PLT Tools: DrScheme Extension Manual

PLT
scheme@cs.rice.edu

Version 103
August 2000

Rice University

Copyright notice

Send us your Web links

If you use any parts or all of the DrScheme package (software, lecture notes) for one of your courses, for your research, or for your work, we would like to know about it. Furthermore, if you use it and publicize the fact on some Web page, we would like to link to that page. Please drop us a line at *scheme@cs.rice.edu*. Evidence of interest helps the DrScheme Project to maintain the necessary intellectual and financial support. We appreciate your help.

Contents

1. Implementing DrScheme Tools

Tools are designed for major extensions in DrScheme's functionality. To extend DrScheme to extend the appearance or the functionality the DrScheme window (say, to annotate programs in certain ways, or to add buttons on the frame) use a tool. The Static Debugger and the Syntax Checker are implemented as tools.

Libraries are for extensions of DrScheme that only want to add new functions and other values bound in the users namespace. See the DrScheme manual for more information on constructing libraries.

Tools rely heavily on MzScheme's units. See units, §7 in *PLT MzScheme: Language Manual* for information on how to construct units. They also require understanding of libraries and collections, §15 in *PLT MzScheme: Language Manual*

When DrScheme starts up, it looks in the tools subdirectory of the **drscheme** collection directory to determine which tools are installed. For each subdirectory of the tools directory, it looks for two files: unit.ss and sig.ss. If sig.ss exists it is loaded when all of the signatures of DrScheme are loaded. The file unit.ss is required to exist. It must evaluate to a unit that imports 6 units matching the signatures:

- `mred^` (all of the names in the mred manual)
- `mzlib:core^`, (defined in the MzLib section of the MzScheme manual, §15 in *PLT MzScheme: Language Manual*),
- `framework^` (all of the names in the framework manual),
- `mzlib:print-convert^` (defined in the MzLib section of the MzScheme manual, §15 in *PLT MzScheme: Language Manual*),
- `drscheme:export^` (defined below), and
- `zodiac:system^` (all of the names in section 3.2).

The `drscheme:export^` signature contains the parameters defined in the parameters section2.11, and the other classes described in the next subsections.

If the tools raises an error as it is loaded or invoked, DrScheme catches the error and displays a message box. Then, DrScheme continues to start up, without the tool.

For example,

```
(unit/sig ()
  (import mred^
          mzlib:core^
          framework^
          [print-convert : mzlib:print-convert^]
          [drscheme : drscheme:export^]
          [zodiac : zodiac:system^])

  (message-box "tool example" "tool loaded"))
```

is a simple tool that opens a dialog as drscheme is started up.

1.1 Common Tools Abstractions

1.1.1 Evaluation

In order to evaluate programs that the user has implemented,

- The text of the program is in a `text%` object, available from the `definitions-text` instance variable of the `drscheme:unit:frame%` class.
- Use `drscheme:basis:process/zodiac` to process the text of the program.
- For evaluation, use the function `drscheme:basis:initialize-parameters`.
- Syntax errors are handled by raising a `exn:syntax` mz:exnsexception, see section of the MzScheme manual.

1.1.2 Getting the same menu items as DrScheme

In order to get frames that tools create and frames that DrScheme creates to have a common subset of menus, be sure to mixin `frame:standard-menus-mixin` and `drscheme:frame:basics-mixin`

1.1.3 Graphical Expressions

In order to add new kinds of graphical expressions to DrScheme, create a new snip that is the graphical representation of the value. If the snip is an expression that requires evaluation, implement Zodiac's `zodiac:expands<%>` interface. If the snip does not implement `zodiac:expands<%>`, it is treated as a value.

If the snip contains texts that have source text that may have errors, be sure to mix in `drscheme:unit:program-editor-mixin`.

2. Tools Reference

2.1 drscheme:frame:basics<%>

This interface is the result of the drscheme:frame:basics-mixin

2.2 drscheme:frame:basics-mixin

Domain: frame:standard-menus<%>

Implements: frame:standard-menus<%>

Implements: drscheme:frame:basics<%>

Use this mixin to establish some common menu items across various DrScheme windows.

file-menu:between-open-and-revert

This method is called between the addition of the open menu-item and before the addition of the revert menu-item to the file-menu menu. Override it to add additional menus at that point.

- (send *a-drscheme:frame:basics-mixin* file-menu:between-open-and-revert *file-menu*) ⇒ void
  file-menu : (instance menu%)

  Adds an *Open Url...* menu item, which invokes help desk's drscheme:help-desk:open-users-url function.

file-menu:new

This method is called when the new menu-item of the file-menu menu is selected. If file-menu:new is bound to #f instead of a procedure, the new menu item will not be created.

- (send *a-drscheme:frame:basics-mixin* file-menu:new *item evt*) ⇒ void
  item : (instance (derived-from menu-item%))
  evt : (instance control-event%)

  Opens a new empty drscheme window

file-menu:new-string

The result of this method is used to construct the name of this menu. It is inserted between "&New" and "" to form the complete name

- (send *a-drscheme:frame:basics-mixin* file-menu:new-string) ⇒ string

  Returns the empty string

file-menu:open

This method is called when the open menu-item of the file-menu menu is selected. If file-menu:open is bound to #f instead of a procedure, the open menu item will not be created.

- (send *a-drscheme:frame:basics-mixin* file-menu:open *item evt*) ⇒ void
  item : (instance (derived-from menu-item%))
  evt : (instance control-event%)

  Calls handler:open-file to open a new file. Note that there is a handler installed already that opens all files in DrScheme frames.

file-menu:open-string

The result of this method is used to construct the name of this menu. It is inserted between "&Open" and "..." to form the complete name

- (send *a-drscheme:frame:basics-mixin* file-menu:open-string) ⇒ string

  Returns the empty string

help-menu:about

This method is called when the about menu-item of the help-menu menu is selected. If help-menu:about is bound to #f instead of a procedure, the about menu item will not be created.

- (send *a-drscheme:frame:basics-mixin* help-menu:about *item evt*) ⇒ void
  item : (instance (derived-from menu-item%))
  evt : (instance control-event%)

  Opens an about box for DrScheme.

help-menu:about-string

The result of this method is used to construct the name of this menu. It is inserted between "About " and "..." to form the complete name

- (send *a-drscheme:frame:basics-mixin* help-menu:about-string) ⇒ string

  Returns the string "DrScheme".

help-menu:after-about

This method is called after the addition of the about menu-item to the help-menu menu. Override it to add additional menus at that point.

- (send *a-drscheme:frame:basics-mixin* help-menu:after-about *help-menu*) ⇒ void
  help-menu : (instance menu%)

  Adds the Help Desk menu item

2.3 drscheme:frame:name-message%

Superclass: canvas%

This class implements the little filename button in the top-right hand side of drscheme's frame.

- (make-object drscheme:frame:name-message% *parent*) ⇒ drscheme:frame:name-message% object
  parent : (instance area-container<%>)

set-message

Sets the names that the button shows.

- (send *a-drscheme:frame:name-message* set-message *name short-name*) ⇒ void
  name : (union string #f)
  short-name : string

  The string *short-name* is the name that is shown on the button and *name* is shown when the button is clicked on, in a separate window. If *name* is #f, a message indicating that the file hasn't been saved is shown.

2.4 drscheme:rep:context<%>

Objects that match this interface provide all of the services that the drscheme:rep:text% class needs to connect with it's context.

disable-evaluation

- (send *a-drscheme:rep:context* disable-evaluation) ⇒ void

  This method enables all user-sponsored evaluation. See also enable-evaluation.

enable-evaluation

- (send *a-drscheme:rep:context* enable-evaluation) ⇒ void

  This method must disable all user-sponsored evaluatoin. It is called once the user starts some evaluation to ensure that only one evaluation proceeds at a time.

ensure-rep-shown

- (send *a-drscheme:rep:context* ensure-rep-shown) ⇒ void

  This method is called to force the rep window to be visible when, for example, an error message is put into the rep.

get-directory

- (send *a-drscheme:rep:context* get-directory) ⇒ : (union string #f)

  The result of this method is used as the initial directory for the user's program to be evaluted in.

`get-user-setting`

This method is called just before execution to determine the user's setting.

- (send *a-drscheme:rep:context* `get-user-setting`) ⇒ setting

  Returns the current preference setting for `'drscheme:settings`. See `preferences:get` for more information on the preferences system.

`needs-execution?`

This method should return `#t` when the state of the program that the repl reflects has changed.

- (send *a-drscheme:rep:context* `needs-execution?`) ⇒ boolean

`not-running`

- (send *a-drscheme:rep:context* `not-running`) ⇒ void

  This method should update some display in the gui that indicates no evaluation is currently proceeding in the user's world.

`running`

- (send *a-drscheme:rep:context* `running`) ⇒ void

  This method should update some display in the gui that indicates evaluation is currently proceeding in the user's world.

`user-setting`

This is bound to the setting that holds the language settings for the current execution. See also `get-user-setting`.

- (ivar *a-drscheme:rep:context* `user-setting`) ⇒ setting

2.5 drscheme:rep:text%

This class implements a read-eval-print loop for DrScheme. User submitted evaluations in DrScheme are evaluated asynchronously, in an eventspace created for the user. No evaluations carried out by this class affect the implementation that uses it.

The language dialog setting can be recovered from the user's see section **??** in *PLT Framework: GUI Application Framework*) with the key `'drscheme:setting`.

- (`make-object drscheme:rep:text%` *context*) ⇒ `drscheme:rep:text%` object
  context : (implements `drscheme:rep:context<%>`)

`break`

This method is called when the user clicks on the break button or chooses the break menu item.

- (send *a-drscheme:rep:text* break) ⇒ void

  This method breaks the evaluation thread.

display-results

- (send *a-drscheme:rep:text* display-results *results*) ⇒ void
  results : (list-of TST)

  This displays each of the elemnts of *results* in the interactions window, expect those elements of *results* that are void. Those are just ignored.

do-many-evals

Use this function to evaluate code or run actions that should mimic the user's interactions. For example, DrScheme uses this function to evaluate expressions in the definitions window and expressions submitted at the prompt.

- (send *a-drscheme:rep:text* do-many-evals *run-loop*) ⇒ void
  run-loop : (((-¿ void) -¿ void) -¿ void)

  The function *run-loop* is called. It is expected to loop, calling it's argument with a thunk that corresponds to the user's evaluation. It should call it's argument once for each expression the user is evaluating. It should pass a thunk to it's argument that actually does the users's evaluation.

do-many-text-evals

This function evaluates all of the expressions in a text.

- (send *a-drscheme:rep:text* do-many-text-evals *text start end*) ⇒ void
  text : a text% object
  start : int
  end : int

  It evaluates all of the expressions in *text* starting at *start* and ending at *end*, calling do-many-evals to handle the evaluation.

format-source-loc

Builds a string, based on the user's preferences, that describes the source position in some file.

- (send *a-drscheme:rep:text* format-source-loc *start-location end-location*) ⇒ string
  start-location : a zodiac:zodiac struct
  end-location : a zodiac:zodiac struct

  Calls drscheme:basis:format-source-loc with the values of the preferences (see section **??** in *PLT Framework: GUI Application Framework*) 'framework:line-offsets and 'framework:display-line-numbers.

highlight-error

Call this method to highlight an error associated with this repl.

- (send *a-drscheme:rep:text* highlight-error *text start-loc end-loc*) ⇒ void
  text : (instance text:basic%)

start-loc : small-integer
end-loc : small-integer

`initialize-console`

- (send *a-drscheme:rep:text* `initialize-console`) ⇒ void

  This inserts the "Welcome to DrScheme" message into the interactions buffer, calls `reset-console`, `insert-prompt`, and `clear-undos`.

`insert-prompt`

- (send *a-drscheme:rep:text* `insert-prompt`) ⇒ void

  Inserts a new prompt at the end of the text.

`kill-evaluation`

This method is called when the user chooses the kill menu item.

- (send *a-drscheme:rep:text* `kill-evaluation`) ⇒ void

`report-error`

This is called to report an error in the user's program.

- (send *a-drscheme:rep:text* `report-error` *start-location end-location type error-message*) ⇒ void
  start-location : a `zodiac:zodiac` struct
  end-location : a `zodiac:zodiac` struct
  type : symbol
  error-message : string

  See *PLT McMicMac: Parser Manual* for the definition of the `zodiac:zodiac` struct.

  The default behavior is to higlight the range from the start-location to end-location in the text named in the `file` field of *start-location*, if the `file` field is a `text%` instance.

  If the file field is not an instance of `text%`, it will pop up a modal dialog with the error message and the source location.

`reset-console`

- (send *a-drscheme:rep:text* `reset-console`) ⇒ void

  Kills the old eventspace, and creates a new parameterization

  Also calls the super method.

  To change/extend the user parameter settings, override this method, and after the call to the super method returns, change the value of the parameters in the user's thread. For example, to add a definition of a function, `f`, to the users' namespace, add this to a interactions class extension (see `drscheme:get/extend:extend-interactions-text` for details)

```
(inherit user-namespace)
(rename [super-reset-console reset-console])
(public
```

```
[reset-console
  (lambda ()
     (super-reset-console)  ;; initialize user-namespace ivar
     (parameterize ([current-namespace user-namespace])
        (global-defined-value 'f (lambda (...) ...))))]))
```

`run-in-evaluation-thread`

This function runs it's arguments in the user evaluation thread. This thread is the same as the user's eventspace main thread.

See also `do-many-evals`.

- `(send` *a-drscheme:rep:text* `run-in-evaluation-thread` *f*`)` ⇒ void
  f : (-¿ void)

  Calls *f*, after switching to the user's thread.

`shutdown`

Shuts down the user's program and all windows. Reclaims any resources the program allocated.

- `(send` *a-drscheme:rep:text* `shutdown)` ⇒ void

`user-thread`

This is the thread that the users code runs in. It is updated with `set!` each time the user clicks on the execute button.

It is `#f` before the first time the user click on the Execute button.

This thread has all of its parameters initialized according to the settings of the curren execution. See parameters, §9.4 in *PLT MzScheme: Language Manual* for more information about parameters.

- `(ivar` *a-drscheme:rep:text* `user-thread)` ⇒ (union #f thread)

2.6 drscheme:unit:definitions-canvas%

Superclass: `editor-canvas%`

Initializes the visiblity of the save button.

- `(make-object drscheme:unit:definitions-canvas%` *parent editor style scrolls-per-page*`)` ⇒ drscheme:unit:defi
  object
  parent : `frame%`, `dialog%`, `panel%`, or `pane%` object
  editor = `#f` : `text%` or `pasteboard%` object or `#f`
  style = `null` : list of symbols in `'(no-hscroll no-vscroll hide-hscroll hide-vscroll)`
  scrolls-per-page = 100 : exact integer in [1, 10000]

  The *style* list can contain the following flags:

  – `'no-hscroll` — disallows horizontal scrolling

- 'no-vscroll — disallows vertical scrolling
- 'hide-hscroll — allows horizontal scrolling, but hides the horizontal scrollbar
- 'hide-vscroll — allows vertical scrolling, but hides the vertical scrollbar

While vertical scrolling of text editors is based on lines, horizontal scrolling and pasteboard vertical scrolling is based on a fixed number of steps per horizontal page. The *scrollsPerPage* argument sets this value.

If a canvas is initialized with #f for *editor*, install an editor later with set-editor.

2.7 drscheme:unit:frame%

Implements: drscheme:rep:context<%>

This frame inserts the Scheme and Language menus into the menu bar as it is initialized.

- (make-object drscheme:unit:frame% *label parent width height x y style*) ⇒ drscheme:unit:frame% object
  label : string
  parent = #f : frame% object or #f
  width = #f : exact integer in [0, 10000] or #f
  height = #f : exact integer in [0, 10000] or #f
  x = #f : exact integer in [0, 10000] or #f
  y = #f : exact integer in [0, 10000] or #f
  style = null : list of symbols in '(no-resize-border no-caption no-system-menu mdi-parent mdi-child)

  The *label* string is displayed in the frame's title bar. If the frame's label is changed (see set-label), the title bar is updated.

  The *parent* argument can be #f or an existing frame. Under Windows, if *parent* is an existing frame, the new frame is always on top of its parent. Also, the *parent* frame may be an MDI parent frame from a new MDI child frame. Under Windows and X (for many window wanagers), a frame is iconized when its parent is iconized.

  If *parent* is #f, then the eventspace for the new frame is the current eventspace, as determined by current-eventspace . Otherwise, *parent*'s eventspace is the new frame's eventspace.

  If the *width* or *height* argument is not #f, it specifies an initial size for the frame (in pixels) assuming that it is larger than the minimum size, otherwise the minimum size is used.

  If the *x* or *y* argument is not #f, it specifies an initial location for the frame. Otherwise, a location is selected automatically (tiling frames and dialogs as they are created).

  The *style* flags adjust the appearance of the frame on some platforms:

  - 'no-resize-border — omits the resizeable border around the window (Windows) or grow box in the bottom right corner (MacOS)
  - 'no-caption — omits the title bar for the frame (Windows)
  - 'no-system-menu — omits the system menu (Windows)
  - 'mdi-child — creates the frame as a MDI (multiple document interface) child frame, mutually exclsuive with 'mdi-parent (Windows)
  - 'mdi-parent — creates the frame as a MDI (multiple document interface) parent frame, mutually exclsuive with 'mdi-child (Windows)

  If the 'mdi-child style is specified, the *parent* must be a frame with the 'mdi-parent style, otherwise an exn:application:mismatch exception is raised.

Even if the frame is not shown, a few notification events may be queued for the frame on creation. Consequently, the new frame's resources (e.g., memory) cannot be reclaimed until some events are handled, or the frame's eventspace is shut down.

`button-panel`

This panel goes along the top of the drscheme window and has buttons for important actions the user frequently executes.

A tool can add a button to this panel to make some new functionality easily accessible to the user.

- `(ivar` *a-drscheme:unit:frame* `button-panel)` ⇒ a `horizontal-panel%` object

`change-to-file`

- `(send` *a-drscheme:unit:frame* `change-to-file` *file*`)` ⇒ void
  file : string

  Loads this file into this already created frame. This method is only called if this is the first frame opened and no editing has occurred.

`definitions-canvas`

This canvas is the canvas containing the `definitions-text`. It is initially the top half of the drscheme window.

This canvas defaults to a `drscheme:unit:definitions-canvas%` object, but if you change the `drscheme:get/extend:extend-definitions-canvas` procedure, it will use the class in the parameter to create the canvas.

- `(ivar` *a-drscheme:unit:frame* `definitions-canvas)` ⇒ a `drscheme:unit:definitions-canvas%` object

`definitions-text`

This text is initially the top half of the drscheme window and contains the users program.

This text defaults to a `text%` object, but if you change `drscheme:get/extend:extend-definitions-text` procedure, it will use the extended class to create the text.

- `(ivar` *a-drscheme:unit:frame* `definitions-text)` ⇒ a `text%` object.

`disable-evaluation`

- `(send` *a-drscheme:unit:frame* `disable-evaluation)` ⇒ void

  Disables the execute button, the interactions window, and the definitions window.

`enable-evaluation`

- `(send` *a-drscheme:unit:frame* `enable-evaluation)` ⇒ void

  Enables the execute button, the interactions window, and the definitions window.

`execute-callback`

This method is called when the user clicks on the execute button.

- `(send` *a-drscheme:unit:frame* `execute-callback)` ⇒ void

  It calls `ensure-rep-shown` and then it calls `do-many-text-evals` passing in the `interactions-text` and its entire range, unless the first two characters are "#!" in which case, it skips the first line.

`get-text-to-search`

Override this method to specify which text to search.

- `(send` *a-drscheme:unit:frame* `get-text-to-search)` ⇒ a `text:searching%` object

  returns the text that is active in the last canvas passed to `make-searchable`

`interactions-canvas`

This canvas is the canvas containing the `interactions-text`. It is initially the bottom half of the drscheme window.

This canvas defaults to a `drscheme:unit:interactions-canvas%` object, but if you use the `drscheme:get/extend:extend-interactions-canvas` procedure, it will use the extended class to create the canvas.

- `(ivar` *a-drscheme:unit:frame* `interactions-canvas)` ⇒ a `drscheme:unit:interactions-canvas%` object

`interactions-text`

This text is initially the bottom half of the drscheme window and contains the users interactions with the REPL.

This text defaults to a `drscheme:rep:text%` object, but if you use the `drscheme:get/extend:extend-interactions-text` procedure, it will use the extended class to create the text.

- `(ivar` *a-drscheme:unit:frame* `interactions-text)` ⇒ a `drscheme:rep:text%` object.

`make-searchable`

- `(send` *a-drscheme:unit:frame* `make-searchable` *canvas*`)` ⇒ void
  canvas : a `drscheme:unit:interactions-canvas%` object

  stores the canvas, until `get-text-to-search` is called.

`update-shown`

This method is called when the user selects items of the View menu.

- `(send` *a-drscheme:unit:frame* `update-shown)` ⇒ void

  Updates the interactions and definitions windows based on the contents of the menus.

2.8 drscheme:unit:interactions-canvas%

- (make-object drscheme:unit:interactions-canvas% *parent editor style scrolls-per-page*) ⇒ drscheme:unit:int
object
parent : frame%, dialog%, panel%, or pane% object
editor = #f : text% or pasteboard% object or #f
style = null : list of symbols in '(no-hscroll no-vscroll hide-hscroll hide-vscroll)
scrolls-per-page = 100 : exact integer in [1, 10000]

The *style* list can contain the following flags:

- 'no-hscroll — disallows horizontal scrolling
- 'no-vscroll — disallows vertical scrolling
- 'hide-hscroll — allows horizontal scrolling, but hides the horizontal scrollbar
- 'hide-vscroll — allows vertical scrolling, but hides the vertical scrollbar

While vertical scrolling of text editors is based on lines, horizontal scrolling and pasteboard vertical scrolling is based on a fixed number of steps per horizontal page. The *scrollsPerPage* argument sets this value.

If a canvas is initialized with #f for *editor*, install an editor later with set-editor.

2.9 Languages

This set of functions deal with the language level settings for DrScheme. Along with that comes a type, setting that captures all of the settings for each language level. These functions operate on elemnts of that type.

current-setting

This is a parameter (see section 9.4 in *PLT MzScheme: Language Manual*) that has the value of the current setting. This parameter's value reflects the current settings of the language in the interactions window, which may be different from the current settings in the language dialog. The language dialog setting can be recovered from the user's preferences (see section **??** in *PLT Framework: GUI Application Framework*) with the key 'drscheme:setting.

- (current-setting) ⇒ setting
Returns the value of the parameter.
- (current-setting *setting*) ⇒ void
setting : setting
Sets the current value of the parameter to *setting*.

drscheme:basis:add-setting

- (drscheme:basis:add-setting *setting*) ⇒ void
setting : setting
Adds *setting* to the list of settings in settings.

drscheme:basis:bottom-escape-handler

This is a parameter that is called when the exception handler does not escape. It must escape. Use this with mzscheme's event-dispatch-handler to set up an escaping continuation for each event in an eventspace, if necessary.

- (drscheme:basis:bottom-escape-handler) ⇒ (-¿ TST)

  Gets the value of the parameter.

- (drscheme:basis:bottom-escape-handler *escape-handler*) ⇒ void
  escape-handler : (-¿ TST)

  Sets the value of the parameter.

drscheme:basis:copy-setting

- (drscheme:basis:copy-setting *setting*) ⇒ setting
  setting : setting

  Makes a copy of *setting*.

drscheme:basis:current-vocabulary

This parameter will be set to a Zodiac vocabulary after calling drscheme:basis:initialize-parameters.

- (drscheme:basis:current-vocabulary) ⇒ vocabulary

  returns the current Zodiac vocabulary

- (drscheme:basis:current-vocabulary *vocab*) ⇒ void
  vocab : zodiac:vocab

  Sets the vocabulary to *vocab*.

drscheme:basis:error-display/debug-handler

This is parameter that is called to display errors. It's default constructs a string from the second argument and calls the built-in error-display-handler.

- (drscheme:basis:error-display/debug-handler) ⇒ (string zodiac:zodiac exn -¿ void)

- (drscheme:basis:error-display/debug-handler *new-handler*) ⇒ void
  new-handler : (string zodiac:zodiac exn -¿ void)

drscheme:basis:find-setting-named

- (drscheme:basis:find-setting-named *name*) ⇒ setting
  name : string

  Finds the setting with the name give by *name*.

drscheme:basis:format-source-loc

Builds a string representing the error location.

- (drscheme:basis:format-source-loc *start-location end-location start-at-one? lines-and-columns?*) ⇒ string
  start-location : a zodiac:zodiac struct
  end-location : a zodiac:zodiac struct

start-at-one? = `#t` : boolean
lines-and-columns? = `#t` : boolean

If *start-at-one?* is `#f`, the line and column offsets start from zero, otherwise they start at one.

If *lines-and-columns?* is `#f`, only the character offset from the start of the file is used, otherwise, the line and column numbers are used (the *start-at-one?* flag also affects the inital offset).

`drscheme:basis:get-default-setting`

Returns a copy of the default setting, the one for the Beginner language level.

- `(drscheme:basis:get-default-setting` *setting*`)` ⇒ setting
  setting : setting

`drscheme:basis:get-default-setting-name`

- `(drscheme:basis:get-default-setting-name` *setting*`)` ⇒ string
  setting : setting

  Gets the default setting's name.

`drscheme:basis:initialize-parameters`

- `(drscheme:basis:initialize-parameters` *custodian* *setting*`)` ⇒ void
  custodian : custodian
  setting : setting

  This initializes the parameters (see section 9.4 in *PLT MzScheme: Language Manual*) for the current thread to enable evaluation in the language level specified by *setting*. The argument *custodian* is installed as the current custodian (see section 9.5 in *PLT MzScheme: Language Manual*).

  This procedure sets the following parameters:

  1. break-enabled
  2. compile-allow-set!-undefined
  3. compile-allow-cond-fallthrough
  4. current-eval
  5. current-load
  6. current-setting
  7. current-custodian
  8. current-exception-handler
  9. current-namespace
  10. current-zodiac-namespace
  11. current-print
  12. current-load-relative-directory
  13. current-require-relative-collection
  14. error-print-width
  15. error-value-¿string-handler
  16. global-port-print-handler
  17. print-graph
  18. print-struct
  19. read-case-sensitive
  20. read-curly-brace-as-paren
  21. read-square-bracket-as-paren
  22. use-compiled-file-kinds

It also sets these Zodiac parameters, which control how code is generated:

1. aries:signal-undefined
2. aries:signal-not-boolean
3. zodiac:allow-reader-quasiquote
4. zodiac:disallow-untagged-inexact-numbers
5. zodiac:allow-improper-lists

It also sets these MzLib (see section 15 in *PLT MzScheme: Language Manual*). parameters,

1. mzlib:print-convert:constructor-style-printing
2. mzlib:print-convert:quasi-read-style-printing
3. mzlib:print-convert:show-sharing
4. mzlib:print-convert:whole/fractional-exact-numbers
5. mzlib:print-convert:abbreviate-cons-as-list
6. mzlib:pretty-print:pretty-print-show-inexactness

Additionally, `zodiac:reset-previous-attribute` is called wih the arguments `#f` and `#f`, unless the language is MrEd Debug, in which case it is called with `#f` and `#t`.

Additionally, the following built in MzScheme primitives may be replaced with version that perform checking, depending on the language level. The replacment only happens in the teaching language levels, (Beginner, Intermediate and Advanced). For more details see **plt/collects/userspce/ricedefr.ss**.

```
<= < > >=
= + * /
cons
set-cdr!
list*
append
append!
```

Additionally, in the non-teaching levels, the variables: `argv` and `program` are set.

`drscheme:basis:number->setting`

- (`drscheme:basis:number->setting` *n*) ⇒ setting
  n : number

  Returns the setting corresponding to the number *name*.

`drscheme:basis:process-file/zodiac`

Use this function to process the contents of a file with zodiac. This function must be called with the parameters controlling the user's environment active.

- (`drscheme:basis:process-file/zodiac` *filename processor annotate?*) ⇒ void
  filename : string
  processor : (((+ process-finish sexp zodiac:parsed) (-¿ void) -¿ void)
  annotate? : boolean

  Iteratively processes the contents of the file named by *filename*. For each expression, calls *processor*. If *annotate?* is `#f`, *processor* receives the parsed form of the expression. If *annotate?* is not `#f`, *processor* recieves an sexpression representing the code to be evaluated for the user's program. Finally, varprocessor will receive an element of the `process-finish` structure after all expressions have been processed.

`drscheme:basis:process-finish?`

- (`drscheme:basis:process-finish?` *object*) ⇒ boolean
  object : TST

  Returns `#t` if *object* is an instance of the `process-finish` struct and `#f` otherwise.

`drscheme:basis:process-sexp/zodiac`

Use this function to process the contents of a file with zodiac. This function must be called with the parameters controlling the user's environment active.

- (`drscheme:basis:process-sexp/zodiac` *sexp processor annotate?*) ⇒ void
  sexp : sexp
  processor : (((+ process-finish sexp zodiac:parsed) (-¿ void) -¿ void)
  annotate? : boolean

  Processes the sexpression *sexp*, and calls *processor*. If *annotate?* is `#f`, *processor* receives the parsed form of the expression. If *annotate?* is not `#f`, *processor* recieves an sexpression representing the code to be evaluated for the user's program. Finally, varprocessor will receive an element of the `process-finish` structure after all expressions have been processed.

`drscheme:basis:r4rs-style-printing?`

- (`drscheme:basis:r4rs-style-printing?` *setting*) ⇒ boolean
  setting : setting

  Returns `#t` if this setting has the R4RS style printing.

`drscheme:basis:setting-name`

- (`drscheme:basis:setting-name` *setting*) ⇒ string
  setting : setting

  Returns the name of the *setting*.

`drscheme:basis:setting-name->number`

- (`drscheme:basis:setting-name->number` *name*) ⇒ number
  name : string

  Returns a number for *setting*. See also `drscheme:basis:number->setting`.

`drscheme:basis:zodiac-vocabulary?`

- (`drscheme:basis:zodiac-vocabulary?` *setting*) ⇒ boolean
  setting : setting

  Returns `#t` if this is a vocabulary that should be processed with zodiac.

`drscheme:language:language-dialog`

This function opens the language dialog and lets the user configure the lagnuage settings.

- (drscheme:language:language-dialog *settings*) ⇒ (union settings #f)
  settings : settings

  The input is a settings struct for the initial state of the dialog. The result is the settings for the new language, based on the user's choices in the dialog. If the uesr cancels the dialog, the result is `#f`.

`drscheme:language:settings-preferences-symbol`

This is defined to be the key used with the preferences system to get the user's current language settings (use this with `preferences:get`).

- *a-drscheme:language:settings-preferences-symbol* ⇒ symbol

`settings`

This list contains one entry for each language level in drscheme.

- *a-settings* ⇒ (list-of setting)

2.10 Processing Programs with Zodiac

These functions are used to process sexpressions, files, and portions of buffers through Zodiac, to retrieve the current vocabulary and other Zodiac related aspects of DrScheme.

`drscheme:basis:process-file/no-zodiac`

- (drscheme:basis:process-file/no-zodiac *filename* *f*) ⇒ void
  filename : string
  f : ((+ process-finish sexp zodiac:parsed) (-¿ void) -¿ void)

  This function process the file named by *filename*. It calls `drscheme:basis:process/no-zodiac`.

`drscheme:basis:process-sexp/no-zodiac`

- (drscheme:basis:process-sexp/no-zodiac *sexp* *f*) ⇒ void
  sexp : sexp
  f : ((+ process-finish sexp zodiac:parsed) (-¿ void) -¿ void)

  This function calls `drscheme:basis:process/no-zodiac`.

`drscheme:basis:process/no-zodiac`

- (drscheme:basis:process/no-zodiac *reader* *f*) ⇒ void
  reader : (-¿ (+ eof sexp))
  f : ((+ sexp drscheme:basis:process-finish) (-¿ void) -¿ void)

  This function is used to process a program, without zodiac. The first argument, *f*, is called until it returns `eof`. The result of the first argument is applied to f, in a similar fashion to `drscheme:basis:process/zodiac`

`drscheme:basis:process/zodiac`

- (`drscheme:basis:process/zodiac` *reader* *f* *annotate?*) ⇒ void
  reader : (-¿ zodiac:sexp)
  f : ((+ drscheme:basis:process-finish sexp zodiac:parsed) (-¿ void) -¿ void)
  annotate? : boolean

  This function is used to process a program with Zodiac. The first argument, *reader* is the result of calling `zodiac:read`. The second argument, *f*, is used to process the intermediate results from zodiac. It must accept either a `drscheme:basis:process-finish` structure, indicating that all of the program is processed, or an sexpression or a `zodiac:parsed` structure. The final parameter `annotate?` determines if *f* receives sexpressions or `zodiac:parsed` structures. If *annotate?* is not `#f`, *f* will be passed sexpressions. If it is `#f`, *f* will be passed `zodiac:parsed` structures.

`drscheme:basis:raw-reader`

This is a parameter that controls the currently active reader for the non-debugging language levels. See also `drscheme:basis:zodiac-reader`.

- (`drscheme:basis:raw-reader` *new-reader*) ⇒ void
  new-reader : (port -¿ sexp)

  sets the new reader to *new-reader*. The first argument of new-reader must default to the value of `(current-input-port)`.

- (`drscheme:basis:raw-reader`) ⇒ (port -¿ sexp)

  Returns the current reader.

`drscheme:basis:zodiac-reader`

This is a parameter that controls the currently active reader for the debugging and teaching language levels. See also `drscheme:basis:raw-reader`.

- (`drscheme:basis:zodiac-reader` *new-reader*) ⇒ void
  new-reader : ((union input-port (-¿ TST)) zodiac-location boolean exact-integer) -¿ (-¿ (union read eof))

  Sets the reader to *new-reader*.

  All four of the arguments to *new-reader* must have defaults. They are (in order):

  - `(current-input-port)`
  - `(zodiac:make-zodiac 1 1 0)`
  - `#t`
  - `1`

- (`drscheme:basis:zodiac-reader`) ⇒ ((union input-port (-¿ TST)) zodiac-location boolean exact-integer) -¿ (-¿ (union read eof))

  Returns the current value of the reader.

`drscheme:interface:static-error`

This is the procedure that is linked into zodiac for static errors.

- (drscheme:interface:static-error *link-text link-tag zodiac fmt args...*) ⇒ TST
  link-text : string
  link-tag : (union symbol #f)
  zodiac : (union zodiac:zodiac zodiac:eof zodiac:period)
  fmt : string
  args... : TST

  Raises an exception. If this function is called in the user's evaluation, and the exception is not caught, DrScheme will highlight the source position given by *zodiac*.

  The *link-text* and *link-tag* arguments are used to index into the documentation. The *link-text* is used as a prefix to *fmt* and if clicked by the user will search for *link-tag* in the manual and go to the first page found there.

  The *fmt* string and the *args* arguments are used to construct the error message. The procedure drscheme:interface:static-error accepts any number of *args* arguments.

drscheme:load-handler:process-text/no-zodiac

- (drscheme:load-handler:process-text/no-zodiac *text f start end*) ⇒ void
  text : a text% object
  f : ((+ process-finish sexp zodiac:parsed) (-¿ void) -¿ void)
  start : int
  end : int

  This function process the text *text*. It calls drscheme:basis:process/no-zodiac.

drscheme:load-handler:process-text/zodiac

- (drscheme:load-handler:process-text/zodiac *text f start end annotate?*) ⇒ void
  text : a text% object
  f : ((+ process-finish sexp zodiac:parsed) (-¿ void) -¿ void)
  start : int
  end : int
  annotate? : boolean

  This function process the text *text*. It calls drscheme:basis:process/zodiac.

drscheme:unit:program-editor-mixin

This mixes in the ability to reset the highlighting for error message when the user modifies the buffer. Use it for editors that have program text where errors can occur. It clears the error highlighting when the buffer is modified.

- (drscheme:unit:program-editor-mixin *text*) ⇒ (extends text:basic%)
  text : (extends text:basic%)

interface:mark-key

This parameter hold the mark key for the source location of syntax and run-time errors.

- (interface:mark-key) ⇒ symbol

  Gets the value of the parameter

- (interface:mark-key *new-key*) ⇒ void
  new-key : symbol

  Sets the value of the parameter.

interface:set-zodiac-phase

This function tells the zodiac interface what phase of zodiac is about to be executed. Call this function before calling zodiac:expand-expr or zodiac:read.

- (interface:set-zodiac-phase *phase*) ⇒ void
  phase : (union 'reader 'expander #f)

  Sets the phase to *phase*.

2.11 Extending the DrScheme Classes

Each of these names is bound to an extender function. In order to change the behavior of drscheme, you can derive new classes from the standard classes for the frame, texts, canvases. Each extender accepts a function as input. The function it accepts must take a class as it's argument and return a classes derived from that class as its result. For example:

```
(drscheme:get/extend:extend-interactions-text
  (lambda (super%)
    (class super%
      (public
        [method1 (lambda (x) ...)]
        ...))))
```

extends the interactions text class with a method named method1.

drscheme:get/extend:extend-definitions-canvas

The unextended class is drscheme:unit:definitions-canvas%. This canvas is used in the top window of drscheme frames.

- (drscheme:get/extend:extend-definitions-canvas *definitions-canvas-mixin*) ⇒ void
  definitions-canvas-mixin : a procedure that accepts a class and produces a class derived from it.

drscheme:get/extend:extend-definitions-text

The unextended class is text:backup-autosave%. This text is used in the top window of drscheme frames.

- (drscheme:get/extend:extend-definitions-text *definitions-text-mixin*) ⇒ void
  definitions-text-mixin : a procedure that accepts a class and produces a class derived from it.

drscheme:get/extend:extend-interactions-canvas

The unextended class is canvas:wide-snip%. This canvas is used in the bottom window of drscheme frames.

- (drscheme:get/extend:extend-interactions-canvas *interactions-canvas-mixin*) ⇒ void
  interactions-canvas-mixin : a procedure that accepts a class and produces a class derived from it.

`drscheme:get/extend:extend-interactions-text`

The unextended class is `drscheme:rep:text%`. This text is used in the bottom window of drscheme frames.

- (`drscheme:get/extend:extend-interactions-text` *interactions-text-mixin*) ⇒ void
  interactions-text-mixin : a procedure that accepts a class and produces a class derived from it.

`drscheme:get/extend:extend-unit-frame`

The unextended class is `drscheme:unit:frame%`. This is the frame that implements the main drscheme window.

- (`drscheme:get/extend:extend-unit-frame` *frame-mixin*) ⇒ void
  frame-mixin : a procedure that accepts a class and produces a class derived from it.

2.12 Help Desk

`drscheme:help-desk:help-desk`

This function opens a help desk window, or brings an already open help desk window to the front. If an argument is specified, that key is searched for.

- (`drscheme:help-desk:help-desk`) ⇒ void

  Opens a help-desk window to the starting page, or just brings a help-desk window to the front (without changing what page it is viewing).

- (`drscheme:help-desk:help-desk` *key*) ⇒ void
  key : string

  Searches for the string *key* as an exact search in both the keyword and the index.

`drscheme:help-desk:open-url`

- (`drscheme:help-desk:open-url` *url*) ⇒ void
  url : string

  Opens *url* in a new help desk window.

`drscheme:help-desk:open-users-url`

- (`drscheme:help-desk:open-users-url` *frame*) ⇒ void
  frame : (union #f (instance `frame%`))

  Queries the user for a URL and opens it in a new help desk window. The *frame* argument is used as a parent for the dialog box.

3. Zodiac Reference

3.1 zodiac:expands<%>

`expand`

This method returns the zodiac sexpression that corresponds to the expansion of this object. It accepts the zodiac read object that corresonds to the object.

See `zodiac:structurize-syntax`.

- (`send` *a-zodiac:expands* `expand` *object*) ⇒ zodiac:sexp
  object : zodiac:zodiac

3.1.0.1 Units

zodiac:unit-form **:** *zodiac:parsed* (imports exports clauses)

imports is a list of *zodiac:lexical-binding*s. exports is a list of pairs. The first projection of each pair contains a *zodiac:top-level-varref/bind*, while the second projection contains a (Zodiac) *zodiac:symbol*. The first projection corresponds to the internal name, and the second projection to the exported name. When no renaming is specified, the same name is used for both projections. clauses is a list of *zodiac:parsed* objects, corresponding to the expressions in the unit.

zodiac:compound-unit-form **:** *zodiac:parsed* (imports links exports)

imports is a list of *zodiac:lexical-binding*s. link is a list of lists. Each list corresponds to one link clause. The `car` of the list is a (Zodiac) *zodiac:symbol* giving the link tag. The `cadr` is a *zodiac:parsed* object holding the expression specifying the unit to link in that clause. The `cddr` is the list of arguments to the unit. Each of the arguments is either a *zodiac:lexical-varref*, corresponding to an imported variable, or a pair of a *zodiac:symbol* (for the link clause) and a *zodiac:symbol* (for the exported name), corresponding to importing from another unit. Finally, *zodiac:exports* is a list of export clauses. The `car` of each clause is a *zodiac:symbol*, naming the link clause; the `cadr` and `cddr` are *zodiac:symbol*s giving the internal and exported names, respectively.

zodiac:invoke-unit-form **:** *zodiac:parsed* (unit variables)
zodiac:invoke-open-unit-form **:** *zodiac:parsed* (unit name-specifier variables)

unit is a *zodiac:parsed* object; variables is a list of *zodiac:parsed* objects. name-specifier can have two forms: `#f`, the false value, if no name prefix is given, or a (Zodiac) *zodiac:symbol* object, giving the specified name prefix.

3.1.0.2 Objects

zodiac:interface-form **:** *zodiac:parsed* (super-exprs variables)

super-exprs is a list of *zodiac:parsed* expressions, and variables is a list of *zodiac:symbol*s.

zodiac:class/names-form* **:** *zodiac:parsed* (this super-init super-expr interfaces init-vars inst-clauses)

this is a *zodiac:lexical-binding* giving the name for the self-variable, super-init is a *zodiac:superinit-binding*, super-expr is of type *zodiac:parsed*, interfaces is a list of type *zodiac:parsed*, init-vars is a

"parallel optional argument list" (see section 3.2.6.1) and inst-clauses is a list of body clauses (see section 3.1.0.2).

Variables

zodiac:supervar-binding : *zodiac:binding* ()
zodiac:superinit-binding : *zodiac:binding* ()
zodiac:public-binding : *zodiac:binding* ()
zodiac:private-binding : *zodiac:binding* ()
zodiac:inherit-binding : *zodiac:binding* ()
zodiac:rename-binding : *zodiac:binding* ()
zodiac:supervar-varref : *zodiac:bound-varref* ()
zodiac:superinit-varref : *zodiac:bound-varref* ()
zodiac:public-varref : *zodiac:bound-varref* ()
zodiac:private-varref : *zodiac:bound-varref* ()
zodiac:inherit-varref : *zodiac:bound-varref* ()
zodiac:rename-varref : *zodiac:bound-varref* ()

Clauses

NOTE: The following convention is used: exports is a list of (Zodiac) *zodiac:symbol*s; internals is a list of the appropriate kind of *zodiac:binding*s; exprs is a list of *zodiac:parsed* expressions; and imports is a list of (Zodiac) *zodiac:symbols*.

zodiac:public-clause (exports internals exprs)
zodiac:private-clause (internals exprs)
zodiac:inherit-clause (internals imports)
zodiac:rename-clause (internals imports)
zodiac:sequence-clause : (exprs)

3.2 Zodiac

3.2.1 Motivation

A typical program-processing tool consists of several components: a reader, a parser, and the actual processing component. The reader converts the input text into some internal representation. This representation is parsed into abstract syntax. The core of the tool processes the abstract syntax and possibly produces some output. The output is finally presented to the programmer.

Ideally, the output of a program-processing tool should be presented in terms of the original program. The best way to achieve this form of reporting is to have **source-object correlation** (or "source correlation"). Unfortunately, Scheme macros can transform programs in numerous ways, making the task of source correlation difficult.

This document describes the Zodiac package, which provides a front-end for Scheme that generates source correlation maps. The front-end consists of a scanner, reader, macro-expander and parser, which can be combined selectively. It provides a common ground from which numerous programming tools can be built and given powerful and convenient user interfaces.

The rest of this document describes each of these three phases. The parser is only sparsely specified, since the actual abstract syntax produced by it is completely controlled by the user. (Indeed, this is one of the features of Zodiac.) Separate documents will describe the default abstract syntaxes provided with Zodiac.

3.2.2 Notations and Terminology

These documents assume a strong familiarity with MzScheme. In particular, the implementation of Zodiac makes extensive use of structures and sub-typing, units and classes.

In these documents, a structure declaration is written as follows:

type (field)

corresponds to the Scheme code

```
(define-struct type (field))
```

Sub-typing is declared as in

sub-type **:** *type* (added-field)

which corresponds to code such as

```
(define-struct (sub-type struct:type) (added-field))
```

In the text, the types are written as *type*, and the fields as field.

Some of the following chapters have sections on the types used and the procedures provided. It will be assumed that the available procedures automatically include all those arising out of the structure declarations mentioned in the types section, even if these are not explicated in the section on procedures.

3.2.3 Core of Zodiac

All structures in these documents, unless otherwise mentioned, are sub-types of a single structure, named *zodiac:zodiac*. This structure has the form

zodiac:zodiac (origin start finish)

where origin is an *zodiac:origin* struct, while start and finish are *zodiac:location* structs. The origin field is currently unused, and the *zodiac:origin* struct is correspondingly unspecified. Locations are represented as a tuple of the line number, column number, file offset and file name:

zodiac:location (line column offset file)

The line and column fields contain positive integers starting at 1, while offset contains a non-negative integer that starts at 0. The type of file is left unspecified. The *zodiac:period* struct provides the location of periods in improper lists:

zodiac:period (location)

Note that *zodiac:origin*, *zodiac:location* and *zodiac:period* are not sub-types of *zodiac:zodiac*.

3.2.4 Scanner

The scanner returns two kinds of objects: tokens in the input program, or the end-of-file delimiter. The latter is returned as an *zodiac:eof* struct:

zodiac:eof (location)

while all other objects returned by the scanner are a sub-type of *zodiac:scanned*:

zodiac:scanned **:** *zodiac* ()

In turn, *zodiac:scanned* has one sub-type: *zodiac:token*, which is the most specific type of all the objects returned by the scanner.

zodiac:token **:** *scanned* (object type)

The object and type fields will be documented later.

3.2.5 Reader

Like the scanner, the reader returns either an end-of-file delimiter or the actual object read. The end-of-file object is of type *zodiac:eof*, as defined in Section **??**. All other values are elements of *zodiac:read*[1]:

zodiac:read **:** *zodiac* (object)

The reader's output is sub-divided into scalar and sequence objects[2]:

zodiacscalar **:** *read* ()
zodiac:sequence **:** *read* (length)

Most of these sub-types should be self-explanatory:

zodiac:string **:** *scalar* ()
zodiac:boolean **:** *scalar* ()
zodiac:number **:** *scalar* ()
zodiac:symbol **:** *scalar* (orig-name marks)
zodiac:char **:** *scalar* ()
zodiac:external **:** *scalar* ()
zodiac:list **:** *sequence* (marks)
zodiac:vector **:** *sequence* ()
zodiac:improper-list **:** *sequence* (period marks)

In the case of *zodiac:scalar* objects, the object field contains the Scheme representation of that object. All *zodiac:sequence* objects have a list of *zodiac:read* objects in their object field; in the case of *zodiac:improper-list*, the length of this list is one greater than the number of pairs that constitute the list.

The *zodiac:external* struct may contain any scheme value in the read field, except those listed in other structures. That is, it will not be a string, boolean, number, symbol or character, but it might be a procedure, the void value, or an instance of a class.

The period field contains a *zodiac:period* which gives the location of the period in the source that marks a list as being improper. The orig-name and marks fields are used by parsers that perform hygienic macro-expansion[3].

3.2.5.1 Argument Lists

Argument lists are encapsulated within a structure:

zodiac:arglist (vars)

[1] Rhymes with "dead", "head", "routinely bled" and "positively fed".
[2] Strings are classified as scalar objects.
[3] There is currently no clean way of hiding this detail from the user of Zodiac; elucubration on this is forthcoming.

The vars field is expected to *always* be a list of *zodiac:binding* identifiers. To distinguish between the different structures of argument lists, a sub-type is used. In Core Scheme, argument lists in the input can only be (syntactic) lists of identifiers:

zodiac:sym-arglist **:** *arglist* ()

Higher language levels may permit more kinds of argument lists.

`zodiac:arglist-decls-vocab`

- `(zodiac:arglist-decls-vocab)` ⇒ void

  UNDOCUMENTED

`zodiac:arglist-pattern`

- `(zodiac:arglist-pattern)` ⇒ void

  UNDOCUMENTED

`zodiac:distinct-valid-id/s?`

- `(zodiac:distinct-valid-id/s?)` ⇒ void

  UNDOCUMENTED

`zodiac:distinct-valid-syntactic-id/s?`

- `(zodiac:distinct-valid-syntactic-id/s?)` ⇒ void

  UNDOCUMENTED

`zodiac:expand-expr`

- `(zodiac:expand-expr` *read env attrib vocab*`)` ⇒ zodiac:parsed
  read : read
  env : zodiac:env
  attrib : zodiac:attr
  vocab : zodiac:vocab

  See Zodiac Environments for information on the *env* argument, Zodiac Attributes for information on the *attrib* argument and Zodiac Vocabularies for information on the *vocab* argument.

`zodiac:extend-parsed->raw`

- `(zodiac:extend-parsed->raw)` ⇒ void

  UNDOCUMENTED

`zodiac:generate-name`

- `(zodiac:generate-name)` ⇒ void

  UNDOCUMENTED

`zodiac:in-lexically-extended-env`

- `(zodiac:in-lexically-extended-env)` ⇒ void

  UNDOCUMENTED

`zodiac:internal-error`

- `(zodiac:internal-error` *zodiac format*`)` ⇒ doesn't
  zodiac : zodiac:zodiac
  format : string

  This function accepts arbitrarily many arguments after *format*.

  The procedure `internal-error` is for critical errors; since it is not possible to guarantee that the object in question is in the Zodiac hierarchy (indeed, that may sometimes be the error), *zodiac:object* is flexible enough to accept any kind of Scheme object.

  The argument *format* is used the format string to `printf`, and the remaining arguments are meant to satisfy parameters in the format string.

`zodiac:language<=?`

- `(zodiac:language<=?)` ⇒ void

  UNDOCUMENTED

`zodiac:language>=?`

- `(zodiac:language>=?)` ⇒ void

  UNDOCUMENTED

`zodiac:lexically-resolved?`

- `(zodiac:lexically-resolved?)` ⇒ void

  UNDOCUMENTED

`zodiac:make-argument-list`

- `(zodiac:make-argument-list)` ⇒ void

  UNDOCUMENTED

`zodiac:make-empty-back-box`

- `(zodiac:make-empty-back-box)` ⇒ void

  UNDOCUMENTED

`zodiac:make-optargument-list`

- `(zodiac:make-optargument-list)` ⇒ void

  UNDOCUMENTED

`zodiac:marks-equal?`

- `(zodiac:marks-equal?)` ⇒ void
  UNDOCUMENTED

`zodiac:name-eq?`

- `(zodiac:name-eq?)` ⇒ void
  UNDOCUMENTED

`zodiac:optarglist-decls-vocab`

- `(zodiac:optarglist-decls-vocab)` ⇒ void
  UNDOCUMENTED

`zodiac:optarglist-pattern`

- `(zodiac:optarglist-pattern)` ⇒ void
  UNDOCUMENTED

`zodiac:parsed->raw`

- `(zodiac:parsed->raw)` ⇒ void
  UNDOCUMENTED

`zodiac:read`

- `(zodiac:read` *input location script? first-column*`)` ⇒ (-¿ (union read eof))
  input = `(current-input-port)` : (union input-port (-¿ TST))
  location = `(make-zodiac 1 1 0)` : zodiac-location
  script? = `#t` : boolean
  first-column = `1` : exact-integer

  When invoked, the reader returns a thunk. Repeatedly invoke this thunk to obtain a series of *zodiac:read* objects until an *zodiac:eof* is returned. The names and the functionality of the optional arguments to the reader, in turn, are:

  input This argument can be either an input port or a thunk from which to take the input. The thunk should return a *zodiac:char*, *zodiac:eof* or an object appropriate for *zodiac:external*.

  initial-location The location used for the first character read from the port; subsequent characters are appropriately offset from it.

  script? Whether or not the file is a script. In a script, if the first two chars from port are `#!`, then the reader will treat the first line as a comment. (This comment can span multiple lines if each preceding line ends in a \ before the newline.)

  first-column The first column of each line is can be changed by this argument. This is useful for treating the entire file as if it were indented by some amount. Note that this parameter is unrelated to the initial location parameter.

  NOTE: It is an error to perform `read-char` on any port passed to the reader, since this may interfere with its operation.

`zodiac:scheme-expand`

- `(zodiac:scheme-expand)` ⇒ void
  UNDOCUMENTED

`zodiac:scheme-vocabulary`

- `(zodiac:scheme-vocabulary)` ⇒ void
  UNDOCUMENTED

`zodiac:sexp->raw`

- `(zodiac:sexp->raw` *sexp*`)` ⇒ sexp
  sexp : zodiac:sexp

  The input is a member of the *zodiac:read* hierarchy. The body is recursively translated into raw Scheme s-expressions. For *zodiac:symbol*s, the value in the object field, not in the orig-name field, is used.

`zodiac:static-error`

- `(zodiac:static-error` *zodiac format*`)` ⇒ doesn't
  zodiac : zodiac:zodiac
  format : string

  This function accepts arbitrarily many arguments after *format*.

  Use `static-error` should be used to report syntactic errors. It will not return.

  The argument *format* is used the format string to `printf`, and the remaining arguments are meant to satisfy parameters in the format string.

`zodiac:structurize-syntax`

- `(zodiac:structurize-syntax` *sexp zodiac marks*`)` ⇒ zodiac:read
  sexp : mixed
  zodiac : zodiac:zodiac
  marks = `???` : marks

  The first argument is a raw Scheme s-expression that has *zodiac:read* objects in one or more positions (type *zodiac:mixed*). The second argument is any object that is an instance of a sub-type of *zodiac:zodiac*. The output is a *zodiac:read* representation of the input. All Scheme s-expressions in the input are recursively converted to *zodiac:read* forms, while *zodiac:read* forms are left untouched (and are not traversed further). For all raw inputs that are converted into *zodiac:read* objects, the origin, start and finish information is extracted from the second argument to `structurize-syntax`. The optional marks argument is used to give *zodiac:symbol*s and *zodiac:list* forms their initial set of marks. Ordinary users may ignore this argument.

`zodiac:syntax-car`

- `(zodiac:syntax-car` *sexp*`)` ⇒ zodiac:read
  sexp : (union zodiac:list zodiac:improper-list)

  Takes the "car" of the syntax. The `read-object` accessors should not be used to access them. Instead use these procedures: `zodiac:syntax-car`, `zodiac:syntax-cdr`, `zodiac:syntax-null?` , and `zodiac:syntax-map`.

Use `zodiac:structurize-syntax` to get the effect of a `zodiac:syntax-cons`.

`zodiac:syntax-cdr`

- (`zodiac:syntax-cdr` *sexp*) ⇒ zodiac:read
  sexp : (union zodiac:list zodiac:improper-list)

  Takes the "cdr" of the syntax. The `read-object` accessors should not be used to access them. Instead use these procedures: `zodiac:syntax-car`, `zodiac:syntax-cdr`, `zodiac:syntax-null?` , and `zodiac:syntax-map`.

  Use `zodiac:structurize-syntax` to get the effect of a `zodiac:syntax-cons`.

`zodiac:syntax-map`

- (`zodiac:syntax-map` *f l1 l2*) ⇒ B
  f : (union (TST TST -¿ B) (TST -¿ B))
  l1 : zodiac:list
  l2 = `#f` : zodiac:list

  As with Scheme's `map`, `syntax-map` can take more than one argument (currently, at most two are allowed).

  The `read-object` accessors should not be used to access them. Instead use these procedures: `zodiac:syntax-car`, `zodiac:syntax-cdr`, `zodiac:syntax-null?` , and `zodiac:syntax-map`.

  Use `zodiac:structurize-syntax` to get the effect of a `zodiac:syntax-cons`.

3.2.6 Parser

Parsers primarily convert *zodiac:read* objects into objects of type *zodiac:parsed*. This section describes the structure hierarchy for Scheme.

3.2.6.1 Preliminaries

zodiac:parsed **:** *zodiac:zodiac* (back)

All the output from the parser is an element of *zodiac:parsed*. Each back contains a distinct box in which information can be stored. All parsed output is either a variable reference, an application, or a special form.

zodiac:form **:** *zodiac:parsed* ()
zodiac:app **:** *zodiac:parsed* (fun args)

The fun field contains a single *zodiac:parsed* object, while zodiac:args holds a list of these.

Variables

zodiac:varref **:** *zodiac:parsed* (var)
zodiac:top-level-varref **:** *zodiac:varref* ()
zodiac:bound-varref **:** *zodiac:varref* (binding)
zodiac:lexical-varref **:** *zodiac:bound-varref* ()
zodiac:binding **:** *zodiac:parsed* (var orig-name)
zodiac:lexical-binding **:** *zodiac:binding* ()

All variable references fall under *zodiac:varref*, whose var field contains the name (Scheme symbol) of the variable (possibly with some consistent renaming performed). The binding field contains a *zodiac:binding* struct. The var field of a zodiac:*binding* contains the same *zodiac:symbol* as in the var field of the referring *zodiac:bound-varref*. The orig-name field contains the original name, as specified in the input or by a rewrite rule.

zodiac:top-level-varref/bind **:** *zodiac:top-level-varref* (slot)

When the procedure `scheme-expand-program` is used, top-level variable references are given an extra field, slot, which contains a box. All top-level uses (which can be definitions, mutations and uses) of the same name point to the same box. Thus, the box can be used to share information between these instances. Furthermore, this box holds a list of all the references (both definitions and uses) to the identifer. The elements of the list are of *zodiac:top-level-varref/bind*s. Top-level references inside a unit are not related to references to identifiers with the same name outside a unit, *i.e.*, they do not share a box in the slot field. They do, however, share a box amongst themselves, one per unit.

NOTE: The box in the slot field is unrelated to the one possessed by every *zodiac:top-level-varref/bind* object by virtue of being a sub-type of *zodiac:parsed*; there is a distinct box of the latter kind for every syntactic occurrence of the top-level variable.

Argument Lists

Regular argument lists are of type *zodiac:arglist*. These do not allow the specification of a default initial value. When initial values are allowed, the initial value expressions may be evaluated in different environments. The type *zodiac:paroptarglist*, short for "parallel optional argument list" (what is optional is the specification of an initial value expression) expands all the expressions in an environment augmented with all the formal variables, so they can be mutually referential. Another kind, *zodiac:optarglist*, is available for incremental environment extension from left to right (as in MzScheme's `opt-lambda` construct).

zodiac:arglist (vars)
zodiac:sym-arglist **:** *zodiac:arglist* ()
zodiac:list-arglist **:** *zodiac:arglist* ()
zodiac:ilist-arglist **:** *zodiac:arglist* ()

vars is always a list of *zodiac:lexical-binding*. The additional structure indicates whether the argument list is a single zodiac:symbol, a proper list or an improper list. In the first of these cases, vars has length one; in the last of these cases, the period before the last argument is implicit in vars.

zodiac:paroptarglist (vars)
zodiac:sym-paroptarglist **:** *zodiac:paroptarglist* ()
zodiac:list-paroptarglist **:** *zodiac:paroptarglist* ()
zodiac:ilist-paroptarglist **:** *zodiac:paroptarglist* ()

The structure of vars in *zodiac:paroptarglist* is similar to that in *zodiac:arglist*. The only exception is that, for expressions where an initial value has been supplied, the element of the list is a pair whose first argument is the *zodiac:lexical-binding* and whose second argument is in *zodiac:parsed*.

3.2.6.2 Core Scheme

NOTE: As a convention in this section, val and body will hold a *zodiac:parsed* object; var will contain a *zodiac:binding*. The plural forms, vals and bodies, contain lists of *zodiac:parsed*.

zodiac:set!-form **:** *zodiac:form* (var val)
zodiac:begin-form **:** *zodiac:form* (bodies)
zodiac:begin0-form **:** *zodiac:form* (bodies)

These structures are explained by the above convention.

zodiac:define-values-form **:** *zodiac:form* (vars val)

vars is a list of *zodiac:binding*.

zodiac:let-values-form **:** *zodiac:form* (vars vals body)
zodiac:letrec-values-form* **:** *zodiac:form* (vars vals body)

The vars fields are lists of lists of *zodiac:binding*.[4]

zodiac:if-form **:** *zodiac:form* (test then else)

test, then and else are *zodiac:parsed*.

[4] Arguably, the fields should have been called varss, but we chose not to play hob with language.

zodiac:quote-form : *zodiac:form* (expr)

expr contains a *zodiac:read* object.

zodiac:case-lambda-form : *zodiac:form* (args bodies)

args is a list of argument lists. Each element of args is an *zodiac:arglist* (see section 3.2.6.1).

zodiac:struct-form : *zodiac:form* (type super fields)

type is a (Zodiac) *zodiac:symbol*; fields is a list of these. *zodiac:super* is either the false value (`#f`) or a *zodiac:parsed* object, depending on whether or not a super-type expression was specified.

3.2.6.3 Image Values

`zodiac:syntax-null?`

- `(zodiac:syntax-null?` *sexp*`)` ⇒ zodiac:read
  sexp : (union zodiac:list zodiac:improper-list)

  Tests to see if the syntax is "null". The `read-object` accessors should not be used to access them. Instead use these procedures: `zodiac:syntax-car`, `zodiac:syntax-cdr`, `zodiac:syntax-null?` , and `zodiac:syntax-map`.

  Use `zodiac:structurize-syntax` to get the effect of a `zodiac:syntax-cons`.

`zodiac:valid-id/s?`

- `(zodiac:valid-id/s?)` ⇒ void
  UNDOCUMENTED

`zodiac:valid-id?`

- `(zodiac:valid-id?)` ⇒ void
  UNDOCUMENTED

`zodiac:valid-syntactic-id/s?`

- `(zodiac:valid-syntactic-id/s?)` ⇒ void
  UNDOCUMENTED

`zodiac:valid-syntactic-id?`

- `(zodiac:valid-syntactic-id?)` ⇒ void
  UNDOCUMENTED

3.3 Environments

An environment maps identifiers in the input to information about their intended behavior in the program. For instance, some identifiers act as keywords that represent a micro or a macro, others are bound by a binding construct, and others are unbound.

Zodiac uses the type *zodiac:env-entry* to range over representations of the possible types of behaviors an identifier can exhibit. *zodiac:env-entry* includes:

zodiac:macro-resolution (rewriter)
zodiac:micro-resolution (rewriter)
zodiac:top-level-resolution ()

The rewriter fields contain a micro or macro, as appropriate. Micros have the type $read \times env \times attr \times vocab \longrightarrow parsed$ while macros have the type $read \times env \longrightarrow read$.

Languages implemented atop Zodiac will extend *env-entry* to reflect their binding constructs. Unless extended, all identifiers that do not resolve to macro or micros will yield *top-level-resolution*s.

`zodiac:extend-env`

- `(zodiac:extend-env` *extension env*`)` ⇒ void
  extension : (list-of (union new-vars marks))
  env : zodiac:env

`zodiac:resolve`

- `(zodiac:resolve` *id env vocab*`)` ⇒ zodiac:env-entry
  id : id
  env : zodiac:env
  vocab : zodiac:vocab

`zodiac:retract-env`

- `(zodiac:retract-env` *retraction env*`)` ⇒ void
  retraction : (list-of new-vars)
  env : zodiac:env

3.4 Attributes

Attributes are used to inherit and synthesize information, and to also communicate it across top-level expression boundaries.

`zodiac:get-attribute`

- `(zodiac:get-attribute` *attr key*`)` ⇒ (union TST #f)
  attr : zodiac:attr
  key : symbol

  `put-attribute` updates the value of an attribute, adding it if not already present.

`zodiac:make-attributes`

- `(zodiac:make-attributes)` ⇒ zodiac:attr

  `make-attributes` creates a new (empty) table of attributes.

`zodiac:put-attribute`

- (`zodiac:put-attribute` *attr key value*) ⇒ zodiac:attr
  attr : zodiac:attr
  key : symbol
  value : TST

  `get-attribute` returns the value of the attribute, if present, and `#f` otherwise. The value of *attr-entry* is fixed by individual applications.

3.5 Vocabulary

Zodiac allows the user to completely specify the syntax of the underlying language. This is done by providing different **vocabularies**, which are collections of expanders for the various parts of the language. Other documentation describes the standard Scheme vocabularies that accompany Zodiac.

A vocabulary consists of micros to manage the treatment of the individual syntactic components: symbols, literals, lists and improper-lists. All sub-types of *scalar* other than *symbol*, in addition to *vector*, are considered "literals"[5]. In addition, micros and macros can be triggered by a leading object of type *symbol* in a *list*.

`zodiac:add-ilist-micro`

- (`zodiac:add-ilist-micro` *vocab micro*) ⇒ void
  vocab : zodiac:vocab
  micro : (zodiac:read zodiac:env zodiac:attr zodiac:vocab -¿ zodiac:parsed)

  `add-ilist-micro` installs the expander for an improper lists of tokens.

`zodiac:add-list-micro`

- (`zodiac:add-list-micro` *vocab micro*) ⇒ void
  vocab : zodiac:vocab
  micro : (zodiac:read zodiac:env zodiac:attr zodiac:vocab -¿ zodiac:parsed)

  `add-list-micro` installs the expander that handles a list of tokens,

`zodiac:add-lit-micro`

- (`zodiac:add-lit-micro` *vocab micro*) ⇒ void
  vocab : zodiac:vocab
  micro : (zodiac:read zodiac:env zodiac:attr zodiac:vocab -¿ zodiac:parsed)

  `add-lit-micro` installs the the expander for processing literals.

`zodiac:add-macro-form`

- (`zodiac:add-macro-form` *macro-name vocab macro*) ⇒ void
  macro-name : symbol
  vocab : zodiac:vocab
  macro : (zodiac:read zodiac:env -¿ zodiac:read)

[5] These correspond to the self-quoting objects in Scheme.

If a list of tokens is headed by a symbol for which a micro or macro has been defined, then the defined micro or macro is invoked; only otherwise is the micro for lists of tokens invoked.

`zodiac:add-micro-form`

- (`zodiac:add-micro-form` *micro-name vocab micro*) ⇒ void
  micro-name : symbol
  vocab : zodiac:vocab
  micro : (zodiac:read zodiac:env zodiac:attr zodiac:vocab -¿ zodiac:parsed)

  If a list of tokens is headed by a symbol for which a micro or macro has been defined, then the defined micro or macro is invoked; only otherwise is the micro for lists of tokens invoked.

`zodiac:add-sym-micro`

- (`zodiac:add-sym-micro` *vocab micro*) ⇒ void
  vocab : zodiac:vocab
  micro : (zodiac:read zodiac:env zodiac:attr zodiac:vocab -¿ zodiac:parsed)

  `add-sym-micro` installs the expander for individual symbols.

`zodiac:copy-vocabulary`

- (`zodiac:copy-vocabulary` *v*) ⇒ zodiac:vocab
  v : zodiac:vocab

  `copy-vocabulary` returns a new vocabulary that contains all the micros and macros contained in the given vocabulary.

  NOTE: `copy-vocabulary` literally makes a copy of the given vocabulary. Any changes made after the copy operation will not be seen by the copy. Thus, the copy should be made only when the programmer is certain the vocabulary being copied has all the appropriate contents.

`zodiac:make-vocabulary`

- (`zodiac:make-vocabulary`) ⇒ zodiac:vocab

  `make-vocabulary` creates a new vocabulary that contains no micros or macros. Any syntactic input parsed with it will result in a syntax error.

`zodiac:merge-vocabulary`

- (`zodiac:merge-vocabulary` *v1 v2*) ⇒ zodiac:vocab
  v1 : zodiac:vocab
  v2 : zodiac:vocab

  `merge-vocabulary` merges two vocabularies; the first argument is destructively updated by each of the entries in the second argument.

3.5.1 Linking

Zodiac has been written so that it can be used independently of the graphical components of DrScheme. Its only requirement is that it be run under MzScheme (or any other "sufficiently compatible" system). Thus, Zodiac can be used with tools both within and without DrScheme. Linking to Zodiac inside

DrScheme is done as part of the standard interface for DrScheme tools. This section describes how a tool linking directly to Zodiac should do so.

The code for Zodiac is found in the `zodiac` directory of the Rice PLT distribution (say this path is bound to `plt-home`). To load Zodiac into the system, use

```
(require-library "fileu.ss") ; to load load-recent
(load-recent (build-path plt-home "zodiac" "load"))
```

This will ensure all the files are loaded, and that the compiled versions are loaded where available and newer than their source. All the Zodiac signatures mentioned below are in the file `sigs.ss`.

Any unit wanting to use the Zodiac procedures must include the signature `zodiac:system^` among its `import`s. The unit `zodiac:system@`, which satisfies this signature, contains all the requisite code. Linking to `zodiac:system@` requires it be passed two parameters, in this order:

Error Interface Zodiac requires an implementation of the error handlers (Section **??**). Thus, a unit satisfying the signature `zodiac:interface^`, containing the error handlers that have the described types, must be provided. Zodiac provides a default unit with no imports, `zodiac:default-interface@`, that meets this signature, but those procedures will likely be unsatisfactory for most presentation needs. They are provided only to provide a template and to reduce the effort needed to start using Zodiac; users are strongly encouraged to replace them.

Language Parameters Zodiac takes several parameters that customize its language. These are listed in the signature `plt:parameters^` (from the file `sparams.ss` in the directory `lib` of the PLT distribution), and the settings for MzScheme are in the unit `plt:mzscheme-parameters@`. Invoking this latter unit with no arguments will yield the appropriate values, which can then be passed to Zodiac.

The implementation of `zodiac:default-interface@`, and a sample linkage, can be found in the file `invoke.ss`.

NOTE: It is suggested that users of Zodiac use the prefix mechanism while importing into a unit to prefix all Zodiac names. Since the system is not entirely documented, this will prevent unexpected name clashes (though if they should arise, the file `sigs.ss` should be consulted to see what names are exported). In addition, Zodiac provides different definitions for standard Scheme primitives such as `read` and `make-vector`. Mixing these values with traditional Scheme primitives will lead to confusion and, sometimes, insidious errors. Using a prefix helps the user clarify when a Zodiac primitive is desired and when the Scheme primitive should be used instead.

3.6 Pattern Matching

Since Zodiac is intended to serve as a platform for writing tools that process programs, it is invaluable to have a utility that syntactically validates and de-constructs input program phrases. Since Zodiac is currently geared toward processing Scheme programs, it currently includes a pattern-matching utility that processes Scheme s-expressions in their Zodiac-enriched forms (*i.e.*, embedded in the *read* type).

The pattern matcher in Zodiac is *procedural* in nature. This means that it does not define any macros or core forms; rather, patterns are defined and matched against using a series of procedure calls. A current area of investigation is into whether there is a reasonable syntactic interface that can be provided for these procedures and, if so, what that interface is.

This document describes Zodiac's pattern matcher and provides some examples of its use.

The pattern matcher includes a **pattern compiler**, which pre-processes patterns to generate efficient code that performs two tasks: to validate the input, and to bind **pattern variables** against the corresponding components of the input.

The pattern matcher introduces four new types: the keyword list, *kwd-list*; the (raw) pattern, *pat*; the compiled pattern, *cpat*; and the pattern environment, *penv*. For now, *kwd-list* is just a synonym for the type *list(scheme-symbol)*.

3.6.1 Examples

The source for `match-and-rewrite` is presented first:

```
(define match-and-rewrite
  (lambda (expr rewriter out kwd env)
    (let ((p-env (match-against rewriter expr env)))
      (and p-env
           (pexpand out p-env kwd))))))
```

This assumes that a compiled pattern has already been generated for use as the `rewriter` argument. A typical use might be:

```
(let* ((kwd '(let))
       (in-pattern '(let ((v e) ...) b))
       (out-pattern '((lambda (v ...) b) e ...))
       (m\&e (make-match\&env in-pattern-1 kwd)))
  (lambda (expr env)
    (or (match-and-rewrite expr m\&e out-pattern kwd env)
        (static-error expr "Malformed let"))))
```

This implements the `let` macro used by many Scheme implementations. Note that the compiled pattern, bound to `m&e`, is created outside the procedure representing the `let` macro.

```
  (let* ((kwd '(lambda))
         (in-pattern '(lambda args body))
         (m&e (make-match&env in-pattern kwd)))
    (lambda (expr env attributes vocab)
      (cond
        ((match-against m&e expr env)
         =>
         (lambda (p-env)
           (let ((args (pexpand 'args p-env kwd))
                 (body (pexpand 'body p-env kwd)))
             (make-lambda-form args body))))
        (else
         (static-error expr "Malformed lambda body")))))
```

In this example, a simplified version of the Scheme `lambda` expression is shown. Note that there is no checking done to ensure that `args` does indeed match against a well-formed argument list. After the pattern variables are expanded, the results are passed to the procedure `make-lambda-form`, which may represent an abstract syntax constructor.

`zodiac:make-match&env`

- (`zodiac:make-match&env` *kl*) ⇒ zodiac:cpat
  kl : (listof keyword)

  `make-match&env` is used to pre-compile patterns. Typically, the computation that compiles patterns will be hoisted out of procedure bodies so that the compilation takes place once while its result can be used several times.

`zodiac:match-against`

- (`zodiac:match-against` *pattern exp env*) ⇒ (union penv #f)
  pattern : zodiac:cpat
  exp : zodiac:read
  env : zodiac:env

  `match-against` performs the actual matching of a given expression (of type *read*) against a compiled pattern[6]. If the expression matches the pattern, a pattern environment, which is a non-false value, is returned; else the result is `#f`.

`zodiac:match-and-rewrite`

- (`zodiac:match-and-rewrite` *sexp pattern1 pattern2 keywords*) ⇒ (union mixed #f)
  sexp : zodiac:read
  pattern1 : zodiac:cpat
  pattern2 : zodiac:pat
  keywords : (list keyword)

  `match-and-rewrite` is used to provide a concise means of writing rewrite rules. It is particularly useful for writing source-to-source transformations (macros). In Section 3.6.1, we will show the source for this procedure.

`zodiac:pexpand`

- (`zodiac:pexpand` *pattern env keywords*) ⇒ mixed
  pattern : zodiac:pat
  env : zodiac:penv
  keywords : (listof keyword)

  `pexpand` expands patterns in the context of a pattern environment and a list of keywords. The first argument is recursively copied verbatim into the output unless an identifier is encountered that is bound in the pattern environment and is not in the keyword list; this identifier is replaced by its binding, which has type *read*, in the pattern environment, and transcription proceeds accordingly. The output of compositing `pexpand` with `zodiac:structurize-syntax` yields an object of type *read*, which can be subjected to further pattern matching, *etc.*

3.7 Core Scheme

The core portions of the Zodiac vocabulary that parse Scheme are found in the unit `zodiac:scheme-core@`, which satisfies the signature `zodiac:scheme-core^`. This document describes the Core Scheme unit.

The primary task of Core Scheme is to create a vocabulary, `scheme-vocabulary`, which will be built up on in the more advanced vocabularies, and to populate it with micros that handle the core behavior of Scheme. For instance, a list of tokens (not headed by a keyword) is treated as an application, an improper list is flagged as an error, and literals are quoted. Vocabularies are provided for parsing argument lists with and without optional initial values. Predicates are provided for determining the syntactic validity of argument lists. The rest of this document describes Core Scheme in detail.

3.7.0.1 Vocabularies

`scheme-vocabulary` is intended to contain all the micros and macros that parse Scheme programs. It is

[6] The environment is provided to determine whether a keyword has been lexically shadowed.

initially populated with micros for handling the different syntactic categories; all list objects are treated as applications, awaiting further layers of Scheme to add the various core forms in the language.

`arglist-decls-vocab` is used to parse argument lists such as those of abstractions. The syntax of arguments accepted is controlled by the language level at which Zodiac is being used.

3.7.0.2 Types

The *parsed* type is used to represent the output from the parser. All *parsed* objects have a back field, which is used to convey information between program processing tools such as analyzers and monitors:

zodiac:parsed (back)

3.7.0.3 Expressions

Any expression is either a variable reference, an application or a special form:

zodiac:varref **:** *parsed* (var)
zodiac:app **:** *parsed* (fun args)
zodiac:form **:** *parsed* ()

The var field of a *varref* is a Scheme symbol. The fun field of *app* is of type *parsed*, while args contains a list of *parsed*. All the special forms — which are defined in other documents — are sub-types of *form*.

3.7.0.4 Identifiers and Binding

The Core Scheme unit recognizes that identifiers may be free or (lexically) bound. To accomodate additional binding forms, a distinction is first drawn between free and bound variables:

zodiac:top-level-varref **:** *varref* ()
zodiac:bound-varref **:** *varref* (binding)

One sub-type of the latter is also defined:

zodiac:lexical-varref **:** *bound-varref* ()

The binding field of a *bound-varref* refers to an object of type *binding*, with a *lexical-varref* referring to an object of type *lexical-binding*:

zodiac:binding **:** *parsed* (var orig-name)
zodiac:lexical-binding **:** *binding* ()

The var field contains a Scheme symbol representing the name of the bound identifier. Since hygienic renaming may have taken place, the orig-name field holds the original name (which may have been provided in the source, or been introduced via a macro or micro).

The binding field may be used to distinguish between bound variables in that exactly all occurrences of the same bound identifier contain the same value in their binding field (in the sense of `eq?`).

Note: There is no justification for *binding* to be a sub-type of *parsed*; this dependency will be elided.

Index