

PLT Tools: DrScheme Extension Manual

PLT
scheme@cs.rice.edu

Version 101
October 1999

Copyright notice

Copyright ©1996-99 PLT, Rice University

Permission to make digital/hard copies and/or distribute this documentation for any purpose is hereby granted without fee, provided that the above copyright notice, author, and this permission notice appear in all copies of this documentation.

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	1
1.1	Common Tools Abstractions	1
1.1.1	Evaluation	1
1.1.2	Getting the same menu items as DrScheme	2
2	Tools Reference	3
2.1	<code>drscheme:frame:basics<%></code>	3
2.2	<code>drscheme:frame:basics-mixin</code>	3
2.3	<code>drscheme:rep:text%</code>	5
2.4	<code>drscheme:unit:definitions-canvas%</code>	7
2.5	<code>drscheme:unit:frame%</code>	7
2.6	<code>drscheme:unit:interactions-canvas%</code>	10
2.7	Userspace	10
2.8	Processing Programs with Zodiac	15
2.9	Extending the DrScheme Classes	16
2.10	Libraries	18
2.11	Help Desk	18
3	Zodiac Reference	19
3.1	Zodiac	19
3.1.1	Motivation	19
3.1.2	Notations and Terminology	19
3.1.3	Core of McMicMac	20
3.1.4	Scanner	20
3.1.5	Reader	20

3.1.6	Parser	26
3.2	Environments	29
3.3	Attributes	30
3.4	Vocabulary	30
3.4.1	Linking	32
3.5	Pattern Matching	33
3.5.1	Examples	33
3.6	Core Scheme	35
	Index	37

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^` and `mzlib:print-convert^` (defined in the `MzLib`, §15 in *PLT MzScheme: Language Manual*), `drscheme:export^` (defined below), and `zodiac:system^`. The `zodiac:system^` signature contains all of the names in section 3.1.

The `drscheme:export^` signature contains the parameters defined in the parameters section 2.9, and the other classes described in the next subsections.

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 mix in `frame:standard-menus-mixin` and `drscheme:frame:basics-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`) \Rightarrow 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`) \Rightarrow 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:rep:text%

User submitted evaluations in DrScheme are evaluated asynchronously.

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.

- (make-object drscheme:rep:text% *line-spacing* *tabstops*) \Rightarrow drscheme:rep:text% object
line-spacing = 1.0 : non-negative real number
tabstops = null : list of real numbers

The *line-spacing* argument sets the additional amount of space (in DC units) inserted between each line in the editor when the editor is displayed. This spacing is included in the reported height of each line.

See `set-tabs` for information about *tabstops*.

A new `keymap%` object is created for the new editor. See also `get-keymap` and `set-keymap`.

A new `style-list` object is created for the new editor. See also `get-style-list` and `set-style-list`.

break

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

- (send *a-drscheme:rep:text* break) \Rightarrow void
This method breaks the evaluation thread.

display-result

- (send *a-drscheme:rep:text* display-result *v*) \Rightarrow void
v : any scheme value
This displays the result of a computation in the bottom window.

do-many-buffer-evals

This function evaluates all of the expressions in a buffer.

- (send *a-drscheme:rep:text* do-many-buffer-evals *text* *start* *end*) \Rightarrow void
text : a text% object
start : int
end : int
It evaluates all of the expressions in *text* starting at *start* and ending at *end*.

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*) \Rightarrow 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.

initialize-console

- (send a-drscheme:rep:text initialize-console) \Rightarrow void

This inserts the “Welcome to DrScheme” message into the interactions buffer.

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) \Rightarrow 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 highlight 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) \Rightarrow 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, write this:

```
(class ...
  (inherit user-thread run-in-evaluation-thread)
  (rename [super-reset-console reset-console])
  (public
    [reset-console
      (lambda ()
        (super-reset-console)
        (run-in-evaluation-thread
          (lambda ()
            (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.

- (send a-drscheme:rep:text run-in-evaluation-thread f) \Rightarrow void
f : (\rightarrow void)

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

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 current 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)

2.4 drscheme:unit:definitions-canvas%

Superclass: `editor-canvas%`

Initializes the visibility of the save button.

- (make-object drscheme:unit:definitions-canvas% parent editor style scrolls-per-page) ⇒ drscheme:unit:definitions-canvas% 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`.

2.5 drscheme:unit:frame%

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 managers), 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 resizable 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 exclusive with `'mdi-parent` (Windows)
- `'mdi-parent` — creates the frame as a MDI (multiple document interface) parent frame, mutually exclusive 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.

`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`

`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` parameter, 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) \Rightarrow a text% object.

disable-evaluation

This method is called to disable evaluation when some evaluation is taking place. See also `enable-evaluation`

- (send *a-drscheme:unit:frame* disable-evaluation) \Rightarrow void
Disables the execute button, the interactions window, and the definitions window.

enable-evaluation

this method is called to enable evaluation after some evaluation has taken place. See also `disable-evaluation`

- (send *a-drscheme:unit:frame* enable-evaluation) \Rightarrow void
Enables the execute button, the interactions window, and the definitions window.

ensure-interactions-shown

Call this method when you want to be sure that the interactions window is shown. If the interactions window is not shown, this method will show it.

- (send *a-drscheme:unit:frame* ensure-interactions-shown) \Rightarrow void

execute-callback

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

- (send *a-drscheme:unit:frame* execute-callback) \Rightarrow void
It calls `ensure-interactions-shown` and then it calls `do-many-buffer-evals` passing in the `interactions-text` and its entire range.

get-text-to-search

Override this method to specify which text to search.

- (send *a-drscheme:unit:frame* get-text-to-search) \Rightarrow 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) \Rightarrow 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) \Rightarrow a drscheme:rep:text% object.

make-searchable

- (send *a-drscheme:unit:frame* make-searchable *canvas*) \Rightarrow 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) \Rightarrow void
 Updates the interactions and definitions windows based on the contents of the menus.

2.6 drscheme:unit:interactions-canvas%

- (make-object drscheme:unit:interactions-canvas% *parent editor style scrolls-per-page*) \Rightarrow drscheme:unit:interactions-canvas% 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 Userspace

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 elements 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: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: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?*)
 \Rightarrow 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 initial 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*) \Rightarrow setting
setting : setting

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

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

Gets the default setting's name.

`drscheme:basis:initialize-parameters`

- (`drscheme:basis:initialize-parameters` *custodian* *setting*) \Rightarrow 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 with 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 replacement only happens in the teaching language levels, (Beginner, Intermediate and Advanced). For more details see **plt/collects/userspace/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) (-i void) -i 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 receives 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*) \Rightarrow 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?*) \Rightarrow 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* receives 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*) \Rightarrow boolean
setting : setting

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

`drscheme:basis:setting-name`

- (`drscheme:basis:setting-name` *setting*) \Rightarrow string
setting : setting

Returns the name of the *setting*.

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

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

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

`drscheme:basis:zodiac-vocabulary?`

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

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

settings

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

- *a-settings* \Rightarrow (list-of setting)

2.8 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*) \Rightarrow void
filename : string
f : ((+ process-finish sexp zodiac:parsed) (-i void) -i 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*) \Rightarrow void
sexp : sexp
f : ((+ process-finish sexp zodiac:parsed) (-i void) -i void)

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

drscheme:basis:process/no-zodiac

- (drscheme:basis:process/no-zodiac *reader* *f*) \Rightarrow void
reader : (-i (+ eof sexp))
f : ((+ sexp drscheme:basis:process-finish) (-i void) -i 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?*) \Rightarrow void
reader : (-i zodiac:sexp)
f : ((+ drscheme:basis:process-finish sexp zodiac:parsed) (-i void) -i void)
annotate? : boolean

This function is used to process a program with McMicMac (see *PLT McMicMac: Parser Manual*). 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:rep:process-text/no-zodiac`

- (`drscheme:rep:process-text/no-zodiac text f start end`) \Rightarrow void
 - text* : a text% object
 - f* : ((+ process-finish sexp zodiac:parsed) (-i void) -i void)
 - start* : int
 - end* : int

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

`drscheme:rep:process-text/zodiac`

- (`drscheme:rep:process-text/zodiac text f start end annotate?`) \Rightarrow void
 - text* : a text% object
 - f* : ((+ process-finish sexp zodiac:parsed) (-i void) -i void)
 - start* : int
 - end* : int
 - annotate?* : boolean

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

`interface:mark-key`

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

- (`interface:mark-key`) \Rightarrow symbol
 - Gets the value of the parameter
- (`interface:mark-key new-key`) \Rightarrow 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`) \Rightarrow void
 - phase* : (union 'reader 'expander #f)
 - Sets the phase to *phase*.

2.9 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`.

Each of these names actually has a percent character appended onto the end of it. Those are not shown here.

`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*) \Rightarrow 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*) \Rightarrow 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*) \Rightarrow 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*) \Rightarrow 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*) \Rightarrow void
frame-mixin : a procedure that accepts a class and produces a class derived from it.

2.10 Libraries

`drscheme:rep:invoke-teachpack`

This function extends the current namespace with the definitions in the teachpack, if any teachpack is set. If not, it does nothing.

- `(drscheme:rep:invoke-teachpack) ⇒ void`

2.11 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

3.1.1 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 **McMicMac** 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 **McMicMac**.) Separate documents will describe the default abstract syntaxes provided with **McMicMac**.

3.1.2 Notations and Terminology

These documents assume a strong familiarity with MzScheme. In particular, the implementation of **McMicMac** 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.

3.1.3 Core of McMicMac

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

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:

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:

period (location)

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

3.1.4 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:

eof (location)

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

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.

token : *scanned* (object type)

The object and type fields will be documented later.

3.1.5 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:

read : *zodiac* (object)

¹Rhymes with “dead”, “head”, “routinely bled” and “positively fed”.

The reader's output is sub-divided into scalar and sequence objects²:

```
scalar : read ()
sequence : read (length)
```

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

```
zodiac:string : read ()
zodiac:boolean : read ()
zodiac:number : read ()
zodiac:symbol : read (orig-name marks)
zodiac:char : read ()
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 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-expansion³.

3.1.5.1 ARGUMENT LISTS

Argument lists are encapsulated within a structure:

```
arglist (vars)
```

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:

```
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

²Strings are classified as scalar objects.

³There is currently no clean way of hiding this detail from the user of *McMicMac*; elucidation on this is forthcoming.

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

- (`zodiac:distinct-valid-id/s?`) \Rightarrow void
UNDOCUMENTED

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

- (`zodiac:distinct-valid-syntactic-id/s?`) \Rightarrow void
UNDOCUMENTED

`zodiac:expand-expr`

- (`zodiac:expand-expr read env attrib vocab`) \Rightarrow `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`) \Rightarrow void
UNDOCUMENTED

`zodiac:generate-name`

- (`zodiac:generate-name`) \Rightarrow void
UNDOCUMENTED

`zodiac:in-lexically-extended-env`

- (`zodiac:in-lexically-extended-env`) \Rightarrow void
UNDOCUMENTED

`zodiac:internal-error`

- (`zodiac:internal-error zodiac format`) \Rightarrow 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 McMicMac 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<=?)` \Rightarrow void
- UNDOCUMENTED

`zodiac:language>=?`

- `(zodiac:language>=?)` \Rightarrow void
- UNDOCUMENTED

`zodiac:lexically-resolved?`

- `(zodiac:lexically-resolved?)` \Rightarrow void
- UNDOCUMENTED

`zodiac:make-argument-list`

- `(zodiac:make-argument-list)` \Rightarrow void
- UNDOCUMENTED

`zodiac:make-empty-back-box`

- `(zodiac:make-empty-back-box)` \Rightarrow void
- UNDOCUMENTED

`zodiac:make-optargument-list`

- `(zodiac:make-optargument-list)` \Rightarrow void
- UNDOCUMENTED

`zodiac:marks-equal?`

- `(zodiac:marks-equal?)` \Rightarrow void
- UNDOCUMENTED

`zodiac:name-eq?`

- `(zodiac:name-eq?)` \Rightarrow void
- UNDOCUMENTED

`zodiac:optarglist-decls-vocab`

- `(zodiac:optarglist-decls-vocab)` \Rightarrow 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*) ⇒ (-i (union read eof))
input = (current-input-port) : (union input-port (-i 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:symbols*, the value in the object field, not in the orig-name field, is used.

`zodiac:static-error`

- (`zodiac:static-error` *zodiac format*) \Rightarrow 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*) \Rightarrow `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:symbols* and *zodiac:list* forms their initial set of marks. Ordinary users may ignore this argument.

`zodiac:syntax-car`

- (`zodiac:syntax-car` *sexp*) \Rightarrow `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*) \Rightarrow `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*) \Rightarrow B

```
f : (union (TST TST -i B) (TST -i 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`.

3.1.6 Parser

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

3.1.6.1 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 identifier. The elements of the list are of `zodiac:top-level-varref/binds`. 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*.

3.1.6.2 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*.

```
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 3.1.6.1).

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

type is a (McMicMac) *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.

3.1.6.3 UNITS

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

imports is a list of *zodiac:lexical-bindings*. *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 (McMicMac) *zodiac:symbol*. The first projection corresponds to the internal name, and the second projection to the

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

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-bindings*. link is a list of lists. Each list corresponds to one link clause. The **car** of the list is a (McMicMac) *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:symbols* 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 (McMicMac) *zodiac:symbol* object, giving the specified name prefix.

3.1.6.4 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:symbols*.

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 3.1.6.1) and inst-clauses is a list of body clauses (see section 3.1.6.4).

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 (McMicMac) *zodiac:symbols*; internals is a list of the appropriate kind of *zodiac:bindings*; exprs is a list of *zodiac:parsed* expressions; and imports is a list of (McMicMac) *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)

`zodiac:syntax-null?`

- (`zodiac:syntax-null? sexp`) \Rightarrow `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?`) \Rightarrow void
 UNDOCUMENTED

`zodiac:valid-id?`

- (`zodiac:valid-id?`) \Rightarrow void
 UNDOCUMENTED

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

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

`zodiac:valid-syntactic-id?`

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

3.2 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.

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

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

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

Languages implemented atop McMicMac 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-resolutions*.

zodiac:extend-env

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

zodiac:resolve

- (zodiac:resolve *id env vocab*) \Rightarrow zodiac:env-entry
id : id
env : zodiac:env
vocab : zodiac:vocab

zodiac:retract-env

- (zodiac:retract-env *retraction env*) \Rightarrow void
retraction : (list-of new-vars)
env : zodiac:env

3.3 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*) \Rightarrow (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) \Rightarrow zodiac:attr
make-attributes creates a new (empty) table of attributes.

zodiac:put-attribute

- (zodiac:put-attribute *attr key value*) \Rightarrow 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.

3.4 Vocabulary

McMicMac 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 McMicMac.

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`) \Rightarrow void
`vocab` : `zodiac:vocab`
`micro` : (`zodiac:read zodiac:env zodiac:attr zodiac:vocab -i zodiac:parsed`)
- `add-ilist-micro` installs the expander for an improper lists of tokens.

`zodiac:add-list-micro`

- (`zodiac:add-list-micro vocab micro`) \Rightarrow void
`vocab` : `zodiac:vocab`
`micro` : (`zodiac:read zodiac:env zodiac:attr zodiac:vocab -i zodiac:parsed`)
- `add-list-micro` installs the expander that handles a list of tokens,

`zodiac:add-lit-micro`

- (`zodiac:add-lit-micro vocab micro`) \Rightarrow void
`vocab` : `zodiac:vocab`
`micro` : (`zodiac:read zodiac:env zodiac:attr zodiac:vocab -i zodiac:parsed`)
- `add-lit-micro` installs the the expander for processing literals.

`zodiac:add-macro-form`

- (`zodiac:add-macro-form macro-name vocab macro`) \Rightarrow void
`macro-name` : symbol
`vocab` : `zodiac:vocab`
`macro` : (`zodiac:read zodiac:env -i zodiac:read`)

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`) \Rightarrow void
`micro-name` : symbol
`vocab` : `zodiac:vocab`
`micro` : (`zodiac:read zodiac:env zodiac:attr zodiac:vocab -i 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.

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

zodiac:add-sym-micro

- (zodiac:add-sym-micro vocab micro) \Rightarrow void
 vocab : zodiac:vocab
 micro : (zodiac:read zodiac:env zodiac:attr zodiac:vocab -j zodiac:parsed)

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

zodiac:copy-vocabulary

- (zodiac:copy-vocabulary v) \Rightarrow 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) \Rightarrow 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) \Rightarrow 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.

3.4.1 Linking

McMicMac 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, McMicMac can be used with tools both within and without DrScheme. Linking to McMicMac inside DrScheme is done as part of the standard interface for DrScheme tools. This section describes how a tool linking directly to McMicMac should do so.

The code for McMicMac is found in the **zodiac** directory of the Rice PLT distribution (say this path is bound to **plt-home**). To load McMicMac 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 McMicMac signatures mentioned below are in the file **sigs.ss**.

Any unit wanting to use the McMicMac procedures must include the signature **zodiac:system^** among its imports. 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 `McMicMac` 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. `McMicMac` 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 `McMicMac`; users are strongly encouraged to replace them.

Language Parameters `McMicMac` 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 `McMicMac`.

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 `McMicMac` use the prefix mechanism while importing into a unit to prefix all `McMicMac` names. Since the system is not entirely documented, this will prevent unexpected name clashes (though if they should arise, the file `signs.ss` should be consulted to see what names are exported). In addition, `McMicMac` 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 `McMicMac` primitive is desired and when the Scheme primitive should be used instead.

3.5 Pattern Matching

Since `McMicMac` 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 `McMicMac` is currently geared toward processing Scheme programs, it currently includes a pattern-matching utility that processes Scheme s-expressions in their `McMicMac`-enriched forms (*i.e.*, embedded in the `read` type).

The pattern matcher in `McMicMac` 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 `McMicMac`'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)*.

3.5.1 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 pattern⁶. If the expression matches the pattern, a pattern environment, which is a non-false value, is returned; else the result is `#f`.

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

zodiac:match-and-rewrite

- (**zodiac:match-and-rewrite** *sexp pattern1 pattern2 keywords*) \Rightarrow (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 3.5.1, we will show the source for this procedure.

zodiac:pexpand

- (**zodiac:pexpand** *pattern env keywords*) \Rightarrow 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 **structurize-syntax** yields an object of type *read*, which can be subjected to further pattern matching, *etc.*

3.6 Core Scheme

The core portions of the McMicMac 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.

3.6.0.1 VOCABULARIES

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

3.6.0.2 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:

parsed (back)

3.6.0.3 EXPRESSIONS

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

varref : *parsed* (var)
app : *parsed* (fun args)
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*.

3.6.0.4 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:

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

One sub-type of the latter is also defined:

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*:

binding : *parsed* (var orig-name)
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

break, 5
button-panel, 8

canvas
 scroll bars, 7, 10
current-setting, 11

definitions-canvas, 8
definitions-text, 8
disable-evaluation, 9
display-result, 5
do-many-buffer-evals, 5
drscheme:basis:add-setting, 11
drscheme:basis:copy-setting, 11
drscheme:basis:current-vocabulary, 11
drscheme:basis:find-setting-named, 11
drscheme:basis:format-source-loc, 11
drscheme:basis:get-default-setting, 12
drscheme:basis:get-default-setting-name, 12
drscheme:basis:initialize-parameters, 12
drscheme:basis:number->setting, 13
drscheme:basis:process-file/no-zodiac, 15
drscheme:basis:process-file/zodiac, 13
drscheme:basis:process-finish?, 14
drscheme:basis:process-sexp/no-zodiac, 15
drscheme:basis:process-sexp/zodiac, 14
drscheme:basis:process/no-zodiac, 15
drscheme:basis:process/zodiac, 15
drscheme:basis:r4rs-style-printing?, 14
drscheme:basis:setting-name, 14
drscheme:basis:setting-name->number, 14
drscheme:basis:zodiac-vocabulary?, 14
drscheme:frame:basics-mixin, 3
drscheme:frame:basics<%>, 3
drscheme:get/extend:extend-definitions-canvas, 17
drscheme:get/extend:extend-definitions-text, 17
drscheme:get/extend:extend-interactions-canvas, 17
drscheme:get/extend:extend-interactions-text, 17
drscheme:get/extend:extend-unit-frame, 17
drscheme:help-desk:help-desk, 18
drscheme:help-desk:open-url, 18
drscheme:help-desk:open-users-url, 18
drscheme:rep:invoke-teachpack, 18
drscheme:rep:process-text/no-zodiac, 16
drscheme:rep:process-text/zodiac, 16

drscheme:rep:text%, 5
drscheme:setting, 5, 11
drscheme:unit:definitions-canvas%, 7
drscheme:unit:frame%, 7
drscheme:unit:interactions-canvas%, 10

editor-canvas%, 7
enable-evaluation, 9
ensure-interactions-shown, 9
execute-callback, 9

file-menu:between-open-and-revert, 3
file-menu:new, 3
file-menu:new-string, 3
file-menu:open, 4
file-menu:open-string, 4
format-source-loc, 5

get-text-to-search, 9

help-menu:about, 4
help-menu:about-string, 4
help-menu:after-about, 4
'hide-hscroll, 7, 10
'hide-vscroll, 7, 10

initialize-console, 6
interactions-canvas, 9
interactions-text, 10
interface:mark-key, 16
interface:set-zodiac-phase, 16

keymaps
 in an editor, 5

make-searchable, 10
'mdi-child, 7
'mdi-parent, 7

'no-caption, 7
'no-hscroll, 7, 10
'no-resize-border, 7
'no-system-menu, 7
'no-vscroll, 7, 10

report-error, 6
reset-console, 6
run-in-evaluation-thread, 6

settings, 15
style lists

in an editor, 5
 update-shown, 10
 user-thread, 7
 vocabularies, 30
 zodiac:add-ilist-micro, 31
 zodiac:add-list-micro, 31
 zodiac:add-lit-micro, 31
 zodiac:add-macro-form, 31
 zodiac:add-micro-form, 31
 zodiac:add-sym-micro, 32
 zodiac:arglist-decls-vocab, 21
 zodiac:arglist-pattern, 21
 zodiac:copy-vocabulary, 32
 zodiac:distinct-valid-id/s?, 22
 zodiac:distinct-valid-syntactic-id/s?, 22
 zodiac:expand-expr, 22
 zodiac:extend-env, 30
 zodiac:extend-parsed->raw, 22
 zodiac:generate-name, 22
 zodiac:get-attribute, 30
 zodiac:in-lexically-extended-env, 22
 zodiac:internal-error, 22
 zodiac:language<=?, 23
 zodiac:language>=?, 23
 zodiac:lexically-resolved?, 23
 zodiac:make-argument-list, 23
 zodiac:make-attributes, 30
 zodiac:make-empty-back-box, 23
 zodiac:make-match&env, 34
 zodiac:make-optargument-list, 23
 zodiac:make-vocabulary, 32
 zodiac:marks-equal?, 23
 zodiac:match-against, 34
 zodiac:match-and-rewrite, 35
 zodiac:merge-vocabulary, 32
 zodiac:name-eq?, 23
 zodiac:optarglist-decls-vocab, 23
 zodiac:optarglist-pattern, 24
 zodiac:parsed->raw, 24
 zodiac:pexpand, 35
 zodiac:put-attribute, 30
 zodiac:read, 24
 zodiac:resolve, 30
 zodiac:retract-env, 30
 zodiac:scheme-expand, 24
 zodiac:scheme-vocabulary, 24
 zodiac:sexp->raw, 24
 zodiac:static-error, 25
 zodiac:structurize-syntax, 25
 zodiac:syntax-car, 25
 zodiac:syntax-cdr, 25
 zodiac:syntax-map, 25
 zodiac:syntax-null?, 29
 zodiac:valid-id/s?, 29
 zodiac:valid-id?, 29
 zodiac:valid-syntactic-id/s?, 29
 zodiac:valid-syntactic-id?, 29