c
(make-web-cell default-expr)
(web-cell-ref wc) → any/c
  wc : web-cell?
(web-cell-shadow wc v) → void
  wc : web-cell?
  v : any/c
```

See web-server/servlet/web-cells.

3.6 File Boxes

```
(require web-server/lang/file-box)
```

As mentioned earlier, it is dangerous to rely on the store in Web Language servlets, due to the deployment scenarios available to them. This module provides a simple API to replace boxes in a safe way.

```
(file-box? v) → boolean?
  v : any/c
```

Checks if v is a file-box.

```
(file-box p v) → file-box?
  p : path-string?
  v : serializable?
```

Creates a file-box that is stored at *p*, with the default contents of *v*.

```
(file-unbox fb) → serializable?
  fb : file-box?
```

Returns the value inside *fb*

```
(file-box-set? fb) → boolean?
  fb : file-box?
```

Returns #t if *fb* contains a value.

```
(file-box-set! fb v) → void
  fb : file-box?
  v : serializable?
```

Saves *v* in the file represented by *fb*.

Warning: If you plan on using a load-balancer, make sure your file-boxes are on a shared medium.

3.7 Stateless Web Parameters

```
(require web-server/lang/web-param)
```

It is not easy to use `parameterize` in the Web Language. This module provides (roughly) the same functionality in a way that is serializable. Like other serializable things in the Web Language, they are sensitive to source code modification.

```
(make-web-parameter default)
```

Expands to the definition of a web-parameter with *default* as the default value. A web-parameter is a procedure that, when called with zero arguments, returns *default* or the last value `web-parameterized` in the dynamic context of the call.

```
(web-parameter? v) → boolean?
  v : any/c
```

Checks if *v* appears to be a web-parameter.

```
(web-parameterize ([web-parameter-expr value-expr] ...) expr ...)
```

Runs (begin *expr* ...) such that the web-parameters that the *web-parameter-expr*s evaluate to are bound to the *value-expr*s. From the perspective of the *value-expr*s, this is like let.

3.8 Soft State

```
(require web-server/lang/soft)
```

Sometimes you want to reference a large data-structure from a stateless program without the data-structure being serialized and increasing the size of the serialization. This module provides support for this scenario.

```
(soft-state? v) → boolean?
  v : any/c
```

Determines if *v* is a soft state record.

```
(make-soft-state thnk) → soft-state?
  thnk : (-> any/c)
```

Creates a piece of soft state that is computed by *thnk*. This value is serializable.

```
(soft-state-ref ss) → any/c
  ss : soft-state?
```

Extracts the value associated with *ss*. If the value is not available (perhaps because of garbage collection, deserialization in an uninitialized process, etc), then the thunk associated with *ss* is invoked and the value is cached.

```
(soft-state expr ...)
```

Equivalent to (make-soft-state (lambda () *expr* ...)).

Here's an example servlet that uses soft state:

```
#lang web-server

(provide interface-version start)
(define interface-version 'stateless)

(define softie
  (soft-state
   (printf "Doing a long computation...~n")
   (sleep 1)))

(define (start req)
  (soft-state-ref softie)
  (printf "Done~n")
  (start
   (send/suspend
    (lambda (k-url)
      `(html (body (a ([href ,k-url]) "Done")))))))
```

When this is run and the link is clicked a few times, the output is:

```
$ plt-web-server -p 8080
Doing a long computation...
Done
Done
Done
Done
```

If the server is restarted or the hostname in the URL is changed to a different host with the same code, and the URL is clicked:

```
^Cuser break
$ plt-web-server -p 8080
Doing a long computation...
Done
```

3.9 Stuffers

```
(require web-server/stuffers)
```

The `web-server` language provides serializable continuations. The serialization functionality is abstracted into *stuffers* that control how it operates. You can supply your own (built with these functions) when you write a stateless servlet.

3.9.1 Basic Combinators

```
(require web-server/stuffers/stuffer)
```

```
(struct stuffer (in out))
  in : (any/c . -> . any/c)
  out : (any/c . -> . any/c)
```

A stuffer is essentially an invertible function captured in this structure. The following should hold:

```
(out (in x)) = x
(in (out x)) = x
```

```
(stuffer/c dom rng) → contract?
  dom : any/c
  rng : any/c
```

Constructs a contract for a stuffer where `in` has the contract `(-> dom rng)` and `out` has the contract `(-> rng dom)`.

```
id-stuffer : (stuffer/c any/c any/c)
```

The identitiy stuffer.

```
(stuffer-compose g f) → (stuffer/c any/c any/c)
  g : (stuffer/c any/c any/c)
  f : (stuffer/c any/c any/c)
```

Composes *f* and *g*, i.e., applies *f* then *g* for `in` and *g* then *f* for `out`.

```
(stuffer-sequence f g) → (stuffer/c any/c any/c)
  f : (stuffer/c any/c any/c)
  g : (stuffer/c any/c any/c)
```

`stuffer-compose` with arguments swapped.

```
(stuffer-if c f) → (stuffer/c bytes? bytes?)
  c : (bytes? . -> . boolean?)
  f : (stuffer/c bytes? bytes?)
```

Creates a stuffer that stuffs with `f` if `c` is true on the input to `in`. Similarly, applies `f` during `out` if it was applied during `in` (which is recorded by prepending a byte.)

```
(stuffer-chain x ...) → stuffer?
  x : (or/c stuffer? (bytes? . -> . boolean?))
```

Applies `stuffer-sequence` and `stuffer-if` to successive tails of `x`.

3.9.2 Serialization

```
(require web-server/stuffers/serialize)
```

```
serialize-stuffer : (stuffer/c serializable? bytes?)
```

A stuffer that uses `serialize` and `write/bytes` and `deserialize` and `read/bytes`.

3.9.3 Base64 Encoding

```
(require web-server/stuffers/base64)
```

```
base64-stuffer : (stuffer/c bytes? bytes?)
```

A stuffer that uses `base64-encode` and `base64-decode`.

Useful for getting URL-safe bytes.

3.9.4 GZip Compression

```
(require web-server/stuffers/gzip)
```

```
gzip-stuffer : (stuffer/c bytes? bytes?)
```

A stuffer that uses `gzip/bytes` and `gunzip/bytes`.

Warning: You should compose this with `base64-stuffer` to get URL-safe bytes.

3.9.5 Key/Value Storage

The web-server/stuffers/hash stuffers rely on a key/value store.

```
(require web-server/stuffers/store)
```

```
(struct store (write read))
  write : (bytes? bytes? . -> . void)
  read : (bytes? . -> . bytes?)
```

The following should hold:

```
  (begin (write k v) (read k)) = v
```

```
(dir-store root) → store?
  root : path-string?
```

A store that stores key key's value in a file located at

```
  (build-path
   root
   (bytes->string/utf-8 key))
```

It should be easy to use this interface to create store for databases like SQLite, CouchDB, or BerkeleyDB.

3.9.6 Hash-addressed Storage

```
(require web-server/stuffers/hash)
```

```
hash/c : contract?
```

Equivalent to (-> bytes? bytes?).

```
(hash-stuffer H store) → (stuffer/c bytes? bytes?)
  H : hash/c
  store : store?
```

A content-addressed storage stuffer that stores input bytes, input, in store with the key (H input) and returns the key. Similarly, on out the original bytes are looked up.

```
(md5-stuffer root) → (stuffer/c bytes? bytes?)
  root : path-string?
```

Equivalent to (hash-stuffer md5 (dir-store root))

3.9.7 HMAC-SHA1 Signing

```
(require web-server/stuffers/hmac-sha1)
```

```
(HMAC-SHA1 kb db) → bytes?
  kb : bytes?
  db : bytes?
```

Performs a HMAC-SHA1 calculation on *db* using *kb* as the key. The result is guaranteed to be 20 bytes. (You could curry this to use it with hash-stuffer, but there is little value in doing so over md5.)

```
(HMAC-SHA1-stuffer kb) → (stuffer/c bytes? bytes?)
  kb : bytes?
```

A stuffer that signs input using HMAC-SHA1 with *kb* as the key. The result of the stuffer is the hash prepended to the input data. When the stuffer is run in reverse, it checks if the first 20 bytes are the correct has for the rest of the data.

Warning: You should compose this with base64-stuffer to get URL-safe bytes.

Warning: Without explicit provision, it is possible for users to modify the continuations they are sent through the other stuffers. This stuffer allows the servlet to certify that stuffed data was truly generated by the servlet. Therefore, you **should** use this if you are not using the hash-stuffers.

Warning: This stuffer does **not** encrypt the data in anyway, so users can still observe the stuffed values.

3.9.8 Helpers

```
(require web-server/lang/stuff-url)
```

```
(is-url-too-big? v) → boolean?
  v : bytes?
```

Determines if stuffing *v* into the current servlet's URL would result in a URL that is too big for Internet Explorer. (IE only supports URLs up to 2048 characters.)

```
(make-default-stuffer root) → (stuffer/c serializable? bytes?)
  root : path-string?
```

Constructs a stuffer that serializes, then if the URL is too big, compresses (and base64-encodes), if the URL is still too big then it stores it in an MD5-indexed database rooted at *root*.

Equivalent to:

```
(stuffer-chain
 serialize-stuffer
 is-url-too-big?
 (stuffer-chain
  gzip-stuffer
  base64-stuffer)
 is-url-too-big?
 (md5-stuffer root))
```

```
default-stuffer : (stuffer/c serializable? bytes?)
```

Equivalent to:

```
(make-default-stuffer
 (build-path
  (find-system-path 'home-dir)
  ".urls"))
```

3.10 Usage Considerations

A stateless servlet has the following process performed on it automatically:

- All uses of `letrec` are removed and replaced with equivalent uses of `let` and imperative features.
- The program is converted into ANF (Administrative Normal Form), making all continuations explicit.
- All continuations and continuations marks are recorded in the continuation marks of the expression they are the continuation of.
- All calls to external modules are identified and marked.

- All uses of `call/cc` are removed and replaced with equivalent gathering of the continuations through the continuation marks installed earlier.
- The program is defunctionalized with a serializable data-structure for each `lambda`.

This process allows the continuations captured by your servlet to be serialized. This means they may be stored on the client's browser or the server's disk. Thus, your servlet has no cost to the server other than execution. This is very attractive if you've used Scheme servlets and had memory problems.

This process is defined on all of PLT Scheme and occurs after macro-expansion, so you are free to use all interesting features of PLT Scheme. However, there are some considerations you must make.

First, this process drastically changes the structure of your program. It will create an immense number of lambdas and structures your program did not normally contain. The performance implication of this has not been studied with PLT Scheme.

Second, the defunctionalization process is sensitive to the syntactic structure of your program. Therefore, if you change your program in a trivial way, for example, changing a constant, then all serialized continuations will be obsolete and will error when deserialization is attempted. This is a feature, not a bug! It is a small price to pay for protection from the sorts of errors that would occur if your program were changed in a meaningful way.

Third, the values in the lexical scope of your continuations must be serializable for the continuations itself to be serializable. This means that you must use `define-serializable-struct` rather than `define-struct`, and take care to use modules that do the same. Similarly, you may not use `parameterize`, because parameterizations are not serializable.

Fourth, and related, this process only runs on your code, not on the code you `require`. Thus, your continuations—to be serializable—must not be in the context of another module. For example, the following will not work:

```
(define requests
  (map (lambda (rg) (send/suspend/url rg))
       response-generators))
```

because `map` is not transformed by the process. However, if you defined your own `map` function, there would be no problem. Another solution is to store the `map` part of the continuation on the server with `serial->native` and `native->serial`:

```
(define requests
  (serial->native
   (map (lambda (rg) (native->serial (send/suspend/url rg)))
       response-generators)))
```

Fifth, the store is **not** serialized. If you rely on the store you will be taking huge risks. You

will be assuming that the serialized continuation is invoked on the same server before the server is restarted or the memory is garbage collected.

This process is derived from the ICFP papers *Continuations from Generalized Stack Inspection* by Pettyjohn et al. in 2005 and *Automatically RESTful Web Applications, Or Marking Modular Serializable Continuations* by Jay McCarthy in 2009. We thank Greg Pettyjohn for his initial implementation of this algorithm.

4 HTTP: Hypertext Transfer Protocol

```
(require web-server/http)
```

The Web Server implements many HTTP RFCs that are provided by this module.

4.1 Requests

```
(require web-server/http/request-structs)
```

```
(struct header (field value))
  field : bytes?
  value : bytes?
```

Represents a header of `field` to `value`.

```
(headers-assq id heads) → (or/c false/c header?)
  id : bytes?
  heads : (listof header?)
```

Returns the header with a field equal to *id* from *heads* or `#f`.

```
(headers-assq* id heads) → (or/c false/c header?)
  id : bytes?
  heads : (listof header?)
```

Returns the header with a field case-insensitively equal to *id* from *heads* or `#f`.

You almost **always** want to use this, rather than `headers-assq` because Web browsers may send headers with arbitrary casing.

```
(struct binding (id))
  id : bytes?
```

Represents a binding of `id`.

```
(struct (binding:form binding) (value))
  value : bytes?
```

Represents a form binding of `id` to `value`.

```
(struct (binding:file binding) (filename headers content))
  filename : bytes?
  headers : (listof header?)
  content : bytes?
```

Represents the uploading of the file `filename` with the id `id` and the content `content`, where `headers` are the additional headers from the MIME envelope the file was in. (For example, the `#"Content-Type"` header may be included by some browsers.)

```
(bindings-assq id binds) → (or/c false/c binding?)
  id : bytes?
  binds : (listof binding?)
```

Returns the binding with an id equal to *id* from *binds* or `#f`.

```
(struct request (method
                 uri
                 headers/raw
                 bindings/raw
                 post-data/raw
                 host-ip
                 host-port
                 client-ip))
  method : bytes?
  uri : url?
  headers/raw : (listof header?)
  bindings/raw : (listof binding?)
  post-data/raw : (or/c false/c bytes?)
  host-ip : string?
  host-port : number?
  client-ip : string?
```

An HTTP `method` request to `uri` from `client-ip` to the server at `host-ip`:`host-port` with `headers/raw` headers, `bindings/raw` GET and POST queries and `post-data/raw` POST data.

You are **unlikely to need to construct** a request struct.

Here is an example typical of what you will find in many applications:

```
(define (get-number req)
  (match
    (bindings-assq
      #"number"
      (request-bindings/raw req))
    [(? binding:form? b)
     (string->number
      (bytes->string/utf-8
       (binding:form-value b)))]
    [_
     (get-number (request-number))]))
```

4.2 Bindings

```
(require web-server/http/bindings)
```

These functions, while convenient, could introduce subtle bugs into your application. Examples: that they are case-insensitive could introduce a bug; if the data submitted is not in UTF-8 format, then the conversion to a string will fail; if an attacker submits a form field as if it were a file, when it is not, then the `request-bindings` will hold a `bytes?` object and your program will error; and, for file uploads you lose the filename. **Therefore, we recommend against their use, but they are provided for compatibility with old code.**

```
(request-bindings req)
 → (listof (or/c (cons/c symbol? string?)
                 (cons/c symbol? bytes?)))
  req : request?
```

Translates the `request-bindings/raw` of *req* by interpreting `bytes?` as `string?`s, except in the case of `binding:file` bindings, which are left as is. Ids are then translated into lowercase symbols.

```
(request-headers req) → (listof (cons/c symbol? string?))
  req : request?
```

Translates the `request-headers/raw` of *req* by interpreting `bytes?` as `string?`s. Ids are then translated into lowercase symbols.

```
(extract-binding/single id binds) → string?
  id : symbol?
  binds : (listof (cons/c symbol? string?))
```

Returns the single binding associated with *id* in the a-list *binds* if there is exactly one binding. Otherwise raises exn:fail.

```
(extract-bindings id binds) → (listof string?)
  id : symbol?
  binds : (listof (cons/c symbol? string?))
```

Returns a list of all the bindings of *id* in the a-list *binds*.

```
(exists-binding? id binds) → boolean?
  id : symbol?
  binds : (listof (cons/c symbol? string))
```

Returns #t if *binds* contains a binding for *id*. Otherwise, #f.

Here is an example typical of what you will find in many applications:

```
(define (get-number req)
  (string->number
   (extract-binding/single
    'number
    (request-bindings req))))
```

4.3 Responses

```
(require web-server/http/response-structs)
```

```
(struct response/basic (code message seconds mime headers))
  code : number?
  message : bytes?
  seconds : number?
  mime : bytes?
  headers : (listof header?)
```

A basic HTTP response containing no body. code is the response code, message the message, seconds the generation time, mime the MIME type of the file, and extras are the extra headers, in addition to those produced by the server.

Example:

```
(make-response/basic
 301 #"Moved Permanently"
 (current-seconds) TEXT/HTML-MIME-TYPE
 (list (make-header #"Location"
                    #"http://www.plt-scheme.org/downloads")))
```

```
(struct (response/full response/basic) (body))
  body : (listof bytes?)
```

As with `response/basic`, except with `body` as the response body.

Example:

```
(make-response/full
 301 #"Moved Permanently"
 (current-seconds) TEXT/HTML-MIME-TYPE
 (list (make-header #"Location"
                    #"http://www.plt-scheme.org/downloads"))
 (list #"<html><body><p>"
       #"Please go to <a href=\""
       #"http://www.plt-scheme.org/downloads"
       #"\">here</a> instead."
       #"</p></body></html>"))
```

```
(struct (response/incremental response/basic) (generator))
  generator : ((() () #:rest (listof bytes?) . ->* . any) . -> . any)
```

As with `response/basic`, except with `generator` as a function that is called to generate the response body, by being given an `output-response` function that outputs the content it is called with. If the `output-response` function is called with arguments of zero length (when concatenated), then the output port is flushed with `flush-output`.

Here is a short example:

```
(make-response/incremental
  200 #"OK" (current-seconds)
  #"application/octet-stream"
  (list (make-header #"Content-Disposition"
                     #"attachment; filename=\"file\""))
  (lambda (output-response)
    (output-response #"Some content")
    (output-response)
    (output-response #"Even" #"more" #"content!")
    (output-response #"Now we're done")))
```

```
response/c : contract?
```

Equivalent to

```
(or/c response/basic?
      (cons/c bytes? (listof (or/c string? bytes?)))
      xexpr/c)
```

```
(make-xexpr-response  xexpr
                     [#:code code
                      #:message message
                      #:seconds seconds
                      #:mime-type mime-type
                      #:headers headers])    → response/full?
  xexpr : xexpr/c
  code : number? = 200
  message : bytes? = #"Okay"
  seconds : number? = (current-seconds)
  mime-type : bytes? = TEXT/HTML-MIME-TYPE
  headers : (listof header?) = empty
```

Equivalent to

```
(make-response/full
 code message seconds mime-type headers
 (list (string->bytes/utf-8 (xexpr->string xexpr))))
```

```
(normalize-response close? response)
 → (or/c response/full? response/incremental?)
  close? : boolean?
  response : response/c
```

Coerces *response* into a full response, filling in additional details where appropriate.

```
TEXT/HTML-MIME-TYPE : bytes?
```

Equivalent to `#"text/html; charset=utf-8"`.

Warning: If you include a Content-Length header in a response that is inaccurate, there **will be an error** in transmission that the server **will not catch**.

4.4 Placing Cookies

```
(require web-server/http/cookie)
```

This module provides functions to create cookies and responses that set them.

```
(make-cookie  name
              value
             [#:comment comment
              #:domain domain
              #:max-age max-age
              #:path path
              #:secure? secure?]) → cookie?
  name : string?
  value : string?
  comment : (or/c false/c string?) = #f
  domain : (or/c false/c valid-domain?) = #f
  max-age : (or/c false/c exact-nonnegative-integer?) = #f
  path : (or/c false/c string?) = #f
  secure? : (or/c false/c boolean?) = #f
```

Constructs a cookie with the appropriate fields.

```
(cookie->header c) → header?
  c : cookie?
```

Constructs a header that sets the cookie.

```
(xexpr-response/cookies cookies xexpr) → response/full?
  cookies : (listof cookie?)
  xexpr : xexpr/c
```

Constructs a response using *xexpr* that sets all the cookies in *cookies*.

Examples:

```
(define time-cookie
  (make-cookie "time" (number->string (current-seconds))))
(define id-cookie
  (make-cookie "id" "joseph" #:secure? #t))

(redirect-to
 "http://localhost/logged-in"
 see-other
 #:headers
 (map cookie->header
      (list time-cookie id-cookie)))

(send/suspend
  (lambda (k-url)
    (xexpr-response/cookies
     (list time-cookie id-cookie)
     `(html (head (title "Cookie Example"))
            (body (h1 "You're cookie'd!"))))))
```

Warning: When using cookies, make sure you follow the advice of the MIT Cookie Eaters, or you will be susceptible to dangerous attacks.

4.5 Extracting Cookies

```
(require web-server/http/cookie-parse)
```

```
(struct client-cookie (name value domain path))
  name : string?
  value : string?
  domain : (or/c false/c valid-domain?)
  path : (or/c false/c string?)
```

While server cookies are represented with `cookie?`s, cookies that come from the client are represented with a `client-cookie` structure.

```
(request-cookies req) → (listof client-cookie?)
  req : request?
```

Extracts the cookies from *req*'s headers.

Examples:

```
(define (start req)
  (define cookies (request-cookies req))
  (define id-cookie
    (findf (lambda (c)
             (string=? "id" (client-cookie-name c)))
           cookies))
  (if id-cookie
      (hello (client-cookie-value id-cookie))
      (redirect-to
       (url->string (request-uri req))
       see-other
       #:headers
       (list
        (cookie->header (make-cookie "id" "joseph"))))))

 (define (hello who)
   `(html (head (title "Hello!"))
          (body
           (h1 "Hello "
               ,who))))
```

4.6 Redirect

```
(require web-server/http/redirect)
```

```
(redirect-to  uri
              [perm/temp
               #:headers headers]) → response/c
  uri : non-empty-string/c
  perm/temp : redirection-status? = temporarily
  headers : (listof header?) = (list)
```

Generates an HTTP response that redirects the browser to *uri*, while including the *headers* in the response.

Example: `(redirect-to "http://www.add-three-numbers.com" permanently)`

```
(redirection-status? v) → boolean?
  v : any/c
```

Determines if *v* is one of the following values.

`permanently` : `redirection-status?`

A `redirection-status?` for permanent redirections.

`temporarily` : `redirection-status?`

A `redirection-status?` for temporary redirections.

`see-other` : `redirection-status?`

A `redirection-status?` for ”see-other” redirections.

4.7 Basic Authentication

```
(require web-server/http/basic-auth)
```

An implementation of HTTP Basic Authentication.

`(make-basic-auth-header` *realm*`)` → `header?`
  realm : `string?`

Returns a header that instructs the Web browser to request a username and password from the client using Basic authentication with *realm* as the realm.

`(request->basic-credentials` *req*`)`
 → `(or/c false/c (cons/c bytes? bytes?))`
  req : `request?`

Returns a pair of the username and password from the authentication header in *req* if they are present, or `#f`.

Example:

```
#lang web-server/insta

(define (start req)
  (match (request->basic-credentials req)
    [(cons user pass)
     `(html (head (title "Basic Auth Test"))
            (body (h1 "User: " ,(bytes->string/utf-8 user))
                  (h1 "Pass: " ,(bytes->string/utf-8 pass))))]
    [else
     (make-response/basic
      401 #"Unauthorized" (current-seconds) TEXT/HTML-MIME-TYPE
      (list
       (make-basic-auth-header
        (format "Basic Auth Test: ~a" (gensym)))))]))
```

4.8 Digest Authentication

```
(require web-server/http/digest-auth)
```

An implementation of HTTP Digest Authentication.

```
(make-digest-auth-header realm
                         private-key
                         opaque)      → header?
  realm : string?
  private-key : string?
  opaque : string?
```

Returns a header that instructs the Web browser to request a username and password from the client using Digest authentication with *realm* as the realm, *private-key* as the server's contribution to the nonce, and *opaque* as the opaque data passed through the client.

```
(request->digest-credentials req)
 → (or/c false/c (listof (cons/c symbol? string?)))
  req : request?
```

Returns the Digest credentials from *req* (if they appear) as an association list.

```
username*realm->password/c : contract?
```

Used to look up the password for a user is a realm.

Equivalent to `(-> string? string? string?)`.

```
username*realm->digest-HA1/c : contract?
```

Used to compute the user's secret hash.

Equivalent to `(-> string? string? bytes?)`.

```
(password->digest-HA1 lookup-password)
 → username*realm->digest-HA1/c
  lookup-password : username*realm->password/c
```

Uses *lookup-password* to find the password, then computes the secret hash of it.

```
(make-check-digest-credentials lookup-HA1)
 → (string? (listof (cons/c symbol? string?)) . -> . boolean?)
  lookup-HA1 : username*realm->digest-HA1/c
```

Constructs a function that checks whether particular Digest credentials (the second argument of the returned function) are correct given the HTTP method provided as the first argument and the secret hash computed by *lookup-HA1*.

This is will result in an exception if the Digest credentials are missing portions.

Example:

```
#lang web-server/insta
(require scheme/pretty)

(define private-key "private-key")
(define opaque "opaque")

(define (start req)
  (match (request->digest-credentials req)
    [#f
     (make-response/basic
      401 #"Unauthorized" (current-seconds) TEXT/HTML-MIME-TYPE
      (list (make-digest-auth-header
             (format "Digest Auth Test: ~a" (gensym))
             private-key opaque)))]
    [alist
     (define check
       (make-check-digest-credentials
        (password->digest-HA1 (lambda (username realm) "pass"))))
     (define pass?
       (check "GET" alist))
     `(html (head (title "Digest Auth Test"))
            (body
             (h1 ,(if pass? "Pass!" "No Pass!"))
             (pre ,(pretty-format alist))))]))
```

5 URL-Based Dispatch

```
(require web-server/dispatch)
```

The library allows the creation of two-way mappings between permanent URLs and request-handling procedures.

This library was inspired by the (planet untyped/dispatch) package.

5.1 Using web-server/dispatch

Suppose you are writing a blog application and want pretty URLs for different views of the site. You would define some URL dispatching rules as follows:

```
> (define-values (blog-dispatch blog-url)
    (dispatch-rules
     [("") list-posts]
     [("posts" (string-arg)) review-post]
     [("archive" (integer-arg) (integer-arg)) review-archive]
     [else list-posts]))
```

And define your request handlers as follows:

```
> (define (list-posts req) `(list-posts))
> (define (review-post req p) `(review-post ,p))
> (define (review-archive req y m) `(review-archive ,y ,m))
```

Now when a request is sent to your application, it will be directed to the appropriate handler:

```
> (define (url->request u)
    (make-request #"GET" (string->url u) empty
                  empty #f "1.2.3.4" 80 "4.3.2.1"))
> (blog-dispatch
   (url->request "http://www.chrlsnchrg.com"))
(list-posts)
> (blog-dispatch
   (url->request "http://www.chrlsnchrg.com/"))
(list-posts)
> (blog-dispatch
   (url->request
    "http://www.chrlsnchrg.com/posts/Extracurricular-Activity"))
(review-post "Extracurricular-Activity")
> (blog-dispatch
   (url->request "http://www.chrlsnchrg.com/archive/1984/10"))
(review-archive 1984 10)
> (blog-dispatch
   (url->request "http://www.chrlsnchrg.com/contact"))
(list-posts)
```

You can also generate these pretty URLs from procedure calls:

```
> (blog-url list-posts)
"/"
> (blog-url review-post "Another-Saturday-Night")
"/posts/Another-Saturday-Night"
> (blog-url review-archive 1984 11)
"/archive/1984/11"
```

After mastering the world of blogging software, you decide to put the ubiquitous Add-Two-Numbers.com out of business with Sum.com:

```
> (define-values (sum-dispatch sum-url)
    (dispatch-rules
     [((integer-arg) ...) sum]
     [else (lambda (req) (sum req empty))]))
> (define (sum req is)
    (apply + is))
> (sum-dispatch (url->request "http://www.sum.com/"))
0
> (sum-dispatch (url->request "http://www.sum.com/2"))
2
> (sum-dispatch (url->request "http://www.sum.com/2/3/4"))
9
> (sum-dispatch (url->request "http://www.sum.com/5/10/15/20"))
50
> (sum-url sum empty)
"/"
> (sum-url sum (list 1))
"/1"
> (sum-url sum (list 2 3 5 7))
"/2/3/5/7"
```

5.2 API Reference

```
(dispatch-rules
 [dispatch-pattern dispatch-fun]
 ...
 [else else-fun])
(dispatch-rules
 [dispatch-pattern dispatch-fun]
 ...)

  else-fun : (request? . -> . response/c)
  dispatch-fun : (request? any/c ... . -> . response/c)
```

Returns two values: the first is a dispatching function with the contract `(-> request? response/c)` that calls the appropriate *dispatch-fun* based on the first *dispatch-pattern* that matches the request's URL; the second is a URL-generating function with the contract `(-> procedure? any/c ... string?)` that generates a URL using *dispatch-pattern* for the *dispatch-fun* given as its first argument.

If *else-fun* is left out, one is provided that calls `(next-dispatcher)` to signal to the Web Server that this dispatcher does not apply.

```
  dispatch-pattern = ()
```

```
                  | (string . dispatch-pattern)
                  | (bidi-match-expander ... . dispatch-pattern)
                  | (bidi-match-expander . dispatch-pattern)
```

```
(dispatch-case
 [dispatch-pattern dispatch-fun]
 ...
 [else else-fun])
(dispatch-case
 [dispatch-pattern dispatch-fun]
 ...)

  else-fun : (request? . -> . response/c)
  dispatch-fun : (request? any/c ... . -> . response/c)
```

Returns a dispatching function as described by `dispatch-rules`.

```
(dispatch-url
 [dispatch-pattern dispatch-fun]
 ...)

  dispatch-fun : (request? any/c ... . -> . response/c)
```

Returns a URL-generating function as described by `dispatch-rules`.

```
(serve/dispatch dispatch) → void
  dispatch : (request? . -> . response/c)
```

Calls `serve/servlet` with appropriate arguments so that every request is handled by *dispatch*.

5.3 Built-in URL patterns

`web-server/dispatch` builds in a few useful URL component patterns.

```
(number-arg)
```

A bi-directional match expander that parses a `number?` from the URL and generates a URL with a number's encoding as a string.

```
(integer-arg)
```

A bi-directional match expander that parses a `integer?` from the URL and generates a URL with a integer's encoding as a string.

`(real-arg)`

A bi-directional match expander that parses a `real?` from the URL and generates a URL with a real's encoding as a string.

`(string-arg)`

A bi-directional match expander that parses a `string?` from the URL and generates a URL containing the string.

`(symbol-arg)`

A bi-directional match expander that parses a `symbol?` from the URL and generates a URL with a symbol's encoding as a string.

5.4 Extending `web-server/dispatch`

```
(require web-server/dispatch/extend)
```

You can create new URL component patterns by defining bi-directional match expanders.

`(define-bidi-match-expander` *`id in-xform out-xform`*`)`

Binds *`id`* to a *bi-directional match expander* where *`in-xform`* is a match expander (defined by `define-match-expander`) that is used when parsing URLs and *`out-xform`* is one used when generating URLs.

`bidi-match-going-in?`

A syntax parameter used by bi-directional match expanders to determine if a URL is being parsed or generated.

When defining new patterns, you may find it useful to use these helper functions:

`(define-coercion-match-expander` *`id test? coerce`*`)`

Binds *`id`* to a match expander that expands `(`*`id x`*`)` to `(? `*`test?`*` (app `*`coerce x`*`))` (i.e., uses *`test?`* to determine if the pattern matches and *`coerce`* to transform the binding.)

```
(make-coerce-safe? coerce) → (any/c . -> . boolean?)
  coerce : (any/c . -> . any/c)
```

Returns a function that returns `#t` if *coerce* would not throw an exception or return `#f` on its input.

Examples:

```
> (define string->number? (make-coerce-safe? string->number))
> (string->number? "1")
#t
> (string->number? "1.2")
#t
> (string->number? "+inf.0")
#t
> (string->number? "one")
#f
```

6 Formlets: Functional Form Abstraction

```
(require web-server/formlets)
```

The Web Server provides a kind of Web form abstraction called a formlet.

Formlets originate in the work of the Links research group in their paper The Essence of Form Abstraction.

6.1 Basic Formlet Usage

Suppose we want to create an abstraction of entering a date in an HTML form. The following formlet captures this idea:

```
(define date-formlet
  (formlet
   (div
    "Month:" ,{input-int . => . month}
    "Day:" ,{input-int . => . day})
   (list month day)))
```

The first part of the `formlet` syntax is the template of an X-expression that is the rendering of the formlet. It can contain elements like `,(=> formlet name)` where *formlet* is a formlet expression and *name* is an identifier bound in the second part of the `formlet` syntax.

This formlet is displayed (with `formlet-display`) as the following X-expression forest (list):

```
(list
 '(div "Month:" (input ([name "input_0"]))
       "Day:" (input ([name "input_1"])))))
```

`date-formlet` not only captures the rendering of the form, but also the request processing logic. If we send it an HTTP request with bindings for `"input_0"` to `"10"` and `"input_1"` to `"3"`, with `formlet-process`, then it returns:

```
(list 10 3)
```

which is the second part of the `formlet` syntax, where `month` has been replaced with the integer represented by the `"input_0"` and `day` has been replaced with the integer represented by the `"input_1"`.

The real power of formlet is that they can be embedded within one another. For instance, suppose we want to combine two date forms to capture a travel itinerary. The following formlet does the job:

```
(define travel-formlet
  (formlet
   (div
```

```
 "Name:" ,{input-string . => . name}
 (div
  "Arrive:" ,{date-formlet . => . arrive}
  "Depart:" ,{date-formlet . => . depart})
(list name arrive depart))))
```

(Notice that `date-formlet` is embedded twice.) This is rendered as:

```
(list
 '(div
   "Name:"
   (input ([name "input_0"]))
   (div
    "Arrive:"
    (div "Month:" (input ([name "input_1"]))
         "Day:" (input ([name "input_2"])))
    "Depart:"
    (div "Month:" (input ([name "input_3"]))
         "Day:" (input ([name "input_4"]))))))
```

Observe that `formlet-display` has automatically generated unique names for each input element. When we pass bindings for these names to `formlet-process`, the following list is returned:

```
(list "Jay"
      (list 10 3)
      (list 10 6))
```

The rest of the manual gives the details of formlet usage and extension.

6.2 Syntactic Shorthand

```
(require web-server/formlets/syntax)
```

Most users will want to use the syntactic shorthand for creating formlets.

(formlet *rendering yields-expr*)

Constructs a formlet with the specified *rendering* and the processing resulting in the *yields-expr* expression. The *rendering* form is a quasiquoted X-expression, with two special caveats:

,{=> *formlet-expr name*} embeds the formlet given by *formlet-expr*; the result of this processing this formlet is available in the *yields-expr* as *name*.

`(#%# xexpr ...)` renders an X-expression forest.

6.3 Functional Usage

```
(require web-server/formlets/lib)
```

The syntactic shorthand abbreviates the construction of *formlet*s with the following library. These combinators may be used directly to construct low-level formlets, such as those for new INPUT element types. Refer to §6.4 "Predefined Formlets" for example low-level formlets using these combinators.

`xexpr-forest/c : contract?`

Equivalent to `(listof xexpr/c)`

`(formlet/c content) → contract?`
`  content : any/c`

Equivalent to `(-> integer? (values xexpr-forest/c (-> (listof binding?) (coerce-contract 'formlet/c content)) integer?))`.

A formlet's internal representation is a function from an initial input number to an X-expression forest rendering, a processing function, and the next allowable input number.

`(pure value) → (formlet/c any/c)`
`  value : any/c`

Constructs a formlet that has no rendering and always returns `value` in the processing stage.

`(cross f g) → (formlet/c any/c)`
`  f : (formlet/c (any/c . -> . any/c))`
`  g : (formlet/c any/c)`

Constructs a formlet with a rendering equal to the concatenation of the renderings of formlets `f` and `g`; a processing stage that applies `g`'s processing result to `f`'s processing result.

`(cross* f g ...) → (formlet/c any/c)`
`  f : (formlet/c (() () #:rest (listof any/c) . ->* . any/c))`
`  g : (formlet/c any/c)`

Equivalent to `cross` lifted to many arguments.

```
(xml-forest r) → (formlet/c procedure?)
  r : xexpr-forest/c
```

Constructs a formlet with the rendering *r* and the identity procedure as the processing step.

```
(xml r) → (formlet/c procedure?)
  r : xexpr/c
```

Equivalent to `(xml-forest (list r))`.

```
(text r) → (formlet/c procedure?)
  r : string?
```

Equivalent to `(xml r)`.

```
(tag-xexpr tag attrs inner) → (formlet/c any/c)
  tag : symbol?
  attrs : (listof (list/c symbol? string?))
  inner : (formlet/c any/c)
```

Constructs a formlet with the rendering `(list (list* tag attrs inner-rendering))` where `inner-rendering` is the rendering of *inner* and the processing stage identical to *inner*.

```
(formlet-display f) → xexpr-forest/c
  f : (formlet/c any/c)
```

Renders *f*.

```
(formlet-process f r) → any/c
  f : (formlet/c any/c)
  r : request?
```

Runs the processing stage of *f* on the bindings in *r*.

6.4 Predefined Formlets

```
(require web-server/formlets/input)
```

These formlets are the main combinators for form input.

```
(make-input render) → (formlet/c (or/c false/c binding?))
  render : (string? . -> . xexpr/c)
```

This formlet is rendered with *render*, which is passed the input name, and results in the extracted binding.

```
(text-input [#:value value
             #:size size
             #:max-length max-length
             #:read-only? read-only?
             #:attributes attrs])
 → (formlet/c (or/c false/c binding?))
  value : (or/c false/c bytes?) = #f
  size : (or/c false/c exact-nonnegative-integer?) = #f
  max-length : (or/c false/c exact-nonnegative-integer?) = #f
  read-only? : boolean? = #f
  attrs : (listof (list/c symbol? string?)) = empty
```

This formlet renders using an INPUT element with the TEXT type and the attributes given in the arguments.

```
(password-input [#:value value
                 #:size size
                 #:max-length max-length
                 #:read-only? read-only?
                 #:attributes attrs])
 → (formlet/c (or/c false/c binding?))
  value : (or/c false/c bytes?) = #f
  size : (or/c false/c exact-nonnegative-integer?) = #f
  max-length : (or/c false/c exact-nonnegative-integer?) = #f
  read-only? : boolean? = #f
  attrs : (listof (list/c symbol? string?)) = empty
```

This formlet renders using an INPUT element with the PASSWORD type and the attributes given in the arguments.

```
(checkbox value checked? [#:attributes attrs])
 → (formlet/c (or/c false/c binding?))
  value : bytes?
  checked? : boolean?
  attrs : (listof (list/c symbol? string?)) = empty
```

This formlet renders using a INPUT elemen with the CHECKBOX type and the attributes

given in the arguments.

```
(required f) → (formlet/c bytes?)
  f : (formlet/c (or/c false/c binding?))
```

Constructs a formlet that extracts the `binding:form-value` from the binding produced by *f*, or errors.

```
(default def f) → (formlet/c bytes?)
  def : bytes?
  f : (formlet/c (or/c false/c binding?))
```

Constructs a formlet that extracts the `binding:form-value` from the binding produced by *f*, or returns *def*.

```
(to-string f) → (formlet/c string?)
  f : (formlet/c bytes?)
```

Converts *f*'s output to a string. Equivalent to `(cross (pure bytes->string/utf-8) f)`.

```
(to-number f) → (formlet/c number?)
  f : (formlet/c string?)
```

Converts *f*'s output to a number. Equivalent to `(cross (pure string->number) f)`.

```
(to-symbol f) → (formlet/c symbol?)
  f : (formlet/c string?)
```

Converts *f*'s output to a symbol. Equivalent to `(cross (pure string->symbol) f)`.

```
(to-boolean f) → (formlet/c boolean?)
  f : (formlet/c bytes?)
```

Converts *f*'s output to a boolean, if it is equal to `#"on"`.

```
input-string : (formlet/c string?)
```

Equivalent to `(to-string (required (text-input)))`.

```
input-int : (formlet/c integer?)
```

Equivalent to `(to-number input-string)`.

```
input-symbol : (formlet/c symbol?)
```

Equivalent to `(to-symbol input-string)`.

6.5 Utilities

```
(require web-server/formlets/servlet)
```

A few utilities are provided for using formlets in Web applications.

```
(send/formlet f [#:wrap wrapper]) → any/c
  f : (formlet/c any/c)
  wrapper : (xexpr/c . -> . response/c)
          = (lambda (form-xexpr)
              `(html (head (title "Form Entry"))
                     (body ,form-xexpr)))
```

Uses `send/suspend` to send *f*'s rendering (wrapped in a FORM tag whose action is the continuation URL (wrapped again by *wrapper*)) to the client. When the form is submitted, the request is passed to the processing stage of *f*.

```
(embed-formlet embed/url f) → xexpr/c
  embed/url : embed/url/c
  f : (formlet/c any/c)
```

Like `send/formlet`, but for use with `send/suspend/dispatch`.

7 Templates: Separation of View

```
(require web-server/templates)
```

The Web Server provides a powerful Web template system for separating the presentation logic of a Web application and enabling non-programmers to contribute to PLT-based Web applications.

Although all the examples here generate HTML, the template language and the §17 "Text Preprocessor" it is based on can be used to generate any text-based format: C, SQL, form emails, reports, etc.

7.1 Static

Suppose we have a file `"static.html"` with the contents:

```
<html>
 <head><title>Fastest Templates in the West!</title></head>
 <body>
  <h1>Bang!</h1>
  <h2>Bang!</h2>
 </body>
</html>
```

If we write the following in our code:

```
(include-template "static.html")
```

Then the contents of `"static.html"` will be read *at compile time* and compiled into a Scheme program that returns the contents of `"static.html"` as a string:

```
  "<html>\n  <head><title>Fastest Templates in the
West!</title></head>\n  <body>\n    <h1>Bang!</h1>\n    <h2>Bang!</h2>\n  </body>\n</html>"
```

7.2 Dynamic

`include-template` gives the template access to the *complete lexical context* of the including program. This context can be accessed via the §3 "@-Reader" syntax. For example, if `"simple.html"` contains:

```
<html>
 <head><title>Fastest @thing in the West!</title></head>
 <body>
  <h1>Bang!</h1>
  <h2>Bang!</h2>
 </body>
</html>
```

Then

```
(let ([thing "Templates"])
  (include-template "simple.html"))
```

evaluates to the same content as the static example.

There are no constraints on how the lexical context of the template is populated. For instance, you can built template abstractions by wrapping the inclusion of a template in a function:

```
(define (fast-template thing)
  (include-template "simple.html"))

(fast-template "Templates")
(fast-template "Noodles")
```

evalutes to two strings with the predictable contents:

```
<html>
 <head><title>Fastest Templates in the West!</title></head>
 <body>
  <h1>Bang!</h1>
  <h2>Bang!</h2>
 </body>
</html>
```

and

```
<html>
 <head><title>Fastest Noodles in the West!</title></head>
 <body>
  <h1>Bang!</h1>
  <h2>Bang!</h2>
 </body>
</html>
```

Furthermore, there are no constraints on the Scheme used by templates: they can use macros, structs, continuation marks, threads, etc. However, Scheme values that are ultimately returned must be printable by the §17 “Text Preprocessor”. For example, consider the following outputs of the title line of different calls to `fast-template`:

- `(fast-template 'Templates)`

  `<head><title>Fastest Templates in the West!</title></head>`

- `(fast-template 42)`

  `<head><title>Fastest 42 in the West!</title></head>`

- `(fast-template (list "Noo" "dles"))`

  `<head><title>Fastest Noodles in the West!</title></head>`

- `(fast-template (lambda () "Thunks"))`

  `<head><title>Fastest Thunks in the West!</title></head>`

- `(fast-template (delay "Laziness"))`

  `<head><title>Fastest Laziness in the West!</title></head>`

7.3 Gotchas

To obtain an `@` character in template output, you must escape the it, because it is the escape character of the §3 "@-Reader" syntax. For example, to obtain:

```
<head><title>Fastest @s in the West!</title></head>
```

You must write:

```
<head><title>Fastest @"@"s in the West!</title></head>
```

as your template: literal `@`s must be replaced with `@"@"`. (Note that the double-quotes are basically a Scheme expression, which can be used for longer strings too.)

The §3 "@-Reader" will read Scheme identifiers, so it does not terminate identifiers on punctuations or XML angle brackets. So,

```
<head><title>Fastest @thing in the @place!</title></head>
```

will complain that the identifier `place!</title></head>` is undefined. You can subvert this by explicitly delimiting the identifer:

```
<head><title>Fastest @thing in the @|place|!</title></head>
```

Another gotcha is that since the template is compiled into a Scheme program, only its results will be printed. For example, suppose we have the template:

```
<table>
 @for[([c clients])]{
  <tr><td>@(car c), @(cdr c)</td></tr>
 }
</table>
```

If this is included in a lexical context with `clients` bound to

```
(list (cons "Young" "Brigham") (cons "Smith" "Joseph"))
```

then the template will be printed as:

```
<table>
</table>
```

because `for` does not return the value of the body. Suppose that we change the template to use `for/list` (which combines them into a list):

```
<table>
 @for/list[([c clients])]{
  <tr><td>@(car c), @(cdr c)</td></tr>
 }
</table>
```

Now the result is:

```
<table>
 </tr>
 </tr>
</table>
```

because only the final expression of the body of the `for/list` is included in the result. We can capture all the sub-expressions by using `list` in the body:

```
<table>
 @for/list[([c clients])]{
  @list{
   <tr><td>@(car c), @(cdr c)</td></tr>
  }
 }
</table>
```

Now the result is:

```
<table>
 <tr><td>Young, Brigham</td></tr>
 <tr><td>Smith, Joseph</td></tr>
</table>
```

The templating library provides a syntactic form to deal with this issue for you called `in`:

```
<table>
 @in[c clients]{
  <tr><td>@(car c), @(cdr c)</td></tr>
 }
</table>
```

Notice how it also avoids the absurd amount of punctuation on line two.

7.4 HTTP Responses

The quickest way to generate an HTTP response from a template is using the `list` response type:

```
(list #"text/html" (include-template "static.html"))
```

If you want more control then you can generate a `response/full` struct:

```
(make-response/full
 200 #"Okay"
 (current-seconds) TEXT/HTML-MIME-TYPE
 empty
 (list (include-template "static.html")))
```

Finally, if you want to include the contents of a template inside a larger X-expression :

```
`(html ,(include-template "static.html"))
```

will result in the literal string being included (and entity-escaped). If you actually want the template to be unescaped, then create a `cdata` structure:

```
`(html ,(make-cdata #f #f (include-template "static.html")))
```

7.5 API Details

`(include-template` *path*`)`

Compiles the template at *path* using the §3 "@-Reader" syntax within the enclosing lexical context.

Example:

```
(include-template "static.html")
```

`(in` *x xs e ...*`)`

Expands into

```
(for/list ([x xs])
 (begin/text e ...))
```

Template Example:

```
@in[c clients]{
 <tr><td>@(car c), @(cdr c)</td></tr>
}
```

Scheme Example:

```
(in c clients "<tr><td>" (car c) ", " (cdr c) "</td></tr>")
```

7.6 Conversion Example

Al Church has been maintaining a blog with PLT Scheme for some years and would like to convert to `web-server/templates`.

The data-structures he uses are defined as:

```
(define-struct post (title body))

(define posts
  (list
   (make-post
    "(Y Y) Works: The Why of Y"
    "Why is Y, that is the question.")
   (make-post
    "Church and the States"
    "As you may know, I grew up in DC, not technically a state.")))
```

Actually, Al Church-encodes these posts, but for explanatory reasons, we'll use structs.

He has divided his code into presentation functions and logic functions. We'll look at the presentation functions first.

The first presentation function defines the common layout of all pages.

```
(define (template section body)
  `(html
    (head (title "Al's Church: " ,section))
    (body
     (h1 "Al's Church: " ,section)
     (div ([id "main"])
          ,@body))))
```

One of the things to notice here is the `unquote-splicing` on the `body` argument. This indicates that the `body` is list of X-expressions. If he had accidentally used only `unquote` then there would be an error in converting the return value to an HTTP response.

```
(define (blog-posted title body k-url)
```

```
  `((h2 ,title)
    (p ,body)
    (h1 (a ([href ,k-url]) "Continue"))))
```

Here's an example of simple body that uses a list of X-expressions to show the newly posted blog entry, before continuing to redisplay the main page. Let's look at a more complicated body:

```
(define (blog-posts k-url)
  (append
   (apply append
          (for/list ([p posts])
            `((h2 ,(post-title p))
              (p ,(post-body p)))))
   `((h1 "New Post")
     (form ([action ,k-url])
           (input ([name "title"]))
           (input ([name "body"]))
           (input ([type "submit"]))))))
```

This function shows a number of common patterns that are required by X-expressions. First, `append` is used to combine different X-expression lists. Second, `apply append` is used to collapse and combine the results of a `for/list` where each iteration results in a list of X-expressions. We'll see that these patterns are unnecessary with templates. Another annoying patterns shows up when Al tries to add CSS styling and some JavaScript from Google Analytics to all the pages of his blog. He changes the `template` function to:

```
(define (template section body)
  `(html
    (head
     (title "Al's Church: " ,section)
     (style ([type "text/css"])
            "body {margin: 0px; padding: 10px;}"
            "#main {background: #dddddd;}"))
    (body
     (script
      ([type "text/javascript"])
      ,(make-cdata
        #f #f
        "var gaJsHost = ((\"https:\" =="
        "document.location.protocol)"
        "? \"https://ssl.\" : \"http://www.\");"
        "document.write(unescape(\"%3Cscript src='\" + gaJsHost"
        "+ \"google-analytics.com/ga.js' "
        "type='text/javascript'%3E%3C/script%3E\"));"))
     (script
```

```
      ([type "text/javascript"])
      ,(make-cdata
        #f #f
        "var pageTracker = _gat._getTracker(\"UA-YYYYYYY-Y\");"
        "pageTracker._trackPageview();"))
     (h1 "Al's Church: " ,section)
     (div ([id "main"])
          ,@body))))
```

Some of these problems go away by using here strings, as described in the documentation on §12.6.6 "Reading Strings".

The first thing we notice is that encoding CSS as a string is rather primitive. Encoding JavaScript with strings is even worse for two reasons: first, we are more likely to need to manually escape characters such as "; second, we need to use a CDATA object, because most JavaScript code uses characters that "need" to be escaped in XML, such as &, but most browsers will fail if these characters are entity-encoded. These are all problems that go away with templates.

Before moving to templates, let's look at the logic functions:

```
(define (extract-post req)
  (define binds
    (request-bindings req))
  (define title
    (extract-binding/single 'title binds))
  (define body
    (extract-binding/single 'body binds))
  (set! posts
        (list* (make-post title body)
               posts))
  (send/suspend
   (lambda (k-url)
     (template "Posted" (blog-posted title body k-url))))
  (display-posts))

(define (display-posts)
  (extract-post
   (send/suspend
    (lambda (k-url)
      (template "Posts" (blog-posts k-url))))))

(define (start req)
  (display-posts))
```

To use templates, we need only change `template`, `blog-posted`, and `blog-posts`:

```
(define (template section body)
  (list TEXT/HTML-MIME-TYPE
```

```
        (include-template "blog.html")))

(define (blog-posted title body k-url)
  (include-template "blog-posted.html"))

(define (blog-posts k-url)
  (include-template "blog-posts.html"))
```

Each of the templates are given below:

"blog.html":

```
<html>
 <head>
  <title>Al's Church: @|section|</title>
  <style type="text/css">
   body {
    margin: 0px;
    padding: 10px;
   }

   #main {
    background: #dddddd;
   }
  </style>
 </head>
 <body>
  <script type="text/javascript">
   var gaJsHost = (("https:" == document.location.protocol) ?
     "https://ssl." : "http://www.");
   document.write(unescape("%3Cscript src='" + gaJsHost +
     "google-analytics.com/ga.js'
      type='text/javascript'%3E%3C/script%3E"));
  </script>
  <script type="text/javascript">
   var pageTracker = _gat._getTracker("UA-YYYYYYY-Y");
   pageTracker._trackPageview();
  </script>

  <h1>Al's Church: @|section|</h1>
  <div id="main">
    @body
  </div>
 </body>
</html>
```

Notice that this part of the presentation is much simpler, because the CSS and JavaScript can be included verbatim, without resorting to any special escape-escaping patterns. Similarly, since the `body` is represented as a string, there is no need to remember if splicing is necessary.

```
"blog-posted.html":

  <h2>@|title|</h2>
  <p>@|body|</p>

  <h1><a href="@|k-url|">Continue</a></h1>

"blog-posts.html":

  @in[p posts]{
   <h2>@(post-title p)</h2>
   <p>@(post-body p)</p>
  }

  <h1>New Post</h1>
  <form action="@|k-url|">
   <input name="title" />
   <input name="body" />
   <input type="submit" />
  </form>
```

Compare this template with the original presentation function: there is no need to worry about managing how lists are nested: the defaults *just work*.

8 Troubleshooting and Tips

8.1 Why are my servlets not updating on the server when I change the code on disk?

By default, the server uses `make-cached-url->servlet` to load servlets from the disk. As it loads them, they are cached and the disk is not referred to for future requests. This ensures that there is a single namespace for each servlet, so that different instances can share resources, such as database connections, and communicate through the store. The default configuration of the server (meaning the dispatcher sequence used when you load a configuration file) provides a special URL to localhost that will reset the cache: `"/conf/refresh-servlets"`. If you want the server to reload your changed servlet code, then GET this URL and the server will reload the servlet on the next request.

8.2 What special considerations are there for security with the Web Server?

The biggest problem is that a naive usage of continuations will allow continuations to subvert authentication mechanisms. Typically, all that is necessary to execute a continuation is its URL. Thus, URLs must be as protected as the information in the continuation.

Consider if you link to a public site from a private continuation URL: the `Referrer` field in the new HTTP request will contain the private URL. Furthermore, if your HTTP traffic is in the clear, then these URLs can be easily poached.

One solution to this is to use a special cookie as an authenticator. This way, if a URL escapes, it will not be able to be used, unless the cookie is present. For advice about how to do this well, see Dos and Don'ts of Client Authentication on the Web from the MIT Cookie Eaters.

Note: It may be considered a great feature that URLs can be shared this way, because delegation is easily built into an application via URLs.

8.3 IE ignores my CSS or behaves strange in other ways

In quirks mode, IE does not parse your page as XML, in particular it will not recognize many instances of "empty tag shorthand", e.g. "", whereas the Web Server uses `xml` to format XML, which uses empty tag shorthand by default. You can change the default with the `empty-tag-shorthand` parameter: `(empty-tag-shorthand 'never)`.

Index