How to Use SLaTeX

Dorai Sitaram
dorai@cs.rice.edu
Department of Computer Science
Rice University
Houston, TX 77251–1892

1. Introduction

SLaTeX is a Scheme program that allows you to write programs or program fragments "as is" in your TeX or LaTeX source. SLaTeX is particularly geared to the programming languages Scheme and other Lisps, e.g., Common Lisp. The formatting of the code includes assigning appropriate fonts to the various tokens in the code (keywords, variables, constants, data), at the same time retaining the proper indentation when going to the non-monospace (non-typewriter) fonts provided by TeX. SLaTeX comes with two databases that recognize the identifier conventions of Scheme and Common Lisp respectively. These can be modified by the user using easy TeX commands. In addition, the user can inform SLaTeX to typeset certain identifiers as specially suited LaTeX expressions (i.e., beyond just fonting them). All this is done without interfering with the identifier conventions of the language of the programming code at all. In sum, no change need be made to your (presumably running) program code in order to get a typeset version suited to the particular need: you can get a spectrum of styles ranging from *no* fonting through basic default fonting to various "mathematical"-looking output for pedagogic or other reasons.

Other packages [4, 5] for typesetting code fragments use a `verbatim` environment where all the characters are in a `monospace typewriter font`. This `monospace` ensures that the indentation is not affected. However, the resulting output fails to distinguish between the various tokens used in the code, e.g., **boldface** for keywords like **define** and **lambda**, sans-serif for constants like #t and 42, and *italics* for variables such as *x* and *y* in (**lambda** (*x y*) (*cons x* (*cons y* '()))).

The program SLaTeX provides a convenient way of capturing the indentation information as well as assigning distinguishing fonts to code tokens without requiring the user to worry about fonting and spacing. It uses temporary files to store its typeset version of the user's code fragments and then calls TeX or LaTeX on the user's TeX files as well as these temporaries.

The following section will introduce you to the basic use of SLaTeX with a small example. Section 3 introduces the SLaTeX style files. Section 4 gives a complete description of all the SLaTeX control sequences. These include commands for manipulating output positioning, enhancing the database, changing the fonting defaults, adding special symbols, and selective disabling of SLaTeX. Section 5 desribes how to set up a preamble that reflects your typesetting taste. Section 6 contains information on obtaining and installing SLaTeX.

2. A quick illustration of using SLaTeX

This section presents a short example of SLaTeX use. We first look at a LaTeX file using SLaTeX commands, and then give a plain TeX version of the same file. We will see that there are minor differences between the ways SLaTeX is used with plain TeX and LaTeX (but see `\defslatex-envstyle` for a way to use the plain-TeX style with the LaTeX format, and conversely, the LaTeX style with the plain format).

2.1. For LaTeX users

Consider the following LaTeX (*and* SLaTeX) file `quick.tex`:

```
% quick.tex
\documentclass{article}
\usepackage{slatex}
%
%That was LaTeX2e.  If you use LaTeX2.09, or LaTeX2e in
%2.09 compatibility mode, use
%
%\documentstyle[slatex]{article}
%
%or
%
%\documentstyle{article}
%\input slatex.sty

In Scheme, the expression \scheme|(set! x 42)| returns
an unspecified value, rather than \scheme'42'.
However, one could get a \scheme{set!} of the latter
style by:

\begin{schemedisplay}
(define-syntax setq
  (syntax-rules ()
    [(setq x a)
     (begin (set! x a)
            x)]))
\end{schemedisplay}

\end{document}
```

First, the SLaTeX definitions in the style file `slatex.sty` are loaded into your LaTeX file. This may be done either via `\usepackage` or `\input` — or, if you use LaTeX 2.09, as an `\documentstyle` option.

In-text code is introduced by the SLaTeX control sequence `\scheme` and is flanked by a pair of identical characters that are not alphabets or "`{`". As a special convenient case, SLaTeX also allows the form `\scheme{...}`.

The SLaTeX control sequences for displayed code are the opening `\begin{schemedisplay}` and the closing `\end{schemedisplay}`.

The file is now SLaTeX'd by running the command `slatex` on it from the Unix or DOS command line:

```
slatex quick
```

or

```
slatex quick.tex
```

This calls a Scheme program `slatex.scm` that typesets the Scheme code fragments in `quick.tex` into temporary files. Thereafter, `quick.tex` along with the temporary files are then passed to LaTeX. (For information on judiciously reusing temporary files, see `\separateincludes`.) The resulting `quick.dvi` file, when viewed or printed looks like:

In Scheme, the expression (**set!** *x* 42) returns an unspecified value, rather than 42. However, one could get a **set!** of the latter style by:

```
(define-syntax setq
  (syntax-rules ()
    [(setq x a)
     (begin (set! x a)
            x)]))
```

Note that **setq**, although not normally a syntactic keyword in Scheme is nevertheless automatically recognized as such because of the context in which it occurs. No special treatment is needed to ensure that it will continue be treated as such in any subsequent Scheme code in the document.

2.2. For plain TeX users

Plain TeX users invoke SLaTeX much the same way, but for only two exceptions. First, since TeX doesn't have `\documentstyle`, the file `slatex.sty` is introduced via an `\input` statement before its commands can be used in the plain TeX source.

Second, since plain TeX does not have LaTeX's `\begin{`*env*`}` ... `\end{`*env*`}` style of environments, any environment commands in SLaTeX are invoked with the opening `\`*env* and the closing `\end`*env*.

The plain TeX version of `quick.tex` looks like:

```
% quick.tex
\input slatex.sty

In Scheme, the expression \scheme|(set! x 42)| returns
an unspecified value, rather than \scheme'42'.
However, one could get a \scheme{set!} of the latter
style by:

\schemedisplay
(define-syntax setq
  (syntax-rules ()
    [(setq x a)
     (begin (set! x a)
            x)]))
\endschemedisplay
\bye
```

The file is now SLaTeX'd by invoking `slatex` as before — SLaTeX is clever enough to figure out whether the file it operates on should later be send to LaTeX or plain TeX.

3. The style files

In short, the LaTeX (or TeX) file that is given to SLaTeX undergoes some code-setting preprocessing and is then handed over to LaTeX (or TeX). The style file `slatex.sty` defines the appropriate commands so that LaTeX (or TeX) can recognize the SLaTeX-specific directives and either process or ignore them. You may either `\input` the file `slatex.sty` as usual, or use it as the `\documentstyle` option `slatex`.

The default database of SLaTeX recognizes the keywords and constants of Scheme. The database can be modified with the commands `\setkeyword`, `\setconstant`, `\setvariable`, `\setspecialsymbol` and `\unsetspecialsymbol` (q.v.). If you're using Common Lisp rather than Scheme, use `cltl.sty` instead of `slatex.sty`. `cltl.sty` loads `slatex.sty` and modifies the database to reflect Common Lisp. You may fashion your own `.sty` files on the model of `cltl.sty`.

4. SLaTeX's control sequences

You've already seen the SLaTeX control sequence `\scheme` and the environment `{schemedisplay}`.[1] These suffice for quite a few instances of handling code. However, you will occasionally require more control on the typesetting process, and the rest of this section describes the complete[2] list of SLaTeX control sequences with examples.

`{schemedisplay}`

Typesets the enclosed code, which is typically several lines of code indented as you normally do in your Scheme files. E.g.,

```
\begin{schemedisplay}
(define compose          ;this is also known as $B$
  (lambda (f g)
    (lambda (x)
      (apply f (g x)))))
\end{schemedisplay}
is the ``compose'' function.
```

produces

(define *compose*   ;this is also known as B
  (lambda (*f g*)
    (lambda (*x*)
      (*apply f* (*g x*)))))

is the "compose" function.

As with all LaTeX environment enders, if the line after `\end{schemedisplay}` contains non-whitespace text, the paragraph continues. Otherwise — i.e., when `\end{schemedisplay}` is followed by at least one blank line — a fresh paragraph is started. Similarly, in plain TeX, a fresh

[1] The notation `{`*envname*`}` is used to denote an environment named *envname*. Usually, in plain TeX, this is the pair `\`*envname* and `\end`*envname*, while in LaTeX, it is the environment formed with `\begin{`*envname*`}` and `\end{`*envname*`}`. But see `\defslatexenvstyle`.

[2] At least that's what you're supposed to think . . .

paragraph is started after a `{schemedisplay}` only if `\endschemedisplay` is followed by at least one blank line.

Comments in Scheme are usually introduced by "`;`" (semicolon). The rest of the line after a "`;`" is set as a line in LaTeX LR mode.

Separate *blocks* of code can either be introduced in different `{schemedisplay}` environments or put in a single `{schemedisplay}` and separated by a line with a "`;`" in the first column. This "`;`" is not typeset and anything following it on the line is set in (La)TeX LR paragraph mode. Consecutive lines with "`;`" in the first column are treated as input for a TeX paragraph, with words possibly moved around from line to line to ensure justification. When in paragraph mode, the first line that has *no* leading "`;`" signals a fresh block of Scheme code within the `{schemedisplay}`. (The `{schemedisplay}` may end, or commence, on either a paragraph or a Scheme code block.)

E.g.,

```
\begin{schemedisplay}
(define even?              ; testing evenness
  (lambda (n)
    (if (= n 0) #t (not (odd? (- n 1))))))
; The procedures {\it even?} above
; and {\it odd?} below are mutually
; recursive.
(define odd?               ; testing oddness
  (lambda (n)
    (if (= n 0) #f (not (even? (- n 1))))))
\end{schemedisplay}
```

produces

(**define** *even?*   ; testing evenness
  (**lambda** (*n*)
    (**if** (= *n* 0) #t (*not* (*odd?* (− *n* 1))))))

The procedures *even?* above and *odd?* below are mutually recursive.

(**define** *odd?*   ; testing oddness
  (**lambda** (*n*)
    (**if** (= *n* 0) #f (*not* (*even?* (− *n* 1))))))

SLaTeX can recognize that blocks of code are separate if you have at least one empty line separating them. I.e., there is no need for empty "`;`" lines. This convenience is to accommodate Scheme files where definitions are usually separated by one or more blank lines.

Intervening paragraphs, either with lines with a leading "`;`", or with blank lines, are ideal spots for `{schemedisplay}` to allow pagebreaks. In fact, the default setting for `{schemedisplay}` also allows pagebreaks *within* a Scheme block, but it is easy to disable this (see entry for `\rightcodeskip`).

The space surrounding displayed Scheme code can be modified by setting the *skip*s `\abovecodeskip`, `\belowcodeskip`, `\leftcodeskip`, and `\rightcodeskip` (q.v.).

Note: see `{schemeregion}`.

`{schemeresponse}`

This is like `{schemedisplay}`, except that the code is displayed as the output of a Scheme evaluation, i.e., as data. In other words, keyword and variable fonts are disabled.

Note: see `\schemeresult` and `{schemeresponsebox}`.

↘

`\scheme`

Typesets its argument, which is enclosed in arbitrary but identical non-alphabetic and non-`{` characters, as in-text code. Special case: `\scheme{...}` is a convenience (provided the '`...`' doesn't contain a '`}`'). E.g., `\scheme|(call/cc (lambda (x) x))|` and `\scheme{(call/cc (lambda (x) x))}` both produce (*call/cc* (**lambda** (*x*) *x*)).

It *is* permitted to intermix calls to `{schemedisplay}` and `\scheme`. Thus,

```
\begin{schemedisplay}
(define factorial
  (lambda (n)
    (if (= n 0) ; \scheme{(zero? n)} also possible
        1 (* n (factorial (- n 1)))))) ; or \scheme{... (sub1 1)}
\end{schemedisplay}
```

produces

(**define** *factorial*
  (**lambda** (*n*)
    (**if** (= *n* 0) ; (*zero? n*) also possible
      1
      (* *n* (*factorial* (− *n* 1)))))) ; or . . . (*sub1* 1)

Note: see `{schemeregion}`.

↘

`\schemeresult`

Typesets its argument, which is enclosed in arbitrary but identical non-alphabetic and non-`{` characters, as in-text Scheme "result" or data: i.e., keyword and variable fonts are disabled. Special convenient case (as for `\scheme`): `\schemeresult{...}`. E.g.,

```
\scheme|((lambda () (cons 'lambda 'cons)))| yields
\schemeresult|(lambda . cons)|.
```

produces

((**lambda** () (*cons* 'lambda 'cons))) yields (lambda . cons).

↘

`{schemebox}`

The `{schemebox}` environment is similar to `{schemedisplay}` except that the code is provided as a "box" (i.e., it is not "displayed" in the standard way). Indeed, when the appropriate skip parameters are set, `{schemedisplay}` itself *may*[3] use a `{schemebox}` to create a box of code that is set off with all-round space as a display.

Saving a `{schemebox}` in an explicit box allows you to move your typeset code arbitrarily.

Note: see `{schemeregion}`.

[3] Yes, *may*: Not all `{schemedisplay}`s invoke `{schemebox}`, and if you're curious why, see entry for `\rightcodeskip`. It is a matter of whether pagebreaks within Scheme code are allowed or not.

↘

`{schemeresponsebox}`

This is like `{schemebox}`, except that the contents are displayed as Scheme data. See also `\schemeresult` and `{schemeresponse}`.

↘

`\schemeinput`

This can be used to input Scheme files as typeset code. (Unlike LaTeX's `\input`, `\schemeinput`'s argument must always be grouped.) The Scheme file can be specified either by its full name, or without its extension, if the latter is `.scm`, `.ss` or `.s`. E.g.,

```
\schemeinput{evenodd.scm}     % the .scm is optional!
```

(where `evenodd.scm` is the name of a Scheme file containing the code for *even?* and *odd?* above) produces the same effect as the `{schemedisplay}` version.

Note: see `{schemeregion}`.

↘

`{schemeregion}`

Calls to `\scheme`, `\schemeresult`, `{schemedisplay}`, `{schemebox}` or `\schemeinput` can be nested in (a Scheme comment) of other calls. In LaTeX text, they can occur in bodies of environments or otherwise grouped. However, they cannot normally be passed as arguments to macros or included in bodies of macro definitions, even though these are complete calls and not parameterized with respect to macro arguments. To be able to do this, you should cordon off such a text with the `{schemeregion}` environment. SLaTeX is fairly generous about where exactly you throw the cordon.

E.g., you cannot have

```
...
The code fragment
$\underline{\hbox{\scheme{(call/cc I)}}}$ is ...
...
```

but you *can* have

```
\begin{schemeregion}
...
The code fragment
$\underline{\hbox{\scheme{(call/cc I)}}}$ is ...
...
\end{schemeregion}
```

and this will produce

...

The code fragment $\underline{\textit{(call/cc I)}}$ is ...

...

Thus, the `{schemeregion}` environment makes it possible to put SLaTeX-specific commands inside macro arguments or macro definitions without causing rupture. Normally, this can't be done since SLaTeX-specific commands correspond to `comment`-like regions of LaTeX code once SLaTeX is done preprocessing your text. These `comment` regions share the characteristic of LaTeX's `verbatim` regions, which also can't appear in macro arguments or definitions.

To solve this, you enclose the offending text in a `{schemeregion}` environment. This "inlines" all the calls to SLaTeX in its body instead of commenting them and then invoking `\input`, thus escaping the fate described above. They are no-ops as far as non-SLaTeX commands are concerned. However, while a `{schemeregion}` allows its constituent SLaTeX commands to be included in macro arguments and bodies, it itself cannot be so included. Thus, your `{schemeregion}` should be in a position that satisfies the property A: either directly at the "top-level" or in a LaTeX environment that satisfies A. Since this recursive rule might look weird, you may just stick to calling `{schemeregion}` at the "top-level". Or, you may even wrap each of your LaTeX files in one huge `{schemeregion}` if you so wish. This will cover any obscure "non-robust" use of the SLaTeX primitives — however, SLaTeX will run slower. (The term "robust" is not necessarily used in the same sense as in LaTeX.)

Note that SLaTeX commands are made robust only if they are surrounded textually (lexically) by a `{schemeregion}`. A region marker doesn't have dynamic scope in the sense that LaTeX files loaded using `\input` from within a `{schemeregion}` will not inherit it. In summary, a `{schemeregion}` makes "robust" all calls to `\scheme`, `{schemedisplay}`, `{schemebox}` and `\schemeinput` within it.

↘

`\setkeyword`
`\setconstant`
`\setvariable`

SLaTeX has a database containing information about which code tokens are to be treated as **keywords**, which as constants, and which as *variables*. However, there will always be instances where the user wants to add their own tokens to these categories, or perhaps even modify the categories as prescribed by SLaTeX. The control sequences that enable the user to do these are `\setkeyword`, `\setconstant`, and `\setvariable`. Their arguments are entered as a (space-separated) list enclosed in braces (`{}`): SLaTeX learns that these are henceforth to be typeset in the appropriate font. E.g.,

```
\setconstant{infinity -infinity}
```

tells SLaTeX that infinity and -infinity are to be typeset as constants.

The user need not use `\setkeyword` specify such new keywords as are introduced by Scheme's and Common Lisp's syntactic definition facilities, viz., **define-syntax/syntax-rules**, **defmacro**, **extend-syntax**, **define-macro!**: SLaTeX automatically recognizes new macros and auxiliary keywords defined using these facilities.

↘

`\setspecialsymbol`
`\unsetspecialsymbol`

These commands are useful to generate "mathematical"-looking typeset versions of your code, over and beyond the fonting capabilities provided by default. For instance, although your code is restricted to using ascii identifiers that follow some convention, the corresponding typeset code could be more mnemonic and utilize the full suite of mathematical and other symbols provided by TeX. This of course should not require you to interfere with your code itself, which should run in its ascii representation. It is only the typeset version that has the new look. For instance, you might want all occurrences of `lambda`, `and`, `equiv?`, `below?`, `above?`, `a1` and `a2` in your code to be typeset as λ, $\wedge$, $\equiv$, $\sqsubseteq$, $\sqsupseteq$, a_1 and a_2 respectively. To do this, you should `\setspecialsymbol` the concerned identifier to the desired TeX expansion, viz.,

```
\setspecialsymbol{lambda}{$\lambda$}
\setspecialsymbol{and}{$\land$}
\setspecialsymbol{equiv?}{$\equiv$}
\setspecialsymbol{below?}{$\sqsubseteq$}
\setspecialsymbol{above?}{$\sqsupseteq$}
\setspecialsymbol{a1}{$a_1$}
\setspecialsymbol{a2}{$a_2$}
```

Now, typing

```
\begin{schemedisplay}
(define equiv?
  (lambda (a1 a2)
    (and (below? a1 a2) (above? a1 a2))))
\end{schemedisplay}
```

produces

(**define** $\equiv$
  (λ (a_1 a_2)
    ($\land$ ($\sqsubseteq$ a_1 a_2) ($\sqsupseteq$ a_1 a_2))))

Notethat with the above settings, `lambda` and `and` have lost their default keyword status, i.e., they will not be typed **boldface**. To retrieve the original status of special symbols, you should use `\unsetspecialsymbol`, e.g.

```
\unsetspecialsymbol{lambda and}
```

Typing the same program after unsetting the special symbols as above produces, as expected:

(**define** $\equiv$
  (**lambda** (a_1 a_2)
    (**and** ($\sqsubseteq$ a_1 a_2) ($\sqsupseteq$ a_1 a_2))))

In effect, `\setspecialsymbol` extends the basic “fonting” capability to arbitrary special typeset versions.

$\searrow$

`\schemecasesensitive`

SLaTeX always typesets output that is of the same case as your input, regardless of the setting of the `\schemecasesensitive` command. However, this command can be used to signal to SLaTeX that all case variations of an identifier are to be treated identically. E.g. typing `\schemecasesensitive{false}` implies that while `lambda` continues to be a keyword, so also are `Lambda`, `LAMBDA` and `LaMbDa`. `\schemecasesensitive{true}` reverts it back to the default mode where case is significant in determining the class of a token.

Note that the status `\schemecasesensitive` also affects the “special symbols” of the previous item. Thus, in the default case-*sensitive* setting, only the case-significant symbol as mentioned in the call to `\setspecialsymbol` will be replaced by the corresponding LaTeX expansion. In a case-*in*sensitive setting, all case variations of the special symbol will be replaced.

↘

```
\abovecodeskip
\belowcodeskip
\leftcodeskip
\rightcodeskip
```

These are the parameters used by `{schemedisplay}` for positioning the displayed code. The default values are

```
\abovecodeskip \medskipamount
\belowcodeskip \medskipamount
\leftcodeskip 0pt
\rightcodeskip 0pt
```

This produces a flushleft display. The defaults can be changed to get new display styles. E.g., this manual sets

```
\leftcodeskip\parindent
```

shifts the display from the left by the same amount as a paragraph indentation.

In both the above cases, the `{schemedisplay}` environment will be broken naturally across page boundaries at the right spot if the code is too long to fit a single page. In fact, automatic pagebreaks within the Scheme code are allowed if and only if `\rightcodeskip` is 0pt (its default value). For all other values of `\rightcodeskip`, each Scheme code block in a `{schemedisplay}` is guaranteed to be on the same page. If you have decided on a left indentation, and you're not sure what value to give to `\rightcodeskip`, but nevertheless don't want Scheme code broken across pages, you could set

```
\rightcodeskip=0.01pt %or
\rightcodeskip=0pt plus 1fil
```

The following explains why the above disable page breaks within the Scheme block. For example, suppose you'd set

```
\leftcodeskip=0pt plus 1fil
\rightcodeskip=0pt plus 1fil
```

This will get you a *centered* display style. This is of course because the skip on each side of the code produces a "spring" [3] that pushes the code to the center. But for this spring action to work nicely, the code must have been collected into an unbreakable box — which is precisely what `{schemedisplay}` does for each of its code blocks whenever it notices that the prevailing value of `\rightcodeskip` is not the default zero.[4]

It is this behind-the-scenes selective boxing that dictates whether a `{schemedisplay}` block can or cannot be broken across a page boundary. And the value of `\rightcodeskip` is used to govern this selection in a "reasonable" manner.

↘

```
\keywordfont
\constantfont
\variablefont
```

[4] 0pt plus 1fil $\neq$ 0pt

These decide the typefaces used for keywords, constants, and variables. The default definitions are:[5]

```
\def\keywordfont#1{{\bf#1}}
\def\constantfont#1{{\sf#1}}
\def\variablefont#1{{\it#1\/}}
```

This is close to the Little Lisper [2] style. Redefine these control sequences for font changes. As an extreme case, defining all of them to `{{\tt#1}}` typesets everything in monospace typewriter font, as, for instance, in SICP [1].

↘

```
\defschemedisplaytoken
\defschemetoken
\defschemeboxtoken
\defschemeresulttoken
\defschemeresponsetoken
\defschemeresponseboxtoken
\defschemeinputtoken
\defschemeregiontoken
```

These define the tokens used by SLaTeX to trigger typesetting of in-text code, display code, box code, and Scheme files. The default tokens are, as already described, `{schemedisplay}`, `\scheme`, `{schemebox}`, `\schemeresult`, `{schemeresponse}`, `{schemerespon-sebox}`, `\schemeinput` and `{schemeregion}` respectively. If you want shorter or more mnemonic tokens, the `\defscheme*token` control sequences prove useful. E.g., if you want `\code` to be your new control sequence for in-text code, use `\defschemetoken{code}`. All instances of `\code+...+` after this definition produce in-text code, unless overridden by an `\un-defschemetoken` command.

One can have at any time any number of tokens for the same activity. One consequence of this is that one can have nested `{schemeregion}`s, provided one has different names for the nested call. Otherwise, the `\end` of an inner region will prematurely terminate an outer region.

↘

```
\undefschemedisplaytoken
\undefschemetoken
\undefschemeboxtoken
\undefschemeresulttoken
\undefschemeresponsetoken
\undefschemeresponseboxtoken
\undefschemeinputtoken
\undefschemeregiontoken
```

These *un*define the tokens used for triggering typesetting in-text code, display code, box code, Scheme files, and robust Scheme regions. Use these if you want to use these tokens for other purposes and do not want to unwittingly trip up the SLaTeX system.

[5] `\sf` in LaTeX is the sans-serif font. Since plain TeX does not define `\sf`, you may define one before `\input`ing `slatex.sty`. If no `\sf` is found, `slatex.sty` has `\let\sf=\rm`. You may redefine it before or after to an appropriate font of your choice. This manual used `\font\sf=cmss10`. Of course, you could redefine `\constantfont` itself to not rely on `\sf`.

↘

`\defschememathescape`
`\undefschememathescape`

`\defschememathescape{$}` defines the character `$` as a mathematical escape character to be used within scheme code. (Any character other than `}` and whitespace may be chosen instead of `$`.) This allows one to use LaTeX-like mathematical subformulas within Scheme code, e.g.,

```
\defschememathescape{$}

\begin{schemedisplay}
(define $\equiv$
  (lambda (a$_1$ a$_2$)
    ($\land$ ($\sqsubseteq$ a$_1$ a$_2$)
             ($\sqsupseteq$ a$_1$ a$_2$))))
\end{schemedisplay}
```

produces

(**define** $\equiv$
  (**lambda** (a_1 a_2)
    ($\land$ ($\sqsubseteq$ a_1 a_2)
      ($\sqsupseteq$ a_1 a_2))))

`\undefschememathescape{$}` disables the math-escape nature, if any, of `$`.

↘

`\slatexdisable`

The tokens for typesetting code, as also the token `\input` (which is sensitive to SLaTeX, since the latter uses it to recursively process files within files), can only be used as calls. If they occur in the bodies of macro definitions, or their names are used for defining other control sequences, SLaTeX will not be able to process them. Sometimes, one wants to use these tokens, say `\input`, without having SLaTeX try to process the inputted file. Or the name `\scheme` may be used in a verbatim environment, and we don't want such an occurrence to trigger the codesetting half of SLaTeX to look for code.

Avoiding such uses altogether can be unduly restrictive.[6] One way out is to judiciously use the `\undefscheme*token` commands to temporarily remove the SLaTeX-specificity of these names. Even this can be painful. SLaTeX therefore provides the commands `\slatexdisable`. This takes one argument word and makes the corresponding control sequence out of it. Further, from this point in the text, SLaTeX is disabled *until* the manufactured control sequence shows up. This mechanism makes it possible to restrict SLaTeX to only appropriate portions of the text. Note that the token `\slatexdisable` itself can appear in the text succeeding its call. The only token that can restore SLaTeX-sensitivity is the one created during the call to `\slatexdisable`.

A typical example of the use of `\slatexdisable` is when you use the names `\scheme` and `\begin{schemedisplay}` in a `verbatim` environment. E.g.,

```
\slatexdisable{slatexenable}
\begin{verbatim}
```

[6] Especially when one is writing a "How to ..." manual like this where one both uses *and* mentions the control sequences!

```
SLaTeX provides the command \scheme and the pair
\begin{schemedisplay} and \end{schemedisplay} to typeset
in-text and displayed Scheme code respectively.
\end{verbatim}
\slatexenable
```

produces the required

```
SLaTeX provides the command \scheme and the pair
\begin{schemedisplay} and \end{schemedisplay} to typeset
in-text and display Scheme code respectively.
```

↘
`\slatexignorecurrentfile`
This is a SLaTeX pragma included to improve efficiency. If you're sure that the remaining portion of a certain LaTeX (or TeX) file (including the files that would be `\input`ed by it) don't contain any SLaTeX commands, then you may place this control sequence in it at this point to signal SLaTeX that no preprocessing is necessary for the rest of the file.

↘
`\defslatexenvstyle`
As section 2 showed, the differences in SLaTeX usage between plain TeX and LaTeX is simply a matter of the difference in the "environment" styles of the two formats. It is easy get the behavior of the one format with the other.

1. If you wish to use the plain-TeX style in LaTeX, type

   ```
   \defslatexenvstyle{tex}
   ```

   before first such use.
2. Similarly, if you wish to use the LaTeX `\begin`/`\end` style in plain TeX, use

   ```
   \defslatexenvstyle{latex}
   ```

   provided you have already defined `\begin` *and* `\end` *appropriately!* One way to accomplish this is:

   ```
   \def\begin#1{\begingroup
     \csname#1\endcsname
     \let\end\endenvironment}
   \def\endenvironment#1{\csname end#1\endcsname
     \endgroup}
   ```

   Here, `\end` is defined within a group because TeX already has an `\end` command whose global definition is used by commands such as `\bye` and therefore should not be changed lightly.

In either case, you can revert to the default style with `\defslatexenvstyle{latex}` and `\defslatexenvstyle{tex}` respectively.

↘
`\slatexseparateincludes`

By default, the temporary files of SLaTeX use the name of the topmost TeX file, i.e., the name stored under `\jobname`. In large LaTeX documents using `\include`, this may be unduly restrictive.

To recapitulate, the `slatex` command creates temporary files to store typeset code and then passes the baton on to TeX or LaTeX. If no significant change has been made to the Scheme code (either in content or in relative positioning) in the document, then successive calls to (La)TeX could be made directly using the old temporary files. This could be a time-saver, since it avoids calling up the Scheme typesetter.

However, in a large LaTeX document with `\include`s, these successive calls to LaTeX often entail juggling the `\include`s that are chosen. In this case, even though the relative position of the Scheme code is preserved within each `\include`d file, the sequence perceived by the main file changes. This spoils the invariance we needed if we'd wanted to avoid calling SLaTeX unnecessarily.

To solve this, the SLaTeX command sequence `\slatexseparateincludes` — which must be called before the first occurrence of Scheme code in your document — guarantees that each `\include`d file will generate its own pool of temp files. Thus, if the SLaTeX files are created once for each `\include`, they will be correctly loaded no matter what sequence of `\include`s is taken.

↘
`\schemecodehook`

The user can define `\schemecodehook` to be anything. The hook will be evaluated inside each subsequent call to `{schemedisplay}` and `{schemebox}`. E.g., if you have `\tiny` defined (as in LaTeX) to reduce font size, then

```
\let\schemecodehook\tiny
```

converts your Scheme displays and boxes into small print.

The default value of the hook is `\relax`, a no-op.

5. Setting up a file that resets SLaTeX's defaults

A sample style modification file for SLaTeX would include redefinition of the names of the codesetting control sequences, adjustment of the display parameters, modification of the font assignments for keywords/constants/variables/special symbols, and addition of new keywords/constants/variables/special symbols to SLaTeX's database.

Let's assume you want

1. a centered display style with no vertical skips;
2. the names `\code`, `{schemefrag}`, `{scmbox}`, `\sinput` instead of `\scheme`, `{schemedisplay}`, `{schemebox}` and `\schemeinput`;
3. tokens to disregard case;
4. the keywords to come out it `typewriter`, the constants in roman, and the variables in *slant*;
5. "`und`" and "`oder`" as keywords, "`true`" and "`false`" as constants, "`define`" as a variable (overriding default as keyword!), "`F`" as a constant (`f` will also be a constant, due to case-insensitivity!);
6. "`top`" and "`bottom`" to print as $\top$ and $\bot$ respectively.

This could be set up as

```
\abovecodeskip 0pt
\belowcodeskip 0pt
\leftcodeskip 0pt plus 1fil
\rightcodeskip 0pt plus 1fil

\undefschemetoken{scheme}
\undefschemeboxtoken{schemebox}
\undefschemedisplaytoken{schemedisplay}
\undefschemeinputtoken{schemeinput}

\defschemetoken{code}
\defschemeboxtoken{scmbox}
\defschemedisplaytoken{schemefrag}
\defschemeinputtoken{sinput}

\schemecasesensitive{false}

\def\keywordfont#1{{\tt#1}}
\def\constantfont#1{{\rm#1}}
\def\variablefont#1{{\sl#1\/}}

\setkeyword{und oder}
\setconstant{true false}
\setvariable{define}
\setconstant{F}

\setspecialsymbol{top}{$\top$}
\setspecialsymbol{bottom}{$\bottom$}
```

This file can then be `\input` in the preamble of your LaTeX document.

6. How to obtain and install SLaTeX

SLaTeX is available via anonymous ftp — its URL is `ftp://cs.rice.edu/public/dorai/slatex`*NN*`.tar.gz`, where *NN* is some number. Login as `anonymous`, give your userid as password, change to the directory `public/dorai`, convert to `bin` mode, and get the file `slatex`*NN*`.tar.gz`, where *NN* is some number. Ungzipping and untarring produces a directory `slatex`, containing the SLaTeX files. (The file `manifest` lists the files in the distribution — make sure that nothing is missing.)

To install SLaTeX on your system:

1. First change directory (`cd`) to `slatex`, the directory housing the SLaTeX files.[7]
2. Edit the file `config.dat` as suggested by the comments in the file itself.
3. Invoke your Scheme or Common Lisp interpreter. Load the file `config.scm`, i.e., type

   (*load* "config.scm")

   at the Scheme (or Common Lisp) prompt. This will configure SLaTeX for your Scheme dialect and operating system, creating a Scheme file called `slatex.scm`. (If you informed

[7] Some of the SLaTeX files use DOS-style CR-LF newlines. You may want to use an appropriate newline modifier to the SLaTeX files to make the files comply with your operating system's newline format.

config.dat that your Scheme dialect is Chez, the file slatex.scm is a compiled version rather than Scheme source.) The configuration process also creates a batch file slatex.bat (on DOS) or a shell script slatex (on Unix), for convenient invocation of SLaTeX from your operating system command line. A Scheme/Common Lisp file callsla.scm is also created — this lets you call SLaTeX from the Scheme/Common Lisp prompt.

4. Exit Scheme/Common Lisp.

To set up paths and modify shell script/batch file:

1. Copy (or move, or link) slatex.scm into a suitable place, e.g., your bin or lib directory, or the system bin or lib.
2. Copy (or move, or link) slatex.sty into a suitable place, i.e., somewhere in your TEXINPUTS path. For installing on a multiuser system, place in the directory containing the LaTeX files (on mine this is /usr/local/lib/tex/macros).
3. Copy (or move, or link) the shell script slatex or the batch file slatex.bat to a suitable place in your PATH, e.g., your bin or the system bin directory. Note that slatex(.bat) sets *SLaTeX.*texinputs**. If you're making the same shell script (or batch file) available to multiple users, you should change the line

   (**set!** *SLaTeX.*texinputs** "...")

   to

   (**set!** *SLaTeX.*texinputs** (*getenv* "TEXINPUTS"))

   or some other dialect-dependent way of obtaining the TEXINPUTS environment variable.
4. Run slatex on slatxdoc.tex (this file!) for documentation. (This also serves as a check that SLaTeX does indeed work on your machine.) Refer to slatxdoc.dvi when befuddled.

If your dialect did not allow a nice enough shell script or batch file, the following provides an alternate route to unlocking SLaTeX.

6.1. Other ways of invoking SLaTeX

The configuration process creates shell script/batch file slatex(.bat) for a standard invoking mechanism for SLaTeX. The shell script/batch file is created to exploit the way your Scheme is called, e.g., matters like whether it accepts echo'd s-expressions (e.g., Chez) , whether it loads command line files (e.g., SCM) , and whether it always checks for an "init" file (e.g., MIT C Scheme).

1. If your Scheme doesn't fall into either of these categories, you may have to write your own shell script/batch file or devise some other mechanism.
2. The shell script/batch file invokes Scheme/Common Lisp. If, however, you are already in Scheme/Common Lisp and spend most of the time continuously at the Scheme/Common Lisp prompt rather than the operating system prompt, you may avoid some of the delays inherent in the shell script/batch file.

The file callsla.scm, which contains just one small procedure named *call-slatex*, and which is created by the configuration process, provides a simple calling mechanism from Scheme/Common Lisp, as opposed to the operating system command line. You may use it as an alternative to the slatex(.bat) shell script/batch file. The usage is as follows: load callsla.scm into Scheme/Common Lisp

(*load* "callsla.scm")

and type

(*call-slatex* ⟨*tex-file*⟩)

when you need to call SLaTeX on the (La)TeX file ⟨*tex-file*⟩. This invokes the SLaTeX preprocessor on ⟨*tex-file*⟩. If your Scheme has a *system* procedure that can call the operating system command line, *call-slatex* will also send your file to TeX or LaTeX. If your Scheme does not have such a procedure, *call-slatex* will simply prod you to call TeX or LaTeX yourself.

The outline of the shell script/batch file or `callsla.scm` or of any strategy you devise for using SLaTeX should include the following actions:

1. Load the file `slatex.scm` (created by the configuration process) into Scheme/Common Lisp.
2. Set the variable *SLaTeX.*texinputs** to the path `TEXINPUTS` or `TEXINPUT` used by TeX[8] to look for `\input` files.
3. Call the procedure *SLaTeX.process-main-tex-file* on the `.tex` file to be processed.
4. Call either `latex` or `tex` on the `.tex` file.

You may devise your own way of calling *SLaTeX.process-main-tex-file*, provided your method makes sure that `slatex.scm` has been loaded, *SLaTeX.*texinputs** set appropriately *before* the call and `latex`/`tex` is called *after* the call.

Note that if you prefer to stay in Scheme/Common Lisp most of the time, it is a good idea to pre-load the procedure *call-slatex*, perhaps through an "init" file. *call-slatex* is just a "one-liner" "call-by-need" hook to SLaTeX and does not take up much resources. (Global name clashes between your own code and SLaTeX code won't occur unless you use variable names starting with "*SLaTeX.*") If you made no calls to *call-slatex*, the bigger file `slatex.scm` is not loaded at all. If you make several calls to *call-slatex*, `slatex.scm` is loaded only once, at the time of the first call.

6.2. Dialects SLaTeX runs on

SLaTeX is implemented in R4RS-compliant Scheme (macros are not needed). The code uses the non-standard procedures *delete-file*, *file-exists?* and *force-output*, but a Scheme without these procedures can also run SLaTeX (the configuration defines the corresponding variables to be dummy procedures, since they are not crucial). The distribution comes with code to allow SLaTeX to run also on Common Lisp. The files `readme` and `install` contain all the information necessary to configure SLaTeX for your system.

SLaTeX has been tested successfully in the following dialects:[9]

On OS/2:
SCM; CLISP.

On Unix:
SCM; Chez Scheme; Ibuki Common Lisp (1987); MIT C Scheme; Elk; Scheme-to-C; UMB Scheme.

---

[8] There is some variation on the name of this environment variable. Unix TeX's prefer `TEX-INPUTS` with an `S`, while DOS (e.g., Eberhard Mattes's emTeX) favors `TEXINPUT` without the `S`.

[9] The authors and copyright holders of the respective Schemes/Lisps are: Scm — Aubrey Jaffer; Chez Scheme — R. Kent Dybvig; Elk — Oliver Laumann; Scheme-to-C — Joel Bartlett; UMB Scheme — William Campbell; MIT C Scheme — MIT; Austin Kyoto Common Lisp — William Schelter's (who modified the KCL of) Taiichi Yuasa and Masami Hagiya's KCL; Ibuki Common Lisp — Ibuki; CLISP — Bruno Haible and Michael Stoll.

On MS-DOS:

SCM; MIT C Scheme; Austin Kyoto Common Lisp; CLISP.

If your Scheme is not mentioned here but *is* R4RS-compliant, please send a note to the author at `dorai@cs.rice.edu` describing your Scheme's procedures for deleting files, testing file existence, and forcing output, if any, and the configuration file will be enhanced to accommodate the new dialect.

Bug reports are most welcome — send to `dorai@cs.rice.edu`.

Bibliography

[1] H. Abelson and G. J. Sussman with J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, 1985.

[2] D. P. Friedman and M. Felleisen. The Little Lisper. Science Research Associates, 1989.

[3] D. E. Knuth. The TeXbook. Addison-Wesley, 1984.

[4] J. Ramsdell. SchemeWeb. Scheme Repository, `ftp://iuvax.cs.indiana.edu/pub/scheme-repository/utl/`.

[5] C. Queinnec. LiSP2TeX. Scheme Repository.

Index