# Introduction to stex

## R. Kent Dybvig and Oscar Waddell

### April 2016

## 1. Overview

The *stex* package consists of two main programs and some supporting items, such as make files, make-file templates, class files, and style files. The two main programs are **scheme-prep** and **html-prep**. **scheme-prep** performs a conversion from "stex"-formatted files into latex-formatted files, while **html-prep** converts (some) latex-formatted files into html-formatted files.

An stex file is really just a latex file extended with a handful of commands for including Scheme code (or pretty much any other kind of code, as long as you don't plan to use the Scheme-specific transcript support) in a document, plus a couple of additional features rather arbitrarily thrown in.

The subset of latex-formatted files **html-prep** is capable of handling is rather small but has nevertheless been useful for our purposes, which include producing html versions of a couple of books (*The Scheme Programming Language*, Editions 2–4 and the Chez Scheme User's Guides for Versions 6–9), the scheme.com web site, class websites, class assignments, and various other documents.

## 2. Installation

A prerequisite to building and using stex is to have Chez Scheme or Petite Chez Scheme installed on your system. You'll also need pdflatex, dvips, ghostscript, and netbpm. We've run stex under Linux and OS X but have not tried to run it under Windows.

The simplest way to install stex for your personal use is to clone the stex directory into your home directory, cd into the stex directory, and run make:

```
make BIN=bindir
```

where *bindir* is the directory where make will find the scheme or petite executables.

This will create a subdirectory, named for the installed Chez Scheme machine type, containing binary versions of the programs.

You can also use "make install" to make stex available for other users.

```
sudo make install BIN=bindir LIB=libdir
```

where *bindir* is as described above, and *libdir* is the directory where the stex library directory should be installed.

## 3. Usage notes

The simplest way to get started with stex is to get this document to build (in the doc directory) and create your own document by cloning this document's source file (stex.stex) and make file (Makefile). If you've installed stex in your home directory, you should be able to build this document by running "make" without arguments in the doc directory. If you've installed stex elsewhere, you'll first have to modify the include for Mf-stex to reflect its installed location.

Makefile is where you declare your stex sources and various other things, like bibtex and graphics files. If you don't have anything much more complicated than this document, you might just need to change the line that declares the main stex entry point, i.e., the line that reads `x = stex`, to reflect the name of your document.

The make is orchestrated by Mf-stex, which knows how to run **scheme-prep**, **html-prep**, **pdflatex** (multiple times), and various other commands to produce both pdf and html versions of the stex document.

You can also consult the more elaborate stex source and make files for the Chez Scheme User's Guide in the csug directory of a Chez Scheme release.

# 4. Basic stex commands

## 4.1. Inline code

An stex document includes inline Scheme (or other) code via the `\scheme` command, e.g.:

```
When called with two arguments, \scheme{cons} creates a pair of the two
arguments, e.g., \scheme{(cons 3 4)} produces \scheme{(3 . 4)}.
```

produces:

> When called with two arguments, `cons` creates a pair of the two arguments, e.g., `(cons 3 4)` produces `(3 . 4)`.

## 4.2. Code displays

An stex document includes out-of-line Scheme (or other) code via `\schemedisplay` and `\endschemedisplay`, e.g.:

```
\schemedisplay
(define fact
  (lambda (x)
    "a light year is a measure of distance"))

(define fib
  (lambda (x)
    "a light year is a measure of time"))
\endschemedisplay
```

produces:

```
(define fact
  (lambda (x)
    "a light year is a measure of distance"))

(define fib
  (lambda (x)
    "a light year is a measure of time"))
```

Within a Scheme display, `;=>` is converted into a double right arrow (⇒), `;->` into a single right arrow (→), and `;==` into a phantom of the same size. This is useful for showing what a piece of code translates or evaluates to, e.g.:

```
A \scheme{let} expression expands into a call to a \scheme{lambda}
expression, e.g.:
\schemedisplay
(let ([a 17]) ;-> ((lambda (a) (+ a a))
  (+ a a))    ;==  17)
\endschemedisplay

A \scheme{let} expression first evaluates the right-hand-side
expression, then evaluates the body in an environment that binds
the left-hand-side variable to the resulting value, e.g.:
\schemedisplay
(let ([a 17]) ;=> 17
  (+ a a))
\endschemedisplay
```

produces

> A `let` expression expands into a call to a `lambda` expression, e.g.:
>
> ```
> (let ([a 17])  →  ((lambda (a) (+ a a))
>   (+ a a))         17)
> ```
>
> A `let` expression first evaluates the right-hand-side expression, then evaluates the body in an environment that binds the left-hand-side variable to the resulting value, e.g.:
>
> ```
> (let ([a 17])  ⇒  17
>   (+ a a))
> ```

## 4.3. Variables

Code can include emphasized variables via the `\var` command, e.g.:

```
\scheme{(let ([\var{x} \var{e}]) \var{body})} binds the variable \var{x} to
the value of \var{e} in \var{body}.
```

produces:

> `(let ([`*x* *e*`])` *body*`)` binds the variable *x* to the value of *e* in *body*.

If the text within a `\var` form contains an underscore, the following character or bracketed subform is converted into a subscript, e.g., `\var{abc_3}` produces $abc_3$, and `\var{7e5_16}` produces $7e5_{16}$.

`\var` forms may appear within a `\scheme` form, within a Scheme display formed by `\schemedisplay` and `\endschemedisplay` commands, or by itself outside of either.

## 4.4. Raw text in code

Raw text can be included in code via the `\raw` command. For example:

```
\schemedisplay
(sqrt \raw{$x$}) \is \raw{$\sqrt{x}$}.
\endschemedisplay
```

produces:

> `(sqrt` $x$`)` $\Rightarrow$ $\sqrt{x}$.

### 4.5. Generated output

Output generated by a Scheme program can be inserted into the output via `\generated` and `\endgenerated` commands, e.g.:

```
\generated
(let ()
  (define fibs
    (lambda (x y n)
      (if (= n 0)
          '()
          (cons x (fibs y (+ x y) (- n 1))))))
  (let ([n 5])
    (printf "first ~r primes: ~{~s~^, ~}\n" n (fibs 0 1 n))))
\endgenerated
```

produces:

first five primes: 0, 1, 1, 2, 3

### 4.6. Verbatim Scheme displays

When special features, like `\var` forms, need to be suppressed within a Scheme display, a document can use `\schemeverbatim` and `\endschemeverbatim` instead of `\schemedisplay` and `\endschemedisplay`. This document makes extensive use of this feature.

## 5. Scheme transcripts

### 5.1. Automatic transcript generation

The **scheme-prep** package supports a `\transcript` command for automatically generating Scheme transcripts from input supplied in the document source. All text from the `\transcript` marker up to and including the `\endtranscript` marker is replaced with a transcript generated by supplying the intervening text as the input to a Scheme café (REPL). If the Scheme transcript needs to contain the sequence `\endtranscript`, a different terminator may be specified as an optional argument to `\transcript`. The terminator must be a backslash followed by one or more alphabetic characters, and is specified without the backslash in the optional argument.

Three pairs of commands may be redefined to customize the typesetting of different elements within generated transcripts. To modify the typesetting of error messages, redefine `\transerr` and `\endtranserr`. To modify the typesetting of user input read from the current input port of the café, redefine `\transin` and `\endtransin`. To modify the typesetting of program output written to the current output port of new café, redefine `\transout` and `\endtransout`.

For example, the following:

```
\transcript
(define f
  (lambda (x)       ; indentation and comments are
    (if (zero? x)   ; preserved in the transcript
        1
        (* x (f (- x 1))))))
(values f (f 0) (f 5) (f 20))
```

```
(trace f)
(f 4)
\endtranscript
```

produces:

```
> (define f
    (lambda (x)       ; indentation and comments are
      (if (zero? x)   ; preserved in the transcript
          1
          (* x (f (- x 1))))))
> (values f (f 0) (f 5) (f 20))
#<procedure f>
1
120
2432902008176640000
> (trace f)
(f)
> (f 4)
|(f 4)
| (f 3)
| |(f 2)
| | (f 1)
| | |(f 0)
| | |1
| | 1
| |2
| 6
|24
24
```

The following example shows how to specify a different transcript terminator and shows the default formatting imposed by `\transerr`, `\transin`, and `\transout`.

```
\transcript[\stopthistranscript]
#e4.5
(begin (display "Enter a character: ") (read-char))
(begin (display "Enter a character: ") (read-char))Z
(begin (clear-input-port) (display "Enter a character: ") (read-char))
Z
(list (read-char) (read-char) (read-char))abc def
(define silly-repl
  (lambda (prompt)
    (display prompt)
    (let ([x (read)])
      (unless (eof-object? x)
        (let ([result (eval x)])
          (unless (eq? result (void))
            (pretty-print result))
          (silly-repl prompt))))))
(silly-repl "Enter a Scheme expression: ")
(list 1
      2
      3)
(silly-repl "Now what? ")
(define interview
  (lambda ()
    (let* ([fname (begin (display "First name: ") (read))]
           [lname (begin (display "Last name: ") (read))])
      (printf "Hello ~a ~a!~%" fname lname))))
(interview)
john
  doe
#!eof
(printf "good to be back~%")
#!eof
\stopthistranscript
```

produces:

```
> #e4.5
9/2
> (begin (display "Enter a character: ") (read-char))
Enter a character: #\newline
> (begin (display "Enter a character: ") (read-char))Z
Enter a character: #\Z
> (begin (clear-input-port) (display "Enter a character: ") (read-char))
Enter a character: Z
#\Z
> (list (read-char) (read-char) (read-char))abc def
(#\b #\c #\a)
>
Exception: variable def is not bound
Type (debug) to enter the debugger.
> (define silly-repl
    (lambda (prompt)
      (display prompt)
      (let ([x (read)])
        (unless (eof-object? x)
          (let ([result (eval x)])
            (unless (eq? result (void))
              (pretty-print result))
            (silly-repl prompt))))))
> (silly-repl "Enter a Scheme expression: ")
Enter a Scheme expression: (list 1
                                 2
                                 3)
(1 2 3)
Enter a Scheme expression: (silly-repl "Now what? ")
Now what? (define interview
            (lambda ()
              (let* ([fname (begin (display "First name: ") (read))]
                     [lname (begin (display "Last name: ") (read))])
                (printf "Hello ~a ~a!~%" fname lname))))
Now what? (interview)
First name: john
```

```
Last name:   doe
Hello john doe!
Now what? #!eof
Enter a Scheme expression: (printf "good to be back~%")
good to be back
Enter a Scheme expression: #!eof
```

Transcripts do not include a trailing prompt by design. This is done in such a way that an explicitly displayed string that happens to look like the prompt is not suppressed. For example:

```
\transcript
(begin (display "> \n") (exit))
\endtranscript
```

should leave the apparent prompt alone since it is generated as program output.

```
> (begin (display "> \n") (exit))
>
```

Prompt suppression works even with changes to `waiter-prompt-string`. For example:

```
\transcript
(waiter-prompt-string "antelope? ")
"no thanks"
\endtranscript
```

produces no trailing "antelope? " prompt:

```
> (waiter-prompt-string "antelope? ")
antelope?  "no thanks"
"no thanks"
```

## 5.2. Loading initialization code

The `stex` commands `\schemeinit` and `\endschemeinit` are used to bracket Scheme expressions that should be evaluated without generating a transcript of the results. This is useful, for example, when writing the description of a programming assignment. The solutions can be loaded via `\schemeinit` and a transcript showing how the solutions behave can be generated using the `\transcript` command.

For example, the following text:

```
\schemeinit
(waiter-prompt-string ">")    ; restore the original prompt setting so we
                              ; don't get "antelope?" as the prompt
(define compute-length
  (lambda (x)
    (cond
      [(list? x) (length x)]
      [(vector? x) (vector-length x)]
      [(string? x) (string-length x)]
      [else (errorf 'compute-length "cannot handle ~s" ls)])))
```

```
\endschemeinit
\emph{The \scheme{compute-length} procedure behaves as follows:}
\transcript
(compute-length '())
(compute-length '(a b c))
(compute-length "abcd")
(compute-length (vector 1 2 3 4 5 6))
(compute-length compute-length)
\endtranscript
```

produces the output shown below.

*The `compute-length` procedure behaves as follows:*

```
> (compute-length '())
0
> (compute-length '(a b c))
3
> (compute-length "abcd")
4
> (compute-length (vector 1 2 3 4 5 6))
6
> (compute-length compute-length)

Exception in compute-length: cannot handle #<procedure compute-length>
Type (debug) to enter the debugger.
```

the last line of input intentionally causes an error, which is displayed just as it would be displayed in a café.

## 6. html-prep support for the `tabular` environment

Support for tables comes with a few caveats:

1. `@{}` directives within `tabular` column specifiers are flat-out ignored for the time being.
2. `|` directives within `tabular` column specifiers are essentially ignored. Their only effect is to globally enable borders for the entire HTML table.
3. `\hrule` and `\cline` are not yet implemented.
4. no warranty is expressed or implied.

The following code:

```
\begin{tabular}{rcl|r}
y &=& f(x) & without loss of generality \\
z & \multicolumn{2}{r}{whee} & this is fun? \\
\multicolumn{4}{c}{
 \begin{tabular}{cc}
   1 & 2 \\
   3 & 4
 \end{tabular}
} \\
a & b & c & d \\
12345 & z & \multicolumn{2}{l}{\scheme{(define~x~"foo")}}
\end{tabular}
```

generates this table:

| | | | |
|---|---|---|---|
| y | = | f(x) | without loss of generality |
| z | whee | | this is fun? |
| 1 2<br>3 4 | | | |
| a | b | c | d |
| 12345 | z | `(define x "foo")` | |