File: Racket File Format Libraries

Version 5.0

June 6, 2010

Contents

1 gzip Compression and File Creation

```
(require file/gzip)
```

The file/gzip library provides utilities to create archive files in gzip format, or simply to compress data using the pkzip "deflate" method.

```
(gzip in-file [out-file]) → void?
  in-file : path-string?
  out-file : path-string? = (string-append in-file ".gz")
```

Compresses data to the same format as the gzip utility, writing the compressed data directly to a file. The *in-file* argument is the name of the file to compress. If the file named by *out-file* exists, it will be overwritten.

```
(gzip-through-ports in
                    out
                    orig-filename
                    timestamp)    → void?
  in : input-port?
  out : output-port?
  orig-filename : (or/c string? false/c)
  timestamp : exact-integer?
```

Reads the port *in* for data and compresses it to *out*, outputting the same format as the gzip utility. The *orig-filename* string is embedded in this output; *orig-filename* can be #f to omit the filename from the compressed stream. The *timestamp* number is also embedded in the output stream, as the modification date of the original file (in Unix seconds, as file-or-directory-modify-seconds would report under Unix).

```
(deflate in out) → exact-nonnegative-integer?
                   exact-nonnegative-integer?
                   exact-nonnegative-integer?
  in : input-port?
  out : output-port?
```

Writes pkzip-format "deflated" data to the port *out*, compressing data from the port *in*. The data in a file created by gzip uses this format (preceded with header information).

The result is three values: the number of bytes read from *in*, the number of bytes written to *out*, and a cyclic redundancy check (CRC) value for the input.

2 gzip Decompression

```
(require file/gunzip)
```

The file/gunzip library provides utilities to decompress archive files in gzip format, or simply to deccompress data using the pkzip "inflate" method.

```
(gunzip file [output-name-filter]) → void?
  file : path-string?
  output-name-filter : (string? boolean? . -> . path-string?)
                     = (lambda (file archive-supplied?) file)
```

Extracts data that was compressed using the gzip utility (or gzip function), writing the uncompressed data directly to a file. The *file* argument is the name of the file containing compressed data. The default output file name is the original name of the compressed file as stored in *file*. If a file by this name exists, it will be overwritten. If no original name is stored in the source file, "unzipped" is used as the default output file name.

The *output-name-filter* procedure is applied to two arguments—the default destination file name and a boolean that is #t if this name was read from *file*—before the destination file is created. The return value of the file is used as the actual destination file name (to be opened with the 'truncate flag of open-output-file).

If the compressed data turns out to be corrupted, the exn:fail exception is raised.

```
(gunzip-through-ports in out) → void?
  in : input-port?
  out : output-port?
```

Reads the port *in* for compressed data that was created using the gzip utility, writing the uncompressed data to the port *out*.

If the compressed data turns out to be corrupted, the exn:fail exception is raised. The unzipping process may peek further into *in* than needed to decompress the data, but it will not consume the unneeded bytes.

```
(inflate in out) → void?
  in : input-port?
  out : output-port?
```

Reads pkzip-format "deflated" data from the port *in* and writes the uncompressed ("inflated") data to the port *out*. The data in a file created by gzip uses this format (preceded with some header information).

If the compressed data turns out to be corrupted, the `exn:fail` exception is raised. The inflate process may peek further into *in* than needed to decompress the data, but it will not consume the unneeded bytes.

3 `zip` File Creation

```
(require file/zip)
```

The `file/zip` library provides utilities to create `zip` archive files, which are compatible with both Windows and Unix (including Mac OS X) unpacking. The actual compression is implemented by `deflate`.

```
(zip zip-file path ...) → void?
  zip-file : path-string?
  path : path-string?
```

Creates *zip-file*, which holds the complete content of all *path*s. The given *path*s are all expected to be relative path names of existing directories and files (i.e., relative to the current directory). If a nested path is provided as a *path*, its ancestor directories are also added to the resulting zip file, up to the current directory (using `pathlist-closure`). Files are packaged as usual for `zip` files, including permission bits for both Windows and Unix (including Mac OS X). The permission bits are determined by `file-or-directory-permissions`, which does not preserve the distinction between owner/group/other permissions. Also, symbolic links are always followed.

```
(zip->output paths [out]) → void?
  paths : (listof path-string?)
  out : output-port? = (current-output-port)
```

Zips each of the given *paths*, and packages it as a zip "file" that is written directly to *out*. Unlike `zip`, the specified *paths* are included as-is; if a directory is specified, its content is not automatically added, and nested directories are added without parent directories.

```
(zip-verbose) → boolean?
(zip-verbose on?) → void?
  on? : any/c
```

A parameter that controls output during a `zip` operation. Setting this parameter to a true value causes `zip` to display to `(current-error-port)` the filename that is currently being compressed.

4 `tar` File Creation

```
(require file/tar)
```

The `file/tar` library provides utilities to create archive files in USTAR format, like the archive that the Unix utility `pax` generates. The USTAR format imposes limits on path lengths. The resulting archives contain only directories and files (symbolic links are followed), and owner information is not preserved; the owner that is stored in the archive is always "root."

```
(tar tar-file path ...) → exact-nonnegative-integer?
  tar-file : path-string?
  path : path-string?
```

Creates *tar-file*, which holds the complete content of all *path*s. The given *path*s are all expected to be relative path names of existing directories and files (i.e., relative to the current directory). If a nested path is provided as a *path*, its ancestor directories are also added to the resulting tar file, up to the current directory (using `pathlist-closure`).

```
(tar->output paths [out]) → exact-nonnegative-integer?
  paths : (listof path?)
  out : output-port? = (current-output-port)
```

Packages each of the given *paths* in a `tar` format archive that is written directly to the *out*. The specified *paths* are included as-is; if a directory is specified, its content is not automatically added, and nested directories are added without parent directories.

```
(tar-gzip tar-file paths ...) → void?
  tar-file : path-string?
  paths : path-string?
```

Like `tar`, but compresses the resulting file with `gzip`.

5 MD5 Message Digest

```
(require file/md5)
```

```
(md5 in) → bytes?
  in : (or/c input-port? bytes? string?)
```

Produces a byte string containing 32 hexadecimal digits (lowercase) that is the MD5 hash of the given input stream or byte string.

Example:

```
> (md5 #"abc")
#"900150983cd24fb0d6963f7d28e17f72"
```

6 GIF File Writing

```
(require file/gif)
```

The `file/gif` library provides functions for writing GIF files to a stream, including GIF files with multiple images and controls (such as animated GIFs).

A GIF stream is created by `gif-start`, and then individual images are written with `gif-add-image`. Optionally, `gif-add-control` inserts instructions for rendering the images. The `gif-end` function ends the GIF stream.

A GIF stream can be in any one of the following states:

- `'init` : no images or controls have been added to the stream
- `'image-or-control` : another image or control can be written
- `'image` : another image can be written (but not a control, since a control was written)
- `'done` : nothing more can be added

```
(gif-stream? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a GIF stream created by `gif-write`, `#f` otherwise.

```
(image-ready-gif-stream? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a GIF stream that is not in `'done` mode, `#f` otherwise.

```
(image-or-control-ready-gif-stream? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a GIF stream that is in `'init` or `'image-or-control` mode, `#f` otherwise.

```
(empty-gif-stream? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a GIF stream that in `'init` mode, `#f` otherwise.

```
(gif-colormap? v) → boolean?
```

```
  v : any/c
```

Returns `#t` if *v* represets a colormap, `#f` otherwise. A colormap is a list whose size is a power of 2 between 2^1 and 2^8, and whose elements are vectors of size 3 containing colors (i.e., exact integers between 0 and 255 inclusive).

```
(color? v) → boolean?
  v : any/c
```

The same as `byte?`.

```
(dimension? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an exact integer between `0` and `65535` inclusive, `#f` otherwise.

```
(gif-state stream) → symbol?
  stream : gif-stream?
```

Returns the state of *stream*.

```
(gif-start out w h bg-color cmap) → gif-stream?
  out : output-port?
  w : dimension?
  h : dimension?
  bg-color : color?
  cmap : (or/c false/c gif-colormap?)
```

Writes the start of a GIF file to the given output port, and returns a GIF stream that adds to the output port.

The width and height determine a virtual space for the overall GIF image. Individual images added to the GIF stream must fit within this virtual space. The space is initialized by the given background color.

Finally, the default meaning of color numbers (such as the background color) is determined by the given colormap, but individual images within the GIF file can have their own colormaps.

A global colormap need not be supplied, in which case a colormap must be supplied for each image. Beware that the bg-color is ill-defined if a global colormap is not provided.

```
(gif-add-image stream
               left
               top
               width
               height
               interlaced?
               cmap
               bstr)       → void?
  stream : image-ready-gif-stream?
  left : dimension?
  top : dimension?
  width : dimension?
  height : dimension?
  interlaced? : any/c
  cmap : (or/c false/c gif-colormap?)
  bstr : bytes?
```

Writes an image to the given GIF stream. The *left*, *top*, *width*, and *height* values specify the location and size of the image within the overall GIF image's virtual space.

If *interlaced?* is true, then *bstr* should provide bytes ininterlaced order instead of top-to-bottom order. Interlaced order is:

- every 8th row, starting with 0
- every 8th row, starting with 4
- every 4th row, starting with 2
- every 2nd row, starting with 1

If a global color is provided with gif-start, a #f value can be provided for *cmap*.

The *bstr* argument specifies the pixel content of the image. Each byte specifies a color (i.e., an index in the colormap). Each row is provided left-to-right, and the rows provided either top-to-bottom or in interlaced order (see above). If the image is prefixed with a control that specifies an transparent index (see gif-add-control), then the corresponding "color" doesn't draw into the overall GIF image.

An exception is raised if any byte value in *bstr* is larger than the colormap's length, if the *bstr* length is not *width* times *height*, or if the *top*, *left*, *width*, and *height* dimensions specify a region beyond the overall GIF image's virtual space.

```
(gif-add-control stream
                 disposal
                 wait-for-input?
                 delay
                 transparent)    → void?
  stream : image-or-control-ready-gif-stream?
  disposal : (one-of/c 'any 'keep 'restore-bg 'restore-prev)
  wait-for-input? : any/c
  delay : dimension?
  transparent : (or/c false/c color?)
```

Writes an image-control command to a GIF stream. Such a control must appear just before an image, and it applies to the following image.

The GIF image model involves processing images one by one, placing each image into the specified position within the overall image's virtual space. An image-control command can specify a delay before an image is added (to create animated GIFs), and it also specifies how the image should be kept or removed from the overall image before proceeding to the next one (also for GIF animation).

The *disposal* argument specifies how to proceed:

- `'any` : doesn't matter (perhaps because the next image completely overwrites the current one)
- `'keep` : leave the image in place
- `'restore-bg` : replace the image with the background color
- `'restore-prev` : restore the overall image content to the content before the image is added

If *wait-for-input?* is true, then the display program may wait for some cue from the user (perhaps a mouse click) before adding the image.

The *delay* argument specifies a delay in 1/100s of a second.

If the *transparent* argument is a color, then it determines an index that is used to represent transparent pixels in the follow image (as opposed to the color specified by the colormap for the index).

An exception is raised if a control is already added to *stream* without a corresponding image.

```
(gif-add-loop-control stream iteration) → void?
```

```
  stream : empty-gif-stream?
  iteration : dimension?
```

Writes a control command to a GIF stream for which no images or other commands have already been written. The command causes the animating sequence of images in the GIF to be repeated 'iteration-dimension' times, where 0 can be used to mean "infinity."

An exception is raise if some control or image has been added to the stream already.

```
(gif-add-comment stream bstr) → void?
  stream : image-or-control-ready-gif-stream?
  bstr : bytes?
```

Adds a generic comment to the GIF stream.

An exception is raised if an image-control command was just written to the stream (so that an image is required next).

```
(gif-end stream) → void?
  stream : image-or-control-ready-gif-stream?
```

Finishes writing a GIF file. The GIF stream's output port is not automatically closed.

An exception is raised if an image-control command was just written to the stream (so that an image is required next).

```
(quantize bstr) → bytes? gif-colormap? (or/c false/c color?)
  bstr : argb-bytes?
```

Each image in a GIF stream is limited to 256 colors, including the transparent "color," if any. The `quantize` function converts a 24-bit image (plus alpha channel) into an indexed-color image, reducing the number of colors if necessary.

Given a set of pixels expressed in ARGB format (i.e., each four bytes is a set of values for one pixel: alpha, red, blue, and green), `quantize` produces produces

- bytes for the image (i.e., a array of colors, expressed as a byte string)
- a colormap
- either `#f` or a color index for the transparent "color"

The conversion treats alpha values less than 128 as transparent pixels, and other alpha values as solid.

The quantization process uses Octrees [Gervautz1990] to construct an adaptive palette for all (non-transparent) colors in the image. This implementation is based on an article by Dean Clark [Clark1996].

To convert a collection of images all with the same quantization, simply append them for the input of a single call of `quantize`, and then break apart the result bytes.

Bibliography

[Gervautz1990] M. Gervautz and W. Purgathofer, "A simple method for color quantization: Octree quantization," Graphics Gems, 1990.

[Clark1996] Dean Clark, "Color Quantization using Octrees," Dr. Dobbs Journal, January 1, 1996. http://www.ddj.com/184409805

Index