

# **DrScheme: PLT Programming Environment**

Version 4.1.5

Robert Bruce Findler  
and PLT Scheme

March 21, 2009

DrScheme is a graphical environment for developing programs using the PLT Scheme programming languages.

# Contents

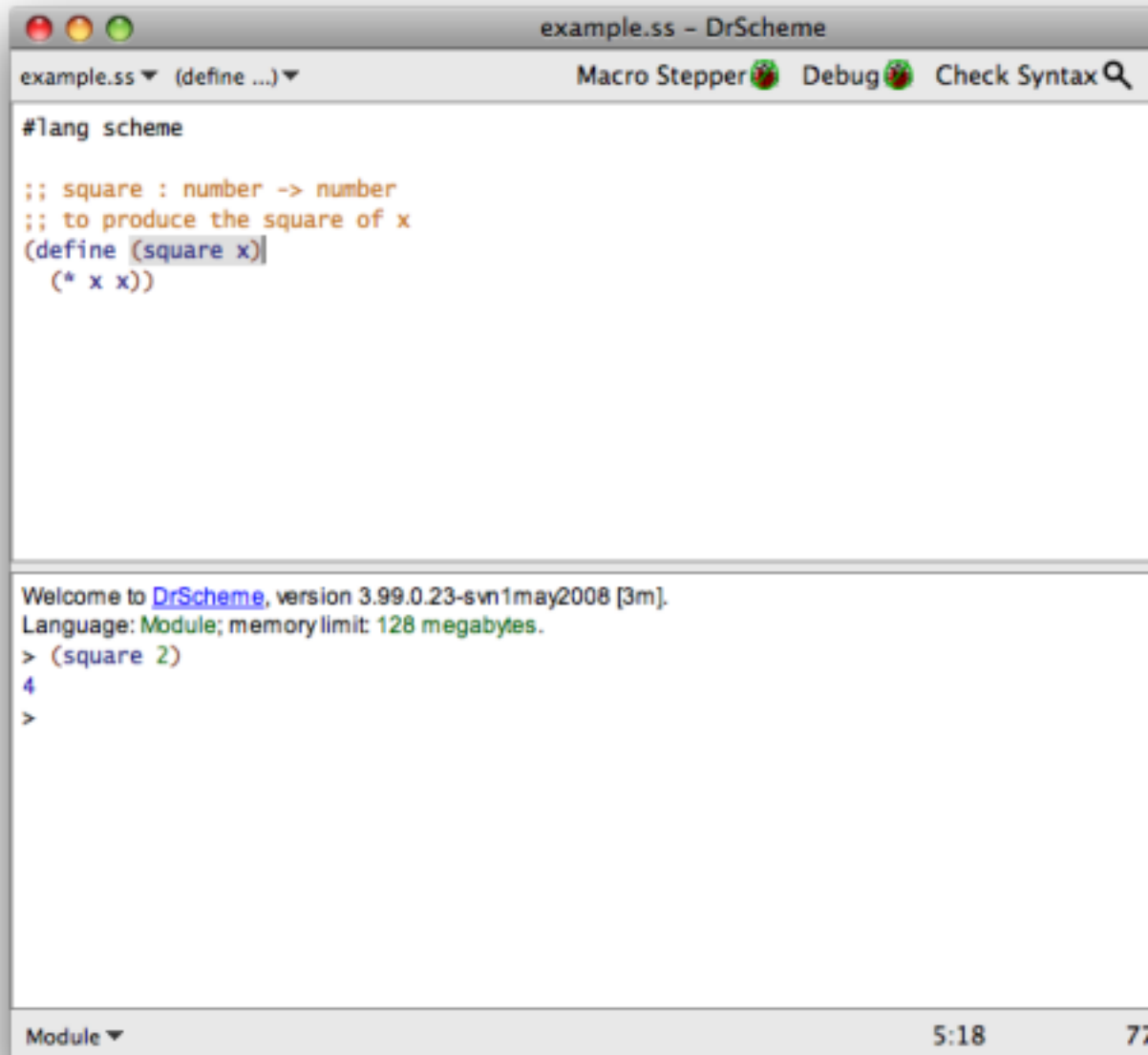
<b>1</b>	<b>Interface Essentials</b>	<b>5</b>
1.1	Buttons . . . . .	7
1.2	Choosing a Language . . . . .	8
1.3	Editing with Parentheses . . . . .	9
1.4	Searching . . . . .	10
1.5	Tabbed Editing . . . . .	11
1.6	The Interactions Window . . . . .	11
1.6.1	Errors . . . . .	12
1.6.2	Input and Output . . . . .	12
1.7	Graphical Syntax . . . . .	15
1.7.1	Images . . . . .	15
1.7.2	XML Boxes and Scheme Boxes . . . . .	16
1.8	Graphical Debugging Interface . . . . .	16
1.8.1	Debugger Buttons . . . . .	16
1.8.2	Definitions Window Actions . . . . .	17
1.8.3	Stack View Pane . . . . .	18
1.8.4	Debugging Multiple Files . . . . .	19
1.9	Creating Executables . . . . .	19
<b>2</b>	<b>Languages</b>	<b>21</b>
2.1	Modules . . . . .	21
2.2	Legacy Languages . . . . .	21
2.3	<i>How to Design Programs</i> Teaching Languages . . . . .	21
2.4	ProfessorJ . . . . .	25

2.5	Other Experimental Languages . . . . .	26
2.6	Output Printing Styles . . . . .	26
<b>3</b>	<b>Interface Reference</b>	<b>28</b>
3.1	Menus . . . . .	28
3.1.1	File . . . . .	28
3.1.2	Edit . . . . .	29
3.1.3	View . . . . .	30
3.1.4	Language . . . . .	31
3.1.5	Scheme . . . . .	32
3.1.6	Insert . . . . .	33
3.1.7	Windows . . . . .	34
3.1.8	Help . . . . .	34
3.2	Preferences . . . . .	34
3.2.1	Font . . . . .	34
3.2.2	Colors . . . . .	34
3.2.3	Editing . . . . .	35
3.2.4	Warnings . . . . .	36
3.2.5	Profiling . . . . .	37
3.2.6	Browser . . . . .	37
3.3	Keyboard Shortcuts . . . . .	37
3.3.1	Moving Around . . . . .	38
3.3.2	Editing Operations . . . . .	39
3.3.3	File Operations . . . . .	40
3.3.4	Search . . . . .	40
3.3.5	Miscellaneous . . . . .	40

3.3.6	Interactions . . . . .	41
3.3.7	LaTeX and TeX inspired keybindings . . . . .	41
3.3.8	Defining Custom Shortcuts . . . . .	44
3.4	DrScheme Files . . . . .	45
3.4.1	Program Files . . . . .	45
3.4.2	Backup and Autosave Files . . . . .	45
3.4.3	Preference Files . . . . .	46
<b>4</b>	<b>Extending DrScheme</b>	<b>47</b>
4.1	Teachpacks . . . . .	47
4.2	Environment Variables . . . . .	48
	<b>Index</b>	<b>50</b>

# 1 Interface Essentials

The DrScheme window has three parts: a row of buttons at the top, two editing panels in the middle, and a status line at the bottom.



The top editing panel, called the *definitions window*, is for defining programs. The above figure shows a program that defines the function `square`.

The bottom panel, called the *interactions window*, is for evaluating Scheme expressions

interactively. The Language line in the interactions window indicates which primitives are available in the definitions and interactions windows. In the above figure, the language is Module.

Clicking the Run button evaluates the program in the definitions window, making the program's definitions available in the interactions window. Given the definition of `square` as in the figure above, typing `(square 2)` in the interactions window produces the result 4.

The *status line* at the bottom of DrScheme's window provides information about the current line and position of the editing caret, whether the current file can be modified, and whether DrScheme is currently evaluating any expression. The recycling icon flashes while DrScheme is "recycling" internal resources, such as memory.

The interactions window is described further in §1.6 "The Interactions Window", later in this manual.

## 1.1 Buttons

The left end of the row of buttons in DrScheme contains a miniature button with the current file's name. Clicking the button opens a menu that shows the file's full pathname. Selecting one of the menu entries produces an open-file dialog starting in the corresponding directory.

Below the filename button is a (define ...) button for a popup menu of names that are defined in the definitions window. Selecting an item from the menu moves the blinking caret to the corresponding definition.

The Save button appears whenever the definitions window is modified. Clicking the button saves the contents of the definitions window to a file. The current name of the file appears to the left of the Save button, but a file-selection dialog appears if the file has never been saved before.

The Step button—which appears only for the *How to Design Programs* teaching languages Beginning Student through Intermediate Student with Lambda—starts the Stepper, which shows the evaluation of a program as a series of small steps. Each evaluation step replaces an expression in the program with an equivalent one using the evaluation rules of DrScheme. For example, a step might replace `(+ 1 2)` with `3`. These are the same rules used by DrScheme to evaluate a program. Clicking Step opens a new window that contains the program from the definitions window, plus several new buttons: these buttons allow navigation of the evaluation as a series of steps.

The Debug button—which does *not* appear for the *How to Design Programs* teaching languages—starts a more conventional stepping debugger. It runs the program in the definitions window like the Run button, but also opens a debugging panel with several other buttons that provide control over the program's execution.

The debugging interface is described further in §1.8 "Graphical Debugging Interface", later in this manual.

Clicking the Check Syntax button annotates the program text in the definitions window. It adds the following annotations:

- **Syntactic Highlighting:** Imported variables and locally defined variables are highlighted with color changes. Documented identifiers are hyperlinked (via a right-click) to the documentation page.
- **Lexical Structure:** The lexical structure is shown with arrows overlaid on the program text. When the mouse cursor passes over a variable, DrScheme draws an arrow from the binding location to the variable, or from the binding location to every bound occurrence of the variable.

In addition to indicating definite references with blue arrows, DrScheme also draws arrows to indicate potential references within macro definitions. Potential arrows are drawn in purple and annotated with a question mark to indicate uncertainty, because DrScheme cannot predict how such identifiers will eventually be used. Their roles may depend on the arguments to the macro and the context the macro is used in.

Additionally, right-clicking (or Control-clicking under Mac OS X) on a variable activates a popup menu that lets you jump from binding location to bound location and vice-versa,  $\alpha$ -rename the variable, or tack the arrows so they do not disappear.

- **Tail Calls:** Any sub-expression that is (syntactically) in tail-position with respect to its enclosing context is annotated by drawing a light purple arrow from the tail expression to its surrounding expression.
- **Require Annotations:** Right-clicking (or Control-clicking under Mac OS X) on the argument to `require` activates a popup menu that lets you open the file that contains the required module.

Passing the mouse cursor over a `require` expression inside a module shows all of the variables that are used from that `require` expression. Additionally, if no variables are used from that `require` expression, it is colored like an unbound variable.

Finally, passing the mouse cursor over a variable that is imported from a module shows the module that it is imported from in a status line at the bottom of the frame.

The Run button evaluates the program in the definitions window and resets the interactions window.

The Break button interrupts an evaluation, or beeps if DrScheme is not evaluating anything. For example, after clicking Run or entering an expression into the interactions window, click Break to cancel the evaluation. Click the Break button once to try to interrupt the evaluation gracefully; click the button twice to kill the evaluation immediately.

## 1.2 Choosing a Language

DrScheme supports multiple dialects of Scheme, as well as some non-Scheme languages. You specify a language in one of two ways:

- Select the Language|Choose Language... menu item, and choose a language other than Module. After changing the language, click Run to reset the language in the interactions window. The bottom-left corner of DrScheme’s main window also has a shortcut menu item for selecting previously selected languages.
- Select the Module language (via the Language|Choose Language... menu item), and then specify a specific language as part of the program usually by starting the definitions-window content with `#lang`.

The latter method, Module with `#lang`, is the recommend mode, and it is described further in §2.1 “Modules”.

The Language|Choose Language... dialog contains a Show Details button for configuring certain details of the chosen language. Whenever the selected options do not match the default language specification, a Custom indicator appears next to the language-selection control at the top of the dialog.

See §2 “Languages” (later in this manual) for more information on the languages that DrScheme supports.

### 1.3 Editing with Parentheses

In Scheme mode, especially, DrScheme’s editor provides special support for managing parentheses in a program. When the blinking caret is next to a parenthesis, DrScheme shades the region between the parenthesis and its matching parenthesis. This feature is especially helpful when for balancing parentheses to complete an expression.

Although whitespace is not significant in Scheme, DrScheme encourages a particular format for Scheme code. When you type Enter or Return, the editor inserts a new line and automatically indents it. To make DrScheme re-indent an existing line, move the blinking caret to the line and hit the Tab key. (The caret can be anywhere in the line.) You can re-indent an entire region by selecting the region and typing Tab.

DrScheme also rewrites parenthesis as you type them, in order to make them match better. If you type a closing parenthesis `)`, a closing square bracket `]`, or a closing curly brace `}`, and if DrScheme can match it back to some earlier opening parenthesis, bracket, or brace, then DrScheme changes what you type to match. DrScheme also rewrites open square brackets, usually to an open parenthesis. There are some exceptions where opening square brackets are not automatically changed to parentheses:

- If the square bracket is after `cond`-like keyword, potentially skipping some of the sub-expressions in the `cond`-like expression (for example, in a `case` expression, the square brackets start in the second sub-expression).

- If the square bracket begins a new expression immediately after a `local`-like keyword. Note that the second expression after a `local`-like keyword will automatically become an ordinary parenthesis.
- If the square bracket is after a parenthesis that is after a `letrec`-like keyword,
- If the square bracket is in a sequence and the s-expression before in the sequence is a compound expression, DrScheme uses the same kind parenthesis, brace, or bracket as before, or
- If the square bracket is in the middle of string, comment, character, or symbol.

The upshot of DrScheme’s help is that you can always use the (presumably unshifted) square brackets on your keyboard to type parenthesis. For example, when typing

```
(define (length l)
  (cond
    [(empty? l) 0]
    [else (+ 1 (length (rest l)))]))
```

If you always type `[` and `]` where any of the square brackets or parentheses appear, DrScheme will change the square brackets to match the code above.

Of course, these features can be disabled and customized in the preferences dialog; see §3.2 “Preferences”. Also, in case DrScheme does not produce the character you want, holding down the control key while typing disables DrScheme’s parenthesis, brace, and bracket converter.

## 1.4 Searching

DrScheme’s search and replace feature is interactive, similar to those in Safari, Firefox, and Emacs, but with a few differences.

To start a search, first select the Find menu item from the Edit menu. This will open a small editor at the bottom of the DrScheme window. Start typing in there and, as you type, all occurrences of the string you’re searching for will be circled in the editor window. Watch the space right next to the search window to see how many occurrences of the search string there are in your file. When you’re ready, you use the Find Again menu item to jump to the first occurrence of the search string. This will color in one of the circles. Use Find Again a second time to jump to the next occurrence.

If you click back into the definitions window, the Find Again menu item, DrScheme will move the selection to the next occurrence of the search string.

DrScheme also supports a mode where typing in the search editor takes you directly to the next occurrence of the search string, without selecting the Find Again menu item. In

the preference dialog, in the Editing section and then in the General section is a checkbox labelled Search using anchors. When it is checked, DrScheme shows a little red dot and a red line indicating where the *search anchor* is. When the search anchor is enabled, typing in the search window jumps to the first occurrence of the search string after the anchor.

## 1.5 Tabbed Editing

DrScheme's allows you to edit multiple files in a single window via tabs. The File|New Tab menu item creates a new tab to show a new file. Each tab has its own interactions window.

In the General sub-pane of the Editing pane in the preferences window, a checkbox labelled Open files in separate tabs causes DrScheme to open files in new tabs in the frontmost window, rather than opening a new window for the file.

The key bindings Control-Pageup and Control-Pagedown move between tabs. Under Mac OS X, Command-Shift-Left and Command-Shift-Right also move between tabs.

## 1.6 The Interactions Window

The interactions window lets you type an expression after the > prompt for immediate evaluation. You cannot modify any text before the last > prompt. To enter an expression, the blinking caret must appear after the last prompt, and also after the space following the prompt.

When you type a complete expression and hit Enter or Return, DrScheme evaluates the expression and prints the result. After printing the result, DrScheme creates a new prompt for another expression. Some expressions return a special "void" value; DrScheme never prints void, but instead produces a new prompt immediately.

If the expression following the current prompt is incomplete, then DrScheme will not try to evaluate it. In that case, hitting Enter or Return produces a new, auto-indented line. You can force DrScheme to evaluate the expression by typing Alt-Return or Command-Return (depending on your platform).

To copy the previous expression to the current prompt, type ESC-p (i.e., type Escape and then type p). Type ESC-p multiple times to cycle back through old expressions. Type ESC-n to cycle forward through old expressions.

Clicking the Run button evaluates the program in the definitions window and makes the program's definitions available in the interactions window. Clicking Run also resets the interactions window, erasing all old interactions and removing old definitions from the interaction environment. Although Run erases old > prompts, ESC-p and ESC-n can still retrieve old expressions.

### 1.6.1 Errors

Whenever DrScheme encounters an error while evaluating an expression, it prints an error message in the interactions window and highlights the expression that triggered the error. The highlighted expression might be in the definitions window, or it might be after an old prompt in the interactions window.

For certain kinds of errors, DrScheme turns a portion of the error message into a hyperlink. Click the hyperlink to get help regarding a function or keyword related to the error.

For some run-time errors, DrScheme shows a bug icon next to the error message. Click the bug icon to open a window that shows a “stack” of expressions that were being evaluated at the time of the error. In addition, if the expressions in the stack appear in the definitions window, a red arrow is drawn to each expression from the next deeper one in the stack.

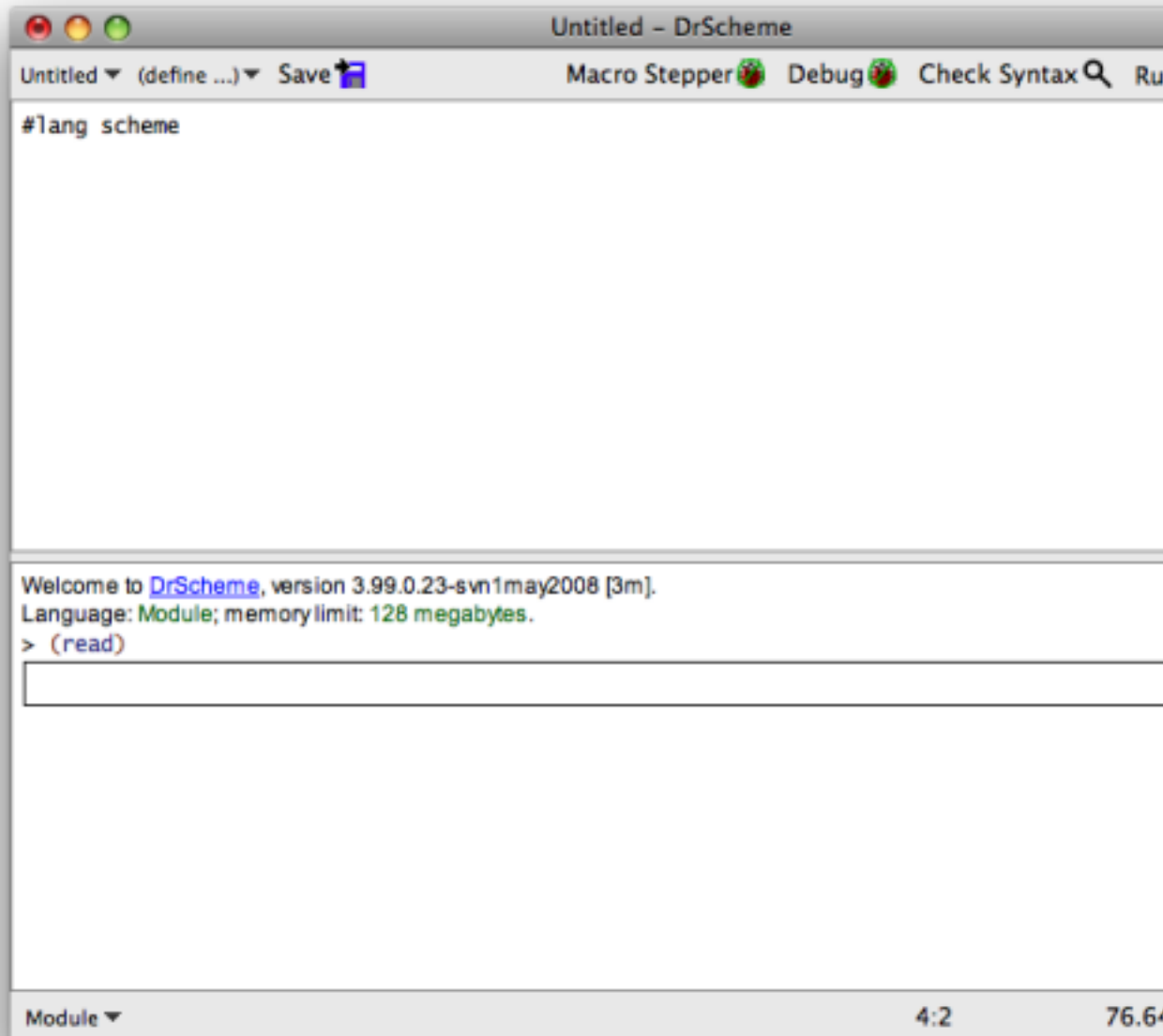
### 1.6.2 Input and Output

Many Scheme programs avoid explicit input and output operations, obtaining input via direct function calls in the interactions window, and producing output by returning values. Other Scheme programs explicitly print output for the user during evaluation using `write` or `display`, or explicitly request input from the user using `read` or `read-char`.

Explicit input and output appear in the interactions window, but within special boxes that separate explicit I/O from normal expressions and results. For example, evaluating

```
> (read)
```

in the interactions window produces a special box for entering input:



Type a number into the box and hit Enter, and that number becomes the result of the `(read)` expression. Once text is submitted for an input box, it is moved outside the input box, and when DrScheme shows a new prompt, it hides the interaction box. Thus, if you type `5` in the above input box and hit Return, the result appears as follows:

```
> (read)
5
5
> _
```

In this case, the first 5 is the input, and the second 5 is the result of the `(read)` expression. The second 5 is colored blue, as usual for a result printed by DrScheme. (The underscore indicates the location of the blinking caret.)

Output goes into the interactions window directly. If you run the program

```
#lang scheme
(define v (read))
(display v) (newline)
v
```

and provide the input S-expression `(1 2)`, the interactions window ultimately appears as follows:

```
(1 2)
(1 2)
(1 2)
> _
```

In this example, `display` produces output immediately beneath the input you typed, and the final result is printed last. The displayed output is drawn in purple. (The above example assumes default printing. With constructor-style value printing, the final before the prompt would be `(list 1 2)`.)

Entering the same program line-by-line in the interactions window produces a different-looking result:

```
> (define v (read))
(1 2)
> (display v)
(1 2)
> v
(1 2)
> _
```

Depending on the input operation, you may enter more text into an input box than is consumed. In that case, the leftover text remains in the input stream for later reads. For example, in the following interaction, two values are provided in response to the first `(read)`, so the second value is returned immediately for the second `(read)`:

```
> (read)
5 6
5
```

```
> (read)
6
> _
```

The following example illustrates that submitting input with Return inserts a newline character into the input stream:

```
> (read)
5

5
> (read-char)
#\newline
> _
```

Within a `#lang scheme` module, the results of top-level expression print the same as the results of an expression entered in the interactions window. The reason is that `#lang scheme` explicitly prints the results of top-level expressions using `(current-print)`, and DrScheme sets `(current-print)` to print values in the same way as for interactions.

## 1.7 Graphical Syntax

In addition to normal textual program, DrScheme supports certain graphical elements as expressions within a program. Plug-in tools can extend the available graphical syntax, but this section describes some of the more commonly used elements.

### 1.7.1 Images

DrScheme's `Insert|Insert Image...` menu item lets you select an image file from disk (in various formats such as GIF, PNG, and BMP), and the image is inserted at the current editing caret.

As an expression an image behaves like a number or string constant: it evaluates to itself. DrScheme's interactions window knows how to draw image-value results or images displayed via `print`.

A program can manipulate image values in various ways, such as using the `htdp/image` library or as an `image-snip%` value.

## 1.7.2 XML Boxes and Scheme Boxes

DrScheme has special support for XML concrete syntax. The Special|Insert XML Box menu item inserts an embedded editor into your program. In that embedded editor, you type XML's concrete syntax. When a program containing an XML box is evaluated, the XML box is translated into an *x-expression* (or *xexpr*), which is an s-expression representation of an XML expression. Each *xexpr* is a list whose first element is a symbol naming the tag, second element is an association list representing attributes and remaining elements are the nested XML expressions.

XML boxes have two modes for handling whitespace. In one mode, all whitespace is left intact in the resulting *xexpr*. In the other mode, any tag that only contains nested XML expressions and whitespace has the whitespace removed. You can toggle between these modes by right-clicking or Control-clicking (Mac OS X) on the top portion of the XML box.

In addition to containing XML text, XML boxes can also contain Scheme boxes. Scheme boxes contain Scheme expressions. These expressions are evaluated and their contents are placed into the containing XML box's *xexpr*. There are two varieties of Scheme box: the standard Scheme box and the splicing Scheme box. The standard Scheme box inserts its value into the containing *xexpr*. The contents of the splice box must evaluate to a list and the elements of the list are "flattened" into the containing *xexpr*. Right-clicking or control-clicking (Mac OS X) on the top of a Scheme box opens a menu to toggle the box between a Scheme box and a Scheme splice box.

## 1.8 Graphical Debugging Interface

**Tip:** Changing the name of a file in the middle of a debugging session will prevent the debugger from working properly on that file.

Like the Run button, the Debug button runs the program in the definitions window. However, instead of simply running it from start to finish, it lets users control and observe the program as it executes. The interface includes a panel of buttons above the definitions window, as well as extensions to the definitions window itself.

The program starts out paused just before the first expression is executed. This is indicated in the definitions window by the presence of a green triangle over this expression's left parenthesis.

### 1.8.1 Debugger Buttons

While execution is paused, several buttons are available:

- The Go button is enabled whenever the program is paused. It causes the program to

resume until it either completes, reaches a breakpoint, or raises an unhandled exception.

- The Step button is enabled whenever the program is paused. It causes the program to make a single step and then pause again.
- The Over button is only enabled when execution is paused at the start of an expression that is not in tail position. It sets a one-time breakpoint at the end of the expression (represented by a yellow circle) and causes the program to proceed. When execution reaches the one-time breakpoint, it pauses and removes that breakpoint.
- The Out button is only enabled when execution is paused within the context of another expression. Like the Over button, it sets a one-time breakpoint and continues execution. In this case, the program stops upon returning to the context or raising an unhandled exception.

If the program is running (not paused), then only the Pause button will be enabled. Clicking it will interrupt execution and pause it. In this case, the current expression may only be known approximately, and it will be represented as a gray triangle. The other features described above will still be available.

At any time, execution may be interrupted by clicking the Stop button. However, unlike with the Pause button, stopped execution cannot be continued.

## 1.8.2 Definitions Window Actions

When execution is paused, the definitions window supports several additional actions:

- Hovering the mouse cursor over a parenthesis may reveal a pink circle. If so, right-clicking or control-clicking (Mac OS X) will open a menu with options to Pause at this point or Continue to this point. The former sets an ordinary breakpoint at that location; the latter sets a one-time breakpoint and resumes execution. An ordinary breakpoint appears as a red circle, and a one-time breakpoint appears as a yellow circle.

**Tip:** If the debugged program is not in the Module language, then the *first time* it is debugged, breakpoints will only become available in expressions as they are evaluated. However, the next time the program is debugged, the debugger will remember the set of breakable locations from the previous session.

**Tip:** Clicking the Run button after a debugging session will cause all breakpoints to disappear from the definitions window. These breakpoints are not forgotten, and clicking Debug again will restore them. However, breakpoints do *not* persist across restarts of DrScheme.

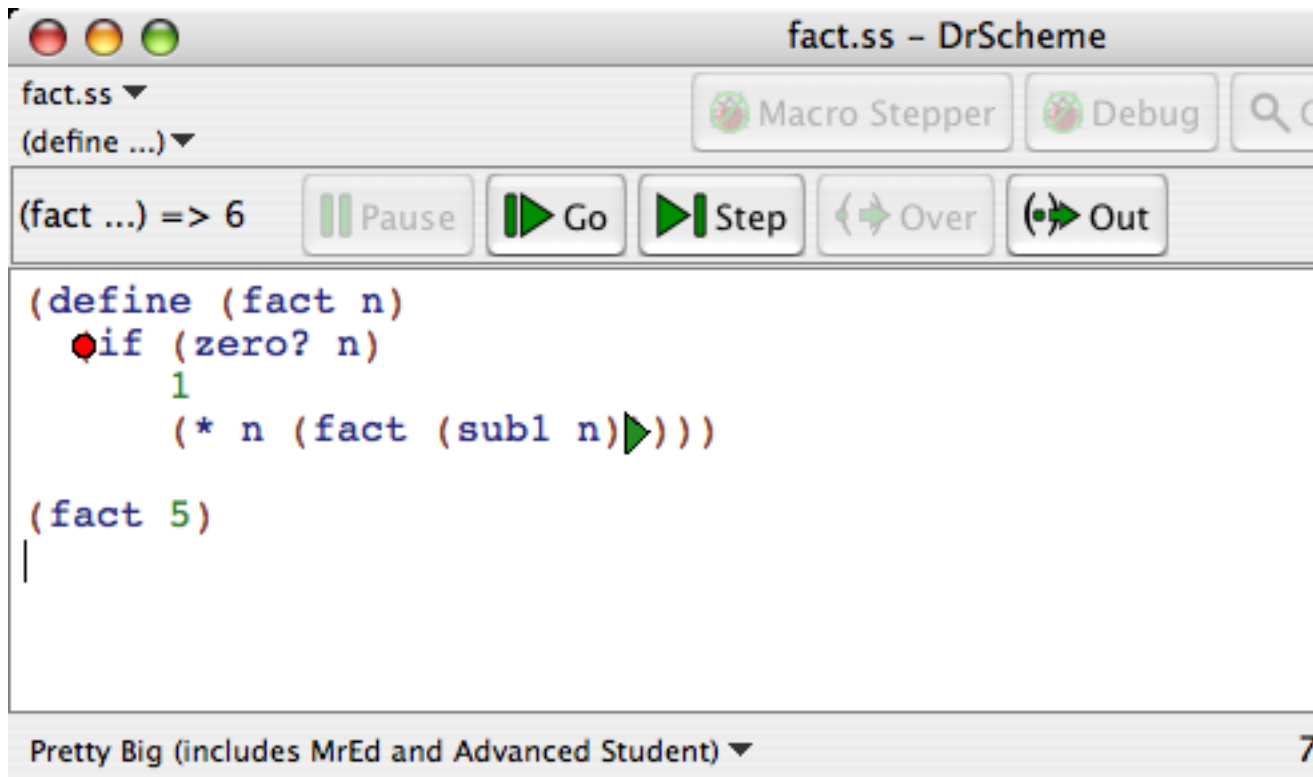
- If execution is paused at the start of an expression, then right-clicking or control-clicking (Mac OS X) on the green triangle opens a menu with the option to Skip expression.... Selecting this opens a text box in which to enter a value for the expression. The expression is skipped, with the entered value substituted for it.
- If execution is paused at the end of an expression, then the expression and its value are displayed to the left of the button bar. Right-clicking or control-clicking (Mac OS X) on the green triangle opens a menu with options to Print return value to console and Change return value.... The former displays the return value in the interactions window; the latter opens a text box in which to enter a substitute value.
- Hovering the mouse cursor over a bound variable displays the variable's name and value to the right of the button bar. Right-clicking or control-clicking (Mac OS X) opens a menu with options to Print value of <var> to console or (set! <var> ...). The former displays the variable's value in the interactions window; the latter opens a text box in which to enter a new value for the variable.

### 1.8.3 Stack View Pane

In addition, while execution is paused, the stack view pane at the right of the DrScheme frame is active. The top of the pane shows a list of active stack frames. Mousing over a frame produces a faint green highlighting of the corresponding expression. Clicking on the frame selects that frame, which makes its lexical variables visible. The selected frame is indicated by a bold font.

The bottom of the pane shows the lexical variables in the selected stack frame.

The following screenshot illustrates several aspects of the debugger interface. The red circle before the `if` is a breakpoint, and the green triangle at the end of the `(fact (sub1 n))` is where execution is currently paused. The expression's return value is displayed at the left of the button bar, and the value of `n` is displayed in the stack view pane.



#### 1.8.4 Debugging Multiple Files

To debug a program that spans several files, make sure that all of the files are open in DrScheme. Click the Debug button in the window containing the main program. As this program loads additional files that are present in other windows or tabs, message boxes will pop up asking whether or not to include the file in the debugging session. Including the file means that it will be possible to set breakpoints, inspect variables, and single-step in that file.

**Tip:** A file may only be involved in one debugging session at a time. If you try to debug a file that loads another file that is already being debugged, a message box will pop up explaining that the file cannot be included in another debugging session.

### 1.9 Creating Executables

DrScheme's Create Executable... menu item lets you create an executable for your program that you can start without first starting DrScheme. To create an executable, first save your

program to a file and set the language and teachpacks. Click Run, just to make sure that the program is working as you expect. The executable you create will not have a read-eval-print-loop, so be sure to have an expression that starts your program running in the definitions window before creating the executable.

Once you are satisfied with your program, choose the Create Executable... menu item from the Scheme menu. You will be asked to choose an executable file name or an archive file name. In the latter case, unpack the generated archive (on this machine or another one) to access the executable. In either case, you will be able to start the executable in the same way that you start any other program on your computer.

The result of Create Executable... is either a *launcher executable*, a *stand-alone executable*, or a *distribution archive*, and it uses either a *MzScheme* (textual) or *MrEd* (graphical) engine. For programs implemented with certain languages, Create Executable... will prompt you to choose the executable type and engine, while other languages support only one type or engine.

Each type has advantages and disadvantages:

- A *launcher executable* uses the latest version of your program source file when it starts. It also accesses library files from your DrScheme installation when it runs. Since a launcher executable contains specific paths to access those files, launchers usually cannot be moved from one machine to another.
- A *stand-alone executable* embeds a compiled copy of your program and any Scheme libraries that your program uses. When the executable is started, it uses the embedded copies and does not need your original source file. It may, however, access your DrScheme installation for DLLs, frameworks, shared libraries, or helper executables. Consequently, a stand-alone executable usually cannot be moved from one machine to another.
- A *distribution archive* packages a stand-alone executable together with any needed DLLs, frameworks, shared libraries, and helper executables. A distribution archive can be unpacked and run on any machine with the same operating system as yours.

In general, DrScheme's Module language gives you the most options. Most other languages only allow one type of executable. The teaching languages create stand-alone executables in distributions. The legacy languages create launchers.

**Tip:** Disable debugging in the language dialog before creating your executable. With debugging enabled, you will see a stack trace with error messages, but your program will run more slowly. To disable debugging, open the language dialog, click the Show Details button, and select No debugging or profiling, if it is available.

## 2 Languages

This chapter describes some of the languages that are available for use within DrScheme. The list here is potentially incomplete, because new languages can be added through DrScheme plug-ins.

### 2.1 Modules

The Module language is really a kind of meta-language, where the program itself specifies its language, usually through a `#lang` line.

More generally, when using Module, the definitions window must contain a module in some form. Besides `#lang`, a Scheme module can be written as `(module ...)`. In any case, aside from comments, the definitions window must contain exactly one module.

### 2.2 Legacy Languages

DrScheme supports several historically useful variants of Scheme without a `#lang` prefix:

- The R5RS language contains those primitives and syntax defined in the R<sup>5</sup>RS Scheme standard. See the `r5rs` library for details.
- The *PLT Pretty Big* language provides a language roughly compatible with a language in earlier versions of DrScheme. It evaluates a program in the same way as `load`, and it starts by importing the following modules: `mzscheme`, `scheme/gui/base`, `mzlib/class`, `mzlib/etc`, `mzlib/file`, `mzlib/list`, `mzlib/unit`, `mzlib/include`, `mzlib/defmacro`, `mzlib/pretty`, `mzlib/string`, `mzlib/thread`, `mzlib/math`, `mzlib/match`, and `mzlib/shared`.
- The Swindle language starts with the same bindings as `swindle`, and evaluates the program like `load`.

### 2.3 *How to Design Programs Teaching Languages*

Five of DrScheme's languages are specifically designed for teaching:

- The Beginning Student language is a small version of Scheme that is tailored for beginning computer science students.

- The Beginning Student with List Abbreviations languages is an extension to Beginning Student that prints lists with `list` instead of `cons`, and accepts quasiquoted input.
- The Intermediate Student language adds local bindings and higher-order functions.
- The Intermediate Student with Lambda language adds anonymous functions.
- The Advanced Student language adds mutable state.

The teaching languages are different from conventional Scheme in a number of ways:

- *Case-sensitive identifiers and symbols* — In a case-sensitive language, the variable names `x` and `X` are distinct, and the symbols `'x` and `'X` are also distinct. In a case-insensitive language, `x` and `X` are equivalent and `'x` and `'X` represent the same value. The teaching languages are case-sensitive by default, and other languages are usually case-insensitive. Case-sensitivity can be adjusted through the detail section of the language-selection dialog.
- *All numbers are exact unless #i is specified* — In the Beginning Student through Intermediate Student with Lambda languages, numbers containing a decimal point are interpreted as exact numbers. This interpretation allows students to use familiar decimal notation without inadvertently triggering inexact arithmetic. Exact numbers with decimal representations are also printed in decimal. Inexact inputs and results are explicitly marked with `#i`.
- *Procedures must take at least one argument* — In the Beginning Student through Intermediate Student languages, defined procedures must consume at least one argument. Since the languages have no side-effects, zero-argument functions are not useful, and rejecting such function definitions helps detect confusing syntactic mistakes.
- *Identifier required at function call position* — In the Beginning Student through Intermediate Student languages, procedure calls must be of the form `(identifier ...)`. This restriction helps detect confusing misuses of parentheses, such as `(1)` or `((+ 3 4))`, which is a common mistake among beginners who are used to the optional parentheses of algebra.
- *Top-level required at function call position* — In the Beginning Student languages, procedure calls must be of the form `(top-level-identifier ...)`, and the number of actual arguments must match the number of formal arguments if `top-level-identifier` is defined. This restriction helps detect confusing misuses of parentheses, such as `(x)` where `x` is a function argument. DrScheme can detect such mistakes syntactically because Beginning Student does not support higher-order procedures.
- *Primitive and defined functions allowed only in function call position* — In Beginning Student languages, the name of a primitive operator or of a defined function

can be used only after the open-parenthesis of a function call (except where teachpack extensions allow otherwise, as in the `convert-gui` teachpack). Incorrect uses of primitives trigger a syntax error. Incorrect uses of defined names trigger a run-time error. DrScheme can detect such mistakes because Beginning Student does not support higher-order procedures.

- *lambda allowed only in definitions* — In the Beginning Student through Intermediate Student languages, `lambda` (or `case-lambda`) may appear only in a definition, and only as the value of the defined variable.
- *Free variables are not allowed* — In the Beginning Student through Advanced Student languages, every variable referenced in the definitions window must be defined, pre-defined, or the name of a local function argument.
- *quote works only on symbols, quasiquote disallowed* — In the Beginning Student language, `quote` and `'` can specify only symbols. This restriction avoids the need to explain to beginners why `1` and `'1` are equivalent in standard Scheme. In addition, `quasiquote`, `⚭`, `unquote`, `,`, `unquote-splicing`, and `,@` are disallowed.
- *Unmatched cond/case is an error* — In the Beginning Student through Advanced Student languages, falling through a `cond` or `case` expression without matching a clause signals a run-time error. This convention helps detect syntactic and logical errors in programs.
- *Conditional values must be true or false* — In the Beginning Student through Advanced Student languages, an expression whose value is treated as a boolean must return an actual boolean, `true` or `false`. This restriction, which applies to `if`, `cond`, `and`, `or`, `nand`, and `nor` expressions, helps detect errors where a boolean function application is omitted.
- *+, \*, and / take at least two arguments* — In the Beginning Student through Advanced Student languages, mathematical operators that are infix in algebra notation require at least two arguments in DrScheme. This restriction helps detect missing arguments to an operator.
- *and, or, nand, and nor require at least 2 expressions* — In the Beginning Student through Advanced Student languages, the boolean combination forms require at least two sub-expressions. This restriction helps detect missing or ill-formed sub-expressions in a boolean expression.
- *set! disallowed on arguments* — In the Advanced Student language, `set!` cannot be used to mutate variables bound by `lambda`. This restriction ensures that the substitution model of function application is consistent with DrScheme's evaluation.
- *Improper lists disallowed* — A *proper list* is either an empty list or a list created by `consing` onto a proper list. In the Beginning Student through Advanced Student languages, `cons` constructs only *proper lists*, signaling an error if the second argument is not a proper list. Since beginning students do not need improper lists, this restriction help detect logical errors in recursive functions.

- *Dot is disallowed* — In the Beginning Student through Advanced Student languages, a delimited period `.` is disallowed, (e.g., as an improper-list constructor in a quoted form, or for defining multi-arity procedures).
- *Syntactic form names disallowed as variable names* — In the Beginning Student through Advanced Student languages, all syntactic form names are keywords that cannot be used as variable names.
- *Re-definitions are disallowed* — In the Beginning Student through Advanced Student languages, top-level names can never be re-defined.
- *Function definitions are allowed only in the definitions window* — In the Beginning Student languages, function definitions are not allowed in the interactions window.

The teaching languages also deviate from traditional Scheme in printing values. Different printing formats can be selected for any language through the detail section of language-selection dialog.

- *Constructor-style output* — See §2.6 “Output Printing Styles”.
- *Quasiquote-style output* — See §2.6 “Output Printing Styles”.
- *Rational number printing* – In the teaching languages, all numbers that have a finite decimal expansion are printed in decimal form. For those numbers that do not have a finite decimal expansion (such as  $4/3$ ) DrScheme offers a choice. It either prints them as mixed fractions or as repeating decimals, where the repeating portion of the decimal expansion is shown with an overbar. In addition, DrScheme only shows the first 25 digits of the number’s decimal expansion. If there are more digits, the number appears with an ellipsis at the end. Click the ellipses to see the next 25 digits of the expansion.

This setting controls only the initial display of a number. Right-clicking or Control-clicking (Mac OS X) on the number lets you change from the fraction representation to the decimal representation.

- *write output* — Prints values with `write`.
- *Show sharing in values* — Prints interaction results using the shared syntax, which exposes shared structure within a value. For example, the list created by `(let ([lt (list 0)]) (list lt lt))` prints as

```
(shared ((-1- (list 0))) (list -1- -1-))
```

instead of

```
(list (list 0) (list 0))
```

A program in the teaching languages should be tested using the check forms — `(check-expect value value)`, `(check-within value value value)`, or `(check-error value string)`. Tests are evaluated when running the program: when there are no

tests, a warning appears in the interactions window; when all tests succeed, an acknowledgment appears in the interactions window; otherwise, a testing window appears to report the results. See §3.1.3 “View” for details on configuring the report behavior.

Tests can be disabled if necessary, see §3.1.5 “Scheme” for details.

## 2.4 ProfessorJ

The ProfessorJ languages are Java based languages designed for teaching and experimentation. There are four teaching based languages:

- The ProfessorJ Beginner language is a small subset of Java, designed for novice computer science students. Each class must contain a constructor that initializes the class’s fields; each method must return a value.
- The ProfessorJ Intermediate language is an extension to ProfessorJ Beginner that adds full class-based inheritance and mutation. Classes do not require constructors and null values may arise.
- The ProfessorJ Intermediate + access language adds access controls, such as public or private, to member definitions and supports overloading constructor definitions.
- The ProfessorJ Advanced language adds arrays, loop constructs, and package specifications.

The remaining two languages support language extensions and experimentations:

- The ProfessorJ Full language supports most of Java 1.1 features, as well as a set of constructs designed for the development of unit tests
- The ProfessorJ Java + dynamic language extends the ProfessorJ Full language with a dynamic type and the ability to import Scheme libraries directly, for developing programs that use both languages.

Value printing can be either *Class* or *Class + Field*, selectable in the show details section of the language selection window. The Class printing style only displays the class name for any object value. The Class + Field style displays the class combined with the names and values for all of the class’s fields; when displaying a recursively defined object, any repeated object reverts to the Class style display for the second appearance. Languages with arrays may opt to always display an entire array or truncate the middle section of longer arrays.

All of the ProfessorJ languages support testing extensions, and tests are required within the teaching languages. The experimental languages, Full and Java + dynamic, allow the removal of these forms within the language selection window.

Programs in the teaching languages must be tested, using a class containing the word 'Example' in the name and the 'check ... expect ...' comparison forms. On run, all Example classes are instantiated and all methods prefixed with the word 'test' are run. When there are no tests, a warning appears in the interactions window; when all tests succeed, an acknowledgement appears in the interactions window; otherwise, a testing window appears to report the results. See §3.1.3 "View" for details on configuring the report behavior. Tests can be disabled if necessary, see §3.1.5 "Scheme" for details.

Unless disabled in the language configuration window, expression-level coverage information is collected during testing. Selecting the buttons within the report modifies the color of the program in the definitions window, to distinguish expressions that were used in the test from those that were not. Typing into the definitions window restores the original coloring.

## 2.5 Other Experimental Languages

For information on Lazy Scheme, see § "Lazy Scheme".

For information on FrTime, see § "FrTime: A Language for Reactive Programs".

For information on Algol 60, see § "Algol 60".

## 2.6 Output Printing Styles

Many Scheme languages support a Output Syntax choice that determines how evaluation results are printed in the interactions window. This setting also applies to output generated by calling `print` explicitly.

The following table illustrates the difference between the different output styles:

Input expression	Constructor	Quasiquote	write
<code>(cons 1 2)</code>	<code>(cons 1 2)</code>	<code>'(1 . 2)</code>	<code>(1 . 2)</code>
<code>(list 1 2)</code>	<code>(list 1 2)</code>	<code>'(1 2)</code>	<code>(1 2)</code>
<code>'(1 2)</code>	<code>(list 1 2)</code>	<code>'(1 2)</code>	<code>(1 2)</code>
<code>(list (void))</code>	<code>(list (void))</code>	<code>'(,(void))</code>	<code>(#&lt;void&gt;)</code>
<code>'(,(void))</code>	<code>(list (void))</code>	<code>'(,(void))</code>	<code>(#&lt;void&gt;)</code>
<code>(vector 1 2 3)</code>	<code>(vector 1 2 3)</code>	<code>(vector 1 2 3)</code>	<code>#(1 2 3)</code>
<code>(box 1)</code>	<code>(box 1)</code>	<code>(box 1)</code>	<code>#&amp;1</code>
<code>(lambda (x) x)</code>	<code>(lambda (a1) ...)</code>	<code>(lambda (a1) ...)</code>	<code>#&lt;procedure&gt;</code>
<code>'sym</code>	<code>'sym</code>	<code>'sym</code>	<code>sym</code>
<code>(make-s 1 2)</code>	<code>(make-s 1 2)</code>	<code>(make-s 1 2)</code>	<code>#(struct:s 1 2)</code>
<code>'()</code>	<code>empty</code>	<code>'()</code>	<code>()</code>
<code>add1</code>	<code>add1</code>	<code>add1</code>	<code>#&lt;procedure:add1&gt;</code>
<code>(delay 1)</code>	<code>(delay ...)</code>	<code>(delay ...)</code>	<code>#&lt;promise&gt;</code>
<code>(regexp "a")</code>	<code>(regexp "a")</code>	<code>(regexp "a")</code>	<code>#rx"a"</code>

The Constructor output mode treats `cons`, `vector`, and similar primitives as value constructors, rather than functions. It also treats `list` as shorthand for multiple `cons`'s ending with the empty list. Constructor output is especially valuable for beginning programmers, because output values look the same as input values.

The Quasiquote output mode is like Constructor output, but it uses `quasiquote` (abbreviated with `⚭`) to print lists, and it uses `unquote` (abbreviated with `⚭`) to escape back to Constructor printing as needed. This mode provides the same benefit as Constructor output, in that printed results are expressions, but it is more convenient for many kinds of data, especially data that represents expressions.

The print output mode corresponds to traditional Scheme printing via the `print` procedure, which defaults to `write`-like printing, as shown in the last column.

DrScheme also sets the `global-port-print-handler` in order to customize a few aspects of the printing for all of these modes, namely printing the symbol `quote` as a single tick mark (mutatis mutandis for `quasiquote`, `unquote`, and `unquote-splicing`), and to print rational real numbers using a special `snip%` object that lets the user choose between improper fractions, mixed fractions, and repeating decimals.

## 3 Interface Reference

### 3.1 Menus

#### 3.1.1 File

- New : Creates a new DrScheme window.
- Open... : Opens a find-file dialog for choosing a file to load into a definitions window.
- Open Recent : Lists recently opened files. Choosing one of them opens that file for editing.
- Install PLT File... : Opens a dialog asking for the location of the ".plt" file (either on the local disk or on the web) and installs the contents of the file.
- Revert : Re-loads the file that is currently in the definitions window. All changes since the file was last saved will be lost.
- Save Definitions : Saves the program in the definitions window. If the program has never been saved before, a save-file dialog appears.
- Save Definitions As... : Opens a save-file dialog for choosing a destination file to save the program in the definitions window. Subsequent saves write to the newly-selected file.
- Save Other : Contains these sub-items
  - Save Definitions As Text... : Like Save Definitions As..., but the file is saved in plain-text format (see §3.4.1 “Program Files”). Subsequent saves also write in plain-text format.
  - Save Interactions : Saves the contents of the interactions window to a file. If the interaction constants have never been saved before, a save-file dialog appears.
  - Save Interactions As... : Opens a save-file dialog for choosing a destination file to save the contents of the interactions window. Subsequent saves write to the newly-selected file.
  - Save Interactions As Text... : Like Save Interactions As..., but the file is saved in plain-text format (see §3.4.1 “Program Files”). Subsequent saves are write in plain-text format.
- Log Definitions and Interactions... : Starts a running of log of the text in the interactions and definitions windows, organized by executions. In a directory of your choosing, DrScheme saves files with the names "01-definitions", "01-interactions", "02-definitions", "02-interactions", etc. as you interact with various programs.

- Print Definitions... : Opens a dialog for printing the current program in the definitions window.
- Print Interactions... : Opens a dialog for printing the contents of the interactions window.
- Search in Files... : Opens a dialog where you can specify the parameters of a multi-file search. The results of the search are displayed in a separate window.
- Close : Closes this DrScheme window. If this window is the only open DrScheme window, then DrScheme quits, except under Mac OS X.
- {Quit or Exit} Exits DrScheme. (Under Mac OS X, this menu item is in the Apple menu.)

### 3.1.2 Edit

All Edit menu items operate on either the definitions or interactions window, depending on the location of the selection or blinking caret. Each window maintains its own Undo and Redo history.

- Undo : Reverses an editing action. Each window maintains a history of actions, so multiple Undo operations can reverse multiple editing actions.
- Redo : Reverses an Undo action. Each window (and boxed-subwindow) maintains its own history of Undo actions, so multiple Redo operations can reverse multiple Undo actions.
- Cut : Copies the selected text to the clipboard and deletes it from the window.
- Copy : Copies the selected text to the clipboard.
- Paste : Pastes the current clipboard contents into the window.
- Delete : or Clear : Deletes the selected text.
- Select All : Highlights the entire text of the buffer.
- Wrap Text : Toggles between wrapped text and unwrapped text in the window.
- Find... : Opens an interactive search window at the bottom of the frame and moves the insertion point to the search string editor (or out of it, if the insertion point is already there).  
See also §1.4 “Searching”.
- Find Again : Finds the next occurrence of the text in the search window.
- Find Again Backwards : Finds the next occurrence of the text in the search window, but searching backwards.

- Replace & Find Again : Replaces the selection with the replace string (if it matches the find string) and finds the next occurrence of the text that was last searched for, looking forwards.
- Replace & Find Again Backwards : Replaces the selection with the replace string (if it matches the find string) and finds the next occurrence of the text that was last searched for, looking backwards.
- Replace All : Replaces all occurrences of the search string with the replace string.
- Find Case Sensitive : Toggles between case-sensitive and case-insensitive search.
- Keybindings :
  - Show Active Keybindings : Shows all of the keybindings available in the current window.
  - Add User-defined Keybindings... : Choosing this menu item opens a file dialog where you can select a file containing Scheme-definitions of keybindings. See §3.3.8 “Defining Custom Shortcuts” for more information.
- Complete Word : Completes the word at the insertion point, using the manuals as a source of completions.
- Preferences... : Opens the preferences dialog. See §3.2 “Preferences”. (Under Mac OS X, this menu item is in the Apple menu.)

### 3.1.3 View

One each of the following show/hide pairs of menu items appears at any time.

- Show Definitions : Shows the definitions window.
- Hide Definitions : Hides the definitions window.
- Show Interactions : Shows interactions window.
- Hide Interactions : Hides interactions window.
- Show Program Contour : Shows a “20,000 foot” overview window along the edge of the DrScheme window. Each pixel in this window corresponds to a letter in the program text.
- Hide Program Contour : Hides the contour window.
- Show Module Browser : Shows the module DAG rooted at the currently opened file in DrScheme.
- Hide Module Browser : Hides the module browser.

- Toolbar :
  - Toolbar on Left : Moves the tool bar (defaultly on the top of DrScheme’s window) to the left-hand side, organized vertically.
  - Toolbar on Top : Moves the toolbar to the top of the DrScheme window.
  - Toolbar on Right : Moves the tool bar to the right-hand side, organized vertically.
  - Toolbar Hidden : Hides the toolbar entirely.
- Show Log : Shows the current log messages.
- Hide Log : Hides the current log messages.
- Show Profile : Shows the current profiling report. This menu is useful only if you have enabled profiling in the Choose Language... dialog’s Details section. Profiling does not apply to all languages.
- Hide Profile : Hides any profiling information currently displayed in the DrScheme window.
- Dock Test Report : Like the dock button on the test report window, this causes all test report windows to merge with the appropriate DrScheme window at the bottom of the frame.
- Undock Test Report : Like the undock button on the test report window, this causes the test reports attached to appropriate DrScheme tabs to become separate windows.
- Show Tracing : Shows a trace of functions called since the last time Run was clicked. This menu is useful only if you have enabled tracing in the Choose Language... dialog’s Details section. Profiling does not apply to all languages.
- Hide Tracing : Hides the tracing display.
- Split : Splits the current window in half to allow for two different portions of the current window to be visible simultaneously.
- Collapse : If the window has been split before, this menu item becomes enabled, allowing you to collapse the split window.

Note: whenever a program is run, the interactions window is made visible if it is hidden.

### 3.1.4 Language

- Choose Language... : Opens a dialog for selecting the current evaluation language. Click Run to make the language active in the interactions window. See §1.2 “Choosing a Language” for more information about the languages.

- **Add Teachpack...** : Opens a find-file dialog for choosing a teachpack to extend the current language. Click Run to make the teachpack available in the interactions windows. See §4 “Extending DrScheme” for information on creating teachpacks.
- **Clear All Teachpacks** : Clears all of the current teachpacks. Click Run to clear the teachpack from the interactions window.

In addition to the above items, a menu item for each teachpack that clears only the corresponding teachpack.

### 3.1.5 Scheme

- **Run** : Resets the interactions window and runs the program in the definitions window.
- **Break** : Breaks the current evaluation.
- **Kill** : Terminates the current evaluation.
- **Limit Memory...** : Allow you to specify a limit on the amount of memory that a program running in DrScheme is allowed to consume.
- **Clear Error Highlight** : Removes the red background that signals the source location of an error.
- **Create Executable...** : Creates a separate launcher for running your program. See §1.9 “Creating Executables” for more info.
- **Module Browser...** : Prompts for a file and then opens a window showing the module import structure for the module import DAG starting at the selected module.  
 The module browser window contains a square for each module. The squares are colored based on the number of lines of code in the module. If a module has more lines of code, it gets a darker color.  
 In addition, for each normal import, a blue line drawn is from the module to the importing module. Similarly, purple lines are drawn for each for-syntax import. In the initial module layout, modules to the left import modules to the right, but since modules can be moved around interactively, that property might not be preserved.  
 To open the file corresponding to the module, right-click or control-click (Mac OS X) on the box for that module.
- **Reindent** : Indents the selected text according to the standard Scheme formatting conventions. (Pressing the Tab key has the same effect.)
- **Reindent All** : Indents all of the text in either the definitions or interactions window, depending on the location of the selection or blinking caret.
- **Comment Out with Semicolons** : Puts ; characters at each of the beginning of each selected line of text.

- Comment Out with a Box : Boxes the selected text with a comment box.
- Uncomment : Removes all ; characters at the start of each selected line of text or removes a comment box around the text. Uncommenting only removes a ; if it appears at the start of a line and it only removes the first ; on each line.
- Disable Tests : Stops tests written in the definitions window from evaluating when the program is Run. Tests can be enabled using the Enable Tests menu item. Disabling tests freezes the contents of any existing test report window.
- Enable Tests : Allows tests written in the definitions window to evaluate when the program is Run. Tests can be disabled using the Disable Tests menu item.

### 3.1.6 Insert

- Insert Comment Box : Inserts a box that is ignored by DrScheme; use it to write comments for people who read your program.
- Insert Image... : Opens a find-file dialog for selecting an image file in GIF, BMP, XBM, XPM, PNG, or JPG format. The image is treated as a value.
- Insert Fraction... : Opens a dialog for a mixed-notation fraction, and inserts the given fraction into the current editor.
- Insert Large Letters... : Opens a dialog for a line of text, and inserts a large version of the text (using semicolons and spaces).
- Insert  $\lambda$  : Inserts the symbol  $\lambda$  (as a Unicode character) into the program. The  $\lambda$  symbol is normally bound the same as lambda.
- Insert Java Comment Box : Inserts a box that is ignored by DrScheme. Unlike the Insert Comment Box menu item, this is designed for the ProfessorJ language levels. See §2.4 “ProfessorJ”.
- Insert Java Interactions Box : Inserts a box that will allow Java expressions and statements within Scheme programs. The result of the box is a Scheme value corresponding to the result(s) of the Java expressions. At this time, Scheme values cannot enter the box. The box will accept one Java statement or expression per line.
- Insert XML Box : Inserts an XML; see §1.7.2 “XML Boxes and Scheme Boxes” for more information.
- Insert Scheme Box : Inserts a box to contain Scheme code, typically used inside an XML box; see §1.7.2 “XML Boxes and Scheme Boxes”.
- Insert Scheme Splice Box : Inserts a box to contain Scheme code, typically used inside an XML box; see also §1.7.2 “XML Boxes and Scheme Boxes”.
- Insert Pict Box : Creates a box for generating a Slideshow picture. Inside the pict box, insert and arrange Scheme boxes that produce picture values.

### 3.1.7 Windows

- Bring Frame to Front... : Opens a window that lists all of the opened DrScheme frames. Selecting one of them brings the window to the front.
- Most Recent Window : Toggles between the currently focused window and the one that most recently had the focus.

Additionally, after the above menu items, this menu contains an entry for each window in DrScheme. Selecting a menu item brings the corresponding window to the front.

### 3.1.8 Help

- Help Desk : Opens the Help Desk. This is the clearing house for all documentation about DrScheme and its language.
- About DrScheme... : Shows the credits for DrScheme.
- Related Web Sites : Provides links to related web sites.
- Tool Web Sites : Provides links to web sites for installed tools.
- Interact with DrScheme in English : Changes DrScheme's interface to use English; the menu item appears only when the current language is not English. Additional menu items switch DrScheme to other languages.

## 3.2 Preferences

The preferences dialog consists of several panels.

### 3.2.1 Font

This panel controls the main font used by DrScheme.

### 3.2.2 Colors

The Colors panel has several sub-panels that let you configure the colors that DrScheme uses for the editor background, for highlighting matching parentheses, for the online coloring for Scheme and Java modes, for Check Syntax, and for the colors of the text in the interactions window.

It also has two buttons, White on Black and Black on White, which set a number of defaults for the color preferences and change a few other aspects of DrScheme's behavior to make DrScheme's colors look nicer for those two modes.

### 3.2.3 Editing

The Editing panel consists of several sub-panels:

- Indenting  
This panel controls which keywords DrScheme recognizes for indenting, and how each keyword is treated.
- Square bracket  
This panel controls which keywords DrScheme uses to determine when to rewrite `[` to `(`. For cond-like keywords, the number in parenthesis indicates how many sub-expressions are skipped before square brackets are started.  
See §1.3 “Editing with Parentheses” for details on how the entries in the columns behave.
- General
  - Number of recent items — controls the length of the Open Recent menu (in the File menu).
  - Auto-save files — If checked, the editor generates autosave files (see §3.4.2 “Backup and Autosave Files”) for files that have not been saved after five minutes.
  - Backup files — If checked, when saving a file for the first time in each editing session, the original copy of the file is copied to a backup file in the same directory. The backup files have the same name as the original, except that they end in either “.bak” or “~”.
  - Map delete to backspace — If checked, the editor treats the Delete key like the Backspace key.
  - Show status-line — If checked, DrScheme shows a status line at the bottom of each window.
  - Count column numbers from one — If checked, the status line's column counter counts from one. Otherwise, it counts from zero.
  - Display line numbers in buffer; not character offsets — If checked, the status line shows a `<line>:<column>` display for the current selection rather than the character offset into the text.
  - Wrap words in editor buffers — If checked, DrScheme editors auto-wrap text lines by default. Changing this preference affects new windows only.

- Use separate dialog for searching — If checked, then selecting the Find menu item opens a separate dialog for searching and replacing. Otherwise, selecting Find opens an interactive search-and-replace panel at the bottom of a DrScheme window.
  - Reuse existing frames when opening new files — If checked, new files are opened in the same DrScheme window, rather than creating a new DrScheme window for each new file.
  - Enable keybindings in menus — If checked, some DrScheme menu items have keybindings. Otherwise, no menu items have key bindings. This preference is designed for people who are comfortable editing in Emacs and find the standard menu keybindings interfere with the Emacs keybindings.
  - Color syntax interactively — If checked, DrScheme colors your syntax as you type.
  - Automatically print to PostScript file — If checked, printing will automatically save PostScript files. If not, printing will use the standard printing mechanisms for your computer.
  - Open files in separate tabs (not separate windows) – If checked, DrScheme will use tabs in the front-most window to open new files, rather than creating new windows for new files.
  - Automatically open interactions window when running a program – If checked, DrScheme shows the interactions window (if it is hidden) when a program is run.
  - Put the interactions window beside the definitions window – If checked, DrScheme puts the interactions window to the right of the definitions window. By default, the interactions window is below the definitions window.
  - Always show the #lang line in the Module language – If checked, the module language always shows the the #lang line (even when it would ordinarily be scrolled off of the page), assuming that the #lang line is the first line in the file.
- Scheme
    - Highlight between matching parens — If checked, the editor marks the region between matching parenthesis with a gray background (in color) or a stipple pattern (in monochrome) when the blinking caret is next to a parenthesis.
    - Correct parens — If checked, the editor automatically converts a typed `)` to `]` to match `[`, or it converts a typed `]` to `)` to match `(`. Also, the editor changes typed `[` to match the context (as explained in §1.3 “Editing with Parentheses”).
    - Flash paren match — If checked, typing a closing parenthesis, square bracket, or quotation mark flashes the matching open parenthesis/bracket/quote.

### 3.2.4 Warnings

- Ask before changing save format — If checked, DrScheme consults the user before saving a file in non-text format (see §3.4.1 “Program Files”).

- Verify exit — If checked, DrScheme consults the user before exiting.
- Only warn once when executions and interactions are not synchronized — If checked, DrScheme warns the user on the first interaction after the definitions window, language, or teachpack is changed without a corresponding click on Run. Otherwise, the warning appears on every interaction.
- Ask about clearing test coverage — If checked, when test coverage annotations are displayed DrScheme prompts about removing them. This setting only applies to the PLT languages. DrScheme never asks in the teaching languages.
- Check for newer PLT Scheme versions — If checked, DrScheme periodically polls a server to determine whether a newer version of DrScheme is available.

### 3.2.5 Profiling

This preference panel configures the profiling report. The band of color shows the range of colors that profiled functions take on. Colors near the right are used for code that is not invoked often and colors on the left are used for code that is invoked often.

If you are interested in more detail at the low end, choose the Square root check box. If you are interested in more detail at the upper end, choose the Square check box.

### 3.2.6 Browser

This preferences panel allows you to configure your HTTP proxy. Contact your system administrator for details.

## 3.3 Keyboard Shortcuts

Most key presses simply insert a character into the editor, such as **a**, **3**, or **(**. Other keys and key combinations act as keyboard shortcuts that move the blinking caret, delete a line, copy the selection, etc. Keyboard shortcuts are usually triggered by key combinations using the Control, Meta, or Command key.

C-*<key>* means press the Control key, hold it down and then press *<key>* and then release them both. For example: C-e (Control-E) moves the blinking caret to the end of the current line.

M-*<key>* is the same as C-*<key>*, except with the Meta key. Depending on your keyboard, Meta may be called “Left,” “Right,” or have a diamond symbol, but it’s usually on the bottom row next to the space bar. M-*<key>* can also be performed as a two-character sequence: first, strike and release the Escape key, then strike *<key>*. Under Windows and Mac OS X, Meta is only available through the Escape key.

Many of the key-binding actions can also be performed with menu items.

DEL is the Delete key.

SPACE is the Space bar.

On most keyboards, “<” and “>” are shifted characters. So, to get M->, you actually have to type Meta-Shift->. That is, press and hold down both the Meta and Shift keys, and then strike “>”.

Under Windows, some of these keybindings are actually standard menu items. Those keybindings will behave according to the menus, unless the Enable keybindings in menus preference is unchecked.

If you are most familiar with Emacs-style key bindings, you should uncheck the Enable keybindings in menus preference. Many of the keybindings below are inspired by Emacs.}

### 3.3.1 Moving Around

- C-f : move forward one character
- C-b : move backward one character
- M-f : move forward one word
- M-b : move backward one word
- C-v : move forward one page
- M-v : move backward one page
- M-< : move to beginning of file
- M-> : move to end of file
- C-a : move to beginning of line (left)
- C-e : move to end of line (right)
- C-n : move to next line (down)
- C-p : move to previous line (up)
- M-C-f : move forward one S-expression
- M-C-b : move backward one S-expression
- M-C-u : move up out of an S-expression
- M-C-d : move down into a nested S-expression
- M-C-SPACE : select forward S-expression

- M-C-p : match parentheses backward
- M-C-left : move backwards to the nearest editor box
- A-C-left : move backwards to the nearest editor box
- M-C-right : move forward to the nearest editor box
- A-C-right : move forward to the nearest editor box
- M-C-up : move up out of an embedded editor
- A-C-up : move up out of an embedded editor
- M-C-down : move down into an embedded editor
- A-C-down : move down into an embedded editor
- C-F6 : move the cursor from the definitions window to the interactions window (or the search window, if it is open).

### 3.3.2 Editing Operations

- C-\_ : undo
- C-+ : redo
- C-x u : undo
- C-d : delete forward one character
- C-h : delete backward one character
- M-d : delete forward one word
- M-DEL : delete backward one word
- C-k : delete forward to end of line
- M-C-k : delete forward one S-expression
- M-w : copy selection to clipboard
- C-w : delete selection to clipboard (cut)
- C-y : paste from clipboard (yank)
- C-t : transpose characters
- M-t : transpose words
- M-C-t : transpose sexpressions

- M-C-m : toggle dark green marking of matching parenthesis
- M-C-k : cut complete sexpression
- M-( : wrap selection in parentheses
- M-[ : wrap selection in square brackets
- M-{ : wrap selection in curly brackets
- M-S-L : wrap selection in `(lambda () ...)` and put the insertion point in the arglist of the lambda
- C-c C-o : the sexpression following the insertion point is put in place of its containing sexpression
- C-c C-e : the first and last characters (usually parentheses) of the containing expression are removed
- C-c C-l : wraps a let around the sexpression following the insertion point and puts a printf in at that point (useful for debugging).
- M-o : toggle overwrite mode

### 3.3.3 File Operations

- C-x C-s : save file
- C-x C-w : save file under new name

### 3.3.4 Search

- C-s : search for string forward
- C-r : search for string backward

### 3.3.5 Miscellaneous

- F5 : Run

### 3.3.6 Interactions

The interactions window has all of the same keyboard shortcuts as the definitions window plus a few more:

- M-p : bring the previously entered expression down to the prompt
- M-n : bring the expression after the current expression in the expression history down to the prompt

### 3.3.7 LaTeX and TeX inspired keybindings

- C-\ M-\ : traces backwards from the insertion point, looking for a backslash followed by a LaTeX macro name; if one is found, it replaces the backslash and the macro's name with the keybinding. These are the currently supported macro names and the keys they map into:

```
\Downarrow
\Downarrow
\downarrow
\Rightarrow
\rightarrow
\searrow
\swarrow
\leftarrow
\uparrow
\Leftarrow
\longrightarrow
\Uparrow
\Leftrightarrow
\updownarrow
\leftrightharrow
\nearrow
\Updownarrow
\aleph
\prime
\emptyset
\nabla
\diamondsuit
\spadesuit
\clubsuit
\heartsuit
\sharp
\flat
```

<code>\natural</code>	
<code>\surd</code>	
<code>\neg</code>	
<code>\triangle</code>	
<code>\forall</code>	
<code>\exists</code>	
<code>\infty</code>	$\infty$
<code>\circ</code>	
<code>\alpha</code>	$\alpha$
<code>\theta</code>	
<code>\tau</code>	
<code>\beta</code>	$\beta$
<code>\vartheta</code>	
<code>\pi</code>	$\pi$
<code>\upsilon</code>	
<code>\gamma</code>	$\gamma$
<code>\varpi</code>	$\pi$
<code>\phi</code>	
<code>\delta</code>	
<code>\kappa</code>	$\kappa$
<code>\rho</code>	
<code>\varphi</code>	
<code>\epsilon</code>	
<code>\lambda</code>	$\lambda$
<code>\varrho</code>	
<code>\chi</code>	
<code>\varepsilon</code>	
<code>\mu</code>	$\mu$
<code>\sigma</code>	$\sigma$
<code>\psi</code>	
<code>\zeta</code>	
<code>\nu</code>	
<code>\varsigma</code>	$\varsigma$
<code>\omega</code>	
<code>\eta</code>	
<code>\xi</code>	
<code>\Gamma</code>	
<code>\Lambda</code>	$\Lambda$
<code>\Sigma</code>	$\Sigma$
<code>\Psi</code>	
<code>\Delta</code>	
<code>\Xi</code>	
<code>\Upsilon</code>	
<code>\Omega</code>	
<code>\Theta</code>	
<code>\Pi</code>	

<code>\Phi</code>	
<code>\pm</code>	
<code>\cap</code>	
<code>\diamond</code>	
<code>\oplus</code>	⊕
<code>\mp</code>	
<code>\cup</code>	∪
<code>\bigtriangleup</code>	
<code>\ominus</code>	⊖
<code>\times</code>	
<code>\uplus</code>	⊕
<code>\bigtriangledown</code>	
<code>\otimes</code>	⊗
<code>\div</code>	
<code>\sqcap</code>	⊓
<code>\triangleleft</code>	
<code>\oslash</code>	⊘
<code>\ast</code>	
<code>\sqcup</code>	⊔
<code>\vee</code>	
<code>\wedge</code>	
<code>\triangleright</code>	
<code>\odot</code>	⊙
<code>\star</code>	
<code>\dagger</code>	
<code>\bullet</code>	
<code>\ddagger</code>	
<code>\wr</code>	
<code>\amalg</code>	
<code>\leq</code>	
<code>\geq</code>	
<code>\equiv</code>	
<code>\models</code>	⊨
<code>\prec</code>	
<code>\succ</code>	
<code>\sim</code>	
<code>\perp</code>	⊥
<code>\preceq</code>	
<code>\succeq</code>	
<code>\simeq</code>	
<code>\ll</code>	≪
<code>\gg</code>	
<code>\asymp</code>	
<code>\parallel</code>	
<code>\subset</code>	⊂
<code>\supset</code>	⊃

```

\approx
\bowtie
\subseteq      Ł
\supseteq      Ł
\cong
\sqsubseteq    Ł
\sqsupseteq    Ł
\neg
\smile
\sqssubseteq    Ł
\sqsupseteq    Ł
\doteq
\frown
\in
\ni
\propto
\vdash         Ł
\dashv         Ł
\skull
\smiley
\blacksmiley
\frownie

```

### 3.3.8 Defining Custom Shortcuts

The Add User-defined Keybindings... menu item in the Keybindings sub-menu of Edit selects a file containing Scheme definitions of keybindings. The file must contain a module that uses a special keybindings language, `framework/keybinding-lang`. To do so, begin your file with this line:

```
#lang s-exp framework/keybinding-lang
```

The `framework/keybinding-lang` languages provides all of the bindings from `scheme`, `scheme/class`, and `drscheme/tool-lib`, except that it adjusts `#:module-begin` to introduce a `keybinding` form:

---

```
(keybinding string-expr proc-expr)
```

Declares a keybinding, where *string-expr* must produce a suitable first argument for `map-function` in `keymap%`, and the *proc-expr* must produce a suitable second argument for `add-function` in `keymap%`.

For example, this remaps the key combination “control-a” key to “!”.

```
#lang s-exp framework/keybinding-lang
(keybinding "c:a" (λ (editor evt) (send editor insert "!")))
```

Note that DrScheme does not reload this file automatically when you make a change, so you'll need to restart DrScheme to see changes to the file.

## 3.4 DrScheme Files

### 3.4.1 Program Files

The standard file extension for a PLT Scheme program file is ".ss". The extensions ".scm" and ".sch" are also popular.

DrScheme's editor can save a program file in two different formats:

- *Plain-text file format* — All text editors can read this format. DrScheme saves a program in plain-text format by default, unless the program contains images or text boxes. (Plain-text format does not preserve images or text boxes.)

Plain-text format is platform-specific because different platforms have different new-line conventions. However, most tools for moving files across platforms support a "text" transfer mode that adjusts newlines correctly.

- *Multimedia file format* — This format is specific to DrScheme, and no other editor recognizes it. DrScheme saves a program in multimedia format by default when the program contains images, text boxes, or formatted text.

Multimedia format is platform-independent, and it uses an ASCII encoding (so that different ways of transferring the file are unlikely to corrupt the file).

### 3.4.2 Backup and Autosave Files

When you modify an existing file in DrScheme and save it, DrScheme copies the old version of the file to a special backup file if no backup file exists. The backup file is saved in the same directory as the original file, and the backup file's name is generated from the original file's name:

- Under Unix and Mac OS X, a "~" is added to the end of the file's name.
- Under Windows, the file's extension is replaced with ".bak".

Every five minutes, DrScheme checks each open file. If any file is modified and not saved, DrScheme saves the file to a special autosave file (just in case there is a power failure or

some other catastrophic error). If the file is later saved, or if the user exists DrScheme without saving the file, the autosave file is removed. The autosave file is saved in the same directory as the original file, and the autosave file's name is generated from the original file's name:

- Under Unix and Mac OS X, a "#" is added to the start and end of the file's name, then a number is added after the ending "#", and then one more "#" is appended after the number. The number is selected to make the autosave filename unique.
- Under Windows, the file's extension is replaced with a number to make the autosave filename unique.

If the definitions window is modified and there is no current file, then an autosave file is written to the user's "documents" directory.

The "documents" directory is determined by `(find-system-path 'doc-dir)`.

### 3.4.3 Preference Files

On start-up, DrScheme reads configuration information from a preferences file. The name and location of the preferences file depends on the platform and user:

The expression `(find-system-path 'pref-file)` returns the platform- and user-specific preference file path.

- Under Unix, preferences are stored in a ".plt-scheme" subdirectory in the user's home directory, in a file "plt-prefs.ss".
- Under Windows, preferences are stored in a file "plt-prefs.ss" in a sub-directory "PLT Scheme" in the user's application-data folder as specified by the Windows registry; the application-data folder is usually "Application Data" in the user's profile directory, and that directory is usually hidden in the Windows GUI.
- Under Mac OS X, preferences are stored in "org.plt-scheme.prefs.ss" in the user's preferences folder.

A lock file is used while modifying the preferences file, and it is created in the same directory as the preferences file. Under Windows, the lock file is named "\_LOCKplt-prefs.ss"; under Unix, it is ".LOCK.plt-prefs.ss"; under Mac OS X, it is ".LOCK.org.plt-scheme.prefs.ss".

If the user-specific preferences file does not exist, and the file "plt-prefs.ss" in the "defaults" collection does exist, then it is used for the initial preference settings. (See §16.2 "Libraries and Collections" for more information about collections.) This file thus allows site-specific configuration for preference defaults. To set up such a configuration, start DrScheme and configure the preferences to your liking. Then, exit DrScheme and copy your preferences file into the "defaults" collection as "plt-prefs.ss". Afterward, users who have no preference settings already will get the preference settings that you chose.

## 4 Extending DrScheme

DrScheme supports two forms of extension to the programming environment:

- A *teachpack* extends the set of procedures that are built into a language in DrScheme. For example, a teachpack might extend the Beginning Student language with a procedure for playing sounds.

Teachpacks are particularly useful in a classroom setting, where an instructor can provide a teachpack that is designed for a specific exercise. To use the teachpack, each student must download the teachpack file and select it through the Language|Add Teachpack... menu item.

See §4.1 “Teachpacks” for information in creating teachpacks.

- A *tool* extends the set of utilities within the DrScheme environment. For example, DrScheme’s Check Syntax button starts a syntax-checking tool. For information on creating tools, see § “**Plugins**: Extending DrScheme”.

### 4.1 Teachpacks

Teachpacks are designed to supplement student programs with code that cannot be expressed in a teaching language. For example, to enable students to play hangman, we supply a teachpack that

- implements the random choosing of a word,
- maintains the state variable of how many guesses have gone wrong, and
- manages the GUI.

All these tasks are beyond students in the third week and/or impose memorization of currently useless knowledge on students. The essence of the hangman game, however, is not. The use of teachpacks enables the students to implement the interesting part of this exercise and still be able to enjoy today’s graphics without the useless memorization.

A single Scheme source file defines a teachpack (although the file may access other files via `require`). The file must contain a module (see §6 “Modules”). Each exported syntax definition or value definition from the module is provided as a new primitive form or primitive operation to the user, respectively.

As an example, the following teachpack provides a lazy cons implementation. To test it, be sure to save it in a file named `"lazycons.ss"`.

```
#lang scheme
```

```

(provide (rename-out [ :lcons lcons]) lcar lcdr)

(define-struct lcons (hd tl))

(define-syntax (:lcons stx)
  (syntax-case stx ()
    [(_ hd-exp tl-exp)
     #'(make-lcons
         (delay hd-exp)
         (delay tl-exp))]))

(define (lcar lcons) (force (lcons-hd lcons)))
(define (lcdr lcons) (force (lcons-tl lcons)))

```

Then, in this program:

```

(define (lmap f l)
  (lcons
   (f (lcar l))
   (lmap f (lcdr l))))

(define all-nums (lcons 1 (lmap add1 all-nums)))

```

the list `all-nums` is bound to an infinite list of ascending numbers.

For more examples, see the "htdp" sub-collection in the "teachpack" collection of the PLT installation.

## 4.2 Environment Variables

Several environment variables can affect DrScheme's behavior:

- `PLTNOTOOLS` : When this environment variable is set, DrScheme doesn't load any tools.
- `PLTONLYTOOL` : When this environment variable is set, DrScheme only loads the tools in the collection named by the value of the environment variable. If the variable is bound to a parenthesized list of collections, only the tools in those collections are loaded (The contents of the environment variable are [read](#) and expected to be a single symbol or a list of symbols).
- `PLTDRCM` : When this environment variable is set, DrScheme installs the compilation manager before starting up, which means that the ".zo" files are automatically kept up to date, as DrScheme's (or a tools) source is modified.

If the variable is set to `trace` then the compilation manager's output is traced, using the `manager-trace-handler` procedure.

- `PLTDRDEBUG` : When this environment variable is set, DrScheme starts up with `errortrace` enabled. If the variable is set to `profile`, DrScheme also records profiling information about itself.
- `PLTDRBREAK` : When this environment variable is set, DrScheme creates a window with a break button, during startup. Clicking the button breaks DrScheme's eventspace's main thread. This works well in combination with `PLTDRDEBUG` since the source locations are reported for the breaks.
- `PLTDRTESTS` : When this environment variable is set, DrScheme installs a special button in the button bar that starts the test suite. (The test suite is available only in the source distribution.)
- `PLTSTRINGCONSTANTS` : When this environment variable is set, DrScheme prints out the string constants that have not yet been translated. If it is set to a particular language (corresponding to one of the files in "string-constants" collection) it only shows the unset string constants matching that language.

This environment variable must be set when ".zo" files are made. To ensure that you see its output properly, run `setup-plt` with the `-c` flag, set the environment variable, and then run `setup-plt` again.

## Index

(define ...) button, 7  
".bak", 35  
".LOCK.org.plt-scheme.prefs.ss", 46  
".LOCK.plt-prefs.ss", 46  
".plt-scheme", 46  
".sch", 45  
".scm", 45  
".ss", 45  
> prompt, 11  
\aleph keyboard shortcut, 41  
\alpha keyboard shortcut, 42  
\amalg keyboard shortcut, 43  
\approx keyboard shortcut, 44  
\ast keyboard shortcut, 43  
\asympt keyboard shortcut, 43  
\beta keyboard shortcut, 42  
\bigtriangledown keyboard shortcut, 43  
\bigtriangleup keyboard shortcut, 43  
\blacksmiley keyboard shortcut, 44  
\bowtie keyboard shortcut, 44  
\bullet keyboard shortcut, 43  
\cap keyboard shortcut, 43  
\chi keyboard shortcut, 42  
\circ keyboard shortcut, 42  
\clubsuit keyboard shortcut, 41  
\cong keyboard shortcut, 44  
\cup keyboard shortcut, 43  
\dagger keyboard shortcut, 43  
\dashv keyboard shortcut, 44  
\ddagger keyboard shortcut, 43  
\Delta keyboard shortcut, 42  
\delta keyboard shortcut, 42  
\diamond keyboard shortcut, 43  
\diamondsuit keyboard shortcut, 41  
\div keyboard shortcut, 43  
\doteq keyboard shortcut, 44  
\Downarrow keyboard shortcut, 41  
\downarrow keyboard shortcut, 41  
\emptyset keyboard shortcut, 41  
\epsilon keyboard shortcut, 42  
\equiv keyboard shortcut, 43  
\eta keyboard shortcut, 42  
\exists keyboard shortcut, 42  
\flat keyboard shortcut, 41  
\forall keyboard shortcut, 42  
\frown keyboard shortcut, 44  
\frownie keyboard shortcut, 44  
\Gamma keyboard shortcut, 42  
\gamma keyboard shortcut, 42  
\geq keyboard shortcut, 43  
\gg keyboard shortcut, 43  
\heartsuit keyboard shortcut, 41  
\in keyboard shortcut, 44  
\infty keyboard shortcut, 42  
\kappa keyboard shortcut, 42  
\Lambda keyboard shortcut, 42  
\lambda keyboard shortcut, 42  
\Leftarrow keyboard shortcut, 41  
\leftarrow keyboard shortcut, 41  
\Leftrightarrow keyboard shortcut, 41  
\leftrightharrow keyboard shortcut, 41  
\leq keyboard shortcut, 43  
\ll keyboard shortcut, 43  
\longrightarrow keyboard shortcut, 41  
\models keyboard shortcut, 43  
\mp keyboard shortcut, 43  
\mu keyboard shortcut, 42  
\nabla keyboard shortcut, 41  
\natural keyboard shortcut, 42  
\nearrow keyboard shortcut, 41  
\neg keyboard shortcut, 42  
\neq keyboard shortcut, 44  
\ni keyboard shortcut, 44  
\nu keyboard shortcut, 42  
\narrow keyboard shortcut, 41  
\odot keyboard shortcut, 43  
\Omega keyboard shortcut, 42  
\omega keyboard shortcut, 42  
\ominus keyboard shortcut, 43  
\oplus keyboard shortcut, 43  
\oslash keyboard shortcut, 43  
\otimes keyboard shortcut, 43

$\backslash$ parallel keyboard shortcut, 43  
 $\backslash$ perp keyboard shortcut, 43  
 $\backslash$ Phi keyboard shortcut, 43  
 $\backslash$ phi keyboard shortcut, 42  
 $\backslash$ Pi keyboard shortcut, 42  
 $\backslash$ pi keyboard shortcut, 42  
 $\backslash$ pm keyboard shortcut, 43  
 $\backslash$ prec keyboard shortcut, 43  
 $\backslash$ preceq keyboard shortcut, 43  
 $\backslash$ prime keyboard shortcut, 41  
 $\backslash$ propto keyboard shortcut, 44  
 $\backslash$ Psi keyboard shortcut, 42  
 $\backslash$ psi keyboard shortcut, 42  
 $\backslash$ rho keyboard shortcut, 42  
 $\backslash$ Rightarrow keyboard shortcut, 41  
 $\backslash$ rightarrow keyboard shortcut, 41  
 $\backslash$ searrow keyboard shortcut, 41  
 $\backslash$ sharp keyboard shortcut, 41  
 $\backslash$ Sigma keyboard shortcut, 42  
 $\backslash$ sigma keyboard shortcut, 42  
 $\backslash$ sim keyboard shortcut, 43  
 $\backslash$ simeq keyboard shortcut, 43  
 $\backslash$ skull keyboard shortcut, 44  
 $\backslash$ smile keyboard shortcut, 44  
 $\backslash$ smiley keyboard shortcut, 44  
 $\backslash$ spadesuit keyboard shortcut, 41  
 $\backslash$ sqcap keyboard shortcut, 43  
 $\backslash$ sqcup keyboard shortcut, 43  
 $\backslash$ sqsubsetb keyboard shortcut, 44  
 $\backslash$ sqsubseteq keyboard shortcut, 44  
 $\backslash$ sqsupsetb keyboard shortcut, 44  
 $\backslash$ sqsupseteq keyboard shortcut, 44  
 $\backslash$ star keyboard shortcut, 43  
 $\backslash$ subset keyboard shortcut, 43  
 $\backslash$ subseteq keyboard shortcut, 44  
 $\backslash$ succ keyboard shortcut, 43  
 $\backslash$ succeq keyboard shortcut, 43  
 $\backslash$ supset keyboard shortcut, 43  
 $\backslash$ supseteq keyboard shortcut, 44  
 $\backslash$ surd keyboard shortcut, 42  
 $\backslash$ swarrow keyboard shortcut, 41  
 $\backslash$ tau keyboard shortcut, 42  
 $\backslash$ Theta keyboard shortcut, 42  
 $\backslash$ theta keyboard shortcut, 42  
 $\backslash$ times keyboard shortcut, 43  
 $\backslash$ triangle keyboard shortcut, 42  
 $\backslash$ triangleleft keyboard shortcut, 43  
 $\backslash$ triangleright keyboard shortcut, 43  
 $\backslash$ Uparrow keyboard shortcut, 41  
 $\backslash$ uparrow keyboard shortcut, 41  
 $\backslash$ Updownarrow keyboard shortcut, 41  
 $\backslash$ updownarrow keyboard shortcut, 41  
 $\backslash$ uplus keyboard shortcut, 43  
 $\backslash$ Upsilon keyboard shortcut, 42  
 $\backslash$ upsilon keyboard shortcut, 42  
 $\backslash$ varepsilon keyboard shortcut, 42  
 $\backslash$ varphi keyboard shortcut, 42  
 $\backslash$ varpi keyboard shortcut, 42  
 $\backslash$ varrho keyboard shortcut, 42  
 $\backslash$ varsigma keyboard shortcut, 42  
 $\backslash$ vartheta keyboard shortcut, 42  
 $\backslash$ vdash keyboard shortcut, 44  
 $\backslash$ vee keyboard shortcut, 43  
 $\backslash$ wedge keyboard shortcut, 43  
 $\backslash$ wr keyboard shortcut, 43  
 $\backslash$ Xi keyboard shortcut, 42  
 $\backslash$ xi keyboard shortcut, 42  
 $\backslash$ zeta keyboard shortcut, 42  
 "\_LOCKplt-prefs.ss", 46  
 A-C-down keybinding, 39  
 A-C-left keybinding, 39  
 A-C-right keybinding, 39  
 A-C-up keybinding, 39  
 About DrScheme... menu item, 34  
 Add Teachpack... menu item, 32  
 Add User-defined Keybindings... menu item, 30  
 Advanced Student language, 22  
 alpha renaming, 8  
 "Application Data", 46  
 automatic parenthesis, 9  
 Backup and Autosave Files, 45  
 Beginning Student language, 21  
 Beginning Student with List Abbreviations

- languages, 22
- Break button, 8
- Break menu item, 32
- Bring Frame to Front... menu item, 34
- Browser, 37
- Buttons, 7
- C-+ keybinding, 39
- C-\ keybinding, M-\ keybinding, 41
- C-\_ keybinding, 39
- C-a keybinding, 38
- C-b keybinding, 38
- C-c C-e keybinding, 40
- C-c C-l keybinding, 40
- C-c C-o keybinding, 40
- C-d keybinding, 39
- C-e keybinding, 38
- C-f keybinding, 38
- C-F6 keybinding, 39
- C-h keybinding, 39
- C-k keybinding, 39
- C-n keybinding, 38
- C-p keybinding, 38
- C-r keybinding, 40
- C-s keybinding, 40
- C-t keybinding, 39
- C-v keybinding, 38
- C-w keybinding, 39
- C-x C-s keybinding, 40
- C-x C-w keybinding, 40
- C-x u keybinding, 39
- C-y keybinding, 39
- changing a parenthesis as you type, 9
- Check syntax, question-mark arrows, 8
- Check syntax, purple arrows, 8
- Check Syntax button, 7
- Choose Language... menu item, 31
- Choosing a Language, 8
- Clear All Teachpacks menu item, 32
- Clear Error Highlight menu item, 32
- Clear menu item, 29
- Close menu item, 29
- Collapse menu item, 31
- Colors, 34
- Comment Out with a Box menu item, 33
- Comment Out with Semicolons menu item, 32
- Complete Word menu item, 30
- Constructor output, 27
- Copy menu item, 29
- Create Executable... menu item, 32
- Creating Executables, 19
- Cut menu item, 29
- cycle back through old expressions, 11
- Debug button, 16
- Debug button, 7
- debugger, 7
- Debugger Buttons, 16
- Debugging Multiple Files, 19
- Defining Custom Shortcuts, 44
- definitions window*, 6
- Definitions Window Actions, 17
- Delete menu item, 29
- Disable Tests menu item, 33
- [display](#), 12
- distribution archive*, 20
- Dock Test Report menu item, 31
- DrScheme Files, 45
- DrScheme Teachpacks, 47
- DrScheme:** PLT Programming Environment, 1
- Edit, 29
- Editing, 35
- Editing Operations, 39
- Editing with Parentheses, 9
- Emacs keybindings, 38
- Enable Tests menu item, 33
- Environment Variables, 48
- error highlighting, 12
- Errors, 12
- evaluating expressions, 11
- Extending DrScheme, 47
- F5 keybinding, 40
- File, 28
- file extension, 45

File Operations, 40  
 filename button, 7  
 Find Again Backwards menu item, 29  
 Find Again menu item, 29  
 Find Case Sensitive menu item, 30  
 Find... menu item, 29  
 flashing parenthesis matches, 9  
 Font, 34  
 formatting Scheme code, 9  
 Go button, 16  
 Graphical Debugging Interface, 16  
 Graphical Syntax, 15  
 gray highlight regions, 9  
 Help, 34  
 Help Desk menu item, 34  
 Hide Definitions menu item, 30  
 Hide Interactions menu item, 30  
 Hide Log menu item, 31  
 Hide Module Browser menu item, 30  
 Hide Profile menu item, 31  
 Hide Program Contour menu item, 30  
 Hide Tracing menu item, 31  
*How to Design Programs* Teaching Languages, 21  
 I/O, 12  
 Images, 15  
 indenting Scheme code, 9  
 Input and Output, 12  
 Insert, 33  
 Insert Comment Box menu item, 33  
 Insert Fraction... menu item, 33  
 Insert Image... menu item, 33  
 Insert Java Comment Box menu item, 33  
 Insert Java Interactions Box menu item, 33  
 Insert Large Letters... menu item, 33  
 Insert Pict Box menu item, 33  
 Insert Scheme Box menu item, 33  
 Insert Scheme Splice Box menu item, 33  
 Insert XML Box menu item, 33  
 Insert  $\lambda$  menu item, 33  
 Install PLT File... menu item, 28  
 Interact with DrScheme in English menu item, 34  
 Interactions, 41  
*interactions window*, 6  
 Interface Essentials, 5  
 Interface Reference, 28  
 Intermediate Student language, 22  
 Intermediate Student with Lambda language, 22  
 keybindings, 37  
 Keybindings menu item, 30  
 Keyboard Shortcuts, 37  
 Kill menu item, 32  
 Language, 31  
 language levels, 8  
 languages, extending, 47  
 Languages, 21  
 LaTeX, 41  
 LaTeX and TeX inspired keybindings, 41  
*launcher executable*, 20  
 Legacy Languages, 21  
 Limit Memory... menu item, 32  
 Log Definitions and Interactions... menu item, 28  
 M-( keybinding, 40  
 M-< keybinding, 38  
 M-> keybinding, 38  
 M-[ keybinding, 40  
 M-b keybinding, 38  
 M-C-b keybinding, 38  
 M-C-d keybinding, 38  
 M-C-down keybinding, 39  
 M-C-f keybinding, 38  
 M-C-k keybinding, 39  
 M-C-k keybinding, 40  
 M-C-left keybinding, 39  
 M-C-m keybinding, 40  
 M-C-p keybinding, 39  
 M-C-right keybinding, 39  
 M-C-SPACE keybinding, 38  
 M-C-t keybinding, 39  
 M-C-u keybinding, 38  
 M-C-up keybinding, 39

- M-d keybinding, 39
- M-DEL keybinding, 39
- M-f keybinding, 38
- M-n keybinding, 41
- M-o keybinding, 40
- M-p keybinding, 41
- M-S-L keybinding, 40
- M-t keybinding, 39
- M-v keybinding, 38
- M-w keybinding, 39
- M-{ keybinding, 40
- Menus, 28
- Miscellaneous, 40
- Module Browser... menu item, 32
- Module language, 21
- Modules, 21
- Most Recent Window menu item, 34
- Moving Around, 38
- Multimedia file format*, 45
- New menu item, 28
- Open Recent menu item, 28
- Open... menu item, 28
- "org.plt-scheme.prefs.ss", 46
- Other Experimental Languages, 26
- Out button, 17
- Output Printing Styles, 26
- Over button, 17
- overwrite mode, 40
- Paste menu item, 29
- Pause, 17
- Plain-text file format*, 45
- PLT Pretty Big* language, 21
- "PLT Scheme", 46
- "plt-prefs.ss", 46
- "plt-prefs.ss", 46
- "plt-prefs.ss", 46
- PLTDRBREAK, 49
- PLTDRCM, 48
- PLTDRDEBUG, 49
- PLTDRTESTS, 49
- PLTNOTOOLS, 48
- PLTONLYTOOL, 48
- PLTSTRINGCONSTANTS, 49
- Preference Files, 46
- Preferences, 34
- Preferences... menu item, 30
- previous expression, 11
- Print Definitions... menu item, 29
- Print Interactions... menu item, 29
- print output, 27
- printing format, 26
- ProfessorJ, 25
- ProfessorJ Advanced language, 25
- ProfessorJ Beginner language, 25
- ProfessorJ Full language, 25
- ProfessorJ Intermediate + access language, 25
- ProfessorJ Intermediate language, 25
- ProfessorJ Java + dynamic language, 25
- Profiling, 37
- Program Files, 45
- Quasiquote output, 27
- R5RS language, 21
- [read](#), 12
- [read-char](#), 12
- recycling icon, 7
- Redo menu item, 29
- Reindent All menu item, 32
- Reindent menu item, 32
- Related Web Sites menu item, 34
- Replace & Find Again Backwards menu item, 30
- Replace & Find Again menu item, 30
- Replace All menu item, 30
- Revert menu item, 28
- Run button, 8
- Run menu item, 32
- Save button, 7
- Save Definitions As Text... menu item, 28
- Save Definitions As... menu item, 28
- Save Definitions menu item, 28
- Save Interactions As Text... menu item, 28
- Save Interactions As... menu item, 28
- Save Interactions menu item, 28

- Save Other menu item, 28
- Scheme, 32
- Search, 40
  - search anchor*, 11
- Search in Files... menu item, 29
- Searching, 10
- Select All menu item, 29
- Show Active Keybindings menu item, 30
- Show Definitions menu item, 30
- Show Interactions menu item, 30
- Show Log menu item, 31
- Show Module Browser menu item, 30
- Show Profile menu item, 31
- Show Program Contour menu item, 30
- Show Tracing menu item, 31
- Split menu item, 31
- Stack View Pane, 18
- stand-alone executable*, 20
- status line*, 7
- Step button, 17
- Step button, 7
- Stepper, 7
- Swindle language, 21
- Tabbed Editing, 11
- tail calls, 8
- teachpack*, 47
- Teachpacks, 47
- The Interactions Window, 11
- tool*, 47
- Tool Web Sites menu item, 34
- Toolbar Hidden menu item, 31
- Toolbar menu item, 31
- Toolbar on Left menu item, 31
- Toolbar on Right menu item, 31
- Toolbar on Top menu item, 31
- Uncomment menu item, 33
- Undo menu item, 29
- Undock Test Report menu item, 31
- View, 30
- Warnings, 36
- Windows, 34
- Wrap Text menu item, 29
- [write](#), 12
- x-expression*, 16
- xexpr*, 16
- XML Boxes and Scheme Boxes, 16
- "~", 35
- $\alpha$ -rename, 8