setup-plt: PLT Configuration and Installation

Version 4.0.1

June 22, 2008

The setup-plt executable finds, compiles, configures, and installs documentation for all collections in a PLT Scheme installation. It can also install single ".plt" files.

Contents

1 Running `setup-plt` Executable **3**

1.1 Controlling `setup-plt` with `"info.ss"` Files 3

2 Running `setup-plt` from Scheme **8**

2.1 Options Unit 8

2.2 Options Signature 9

3 `".plt"` Archives **12**

3.1 Making `".plt"` Archives 12

3.2 Installing a Single `".plt"` File 16

3.2.1 Non-GUI Installer 16

3.2.2 GUI Installer 16

3.2.3 GUI Unpacking Signature 17

3.2.4 GUI Unpacking Unit 18

3.3 Unpacking `".plt"` Archives 18

3.4 Format of `".plt"` Archives 20

4 Finding Installation Directories **23**

5 `"info.ss"` File Format **26**

6 Reading `"info.ss"` Files **28**

7 Paths Relative to `"collects"` **30**

8 Cross-References for Installed Manuals **31**

Index **32**

1 Running setup-plt Executable

The setup-plt executable performs two main services:

- **Compiling and setting up all (or some of the) collections:** When setup-plt is run without any arguments, it finds all of the current collections (see §16.2 "Libraries and Collections") and compiles libraries in each collection.

  An optional "info.ss" within the collection can indicate specifically how the collection's files are to be compiled and other actions to take in setting up a collection, such as creating executables or building documentation. See §1.1 "Controlling setup-plt with "info.ss" Files" for more information.

  The --clean (or -c) flag to setup-plt causes it to delete existing ".zo" files, thus ensuring a clean build from the source files. The exact set of deleted files can be controlled by "info.ss"; see clean for more information.

  The -l flag takes one or more collection names and restricts setup-plt's action to those collections.

  The --mode ⟨*mode*⟩ flag causes setup-plt to use a ".zo" compiler other than the default compiler, and to put the resulting ".zo" files in a subdirectory (of the usual place) named by ⟨*mode*⟩. The compiler is obtained by using ⟨*mode*⟩ as a collection name, finding a "zo-compile.ss" module in that collection, and extracting its zo-compile export. The zo-compile export should be a function like compile; see the "errortrace" collection for an example.

- **Unpacking ".plt" files:** A ".plt" file is a platform-independent distribution archive for software based on PLT Scheme. When one or more file names are provided as the command line arguments to setup-plt, the files contained in the ".plt" archive are unpacked (according to specifications embedded in the ".plt" file) and only collections specified by the ".plt" file are compiled and setup.

Run setup-plt with the -h flag to see a list of all options accepted by the setup-plt executable.

1.1 Controlling setup-plt with "info.ss" Files

To compile a collection's files to bytecode, setup-plt uses the compile-collection-zos procedure. That procedure, in turn, consults the collection's "info.ss" file, if it exists, for specific instructions on compiling the collection. See compile-collection-zos for more information on the fields of "info.ss" that it uses, and see §5 ""info.ss" File Format" for information on the format of an "info.ss" file.

Optional "info.ss" fields trigger additional actions by setup-plt:

- `scribblings` : `(listof (cons/c string? list?))` — A list of documents to build. Each document in the list is itself represented as a list, where each document's list starts with a string that is a collection-relative path to the document's source file.

  More precisely a `scribblings` entry must be a value that can be generated from an expression matching the following `entry` grammar:

```
  entry = (list doc ...)

    doc = (list src-string)
        | (list src-string flags)
        | (list src-string flags category)
        | (list src-string flags category name-string)

  flags = (list mode-symbol ...)

category = (list category-symbol)
         | (list category-symbol sort-number)
```

  A document's list optionally continues with information on how to build the document. If a document's list contains a second item, it must be a list of mode symbols (described below). If a document's list contains a third item, it must be a list that categorizes the document (described further below). If a document's list contains a fourth item, it is a name to use for the generated documentation, instead of defaulting to the source file's name (sans extension).

  Each mode symbol in *flags* can be one of the following, where only `'multi-page` is commonly used:

  - `'multi-page` : Generates multi-page HTML output, instead of the default single-page format.
  - `'main-doc` : Indicates that the generated documentation should be written into the main installation directory, instead of to a user-specific directory. This mode is the default for a collection that is itself located in the main installation.
  - `'user-doc` : Indicates that the generated documentation should be written a user-specific directory. This mode is the default for a collection that is not itself located in the main installation.
  - `'depends-all` : Indicates that the document should be re-built if any other document is rebuilt—except for documents that have the `'no-depends-on` mode.
  - `'depends-all-main` : Indicates that the document should be re-built if any other document is rebuilt that is installed into the main installation—except for documents that have the `'no-depends-on` mode.
  - `'always-run` : Build the document every time that `setup-plt` is run, even if none of its dependencies change.
  - `'no-depend-on` : Removes the document for consideration for other dependencies. This mode is typically used with `'always-run` to avoid unnecessary dependencies that prevent reaching a stable point in building documentation.

- `'main-doc-root` : Designates the root document for the main installation. The document that currently has this mode should be the only one with the mode.
- `'user-doc-root` : Designates the root document for the user-specific documentation directory. The document that currently has this mode should be the only one with the mode.

The *category* list specifies how to show the document in the root table of contents. The list must start with a symbol, usually one of the following categories, which are ordered as below in the root documentation page:

- `'getting-started` : High-level, introductory documentation.
- `'language` : Documentation for a prominent programming language.
- `'tool` : Documentation for an executable.
- `'gui-library` : Documentation for GUI and graphics libraries.
- `'net-library` : Documentation for networking libraries.
- `'parsing-library` : Documentation for parsing libraries.
- `'tool-library` : Documentation for programming-tool libraries (i.e., not important enough for the more prominent `'tool` category).
- `'interop` : Documentation for interoperability tools and libraries.
- `'library` : Documentation for libraries; this category is the default and used for unrecognized category symbols.
- `'legacy` : Documentation for deprecated libraries, languages, and tools.
- `'experimental` : Documentation for an experimental language or library.
- `'other` : Other documentation.
- `'omit` : Documentation that should not be listed on the root page.

If the category list has a second element, it must be a real number that designates the manual's sorting position with the category; manuals with the same sorting position are ordered alphabetically. For a pair of manuals with sorting numbers *n* and *m*, the groups for the manuals are separated by space if `(truncate (/ n 10))`and `(truncate (/ m 10))` are different.

- `mzscheme-launcher-names` : `(listof string?)` — A list of executable names to be generated in the installation's executable directory to run MzScheme-based programs implemented by the collection. A parallel list of library names must be provided by `mzscheme-launcher-libraries` or `mzscheme-launcher-flags`.

  For each name, a launching executable is set up using `make-mzscheme-launcher`. The arguments are `-l-` and ⟨*colls*⟩`/.../`⟨*file*⟩, where ⟨*file*⟩ is the file named by `mzscheme-launcher-libraries` and ⟨*colls*⟩`/...` are the collections (and subcollections) of the `"info.ss"` file.

  In addition,

```
(build-aux-from-path
 (build-path (collection-path ⟨colls⟩ ...) ⟨suffixless-file⟩))
```

is provided for the optional `aux` argument (for icons, etc.) to `make-mzscheme-launcher`, where where ⟨*suffixless-file*⟩ is ⟨*file*⟩ without its suffix.

If `mzscheme-launcher-flags` is provided, it is used as a list of command-line arguments passed to `mzscheme` instead of the above default, allowing arbitrary command-line arguments. If `mzscheme-launcher-flags` is specified together with `mzscheme-launcher-libraries`, then the flags will override the libraries, but the libraries can still be used to specify a name for `build-aux-from-path` (to find related information like icon files etc).

- `mzscheme-launcher-libraries` : `(listof path-string?)` — A list of library names in parallel to `mzscheme-launcher-names`.
- `mzscheme-launcher-flags` : `(listof string?)` — A list of command-line flag lists, in parallel to `mzscheme-launcher-names`.
- `mred-launcher-names` : `(listof string?)` — Like `mzscheme-launcher-names`, but for MrEd-based executables. The launcher-name list is treated in parallel to `mred-launcher-libraries` and `mred-launcher-flags`.
- `mred-launcher-libraries` : `(listof path-string?)` — A list of library names in parallel to `mred-launcher-names`.
- `mred-launcher-flags` : `(listof string?)` — A list of command-line flag lists, in parallel to `mred-launcher-names`.
- `install-collection` : `path-string?` — A library module relative to the collection that provides `installer`. The `installer` procedure accepts either one or two arguments. The first argument is a directory path to the parent of the PLT installation's `"collects"` directory; the second argument, if accepted, is a path to the collection's own directory. The procedure should perform collection-specific installation work, and it should avoid unnecessary work in the case that it is called multiple times for the same installation.
- `pre-install-collection` : `path-string?` — Like `install-collection`, except that the corresponding installer is called *before* the normal `".zo"` build, instead of after. The provided procedure should be named `pre-installer` in this case, so it can be provided by the same file that provides an `installer`.
- `post-install-collection` : `path-string?` — Like `install-collection`. It is called right after the `install-collection` procedure is executed. The only difference between these is that the `--no-install` flag can be used to disable the previous two installers, but not this one. It is therefore expected to perform operations that are always needed, even after an installation that contains pre-compiled files. The provided procedure should be named `post-installer` in this case, so it can be provided by the same file that provides the previous two.

- `clean` : `(listof path-string?)` — A list of pathnames to be deleted when the `--clean` or `-c` flag is passed to `setup-plt`. The pathnames must be relative to the collection. If any path names a directory, each of the files in the directory are deleted, but none of the subdirectories of the directory are checked. If the path names a file, the file is deleted. The default, if this flag is not specified, is to delete all files in the `"compiled"` subdirectory, and all of the files in the platform-specific subdirectory of the compiled directory for the current platform.

  Just as compiling `".zo"` files will compile each module used by a compiled module, deleting a module's compiled image will delete the `".zo"` of each module that is used by the module. More specifically, used modules are determined when deleting a `".dep"` file, which would have been created to accompany a `".zo"` file when the `".zo"` was built by `setup-plt`. If the `".dep"` file indicates another module, that module's `".zo"` is deleted only if it also has an accompanying `".dep"` file. In that case, the `".dep"` file is deleted, and additional used modules are deleted based on the used module's `".dep"` file, etc. Supplying a specific list of collections to `setup-plt` disables this dependency-based deletion of compiled files.

2 Running setup-plt from Scheme

The setup/setup-unit library provides setup-plt in unit form. The associated setup/option-sig and setup/option-unit libraries provides the interface for setting options for the run of setup-plt.

For example, to unpack a single ".plt" archive "x.plt", set the archives parameter to (list "x.plt") and leave specific-collections as null.

Link the options and setup units so that your option-setting code is initialized between them, e.g.:

```
(compound-unit
  ...
  (link ...
    [(OPTIONS : setup-option^) setup:option@]
    [() my-init-options@ OPTIONS]
    [() setup@ OPTIONS ...])
  ...)
```

#<part-start>

```
(require setup/setup-unit)
```

---

setup@ : unit?

Imports

- setup-option^
- compiler^
- compiler:option^
- launcher^

and exports nothing. Invoking setup@ starts the setup process.

2.1 Options Unit

```
(require setup/option-unit)
```

---

setup:option@ : unit?

Imports nothing and exports `setup-option^`.

2.2 Options Signature

```
(require setup/option-sig)
```

`setup-option^` : signature

Provides parameters used to control `setup-plt` in unit form.

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

If on, prints message from `make` to `stderr`. The default is `#f`.

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

If on, verbose `make`. The default is `#f`.

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

If on, verbose `compiler`. The default is `#f`.

```
(clean) → boolean?
(clean on?) → void?
  on? : any/c
```

If on, delete `".zo"` and `".so"/".dll"/".dylib"` files in the specified collections. The default is `#f`.

```
(compile-mode) → (or/c path? false/c)
(compile-mode path) → void?
  path : (or/c path? false/c)
```

If a *path* is given, use a `".zo"` compiler other than plain `compile`, and build to `(build-path "compiled" (compile-mode))`. The default is `#f`.

```
(make-zo) → boolean?
(make-zo on?) → void?
  on? : any/c
```

If on, compile ".zo". The default is #t.

```
(make-so) → boolean?
(make-so on?) → void?
  on? : any/c
```

If on, compile ".so"/".dll" files. The default is #f.

```
(make-launchers) → boolean?
(make-launchers on?) → void?
  on? : any/c
```

If on, make collection "info.ss"-specified launchers. The default is #t.

```
(make-info-domain) → boolean?
(make-info-domain on?) → void?
  on? : any/c
```

If on, update "info-domain/compiled/cache.ss" for each collection path. The default is #t.

```
(call-install) → boolean?
(call-install on?) → void?
  on? : any/c
```

If on, call collection "info.ss"-specified setup code. The default is #t.

```
(force-unpack) → boolean?
(force-unpack on?) → void?
  on? : any/c
```

If on, ignore version and already-installed errors when unpacking a ".plt" archive. The default is #f.

```
(pause-on-errors) → boolean?
(pause-on-errors on?) → void?
  on? : any/c
```

If on, in the event of an error, prints a summary error and waits for stdin input before terminating. The default is #f.

```
(specific-collections) → (listof path-string?)
(specific-collections coll) → void?
  coll : (listof path-string?)
```

A list of collections to set up; the empty list means set-up all collections if the archives list is also empty The default is `null`.

```
(archives) → (listof path-string?)
(archives arch) → void?
  arch : (listof path-string?)
```

A list of `".plt"` archives to unpack; any collections specified by the archives are set-up in addition to the collections listed in specific-collections. The default is `null`.

```
(current-target-directory-getter) → (-> . path-string?)
(current-target-directory-getter thunk) → void?
  thunk : (-> . path-string?)
```

A thunk that returns the target directory for unpacking a relative `".plt"` archive; when unpacking an archive, either this or the procedure in `current-target-plt-directory-getter` will be called. The default is `current-directory`.

```
(current-target-plt-directory-getter)
 → (path-string?
    path-string?
    (listof path-string?) . -> . path-string?)
(current-target-plt-directory-getter proc) → void?
  proc : (path-string?
          path-string?
          (listof path-string?) . -> . path-string?)
```

A procedure that takes a preferred path, a path to the parent of the main `"collects"` directory, and a list of path choices; it returns a path for a ”plt-relative” install; when unpacking an archive, either this or the procedure in `current-target-directory-getter` will be called, and in the former case, this procedure may be called multiple times. The default is `(lambda (preferred main-parent-dir choices) preferred)`.

3 ".plt" Archives

3.1 Making ".plt" Archives

```
(require setup/pack)
```

Although the mzc executable can be used to create ".plt" files (see §"mzc: PLT Compilation and Packaging"), the setup/pack library provides a more general Scheme API for making ".plt" archives:

```
(pack-collections-plt
 dest
 name
 collections
 [#:replace? replace?
 #:at-plt-home? at-home?
 #:test-plt-collects? test?
 #:extra-setup-collections collection-list
 #:file-filter filter-proc])
→ void?
 dest : path-string?
 name : string?
 collections : (listof (listof path-string?))
 replace? : boolean? = #f
 at-home? : boolean? = #f
 test? : boolean? = #t
 collection-list : (listof path-string?) = null
 filter-proc : (path-string? . -> . boolean?) = std-filter
```

Creates the ".plt" file specified by the pathname *dest*, using the *name* as the name reported to setup-plt as the archive's description.

The archive contains the collections listed in *collections*, which should be a list of collection paths; each collection path is, in turn, a list of relative-path strings.

If the #:replace? argument is #f, then attempting to unpack the archive will report an error when any of the collections exist already, otherwise unpacking the archive will overwrite an existing collection.

If the #:at-plt-home? argument is #t, then the archived collections will be installed into the PLT installation directory instead of the user's directory if the main "collects" directory is writable by the user. If the #:test-plt-collects? argument is #f (the default is #t) and the #:at-plt-home? argument is #t, then installation fails if the main "collects" directory is not writable.

The optional `#:extra-setup-collections` argument is a list of collection paths that are not included in the archive, but are set-up when the archive is unpacked.

The optional `#:file-filter` argument is the same as for `pack-plt`.

```
(pack-collections  dest
                   name
                   collections
                   replace?
                   extra-setup-collections
                  [filter
                   at-plt-home?])          → void?
  dest : path-string?
  name : string?
  collections : (listof (listof path-string?))
  replace? : boolean?
  extra-setup-collections : (listof path-string?)
  filter : (path-string? . -> . boolean?) = std-filter
  at-plt-home? : boolean? = #f
```

Old, keywordless variant of `pack-collections-plt` for backward compatibility.

```
(pack-plt  dest
           name
           paths
          [#:file-filter filter-proc
           #:encode? encode?
           #:file-mode file-mode-sym
           #:unpack-unit unit200-expr
           #:collections collection-list
           #:plt-relative? plt-relative?
           #:at-plt-home? at-plt-home?
           #:test-plt-dirs dirs
           #:requires mod-and-version-list
           #:conflicts mod-list])          → void?
  dest : path-string?
  name : string?
  paths : (listof path-string?)
  filter-proc : (path-string? . -> . boolean?) = std-filter
  encode? : boolean? = #t
  file-mode-sym : symbol? = 'file
  unit200-expr : any/c = #f
  collection-list : (listof path-string?) = null
  plt-relative? : any/c = #f
  at-plt-home? : any/c = #f
```

```
dirs : (or/c (listof path-string?) false/c) = #f
mod-and-version-list : (listof (listof path-string?)    = null
                               (listof exact-integer?))
mod-list : (listof (listof path-string?)) = null
```

Creates the `".plt"` file specified by the pathname *dest*, using the string *name* as the name reported to `setup-plt` as the archive's description. The *paths* argument must be a list of relative paths for directories and files; the contents of these files and directories will be packed into the archive.

The `#:file-filter` procedure is called with the relative path of each candidate for packing. If it returns `#f` for some path, then that file or directory is omitted from the archive. If it returns `'file` or `'file-replace` for a file, the file is packed with that mode, rather than the default mode. The default is `std-filter`.

If the `#:encode?` argument is `#f`, then the output archive is in raw form, and still must be gzipped and mime-encoded (in that order). The default value is `#t`.

The `#:file-mode` argument must be `'file` or `'file-replace`, indicating the default mode for a file in the archive. The default is `'file`.

The `#:unpack-unit` argument is usually `#f`. Otherwise, it must be an S-expression for a `mzlib/unit200`-style unit that performs the work of unpacking; see §3.4 "Format of `".plt"` Archives" more information about the unit. If the `#:unpack-unit` argument is `#f`, an appropriate unpacking unit is generated.

The `#:collections` argument is a list of collection paths to be compiled after the archive is unpacked. The default is the `null`.

If the `#:plt-relative?` argument is true (the default is `#f`), the archive's files and directories are to be unpacked relative to the user's add-ons directory or the PLT installation directories, depending on whether the `#:at-plt-home?` argument is true and whether directories specified by `#:test-plt-dirs` are writable by the user.

If the `#:at-plt-home?` argument is true (the default is `#f`), then `#:plt-relative?` must be true, and the archive is unpacked relative to the PLT installation directory. In that case, a relative path that starts with `"collects"` is mapped to the installation's main `"collects"` directory, and so on, for the following the initial directory names:

- `"collects"`
- `"doc"`
- `"lib"`
- `"include"`

If `#:test-plt-dirs` is a `list`, then `#:at-plt-home?` must be `#t`. In that case, when the archive is unpacked, if any of the relative directories in the `#:test-plt-dirs` list is unwritable by the current user, then the archive is unpacked in the user's add-ons directory after all.

The `#:requires` argument should have the shape `(list (list coll-path version) ...)` where each *coll-path* is a non-empty list of relative-path strings, and each *version* is a (possibly empty) list of exact integers. The indicated collections must be installed at unpacking time, with version sequences that match as much of the version sequence specified in the corresponding *version*. A collection's version is indicated by the `version` field of its `"info.ss"` file.

The `#:conflicts` argument should have the shape `(list coll-path ...)` where each *coll-path* is a non-empty list of relative-path strings. The indicated collections must *not* be installed at unpacking time.

```
(pack  dest
       name
       paths
       collections
      [filter
       encode?
       file-mode
       unpack-unit
       plt-relative?
       requires
       conflicts
       at-plt-home?]) → void?
  dest : path-string?
  name : string?
  paths : (listof path-string?)
  collections : (listof path-string?)
  filter : (path-string? . -> . boolean?) = std-filter
  encode? : boolean? = #t
  file-mode : symbol? = 'file
  unpack-unit : boolean? = #f
  plt-relative? : boolean? = #t
  requires : (listof (listof path-string?)    = null
                     (listof exact-integer?))
  conflicts : (listof (listof path-string?)) = null
  at-plt-home? : boolean? = #f
```

Old, keywordless variant of `pack-plt` for backward compatibility.

```
(std-filter p) → boolean?
```

```
  p : path-string?
```

Returns #t unless *p*, after stripping its directory path and converting to a byte string, matches one of the following regular expressions: ^CVS$, ^[.]svn$, ^[.]cvsignore, ^compiled$, ^doc, ~$, ^#.*#$, ^[.]#, or [.]plt$.

```
(mztar path output filter file-mode) → void?
  path : path-string?
  output : output-port?
  filter : (path-string? . -> . boolean?)
  file-mode : (symbols 'file 'file-replace)
```

Called by pack to write one directory/file *path* to the output port *output* using the filter procedure *filter* (see pack for a description of *filter*). The *file-mode* argument specifies the default mode for packing a file, either 'file or 'file-replace.

3.2 Installing a Single ".plt" File

The setup/plt-single-installer module provides a function for installing a single ".plt" file, and setup/plt-single-installer wraps it with a GUI interface.

3.2.1 Non-GUI Installer

```
(require setup/plt-single-installer)
```

```
(run-single-installer file get-dir-proc) → void?
  file : path-string?
  get-dir-proc : (-> (or/c path-string? false/c))
```

Creates a separate thread and namespace, runs the installer in that thread with the new namespace, and returns when the thread completes or dies. It also creates a custodian (see §13.6 "Custodians") to manage the created thread, sets the exit handler for the thread to shut down the custodian, and explicitly shuts down the custodian when the created thread terminates or dies.

The *get-dir-proc* procedure is called if the installer needs a target directory for installation, and a #f result means that the user canceled the installation. Typically, *get-dir-proc* is current-directory.

3.2.2 GUI Installer

```
(require setup/plt-installer)
```

The "setup/plt-installer" library in the setup collection defines procedures for installing a ".plt" archive with a GUI (using the facilities of scheme/gui/base).

```
(run-installer filename) → void?
  filename : path-string?
```

Run the installer on the ".plt" file in *filename* and show the output in a window. This is a composition of with-installer-window and run-single-installer with a get-dir-proc that prompts the user for a directory (turning off the busy cursor while the dialog is active).

```
(on-installer-run) → (-> any)
(on-installer-run thunk) → void?
  thunk : (-> any)
```

A thunk that is run after a ".plt" file is installed.

```
(with-installer-window do-install
                       cleanup-thunk) → void?
  do-install : ((or/c (is-a?/c dialog%) (is-a?/c frame%))
                . -> . void?)
  cleanup-thunk : (-> any)
```

Creates a frame, sets up the current error and output ports, and turns on the busy cursor before calling *do-install* in a separate thread.

Returns before the installation process is complete; *cleanup-thunk* is called on a queued callback to the eventspace active when with-installer-window is invoked.

```
(run-single-installer file get-dir-proc) → void?
  file : path-string?
  get-dir-proc : (-> (or/c path-string? false/c))
```

The same as the sole export of setup/plt-single-installer, but with a GUI.

3.2.3 GUI Unpacking Signature

```
(require setup/plt-installer-sig)
```

`setup:plt-installer^` : signature

Provides two names: `run-installer` and `on-installer-run`.

3.2.4 GUI Unpacking Unit

```
(require setup/plt-installer-unit)
```

Imports `mred^` and exports `setup:plt-installer^`.

3.3 Unpacking `".plt"` Archives

```
(require setup/unpack)
```

The `setup/unpack` library provides raw support for unpacking a `".plt"` file.

```
(unpack  archive
        [main-collects-parent-dir
         print-status
         get-target-directory
         force?
         get-target-plt-directory]) → void?
  archive : path-string?
  main-collects-parent-dir : path-string? = (current-directory)
  print-status : (string? . -> . any)
               = (lambda (x) (printf "~a\n" x))
  get-target-directory : (-> path-string?)
                       = (lambda () (current-directory))
  force? : any/c = #f
  get-target-plt-directory : (path-string?
                              path-string?
                              (listof path-string?)
                              . -> . path-string?)
                           = (lambda (preferred-dir main-dir options)
                               preferred-dir)
```

Unpacks *archive*.

The *main-collects-parent-dir* argument is passed along to *get-target-plt-directory*.

The *print-status* argument is used to report unpacking progress.

The *get-target-directory* argument is used to get the destination directory for unpacking an archive whose content is relative to an arbitrary directory.

If *force?* is true, then version and required-collection mismatches (comparing information in the archive to the current installation) are ignored.

The *get-target-plt-directory* function is called to select a target for installation for an archive whose is relative to the installation. The function should normally return one if its first two arguments; the third argument merely contains the first two, but has only one element if the first two are the same. If the archive does not request installation for all uses, then the first two arguments will be different, and the former will be a user-specific location, while the second will refer to the main installation.

```
(fold-plt-archive archive
                  on-config-fn
                  on-setup-unit
                  on-directory
                  on-file
                  initial-value) → any/c
  archive : path-string?
  on-config-fn : (any/c any/c . -> . any/c)
  on-setup-unit : (any/c input-port? any/c . -> . any/c)
  on-directory : (path-string? any/c . -> . any/c)
  on-file : (path-string? input-port? any/c . -> . any/c)
  initial-value : any/c
```

Traverses the content of *archive*, which must be a `".plt"` archive that is created with the default unpacking unit and configuration expression. The configuration expression is not evaluated, the unpacking unit is not invoked, and not files are unpacked to the filesystem. Instead, the information in the archive is reported back through `on-config`, *on-setup-unit*, *on-directory*, and *on-file*, each of which can build on an accumulated value that starts with *initial-value* and whose final value is returned.

The *on-config-fn* function is called once with an S-expression that represents a function to implement configuration information. The second argument to `on-config` is *initial-value*, and the function's result is passes on as the last argument to *on-setup-unit*.

The *on-setup-unit* function is called with the S-expression representation of the installation unit, an input port that points to the rest of the file, and the accumulated value. This input port is the same port that will be used in the rest of processing, so if *on-setup-unit* consumes any data from the port, then that data will not be consumed by the remaining functions. (This means that on-setup-unit can leave processing in an inconsistent state, which is not checked by anything, and therefore could cause an error.) The result of *on-setup-unit* becomes the new accumulated value.

For each directory that would be created by the archive when unpacking normally, *on-*

directory is called with the directory path and the accumulated value up to that point, and its result is the new accumulated value.

For each file that would be created by the archive when unpacking normally, *on-file* is called with the file path, an input port containing the contents of the file, and the accumulated value up to that point; its result is the new accumulated value. The input port can be used or ignored, and parsing of the rest of the file continues the same either way. After *on-file* returns control, however, the input port is drained of its content.

3.4 Format of ".plt" Archives

The extension ".plt" is not required for a distribution archive, but the ".plt"-extension convention helps users identify the purpose of a distribution file.

The raw format of a distribution file is described below. This format is uncompressed and sensitive to communication modes (text vs. binary), so the distribution format is derived from the raw format by first compressing the file using gzip, then encoding the gzipped file with the MIME base64 standard (which relies only the characters A-Z, a-z, 0-9, +, /, and =; all other characters are ignored when a base64-encoded file is decoded).

The raw format is

- PLT are the first three characters.
- A procedure that takes a symbol and a failure thunk and returns information about archive for recognized symbols and calls the failure thunk for unrecognized symbols. The information symbols are:
  - 'name — a human-readable string describing the archive's contents. This name is used only for printing messages to the user during unpacking.
  - 'unpacker — a symbol indicating the expected unpacking environment. Currently, the only allowed value is 'mzscheme.
  - 'requires — collections required to be installed before unpacking the archive, which associated versions; see the documentation of pack for details.
  - 'conflicts — collections required *not* to be installed before unpacking the archive.
  - 'plt-relative? — a boolean; if true, then the archive's content should be unpacked relative to the plt add-ons directory.
  - 'plt-home-relative? — a boolean; if true and if 'plt-relative? is true, then the archive's content should be unpacked relative to the PLT Scheme installation.
  - 'test-plt-dirs — #f or a list of path strings; in the latter case, a true value of 'plt-home-relative? is cancelled if any of the directories in the list (relative to the PLT Scheme installation) is unwritable by the user.

The procedure is extracted from the archive using the `read` and `eval` procedures in a fresh namespace.

- An old-style, unsigned unit using `(lib mzlib/unit200)` that drives the unpacking process. The unit accepts two imports: a path string for the parent of the main `"collects"` directory and an `unmztar` procedure. The remainder of the unpacking process consists of invoking this unit. It is expected that the unit will call `unmztar` procedure to unpack directories and files that are defined in the input archive after this unit. The result of invoking the unit must be a list of collection paths (where each collection path is a list of strings); once the archive is unpacked, `setup-plt` will compile and setup the specified collections.

  The `unmztar` procedure takes one argument: a filter procedure. The filter procedure is called for each directory and file to be unpacked. It is called with three arguments:

  - `'dir`, `'file`, `'file-replace` — indicates whether the item to be unpacked is a directory, a file, or a file to be replaced,
  - a relative path string — the pathname of the directory or file to be unpacked, relative to the unpack directory, and
  - a path string for the unpack directory (which can vary for a PLT-relative install when elements of the archive start with `"collects"`, `"lib"`, etc.).

  If the filter procedure returns `#f` for a directory or file, the directory or file is not unpacked. If the filter procedure returns `#t` and the directory or file for `'dir` or `'file` already exists, it is not created. (The file for `file-replace` need not exist already.)

  When a directory is unpacked, intermediate directories are created as necessary to create the specified directory. When a file is unpacked, the directory must already exist.

  The unit is extracted from the archive using `read` and `eval`.

Assuming that the unpacking unit calls the `unmztar` procedure, the archive should continue with unpackables. Unpackables are extracted until the end-of-file is found (as indicated by an `=` in the base64-encoded input archive).

An *unpackable* is one of the following:

- The symbol `'dir` followed by a list. The `build-path` procedure will be applied to the list to obtain a relative path for the directory (and the relative path is combined with the target directory path to get a complete path).

  The `'dir` symbol and list are extracted from the archive using `read` (and the result is *not* `eval`uated).

- The symbol `'file`, a list, a number, an asterisk, and the file data. The list specifies the file's relative path, just as for directories. The number indicates the size of the file to

be unpacked in bytes. The asterisk indicates the start of the file data; the next n bytes are written to the file, where n is the specified size of the file.

The symbol, list, and number are all extracted from the archive using `read` (and the result is *not* `eval`uated). After the number is read, input characters are discarded until an asterisk is found. The file data must follow this asterisk immediately.

- The symbol `'file-replace` is treated like `'file`, but if the file exists on disk already, the file in the archive replaces the file on disk.

4 Finding Installation Directories

```
(require setup/dirs)
```

The setup/dirs library provides several procedures for locating installation directories:

(find-collects-dir) → (or/c path? false/c)

Returns a path to the installation's main "collects" directory, or #f if none can be found. A #f result is likely only in a stand-alone executable that is distributed without libraries.

(find-user-collects-dir) → path?

Returns a path to the user-specific "collects" directory; the directory indicated by the returned path may or may not exist.

(get-collects-search-dirs) → (listof path?)

Returns the same result as (current-library-collection-paths), which means that this result is not sensitive to the value of the use-user-specific-search-paths parameter.

(find-doc-dir) → (or/c path? false/c)

Returns a path to the installation's "doc" directory. The result is #f if no such directory is available.

(find-user-doc-dir) → path?

Returns a path to a user-specific "doc" directory. The directory indicated by the returned path may or may not exist.

(get-doc-search-dirs) → (listof path?)

Returns a list of paths to search for documentation, not including documentation stored in individual collections. Unless it is configured otherwise, the result includes any non-#f result of (find-doc-dir) and (find-user-doc-dir)—but the latter is included only if the value of the use-user-specific-search-paths parameter is #t.

(find-lib-dir) → (or/c path? false/c)

Returns a path to the installation's "lib" directory, which contains libraries and other build information. The result is #f if no such directory is available.

(find-dll-dir) → (or/c path? false/c)

Returns a path to the directory that contains DLLs for use with the current executable (e.g., "libmzsch.dll" under Windows). The result is #f if no such directory is available, or if no specific directory is available (i.e., other than the platform's normal search path).

(find-user-lib-dir) → path?

Returns a path to a user-specific "lib" directory; the directory indicated by the returned path may or may not exist.

(get-lib-search-dirs) → (listof path?)

Returns a list of paths to search for libraries. Unless it is configured otherwise, the result includes any non-#f result of (find-lib-dir), (find-dll-dir), and (find-user-lib-dir)—but the last is included only if the value of the use-user-specific-search-paths parameter is #t.

(find-include-dir) → (or/c path? false/c)

Returns a path to the installation's "include" directory, which contains ".h" files for building MzScheme extensions and embedding programs. The result is #f if no such directory is available.

(find-user-include-dir) → path?

Returns a path to a user-specific "include" directory; the directory indicated by the returned path may or may not exist.

(get-include-search-dirs) → (listof path?)

Returns a list of paths to search for ".h" files. Unless it is configured otherwise, the result includes any non-#f result of (find-include-dir) and (find-user-include-dir)—but the latter is included only if the value of the use-user-specific-search-paths parameter is #t.

(find-console-bin-dir) → (or/c path? false/c)

Returns a path to the installation's executable directory, where the stand-alone MzScheme executable resides. The result is `#f` if no such directory is available.

`(find-gui-bin-dir) → (or/c path? false/c)`

Returns a path to the installation's executable directory, where the stand-alone MrEd executable resides. The result is `#f` if no such directory is available.

`absolute-installation? : boolean?`

A binary boolean flag that is true if this installation is using absolute path names.

5 "info.ss" File Format

```
#lang setup/infotab
```

In each collection, a special module file "info.ss" provides general information about a collection for use by various tools. For example, an "info.ss" file specifies how to build the documentation for a collection, and it lists plug-in tools for DrScheme that the collection provides.

Although an "info.ss" file contains a module declaration, the declaration has a highly constrained form. It must match the following grammar of *info-module*:

```
       info-module = (module info intotab-mod-path
                       (define id info-expr)
                       ...)

  intotab-mod-path = (lib "infotab.ss" "setup")
                   | setup/infotab

         info-expr = 'datum
                   | `datum
                   | (info-primitive info-expr ...)
                   | id
                   | string
                   | number
                   | boolean
                   | (string-constant identifier)

    info-primitive = cons
                   | car
                   | cdr
                   | list
                   | list*
                   | reverse
                   | append
                   | string-append
                   | path->string
                   | build-path
                   | collection-path
                   | system-library-subpath
```

For example, the following declaration could be the "info.ss" library of the "help" collection. It contains definitions for three info tags, name, mzscheme-launcher-libraries, and mzscheme-launcher-names.

```
#lang setup/infotab
```

```
(define name "Help")
(define mzscheme-launcher-libraries '("help.ss"))
(define mzscheme-launcher-names     '("PLT Help"))
```

As illustrated in this example, an "info.ss" file can use #lang notation, but only with the setup/infotab language.

See also get-info from setup/getinfo.

6 Reading "info.ss" Files

```
(require setup/getinfo)
```

The setup/getinfo library provides functions for accessing fields in "info.ss" files.

```
(get-info collection-names) → (or/c
                                (symbol? [(-> any)] . -> . any)
                                false/c)
  collection-names : (listof string?)
```

Accepts a list of strings naming a collection or sub-collection, and calls get-info/full with the full path corresponding to the named collection.

```
(get-info/full path) → (or/c
                         (symbol? [(-> any)] . -> . any)
                         false/c)
  path : path?
```

Accepts a path to a directory. It returns #f if there is no "info.ss" file in the directory. If the "info.ss" file has the wrong shape (i.e., not a module using setup/infotab or (lib "infotab.ss" "setup")), or if the "info.ss" file fails to load, then an exception is raised.

Otherwise, get-info/full returns an info procedure of one or two arguments. The first argument to the info procedure is always a symbolic name, and the result is the value of the name in the "info.ss" file, if the name is defined. The optional second argument, *thunk*, is a procedure that takes no arguments to be called when the name is not defined; the result of the info procedure is the result of the *thunk* in that case. If the name is not defined and no *thunk* is provided, then an exception is raised.

```
(find-relevant-directories syms [mode]) → (listof path?)
  syms : (listof symbol?)
  mode : (symbols 'preferred 'all-available) = 'preferred
```

Returns a list of paths identifying installed directories (i.e., collections and installed PLaneT packages) whose "info.ss" file defines one or more of the given symbols. The result is based on a cache that is computed by setup-plt and stored in the "info-domain" sub-directory of each collection directory (as determined by the PLT_COLLECTION_PATHS environment variable, etc.) and the file "cache.ss" in the user add-on directory.

The result is in a canonical order (sorted lexicographically by directory name), and the paths it returns are suitable for providing to get-info/full.

If *mode* is specified, it must be either `'preferred` (the default) or `'all-available`. If mode is `'all-available`, `find-relevant-collections` returns all installed directories whose info files contain the specified symbols—for instance, all installed PLaneT packages will be searched if `'all-available` is specified. If mode is `'preferred`, then only a subset of "preferred" packages will be searched, and in particular only the directory containing the most recent version of any PLaneT package will be returned.

No matter what *mode* is specified, if more than one collection has the same name, `find-relevant-directories` will only search the one that occurs first in the `PLT_COLLECTION_PATHS` environment variable.

`(reset-relevant-directories-state!)` → `void?`

Resets the cache used by `find-relevant-directories`.

7 Paths Relative to "collects"

```
(require setup/main-collects)
```

```
(path->main-collects-relative path)
 → (or/c path? (cons/c 'collects (listof bytes?)))
  path : (or bytes? path-string?)
```

Checks whether *path* has a prefix that matches the prefix to the main "collects" directory as determined by (find-collects-dir). If so, the result is a list starting with 'collects and containing the remaining path elements as byte strings. If not, the path is returned as-is.

The *path* argument should be a complete path. Applying simplify-path before path->main-collects-relative is usually a good idea.

For historical reasons, *path* can be a byte string, which is converted to a path using bytes->path.

```
(main-collects-relative->path rel) → path?
  rel : (or/c path?
              (cons/c 'collects
                      (or/c (listof bytes?) bytes?)))
```

The inverse of path->main-collects-relative: if *rel* is a pair that starts with 'collects, then it is converted back to a path relative to (find-collects-dir).

For historical reasons, a single byte string is allowed in place of a list of byte strings after 'collects, in which case it is assumed to be a relative path after conversion with bytes->path.

Also for historical reasons, if *rel* is any kind of value other than specified in the contract above, it is returned as-is.

8 Cross-References for Installed Manuals

```
(require setup/xref)
```

```
(load-collections-xref [on-load]) → xref?
  on-load : (-> any/c) = (lambda () (void))
```

Like `load-xref`, but automatically find all cross-reference files for manuals that have been installed with `setup-plt`.

Index

".plt" Archives, 12
"info.ss" File Format, 26
absolute-installation?
archives, 11
call-install
clean, 9
compile-mode, 9
compiler-verbose, 9
Controlling setup-plt with "info.ss" Files, 3
Cross-References for Installed Manuals, 31
current-target-directory-getter, 11
current-target-plt-directory-getter, 11
find-collects-dir
find-console-bin-dir, 24
find-dll-dir, 24
find-doc-dir, 23
find-gui-bin-dir, 25
find-include-dir, 24
find-lib-dir, 23
find-relevant-directories, 28
find-user-collects-dir, 23
find-user-doc-dir, 23
find-user-include-dir, 24
find-user-lib-dir, 24
Finding Installation Directories, 23
fold-plt-archive, 19
force-unpack, 10
Format of ".plt" Archives, 20
get-collects-search-dirs
get-doc-search-dirs, 23
get-include-search-dirs, 24
get-info, 28
get-info/full, 28
get-lib-search-dirs, 24
GUI Installer, 16
GUI Unpacking Signature, 17
GUI Unpacking Unit, 18
"info-domain"
Installing a Single ".plt" File, 16
load-collections-xref
main-collects-relative->path
make-info-domain, 10
make-launchers, 10
make-so, 10
make-verbose, 9
make-zo, 10
Making ".plt" Archives, 12
mztar, 16
Non-GUI Installer
on-installer-run
Options Signature, 9
Options Unit, 8
pack
pack-collections, 13
pack-collections-plt, 12
pack-plt, 13
path->main-collects-relative, 30
Paths Relative to "collects", 30
pause-on-errors, 10
Reading "info.ss" Files
reset-relevant-directories-state!, 29
run-installer, 17
run-single-installer, 16
run-single-installer, 17
Running setup-plt Executable, 3
Running setup-plt from Scheme, 8
scribblings
setup-option^, 9
setup-plt: PLT Configuration and Installation, 1
setup/dirs, 23
setup/getinfo, 28
setup/infotab, 26
setup/main-collects, 30
setup/option-sig, 9
setup/option-unit, 8
setup/pack, 12
setup/plt-installer, 16
setup/plt-installer-sig, 17

setup/plt-installer-unit, 18
setup/plt-single-installer, 16
setup/setup-unit, 8
setup/unpack, 18
setup/xref, 31
setup:option@, 8
setup:plt-installer^, 18
setup@, 8
specific-collections, 11
std-filter, 15
unpack
unpackable, 21
Unpacking ".plt" Archives, 18
verbose
with-installer-window