The SDF Software Manager

2019-07-14

Chris Hanson
Gerald Jay Sussman

This manual is for the SDF Software Manager.

1 Introduction

This manual documents the software management program included with the software for the book *Software Design for Flexibility*. The manager is a tool to help one use the associated software. For example, it simplifies the job of loading the software needed to reproduce the examples in the book and to support the solution of the book's exercises. The book contains an appendix *Supporting Software* that explains where the software can be obtained and an overview of how it is used.

Both the management software and the book's software is designed to be run using the MIT/GNU Scheme (`http://www.gnu.org/software/mit-scheme`) implementation, version 10.1.9 or later. The Scheme system must be separately downloaded and installed.

To use the manager, start up the Scheme system and evaluate the following expression:

```
(load "foo/sdf/manager/load")
```

where *foo/* is the path to the directory in which the software archive was unpacked. The manager creates a single definition in the global environment, called `manage`. Once loaded, it's not necessary to reload the manager unless a new instance of Scheme is started.

`manage` *command arg ...* [Procedure]

This procedure is the entry point for all of the operations that can be performed by the manager. The *command* argument is a symbol that specifies the operation to be performed. The *arg*s, if any, are command-specific and documented for each command in the following sections.

As a convenience, *command* can be a string (or symbol) that is a substring of the desired command's name. If so, the manager finds all of the command names that contain the given substring; if there is exactly one, the manager runs that command. Otherwise, it prints an error message.

The description of each command is of the form:

`command-name args` [Manager Command]

where *command-name* is the name of the command (a symbol), and *args* are the arguments required by the command. Any *args* surrounded by square brackets (‘`[]`’) are optional, while a trailing ... indicates that any number of arguments may be provided. To invoke a command, call the `manage` procedure like this:

```
(manage 'new-environment 'checkers-new)
```

This specifies the `new-environment` command and provides it with a single argument, the symbol `checkers-new`.

The most important command is `help`:

`help` [Manager Command]

The `help` command prints out all of the supported commands and brief documentation for each.

Usage: ‘`(manage 'help)`’.

Another useful command is `command-apropos`:

`command-apropos` *string* [Manager command]

Returns a list of the command names containing STRING. This is useful for finding a minimal string that specifies a unique command, as well as finding related command names.

2 Working Environments

The primary purpose of the manager is to create working environments that contain the code necessary to explore a specific section of the book. The book's software is organized into *flavors*, which roughly correspond to individual sections of the book. The specific relationships between the book sections and the flavors are spelled out in the book's *Supporting Software* appendix.

A *working environment* as defined here is a Scheme environment that inherits all of Scheme's global definitions and into which all of the necessary program code has been loaded. To use a working environment, one "selects" it by changing the Scheme REPL (Read-Eval-Print Loop) to use that environment. After doing so, any expressions evaluated by the REPL will be interpreted in that environment. For convenience, `manage` provides high-level commands for switching between working environments, so it's not necessary to know the details about how to change the environment of the REPL.

When we say that a "flavor is loaded", that means the files for the flavor are retrieved from the file system and loaded into the environment. If you have made changes to those files, your changes will be visible in the environment.

As with specification of command names, a flavor name passed as an argument to one of these commands may be specified by a substring that matches exactly one flavor name.

The most basic working environment command is `new-environment`.

`new-environment` *flavor ...* [Manager Command]

This command creates a new working environment, loads all of the software specified by the *flavor* arguments, and changes the REPL to be in the new environment. The previous environment, if any, is dropped and will be inaccessible unless it has been given a name.

Examples: '`(manage 'new-environment)`' creates a new working environment without loading any software. '`(manage 'new-environment 'combinators)`' creates a new working environment and loads the software for flavor `combinators`.

Note to advanced users: if the REPL is nested, such as that resulting from an error, this command will abort the nested REPL and return to the top-level REPL.

`add-flavor` *flavor* [Manager Command]

This command modifies the current working environment by loading the software for *flavor* into it. `add-flavor` is useful when some work has been done in a working environment and one decides that another section provides useful functionality for continuing the work. If instead the flavors to be used are known in advance, the `new-environment` command is sufficient.

Some examples:

```
(manage 'new-environment)
```

```
(manage 'add-flavor flavor1)
;; is equivalent to:
(manage 'new-environment flavor1)

;; Likewise
(manage 'new-environment flavor1)
(manage 'add-flavor flavor2)
;; is equivalent to:
(manage 'new-environment flavor1 flavor2)
```

It might be useful to know what flavors are available to be used:

`list-flavors` [Manager Command]
: Returns a list of the known flavors.

`flavor-apropos` *string* [Manager command]
: Returns a list of the flavor names containing STRING. This is useful for finding a minimal string that specifies a unique flavor, as well as finding related flavor names.

By giving names to working environments, we can have more than one and switch among them. The following commands manage named environments. The *name* arguments that they accept can be any object, but they must be comparable using the `eqv?` procedures. Consequently the most useful names are things like symbols or numbers, and *not* strings or lists.

`name-current-environment` *name* [Manager Command]
: Gives the current working environment the name *name*. The *name* must not be a known working environment name.

As with specification of command names and flavor names, an environment name passed as an argument to one of the following commands may be specified by a substring that matches exactly one environment name.

`use-environment` *name* [Manager Command]
: Selects the environment named *name* by changing the REPL to be in that environment.

  Note to advanced users: if the REPL is nested, such as that resulting from an error, this command will abort the nested REPL and return to the top-level REPL.

`remove-environment-name` *name* [Manager Command]
: Removes the name *name* from whatever working environment it identifies. The *name* must be a known environment name. If the named environment has no other names and is not the current working environment, then it becomes inaccessible and will be garbage collected.

`environment-names` [Manager Command]
: Returns a list of the known environment names.

`environment-apropos` *string* [Manager command]
: Returns a list of the environment names containing STRING. This is useful for finding a minimal string that specifies a unique environment, as well as finding related environment names.

3 Analyzing the Software

Reading software is complicated, especially when it is spread over many files. Professional programmers often use tools like `grep` and Integrated Development Environments to help them understand how the parts are connected to one another.

The software manager maintains an index of definitions and references showing where a symbol is given a definition and where it is used. The commands in this section can be used to explore that index. The *name* arguments to these commands must be symbols. The *filename* arguments must be filenames that refer to specific files in the tree.

`defining-files` *name* [Manager Command]
: Returns a list of the filenames that contain a definition of *name*. There may be more than one such file, but those files are rarely loaded together.

`referring-files` *name* [Manager Command]
: Returns a list of the filenames that contain one or more free references to *name*.

`defined-in-file` *filename* [Manager Command]
: Returns a list of the symbols that are defined in the file specified by `filename`.

`references-in-file` *filename* [Manager Command]
: Returns a list of the symbols that are free references in the file specified by `filename`.

`check-file-analysis` [Manager Command]
: Checks the analyzed-file index for potential problems. For example, some names are defined in multiple files, which may or may not be a problem.

`refresh-file-analysis` [Manager Command]
: This command rebuilds the index used by the other commands in this section. It is not needed unless changes have been made to the software tree. `refresh-file-analysis` takes a while to run.

4 Running Tests

All of the book's software contains tests that checks that its behavior is as expected. The manager provides several commands to run those tests. Generally these commands do not affect the current working environment nor any named working environment.

`run-tests` [Manager Command]
: Runs all tests for the current working environment. This is all of the tests for all of the loaded flavors. The working environment is not modified by the tests, but the current contents of the working environment, including all changes made since loading, will be used when running the tests.

`run-all-tests` *flavor ...* [Manager Command]
: Runs the tests for the specified *flavor*s. If no *flavor*s are specified, runs the tests for all flavors.

  These tests are run in hermetic environments: for each flavor, a new working environment is created, that flavor is loaded into it, and that flavor's tests are run. Since the tests all run in independent environments, they don't affect one another, nor do they affect any other working environment.

`show-individual-tests` *boolean* [Manager Command]

Normally when running tests, the individual tests are not shown as they run. Instead only the test files are shown, along with summaries of the test results.

The ability to see the individual tests is managed by a flag this command sets to the value of *boolean*. If *boolean* is true, then individual tests are shown, otherwise they are not. After running this command, the flag remains set to *boolean* until it is changed by a subsequent invocation.

5 Miscellaneous

The remaining commands are for special situations and are best suited to experienced users of MIT/GNU Scheme.

`manager-environment` [Manager Command]

Returns the environment in which the software manager is loaded. This is useful for debugging or modifying the manager itself.

`working-environment` [Manager Command]

Returns the current working environment, which can then be manipulated using MIT/GNU Scheme's environment procedures.

`debug-internal-errors` *boolean* [Manager Command]

Controls support for debugging the manager's internal errors. If `boolean` is false (the default), any error in the manager is caught, its message printed out, and then `manage` returns normally. If `boolean` is true, an internal error throws an exception that stops the manager's evaluation and enters a nested REPL in which the error can be debugged.

This command changes an internal flag in the manager, which remains set to *boolean* until a subsequent use of the command.

`load-test-only-files` [Manager Command]

Loads the test-only files into the current working environment. Useful for debugging tests.

Command Index