# TEX←Scm3−1/10*
## file: texscm3.scm – Dialog by TEX←Scm3 – 1/10:

Keith Wright

April 5, 2026 at 14:37 EST



## Contents





# Preface: Building and Copying

This is a literate programming [5] tool for the Scheme programming language [18, 4], inspired by Knuth's Web [8, 7] and Plato's Dialogs [11].

This program is distributed under the terms of the GNU General Public Licence (GPL) version 2. Except it is not distributed because it is not done yet.

> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> A copy of the GNU General Public License is given in the appendix to this document. Further copies can be obtained by writing to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

If you are reading a typeset document, skip ahead to the introduction, otherwise make a document by running LaTeX on `texscm.tex` and read that. If there is no `texscm.tex`, make one. If `guile` and LaTeX are both installed properly, `make` should work, otherwise make what the Makefile needs be made.

I run GNU Guile 3.0.8 Scheme in emacs by `alt-x run-scheme` and find it useful to have this in my home directory `.guile`

```
(display "loading .guile")(newline)
(display "-- loading texscm.scm & tstprocs.scm")(newline)
(display "-- (reload) to load them again")(newline)
(define (reload)
 (load "srckaw/texscm/texscm.scm")
 (load "srckaw/texscm/tstprocs.scm"))
(reload)
```

# Preface: Version

Version "$3 - 1/9$" is saved as `texscmv23` . This is version "$3 - 1/10$" and it is *ts:make-reply* only. The objective is to make it work in accord with the manual and to delete deprecated code.

The *zip* and *unzip* procedures along with most of their subs have been removed. The Sorting and Tabbing procedures are not called, but have not been removed. They must be rewritten to work with the new version.

At first I was reluctant to go to two digit reciprocal version numbers. I made version "$3 - 1/A$" thinking I would use hex digits. I got to thinking about "$3 - 1/G$" and wondering what Unicode comes after the Roman alphabet. That way madness lies.

Today's plan is to make experimental subversions of a reciprocal integer version that have (almost) the same Scheme code, but different procedures are called by the `main` procedure. The subversion is a single letter set at run-time. It is printed on the title page of the LaTeX TeX←`Scm` document and can be reused.

# 1 Introduction

Philosophy consists of a series of footnotes[1]
— Whitehead [22, Part II,Ch.I§I,p.63]

The file `texscm.scm` contains a Scheme program which should be ready to load into a Scheme implementation and run without further ado (no special pre-processing needed).

The procedures in that file will read a file which contains an ASCII encoded Scheme program and print it nicely. It may seem to do that in an unnecesarily complicated way. No doubt, some of this complication is just a mistake on my part, but some of the complication results because this program is a small part of a much larger project.

Some Pretty Print programs take an unformatted program (without meaningful spacing and indentation) and format it by inserting line breaks, spaces, and indentation. The editor `emacs` can do that for Scheme programs. This program takes a program that has meaningful spacing already in the text. It leaves the indentation and line breaks as they are in the source; we assume that the programmer had reasons for putting the line breaks as they are and therefore they should not be auto-broken.

Instead, this program turns the Scheme source into a TEX file. Comments are formatted as TEX, while the Scheme commands and definitions are automatically printed in various fonts depending how they are used—keywords in bold face, variables in italic, and symbols in sans serif. Automatic choice of font is an exercise in *really* understanding macros.

The main purpose is to explain the program to a human programmer. It does little good to have compiler input (the "source code") if it is obfuscated; yet even binary machine language can be understood and used if it is accompanied by a good explanation.

The file `poly.scm` contains (what I like to think is) a good example of a Scheme program that can be processed by TEX←`Scm` and LATEX to produce `poly.ps`, which contains the resulting output. You should look at both of those before reading this program. When you are ready to read TEX←`Scm`, §2.4 contains examples of interactive use.

This program (TEX←`Scm` version 3-1/n) suffices to format itself (you are reading its output) and `poly.scm` version 3, but it is still under development. Unexpected input will result in pathological formatting, or a total crash.

Originally, all comments were in blocks with three semicolons at the start of each line, but version 2.2 can process the new sharp-bar (i.e. `#`|these|`#`) comments. It also uses them, so older versions can not process version 2.2 or newer.

Version 3 processes remarks. There may be intermediate versions that tried to process remarks but failed. Any Scheme implementation will treat remarks as comments and ignore them. TEX←`Scm` should at least not crash, but version $3 - 1/n$ might.

I am using version numbers with reciprocal integers subtracted for pre-release that don't really work yet; as integers increase I get closer to the destination version.

This program works with Scheme programs encoded in several ways

1. external representations,
   (a) ASCII or UTF-8, as defined in the R$^n$RS Repeatedly Revised Reports.
   (b) TEX or its printed result
2. internal representations, which can be easily written and read back by a computer
   (a) Lexeme lists
   (b) S-expressions i.e. ⟨datum⟩s

For more details on character encoding, see §8.1 of Appendix B, page 83. For more details on Lexeme lists, see §8.2 page 87.

We also want to save the results of evaluation of the program. Sometimes the results should be displayed together with the program as part of the same document, to be read by a person;

[1] to Plato

sometimes the results should be saved to be read by the computer for regression testing; sometimes both. The results can also be encoded in any of the ways listed above.

### 1.1 Set Up Configuration and Title Page

We need the *read-line* procedure. It is an R$^7$RS standard, but GNU guile has it in a special module.

```
(cond-expand (guile (use-modules (ice-9 rdelim )) ))
```

The next few definitions set some parameters that determine the content of the title page of the document that will be produced.

The title page will always show the name of the input file and the version of TEX←Scm that processed it, that is, the version of the program you are now reading. The `input-file-name` is taken automatically from the command line. The version is hard-coded in the text of this program below.

```
(define ts:input-file-name "TBD" )
(define ts:version "$3-1/10$" )
(define ts:subversion "" )
```

This is a table of names and default values of parameters for the title page. The values can be changed by editorial tremarks in the program in the input file.

```
(define ts:parameters    '(   (program-title . "Gun␣Hunting" )
                              (copyright . "\\copyright\\␣N.U.G." )
                              (author . "by␣Ung" )
                              (title-foot . "\\noindent␣Thank␣Gnu!" )   )   )
(define (ts:parm-val parm-name )
        (let ((res (assq parm-name ts:parameters )))
             (if res (cdr res ) "???" ) ))
```

These editorial remarks set up the title page of TEX←Scm itself.

2025-10-24: make-reply does not print them. 2025-10-27: But they get done. 2925-10-29: they are printed, but with extra blank line after 2025-11-22 xx version requires blank comments to keep one remark per block

:= program-title "\\TeXScm3$-1/10$"

:= author "Keith␣Wright"

:= copyright "Copyright␣\\copyright\\␣2010\\ldots␣2025␣by␣Keith␣Wright"

:= title-foot "\\noindent␣␣Permission␣to␣copy␣is␣granted␣under␣Creative␣Commons␣BY/SA␣licence␣or␣GPL

## 2 Read and Write Scheme lexemes and blocks

**Lexemes** The lexemes are pairs, the first member of the pair is a lexeme type, the second member is either a string or a list.

The lexeme types that can be read from a file are: Id, Number, NewLine, WhtSpc, Abbrev, Semicolon, Char, String, Boolean, Open, VOpen, Close, Dot, Error, SemiSharp, Remark

— These are not lexemes, but replaced by blocks CmntShort, CmntLong.

The end of file is a lexeme of type EOF, if blocks are read from several files in succession the EOF breaks any ⟨datum⟩, resets line numbers...

TEX←Scm changes the type of some lexemes from Id to one of Vb, Kw, or Sy. This is called "sorting identifiers".

```
(define (lxm-type lx )
  (if (eof-object? lx ) 'EOF (and (pair? lx )(symbol? (car lx ))   (car lx ))))
```

They are all capitalized to make them easy to recognize (—should they all start with Lx?). This program should continue to work if all identifiers are converted to a single case because there are no symbols or identifiers that are the same except for case. On the other hand, NewLine and newline are different symbols, so don't mix them. We use only NewLine as a symbol; *newline* is a variable; `"newline"` is a string.

Since this is intended for boot-strapping, it reads only Ascii in the program text (no higher Unicode in identifiers except between ⟨vertical line⟩s). We do not know Unicode character classes, but non-Ascii characters as ⟨comment text⟩, ⟨symbol element⟩ or in strings are simply passed through.

See the appendix [§8.1, page 84) for more detailed information on the syntax of lexemes.

The few following procedures should be made local or deleted when blocks work.

Procedure *WhtSpc-col* gets the column at which the whitespace ends.

```
(define WhtSpc-col caddr )
(define (n-WhtSpc lxm )
  (let ((arg (cadr lxm )))
    (if (number? arg )
        arg
        (if (pair? arg )(+ (cdr arg )(* 8 (car arg )))) ))) #|???|#
```

a one line comment just before a datum gets misplaced in *-r.scm but two lines are OK! Should is-token? be false for SharpBar?

```
(define (is-token? lexeme )
  (case (lxm-type lexeme )
    ((Semicolon SemiSharp NewLine WhtSpc EndLine ) #f )
    (else #t ) ))
(define (is-comment? lexeme )
(case (lxm-type lexeme )
    ((Semicolon ??? ) #t )
    (else #f ) ))
```

Version 2.1 of this program had a function *read-lexeme* which read one character at a time until it had read a complete lexeme. This updated version reads a line at a time. To update it without breaking it, I changed all calls to it to calls to *ts:read-lexeme*, and assigned (**set!** *ts:read-lexeme read-lexeme*). Then I wrote a new procedure which I called *:read-lexeme* and I could easily switch between them by setting *ts:read-lexeme* to one or the other of either *read-lexeme* or *:read-lexeme* . That is now obsolete and I have totally removed the old procedures. The only trace remaining is that there are several procedures with names that start with a colon. They are analogous to built-in procedures with the same name without the colon. They are: *:peek-char*, *:read-char*, *:display*.

I used the same strategy to replace *read-datum-as-lexemes* by *read-data-blk*.

**BUG:** There is no code to recognize peculiar identifiers other than"+", "−". Italic plus looks funny. Maybe give them a new format `\pvb{}` (peculiar variable) instead of `\vb{}` (variable).

## 2.1 Reading from a Scheme file (a big let)

The following **let** expression keeps a table of character types, *char-types*, one line of text, *line-buf*, and a few indices as local variables. It exports procedures by assigning them to global variables. In version 2.2 *ts:read-lexeme* is the only one of these left. In version 3 it will be replaced by *ts:read-block*. It is defined here and set to its final value at the end of the big **let** on page 16.

Procedure *ts:read-lexeme* expects the *peek-index* to point to the start of the lexeme to be read. It reads and returns that lexeme, and leaves the *peek-index* pointing at a *peek-char* which is the next character (occurrence) after the lexeme which was just read.

These three variables will have real values filled in later.

```
(define ts:read-lexeme #f )
(define ts:read-block #f )
```

2025-11-20: without this define it still makes but gets Unbound variable: xx:read-block

```
(define xx:read-block #f )
```

The character's type is higher if it is more likely to be part of the preceding and ongoing lexeme. In other words, low types catch attention. The character types are listed below. See the Appendix §8.1, page 83 for the definition of these terms.

4 — initial

3 — subsequent, not initial

2 — delimiter
1 — non-ascii
0 — unworthy

```
(let ( (char-types
          (let ( (v (make-vector 128 0 )) )
            #| We classify characters by making strings of characters in each class and
               initializing the char-type vector by scanning them. |#
            (define letters "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" )
            (define digits "0123456789" )
            (define hex-digits "0123456789abcdefABCDEF" )
            (define special-initials "!$%&*/:<=>?@^_~" )
            (define delimiters "␣\t\n|()\";" )
            (define inline-hex-escapes "" )
            (define special-subsequents "+-." )
            (define initials (string-append letters special-initials inline-hex-escapes ))
            (define subsequents (string-append initials digits special-subsequents ))
            (define (set!-type s t )
              (let ((n (string-length s )) )
                (do ((k 0 (+ 1 k )))
                    ((>= k n ) #f )
                  (vector-set! v (char→integer (string-ref s k )) t ))))
            (set!-type delimiters 2 )
            (set!-type subsequents 3 )
            (set!-type initials 4 )
            #| The initalized vector v is bound to char-types.|#
            v ))
        (line-buf #f )
        (peek-index 0 )
        (line-number 0 )
        (sharp-bar-cmnt-depth 0 ) #|This is not used.|#
    )
  #| Procedure type-of gets the type of a character from the char-type table.
     There are no non-ascii (type 1) characters in this table because any
     character not in the table is non-ascii. |#
  (define (type-of ch )
    (let ((c (char→integer ch )))
      (if (>= c 128 ) 1 (vector-ref char-types c ))))
  (define (tp-less-index max-type start str )
    (let ( (len (string-length str )) )
      (let scan ((k (+ 1 start )))
        (if (< k len )
            (if (<= (type-of (string-ref str k )) max-type )
                k
                (scan (+ 1 k )) )
            len ) )))
  (define (find-index ch start str )
    (let ( (len (string-length str )) )
      (let scan ( (k start ) )
        (if (< k len )
            (if (char=? ch (string-ref str k ))
                k
                (scan (+ 1 k )))
```

```
                #f ) )))
(define (cut-trailing-blanks line )
   (if (not (string? line ))
       line
       (let ( (len (string-length line )) )
          (if (> len 0 )
              (let ( (end (string-ref line (- len 1 ))) )
                 (if (or (char=? end #\space )(char=? end #\tab ))
                     (cut-trailing-blanks (substring line 0 (- len 1 )))
                     line ))
              line ))))
(define (fill-line )
   (begin
      (set! line-number (+ 1 line-number ))
      (set! line-buf (cut-trailing-blanks (read-line )))
      (set! peek-index 0 ) ))
(define (semicolon-count )
   (let scan ( (k 0 )
               (ch (:peek-char )) )
      (if (char=? ch #\; )
          (begin (:read-char ) (scan (+ 1 k ) (:peek-char )))
          k )))
(define (read-rest-of-line )
   (let ( (rest (substring line-buf peek-index (string-length line-buf )) ))
      (set! peek-index (string-length line-buf ))
      rest ))
(define (remove-atmos lexs )
   (if (null? lexs )
       '()
       (if (eq? 'WhtSpc (lxm-type (car lexs )))
           (remove-atmos (cdr lexs ))
           (cons (car lexs )(remove-atmos (cdr lexs ))) )))
```

Procedure *read-line-to-bar-sharp* should be called after reading sharp-bar (`#|`), possibly on a preceding line. It reads and returns everything up to the matching bar-sharp (`|#`) provided it is on the current line. Otherwise it unreads it all and returns false having read nothing.

```
(define (read-line-to-bar-sharp depth )
   (let    ( (start-index peek-index ) )
      (let find-end ( (bar-index (find-index #\| peek-index line-buf )) )
         (if bar-index
             (begin
                (set! peek-index bar-index )
                (if (string=? "|#" (string (:peek-char ) (peek-after )))
                    (begin
                       (:read-char )
                       (:read-char )
                       (substring line-buf start-index bar-index ) )
                    (begin
                       (:read-char )
                       (find-end (find-index #\| peek-index line-buf )) )))
             (begin
                (set! peek-index start-index )
                #f )))))
```

The Scheme Report calls sharp-bar comments "nested", but nesting does not actually work here. See §8.3.1 on page 94. This may be more complicated than simple recursion on depth, because comments are LaTeX input and "#" is a special character in macros.

This is first called with peek-index just after a sharp-bar, or the first character on the line if the sharp-bar was on a preceding line. It returns either a character string, if the comment ends on the same line, or a list of them, with partial lines at start and end.

```
(define (read-nested-comment depth )
  (let ( (ln0 (read-line-to-bar-sharp depth )) )
    (or ln0
        (let* ( (st (read-rest-of-line ))
                (rest-cmnt (begin
                             (fill-line )
                             (if (eof-object? line-buf )
                                 (list 'Error "end␣of␣file␣before␣|#" )
                                 (read-nested-comment depth )))))
          (cons st (if (string? rest-cmnt ) (list rest-cmnt ) rest-cmnt )) ))))
(define (read-delimited-word )
  (let* ( (pk peek-index )
          (del-index    (tp-less-index 2 peek-index line-buf )) )
    (set! peek-index del-index )
    (substring line-buf pk del-index )) )
(define (read-hex-digits )
  (let ((word (read-delimited-word )))
    word ))
(define (read-hex-number )
  (string→number (read-delimited-word ) 16 ))
(define (hex-digit? ch )
  (or (char<=? #\a ch #\f ) (char<=? #\A ch #\F ) (char<=? #\0 ch #\9 )) )
(define (letter? ch )
  (or (char<=? #\a ch #\z ) (char<=? #\A ch #\Z )) )
(define (read-sharp-lexeme )
  (begin
    (:read-char )
    (case (:peek-char )
      #| — BUG: this borks #false and #true |#
      ((#\f #\F ) (begin (:read-char ) '(Boolean . #f )))
      ((#\t #\T ) (begin (:read-char ) '(Boolean . #t )))
      ((#\\ ) (begin
                (:read-char )
                (if (eq? (:peek-char ) #\x )
                    (begin (:read-char )
                           (if (hex-digit? (:peek-char ))
                               (cons 'hex-char (read-hex-number ))
                               (cons 'Char #\x )))
                    (if (letter? (:peek-char ))
                        (let ((name (read-delimited-word )))
                          (if (= 1 (string-length name ))
                              (cons 'Char (string-ref name 0 ))
                              (cons 'Char name )))
                        (cons 'Char (:read-char )) ))))
      ((#\( ) (begin (:read-char ) '(VOpen )))
      ((#\! ) (cons 'Directive    (read-delimited-word )))
```

```
        ((#\| ) (begin
                  (:read-char )
                  (let ( (txt (read-nested-comment 1 )) )
                    (if (string? txt )
                        (cons 'CmntShort txt )
                        (cons 'CmntLong txt ) ) )))
        (else (list 'Error "unknown sharp" (:peek-char ))) )))
(define (read-abbrev-token )
  (let ( (ch (:peek-char )) )
    (:read-char )
    (case ch
      ((#\' ) 'quote )
      ((#\` ) 'quasiquote )
      ((#\, )
       (if (char=? (:peek-char ) #\@ )
           (begin (:read-char )    'unquote-splicing )
           'unquote ))
      (else (list 'Error "abbrev error" )) )))
```

(*self-contained-lexeme* ch) reads a lexeme that need not be followed by a delimiter; result is `#f` if it does not find one. in the case of Semicolon and SemiSharp read to #\newline but do not read that.

```
(define (self-contained-lexeme )
      (case (:peek-char )
        ((#\( )       (begin (:read-char ) '(Open )))
        ((#\) )       (begin (:read-char ) '(Close )))
        ((#\; )       (if (eq? (peek-after ) #\# )
                          (cons 'SemiSharp
                                (begin (:read-char )(:read-char )
                                       (read-lexemes-on-line )))
                          (cons 'Semicolon (cons (semicolon-count )
                                                 (read-rest-of-line ))) ))
        ((#\, #\' #\` )
                       (cons 'Abbrev (read-abbrev-token )))
        ((#\" )       (cons 'String (read-string-elements )))
        ((#\# )       (read-sharp-lexeme ))
        ((#\| )       (begin
                        (:read-char )
                        (let ( (vbar (find-index #\| peek-index line-buf ))
                               (start-tok peek-index ) )
                          (if vbar
                              (begin
                                (set! peek-index (+ 1 vbar ))
                                (cons 'Id (substring line-buf start-tok vbar )))
                              (list 'Error "unpaired vertical bar " )) )))
        (else #f ) ) )
(define (is-id? word )
  (if (or (string=? word "-" )(string=? word "+" )
          (string=? word "->" )(string=? word "..." ))
      #t
      (if (< (type-of (string-ref word 0 )) 4 )
          #f
          (let check-rest ((k 1 ))
            (if (>= k (string-length word ))
                #t
```

```
                    (if (< (type-of (string-ref word 0 )) 3 )
                        #f
                        (check-rest (+ 1 k ))))) )))
(define (is-number? word )
  (string→number word ))
(define (is-dot? word ) (equal? word "." ))
#| Procedure delimited-lexeme reads everything up to (but not including)
the next delimiter, and then classifies what it finds. |#
(define (delimited-lexeme )
  (let ( (word (read-delimited-word )) )
    (cons
     (cond
      ((is-number? word ) 'Number )
      ((is-dot? word )    'Dot )
      ((is-id? word )     'Id )
      (else 'Error ))
     word )) )
(define (read-string-elements )
  (let read-more ( (text (begin (:read-char ) "" )) )
    (if (eq? (:peek-char ) #\" )
        (begin (:read-char ) text )
        (begin
          (read-more
           (string-append
            text
            (if (char=? (:peek-char ) #\\ )
                (begin
                  If we see a backslash, look at the next character and interpret the
                  mnemonic escape as an actual character. Return a one character string[a].
                  The :peek-char has been seen to be a backslash, so the next :read-char
                  gets it and the following one gets the escaped character. Note that the
                  footnote goes at the bottom of the comment. Is there a way to put it at
                  the bottom of the page?
                  [a] Is there a way to have the underlying system do this? Using eval? It could use
                  display and read, but that would bring in the whole file system.
                  (:read-char )
                  (let ((ch (:read-char )))
                    (case ch
                      ((#\\ ) "\\" )
                      ((#\" ) "\"" )
                      ((#\t ) "\t" )
                      ((#\n ) "\n" )
                      ((#\r ) "\r" )
                      ((#\x )
                       (let ( (scalar (read-hex-digits )) )
                         (if (char=? #\; (:peek-char ))
                             (begin
                               (:read-char )
                               (string (integer→char scalar )) )
                             "\\x(not␣hex;)" )))
                      (else "\\???" )     )))
                (string (:read-char ))))))))
```

```
#| Procedure white-space reads and returns one white lexeme, either NewLine or WhtSpc, or returns false without reading. |#
(define white-space
  (let ( (tab-adjust 0 ) )
    (lambda ()
      (let scan
          (   (ch0 (:peek-char ))
              (blanks 0 )
              (tbaj tab-adjust ) )
        (if (eof-object? ch0 ) (set!     ch0 #\? ))       #| any non-blank |#
        (cond
         ((char=? ch0 #\newline )
              (:read-char )
              (set! tab-adjust 0 )
              (list 'NewLine line-number ) )
         ((char=? ch0 #\space )
              (:read-char )
              (scan (:peek-char ) (+ 1 blanks ) tbaj ) )
         ((char=? ch0 #\tab )
              (:read-char )
              (let ((xsp (- 8 (modulo (+ tbaj peek-index ) 8 ))))
                 (scan (:peek-char ) (+ 1 xsp blanks ) (+ xsp tbaj )) ))
         (else    (set! tab-adjust tbaj )
                  (if    (= blanks 0 )
                         #f
                         (list 'WhtSpc blanks (+ peek-index tab-adjust )) )) ) ))))
(define (:peek-char )
  (if (not line-buf ) (fill-line ))
  (if (eof-object? line-buf )
      (let ((eof line-buf ))
        (set! line-buf #f )
        (set! peek-index 0 )
        eof )
      (if (< peek-index (string-length line-buf ))
          (string-ref line-buf peek-index )
          #\newline )))
(define (peek-after )
  (if (< (+ 1 peek-index ) (string-length line-buf ))
      (string-ref line-buf (+ 1 peek-index ))
      #\newline )   )
(define (:read-char )
  (let ((ch (:peek-char )))
    (if (and line-buf (< peek-index (string-length line-buf )))
        (set! peek-index (+ peek-index 1 ))
        (fill-line ) )
    ch ))
#| Procedure read-lexemes-on-line reads to NewLine and returns a list of lexemes on the current line. Don't read the \n! It can't be unread! Next read will get it. What needs an unread besides sharpbar? |#
(define (xx:read-lexemes-on-line )
  (let scanline ( (lst '()) )
    (if (or (eof-object? (:peek-char ))(char=? (:peek-char ) #\newline ))
        (reverse (cons (list 'NewLine 0 ) lst ))
        (scanline (cons (read-lexeme ) lst )) )))
```

```
#| Let's not but say we did. Without Newline line-buf is not flushed. |#
(define (read-lexemes-on-line )
  (let scanline ( (lst '()) )
    (if (or (eof-object? (:peek-char ))(char=? (:peek-char ) #\newline ))
        (reverse (cons (list 'EndLine 0 ) lst ))
        (scanline (cons (read-lexeme ) lst )) )))
```

We call *read-lexeme* to get the next lexeme. NewLine is a lexeme. (white-space) refills the buffer at `\n`

BUG!!! end of file in middle of lexeme (stray quote) crashes

```
(define (read-lexeme )
  (let ( (ch1 (:peek-char )) )
    (if (eof-object? ch1 )
        '(EOF )
        (or (white-space )
            (self-contained-lexeme )
            (delimited-lexeme ) )) ))
```

Read white space. Return number of lines and spaces. It quits reading when the white space ends. That is, the next call to *read-lexeme* will return the next lexeme that is not whitespace.

Lines and spaces constitute a ⟨white2⟩. No distinction is maintained between blank lines that contain only spaces and tabs, and blank lines that contain nothing at all. Imagine the ⟨white2⟩ is a two dimensional rectangle. It should be called with the peek-char at the first character on the line after the end of the preceding block (or first in file).

```
(define (read-white2 lines indent )
  (let ( (lxm (white-space )) )
    (if lxm
        (if (eq? (lxm-type lxm ) 'NewLine )
            (read-white2 (+ lines 1 ) 0 )
            (if (eq? (lxm-type lxm ) 'WhtSpc )
                (read-white2 lines (WhtSpc-col lxm ))))
        (cons lines indent ) )))
```

Procedure *read-remark-blk* is called, only by *read-block*, after reading a SemiSharp lexeme, which marks the start of a remark. The SemiSharp and all lexemes on the rest of the line are one compound lexeme. In particular *lxm* is a list of SemiSharp followed by all lexemes on this line.

After ";#=>" read lexemes:

Returns a list of lexemes, which includes all ⟨token⟩s and ⟨atmo⟩s after initial SemiSharp and each following line that starts with a properly indented SemiSharp.)

After ";#->" the lexemes are all semicolon comments. This allows arbitrary strings to be written without escapes.

For now we don't make such continuation blocks.

lxm is certainly a type SemiSharp, which contains the lexemes on the rest of the line in its cdr. Next read gets the NewLine.

```
#| should return a list of remarks and results that belong in one block|#
(define (rr:read-remark-blk     white2 lxm )
  (let collect-block ((blk (list (rr:read-one-remark white2 lxm ))))
    (let* ( (rdw (read-white2 0 0 ))
            (rlx (read-lexeme )) )
      (if (and (eq? (lxm-type rlx ) 'SemiSharp )
               (= (cdr white2 ) (cdr rdw )) )
          (collect-block
```

```
                    (append blk (list (rr:read-one-remark rdw rlx ))    ))
                  (begin (set! peek-index 0 ) blk )) )))
(define (dbg:read-remark-blk     white2 lxm )
  (let collect-block ((blk (list (dbg:read-one-remark white2 lxm ))))
    (let* ( (rdw (read-white2 0 0 ))
            (rlx (read-lexeme )) )
      (debug "rdrem␣blk=[" blk "]␣rlx=" rlx )
      (if (and (eq? (lxm-type rlx ) 'SemiSharp )
               (= (cdr white2 ) (cdr rdw )) )
          (collect-block
           (append blk (list (dbg:read-one-remark rdw rlx ))    ))
          (begin (set! peek-index 0 ) blk )) )))
```

#| called with lxm a 'SemiSharp cons lexemlist on rest of line return all lexemes in an extended remark. SemiSharp is one line; Remark includes continuations. |#

#| this seems to put continuation lines outside the SemiSharp |#

#| (let collect-remark ( (lines (list (cdr lxm)) )) works when there is no continuation |#

```
(define (rr:read-one-remark white2 lxm )
  (let collect-remark ( (line-lxs (cdr lxm )) )
    (let* ( (rdw (read-white2 0 0 ))
            (rlx (read-lexeme )) )
      (if (and (eq? (lxm-type rlx ) 'SemiSharp )
               (= (cdr white2 ) (cdr rdw ))
               (eq? (lxm-type (cadr rlx )) 'WhtSpc ) )
          (collect-remark (append line-lxs '((NewLine 0 )) (cdr rlx ) ))
          (begin (set! peek-index 0 )
                 (cons 'Remark (list line-lxs ))) ))))
(define (dbg:read-one-remark white2 lxm )
  (let collect-remark ( (line-lxs (cdr lxm )) )
    (let* ( (rdw (read-white2 0 0 ))
            (rlx (read-lexeme )) )
      (debug "onerem␣lnlx=[" line-lxs "]␣rlx=" rlx )
      (if (and (eq? (lxm-type rlx ) 'SemiSharp )
               (= (cdr white2 ) (cdr rdw ))
               (eq? (lxm-type (cadr rlx )) 'WhtSpc ) )
          (collect-remark (append line-lxs '((NewLine 0 )) (cdr rlx ) ))
          (begin (set! peek-index 0 )
                 (cons 'Remark (list line-lxs ))) ))))
```

This is called after reading some whitespace and then semicolons. Those are passed as parameters. We must collect the strings of all following comments that have the same indentation and semicolon count. Return a list of comment strings that belong together in one block.

#| |#

```
(define (read-cmnt-blk white2 lxm )
  (let collect ((line (cddr lxm ))
                (lines '()) )
    (let* ( (rdw (read-white2 0 0 ))
            (rlx (read-lexeme )) )
      (if (and (eq? (lxm-type rlx ) 'Semicolon )
               (= (cdr white2 ) (cdr rdw )) )
          (collect (cddr rlx ) (cons line lines ))
          (begin (set! peek-index 0 )
                 (reverse (cons line lines ))) )) )
```

```
   Procedure (read-data-blk lxm) reads lexemes until parentheses balance
   and the line ends. It returns a list of lexemes that encode just one com-
#| plete ⟨datum⟩ with atmosphere and possibly a few more short ⟨datums⟩. |#
   It is an error if the line ends with an incomplete datum. (What about
   an incomplete sharp-bar?) This expects read-lexemes-on-line to return
   list in LR order.
(define (read-data-blk white2 lxm )
  (let ( (depth 0 ) )
    (define (done lxs new )
      (append (reverse lxs ) new ))
    (let get-lexemes
         ( (lexemes (list ))
           (new-lexeme lxm ) )
      (if (eq? (lxm-type new-lexeme ) 'EOF )
          (done lexemes (list new-lexeme ))
          (begin
            (case (lxm-type new-lexeme )
              ((Open ) (set! depth (+ depth 1 )))
              ((Close ) (set! depth (- depth 1 ))) )
            (if (<= depth 0 )
                (done (cons new-lexeme lexemes ) (read-lexemes-on-line ))
                (get-lexemes (cons new-lexeme lexemes ) (read-lexeme )) ))))))
#| This doesn't work here — ;;;\newpage — |#
#| Neither does this — #|\newpage|# — |#

   Procedure read-block scans through whitespace to find the first visible
   lexeme, which determines the block kind, it then reads and saves lexemes
   until it reaches the end of the block. It returns the block encoded as
#| '(,⟨kind⟩ ,⟨white square⟩ ,@⟨lexeme list⟩)   . |#
   – I think that should be   '(,⟨kind⟩ ,⟨white square⟩ ,⟨lexeme list⟩)   .
   There may later be other information added; e.g. lexdata encoding.
   See §8.2 on page 88 for the definition of a "block".

   Procedure read-block will become ts:read-block. It is called only by
#| ts:make-reply It reads lexemes from standard input until the end of file |#
   or until it reads a complete block. A block is either a datum (possibly
   with internal atmosphere), or a comment block or a Remark block.
(define (rm:read-block )
  (let* ( (white2 (read-white2 0 0 ))
          (lxm (read-lexeme )) )
    (cond ( (eq? (lxm-type lxm ) 'EOF )
            (list 'BkEof white2 '(EOF )))
          ( (is-token? lxm )
            (list 'BkData white2 (read-data-blk white2 lxm )))
          ( (eq? (lxm-type lxm ) 'SharpBar )
            (list 'BkComment0 white2 (read-nested-comment 1 )))
          ( (eq? (lxm-type lxm ) 'Semicolon )
            (list 'BkComment3 white2 (read-cmnt-blk white2 lxm )))
          ( (eq? (lxm-type lxm ) 'SemiSharp )
            (list 'BkRemark white2 (rr:read-remark-blk white2 lxm )))
          ( else
            (list 'Error "bad␣block" white2 lxm )) )))
(define (dbg:read-block )
  (let* ( (white2 (read-white2 0 0 ))
          (lxm (read-lexeme )) )
    (debug "read-block␣white2=" white2 ",␣lxm=" lxm )
```

```
      (cond ( (eq? (lxm-type lxm ) 'EOF )
                (list 'BkEof white2 '(EOF )))
            ( (is-token? lxm )
                (list 'BkData white2 (read-data-blk white2 lxm )))
            ( (eq? (lxm-type lxm ) 'SharpBar )
                (list 'BkComment0 white2 (read-nested-comment 1 )))
            ( (eq? (lxm-type lxm ) 'Semicolon )
                (list 'BkComment3 white2 (read-cmnt-blk white2 lxm )))
            ( (eq? (lxm-type lxm ) 'SemiSharp )
                (list 'BkRemark white2 (rr:read-remark-blk white2 lxm )))
            ( else
                (list 'Error "bad␣block" white2 lxm )) ))
  (if #t (set! ts:read-lexeme read-lexeme ))
  (if #t (set! ts:read-block rm:read-block )   )
  (if #t (set! xx:read-block dbg:read-block )   )
  # The following right parenthesis closes the let that began on page 7.  #
  )
```

## 2.2 Write a Scheme file

These global variables will be given final values by the following "let".

```
(define ts:write-scheme-lxs #f )
(define ts:write-scheme-blk #f )
```

The procedure *ts:write-scheme-lxs* writes data to a port when given a list of lexemes. If the lexeme list is the result of reading a Scheme program as lexemes, then the displayed data should be equivalent to the original Scheme. See §9.2 for discussion of the sense of "equivalent".

A lexeme is a pair consisting of a *lex-type* together with extra data that depends upon the type. See 8.2 for more information.

The code for a String is cheesy. We should have a layout for the string that puts back line breaks and intraline whitespace.

This version of *write-scheme-lxs* that uses *write-line*.

```
(let ( (line-buf "" )
       (inremark #f ) )
  (define (write-scheme-remarks rems )
    (for-each
     (lambda (rem )
       (case (lxm-type rem )
         ((Result ) (begin (:newline ) (:display "!" )(:write (cdr rem ))))
         ((Result-TeX ) (begin (:display "->␣" )(:write (cdr rem ))))
         ((Resultlxs ) (:display "=>!" ) (write-scheme-lxs (cadr rem )) )
         ((SemiSharp )   (:display "?" ) (write-scheme-lxs (cadr rem )) (:newline ))
         ((Remark )   (:display "" ) (write-scheme-lxs (cadr rem )) (:newline ))
         ((NewLine ) (:newline ))
         (else (:display "bad␣Remark[" ) (:write rem ) (:display "]" ) ) )
         )
     rems ))
  (define (write-scheme-blk blk )
    (let nlines ((n (- (car (cadr blk )) 1 ) ))
      (if (> n 0 )(begin (:newline ) (nlines (- n 1 )))))
    (set! line-buf "" )
    (if (pair? blk )
        (case (car blk )
          ((BkComment3 )
```

```
            (for-each (lambda (s )
                              (:display     "; ; ;" )
                              (:display s )
                              (:newline ))
                        (caddr blk )) )
          ((BkData )
           (set! inremark #f )
           (write-scheme-lxs (caddr blk ))
           (:newline ) )
          ((BkRemark )
           (set! inremark #t )
           (set! line-buf ";#" )
           (write-scheme-remarks (caddr blk ))
           (if #f (:newline ))
           (set! line-buf "" )
           (set! inremark #f ) )
          ((BkEof ) #t )
          (else (:display "bad Bloxk[" ) (:write blk ) (:display "]" ) )
    )))
(define char-escapes
   '((#\\ . "\\\\" ) (#\" . "\\\"" )
    (#\tab . "\\t" ) (#\newline . "\\n" ) (#\return . "\\r" )))
(define (display-escaped s )
   (let ((len (string-length s )))
      (let scan-from ( (k 0 ) )
         (if (< k len )
             (let ((c (string-ref s k )))
                (let ((p (assv c char-escapes )))
                   (:display (if p (cdr p ) c )))
                (scan-from (+ 1 k ))) ))))
(define (:display s )
   (set! line-buf
          (string-append
           line-buf
           (cond
            ((string? s ) s )
            ((char? s ) (string s ))
            (else (string-append "write->"
                                 (:writing s )
                                 "<-not :displayed" ) )))))
(define (:display-n n s )
   (if (> n 0 )
       (begin
          (:display s )
          (:display-n (- n 1 ) s ))))
(define (:writing s )
   (let* ( (ost (open-output-string ))
           (str (begin (write s ost ) (get-output-string ost ))) )
      (close-port ost )
      str    ))
(define (:write s )
   (let* ( (ost (open-output-string ))
           (str (begin (write s ost ) (get-output-string ost ))) )
      (:display str )
```

```
      (close-port ost )   ))
(define (:newline )
  (write-line line-buf )
  (set! line-buf (if inremark ";#"    "" ) ))
(define (write-white-to col )
  (:display "␣" )
  (if (< (string-length line-buf ) col )
      (write-white-to col )) )
(define (write-scheme-lexeme tok )
      (if (pair? tok )
          (case (car tok )
            ((Id Vb Kw Sy ) (:display (cdr tok )))
            ((Result ) (begin (:newline ) (:display ";#=>␣" )(:write (cdr tok ))))
            ((Result-TeX ) (begin (:display ";#->␣" )(:write (cdr tok ))))
            ((Resultlxs ) (:display "=>" ) (write-scheme-lxs (cadr tok )) )
            ((Number ) (:display (cdr tok )))
            ((NewLine ) (:newline ))
          #| ((EndLine) (write-line line-buf)(set! line-buf "")) |#
            ((EndLine ) (display "" ))
            ((WhtSpc ) (write-white-to (WhtSpc-col tok ) ))
            ((Abbrev )
             (:display
              (case (cdr tok )
                ((quote ) "'" )
                ((quasiquote ) "`" )
                ((unquote ) "," )
                ((unquote-splicing ) ",@" ) )))
            ((Semicolon )
             (:display-n (car (cdr tok )) ";" )
             (:display (cdr (cdr tok ))) )
            ((SemiSharp )
             (:display ";#" )
             (write-scheme-lxs (cdr tok )) )
            ((Remark )
             (write-scheme-lxs (cadr tok ))
             (write-line line-buf )
             (set! line-buf "" ))
            ((Char )              (:display "#\\" )(:display (cdr tok )))
            ((String ) (:display "\"" ) (display-escaped (cdr tok ))(:display "\"" ))
            ((Boolean ) (:display (if (cdr tok ) "#t" "#f" )))
            ((Open )     (:display "(" ))
            ((VOpen )    (:display "#(" ))
            ((Close )    (:display ")" ))
            ((Dot )      (:display "." ))
            ((CmntShort ) (:display "#|" ) (:display (cdr tok )) (:display "|#" ))
            ((CmntLong )
                         (:display "#|" )(:display (car (cdr tok )))
                         (for-each
                           (lambda (ln ) (:newline ) (:display ln ))
                           (cdr (cdr tok )))
                         (:display "|#" ))
            (else (:display "bad␣pair␣S[" ) (:write tok ) (:display "]" ) ) )
          (begin (:display "not␣atom" ) (:write tok )(:display "<-" ))) )
(define (xx:write-scheme-lxs lexeme-list )
```

```
    (for-each write-scheme-lexeme
                lexeme-list )
    (if (< 0 (string-length line-buf )) (:newline )))
  (define (write-scheme-lxs lexeme-list )
    (for-each write-scheme-lexeme
                lexeme-list ))
  (set! ts:write-scheme-blk write-scheme-blk )
  (set! ts:write-scheme-lxs xx:write-scheme-lxs )
  )   #|end of let|#
```

Included File `tstprocs-ri.tex`

## 2.3 File: `tstprocs.scm` — Procedures needed for testing

This section is not a part of the TEX←`Scm` program. The procedures in it are not called anywhere in the file `texscm.scm`, but its TEX form `tstprocs.tex` is `\input` so that it shows up in the final printed form.

Instead the procedures in this section make or display scratch files needed to test those in `texscm.scm`. The file should be processed with the "`noeval`" option.

Procedure *write-lines-to* writes some strings into a scratch file so that the read procedures which are to be tested will have something to read. It assumes that the scratch file is erased and re-written. That is not guaranteed. R7RS §6.13.1 `open-output-file` says: If a file with the given name already exists, the effect is unspecified.

Maybe it should check that the file does not already exist. That would require deleting it when done. Should that be done by this program, the Makefile, or...? The R7RS report has procedures `file-exists?` and `delete-file`. So does R6RS.

```
(define (write-lines-to fname lines )
  (call-with-output-file fname
    (lambda (op )
      (let write-lines ((lines lines ))
        (if (pair? lines )
            (begin
              (display (car lines ) op )
              (newline op )
              (write-lines (cdr lines )) ))))) )
(define (write-lines lines ) (write-lines-to "scratch.txt" lines )   )
(define (eof-lxm? lex ) (or (eof-object? lex ) (eq? (lxm-type lex ) 'EOF )))
```

Procedure *file-verbatim* generates LATEX commands to show the contents of a file in a fixed width font with visible spaces.

Old versions used `\begin` and `\end{verbatim*}`. This worked for most files, but crashed if the file contained `\end{verbatim*}` (verbatim can not be nested so it doesn't matter if `\begin{verbatim*}` occurs first).

```
(define (old-file-verbatim fname )
  (append
   '("\\begin{verbatim*}" )
       #|much as below|#
   '("\\end{verbatim*}" )    ))
```

The current version wraps each line separately in LATEX one line `\verb*\r` ... `\r`. Can anybody get a raw carriage return into the middle of a line? Not by accident, I bet!

Also wrapped minipage around it.

```
(define (file-verbatim fname )
  (define (verbline ln )
    (string-append "\\verb*\r" ln "\r\\\\" ))
  (append
```

```
    (list (string-append
            "\\begin{minipage}{5in}"
            "\\quad\\\\␣\\hrules\\quad\\hrules\\␣file␣verbatim:␣{\\tt␣"
            fname
            "}␣\\hrules\\quad\\hrules\\\\" ))
    (with-input-from-file fname
      (lambda ()
        (let loop ((ln (read-line ))
                   (lns '()) )
          (if (eof-object? ln )
              (reverse lns )
              (loop (read-line ) (cons (verbline ln ) lns )) ))))
    (list (string-append
            "\\hrules\\quad\\hrules\\␣end␣verbatim:␣{\\tt␣"
            fname
            "}\\hrules\\quad\\hrules\\\\"
            "\\end{minipage}\\quad\\\\"    ))     ))
```

Procedure *file-TeX* gets LaTeX commands from a file and arranges for them to be included in the document.

```
(define (file-TeX fname )
  (append
    (list (string-append
            "\\begin{minipage}{5in}"
            "\\quad\\\\␣\\hrules\\quad\\hrules\\␣file␣\\TeX:␣{\\tt␣"
            fname
            "}␣\\hrules\\quad\\hrules\\␣\\\\" ))
    (with-input-from-file fname
      (lambda ()
        (let loop ((ln (read-line ))
                   (lns '()) )
          (if (eof-object? ln )
              (reverse lns )
              (loop (read-line ) (cons ln lns )) ))))
    (list (string-append
            "␣\\hrules\\quad\\hrules\\␣␣end␣\\TeX:␣{\\tt␣"
            fname
            "}␣␣\\hrules\\quad\\hrules\\␣\\\\"
            "\\end{minipage}\\quad\\\\"    ))     ))
```

Procedure *read-lxms-from-lines* can be given several arguments, all of which are strings. These are written to a file and then read as lexemes which are returned as lexeme list.

```
(define (read-lxms-from-lines . lines )
  (begin
    (write-lines-to "scratch.txt" lines )
    (with-input-from-file    "scratch.txt"
      (lambda ()
        (let loop ((lx (ts:read-lexeme ))
                   (ls '()) )
          (if (eof-lxm? lx )
              (reverse ls )
              (loop (ts:read-lexeme ) (cons lx ls ))     ))))))
(define (read-lxms-from fname )
  (begin
    (with-input-from-file fname
      (lambda ()
```

```
    (let loop ((lx (ts:read-lexeme ))
               (ls '()) )
      (if (eof-lxm? lx )
          (reverse ls )
          (loop (ts:read-lexeme ) (cons lx ls ))     ))))))
(define (read-lxms ) (read-lxms-from "scratch.txt" ))
```

### 2.3.1 Procedures to Read Blocks of Scheme

This was in `tstblocks.scm` moved here because it is needed also in `tstsort.scm`.

```
(define (read-blocks-from fname )
  (begin
    (with-input-from-file fname
      (lambda ()
        (let loop ((lx (ts:read-block ))     #|note font|#
                   (ls '()) )
          (if (or (eof-lxm? lx )(eq? (lxm-type lx ) 'BkEof ))
              (reverse (cons lx ls ))
              (loop (ts:read-block ) (cons lx ls ))     ))))))
(define (xx:read-blocks-from fname )
  (begin
    (with-input-from-file fname
      (lambda ()
        (let loop ((lx (xx:read-block ))     #|note font|#
                   (ls '()) )
          (if (or (eof-lxm? lx )(eq? (lxm-type lx ) 'BkEof ))
              (reverse (cons lx ls ))
              (loop (xx:read-block ) (cons lx ls ))     ))))))
```

Procedure *read-blocks-from-lines* first writes some lines to a scratch file then reads them back as blocks and returns a list of blocks.

```
(define (read-blocks-from-lines . lines )
  (write-lines-to "scratch.txt"    lines )
  (read-blocks-from "scratch.txt" ) )
(define (xx:write-scheme-blocks-to fname blocks )
  (with-output-to-file fname
    (lambda () (for-each xx:write-scheme-blk blocks )) ))
(define (xx:write-TeX-blocks-to fname blocks )
  (with-output-to-file fname
    (lambda () (for-each ts:write-TeX-blk blocks )) ))
(define (write-scheme-blocks-to fname blocks )
  (with-output-to-file fname
    (lambda () (for-each ts:write-scheme-blk blocks )) ))
(define (write-TeX-blocks-to fname blocks )
  (with-output-to-file fname
    (lambda () (for-each ts:write-TeX-blk blocks )) ))
```

2025-11-03: It looks like the file must end with a data block or the \hrule goes crooked.

```
'()
```

End of file `tstprocs-ri.tex`

Included File `tstread-riA.tex`

## 2.4 File: `tstread.scm` — Test Scheme Lexeme Transput

This section is not a part of the TEX←`Scm` program. The procedures in it are not called anywhere in the file `texscm.scm`. Instead the procedures in this section call those in `texscm.scm` to test them. In an emacs buffer running Scheme after loading the TEX←`Scm` program (by `(load "srckaw/texscm/texscm.scm")`), the procedures in it can be called "by hand". That is, I can enter calls to procedures defined there and see the answer written in the buffer.

If this results in anything worth remembering, copy it to the end of this file (`tstread.scm`) below. The `Makefile` will ensure that it is processed so that it can be included the the TEXed document `texscm.ps`. — Or is it `texscm-r.ps`?

> We want these tests to actually work (i.e. be executed, maybe pass). Therefore the file `tstread-x.tex` is actually included, because it is produced by the old *ts:layout* procedure with evaluation.

— 2025-11-05: that is no longer true. The new plan is `ts:make-reply` and we now use it.
(*list* ts:version ′: ts:subversion )
=>
?⟹
("`$3-1/10$`" : "" )
(*load* "`tstprocs.scm`" )
(*load* "`texscm.scm`" )
(*list* ts:version ′: ts:subversion )
=>
?⟹
("`$3-1/11$`" : "" )

In `*-ri.tex` these show `A` and `F`, respectively. In `*-riF.tex` they are both `F`. This shows that the new version is the only one in use. texscm3-r includes `*-ri.tex` How to use the old while testing the new? This shows that the new version overwrites the old. We no longer rely so much upon the behaviour explained in § 8.3 but neither do we prevent it. —Does not testing the new using the old rely upon it?

### 2.4.1 Test Read Lexemes

> This should be changed to allow better formating, but for now, in order to see the whole result, just convert it all to a character string in a fixed width font and break it into 80 character fragments with double bars ( ‖ ) to indicate the breaks. See procedure *write-cut* on page **??**.

—2025-11-03: The version 3 plan is that the result of evaluating an expression by TEX←`Scm` is returned as a list of lexemes so that editorial remarks can be applied to it, which is, after all, the whole point. — Why not just return ascii? — Properly scoped sorting of identifiers would require scanning and parsing anyway, why not do it early? Avoid scanning and parsing what is already a lexlist. Editorial remarks like `;#=>` and `;#->` will determine that.
(read-lxms-from-lines "`(*␣2␣pi)`" )
cut
‖ ⟹`((Open) (Id . "*") (WhtSpc 1 3) (Number . "2") (WhtSpc 1 5) (Id . "pi") (Clo`‖
‖ `se) (NewLine 2))`

=> 9
?⟹
((Open )(Id . "*" )(WhtSpc 1 3 )(Number . "2" )(WhtSpc 1 5 )(Id
. "pi" )(Close )(NewLine 2 ))

- Test multiple lines

(**define** lexs
(read-lxms-from-lines
";;; This is a test."
"(define pi 3)"
";#:= author \"Keith Wright\"" ))
lexs
=>
?⟹
((Semicolon . (3 . " This is a test." ))(NewLine 4 )(Open )(Id . "define" )(WhtSpc 1 8 )(Id . "pi" )(WhtSpc 1 11 )(Number . "3" )(Close )(NewLine 5 )(SemiSharp ( Id . ":=" )(WhtSpc 1 5 )(Id . "author" )(WhtSpc 1 12 )(String . "Keith Wright" )(EndLine 0 ))(NewLine 6 ))

- vertical line identifiers

(read-lxms-from-lines "|var| = |8?|;or not" )
=>
?⟹
((Id . "var" )(WhtSpc 1 6 )(Id . "=" )(WhtSpc 1 8 )(Id . "8?" )
(Semicolon . (1 . "or not" ))(NewLine 8 ))

- sharp-bar comments

(read-lxms-from-lines "#|boo|# xx" )
=>
?⟹
((CmntShort . "boo" )(WhtSpc 1 8 )(Id . "xx" )(NewLine 10 ))
(read-lxms-from-lines "#|boo" "hoo|# yy" )
=>
?⟹
((CmntLong "boo" "hoo" )(WhtSpc 1 6 )(Id . "yy" )(NewLine 13 ))
(read-lxms-from-lines "#|boo" "|# zz" )
=>
?⟹
((CmntLong "boo" "" )(WhtSpc 1 3 )(Id . "zz" )(NewLine 16 ))
(read-lxms-from-lines "#(1 2 3)" )
=>
?⟹
((VOpen )(Number . "1" )(WhtSpc 1 4 )(Number . "2" )(WhtSpc 1 6 )(Number . "3" )(Close )(NewLine 18 ))

- Peculiar identifiers

(read-lxms-from-lines ". . . . . .. => -> ++ --> ==> --" )
=>
?⟹
((Id . "..." )(WhtSpc 1 4 )(Dot . "." )(WhtSpc 1 6 )(Error . ".." )
(WhtSpc 1 9 )(Id . "=>" )(WhtSpc 1 12 )(Id . "->" )(WhtSpc 1 15 )(
Error . "++" )(WhtSpc 1 18 )(Error . "-->" )(WhtSpc 1 22 )(Id . "==>" )(WhtSpc 1 26 )(Error . "--" )(NewLine 20 ))

2025-11-02: Both .. and ++ should be identifiers, but get `Error`s.

- input tabs

(read-lxms-from-lines "xx\t zz \tw " )
=>

?⟹
((Id . “xx” )(WhtSpc 7 9 )(Id . “zz” )(WhtSpc 5 16 )(Id . “w” )
(NewLine 22 ))

- Remarks

(read-lxms-from-lines “;#set!␣title␣\"Good␣Stuff\"” )
=>
?⟹
((SemiSharp (Id . “set!” )(WhtSpc 1 7 )(Id . “title” )(WhtSpc 1 13 )(String .
“Good␣Stuff” )(EndLine 0 ))(NewLine 24 ))
(read-lxms-from-lines “;#:=␣title␣\"Good␣Stuff\"” )
=>
?⟹
((SemiSharp (Id . “:=” )(WhtSpc 1 5 )(Id . “title” )(WhtSpc 1 11 )(String .
“Good␣Stuff” )(EndLine 0 ))(NewLine 26 ))
(read-lxms-from-lines “;#:=␣author␣\"Keith␣Wright\"” )
=>
?⟹
((SemiSharp (Id . “:=” )(WhtSpc 1 5 )(Id . “author” )(WhtSpc 1 12 )(String .
“Keith␣Wright” )(EndLine 0 ))(NewLine 28 ))
(read-lxms-from-lines
“;#=>␣(define␣id”
“;#␣(lambda␣(x)”
“;#␣␣x))” )
=>
?⟹
((SemiSharp (Id . “=>” )(WhtSpc 1 5 )(Open )(Id . “define” )(WhtSpc 1 13 )
(Id . “id” )(EndLine 0 ))(NewLine 30 )(SemiSharp (WhtSpc 1 3 )(Open )(Id
. “lambda” )(WhtSpc 1 11 )(Open )(Id . “x” )(Close )(EndLine 0 ))(NewLine
31 )(SemiSharp (WhtSpc 2 4 )(Id . “x” )(Close )(Close )(EndLine 0 ))(
NewLine 32 ))

2025-11-07: There have been problems with dotted lists.

'(3 . “:lexlist<-datum” )
=>
?⟹
(3 . “:lexlist<-datum” )

2025-11-10: ... and with vectors `#(5 55 22)`

### 2.4.2 Test Write Lexemes

In an emacs buffer running Scheme after calling `(load "srckaw/texscm/texscm.scm")`, I can call `ts:write-scheme-lxs` as shown below and see the answer written in the buffer. The `NewLine` is needed to flush the buffer. — 2025-04-06(Sun)

```
scheme@(guile-user)>
 (ts:write-scheme-lxs
      (list '(Open) '(Number . "3.14") '(Close) '(NewLine 0)  )  )
(3.14)
```

It will be convenient to automate some of that.

- read some lexemes from a file

(**define** lexs
(read-lxms-from-lines
“;;;␣This␣is␣a␣test.”
“;;;␣of␣this.”
“(define␣pi␣#|not␣3|#”
“3.14159)” ))
See what they look like
lexs
=> xx
?⟹
((Semicolon . (3 . “␣This␣is␣a␣test.” ))(NewLine 34 )(Semicolon . (3 . “␣of␣this.” ))(NewLine 35
)(Open )(Id . “define” )(WhtSpc 1 8 )(Id . “pi” )(WhtSpc 1 11 )(CmntShort
. “not␣3” )(NewLine 36 )(Number . “3.14159” )(Close )(NewLine 37 ))
Now write those lexemes as Scheme
(*with-output-to-file* “scratch-r.scm”
(**lambda** () (ts:write-scheme-lxs lexs )))
(file-verbatim “scratch-r.scm” )
→

___ ___ file verbatim: scratch-r.scm ___ ___
;;;␣This␣is␣a␣test.
;;;␣of␣this.
⟶ (define␣pi␣#|not␣3|#
3.14159)
___ ___ end verbatim: scratch-r.scm___ ___

The result looks the same as the original input

◦ Do that again, this time with a semi-sharp remark
(**define** lexs
(read-lxms-from-lines
“;#:=␣author␣\"Keith␣Wright\"”
“;;;␣This␣is␣a␣{\\em␣test}␣---␣$1+e^{i\\pi}=0$.”
“(define␣pi␣3)”
))
lexs
=>
?⟹
((SemiSharp (Id . “:=” )(WhtSpc 1 5 )(Id . “author” )(WhtSpc 1 12 )(String .
“Keith␣Wright” )(EndLine 0 ))(NewLine 39 )(Semicolon . (3 . “␣This␣is␣a␣{\\em␣test}␣---␣$1+e^{i\\pi}=0$.
)(Id . “define” )(WhtSpc 1 8 )(Id . “pi” )(WhtSpc 1 11 )(Number . “3” )
(Close )(NewLine 41 ))
(*with-output-to-file* “scratch-r.scm”
(**lambda** () (ts:write-scheme-lxs lexs )))
(file-verbatim “scratch-r.scm” )
→

___ ___ file verbatim: scratch-r.scm ___ ___
;#:=␣author␣"Keith␣Wright"
⟶ ;;;␣This␣is␣a␣{\em␣test}␣---␣$1+e^{i\pi}=0$.
(define␣pi␣3)
___ ___ end verbatim: scratch-r.scm___ ___

2025-11-02: Extra blank line between SemiSharp and comment. Tried to fix it by use of EndLine in *read-lexemes-on-line*. This caused missing lines later. — Who cares? Get blocks working. —

Blocks don't have much to do with writing Scheme.

Also write to TEX
(*with-output-to-file* "`scratch-r.tex`"
(**lambda** () (write-TeX-lxms lexs )))
(file-verbatim "`scratch-r.tex`" )
→

```
___ ___ file verbatim: scratch-r.tex ___ ___
\begin{scheme}%
{:=}␣␣{author}␣␣``\texttt{Keith\symbol{32}Wright}''
\end{scheme}

⟶ ␣This␣is␣a␣{\em␣test}␣---␣$1+e^{i\pi}=0$.
\begin{scheme}%
({define}␣␣{pi}␣␣3␣)\\

\end{scheme}
___ ___ end verbatim: scratch-r.tex___ ___
```

(file-TeX "`scratch-r.tex`" )
→

___ ___ file TEX: `scratch-r.tex` ___ ___

:= author "`Keith␣Wright`"
⟶ This is a *test* — $1 + e^{i\pi} = 0$.
(define pi 3 )

___ ___ end TEX: `scratch-r.tex` ___ ___

— BUG! The datum disappears if it is the last thing in the file. The last two lines of this file are definitions of *stop* and *go*. If you see only one, you see the bug, The definition of *go* will not appear in `tstread.tex` (or in `tstread.ps`) unless it is followed by (at least) a blank line. Actually, several things go sideways if there is no blank line at the end. The last line is missing from the *-s.scm, *-x.tex, and *-y.tex files. It *is* in the *-r.scm and *-ri.tex reply files.
(**define** stop "`.`" )
(**define** go "`;`" )

End of file `tstread-riA.tex`

# 3 Write a TEX file

## 3.1 TEX←Scm-modes

The *TeXscm-mode* keeps track of whether we are inside a pair of **begin**{scheme} — **end**{scheme} LATEX commands. in which case tabbing is in effect.

The modes are

outTeXMode — This is the starting mode, used for non-indented comments. It's just the body of an ordinary LATEX document.

SchemeMode — This for Scheme between \begin{scheme} and \end{scheme}. LATEX tabs are set and used and each line must end with \\

inTeXMode — This is for indented comments inside a block of Scheme, and is between \begin{minipage} and \end{minipage}, or inside a \mbox, or \verb.

```
(define TeX-linestarted #f )
(define TeXscm-mode 'outTeXMode )
(define (change-TeXscm-mode  newmode )
  (cond
   ((and (eq? TeXscm-mode 'outTeXMode ) (eq? newmode 'SchemeMode ))
    (begin (display "\\begin{scheme}%" ) (newline ) ))
   ((and (eq? TeXscm-mode 'SchemeMode ) (eq? newmode 'outTeXMode ))
    (begin (newline ) (display "\\end{scheme}" ) (newline ))))
  (set! TeXscm-mode newmode ) )
(define (TeX-newline mode )
  (begin
    (if (and (eq? mode 'SchemeMode ) TeX-linestarted ) (display "\\\\" ))
    (newline ) ))
```

## 3.2 Make Title Page

```
(define (write-TeX-document-pre )
  (let ( (subtitle (string-append
                    "file:␣\\ty{" ts:input-file-name
                    "}␣--␣Dialog␣by␣\\TeXScm" ts:version ":" ts:subversion )) )
    (display "\\input{tspreamble}" ) (newline )
    (display "\\begin{document}" ) (newline )
    (display "%" ) (newline )
    (display "\\title{" )
      (display (ts:parm-val 'program-title )) (display "%" ) (newline )
    (display "\\thanks{" ) (display (ts:parm-val 'title-foot ))
                                        (display "}" ) (newline )
    (display "\\\\\\Large␣" )  (display subtitle )  (display "}" ) (newline )
    (display "\\author{" )
      (display (ts:parm-val 'author ))
      (display "}" ) (newline )
    (display "\\date{\\today\\␣at␣\\timeofday\\␣\\small␣EST}" )(newline )
    (display "\\maketitle" ) (newline )
    (display (ts:parm-val 'copyright )) (newline )
    (display "\\tableofcontents" ) (newline )
    (display "%" ) (newline ) ))
(define (write-TeX-document-post )
    (display "\\end{document}" )
    (newline ) )
```

## 3.3 Write TEX Lexemes

Procedure *write-TeX-lxms* has one argument, which is a list of lexemes. Used by ts:write-TeX-blk to write a data block.

```
(define (write-TeX-lxms lexeme-list )
  (let ((bad #f )
        (needdelim #f )
        (linend #f ) )
    (for-each
     (lambda (tok )
       (set! bad #f )
       (set! linend #f )
       (case (lxm-type tok )
```

```
        ((Semicolon SemiSharp Remark Result Resultlxs Result-TeX )
         (change-TeXscm-mode 'outTeXMode ))
        ((NewLine ) (set! linend #t ))
        ((WhtSpc EndLine ) #f )
        ((Id Kw Vb Sy Number Char String Boolean Dot )
         (change-TeXscm-mode 'SchemeMode )(set! needdelim #t ))
        ((CmntLong CmntShort Abbrev Open VOpen Close )
         (change-TeXscm-mode 'SchemeMode )(set! needdelim #f ))
        (else (set! bad #t )) )
      (if bad
          (begin (display "bad␣TeX␣tok[" )(write tok )(display "]" ))
          (begin (display-TeX-lxm TeXscm-mode tok )
                 (if #f
                     (begin (display "\\quad\\\\" )(newline )))
                 (if needdelim (display "␣" ))
                 (set! needdelim #f )))
      (set! TeX-linestarted (not (eq? (lxm-type tok ) 'NewLine ))) )
    lexeme-list )
  (change-TeXscm-mode 'outTeXMode )  ))  #| TeX chokes if removed |#
```

change linend to items; probably better to get NewLine into remarks This is used to write SemiSharp and Results Should insert NewLines and call above procedure. — Better change above to insert NewLine if there isn't one soon enough. What about LxVector?

```
(define (:write-TeX-lxms lexeme-list )
  (let ((bad #f )
        (needdelim #f )
        (items 0 ) )
    (if (not (list? lexeme-list ))
        (begin (display "%Tlxms[" )(write lexeme-list )(display "]" )(newline ))
    (for-each
     (lambda (tok )
       (set! bad #f )
       (set! items (+ 1 items ))
       (case (lxm-type tok )
         ((Semicolon SemiSharp Remark Result Resultlxs Result-TeX )
          (change-TeXscm-mode 'outTeXMode ))
         ((NewLine ) #f )
         ((WhtSpc EndLine ) #f )
         ((Id Kw Vb Sy Number Char String Boolean Dot LxVector )
          (change-TeXscm-mode 'SchemeMode )(set! needdelim #t ))
         ((CmntLong CmntShort Abbrev Open VOpen Close )
          (change-TeXscm-mode 'SchemeMode )(set! needdelim #f ))
         (else (set! bad #t )) )
       (if bad
           (begin (display "bad␣TeX␣:tok[" )(write tok )(display "]" ))
           (begin (display-TeX-lxm TeXscm-mode tok )
                  (if (> items 25 )
                      (begin (display "\\quad\\\\" )(newline )
                             (set! items 0 )))
                  (if needdelim (display "␣" ))
                  (set! needdelim #f )))
       (set! TeX-linestarted (not (eq? (lxm-type tok ) 'NewLine ))) )
     lexeme-list ))
   (change-TeXscm-mode 'outTeXMode )  ))  #| TeX chokes if removed |#
```

First we define some handy strings that cause TEX to write some special characters. Double

backslashes are: one to "escape" the next character in a Scheme string so that it goes through to TEX and one for TEX itself. There are extra braces around each string so that the dollar sign at the end of one does not touch the dollar sign at the beginning of the next to form a double-dollar, which is a different TEX command.

```
(define TeX-sharp "{$}^{\\#}$}" )
(define TeX-sharp-backslash "{$}^{\\#}\\backslash$}" )
(define TeX-tt-double-backslash "{\\char`\\\\\\char`\\\\}" )
(define TeX-tt-backslash "{\\char`\\\\}" )
(define TeX-quote "{$\\thickspace}^\\prime$}" )
(define TeX-quasiquote "{$\\thickspace}^\\backprime$}" )
(define TeX-comma "{\\tt␣,}" )
(define TeX-comma-at "{\\tt␣,@}" )
(define TeX-big-sharp-bar "{$\\#}$\\rule[-1em]{0.3pt}{2em}}" )
(define TeX-big-bar-sharp "{\\rule[-1em]{0.3pt}{2em}$\\#}$}" )
(define TeX-sharp-bar "{$}^{\\#}|$}" )
(define TeX-bar-sharp "{$|{}^{\\#}$}" )
```

This a quick hack to make sure that the answer does not run off the right side of the page. It writes its argument and breaks it into 76 character pieces using one line verbatim with delimiting character that can not occur. Written 2024-05-14(Tue).

Note that `write-cut` write directtly into the TEX reply file.

```
(define (write-cut x )
  (let* ( (ost (open-output-string ))
          (str (begin (write x ost ) (get-output-string ost ))) )
    (do ( (k 0 (+ k 76 )) )
        ( (>= (+ k 76 ) (string-length str ) )
          (display "\\verb\r" )
          (display (substring str k (string-length str )))
          (display "\r\\\\" ) (newline ))
      (display "\\verb\r" )
      (display (substring str k (+ k 76 )))
      (display "\r$\\|$\\\\$\\|$" ) (newline ) )
    (close-port ost )  ))
```

This does the same, but writes to a Ascii file

```
(define (write-cut-pt x pt )
  (let* ( (ost (open-output-string ))
          (str (begin (write x ost ) (get-output-string ost ))) )
    (do ( (k 0 (+ k 60 )) )
        ( (>= (+ k 60 ) (string-length str ) )
          (display (substring str k (string-length str )) pt ) )
      (display (substring str k (+ k 60 )) pt )
      (newline pt )          )
    (close-port ost )  ))
(define cmntn-width 110 )
```

Display the TEX commands needed to typeset a single lexeme.

The code to display Result should be re-written to display the result with atmosphere using *write-TeX-lxms*. To avoid breaking what works make a new Resultlxs to do that.

```
(define (display-TeX-lxm mode lxm )
  (case (lxm-type lxm )
    ((Id Vb Kw Sy )      (display-TeX-id lxm ) )
    ((Number )     (display (cdr lxm )))
    ((Result )
     (begin (display "$\\noindent␣\\|\\Longrightarrow$" )
```

```
            (write-cut (cdr lxm ))
            (newline ) ))
    ((Resultlxs )
     (begin (display "\\noindent␣{\\large␣?}$\\Longrightarrow$" )
            (:write-TeX-lxms (car (cdr lxm )))
            (newline ) ))
    ((Result-TeX ) (begin
                     (display "$\\longrightarrow$" )
                     (if (string? (cdr lxm ))
                         (display (cdr lxm ))
                         (begin (newline )
                                (for-each
                                 (lambda (line ) (display line )(newline ))
                                 (cdr lxm )))) ))
    ((NewLine )   (TeX-newline mode ) )
    ((EndLine )   #f )
    ((WhtSpc )    (TeX-space mode lxm ))
    ((Abbrev )
     (display
      (case (cdr lxm )
        ((unquote ) TeX-comma )
        ((quote ) TeX-quote )
        ((quasiquote ) TeX-quasiquote )
        ((unquote-splicing ) TeX-comma-at ))))
    ((Semicolon )
     (display-TeX-comment (car (cdr lxm )) (cdr (cdr lxm ))))
    ((SemiSharp ) (:write-TeX-lxms (cdr lxm )) )
    ((Remark    ) (:write-TeX-lxms (cadr lxm )) )
    ((Char )
     (display
      (string-append TeX-sharp-backslash "{\\tt␣" (TeX-char (cdr lxm )) "}" )) )
    ((String ) (TeX-string (cdr lxm )))
    ((Boolean ) (display
                 (string-append TeX-sharp (if (cdr lxm ) "\\kw{t}" "\\kw{f}" ))))
    ((Open ) (display "(" ))
    ((VOpen ) (display "\\#(" ))
    ((Close ) (display ")" ))
    ((Dot ) (display "." ))
    ((LxVector )   (write-cut (cdr lxm )) )
    ((CmntShort ) (display TeX-sharp-bar )
     (display (cdr lxm ))
     (display TeX-bar-sharp ))
    ((CmntLong )
     (display (string-append TeX-big-sharp-bar "\\␣\\begin{cmntn}{" ))
     (display cmntn-width )
     (display "mm}" )
     (for-each (lambda (ln ) (display ln )(newline )) (cdr lxm ))
     (display (string-append "\\end{cmntn}␣" TeX-big-bar-sharp )))
    (else (display "%bad␣TeX␣pair␣[" ) (write lxm ) (display "]" ) )))
```

Typeset an identifier. Identifiers can be sorted as variables, keywords, or symbols. When they are first read they are simply identifiers, in which case they are printed according to the keyword list above. There may be a second pass over the data to do a more sophisticated sorting in which case we print according to the new sort.

```
(define (display-TeX-id tok )
```

```
(let ((str (cdr tok )))
  (let ((len (string-length str )))
    (display
     (case (lxm-type tok )
       ((Id ) "{" )   ((Vb ) "\\vb{" )   ((Kw ) "\\kw{" )   ((Sy ) "\\sy{" )) )
    (let display-kth ( (k 0 ) )
      (if (< k len )
          (let ((ch (string-ref str k ))
                (chn (if (< (+ 1 k ) len )
                         (string-ref str (+ 1 k ))
                         #\newline )) )
            (cond
             ((and (char=? ch #\< )(char=? chn #\- ))
              (begin
                (display "$\\leftarrow$" )
                (display-kth (+ k 2 ))))
             ((and (char=? ch #\- )(char=? chn #\> ))
              (begin
                (display "$\\rightarrow$" )
                (display-kth (+ k 2 ))))
             (else
              (display (TeX-char ch ))
              (display-kth (+ k 1 )) )))))
    (display "}" ) )))
```

I wanted to put a single blank instead of that first `\hs{1ex}` but that loses at the start of a line. (if (eq? mode 'outTeXMode) did not work out.

```
(define (TeX-space mode lxm )
    (define (display-n n s )
      (if (> n 0 )
          (begin
            (display s )
            (display-n (- n 1 ) s ))))
    (let ((arg (cadr lxm )))
      (let ( (nsp (if (number? arg ) arg (cdr arg )))
             (ntb (if (number? arg ) 0 (car arg ))) )
        (if (and (= ntb 0 )(= nsp 1 ))
            (display "␣" )
            (if (> nsp 0 )
                (begin
                  (display-n ntb "\\tb" )
                  (for-each display '("\\hs{" ,(* 2.0 nsp ) "mm}" )))
                (begin
                  (display-n (- ntb 1 ) "\\tb" )
                  (display "\\>" ) )))))
    (if (member 'TabSet (cdr lxm ))
        (display "\\=" )))
```

Make TEX print a given string, surround it by double quotes and use typewriter font. Spaces are printed as visible symbols like this: "`This␣is␣a␣string`" The visible space is coded as 32 in this font. Backslashes are doubled because they are escape characters in Scheme — only one goes through to TEX.

```
(define (TeX-string str )
  (let ((len (string-length str )) )
    (display "``\\texttt{" )
```

```
(let display-kth
      ( (k 0 ) )
   (if (< k len )
         (let ((ch (string-ref str k )))
            (display
             (case ch
                ((#\space ) "\\symbol{32}" )
                ((#\\ ) TeX-tt-double-backslash )
                ((#\" )   (string-append TeX-tt-backslash "\"" ))
                ((#\tab ) (string-append TeX-tt-backslash "t" ))
                ((#\return ) (string-append TeX-tt-backslash "r" ))
                ((#\newline ) (string-append TeX-tt-backslash "n" ))
                (else (TeX-char ch )) ))
            (display-kth (+ k 1 )) )))
   (display "}'" ) ))
```

A block comment is just passed on to TEX.

```
(define (display-TeX-comment n txt ) (display txt ))
```

Given a character or character name, *TeX-char* returns the string needed to produce that character in TEX. In most cases that is just the same character, but there are a few characters that are special in TEX.

Most special cases are self explanatory. Certain characters need to be escaped or are only available in math mode.

There does not seem to be a curly brace in typewriter font, so I make sure those are set in roman, but put a bit of extra space around it to simulate fixed width. The space is given in units of "ex" (the width of the character "x") in attempt to make it work with any font. This is not perfect, but will work reaonably well. These are braces {{}}. This shows how it looks in the current typewriter font:

"{{{{{x{{{{{␣␣␣}}}}}x}}}}}"

"01234x56789...01234x56789"

```
(define (TeX-char ch )
   (if (char? ch )
         (case ch
            ((#\\ ) "{$\\backslash$}" )
            ((#\< ) "{$<$}" )
            ((#\> ) "{$>$}" )
            ((#\^ ) "{${}^\\wedge$}" )
            ((#\~ ) "{$\\thicksim$}" )
            ((#\{ ) "{\\rm\\footnotesize\\{\\hs{0.3ex}}" )
            ((#\} ) "{\\rm\\footnotesize\\hs{0.3ex}\\}}" )
            ((#\# #\$ #\_ #\& #\% )
             (string-append "\\" (string ch )))
            (else (string ch )) )
         ch ))
```

applies a function to each of a list of S-expressions and a list of layouts.

```
(define (zz:for-each-form f sforms list-form )
   (if (null? sforms )
         '()
         (if (pair? sforms )
               (begin
                  (f (car sforms )(car list-form ))
                  (for-each-form f (cdr sforms ) (solid-cdr list-form )) )
               (if (eq? (lxm-type (car list-form )) 'Dot )
                     (f sforms (car (solid-cdr (car list-form ))))
```

```
                (begin (display "list␣ends␣without␣dot␣layout" )
                       (display sforms )
                       (display list-form )
                       (newline )    )))))
```

*meaning* returns a keyword identifier if it is being used as a keyword, but if the identifier has been rebound then it returns something rather arbitrary. ??? This should be fixed to ensure that the identifier is being used as the *original* keyword.

Used only in *def-binds* and *sort-ids-in-form!*.

```
(define (meaning id env )
  (if (eq? (sort-of id env ) 'Kw ) id 'notkw ))
```

Procedure *def-binds* is deprecated *:collect-bindings*. For this process we do not need the layout.

```
(define (def-binds scheme-form env )
  (let find-defs ( (s scheme-form )
                   (ls '()) )
    (if (null? s )
        ls
        (let ((first-form (car s )))
          (if (pair? first-form )
              (find-defs (cdr s )
                         (case (meaning (car first-form ) env )
                           ((define ) (cons (cons (dbound-var first-form ) 'Vb ) ls ))
                           ((define-syntax )(cons (cons (dbound-var first-form ) 'Kw ) ls ))
                           ((begin )          (append! (def-binds (cdr first-form ) env ) ls ))
                           (else ls ) ))
              (find-defs (cdr s ) ls ))))))
(define (sort-as-vb ids ) (imap (lambda (id ) (cons id 'Vb )) ids ))
```

*imap* is like *map* except that if given an improper list, the final non-list cdr is treated as though it were part of the list. If given a non-list it is treated as a list of one.

This is wrong??? Thus (imap f '(a b c . z)) = (map f '(z c b a))

```
(define (imap f ls )
  (let im ( (rs '())
            (ls ls ) )
    (if (null? ls )
        (reverse rs )
        (if (pair? ls )
            (im (cons (f (car ls )) rs ) (cdr ls ))
            (cons (f ls ) rs ) ))))
```

(*add-bindings! bindings env*) adds the bindings to the first locale in the environment, and returns the environment.

```
(define (add-bindings! bindings env )
  (begin (set-car! env (append bindings (car env ))) env ))
(define (add-locale env )
  (cons '() env ) )
```

Assume the form is a definition, extract the newly defined and bound variable. Does not handle curried definitions

```
(define (dbound-var sform )
   (let ( (dfv (cadr sform )) )
     (if (pair? dfv ) (car dfv ) dfv ) ))
```

Assume the form is a definition, extract the newly bound variable which are the parameters of the definition. Does not handle curried definitions

```
(define (bound-vars sform )
   (let ( (dfv (cadr sform )) )
```

```
    (if (pair? dfv ) (cdr dfv ) '()) ))
```

This is a list of the syntactic keywords that are defined in the sixth or seventh Revised Report on Scheme (R⁶RS, R⁷RS).

The *:primal-environ* has just these "top level" identifiers defined.

```
(define :keywords
  '(define if let set! cond begin and or lambda parameterize
    case quote let* letrec letrec* let-values let*-values
    quasiquote unquote unquote-splicing let-syntax syntax-rules
    identifer-syntax else define-syntax do cond-expand ))
(define :variables
  '(abs acos assv
    car cdr cadr cddr caar cdar
    caddr cdddr caadr cdadr caaar cdaar cadar cddar
    call-with-input-file  call-with-output-file
    set-car! set-cdr!
    append! load equal? floor
    eval apply
    list null? reverse pair? eq? cons newline
    with-input-from-file with-output-to-file eof-object? write read
    read-char peek-char
    number? char? boolean? list? symbol? string?
    append string-append not string interaction-environment
    for-each display map char=? + - * < = > >= <=
    modulo
    vector make-vector vector-set! vector-ref zero?
    string-length string-ref assoc member
    string→symbol string→number
    symbol→string number→string
    integer→char char→integer
    inexact→exact exact→inexact ))
(define :primal-environ
  (list
   (append
    (map (lambda (id ) (cons id 'Vb )) :variables )
    (map (lambda (id ) (cons id 'Kw )) :keywords ))))
```

## 3.4 Use tabbing

The first re-layout procedure, *sort-ids-in-body!* (see section 5.1), sorts identifiers so they can be printed in the proper font.

The second re-layout procedure, *re-indent*, changes the indentation and spacing commands to make it all line up better. It has one argument, which is a lexeme list[2]. We don't need an environment, or the actual Scheme program, as long as the spacing and indentation depends only upon the line breaks and indentation of the source. [This assumption would have to be revised if, for example, we wanted to indent a list that begins with a keyword differently from from one that begins with a variable.] — No, just re-indent the input if that is wanted.

LaTeX has a tabbing environment[9, p80]. Scheme data will be set in such a LaTeX environment. We want to set a tab where printed data should line up, and later use a tab to align printing on a later line. A tab stop is set by the LaTeX command "\=", and used in a later line with the command "\>".

In addition to indentation there are sometimes significant spaces within a Scheme line.

Here is an example that illustrates a difficulty. If we have an input like

[2] list layout = template theme presentation shape mold

```
(someproc (if (boolean)
              (let ( (a aval)
                     (b bval) ) e)
               #t)
          (foo bar) )
```

we need to have set a tab at "`(if`" in order to get the "`(foo bar)`" lined up. On the other hand, if "`(foo bar)`" had not been an argument to "`someproc`", then that tab stop would not be needed

```
(someproc (if (boolean)
              (let ( (a aval)
                     (b bval) ) e)
               #t) )
(foo bar)
```

So the ")" at the end of one of the last two lines determines whether or not a tab stop must be set in the first line. — No. It's not the parentheses, it's the indentation of the input

```
(and (if (boolean)          ; we might want
         (let ( (a aval) )  ; these comments
           e)               ; to line up.
         #t)
     (foo bar) )
```

We have the option of falling back to the old method, i.e. if re-layout does nothing then everything still works as before.

We *could* just insert tabs everywhere they could possibly be wanted. This would not work with the visible tabs in the final layout. Instead we just make two passes when tabbing, the first to decide where tabs should be set, the second to insert the proper tab uses and spaces. We assume when a line occurs which passes a tab stop without using it, then that tab stop can be forgotten— no following line needs to line up on a tab opportunity that has been passed by and ignored. For that reason, there is no need to make the first pass through the entire file and then the second pass—they can be interleaved.

Nevertheless, it is currently done in two passes for simplicity.

Tabs are used for indentation and for internal alignment. A tab may be set or used when an unecessary space occurs in the source. A space in unnecessary if it follows white space or a left parenthesis. Spaces at the beginning of a line are unnecessary, but this is subsumed by the genersl rule if we count a newline as white space. Most other spaces are necessary, and so do not become tabs.

Let's say, for now, that a tab could be set at (just before) the first visible character after an unnecessary #\`space` character in the input. These are called tab opportunities.

Here is the plan for the first pass: For the start of each line: if the first visible character on the line uses a tab opportunity from any preceding line, then take it. For the rest of the line: if a tab opportunity on this line matches one from the immediately preceding line, then take it, delete any other internal tab opportunities from the preceding line.

The *tab-columns* are the character positions of tab opportunities. If a tab opportunity is used in some later line it is converted to a tab stop.

We only line things up under a tab stop. Sometimes a line is to be indented a bit more than a tab stop. This is done with spaces of some convenient width which have nothing to do with the spacing of preceding lines. Naturally, all tabs come before any spaces, so it suffices to keep a count of each. The extra spaces may constitute a tab opportimity for succeeding lines.

Unused tab opportunities become garbage for the collector.

Tab opportunities are represented as a-list pairs each consisting of a source column paired with a boolean indicating whether the the opportunity has been taken, and (a pointer to) the space lexeme after which a tab may be set.

To take a tab opportunity (i.e. set a tab) we add a TabSet symbol to the WhtSpc.

Indentation and internal tabs are handled differently. Internal tab opportunities occur when there are unnecessary spaces in the input. If they are not taken by the next line, then they can be forgotten immediately. Indentation, on the other hand, presents a tab opportunity which persists even if not taken immediately.

```
(define current-tab-ops    #f )
(define prev-tab-ops #f )
(define (re-indent list-layout )
  (settabs list-layout )
  (usetabs list-layout ))
(define (dont-usetabs list-layout ) #f )
```

A tab opportunity must be some kind of space, just add a TabSet to the end of the parameter list.

```
(define (take-tbo ins )
  (if (> (car ins ) 0 )
      (case (cadr ins )
        ( (WhtSpc )   (set-cdr! (cdddr ins ) (list 'TabSet )) ) )))
```

Put the column at which the space ends together with the space itself on the list of noted tab opportunities. *lxm* should be a (pointer to) a WhtSpc lexeme. This just adds to the list of tab opportunities. Any thoughtful processing takes place at the end of the line in *merge-tbos*.

Note that the list *current-tab-ops* is in decreasing order, that is, the leftmost tab opportunity is the last on the list.

```
(define (note-tab-op lxm c )
  (set! current-tab-ops
        (cons (cons c lxm )
              current-tab-ops )))
```

Process a newline lexeme, indent is the column to indent to. It may be that the newline should be a separate lexeme and the indentation should be an ordinary Space, but for now just keep it the way it is.

To finish the current line, go through tab opportunities on the current line, current-tab-ops. Take any tab ops that properly match the previous tab ops. Merge the current with the previous, to create a list of tab ops that will still be open in the next line.

Save any that precede the first visible character, but replace later ones with those of the current line, then advance by making the current line previous, the indentation becomes the first tab op of the line that is starting.

I am not even sure I understand how LaTeX tabs work in all cases, To get this working to generate test cases for later processing, write a version of merge that only merges indentation.

The procedure *merge-tbos* should be called when at the end of a line to compute the tab opportunities that are still open after those of the line just ended are merged into the list of previously open tab ops.

The internal procedure *tabbefore* gets the indentation from the previous list of tab ops, which is assumed to be the tail beyond the first tab op further left than the given column.

```
(define (merge-tbos cur pre )
  (cond ((null? cur ) pre )
        ((null? pre ) cur )
        (else
          (let ( (indnt (last cur )) )
            (define (tabbefore col tbos )
              (if (or (null? tbos ) (>= col (caar tbos )))
                  tbos
                  (tabbefore col (cdr tbos ))))
            (let* ( (col (if (and (pair? indnt ) (pair? (car indnt )))
                             (caar indnt )
```

```
                                0 ))
                        (prvtab (tabbefore col pre )) )
                (if (and (pair? prvtab ) (pair? (car prvtab )))
                    (begin
                        (if (= (caar prvtab ) col )
                            (begin (set-car! indnt (car prvtab ))
                                   (set-cdr! indnt (cdr prvtab )))
                            (set-cdr! indnt prvtab ) )
                        (take-tbo (car prvtab ))))
                cur )))))
```

Return one element list, which is the last of *ls*.

```
(define (last ls )
  (if (not (pair? ls ))
      "error: last of non-list"
      (if (null? (cdr ls ))
          ls
          (last (cdr ls )) )))
```

Make an empty list that can be modified. (That's non-sense. What was I thinking?)

```
(define No-tabops (list ))
```

Procedure *settabs* goes through the layout of a list of ⟨datum⟩s (presumably *unzip*-ed from a Scheme program as can be read from a *.scm file). For each line of "code", it makes a list of tab opportunities on that line.

```
(define (settabs list-shape )
  (set! current-tab-ops No-tabops )
  (set! prev-tab-ops No-tabops )
  (let settabs-to-NL
       ( (ls list-shape )
         (elem-count 0 ) )
    (if (null? ls )
        #f
        (let ((lxm (car ls )))
          (case (lxm-type lxm )
            ((NewLine )
             (set! prev-tab-ops (merge-tbos current-tab-ops prev-tab-ops ))
             (set! current-tab-ops No-tabops )
             (settabs-to-NL (cdr ls ) 0 ) )
            ((WhtSpc )
                (if (or (< elem-count 2 ) (> (n-WhtSpc lxm ) 1 ))
                    (note-tab-op lxm (WhtSpc-col lxm )))
                (settabs-to-NL (cdr ls ) elem-count ) )
            ((List Abbrev )
                (settabs-to-NL (cdr lxm ) 0 )
                (settabs-to-NL (cdr ls ) (+ 1 elem-count )) )
            ((Semicolon Result Resultlxs Result-TeX SemiSharp Remark )
             (set! current-tab-ops No-tabops )
             (set! prev-tab-ops No-tabops )
             (settabs-to-NL (cdr ls ) 0 ))
            (else
                (settabs-to-NL (cdr ls ) (+ 1 elem-count ))) ))))))
```

Go through the *list-shape* and change some WhtSpc space numbers to tab and space numbers. Specifically, put out as many tabs as possible without running past the intended column, then spaces to make up any remainder. The vector tab-set-cols stores the (input) columns at which a tab has

been set, while *n-tab-set* is the cardinality of them. *tabs-to-col* computes the number of tabs (as recorded in *tab-set-cols*) needed get close to the given column without going over.

```
(define maxtab 20 )
(define usetabs
  (let ( (tab-set-cols (make-vector maxtab 4242 ))
         (n-tab-set 0 )
         (tabuse-on-line 0 )
         (tabset-on-line 0 ) )
    (define (tabs-to-col col after )
      (if (and (< after n-tab-set )
               (<= (vector-ref tab-set-cols after ) col ) )
          (tabs-to-col col (+ 1 after ))
          after ))
    (vector-set! tab-set-cols    0 0 )
    (lambda (list-shape )
      (if (null? list-shape )
          'void
          (let ( (lxm (car list-shape ))
                 (rst (cdr list-shape )) )
            (case (lxm-type lxm )
              ((NewLine )
               (set! tabuse-on-line 0 )
               (set! tabset-on-line 0 ))
              ((WhtSpc )
               (let* ( (indent-column (WhtSpc-col lxm ))
                       (ntabs (tabs-to-col indent-column 0 ))
                       (nspc (- indent-column
                                (if (> ntabs 0 )
                                    (vector-ref tab-set-cols (- ntabs 1 ))
                                    0 ))) )
                 (if (and (> (n-WhtSpc lxm ) 1 )
                          (= 0 tabuse-on-line )
                          (= 0 tabset-on-line ) )
                     (let ( (moretabs    (- ntabs tabuse-on-line )) )
                       (set-car! (cdr lxm ) (cons moretabs nspc ) )
                       (set! tabuse-on-line ntabs )))
                 (if (member 'TabSet (cdr lxm ))
                     (begin
                       (vector-set! tab-set-cols
                                    n-tab-set indent-column )
                       (set! n-tab-set (+ 1 tabuse-on-line tabset-on-line ))
                       (set! tabset-on-line (+ 1 tabset-on-line )) )   )) )
              ((List Abbrev )
               (let ((save n-tab-set ))
                 (usetabs (cdr lxm ))
                 (set! n-tab-set save ))) )
              (usetabs rst ) )))))
```

The association list *charnames* maps names of characters to the characters themselves.

```
(define charnames
  '( ("nul" . #\nul )   ("newline" . #\newline )("return" . #\return )
     ("tab" . #\tab )   ("space" . #\space ) ))
```

The inverse association list, *namechars*, maps characters to their names.

```
(define namechars (map (lambda (p ) (cons (cdr p )(car p ))) charnames ))
```

Procedure *name-char* takes a character and returns its name.

```
(define (name-char ch )
  (let ((pair (assoc ch namechars )))
    (if pair
        (cdr pair )
        ch )))
```

# 4 Blocks

Version 3 reads and writes blocks of Scheme. This is done by procedures defined in the Big Let (§2.1, p.7). It also writes (but never reads) LaTeX documents.

The original purpose of breaking the source code into blocks was so that the *TeXscm-mode* (See §3.1, p.26) can be set before the need to set tabs. With the version 3 re-design, remarks are processed as blocks. In this version all blocks are "top level".

The Scheme Report R$^7$RS [17, §3.1] says: "Scheme is a language with block structure". The Report does not define the word "block", but uses it to refer to the fact that there are nested regions in which each binding of an identifier is "visible". The word "block" is not in the index, but "region" is. In any case, blocks here are not the blocks of the report. Rather, TeX←`Scm` blocks are contiguous lines of a Scheme program that are processed as a unit.

The procedure *ts:write-scheme-blk* writes a block of scheme. It should be a two-sided inverse of: the procedure *ts:read-block* which returns a block encoded as a list of the form

( ⟨block type⟩ ⟨white2⟩ ⟨other data⟩ ) .

The block types are 'BkData, 'BkRemark, 'BkEof, and 'BkComment0, 'BkComment3

The *ts:read-block* procedure first scans through white space until it finds a token. That first token determines the block type, as follows: a sharp-bar (#|) is a Comment0, a semicolon-sharp (;#) is a Remark, a semicolon followed by anything else is a comment (what type?). A left parenthesis is the start of Data that includes everything up to the matching right parenthesis and anything that follows on the same line. Any other lexeme is a block by itself. — What about quotes, quasi-quotes, syntax quotes, and sharp-semicolon (#;)? (which starts a R$^7$RS datum comment.) — All abbreviations and datum comments must be followed by a datum. Demand one, even if it starts with a comment or remark. — Is there any other lexeme that can not be the start of a ⟨datum⟩?

Make a unique value
(**define** none (*list* 'NONE ))

A datum block looks like this: (`BkData` ⟨white2⟩ ⟨lexeme list⟩)

During processing, a Data a list of the form

(`BkData` ⟨white2⟩ ⟨lexeme-list⟩ ⟨lexeme-data⟩) .

— Why do that?

A remark block looks like this:/ (`BkRemark` ⟨white2⟩ (⟨semisharp⟩ . . . ))
where ⟨semisnarp⟩ ::= 'SemiSharp ⟨lexeme-list⟩

Procedure *debug* takes a list of an even number of arguments that come in pairs, each of which is a label string followed by a ⟨datum⟩. If a debug-port has been opened, the lables and data are written to the port.

```
(define (debug . args )
  (if debug-port
      (if (null? args )
          (begin
            (display "--Dbg;" debug-port )
            (newline debug-port ) )
          (begin
            (display (car args ) debug-port )
            (write-cut-pt (cadr args ) debug-port )
            (apply debug (cddr args )) ))))
```

## 4.1 Read data from lexeme lists

The three following procedures are modeled on R$^7$RS §6.13, but instead of reading characters or bytes from a file they read lexemes from a lexeme list. Procedure *read-a-lexdatum* reads lexemes until it gets a lex-⟨datum⟩.

A lex-⟨datum⟩ is a ⟨datum⟩ as in R$^7$RS, but with lexeme occurrences in place of ⟨simple datum⟩s. A lexeme occurrence is a lexeme that is `eq?` to one on the list *lxms*.

Procedure *open-input-lexemes* returns a port, which is coded as a procedure which returns another lexeme from the list each time it is called.

```
(define (open-input-lexemes lxms )
  (let ((rst lxms ))
    (lambda ()
      (if (pair? rst )
          (let ((t rst ))
            (begin (set! rst (cdr rst ))
                   (car t ) ))
          '(EOF ) ))))
(define (lx-eof? lx ) (equal? lx '(EOF )))
(define      (read-a-lexdatum lxport )
    (let ((lx (lxport )))
      (case (lxm-type lx )
        ((WhtSpc NewLine EndLine CmntShort ) (read-a-lexdatum lxport ) )
        ((Id Kw Vb Sy String Number ) lx )
        ((Abbrev )(cons (cons 'Id (symbol→string (cdr lx )))
                        (list (read-a-lexdatum lxport ))))
        ((Boolean ) lx )
        ((Dot Close EOF ) lx )
        ((VOpen ) (cons 'LxVector (read-lexdata lxport )))
        ((Open ) (read-lexdata lxport ))
        (else (list 'Error "bad␣lex" lx ))
      )))
```

Read ⟨datum⟩s from the list until Close, return list of results

```
(define      (read-lexdata lxport )
    (let ((one (read-a-lexdatum lxport )))
      (case (lxm-type one )
        ((WhtSpc NewLine EndLine ) (read-lexdata lxport ) )
        ((Id Kw Vb Sy String Number )
          (cons one (read-lexdata lxport ))   )
        ((Dot ) (let ((last (read-a-lexdatum lxport )))
                  (let ((closing (read-a-lexdatum lxport )))
                    (if (eqv? (lxm-type closing ) 'Close )
                        last
                        '(Error "dot␣?" )))))
        ((Close ) (list ))
        ((Error ) (list 'Error "Erorr[" one "]" ))
        ((EOF ) '(Error . "missing␣close" ))
        (else (cons one (read-lexdata lxport )))   )))
```

## 4.2 Evaluate Blocks

```
(define :eval-env (interaction-environment ))
```

evaluate in order all datums in a block and return the result of the last one.

2025-10-30: this takes block data as lexlist but returns ans raw.

```
(define (eval-blk block )
  (let ((ans none ))
    (display "**eval-blk:" )(write block )(newline )
    (let evalnext ((data (datum←lexdatum (:lexdata←blk block ) )))
      (if (null? data )
```

```
            ans
            (begin
                 (display "**eval[->" )(write (car data ))
               (set! ans (eval (car data ) :eval-env ))
                 (display "<-]␣=␣" )(write ans )(newline )
               (evalnext (cdr data ))                  )))))
(define (displaynl s )(display s )(newline ))
(define (ts:write-TeX-blk blk )
  (case (car blk )
    ((BkComment3 ) (for-each displaynl (caddr blk )))
    ((BkData )     (write-TeX-lxms (caddr blk )) )
    ((BkRemark ) (write-TeX-lxms (caddr blk )) )
    ((BkEof ) (change-TeXscm-mode 'outTeXMode )(display "%EOF" ) (newline ))
    (else #f ) ))
```

To get a datum from an lex-datum, just copy it but put a ⟨simple datum⟩ [R$^7$RS §7.1.2], in place of each lexeme occurrence.

```
(define (datum←lexdatum out )
  (if (null? out ) '()
      (if (pair? out )
          (if (symbol? (car out ))
              (case (car out )
                ((Number ) (string→number (cdr out )))
                ((Id Kw Vb Sy ) (string→symbol (cdr out )))
                ((String ) (cdr out ))
                ((Boolean ) (cdr out ))
                ((LxVector )(cdr out ))
                ((Abbrev ) (begin (debug "Abbrev=" out )) "Abr?" )
                (else (list 'Error "bad␣lexeme" (car out ) )))
              (cons (datum←lexdatum (car out ))
                    (datum←lexdatum (cdr out ))   ))
          (list 'Error "lxd␣not␣pair" out ) )))
```

#### 4.2.1 `endcsname` BUG

There was a bug that got this obscure error message
`! Missing \endcsname inserted.` This error is caused when an attempt is made to set a tab outside a tabbing environment. Specifically when "`\=`" comes before "`\begin{scheme}`".

```
(define debug-port #f )
(define atestproc #f )
(define btestproc #f )
```

Included File `tstblocks-riA.tex`

## 4.3 File: `tstblock.scm` — Test Blocks

This file contains a continuation of testing so load the testing procedures

```
(list ts:version ts:subversion )
=>
?⟹
("$3-1/10$" "" )
(load "tstprocs.scm" )
(load "texscm.scm" )
(list ts:version ts:subversion )
```

=>
?⟹
(“$3-1/11$” “” )

This should be fixed so that the font of *ts:read-block* noted below comes out right. (See **Plans** §8.3.1 pg.94.) Make the *load*-ed file have information about the sort of identifiers. Put this in its own remarks. That should be done by an imperative remark that also updates the current environment with top level bindings in that file.

### 4.3.1 Read Blocks of Scheme

Define a list of blocks by writing lines of Scheme to the scratch file and reading them back as blocks:

```
(write-lines-to “scratch.scm”
  (list
        “;;; three blocks are written.”
        “;;; These are comment, remark, and datum.”
        “;#=> foo 0”
        “(define exp 2)” ))
```

Now the scratch file contains those lines:
(file-verbatim “scratch.scm” )
→

```
    ___  ___ file verbatim: scratch.scm ___  ___
    ;;; three blocks are written.
    ;;; These are comment, remark, and datum.
⟶  ;#=> foo 0
    (define exp 2)
    ___  ___ end verbatim: scratch.scm___  ___
```

. . . now read that file as blocks and show the result:
(**define** three-blocks (read-blocks-from “scratch.scm” ))
three-blocks
cut

```
‖ ⟹((BkComment3 (0 . 0) (" three blocks are written." " These are comment, rema‖
‖ rk, and datum.")) (BkRemark (0 . 0) ((Remark ((Id . "=>") (WhtSpc 1 5) (Id .‖
‖  "foo") (WhtSpc 1 9) (Number . "0") (EndLine 0))))) (BkData (0 . 0) ((Open)‖
‖ (Id . "define") (WhtSpc 1 8) (Id . "exp") (WhtSpc 1 12) (Number . "2") (Clos‖
‖ e) (EndLine 0))) (BkEof (1 . 0) (EOF)))
```

Write the blocks to scratch reply files, both Scheme or TEX.
(write-scheme-blocks-to “scratch-r.scm” three-blocks )
(write-TeX-blocks-to “scratch-r.tex” three-blocks )
. . . look at the Scheme file
(file-verbatim “scratch-r.scm” )
→

```
    ___  ___ file verbatim: scratch-r.scm ___  ___
    ;;; three blocks are written.
    ;;; These are comment, remark, and datum.
⟶  ;#=> foo 0
    (define exp 2)
    ___  ___ end verbatim: scratch-r.scm___  ___
```

Then the LATEX file and the result of running it through LATEX.
(file-verbatim “scratch-r.tex” )

→

```
___ ___ file verbatim: scratch-r.tex ___ ___
␣three␣blocks␣are␣written.
␣These␣are␣comment,␣remark,␣and␣datum.
\begin{scheme}%
{=${$>$}}␣␣{foo}␣␣0
\end{scheme}
\begin{scheme}%
({define}␣␣{exp}␣␣2␣)
\end{scheme}
%EOF
___ ___ end verbatim: scratch-r.tex___ ___
```

⟶

The result has not been typeset with sorting and tabbing. This just proves that blocks can be written in proper order.
(file-TeX "`scratch-r.tex`" )
→

___ ___ file TEX: `scratch-r.tex` ___ ___
three blocks are written. These are comment, remark, and datum.
=> foo 0
(define exp 2 )
___ ___ end TEX: `scratch-r.tex` ___ ___

⟶

- — Try extended remarks

```
(write-lines-to "scratch3.scm"
  (list "(define␣zero␣0)"
        ";#=>␣"
        ";#␣(␣"
        ";#␣\"zero\"␣"
        ";#␣)"
        ";#:=␣buz"
        ";#:=␣cut" ))
```

(**define** blocks (read-blocks-from "`scratch3.scm`" ))
. . . and this is the result of reading that as blocks:
blocks
cut

```
‖ ⟹((BkData (0 . 0) ((Open) (Id . "define") (WhtSpc 1 8) (Id . "zero") (WhtSpc‖
‖ 1 13) (Number . "0") (Close) (EndLine 0))) (BkRemark (1 . 0) ((Remark ((Id .‖
‖  "=>") (EndLine 0) (NewLine 0) (WhtSpc 1 3) (Open) (EndLine 0) (NewLine 0) (‖
‖ WhtSpc 1 3) (String . "zero") (EndLine 0) (NewLine 0) (WhtSpc 1 3) (Close) (‖
‖ EndLine 0))) (Remark ((Id . ":=") (WhtSpc 1 5) (Id . "buz") (EndLine 0))) (R‖
‖ emark ((Id . ":=") (WhtSpc 1 5) (Id . "cut") (EndLine 0))))) (BkEof (0 . 0)‖
‖ (EOF)))
```

Write those blocks as Scheme:
(write-scheme-blocks-to "`scratch-r.scm`" blocks )
(file-verbatim "`scratch-r.scm`" )
→

file verbatim: `scratch-r.scm`

```
(define␣zero␣0)
;#=>
;#␣(
;#␣"zero"
;#␣)
;#:=␣buz
;#:=␣cut
```

end verbatim: `scratch-r.scm`

— and Lo! We get the same thing back.
(write-TeX-blocks-to "`scratch-r.tex`" blocks )
Then the LaTeX file and the result of running it through LaTeX.
(file-verbatim "`scratch-r.tex`" )
→

file verbatim: `scratch-r.tex`

```
\begin{scheme}%
({define}␣␣{zero}␣␣0␣)
\end{scheme}
\begin{scheme}%
{={$>$}}␣\\
␣(\\
␣‘‘\texttt{zero}’’␣\\
␣)
\end{scheme}
\begin{scheme}%
{:=}␣␣{buz}
\end{scheme}
\begin{scheme}%
{:=}␣␣{cut}
\end{scheme}
%EOF
```

end verbatim: `scratch-r.tex`

The result has not been typeset with sorting and tabbing. This just proves that blocks can be written in proper order.
(file-TeX "`scratch-r.tex`" )
→

file TeX: `scratch-r.tex`

```
(define zero 0 )
=>
(
"zero"
)
:= buz
:= cut
```

end TeX: `scratch-r.tex`

. . . or at least it will. 2026-01-15: Even without typesetting, the remarks should be preceded by ;#, but they are not.

◦ — **Bug! MisRem** 2026-01-12: This is in `texscm.scm` but not in `texscm-rA.scm` — Update 2026-01-12: I tried to figure out why the complicated editorial remarks in TeX←`Scm` were not getting

processed by copying them here. It turns out complication was not the problem. *No* remarks were processed but TEX←Scm itself has no others.

⟨Unnecessary test deleted.⟩

◦ — The missing Remark bug runs but can not hide. The symptom is that the editorial remarks in texscm.scm don't work.

—2026-01-15: texscm-r.tex looks good, but texscm-rA.tex gets "bad TeX tok[Remark]"

```
% update on promotion program-title
bad TeX tok[Remark]\begin{scheme}%
{:=}  {program-title}  ``\texttt{{\char`\\\char`\\}TeXScm3\$-1/10\$}''
\end{scheme}

bad TeX tok[Remark]\begin{scheme}%
{:=}  {author}  ``\texttt{Keith\symbol{32}Wright}''
\end{scheme}
```

◦ — Try to reproduce that with a block with several remarks:

```
(write-lines-to "scratch.scm"
  (list
      ";;; This makes it look like \\TeXScm."
      ";#:= program-title \"Title\""
      ";;;"
      ";#:= author \"Awe Thor\""
      ";#:= copyright \"Copyright \\\\copyright\\\\ 2026 B.C.\""
      ";#:= title-foot \"thanks\"" ))
(define blocks (read-blocks-from "scratch.scm" ))
```

. . . and this is the result of reading that as blocks:

blocks

cut

```
|| ⟹((BkComment3 (0 . 0) (" This makes it look like \\TeXScm.")) (BkRemark (0 .||
|| 0) ((Remark ((Id . ":=") (WhtSpc 1 5) (Id . "program-title") (WhtSpc 1 19) (||
|| String . "Title") (EndLine 0))))) (BkComment3 (0 . 0) ("")) (BkRemark (0 . 0||
|| ) ((Remark ((Id . ":=") (WhtSpc 1 5) (Id . "author") (WhtSpc 1 12) (String .||
||  "Awe Thor") (EndLine 0))) (Remark ((Id . ":=") (WhtSpc 1 5) (Id . "copyrigh||
|| t") (WhtSpc 1 15) (String . "Copyright \\copyright\\ 2026 B.C.") (EndLine 0)||
|| )) (Remark ((Id . ":=") (WhtSpc 1 5) (Id . "title-foot") (WhtSpc 1 16) (Stri||
|| ng . "thanks") (EndLine 0))))) (BkEof (0 . 0) (EOF)))
```

2026-01-09: It looks right, but the second remark is inside the first. — 2026-01-10: Fixed.

Check that writing the block as Scheme works:

```
(write-scheme-blocks-to "scratch-r.scm" blocks )
(file-verbatim "scratch-r.scm" )
```

→

```
___  ___ file verbatim: scratch-r.scm ___  ___
;;; This makes it look like \TeXScm.
;#:= program-title "Title"
;;;
;#:= author "Awe Thor"
;#:= copyright "Copyright \\copyright\\ 2026 B.C."
;#:= title-foot "thanks"
___  ___ end verbatim: scratch-r.scm___  ___
```

⟶

```
(write-TeX-blocks-to "scratch-r.tex" blocks )
```

Then the LaTeX result
(file-TeX “`scratch-r.tex`” )
→

```
___ ___ file TeX: scratch-r.tex ___ ___
This makes it look like TeX←Scm.
:= program-title “Title”
:= author “Awe␣Thor”
:= copyright “Copyright␣\\copyright\\␣2026␣B.C.”
:= title-foot “thanks”
___ ___ end TeX: scratch-r.tex ___ ___
```

That part works now, but here is a demonstation of the debug procedure. The first step is to put calls to the *debug* procedure at strategic places in the `texscm.scm` program. Specifically, I put:

```
(define (:read-block)
  (let* ( (white2 (read-white2 0 0))
          (lxm (read-lexeme)) )
    (debug "read-block white2=" white2 ", lxm=" lxm)
    (cond ( (eq? (lxm-type lxm) 'EOF)
    ... ... ))))
```

into the source and remade. I have taken it out again, but here is how it looked:

```
___ ___ file verbatim: debug2.txt ___ ___
read-block␣white2=(2␣.␣0),␣lxm=(Semicolon␣3␣.␣"␣Debug␣now␣on")--Dbg;
read-block␣white2=(0␣.␣0),␣lxm=(Open)--Dbg;
debug-test="begin␣test"--Dbg;
read-block␣white2=(1␣.␣0),␣lxm=(Open)--Dbg;
read-block␣white2=(0␣.␣0),␣lxm=(Semicolon␣3␣.␣"␣This␣makes␣it␣look␣like␣\\TeXScm.")--Dbg;
read-block␣white2=(0␣.␣0),␣lxm=(SemiSharp␣(Id␣.␣":=")␣(WhtSpc␣1␣5)␣(Id␣.␣"program-title")␣
WhtSpc␣1␣19)␣(String␣.␣"Title")␣(EndLine␣0))--Dbg;
→ read-block␣white2=(0␣.␣0),␣lxm=(Semicolon␣3␣.␣"")--Dbg;
read-block␣white2=(0␣.␣0),␣lxm=(SemiSharp␣(Id␣.␣":=")␣(WhtSpc␣1␣5)␣(Id␣.␣"author")␣(WhtSpc
1␣12)␣(String␣.␣"Awe␣Thor")␣(EndLine␣0))--Dbg;
read-block␣white2=(0␣.␣0),␣lxm=(EOF)--Dbg;
read-block␣white2=(2␣.␣0),␣lxm=(Semicolon␣3␣.␣"␣file-verbatim␣must␣open␣the␣file---trouble
if␣already␣open")--Dbg;
read-block␣white2=(0␣.␣0),␣lxm=(Open)--Dbg;
___ ___ end verbatim: debug2.txt___ ___
```

Save the current debug port and open a new one.
(**define** dbsave debug-port )
(**set!** debug-port (open-output-file “`debug2.txt`” ))
Debug now on
(debug “`debug-test=`” “`begin␣test`” )
(**define** blocks (read-blocks-from “`scratch.scm`” ))
file-verbatim must open the file—trouble if already open
(**begin** (close-port debug-port ) (**set!** debug-port #**f** ))
(**and** #**f** (file-verbatim “`debug2.txt`” ))
;#->
Now that's done, put debug-port back as it was.
(**set!** debug-port dbsave )
This all shows that the program can read and write blocks in new version 3 encoding. Problem is there are two versions of `reply-to-remark`, one does what it is told but screws up block.

(read-blocks-from-lines “(*␣2␣pi)”
“;;;␣line”
“␣␣␣␣;;;␣line1”
“␣␣␣␣;;;␣line2”
“␣␣␣␣;;;␣line3␣␣␣␣”
“;#=>␣(*␣2␣pi)”
“␣␣␣␣(*␣2␣pi)”
“␣x␣␣” )
=>
?⟹
((BkData (0 . 0 )((Open )(Id . “*” )(WhtSpc 1 3 )(Number . “2”
)(WhtSpc 1 5 )(Id . “pi” )(Close )(EndLine 0 )))(BkComment3 (1 . 0
)(“␣line” ))(BkComment3 (0 . 4 )(“␣line1” “␣line2” “␣line3” ))(BkRemark (0 . 0 )(
(Remark ((Id . “=>” )(WhtSpc 1 5 )(Open )(Id . “*” )(WhtSpc 1 8 )
(Number . “2” )(WhtSpc 1 10 )(Id . “pi” )(Close )(EndLine 0 )))))
(BkData (0 . 4 )((Open )(Id . “*” )(WhtSpc 1 7 )(Number . “2” )
(WhtSpc 1 9 )(Id . “pi” )(Close )(EndLine 0 )))(BkData (1 . 1 )
((Id . “x” )(EndLine 0 )))(BkEof (1 . 0 )(EOF )))

Now the scratch file now contains those lines;
(file-verbatim “scratch.txt” )
→

___ ___ file verbatim: scratch.txt ___ ___
(*␣2␣pi)
;;;␣line
␣␣␣␣;;;␣line1
␣␣␣␣;;;␣line2
⟶ ␣␣␣␣;;;␣line3
;#=>␣(*␣2␣pi)
␣␣␣␣(*␣2␣pi)
␣x
___ ___ end verbatim: scratch.txt___ ___

(**define** blocks
(read-blocks-from “scratch.txt” ))
. . . and this is the result of reading that as blocks:
blocks
cut

```
‖ ⟹((BkData (0 . 0) ((Open) (Id . "*") (WhtSpc 1 3) (Number . "2") (WhtSpc 1 5)‖
‖  (Id . "pi") (Close) (EndLine 0))) (BkComment3 (1 . 0) (" line")) (BkComment‖
‖ 3 (0 . 4) (" line1" " line2" " line3")) (BkRemark (0 . 0) ((Remark ((Id . "=‖
‖ >") (WhtSpc 1 5) (Open) (Id . "*") (WhtSpc 1 8) (Number . "2") (WhtSpc 1 10)‖
‖  (Id . "pi") (Close) (EndLine 0))))) (BkData (0 . 4) ((Open) (Id . "*") (Wht‖
‖ Spc 1 7) (Number . "2") (WhtSpc 1 9) (Id . "pi") (Close) (EndLine 0))) (BkDa‖
‖ ta (1 . 1) ((Id . "x") (EndLine 0))) (BkEof (1 . 0) (EOF)))
```

(file-verbatim “scratch.txt” )
→

```
___ ___ file verbatim: scratch.txt ___ ___
(*␣2␣pi)
;;;␣line
␣␣␣␣;;;␣line1
␣␣␣␣;;;␣line2
␣␣␣␣;;;␣line3
;#=>␣(*␣2␣pi)
␣␣␣␣(*␣2␣pi)
␣x
___ ___ end verbatim: scratch.txt___ ___
```

Trailing blanks have been deleted. Where did they go? — 2025-20-27 Rather than answer that question I always delete them. See procedure *cut-trailing-blanks*. Try a few data blocks

```
(read-blocks-from-lines
        "(define␣zero␣0)"
        "␣␣␣(define␣one␣1)"
        "(define␣two␣2)␣␣#|␣II␣|#"
        "(define␣three␣3)␣;␣III"
        "(define␣four␣4)␣(define␣five␣5)" )
=>
?⟹
((BkData (0 . 0 )((Open )(Id . "define" )(WhtSpc 1 8 )(Id . "zero"
)(WhtSpc 1 13 )(Number . "0" )(Close )(EndLine 0 )))(BkData (1 . 3
)((Open )(Id . "define" )(WhtSpc 1 11 )(Id . "one" )(WhtSpc 1 15 )(
Number . "1" )(Close )(EndLine 0 )))(BkData (1 . 0 )((Open )(Id
. "define" )(WhtSpc 1 8 )(Id . "two" )(WhtSpc 1 12 )(Number . "2" )(Close )
(WhtSpc 2 16 )(CmntShort . "␣II␣" )(EndLine 0 )))(BkData (1 . 0 )((Open
)(Id . "define" )(WhtSpc 1 8 )(Id . "three" )(WhtSpc 1 14 )(Number . "3" )
(Close )(WhtSpc 1 17 )(Semicolon . (1 . "␣III" ))(EndLine 0 )))(BkData (
1 . 0 )((Open )(Id . "define" )(WhtSpc 1 8 )(Id . "four" )(WhtSpc 1
13 )(Number . "4" )(Close )(WhtSpc 1 16 )(Open )(Id . "define" )(WhtSpc 1
24 )(Id . "five" )(WhtSpc 1 29 )(Number . "5" )(Close )(EndLine 0 )))
(BkEof (1 . 0 )(EOF )))
```

Note that the indentation in define one shows up in white square after BkData, not in WhtSpc.

```
(read-blocks-from-lines
        ";#=>␣66␣99"
        ";#=>␣one␣two␣three"
        ";#␣␣four␣five␣six"
        ";#␣␣seven␣eight␣nine"
        ";#=>␣ten" )
=>
?⟹
((BkRemark (0 . 0 )((Remark ((Id . "=>" )(WhtSpc 1 5 )(Number . "66"
)(WhtSpc 1 8 )(Number . "99" )(EndLine 0 )))(Remark ((Id . "=>" )(
WhtSpc 1 5 )(Id . "one" )(WhtSpc 1 9 )(Id . "two" )(WhtSpc 1 13 )(Id
. "three" )(EndLine 0 )(NewLine 0 )(WhtSpc 2 4 )(Id . "four" )(WhtSpc 1 9 )
(Id . "five" )(WhtSpc 1 14 )(Id . "six" )(EndLine 0 )(NewLine 0 )(WhtSpc 2
4 )(Id . "seven" )(WhtSpc 1 10 )(Id . "eight" )(WhtSpc 1 16 )(Id . "nine"
)(EndLine 0 )))(Remark ((Id . "=>" )(WhtSpc 1 5 )(Id . "ten" )(
EndLine 0 )))))(BkEof (0 . 0 )(EOF )))
```

(**define** block
  (read-blocks-from "scratch.txt" ))

. . . and this is the result of reading that as blocks:

block

cut

```
‖ ⟹((BkRemark (0 . 0) ((Remark ((Id . "=>") (WhtSpc 1 5) (Number . "66") (WhtSp‖
‖ c 1 8) (Number . "99") (EndLine 0))) (Remark ((Id . "=>") (WhtSpc 1 5) (Id .‖
‖  "one") (WhtSpc 1 9) (Id . "two") (WhtSpc 1 13) (Id . "three") (EndLine 0) (‖
‖ NewLine 0) (WhtSpc 2 4) (Id . "four") (WhtSpc 1 9) (Id . "five") (WhtSpc 1 1‖
‖ 4) (Id . "six") (EndLine 0) (NewLine 0) (WhtSpc 2 4) (Id . "seven") (WhtSpc‖
‖ 1 10) (Id . "eight") (WhtSpc 1 16) (Id . "nine") (EndLine 0))) (Remark ((Id‖
‖ . "=>") (WhtSpc 1 5) (Id . "ten") (EndLine 0))))) (BkEof (0 . 0) (EOF)))
```

### 4.3.2 Write LaTeX and Scheme from Blocks

The variable three-blocks was defined by reading a file back 4.3.1. Do it again, but write it this time.

(**define** some-blocks
  (read-blocks-from-lines
   “;;;␣For␣this␣test␣some␣blocks␣are␣read,␣same␣as␣before,␣but␣longer.”
   “;;;␣Another␣comment␣block;␣following␣are␣a␣datum␣and␣remark”
   “(define␣pi␣3)”
   “;#=>␣foo␣0”
   “;;;␣This␣is␣a␣comment␣block;␣following␣is␣a␣long␣remark␣and␣some␣data.”
   “;#=>␣”
   “;#␣(␣bar␣none”
   “;#␣␣␣␣NO␣bar␣)”
   “(define␣i␣-1)”
   “(define␣e␣2)”
   ))

some-blocks

=>

?⟹

((BkComment3 (0 . 0 )(“␣For␣this␣test␣some␣blocks␣are␣read,␣same␣as␣before,␣but␣longer.” “␣Another␣
. “define” )(WhtSpc 1 8 )(Id . “pi” )(WhtSpc 1 11 )(Number . “3” )(Close )
(EndLine 0 )))(BkRemark (1 . 0 )((Remark ((Id . “=>” )(WhtSpc 1 5
)(Id . “foo” )(WhtSpc 1 9 )(Number . “0” )(EndLine 0 ))))))(BkComment3
(0 . 0 )(“␣This␣is␣a␣comment␣block;␣following␣is␣a␣long␣remark␣and␣some␣data.” ))(BkRemark (0 . 0 )((R
EndLine 0 )(NewLine 0 )(WhtSpc 1 3 )(Open )(WhtSpc 1 5 )(Id . “bar” )(
WhtSpc 1 9 )(Id . “none” )(EndLine 0 )(NewLine 0 )(WhtSpc 3 5 )(Id . “NO”
)(WhtSpc 1 8 )(Id . “bar” )(WhtSpc 1 12 )(Close )(EndLine 0 ))))
)(BkData (0 . 0 )((Open )(Id . “define” )(WhtSpc 1 8 )(Id . “i”
)(WhtSpc 1 10 )(Number . “-1” )(Close )(EndLine 0 )))(BkData (1 . 0
)((Open )(Id . “define” )(WhtSpc 1 8 )(Id . “e” )(WhtSpc 1 10 )(
Number . “2” )(Close )(EndLine 0 )))(BkEof (1 . 0 )(EOF )))

Write them to a scratch reply file.

(write-scheme-blocks-to “scratch-r.scm” some-blocks )

Now show the contents of the reply file:
(file-verbatim “scratch-r.scm” )

→

file verbatim: `scratch-r.scm`

```
;;; For this test some blocks are read, same as before, but longer.
;;; Another comment block; following are a datum and remark
(define pi 3)
;#=> foo 0
;;; This is a comment block; following is a long remark and some data.
;#=>
;# ( bar none
;#    NO bar )
(define i -1)
(define e 2)
```

end verbatim: `scratch-r.scm`

(write-TeX-blocks-to "`scratch-r.tex`" some-blocks )
Now show the contents of the reply file:
(file-verbatim "`scratch-r.tex`" )
→

file verbatim: `scratch-r.tex`

```
 For this test some blocks are read, same as before, but longer.
 Another comment block; following are a datum and remark
\begin{scheme}%
({define} {pi} 3 )
\end{scheme}
\begin{scheme}%
{={$>$}} {foo} 0
\end{scheme}
 This is a comment block; following is a long remark and some data.
\begin{scheme}%
⟶ {={$>$}} \\
 ( {bar} {none} \\
\hs{6.0mm}{NO} {bar} )
\end{scheme}
\begin{scheme}%
({define} {i} -1 )
\end{scheme}
\begin{scheme}%
({define} {e} 2 )
\end{scheme}
%EOF
```

end verbatim: `scratch-r.tex`

(file-TeX "`scratch-r.tex`" )
→

file TEX: `scratch-r.tex`

For this test some blocks are read, same as before, but longer. Another comment block; following are a datum and remark

(define pi 3 )

=> foo 0

This is a comment block; following is a long remark and some data.

⟶ =>

( bar none
NO bar )

(define i -1 )

(define e 2 )

end TEX: `scratch-r.tex`

2025-10-08 There are erroneous blank lines before and after data block — Seems to be fixed. 2025-06-12 SharpBar comment causes crash — 2025-06-13 or maybe I called the wrong procedure. seems fixed now.

```
(define shbtest
  (read-blocks-from-lines
    "(define␣(nill)"
    "␣(begin␣␣#|␣test␣|#"
    "␣␣(list)␣))" ))
```

shbtest

=>

?⟹

```
((BkData (0 . 0 )((Open )(Id . "define" )(WhtSpc 1 8 )(Open )(
Id . "nill" )(Close )(NewLine 106 )(WhtSpc 1 1 )(Open )(Id . "begin" )(WhtSpc
2 9 )(CmntShort . "␣test␣" )(NewLine 107 )(WhtSpc 2 2 )(Open )(Id . "list" )(
Close )(WhtSpc 1 9 )(Close )(Close )(EndLine 0 )))(BkEof (1 . 0 )
(EOF )))
```

(*caddr* (*car* shbtest ) )

=>

?⟹

```
((Open )(Id . "define" )(WhtSpc 1 8 )(Open )(Id . "nill" )(Close )(
NewLine 106 )(WhtSpc 1 1 )(Open )(Id . "begin" )(WhtSpc 2 9 )(CmntShort . "␣test␣" )
(NewLine 107 )(WhtSpc 2 2 )(Open )(Id . "list" )(Close )(WhtSpc 1 9 )(
Close )(Close )(EndLine 0 ))
```

### 4.3.3 Make Lexeme-Data

◦ — Make a lexeme list by reading it from the given lines.

```
(define def-use-pi
  (read-lxms-from-lines
    "(define␣pi␣3.14)"
    "pi" ))
```

def-use-pi

=>

?⟹

```
((Open )(Id . "define" )(WhtSpc 1 8 )(Id . "pi" )(WhtSpc 1 11 )(Number
. "3.14" )(Close )(NewLine 111 )(Id . "pi" )(NewLine 112 ))
```

Note that "define" and "pi" which were read as unsorted identifiers, may be sorted. If so, this is the two-pass bug. See 8.3.1.

◦ — Set up to read the lexeme list that was just made; read two ⟨datums⟩s, and end of file.

(**define** lx-port (open-input-lexemes def-use-pi ))
(**define** pi-def (read-a-lexdatum lx-port ))
pi-def
=>
?⟹
((Id . "define" )(Id . "pi" )(Number . "3.14" ))
(**define** pi-use (read-a-lexdatum lx-port ))
pi-use
=>
?⟹
(Id . "pi" )
(read-a-lexdatum lx-port )
=>
?⟹
(EOF )
(**define** lvl (read-lxms-from-lines "(set!␣v␣#(3␣6␣9))" ))
lvl
=>
?⟹
((Open )(Id . "set!" )(WhtSpc 1 6 )(Id . "v" )(WhtSpc 1 8 )(VOpen
)(Number . "3" )(WhtSpc 1 12 )(Number . "6" )(WhtSpc 1 14 )(Number . "9" )
(Close )(Close )(NewLine 114 ))
(**define** lx-port (open-input-lexemes lvl ))
(read-a-lexdatum lx-port )
→
⟶ (Id . set!) (Id . v) (LxVector (Number . 3) (Number . 6) (Number . 9))
cut
‖ ⟹((Id . "set!") (Id . "v") (LxVector (Number . "3") (Number . "6") (Number .‖
‖ "9")))

2025-11-10: I don't know why there are three lexemes rather than one list of them.
′()

End of file `tstblocks-riA.tex`

# 5 Typesetting

The following was written years ago:

> Once a Scheme program has been unzipped into the Scheme data and the layout, it becomes possible to change the layout without messing up the Scheme program. In this section there are procedures that do that in various ways. In future versions, this might include ways to edit the comments in a program without touching the "code", so that the compiler will not need to be rerun.

— No. That never worked out, partly because there was no good way to edit "layout", and partly because the objective expanded. We now want not only to typeset the program itself, but also to compute and typeset the results of evaluating parts of the program in a Read-Eval-Print-Edit Loop (REPEL).

One of the design principles of version 3 is that the source code is a Scheme program. We can edit that with `emacs` or any other Ascii editor.

Since that is the new plan, we don't need (un-)zipping, there is no separate layout, and all the following procedures should be re-written so they change lists of lexemes to lists of lexemes. The names of the new re-written procedures all begin with a colon.

## 5.1 Sorting Identifiers

The word "sorting" is not used in the sense of permuting a sequence into a monotonic sequence, but in the sense of multi-sorted algebra, or sorting socks. An identifier has a sort which partially determines the grammar of forms which contain that identifier and also determines how it should be printed. "Sorting" just means determining the sort of the identifier.

The sorts are Keyword, Variable, or Symbol. We want to use a different font to print each sort.

Main procedure *make-reply* (eventually) calls (*:block-lxmdata¡-lxmlist! lxms*) which parses the lexeme list into a lexdatum (See 4.1) which is treated as a Scheme syntactic form and traversed calling sort procedures for each sub-form. Each of those procedures determines the sort of each identifier occurrence it finds and destructively updates it by changing each occurrence of an Id into one of Vb, Kw, or Sy, that is, it sorts identifiers as variables, keywords, or symbols. This means that the original lexeme list is updated with the sort of each identifier occurrence.

### 5.1.1 Procedures that sort

The sort procedures have names like *sort-ids-in-⟨form⟩!* . Each takes two arguments, a lexdatum and an eviron.

An environ is a list of alists and quasi-quote depth indicators. Each alist in the environ corresponds to one region of code [18, §5.2] [17, §3.1] and it pairs identifiers with sorts. Each such a list is called also called a region.

Procedure *sort-ids-in-body!* assumes that the scheme form is a body, i.e. a list of definitions and expressions. Definitions within the body bind variables throughout the body.

The environ should already include the region in which definitions add bindings. That region starts empty.

:lexdata¡-lexlist is used only in :lexdata¡-blk, which is used in eval-blk

The caddr of a data block is a lexeme list, which may encode several ⟨lexdatum⟩s. Return a list of those ⟨lexdatum⟩s

```
(define (:lexdata←blk block ) (:lexdata←lexlist (caddr block )))
```

A lexeme list is just a flat list of lexemes. A lexdata is ⟨datum⟩

```
(define (:lexdata←lexlist lxms )
  (let ((lxport (open-input-lexemes lxms )))
    (let read-one
         ( (data '())
           (datum (read-a-lexdatum lxport )) )
      (if (eq? (lxm-type datum ) 'EOF )
          data
          (read-one (append data (list datum )) (read-a-lexdatum lxport ))))))
```

Why does this not call :lexdata¡-lexlist ? used only in :make-reply

```
(define (:block-lxmdata←lxmlist! blk )
  (let ((lxin (open-input-lexemes (caddr blk ))) )
    (let do-datum ( (lexdatum (read-a-lexdatum lxin ))
                    (lxds '()) )
      (if (lx-eof? lexdatum )
          (begin (set-cdr! (cddr blk ) (list (reverse lxds )))
                 (cadddr blk ) )
          (begin
            (do-datum (read-a-lexdatum lxin ) (cons lexdatum lxds )) )))))
```

The *sort-in-datum!* procedure sorts a the whole datum with the same environment, ignoring any binding forms. This allows testing of font change without complicated font choice.

```
(define (:sort-in-datum! lxd env )
    (if (lxm-type lxd )
        (:sort-lexeme! lxd env )
        (if (null? lxd )
            '()
```

```
                (if (pair? lxd )
                    (begin (:sort-in-datum! (car lxd ) env )
                           (:sort-in-datum! (cdr lxd ) env ) )
                    "form␣err" ) )))
```

Given a definition return (as a symbol) the variable that is bound in the enclosing region. E.g. given "`(define pi 3)`" return `pi`, given "`(define (f x) x)`" return `f`.

```
(define (:dbound-var def )
  (let* ( (dfv (cadr def ))
          (v (if (lxm-type dfv ) dfv (car dfv ))) )
    (string→symbol (cdr v ))))
```

This re-names and replaces *def-binds*.

Procedure *:collect-bindings* takes a lexdatum, which is assumed to encode a body, that is, a list of definitions and expressions. It returns a list of pairs. Each pair is an identifier together with a one of the symbols `Vb` or `Kw`. An identifier is sorted into the list as a variable if it is bound by an ordinary definition and is sorted as a keyword if it is bound by a syntax definition.

It must make a pass through the top level of whole body sorting the label of each form to see if it a keyword and if so whether it is a definition. We look for occurrences of **define** or **define-syntax**, which introduce new bindings, and for occurrences of **begin**, which collects forms without creating a new range in the program.

When a definition is found, the identifiers it defines, each with the appropriante sort, are added to the collection of bindings. This collection will be used to sort bound occurrences when the body is scanned in depth.

```
(define (:collect-bindings lexdat-body env )
  (let find-defs ( (s lexdat-body )
                   (ls '()) )
    (if (not (pair? s ))
        ls
        (find-defs
          (cdr s )
          (let* ( (first-form (car s ))
                  (op (and (pair? first-form ) (car first-form )))
                  (symop (and (string? (cdr op ))(string→symbol (cdr op ))))
                  (sop (sort-of symop env )) )
            (if (eq? sop 'Kw )
                (case symop
                  ((define ) (cons (cons (:dbound-var first-form ) 'Vb ) ls ))
                  ((define-syntax )(cons (cons (:dbound-var first-form ) 'Kw ) ls ))
                  ((begin ) (append! (:collect-bindings (cdr first-form ) env ) ls ))
                  (else ls ) )
                ls ) ))))))
(define (:sort-ids-in-body! scheme-exps env )
  (let ( (e (add-bindings! (:collect-bindings scheme-exps env ) (cons (list ) env ))) )
    (let sort ( (s scheme-exps ) )
      (if (null? s )
          'done
          (begin
            (:sort-ids-in-form! (car s ) e )
            (sort (cdr s )  ) ))))))
(define (:sort-lexeme! lexeme env )
  (let ( (sort (lxm-type lexeme )) )
    (case sort
      ((Id Vb Sy Kw )
       (begin
         (set! sort (sort-of (string→symbol (cdr lexeme )) env ))
```

```
        (set-car! lexeme sort )) )
      ((Number String Char Boolean LxVector Result Resultlxs Result-TeX ) '(no-op ))
      (else (set! sort 'LexErr ) ) )
    sort ))
```

A form may be a definition or an expression.

```
(define (xx:sort-ids-in-form! sform env )
  (case (lxm-type sform )
        ((Id Vb Sy Kw ) (set-car! sform (sort-of (string→symbol (cdr sform )) env )))
        ((Number String Char Boolean LxVector Result Resultlxs Result-TeX ) '())
        ((Abbrev ) (error "Abbrev␣in␣lex-datum" ) )
        ((#f )
         (if (null? sform )
             '()
             (if (pair? sform )
                 (begin (:sort-ids-in-form! (car sform ) env )
                        (:sort-ids-in-form! (cdr sform ) env ) )
                 "form␣err" ) ))))
```

Procedure *:keyword-meaning* checks whether its argument is a keyword and if so returns something which determines its binding structure. This is not yet designed; this stub just returns a symbol.

```
(define (:keyword-meaning id env )
  (if (and (lxm-type id ) (eq? (sort-of (string→symbol (cdr id )) env ) 'Kw ))
      (string→symbol (cdr id ))
      #f ))
```

A form may be a definition or an expression. If it is a definition, the defined identifier has already been entered into the *env* by *:collect-bindings*.

```
(define (:sort-ids-in-form! sform env )
  (if (null? sform )
      '()
      (if (lxm-type sform )
          (:sort-lexeme! sform env )
          (if (pair? sform )
              (let ((m (:keyword-meaning (car sform ) env )))
                (if #f
                    (let ((p (assoc m :keyword-sort-procs )) )
                      #f )
                    (begin (:sort-ids-in-form! (car sform ) env )
                           (:sort-ids-in-form! (cdr sform ) env ) ) ))
              "form␣err" ) )))
(define (:sort-ids-in-let! sform    env )
  (let ( (named? (eq? (lxm-type (cadr sform )) 'Id ) ))
    (let ( (name (if named? (cadr sform ) #f ))
           (bnds ((if named? caddr cadr ) sform ))
           (body ((if named? cdddr cddr ) sform )) )
      (let ((fmls (map car bnds )))
        (let ((e (add-bindings!
                   (sort-as-vb
                     (if named? (cons name fmls ) fmls ))
                   (add-locale env ))))
          (if named? (:sort-lexeme! name e ))
          (:for-each-form
            (lambda (bind )
              (:sort-ids-in-form! (car bind )   env )
```

```
                bnds )
              (:sort-ids-in-body! body e ) ))))))
(define (:sort-ids-in-def! sform env ) #f )
(define (:sort-ids-in-lambda! sform env ) #f )
(define (:for-each-form f formlist ) #f )
(define :keyword-sort-procs
  '(
    (let . ,:sort-ids-in-let! )
    (define . ,:sort-ids-in-def! )
    (lambda . ,:sort-ids-in-lambda! )
    ))
```

Included File `tstsort-riA.tex`

## 5.2 File: `tstsort.scm` — Test Identifier Sorting

Test sorting procedures

This file contains a continuation of testing so load the testing procedures

```
(list ts:version ts:subversion )
=>
?⟹
("$3-1/10$" "" )
(load "tstprocs.scm" )
(load "texscm.scm" )
(list ts:version ts:subversion )
=>
?⟹
("$3-1/11$" "" )
```

The *:primal-environ* is too long. Make a new short environment that we can show and look at without vexation.

```
(define stub-environ
  (list
   (append
    (map (lambda (id ) (cons id 'Vb )) '(acos append car cdr ) )
    (map (lambda (id ) (cons id 'Kw )) '(define lambda begin define-syntax ) ))))
```

Now make a block. (Really a list of two blocks, a data block and an end of file.)

```
(define test-blocks
  (read-blocks-from-lines
   "(define␣exp␣((lambda␣(x)␣x␣x)"
   "␣␣(lambda␣(define)␣append)))" ))
(define test-block (car test-blocks ))
(define lexlist (caddr test-block ))
(define lexdata (:block-lxmdata←lxmlist! test-block ))
```

See how it looks.

```
test-block
=>
?⟹
(BkData (0 . 0 )((Open )(Id . "define" )(WhtSpc 1 8 )(Id . "exp" )
(WhtSpc 1 12 )(Open )(Open )(Id . "lambda" )(WhtSpc 1 21 )(Open )(Id
. "x" )(Close )(WhtSpc 1 25 )(Id . "x" )(WhtSpc 1 27 )(Id . "x" )
(Close )(NewLine 2 )(WhtSpc 2 2 )(Open )(Id . "lambda" )(WhtSpc 1 10 )(
Open )(Id . "define" )(Close )(WhtSpc 1 19 )(Id . "append" )(Close )(Close )
```

(Close )(EndLine 0 ))(((Id . “define” )(Id . “exp” )(((Id . “lambda”
)((Id . “x” ))(Id . “x” )(Id . “x” ))((Id . “lambda” )(
(Id . “define” ))(Id . “append” ))))))

The BkData marker is followed by the White2 spec, the Lexeme list as it was read, and the lxmdata that was computed and saved by procedure *:block-lxmdata¡-lxmlist!*.

the lexlist is a linear list of lexemes:

lexlist

=>

?⟹

((Open )(Id . “define” )(WhtSpc 1 8 )(Id . “exp” )(WhtSpc 1 12 )(Open
)(Open )(Id . “lambda” )(WhtSpc 1 21 )(Open )(Id . “x” )(Close )(
WhtSpc 1 25 )(Id . “x” )(WhtSpc 1 27 )(Id . “x” )(Close )(NewLine 2 )
(WhtSpc 2 2 )(Open )(Id . “lambda” )(WhtSpc 1 10 )(Open )(Id . “define” )
(Close )(WhtSpc 1 19 )(Id . “append” )(Close )(Close )(Close )(EndLine 0 )
)

The lexdata is a tree structure with lexemes at the leaves. All the Open and Close lexemes have been replaced by real parentheses.

lexdata

=>

?⟹

(((Id . “define” )(Id . “exp” )(((Id . “lambda” )((Id . “x” ))
(Id . “x” )(Id . “x” ))((Id . “lambda” )((Id . “define” ))(Id
. “append” )))))

The *sort-in-datum!* procedure sorts a the whole datum with the same environment, ignoring any binding forms. This allows testing of font change without complicated font choice.

```
(define (sort-in-datum! lxd env )
      (if (lxm-type lxd )
            (:sort-lexeme! lxd env )
            (if (null? lxd )
                  '()
                (if (pair? lxd )
                      (begin (sort-in-datum! (car lxd ) env )
                                  (sort-in-datum! (cdr lxd ) env ) )
                      “form␣err” ) )))
```

Apply that procedure, For prototype, sort all ids as in stub environ.

(sort-in-datum! lexdata stub-environ )

The identifers occurrences in *lexdata* are sorted.

lexdata

=>

?⟹

(((Kw . “define” )(Id . “exp” )(((Kw . “lambda” )((Id . “x” ))
(Id . “x” )(Id . “x” ))((Kw . “lambda” )((Kw . “define” ))(Vb
. “append” )))))

But they are the same as the identifiers occurrences in *lexlist*.

lexlist

=>

?⟹

((Open )(Kw . “define” )(WhtSpc 1 8 )(Id . “exp” )(WhtSpc 1 12 )(Open
)(Open )(Kw . “lambda” )(WhtSpc 1 21 )(Open )(Id . “x” )(Close )(
WhtSpc 1 25 )(Id . “x” )(WhtSpc 1 27 )(Id . “x” )(Close )(NewLine 2 )
(WhtSpc 2 2 )(Open )(Kw . “lambda” )(WhtSpc 1 10 )(Open )(Kw . “define” )
(Close )(WhtSpc 1 19 )(Vb . “append” )(Close )(Close )(Close )(EndLine 0 )
)

Write it as Scheme.

```
(with-output-to-file "scratch-rd.scm"
  (lambda () (ts:write-scheme-lxs lexlist )))
(file-verbatim "scratch-rd.scm" )
→

    ___ ___ file verbatim: scratch-rd.scm ___ ___
    (define␣exp␣((lambda␣(x)␣x␣x)
⟶  ␣␣(lambda␣(define)␣append)))
    ___ ___ end verbatim: scratch-rd.scm___ ___
```

2025-11-03: Oh, Oh!!! only one line shows up on the printed page, but there are two (correct) lines in the file. — They are there if `NewLine` instead of `EndLine` is used in *read-lexemes-on-line*. I wondered if something was wrong with reading or writing the file.

```
bash> cat scratch-rd.scm
(define exp ((lambda (x) x x)
  (lambda (define) append)))

bash> od -x scratch-rd.scm
  0000000 6428 6665 6e69 2065 7865 2070 2828 616c | (define exp ((la
  0000020 626d 6164 2820 2978 7820 7820 0a29 2020 | mbda (x) x x)\n
  0000040 6c28 6d61 6462 2061 6428 6665 6e69 2965 | (lambda (define)
  0000060 6120 7070 6e65 2964 2929 000a           |  append)))?\0
  0000073
```

That was a goose chase. I found nothing wrong with either file-verbatim or read-line.

2025-11-04: This version works with `EndLine` flushes *line-buf* in *write-scheme-lexeme*. It seems that `texscm.scm` and `texscm-r.scm` are identical, but it messes up on copying remarks.

```
(with-output-to-file "scratch-r2.scm"
  (lambda () (ts:write-scheme-blk test-block )))
(file-verbatim "scratch-r2.scm" )
→

    ___ ___ file verbatim: scratch-r2.scm ___ ___
    (define␣exp␣((lambda␣(x)␣x␣x)
⟶  ␣␣(lambda␣(define)␣append)))
    ___ ___ end verbatim: scratch-r2.scm___ ___
```

Also write to TEX

```
(with-output-to-file "scratch-r.tex"
  (lambda () (write-TeX-lxms lexlist )))
(file-verbatim "scratch-r.tex" )
→

    ___ ___ file verbatim: scratch-r.tex ___ ___
    \begin{scheme}%
    (\kw{define}␣␣{exp}␣␣((\kw{lambda}␣␣({x}␣)␣{x}␣␣{x}␣)\\
⟶  \hs{4.0mm}(\kw{lambda}␣␣(\kw{define}␣)␣\vb{append}␣)))
    \end{scheme}
    ___ ___ end verbatim: scratch-r.tex___ ___
```

And show the final result.

```
(file-TeX "scratch-r.tex" )
→
```

file TEX: `scratch-r.tex`

```
⟶ (define exp ((lambda (x ) x x )
      (lambda (define ) append )))
```

end TEX: `scratch-r.tex`

Now make another block list

```
(define test-blocks
  (read-blocks-from-lines
    "(␣(define␣e␣2)"
    "␣␣␣(define␣pi␣3)"
    "␣␣␣(begin␣(define␣a␣'a)␣(define␣b␣'b))"
    "␣␣␣(define-syntax␣fst␣car)␣)" ))
```

See how it looks.

```
test-blocks
=>
?⟹
((BkData (0 . 0 )((Open )(WhtSpc 1 2 )(Open )(Id . "define" )(
WhtSpc 1 10 )(Id . "e" )(WhtSpc 1 12 )(Number . "2" )(Close )(NewLine 6 )
(WhtSpc 2 2 )(Open )(Id . "define" )(WhtSpc 1 10 )(Id . "pi" )(WhtSpc 1
13 )(Number . "3" )(Close )(NewLine 7 )(WhtSpc 2 2 )(Open )(Id . "begin"
)(WhtSpc 1 9 )(Open )(Id . "define" )(WhtSpc 1 17 )(Id . "a" )(WhtSpc
1 19 )(Abbrev . quote )(Id . "a" )(Close )(WhtSpc 1 23 )(Open )(Id
. "define" )(WhtSpc 1 31 )(Id . "b" )(WhtSpc 1 33 )(Abbrev . quote )(Id .
"b" )(Close )(Close )(NewLine 8 )(WhtSpc 2 2 )(Open )(Id . "define-syntax" )(
WhtSpc 1 17 )(Id . "fst" )(WhtSpc 1 21 )(Id . "car" )(Close )(WhtSpc 1 26
)(Close )(EndLine 0 )))(BkEof (1 . 0 )(EOF )))
(define test-block (car test-blocks ))
(define lexlist (caddr test-block ))
lexlist
=>
?⟹
((Open )(WhtSpc 1 2 )(Open )(Id . "define" )(WhtSpc 1 10 )(Id . "e"
)(WhtSpc 1 12 )(Number . "2" )(Close )(NewLine 6 )(WhtSpc 2 2 )(Open )
(Id . "define" )(WhtSpc 1 10 )(Id . "pi" )(WhtSpc 1 13 )(Number . "3" )(
Close )(NewLine 7 )(WhtSpc 2 2 )(Open )(Id . "begin" )(WhtSpc 1 9 )(Open
)(Id . "define" )(WhtSpc 1 17 )(Id . "a" )(WhtSpc 1 19 )(Abbrev . quote )
(Id . "a" )(Close )(WhtSpc 1 23 )(Open )(Id . "define" )(WhtSpc 1 31 )
(Id . "b" )(WhtSpc 1 33 )(Abbrev . quote )(Id . "b" )(Close )(Close )
(NewLine 8 )(WhtSpc 2 2 )(Open )(Id . "define-syntax" )(WhtSpc 1 17 )(Id . "fst"
)(WhtSpc 1 21 )(Id . "car" )(Close )(WhtSpc 1 26 )(Close )(EndLine 0 )
)
(datum←lexdatum lexdata )
=>
?⟹
((define exp ((lambda (x )x x )(lambda (define )append ))))
(define lexdata (:block-lxmdata←lxmlist! test-block ))
lexdata
=>
?⟹
```

((((Id . “define” )(Id . “e” )(Number . “2” ))((Id . “define” )(
Id . “pi” )(Number . “3” ))((Id . “begin” )((Id . “define” )(Id . “a”
)((Id . “quote” )(Id . “a” )))((Id . “define” )(Id . “b” )(
(Id . “quote” )(Id . “b” ))))((Id . “define-syntax” )(Id . “fst” )(Id
. “car” ))))

Note that lexdata has been saved in the block
test-block

=>

?⟹

(BkData (0 . 0 )((Open )(WhtSpc 1 2 )(Open )(Id . “define” )(WhtSpc
1 10 )(Id . “e” )(WhtSpc 1 12 )(Number . “2” )(Close )(NewLine 6 )(
WhtSpc 2 2 )(Open )(Id . “define” )(WhtSpc 1 10 )(Id . “pi” )(WhtSpc 1 13
)(Number . “3” )(Close )(NewLine 7 )(WhtSpc 2 2 )(Open )(Id . “begin” )
(WhtSpc 1 9 )(Open )(Id . “define” )(WhtSpc 1 17 )(Id . “a” )(WhtSpc 1
19 )(Abbrev . quote )(Id . “a” )(Close )(WhtSpc 1 23 )(Open )(Id .
“define” )(WhtSpc 1 31 )(Id . “b” )(WhtSpc 1 33 )(Abbrev . quote )(Id . “b”
)(Close )(Close )(NewLine 8 )(WhtSpc 2 2 )(Open )(Id . “define-syntax” )(WhtSpc
1 17 )(Id . “fst” )(WhtSpc 1 21 )(Id . “car” )(Close )(WhtSpc 1 26 )
(Close )(EndLine 0 ))((((Id . “define” )(Id . “e” )(Number . “2” )
)((Id . “define” )(Id . “pi” )(Number . “3” ))((Id . “begin” )((
Id . “define” )(Id . “a” )((Id . “quote” )(Id . “a” )))((Id .
“define” )(Id . “b” )((Id . “quote” )(Id . “b” ))))((Id . “define-syntax”
)(Id . “fst” )(Id . “car” )))))

(:collect-bindings (*car* lexdata ) stub-environ )

=>

?⟹

((fst . Kw )(b . Vb )(a . Vb )(pi . Vb )(e . Vb )
)

(add-bindings! (:collect-bindings (*car* lexdata ) stub-environ )
(*cons* (*list* ) stub-environ ))

=>

?⟹

(((fst . Kw )(b . Vb )(a . Vb )(pi . Vb )(e . Vb
))((acos . Vb )(append . Vb )(car . Vb )(cdr . Vb )(define .
Kw )(lambda . Kw )(begin . Kw )(define-syntax . Kw )))

Don’t do it again! Prove the two-pass bug is fixed. sort the identifiers

(:sort-ids-in-body! (*car* lexdata ) stub-environ )

They should have changed
lexdata

=>

?⟹

((((Kw . “define” )(Vb . “e” )(Number . “2” ))((Kw . “define” )(
Vb . “pi” )(Number . “3” ))((Kw . “begin” )((Kw . “define” )(Vb . “a”
)((Id . “quote” )(Vb . “a” )))((Kw . “define” )(Vb . “b” )(
(Id . “quote” )(Vb . “b” ))))((Kw . “define-syntax” )(Kw . “fst” )(Vb
. “car” ))))

test-block

=>

?⟹

(BkData (0 . 0 )((Open )(WhtSpc 1 2 )(Open )(Kw . “define” )(WhtSpc
1 10 )(Vb . “e” )(WhtSpc 1 12 )(Number . “2” )(Close )(NewLine 6 )(
WhtSpc 2 2 )(Open )(Kw . “define” )(WhtSpc 1 10 )(Vb . “pi” )(WhtSpc 1 13
)(Number . “3” )(Close )(NewLine 7 )(WhtSpc 2 2 )(Open )(Kw . “begin” )
(WhtSpc 1 9 )(Open )(Kw . “define” )(WhtSpc 1 17 )(Vb . “a” )(WhtSpc 1
19 )(Abbrev . quote )(Vb . “a” )(Close )(WhtSpc 1 23 )(Open )(Kw .
“define” )(WhtSpc 1 31 )(Vb . “b” )(WhtSpc 1 33 )(Abbrev . quote )(Vb . “b”
)(Close )(Close )(NewLine 8 )(WhtSpc 2 2 )(Open )(Kw . “define-syntax” )(WhtSpc
1 17 )(Kw . “fst” )(WhtSpc 1 21 )(Vb . “car” )(Close )(WhtSpc 1 26 )
(Close )(EndLine 0 ))((((Kw . “define” )(Vb . “e” )(Number . “2” )
)((Kw . “define” )(Vb . “pi” )(Number . “3” ))((Kw . “begin” )((
Kw . “define” )(Vb . “a” )((Id . “quote” )(Vb . “a” )))((Kw .
“define” )(Vb . “b” )((Id . “quote” )(Vb . “b” ))))((Kw . “define-syntax”
)(Kw . “fst” )(Vb . “car” )))))

Write it as Scheme.

(*with-output-to-file* “scratch-r.scm”
(**lambda** () (ts:write-scheme-lxs (*caddr* test-block ))))
(file-verbatim “scratch-r.scm” )
→

```
    ___ ___ file verbatim: scratch-r.scm ___ ___
    (␣(define␣e␣2)
    ␣␣(define␣pi␣3)
⟶  ␣␣(begin␣(define␣a␣'a)␣(define␣b␣'b))
    ␣␣(define-syntax␣fst␣car)␣)
    ___ ___ end verbatim: scratch-r.scm___ ___
```

Also write to TEX and show the final result.

(*with-output-to-file* “scratch-r.tex”
(**lambda** () (write-TeX-lxms (*caddr* test-block ))))
(file-TeX “scratch-r.tex” )
→

```
    ___ ___ file TEX: scratch-r.tex ___ ___

    ( (define e 2 )
⟶    (define pi 3 )
      (begin (define a 'a ) (define b 'b ))
      (define-syntax fst car ) )
    ___ ___ end TEX: scratch-r.tex ___ ___
```

We could spend a lot of time at this, but it's useless unless it's usable on itself. Get rid of the `-x` version This? `(ts:make-reply "tstsort" "noeval" "includable")`
Stomps on itself, changes results of above evaluations to “none”.

Things go funny unless this is last in the file:

(**define** z 0 )

End of file `tstsort-riA.tex`

```
(define (sort-ids-in-body! scheme-exps list-layout env )
  (let ( (e (add-bindings! (def-binds scheme-exps env ) env )) )
    (let clsfy ( (s scheme-exps )
                 (f (blow-atmosphere list-layout )) )
      (if (null? f )
          'done
          (if (not (and (pair? f ) (pair? s )))
```

```
                (list "body-error␣" s f )
                (begin
                  (sort-ids-in-form! (car s )(car f ) e )
                  (clsfy (cdr s ) (solid-cdr f )    ) )))))
```

A form may be a definition or an expression.

A definition may modify the current locale, that is, the first one on the environ list, but does not update the environ itself.

```
(define (sort-ids-in-form! sform layout env )
  (if (pair? layout )
      (case (car layout )
        ((Id Vb Sy Kw ) (set-car! layout (sort-of sform env )))
        ((Number String Char Boolean LxVector Result Resultlxs Result-TeX ) '())
        ((Abbrev )
         (case (car sform )
           ((quote )
            (sort-ids-in-quote! sform layout env ))
           ((qquote )
            (sort-ids-in-qquote! sform layout env ))
           ((unquote unquote-splicing ) '()) ))
        ((Dot ) '())
        ((List )
         (if (null? sform )
             '()
             (if (pair? sform )
                 (let ((m (meaning (car sform ) env )))
                   (if (eq? m 'notkw )
                       (for-each-form (lambda (s f ) (sort-ids-in-form! s f env ))
                                      sform (solid-cdr layout ))
                       (let ( (list-layout (solid-cdr layout ))
                              (p (assoc m keyword-sort-procs )) )
                         (set-car! (car list-layout ) 'Kw )
                         (if p
                             ((cdr p ) sform list-layout env )
                             (for-each-form
                               (lambda (s f ) (sort-ids-in-form! s f env ))
                               (cdr sform ) (solid-cdr list-layout ))))))
                 (begin (display "bad␣sform␣" ) (display sform )
                        (display ";␣with␣layout␣" ) (display layout )(newline )) )))
        ((CmntShort CmntLong SemiSharp Remark Error ) '())
        (else (display "bad␣layout␣" ) (display layout ) (newline )))
      (begin (display "null␣layout␣" ) (display layout ) (newline ))     ))
```

The arguments are of the form

*sform* = (**define** (*f x*...) ⟨body⟩)

*layout* = ((Id) (List (Id) (Id)...) ⟨body-layout⟩)

```
(define (sort-ids-in-def! sform layout env )
  (let ( (defs (bound-vars sform )) )
    (let ((e (add-bindings! (sort-as-vb defs ) (add-locale env ))))
      (if (list? (cadr sform ))
          (for-each-form
            (lambda (s f ) (sort-ids-in-form! s f e ))
            (cadr sform )
            (list-form (solid-cadr layout )) )
          (sort-ids-in-form! (cadr sform ) (solid-cadr layout ) e ))
```

```
        (sort-ids-in-body!
          (cdr (cdr sform ))
          (solid-cdr (solid-cdr layout ))
          e ))))
```

Convert a layout for a list to a list of layouts.

```
(define (list-form ls )
  (if (eq? (car ls ) 'List )
      (solid-cdr ls )
      (begin (newline )
             (display "not list form" ) (display ls ) (newline )  )))
(define (sort-ids-in-let*! sform layout env )
  (let ( (bnds (cadr sform ))
         (fbnds (solid-cadr layout ))
         (body (cddr sform ))
         (fbody (solid-cddr layout ))
         (new-env env ) )
    (for-each-form
     (lambda (bind fbind )
       (set! new-env (add-bindings! (sort-as-vb (car bind ))
                                    (add-locale new-env )) )
       (sort-ids-in-form! (car bind )
                          (car (list-form fbind )) new-env )
       (sort-ids-in-form! (cadr bind )
                          (solid-cadr (list-form fbind )) env )
       (set! env new-env ) )
     bnds (list-form fbnds ))
    (sort-ids-in-body! body fbody new-env ) ))
```

Sort the identifiers in a **let** form. This handles named let, but not **let***, **letrec**, **letrec***, or **let-values**.

```
(define (sort-ids-in-let! sform layout env )
  (let ( (named? (symbol? (cadr sform )))
         (lsf layout ) )
    (let ( (name (if named? (cadr sform ) #f ))
           (fname (if named? (solid-cadr lsf ) #f ))
           (bnds ((if named? caddr cadr ) sform ))
           (fbnds ((if named? solid-caddr solid-cadr ) lsf ))
           (body ((if named? cdddr cddr ) sform ))
           (fbody ((if named? solid-cdddr solid-cddr ) lsf )) )
      (let ((fmls (map car bnds )))
        (let ((e (add-bindings! (sort-as-vb
                                 (if named? (cons name fmls ) fmls ))
                                (add-locale env ))))
          (if named? (sort-ids-in-form! name fname e ))
          (for-each-form
           (lambda (bind fbind )
             (sort-ids-in-form! (car bind ) (car (list-form fbind )) e )
             (sort-ids-in-form! (cadr bind ) (solid-cadr (list-form fbind )) env ) )
           bnds (list-form fbnds ))
          (sort-ids-in-body! body fbody e ) ))))))
```

Sort the identifiers in a **lambda** form. The formals are restricted to be a list, not a single identifier or an improper (dotted) list. (a bug)

```
(define (sort-ids-in-lambda! sform layout env )
  (let ((e (add-bindings! (sort-as-vb (cadr sform ))(add-locale env ))))
```

```
    (if (pair? (cadr sform ))
        (for-each-form
          (lambda (s f ) (sort-ids-in-form! s f e ))
          (cadr sform )
          (list-form (solid-cadr layout )) )
        (sort-ids-in-form! (cadr sform ) (solid-cadr layout ) e ) )
    (sort-ids-in-body!
      (cdr (cdr sform ))
      (solid-cdr (solid-cdr layout ))
      e )))
```

Sort the identifiers in a **case** form. Identifiers in the ⟨datum⟩ part of each ⟨case clause⟩ are sorted as symbols[18, §11.4.5]

```
(define (sort-ids-in-case! sform rest-layout env )
  (let ( (qe (cons 'quote env )) )
    (sort-ids-in-form! (cadr sform ) (solid-cadr rest-layout ) env )
    (for-each-form
      (lambda (s f )
        (sort-ids-in-form! (car s ) (car (list-form f )) qe )
        (for-each-form
          (lambda (se fe )
            (sort-ids-in-form! se fe env ))
          (cdr s ) (solid-cdr (list-form f ))))
      (cddr sform ) (solid-cddr rest-layout ))))
```

The arguments are of the form
*sform* = (quoted-form ...)
*layout* = (⟨layout⟩ ...)

```
(define (sort-ids-in-quote! sform layout env )
  (let ( (e (cons 'quote env )) )
    (for-each-form
      (lambda (s f ) (sort-ids-in-form! s f e ))
      (cdr sform ) (solid-cdr layout )) ))
(define (sort-ids-in-qquote! sform layout env )
  (let ( (e (cons 'qquote env )) )
    (for-each-form
      (lambda (s f ) (sort-ids-in-form! s f e ))
      (cdr sform ) (solid-cdr layout )) ))
```

This is a list of pairs, consisting of an identifer (a keyword) and the procedure that sorts identifiers in a form begining with that keyword. Forms that begin with a variable do nothing to region sorting already in effect.

In most cases an identifier can be inserted into a quasiquote list simply by writing the identifier. The **unquote** and **unquote-splicing** keywords are exceptions. These must be enclosed in an evaluated quotation to avoid being interpreted.

```
(define keyword-sort-procs
  `(
    (let . ,sort-ids-in-let! )
    (let* . ,sort-ids-in-let*! )
    (define . ,sort-ids-in-def! )
    (lambda . ,sort-ids-in-lambda! )
    (case . ,sort-ids-in-case! )
    (, 'unquote . ,sort-ids-in-quote! )
    (, 'unquote-splicing . ,sort-ids-in-quote! )
    (, 'quote . ,sort-ids-in-quote! )
    (, 'qquote . ,sort-ids-in-qquote! )   ))
```

```
(define (sort-of id env )
  (if (null? env )
      'Id
      (let ((loc (car env )))
        (if (or (eq? loc 'quote ) (eq? loc 'qquote ))
            'Sy
            (let ((pr (assoc id loc )))
              (if pr (cdr pr ) (sort-of id (cdr env ))))))))
```

The old layout procedure prints Scheme with (some) identifiers sorted and with tab ticks. While working on that, it is important to make sure the pipeline goes all the way through.

Included File `tsteval-riA.tex`

## 5.3 File: `tsteval.scm` — Test Evaluate and Reply

This file contains a continuation of testing so load the testing procedures

```
(list ts:version ts:subversion )
=>
?⟹
("$3-1/10$" "" )
(load "tstprocs.scm" )
(load "texscm.scm" )
(list ts:version ts:subversion )
=>
?⟹
("$3-1/11$" "" )
```

In `*-ri.tex` subversions are both empty. In `*-riF.tex` they are `F` and empty repectively. This shows what?

What would be different if `texscm` were loaded first, then `tstprocs`?

Make a couple of procedures that will be convenient.

```
(define (write-scheme-blocks-to fname blocks )
  (with-output-to-file fname
    (lambda () (for-each ts:write-scheme-blk blocks )) ))
(define (write-TeX-blocks-to fname blocks )
  (with-output-to-file fname
    (lambda () (for-each ts:write-TeX-blk blocks )) ))
```

### 5.3.1 Reply to Remark

The following four lines in the `tsteval.scm` source are:

```
42
;#=> 9
(* 6 8)
;#=> 0
```

After processing they look like:

```
42
=> 9
?⟹
42
(* 6 8 )
=> 0
?⟹
48
```

Now try feeding a string directly to LaTeX.
"this␣is␣a␣test"
→
⟶this is a test
"$e^{i\\pi}␣+␣1␣=␣0$"
TeX converts that string to
→
$\longrightarrow e^{i\pi} + 1 = 0$ which is a beautiful equation.

By accident I got the reply into the middle of a line. Should this be a documented feature? Is there a better way to do it?
Try *list* and *cons*.

```
(list 'a '(12 77 ) 'c (list 'x 'y ))
=> 0
?⟹
(a (12 77 )c (x y ))
(list 'a '(12 77 ) 'c (cons 'x 'y ))
=> 0
?⟹
(a (12 77 )c (x . y ))
```

### 5.3.2 Evaluate and Reply

— Try to evaluate and make a reply

Start by writing blocks; a remark comment data block and a remark block.

```
(write-lines-to "scratch.scm"
      '( "(define␣pi␣3.14)␣␣(*␣2␣pi)"
         ";#=>"
       ))
(define twoblocks (read-blocks-from "scratch.scm" ))
(define rsvp-block (cadr twoblocks ))
rsvp-block
=>
?⟹
(BkRemark (1 . 0 )((Remark ((Id . "=>" )(EndLine 0 )))))
(define rxreply (rx:reply-to-remark-blk rsvp-block 6.28 ))
rxreply
=>
?⟹
(BkRemark (1 . 0 )((Remark ((Id . "=>" )(EndLine 0 )))(ResultIxs ((
Number . "6.28" )))))
(write-lines-to "scratch.scm"
   (list
        ";;;␣This␣makes␣it␣look␣like␣\\TeXScm."
        ";#:=␣program-title␣\"Title\""
        ";;;"
        ";#:=␣author␣\"Awe␣Thor\""
        ";#:=␣copyright␣\"Copyright␣\\\\copyright\\\\␣2026␣B.C.\""
        ";#:=␣title-foot␣\"thanks\"" ))
(define blocks (read-blocks-from "scratch.scm" ))
```

. . . and this is the result of reading that as blocks:

```
blocks
```

cut
‖ ⟹((BkComment3 (0 . 0) (" This makes it look like \\TeXScm.")) (BkRemark (0 .‖
‖ 0) ((Remark ((Id . ":=") (WhtSpc 1 5) (Id . "program-title") (WhtSpc 1 19) (‖
‖ String . "Title") (EndLine 0))))) (BkComment3 (0 . 0) ("")) (BkRemark (0 . 0‖
‖ ) ((Remark ((Id . ":=") (WhtSpc 1 5) (Id . "author") (WhtSpc 1 12) (String .‖
‖  "Awe Thor") (EndLine 0))) (Remark ((Id . ":=") (WhtSpc 1 5) (Id . "copyrigh‖
‖ t") (WhtSpc 1 15) (String . "Copyright \\copyright\\ 2026 B.C.") (EndLine 0)‖
‖ )) (Remark ((Id . ":=") (WhtSpc 1 5) (Id . "title-foot") (WhtSpc 1 16) (Stri‖
‖ ng . "thanks") (EndLine 0))))) (BkEof (0 . 0) (EOF)))

(**define** rsvp-block (*cadr* blocks ))
rsvp-block
=>
?⟹
(BkRemark (0 . 0 )((Remark ((Id . "`:=`" )(WhtSpc 1 5 )(Id . "`program-title`" )
(WhtSpc 1 19 )(String . "`Title`" )(EndLine 0 )))))
(**define** rxreply (rx:reply-to-remark-blk rsvp-block "`Done`" ))
rxreply
=>
?⟹
(BkRemark (0 . 0 )((Remark ((Id . "`:=`" )(WhtSpc 1 5 )(Id . "`program-title`" )
(WhtSpc 1 19 )(String . "`Title`" )(EndLine 0 )))))
(write-scheme-blocks-to "`scratch-r.scm`" (*list* rxreply ))
(write-TeX-blocks-to "`scratch-r.tex`" (*list* rxreply ))
Look at the Scheme:
(file-verbatim "`scratch-r.scm`" )
→

    ___ ___ file verbatim: `scratch-r.scm` ___ ___
⟶ `;#:=␣program-title␣"Title"`
    ___ ___ end verbatim: `scratch-r.scm` ___ ___

(file-TeX "`scratch-r.tex`" )
→

    ___ ___ file TEX: `scratch-r.tex` ___ ___

⟶ := program-title "`Title`"
    ___ ___ end TEX: `scratch-r.tex` ___ ___

(**define** dbsave debug-port )
(**set!** debug-port (open-output-file "`debug-reply.txt`" ))
(debug "`--debug-test=`" "`begin␣reply␣test`" )
(rx:reply-to-remark-blk rsvp-block "`Done1`" )
(rx:reply-to-remark-blk (cadddr blocks ) "`Done2`" )
(debug "`--debug-test=`" "`end␣test`" )
(**begin** (close-port debug-port ) (**set!** debug-port #**f** ))
ts:parameters
cut
‖ ⟹((program-title . "Title") (copyright . "Copyright \\copyright\\ 2026 B.C.")‖
‖  (author . "Awe Thor") (title-foot . "thanks"))

```
(write-lines-to “scratch.scm”
    ′( “(define␣pi␣3.14)␣␣(*␣2␣pi)”
       “;#=>␣99”
       “;#->”
       “;#␣␣a␣b␣c”
       “;#␣␣d␣e␣f”
       “;#cut”
       ))
```

See the lines in the file.

```
(file-verbatim “scratch.scm” )
```

→

```
___  ___ file verbatim: scratch.scm ___  ___
(define␣pi␣3.14)␣␣(*␣2␣pi)
;#=>␣99
;#->
;#␣␣a␣b␣c
;#␣␣d␣e␣f
;#cut
___  ___ end verbatim: scratch.scm___  ___
```

Now try reading it as blocks
(**define** defpi-eval (read-blocks-from “scratch.scm” ))
Try looking at the same blocks in three different ways.
defpi-eval

As lexemes:

```
=>
?⟹
((BkData (0 . 0 )((Open )(Id . “define” )(WhtSpc 1 8 )(Id . “pi”
)(WhtSpc 1 11 )(Number . “3.14” )(Close )(WhtSpc 2 18 )(Open )(Id . “*”
)(WhtSpc 1 21 )(Number . “2” )(WhtSpc 1 23 )(Id . “pi” )(Close )(EndLine
0 )))(BkRemark (1 . 0 )((Remark ((Id . “=>” )(WhtSpc 1 5 )(
Number . “99” )(EndLine 0 )))(Remark ((Id . “->” )(EndLine 0 )(NewLine 0 )
(WhtSpc 2 4 )(Id . “a” )(WhtSpc 1 6 )(Id . “b” )(WhtSpc 1 8 )(
Id . “c” )(EndLine 0 )(NewLine 0 )(WhtSpc 2 4 )(Id . “d” )(WhtSpc 1 6
)(Id . “e” )(WhtSpc 1 8 )(Id . “f” )(EndLine 0 )))(Remark ((
Id . “cut” )(EndLine 0 ))))))(BkEof (0 . 0 )(EOF )))
```

As TEX:

```
→
⟶ (BkData (0 . 0) ((Open) (Id . define) (WhtSpc 1 8) (Id . pi) (WhtSpc 1 11) (Number . 3.14)
(Close) (WhtSpc 2 18) (Open) (Id . *) (WhtSpc 1 21) (Number . 2) (WhtSpc 1 23) (Id . pi) (Close)
(EndLine 0))) (BkRemark (1 . 0) ((Remark ((Id . =¿) (WhtSpc 1 5) (Number . 99) (EndLine 0)))
(Remark ((Id . -¿) (EndLine 0) (NewLine 0) (WhtSpc 2 4) (Id . a) (WhtSpc 1 6) (Id . b) (WhtSpc
1 8) (Id . c) (EndLine 0) (NewLine 0) (WhtSpc 2 4) (Id . d) (WhtSpc 1 6) (Id . e) (WhtSpc 1 8)
(Id . f) (EndLine 0))) (Remark ((Id . cut) (EndLine 0))))) (BkEof (0 . 0) (EOF))
```

And as cut write:

```
cut
‖ ⟹((BkData (0 . 0) ((Open) (Id . "define") (WhtSpc 1 8) (Id . "pi") (WhtSpc 1‖
‖ 11) (Number . "3.14") (Close) (WhtSpc 2 18) (Open) (Id . "*") (WhtSpc 1 21)‖
```

```
‖ (Number . "2") (WhtSpc 1 23) (Id . "pi") (Close) (EndLine 0))) (BkRemark (1‖
‖ . 0) ((Remark ((Id . "=>") (WhtSpc 1 5) (Number . "99") (EndLine 0))) (Remar‖
‖ k ((Id . "->") (EndLine 0) (NewLine 0) (WhtSpc 2 4) (Id . "a") (WhtSpc 1 6)‖
‖ (Id . "b") (WhtSpc 1 8) (Id . "c") (EndLine 0) (NewLine 0) (WhtSpc 2 4) (Id‖
‖ . "d") (WhtSpc 1 6) (Id . "e") (WhtSpc 1 8) (Id . "f") (EndLine 0))) (Remark‖
‖  ((Id . "cut") (EndLine 0))))) (BkEof (0 . 0) (EOF)))
```

Data block

(**define** dblock (*car* defpi-eval ))

dblock

=>

?⟹

(BkData (0 . 0 )((Open )(Id . “define” )(WhtSpc 1 8 )(Id . “pi” )
(WhtSpc 1 11 )(Number . “3.14” )(Close )(WhtSpc 2 18 )(Open )(Id . “*” )
(WhtSpc 1 21 )(Number . “2” )(WhtSpc 1 23 )(Id . “pi” )(Close )(EndLine 0
)))

Get the remark block

(**define** rblock (*cadr* defpi-eval ))

rblock

=>

?⟹

(BkRemark (1 . 0 )((Remark ((Id . “=>” )(WhtSpc 1 5 )(Number . “99” )
(EndLine 0 )))(Remark ((Id . “->” )(EndLine 0 )(NewLine 0 )(WhtSpc 2 4
)(Id . “a” )(WhtSpc 1 6 )(Id . “b” )(WhtSpc 1 8 )(Id . “c” )
(EndLine 0 )(NewLine 0 )(WhtSpc 2 4 )(Id . “d” )(WhtSpc 1 6 )(Id .
“e” )(WhtSpc 1 8 )(Id . “f” )(EndLine 0 )))(Remark ((Id . “cut” )
(EndLine 0 )))))

(**define** xevalue (eval-blk dblock ) )

xevalue

=>

?⟹

6.28

(**define** xreply (rx:reply-to-remark-blk rblock xevalue ))

xreply

=>

?⟹

(BkRemark (1 . 0 )((Remark ((Id . “=>” )(WhtSpc 1 5 )(Number . “99” )
(EndLine 0 )))(Resultlxs ((Number . “6.28” )))(Remark ((Id . “->” )(EndLine
0 )(NewLine 0 )(WhtSpc 2 4 )(Id . “a” )(WhtSpc 1 6 )(Id . “b” )
(WhtSpc 1 8 )(Id . “c” )(EndLine 0 )(NewLine 0 )(WhtSpc 2 4 )(Id .
“d” )(WhtSpc 1 6 )(Id . “e” )(WhtSpc 1 8 )(Id . “f” )(EndLine 0 )
))(Result-TeX . 6.28 )(Remark ((Id . “cut” )(EndLine 0 )))(Result . 6.28 )
))

get the SemiSharp remark lexemes

(**define** ss (*car* (*caddr* rblock )))

ss

cut

```
‖ ⟹(Remark ((Id . "=>") (WhtSpc 1 5) (Number . "99") (EndLine 0)))
```

and reply with some bogons

(**define** rrp (rx:reply-to-a-remark (*cadr* ss ) 874 ))

rrp
cut
‖ ⟹`(Resultlxs ((Number . "874")))`

now try reply to whole block at once
(**define** rxreply (rx:reply-to-remark-blk rblock ′(4870 )))
rxreply
cut
‖ ⟹`(BkRemark (1 . 0) ((Remark ((Id . "=>") (WhtSpc 1 5) (Number . "99") (EndLin‖`
‖ `e 0))) (Resultlxs ((Open) (Number . "4870") (Close))) (Remark ((Id . "->") (‖`
‖ `EndLine 0) (NewLine 0) (WhtSpc 2 4) (Id . "a") (WhtSpc 1 6) (Id . "b") (WhtS‖`
‖ `pc 1 8) (Id . "c") (EndLine 0) (NewLine 0) (WhtSpc 2 4) (Id . "d") (WhtSpc 1‖`
‖ `6) (Id . "e") (WhtSpc 1 8) (Id . "f") (EndLine 0))) (Result-TeX 4870) (Rema‖`
‖ `rk ((Id . "cut") (EndLine 0))) (Result 4870)))`

Write those blocks to both Scheme and TEX files
(write-scheme-blocks-to "`scratch-r.scm`" (*list* rxreply ))
(write-TeX-blocks-to "`scratch-r.tex`" (*list* rxreply ))
Look at the Scheme:
(file-verbatim "`scratch-r.scm`" )
→

___ ___ file verbatim: `scratch-r.scm` ___ ___

```
;#=>␣99
;#=>!(4870)->
;#␣␣a␣b␣c
;#␣␣d␣e␣f
;#->␣(4870)cut
;#
```

___ ___ end verbatim: `scratch-r.scm` ___ ___

2025-11-07: That worked pretty well given that I have not yet written the procedures to keep and check the old suggested answer.

Look at the LATEX:
(file-TeX "`scratch-r.tex`" )
→

___ ___ file TEX: `scratch-r.tex` ___ ___

=> 99
?⟹
(4870 )
→
⟶ a b c
d e f
⟶ 4870
cut
‖ ⟹`(4870)`

___ ___ end TEX: `scratch-r.tex` ___ ___

What if we just read, not as blocks?
(**define** pt (open-input-file "`scratch.scm`" ))
(**define** rd (*list* (*read* pt ) "␣`and`␣" (*read* pt )))

(close-port pt )
rd
cut
‖ ⟹`((define pi 3.14) " and " (* 2 pi))`

—2025-11-07: The following does not work. —2025-11-08: Fixed. It was meant to be a test of dotted list, but was crashing when printing a list with a procedure in it.
(**define** (f . a ) (*list* f a ))
(f 2 3 5 7 11 )
=>
?⟹
(“#<`procedure␣f␣rest`>” (2 3 5 7 11 ))
So let's try it without that.
(**define** (f . ls ) (*cons* ′f ls ))
(f 2 3 5 7 11 )
=> 99
?⟹
(f 2 3 5 7 11 )
(**define** (dpc rot . p ) (*list* rot p ))
(dpc 66 1 2 3 )
=>
?⟹
(66 (1 2 3 ))
(**define** vbs (read-blocks-from-lines “`(set!␣v␣#(3␣6␣9))`” ))
vbs
=>
?⟹
((BkData (0 . 0 )((Open )(Id . “`set!`” )(WhtSpc 1 6 )(Id . “v”
)(WhtSpc 1 8 )(VOpen )(Number . “3” )(WhtSpc 1 12 )(Number . “6” )(WhtSpc
1 14 )(Number . “9” )(Close )(Close )(EndLine 0 )))(BkEof (1 . 0
)(EOF )))
(**define** vbd (*car* vbs ))
vbd
=>
?⟹
(BkData (0 . 0 )((Open )(Id . “`set!`” )(WhtSpc 1 6 )(Id . “v” )
(WhtSpc 1 8 )(VOpen )(Number . “3” )(WhtSpc 1 12 )(Number . “6” )(WhtSpc 1
14 )(Number . “9” )(Close )(Close )(EndLine 0 )))
(**define** lvbd (:lexdata←blk vbd ))
lvbd
=>
?⟹
(((Id . “`set!`” )(Id . “v” )(LxVector (Number . “3” )(Number . “6” )(Number
. “9” ))))

```
(define (ncube-verts dim )
  (if (zero? dim )
      '(())
      (let ( (face (ncube-verts (- dim 1 )))
             (cons-n-all (lambda (n pts ) (map (lambda (pt ) (cons n pt ))
                                               pts ) ) ) )
        (append (cons-n-all  -1  face )(cons-n-all  1  face )) )))
```

Now make a vector of coordinates of the vertices of a three dimensional cube.
(**define** cube-verts (*apply vector* (ncube-verts 3 )))
cube-verts
cut
```
|| ⟹#((-1 -1 -1) (-1 -1 1) (-1 1 -1) (-1 1 1) (1 -1 -1) (1 -1 1) (1 1 -1) (1 1 1||
|| ))
```

=>
?⟹
```
#((-1 -1 -1) (-1 -1 1) (-1 1 -1) (-1 1 1) (1 -1 -1) (1 -1 1) (1 1 -1) (1 1 1||
|| ))
```

(*list* "xxx" cube-verts "yyy" ( * 6 7 ) (sqrt 2 ) )
cut
```
|| ⟹("xxx" #((-1 -1 -1) (-1 -1 1) (-1 1 -1) (-1 1 1) (1 -1 -1) (1 -1 1) (1 1 -1)||
||  (1 1 1)) "yyy" 42 1.4142135623730951)
```

=>
?⟹
```
("xxx" #((-1 -1 -1) (-1 -1 1) (-1 1 -1) (-1 1 1) (1 -1 -1) (1 -1 1) (1 1 -1) (1 1 1||
|| ))
"yyy" 42 1.4142135623730951 )
```

End of file `tsteval-riA.tex`

# 6 Main Procedures

Procedure *ts:layout* was the main program of version 2.2. It was replaced by *ts:make-reply*.

The first argument to *ts:make-reply* is a string which is the (stem) name of a Scheme file. The filename is ⟨stem⟩`.scm`. The reply comes in the form of several files named ⟨stem⟩`-r.scm`, ⟨stem⟩`-r.tex`.

If all goes well, the ⟨stem⟩`-r.tex` file can be processed with LaTeX to get a typeset document and the ⟨stem⟩`-r.scm` file can replace the ⟨stem⟩`.scm` file. Those two files contain the same ⟨data⟩ but the reply has updated remarks.

As a special case, if the given first argument ends with `-t`, that part will be ignored; it will not be part of the ⟨stem⟩. The file ⟨stem⟩`-t.scm` will be read but treated as though it were ⟨stem⟩`.scm`. For example, the reply to `texscm-t.scm` is written to `texscm-r.scm` and `texscm-r.tex`.

This is so that if compilation of the Scheme file is expensive it can be copied into a ⟨stem⟩`-t.scm` file, where the comments and remarks can be changed and *ts:make-reply* re-run many times without triggering a re-make with re-compilation.

This procedure also reads ⟨datum⟩s from ⟨stem⟩`.scm` using the Scheme `(read)` procedure, which reads only the data, ignoring atmosphere. The file is not written and the modification time does not change, but if the data (ignoring atmosphere) differs from the data in `texscm-r.scm`, a warning will be shown. If this is not a mistake, then the ⟨stem⟩`-r.scm` file should be copied to ⟨stem⟩`.scm` so that the new code will be used.

Blocks are read and processed one at a time. There is no need to keep a list of all of them. We do need to keep two output ports open at the same time, writing to them alternately.

## 6.1 Reply to remarks

This exceptionally cheesy code should be fixed as soon as possible. It relies upon white space in exactly the right place. — A feature!

The xx: procedures are subversion C. They do not exist in texscm3, so the Makefile must ensure that tsteval, which tests the xx: procedures, must not be input to texscm3.

```
(define xx:make-reply #f )
(define xx:reply-to-a-remark #f )
(define xx:reply-to-remark-blk #f )
(define ts:reply-to-a-remark #f )
(define ts:reply-to-remark-blk #f )
(define rx:reply-to-a-remark #f )
(define rx:reply-to-remark-blk #f )
(define ts:make-reply #f )
(define ts:make-replyX #f )
(define ts:reply-to-remark #f )
```

The **let** is intended to prevent an *eval* from overwriting the procedures while they are in use

```
(let ( (:sort-in-datum! :sort-in-datum! )
       (:block-lxmdata←lxmlist! :block-lxmdata←lxmlist! )
       (:primal-environ    :primal-environ )
       (ts:read-block ts:read-block ) )
  (define (:lexlist←datum dat )
    (cond ( (number? dat )
              (list (cons 'Number (number→string dat ))) )
          ( (symbol? dat )
              (list (cons 'Sy (symbol→string dat )) ) )
          ( (boolean? dat ) (cons 'Boolean (if dat "#t" "#f" )) )
          ( (string? dat )
              (list (cons 'String dat )) )
          ( (vector? dat )
              (list (cons 'LxVector dat )) )
```

```
        ( (pair? dat )
          (if (list? (cdr dat ))
              (apply append (list '((Open )) (:lexlist←datum (car dat ))
                                  (apply append (map :lexlist←datum (cdr dat ))) '((Close ))))
              (append '((Open )) (:lexlist←datum (car dat ))
                      '((Dot ))   (:lexlist←datum (cdr dat )) '((Close ))) ))
        ( (null? dat ) '() )
        (else
         (let* ( (ost (open-output-string ))
                 (str (begin (write dat ost ) (get-output-string ost ))) )
           (list (cons 'String str ))))))
```

#| Given a block of remarks reply with a block of remarks which should replace it. This old version assumes one remark per remark block. If it is a question replace it with a reply. |#

```
(define (:reply-to-remark-blk rsvp-block evalue )
  (let ((ss (car (caddr rsvp-block ))))
    (if (eq? (lxm-type ss ) 'SemiSharp )
        (list 'BkRemark '(0 . 0 )
              (:reply-to-a-remark (cadr ss ) evalue ))
        (if (eq? (lxm-type ss ) 'Remark )
            (list 'BkRemark '(0 . 0 )
                  (:reply-to-a-remark (cadr ss ) evalue ))
            (list 'Error "blk␣SemiSharp" ss )))))
```

```
(define (:reply-to-a-remark rsvp-rem evalue )
    (define (:set-diavar! sym val )
      (let ((p (assoc sym ts:parameters )))
        (if p (begin (set-cdr! p val ) p ) (list 'Error "bad␣parameter" sym )) ))
    (let ( (lxs rsvp-rem ))
      (cond
       ( (equal? (car lxs ) '(Id . "=>" ))
         (begin (list (list 'Resultlxs (:lexlist←datum evalue ))) ))
       ( (equal? (car lxs ) '(Id . "->" ))
         (begin    (list (cons 'Result-TeX evalue ))) )
       ( (equal? (car lxs ) '(Id . "cut" ))
         (begin (list (cons 'Result evalue ))))
       ( (equal? (car lxs ) '(Id . ":=" ))
         (begin (:set-diavar! (string→symbol (cdr (caddr lxs )))
                              (cdr (caddr (cddr    lxs )))))
           (cons 'Remark lxs ) )
       (else (list 'Result "remark␣error" lxs )) )))
```

#| The new version must process all remarks in a block. Do NewLine come within or between SemiSharp? |#

```
(define (rr:reply-to-remark-blk rsvp-block evalue )
  (let qa ( (out '() )
            (in (caddr rsvp-block )) )
    (if (null? in )
        (list 'BkRemark (cadr rsvp-block ) (reverse out ))
        (let ((blk (car in )))
          (if (eq? (lxm-type blk ) 'SemiSharp )
              (let ( (rply (rr:reply-to-a-remark (cadr blk ) evalue )) )
                (qa (if rply
                        (cons rply (cons blk out ))
```

```
                    (cons blk out ))
                (cdr in ) ))
          (if (eq? (lxm-type blk ) 'Remark )
              (let ( (rply (rr:reply-to-a-remark (cadr blk ) evalue )) )
                (qa (if rply
                        (cons rply (cons blk out ))
                        (cons blk out ))
                    (cdr in ) ))
              (list 'Error "blk␣SemiSharp" ) ))))))


(define (rr:reply-to-a-remark rsvp-rem evalue )
    (define (:set-diavar! sym val )
      (debug "set␣sym=" sym ",␣val=" val )
      (let ((p (assoc sym ts:parameters )))
        (if p (begin (set-cdr! p val ) p ) (list 'Error "bad␣parameter" sym )) ))
    (let ( (lxs rsvp-rem ))
      (debug "reply-to" lxs )
      (cond
       ( (equal? (car lxs ) '(Id . "=>" ))
         (begin (list 'Resultlxs (:lexlist←datum evalue )) ))
       ( (equal? (car lxs ) '(Id . "->" ))
         (begin    (cons 'Result-TeX evalue )) )
       ( (equal? (car lxs ) '(Id . "cut" ))
         (begin (cons 'Result evalue )))
       ( (equal? (car lxs ) '(Id . ":=" ))
         (begin (:set-diavar! (string→symbol (cdr (caddr lxs )))
                              (cdr (caddr (cddr    lxs )))))
           #f )
       (else (list 'Error "remark␣error" lxs )) )))
  (define (:make-reply . args )
    (debug "make-reply(" args )
    (let ( (fstem (if (null? args ) "scratchx" (car args )))
           (quiet (not (member "verbose" args )))
           (eval-it (and (not (null? args ))(not (member "noeval" args ))))
           (force-eval (member "eval" args ))
           (tabbing (not (member "notabs" args )))
           (debugging (member "debug" args ))
           )
      (let ( (Scheme-port (open-output-file
                           (string-append fstem "-r" ts:subversion ".scm" )))
             (TeX-port (open-output-file
                        (string-append fstem "-ri" ts:subversion ".tex" ))) )
        (set! ts:input-file-name (string-append fstem ".scm" ))
        (set! eval-it #f ) #| turn it off 'til it works |#
        (debug "infile=␣" ts:input-file-name )
        (set! TeXscm-mode 'outTeXMode )
        (if debugging (set! debug-port (open-output-file "debug.txt" )))
        (with-input-from-file ts:input-file-name
          (lambda ()
            (let ( (done #f )
                   (evalue "none" )
                   (block-to-write #f ) )
              (do ((block (ts:read-block )(ts:read-block )))
```

```
                    (done )
                  (case (car block )
                    ( (BkData )
                      (:sort-in-datum! (:block-lxmdata←lxmlist! block )
                                       :primal-environ )
                      (set! block-to-write block )
                      (if (or eval-it force-eval )
                          (set! evalue (eval-blk block ) )))
                    ( (BkComment0 BkComment3 )
                      (set! block-to-write block ) )
                    ( (BkkkRemark )
                      (let ((reply
                              (if (or eval-it force-eval )
                                  (rr:reply-to-remark-blk block evalue )
                                  block )))
                        (set! block-to-write reply )))
                    ( (BkRemark )
                      (let ((reply (rr:reply-to-remark-blk
                                    block
                                    (if (or eval-it force-eval )
                                        evalue
                                        "None" ))))
                        (set! block-to-write reply )))
                    ( (BkEof ) (set! done #t ))
                    (else (list 'Error    "bad␣block" block )))
                  #| it not always one block to write? |#
                  (if block-to-write
                      (begin
                        (parameterize ((current-output-port Scheme-port ))
                          (ts:write-scheme-blk block-to-write ) )
                        (parameterize ((current-output-port TeX-port ))
                          (ts:write-TeX-blk block-to-write ))))
                  (set! block-to-write #f )))))
          (let (( TeXdoc-port
                  (open-output-file
                    (string-append fstem "-r" ts:subversion ".tex" ))) )
            (parameterize ((current-output-port TeXdoc-port ))
              (set! TeXscm-mode 'outTeXMode )
              (write-TeX-document-pre )
              (display "\\input{" )
              (display (string-append fstem "-ri" ts:subversion ".tex" ))
              (display "}" ) (newline )
              (write-TeX-document-post ) )
            (close-port TeXdoc-port )      )
          (close-port Scheme-port )
          (close-port TeX-port )       )))
(define (:make-replyX subv . args )
  (set! ts:subversion subv )
  (apply :make-reply args ))
(set! xx:reply-to-remark-blk rr:reply-to-remark-blk )
(set! xx:reply-to-a-remark rr:reply-to-a-remark )
(set! rx:reply-to-remark-blk rr:reply-to-remark-blk )
(set! rx:reply-to-a-remark rr:reply-to-a-remark )
(set! xx:make-reply :make-reply )
```

```
(set! ts:make-replyX :make-replyX )
(set! ts:reply-to-remark-blk :reply-to-remark-blk )
(set! ts:reply-to-a-remark :reply-to-a-remark )
(set! ts:make-reply :make-reply )
)
```

Included File `demo-ri.tex`

# 7 Appendix A: Some Tests and Demonstrations

## 7.1 Kino

This is a simple program I wrote many years ago (1995?). It provides a demonstration of "bignums". It computes the probability of winning a simple Kino-like game. The odds are so bad that hardware arithmetic fails.

```
(define (show a ) a )
```

Procedure *choose* computes binomial coefficients (*choose* $n$ $k$) = $\binom{n}{k}$ — from $n$ choose $k$.

```
(define (choose n k )
  (let loop ((p 1 )(num n )(den 1 ))
    (if (> den k )
        p
        (loop (/ (* p num ) den ) (- num 1 ) (+ den 1 )))))
(choose 7 2 )
=>
?⟹
21
(choose 10 5 )
=>
?⟹
252
(choose 100 50 )
=>
?⟹
100891344545564193334812497256
```

From an urn of $b$ balls, $r$ of which are red, take $t$ of them, what is the probability of getting exactly $g$ red ones? The number $g$ is good; it wins the game. The formula is $\binom{r}{g}\binom{b-r}{t-g}\Big/\binom{b}{t}$.

```
(define (kino balls red take good )
  (/ (* (choose red good )
        (choose (- balls red ) (- take good )))
     (choose balls take )))
(kino 100 2 2 2 )
=>
?⟹
1/4950
(kino 100 10 5 2 )
=>
?⟹
1335/19012
(kino 100 10 10 10 )
=>
```

?⟹
1/17310309456440

## 7.2 Mathematical Typesetting

In addition to showing the result as a ⟨datum⟩ it can be shown as TEX. The result must be a string or list of strings, which are displayed one per line and inserted at that point in the document. These lines are processed by LATEX to produce the printed output.

```
(show '("{\\it␣Hello},␣world!␣\\copyright\\␣"
        "%␣This␣is␣a␣one␣line␣TeX␣comment,␣not␣seen␣in␣output."
         "---␣or␣in␣German:␣Gru\\ss\\␣Gott,␣Welt!" ))
```

→
⟶ *Hello*, world! © — or in German: Gruß Gott, Welt!
It seems to break if the expression starts with a quote.

Obviously math[3] works.

```
(define sqrt-pi
  "$\\sqrt{\\pi}␣=␣\\int_{-\\infty}^{+\\infty}␣e^{-x^2}␣\\mathrm{d}x$" )
(show '("Everyone␣should␣know␣that␣" ,sqrt-pi ))
```

→
⟶ Everyone should know that $\sqrt{\pi} = \int_{-\infty}^{+\infty} e^{-x^2}\mathrm{d}x$ ... in fact, it is worth displaying larger:

```
(string-append "$" sqrt-pi "$" )
```

→
⟶

$$\sqrt{\pi} = \int_{-\infty}^{+\infty} e^{-x^2}\mathrm{d}x$$

In case you don't know what $\pi$ is, just remember[4]:

```
(list
  "%␣\\pi␣day␣greeting␣from␣John␣Luciani"
  "$$\\pi␣=␣2␣\\cdot␣\\cfrac{2}{\\sqrt{2}}␣\\cdot"
  "␣\\cfrac{2}{\\sqrt{2␣+␣\\sqrt{2}}}"
  "\\cdot␣\\cfrac{2}{\\sqrt{2␣+␣\\sqrt{2␣+␣\\sqrt{2}}}}␣\\cdots$$" )
```

→
⟶

$$\pi = 2 \cdot \frac{2}{\sqrt{2}} \cdot \frac{2}{\sqrt{2+\sqrt{2}}} \cdot \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \cdots$$

Now we use that formula to compute some approximations of $\pi$.

```
(define (root2plus n ) (if (= n 0 ) 0 (sqrt (+ 2 (root2plus (- n 1 ))))))
(define (factors n )
  (if (= n 0 ) 1 (* (/ 2 (root2plus n )) (factors (- n 1 )))))
(define (pi n )( * 2 (factors n )))
(pi 2 )
=>
?⟹
3.0614674589207183
(pi 12 )
```

[3] Exactly this formula is: [20, p.74] more generally, Formula 492 of [10]

[4] This is called Viète's formula, even though "Fransisci Vietæ" is written on the cover of his book [21]. See `https://en.wikipedia.org/wiki/Vi%C3%A8te%27s_formula` .

=>
?⟹
3.141592576584873
(pi 22 )
=>
?⟹
3.1415926535897225

### 7.3 Associative Law

This a Scheme demonstration of the associative law for an "unbiased" monoidal category according to Leinster [1, Def.3.1.1,p.68].

```
(define lss '(((a1 a2 a3 )(b1 b2 )(c ))
                ((d1 d2 d3 d4 )(e1 e2 )(f1 )()(h1 h2 h3 h4 h5 ))
                ((j1 j2 )(k1 k2 k3 k4 )))   )
(define (app ls ) (apply append ls ))
(app lss )
=>
?⟹
((a1 a2 a3 )(b1 b2 )(c )(d1 d2 d3 d4 )(e1 e2 )(f1 )
(h1 h2 h3 h4 h5 )(j1 j2 )(k1 k2 k3 k4 ))
(map app lss )
=>
?⟹
((a1 a2 a3 b1 b2 c )(d1 d2 d3 d4 e1 e2 f1 h1 h2 h3 h4 h5 )(j1 j2
k1 k2 k3 k4 ))
(app (app lss ))
=>
?⟹
(a1 a2 a3 b1 b2 c d1 d2 d3 d4 e1 e2 f1 h1 h2 h3 h4 h5 j1 j2 k1 k2 k3 k4 )

(app (map app lss ))
=>
?⟹
(a1 a2 a3 b1 b2 c d1 d2 d3 d4 e1 e2 f1 h1 h2 h3 h4 h5 j1 j2 k1 k2 k3 k4 )
```

## References


[1] Tom Leinster, *Higher Operads, Higher Categories*, London Mathematical Society Lecture Notes, Cambridge University Press (2003)

[2] Fransisci Vietæ, *Variorum de rebus mathematicis responsorum* (1593)


### 7.4 Failing Tests

Bug in tab ticks. The tab-tick line runs over the **if** in the last line. The close parenthesis make a big difference. If it is on the previous line, then the extra tick goes away but the close paren and comment are too far right. Two more spaces in front of it make it all look good

```
(define st:read-block #f )
```

```
(let ( (char-types 5 )
       )   #| This makes a big difference! |#
  (define read-block #t )
  (define (readlock )
    (let* ( (w 0 ) )
      #f ))
  (if #t (set! st:read-block read-block )   )   #| BugTbTk |#
)
```

This a test of three kinds of quotes and two unquotes: This is a test of big spaces. They don't quite line up.

```
(define lexeme-types
   '(Id      Result    Result-TeX    Number    NewLine       TabSpc    Abbrev Semicolon
     Char String    Boolean         Open      VOpen         Close     Dot ))
```

This uses tabs (#\tab=#\x09) in the input.

```
(define (xp f ls )
              (let ((rs '())
                    (ls ls ))
              #f ))
```

This is broken:

```
(define (dont fix )
  (do ( (x x0 xn ) )
      ( done
        stop )
    (begin
      (next )
      (if (not broke )
          (dont fix )))) )
(define charnames
  '(("nul" . #\nul ) ("newline" . #\newline )
    ("space" . #\space ) ("tab" . #\tab )))
(define test `("two plus two is" ,@(list "(" '+ '2 '2 ") =" ) ,(+ 2 2 )))
```

here are some quotations.

```
(quote (a b c (e f b ) h ))
'(a b c (e f b ) h )
(quote b )
'b
```

Here is a double quoted symbol. Note that the first quote is a keyword, the second is a symbol.

```
(quote (quote b ))
''b
```

This is a test of comments They used to mess up identifier sorting.

```
(define (factorial n )
  (if (= n 0 )
      #|then|# 1
      #|else|# (* n (factorial (- n 1 )))))
```

Here we define "define" to be a variable.

```
(define (factorial define )
  (if (= define 0 )
      1
      (* define (factorial (- define 1 )))))
```

these are comment symbols #|test|# do they work?

```
(define test `("two plus two is" ,@(list "(" '+ '2 '2 ") =" ) ,(+ 2 2 )))
```

test

The next two expressions show the difference between **let** and **let*** The first interprets the initialization expressions in the ambiant environment, while **let*** interprets each initialzer in an environment in which the previous binding are in effect.

```
(let ( (if (if #t 3 5 ))
          (let (if #t 4 8 )))
    (+ if let ))
(let* ( (if (if #t 3 5 ))
           (let* (* if if )))
    (+ if let* ))
```

This is a test of indentation

```
(cons (if #t
              (let ((a  '(x y ))
                       (b  '(w z ))) (cons a b ))
              #t )
         (+ 1 ))
(car    (if #t
              (let ((a  '(x y ))
                       (b  '(w z ))) (cons a b ))
              #t ))
(+ 1 )
(let* ((if (if #t 3 5 ))
           (let* (* if if )))
    (+ if let* ))
```

This used to produce a backward tab. Now it loses a single space at the begining of the line.

```
(let* ((if (if #t 3 5 ))
           (let* (* if if )))
    (+ if let* ))
```

So did this. That's fixed but the final paren is too far right

```
(let* ((if (if #t 3 5 ))
           (let* (* if if )))
     (+ if let* ))
(define alphabet  '(a b c d e f g
                        h i j k l m ))
```

The close paren so far right because it is preceded by two spaces.

```
(define (repel ) #f    )
```

one looks like this:

```
(define (repel ) #f )
```

The following does not work and is commented out comments. Bug3: The named let* is invalid code, but it should not cause a crash.

Bug4 Without an extra blank line after this and before the end of file, we get a crash, because the end{verbatim} does not get copied to the *.tex file.

```
 (define crash
  (let* name ((a 5))
       (name 6) ))
```

End of file `demo-ri.tex`

# 8 Appendix B: Technical Details

## 8.1 Scheme Source Code Syntax

### Character Sets

Ascii

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0: | NUL | SOH | STX | ETX | EOT | ENQ | ACK | \a | \b | \t | \n | \v | \f | \r | SO | SI |
| 1: | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2: | | ! | " | # | $ | % | & | ’ | ( | ) | * | + | , | - | . | / |
| 3: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4: | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5: | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6: | ‘ | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7: | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |

The character #\x20 is a blank space, printed as "␣" when it is important to see and count blanks. An unprintable character is #\x7F or in the range #\x00–#\x1F with the exception of #\x09(=#\tab≃"\t"). It is a lexical error for an unprintable character to appear anywhere at all in a purported Scheme program unless it is part of a ⟨line ending⟩, which will be interpreted by the underlying system, and never seen by a program that uses *read-line*, as TEX←Scm does.

We need to handle tabs #\tab = #\x09 as worthy whitespace, because editors will put them in. A tab is equivalent to a sequence of blanks. A tab character in the input may be changed to a sequence of blanks in the reconstruction of Scheme from a lexeme list. If that happens "`diff -w file0.scm file1.scm` might show them the same when "`diff`" without "`-w`" would list a great many invisible differences.

| | RFC-3629 STD-63 | |
|---|---|---|
| | Unicode | UTF-8 |
| 7 bits | 0000,0000 – 0000,007F | 0xxx,xxxx |
| 11 bits | 0000,0080 – 0000,07FF | 110x,xxxx 10xx,xxxx |
| 16 bits | 0000,0800 – 0000,FFFF | 1110,xxxx 10xx,xxxx[2] |
| 21 bits | 0001,0000 – 0010,FFFF | 1111,0xxx 10xx,xxxx[3] |

Seven bits suffice to encode Ascii; UTF-8 uses exactly the same codes. Eleven bits (2048 codes) suffice to encode Latin-based characters (including non-Ascii, such as ¿, ß, é, ï, and Æ), as well as Greek, Cyrillic, Hebrew, Arabic and more. Sixteen bits ($2^{16}$ codes) suffice for the Basic Multilingual Plane, which also covers Chinese, Japanese, Korean, and more. Twenty bits suffice to encode sixteen higher planes, that is, numbers 0001,0000 – 0010,FFFF. In UTF-16, such scalar values are encoded by surrogate pairs with ten bits encoded in each surrogate. In UTF-8, twenty-one bits cover (more than) all seventeen planes (0–16). See https://en.wikipedia.org/wiki/Plane_(Unicode).

A worthy character is either printable Ascii or higher Unicode. In terms of UTF-8, higher Unicode just means any character that contains octets in the range 80–FF. Since this program reads and writes whole lines, and a line contains only whole characters, we don't need to worry much about them. In comments, character stings, and bar-enclosed identifiers, just copy them; in other ⟨lexeme⟩s, they are illegal.

An unworthy character is #\x7F; or in the range #\x00;–#\x1F; (i.e. it is Ascii but not printable). The characters #\n and #\r are unworthy even though the Scheme Report [17, §7.1.1, p.62] says they can be part of a ⟨line ending⟩. The underlying Scheme and Operating System should handle line endings, we don't care how. This program should never see either of them in the Scheme it reads. Neither should it see an Ascii DC3 — but escapes like "one\ntwo" and #\x13 are valid and do not contain any unworthy characters, just their names.

There are a semi-exceptions. An input tab is just white space, a compiler could treat it as a space, but it may affect indentation. A newline will never be read, but it may be inserted by the program itself to mark the end of line. This works *because* it can never be read. A carriage return

may be used in a TEX "\verb" command because it will never be printed either.

We might need to count characters in #\Exxx. Or insist that single character constants must be followed by a delimiter. See R7RS § 6.6 "If ⟨character⟩ is alphabetic then any character immediately following can not be one that can appear in an identifier". So is #\E alphabetic? I suppose so, but I don't want to write code to decide. Fail safe is assume any Unicode (in this context) is alphabetic. What is the difference between "can't appear in an identifier" and "is a delimiter"?

Everyone [R5RS, §2.1] [R6RS, §4.2.1] [R7RS, §2.1] agrees that the 18 extended identifier (or alphabetic) characters are ! $ % & * + - . / : < = > ? @ ^ _ ~ but what are they for exactly?

According to R7RS, as re-revised[5]

⟨special initial-7⟩ ::= ! | $ | % | & | * | / | : | < | = | > | ? | @ | ^ | _ | ~

So ⟨extended id⟩ ::= ⟨special initial-7⟩ | + | - | .

The worthy characters can be divided into five sets

⟨worthy character⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨extended id⟩ | ⟨whitespace⟩ | ⟨unicode⟩ | ⟨other⟩

Where ⟨other⟩ is whatever remains. The 14 other characters are (in Ascii order)

⟨other⟩ ::= " | # | ' | ( | ) | , | ; | [ | \ | ] | ` | { | | | }

There are 15 characters mentioned in R7RS §2.3: *Other Notations*. They are . + - ( ) ' ` , " \ [ ] { } # (in order of mention) while | and ; are described in the preceding part of §2.

Altogether the 17 §2 characters are ⟨§2⟩ ::= ⟨other⟩ | + | - | .

**Lexical Syntax**

The lexical syntax(es) of Scheme, as given in the fifth, sixth, and seventh Repeatedly Revised Reports ( R5RS), §7.1, (R6RS), §4.2, and (R7RS), §7.1, are shown below by putting a hyphen and a revision number after the defining occurrence of the name of a non-terminal in angle brackets. Uses of non-terminals may be interpreted by appending your choice of number. Defining occurrences of syntactic variables without numbers give the intended grammar of the input to be processed by this program (TEX←Scm).

R6 has used the word "lexeme" for what R5 and R7 call "token". I would like to use "token" in its original and latest sense, while using "lexeme" for any sequence of characters that are a significant unit to this program (TEX←Scm) thus including whitespace, comments, and remarks.

⟨token-5⟩ ::= ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩ | ⟨character⟩ | ⟨string⟩ | ( | ) | #( | ' | ` | , | ,@ | .
⟨lexeme-6⟩ ::= ⟨token-5⟩ | #vu8( | #' | #` | #, | #,@ | [ | ]
⟨token-7⟩ ::= ⟨token-5⟩ | #u8(
⟨token⟩ ::= ⟨token-5⟩ | #⟨identifier⟩( | #' | #` | #, | #,@
⟨lexeme⟩ ::= ⟨token⟩ | ⟨atmosphere⟩
⟨delimiter-5⟩ ::= ⟨whitespace⟩ | ( | ) | " | ;
⟨delimiter-6⟩ ::= ⟨delimeter-5⟩ | [ | ] | #
⟨delimiter-7⟩ ::= ⟨delimeter-5⟩ | ⟨vertical line⟩

It seems that # is not a delimiter in R7RS. I think it *is* a delimiter in Guile, because:

(guile)> '(#T#F#T#T#F#F) ⇒ (#t #f #t #t #f #f)

But this is strange: (guile)> '(X#T Y#F#T Z#F) ⇒ (#X#T# #Y#F#T# #Z#F#)

The Guile Reference Manual, Edition 2.0.11, revision 1, §6.6.7.6 says that #{foo bar}# and #{what ever}# are symbols. No thank you! I use ⟨vertical bar⟩s and no unworthy characters in an identifier. (The second example has a #\newline character and maybe a #\return in it.)

Some Ascii characters are neither ⟨subsequent⟩ nor ⟨delimiter⟩, but just odd characters. These should not occur just before the end of file.

⟨odd character⟩ ::= ⟨unworthy character⟩ | ' | ` | ,

Can we just say ⟨delimited word⟩::=⟨worthy non-delimiter⟩*⟨delimiter⟩ and a ⟨delimited word⟩ without the ⟨delimiter⟩ might be a token? ⟨maybe token⟩::=⟨delimited word⟩/⟨delimiter⟩ and then decide

[5] See R7RSSmallErrata: "In Section 7.1.1, the lexical rule ⟨special initial⟩ incorrectly omits @."

⟨token⟩ ::= ⟨maybe token⟩ & ( ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩ | ⟨character⟩ | ⟨string⟩
| `(` | `.` | `)` | ⟨vector open⟩ | ⟨abbreviation⟩)
⟨vector open⟩ ::= `#u8(` | `#vu8(` | `#(` | . . . `#`⟨uv type⟩`(`
⟨abbreviation⟩::=| `'` | `` ` `` | `,` | `,@` | `#'` | `` #` `` | `#,` | `#,@`
But what about `[`, `]`, {, and }? Are these unworthy? [R$^7$RS, §2.2] says they are reserved for future extensions. [R$^6$RS, §421] says `[` and `]` are lexemes and delimiters, while { and } are reserved

⟨whitespace-5⟩ ::= ⟨space or newline⟩
⟨whitespace-6⟩ ::= ⟨character tabulation⟩ | ⟨line feed⟩ | ⟨line tabulation⟩ | ⟨form feed⟩
| ⟨carriage return⟩ | ⟨next line⟩ | ⟨unicode separator character⟩
⟨whitespace-7⟩ ::= ⟨intraline whitespace-7⟩ | ⟨line ending-7⟩
⟨intraline whitespace-7⟩ ::= ⟨space or tab⟩
⟨line ending-6⟩ ::= ⟨linefeed⟩ | ⟨carriage return⟩ | ⟨carriage return⟩ ⟨linefeed⟩ | ⟨next line⟩
| ⟨carriage return⟩ ⟨next line⟩ | ⟨line separator⟩
⟨line ending-7⟩ ::= ⟨newline⟩ | ⟨return⟩ | ⟨return⟩ ⟨newline⟩

There is no ⟨line ending-5⟩, presumably it is left to the operating system to break text into lines. What is the difference between ⟨linefeed⟩, ⟨next line⟩, and ⟨newline⟩? — I do not know.
("line feed" is an alternate name for U+21B4, RIGHT ARROW WITH CORNER DOWNWARD.)

⟨comment-5⟩ ::= `;` ⟨all subsequent characters up to a line break⟩
⟨comment-6⟩ ::= `;` ⟨all subsequent characters up to a ⟨line ending⟩ or ⟨paragraph separator⟩⟩
| ⟨nested comment⟩ | `#;` ⟨interlexeme space⟩ ⟨datum⟩ | `#!r6rs`
⟨comment-7⟩ ::= `;` ⟨all subsequent characters up to a line ending⟩
| ⟨nested comment⟩ | `#;` ⟨intertoken space⟩ ⟨datum⟩
⟨nested comment⟩ ::= `#|` ⟨comment text⟩ (⟨nested comment⟩ ⟨comment text⟩)* `|#`
⟨comment text⟩ ::= ⟨character sequence not containing `#|` or `|#` ⟩
⟨directive-7⟩ ::= `#!fold-case` | `#!no-fold-case`

There is no *use* of the non-terminal ⟨directive⟩. We want
⟨datum⟩ ::= ⟨token⟩ | ( ⟨lexeme⟩ * )
⟨lexeme⟩ ::= (⟨token⟩ | ⟨atmosphere⟩ | ⟨directive⟩)*
⟨atmosphere-6⟩ ::= ⟨white space⟩ | ⟨comment⟩
⟨atmosphere-7⟩ ::= ⟨white space⟩ | ⟨comment⟩ | ⟨directive⟩
⟨interlexeme space⟩ ::= ⟨atmosphere⟩*
⟨identifier-56⟩ ::= ⟨initial⟩ ⟨subsequent⟩* | ⟨peculiar identifier⟩
⟨identifier-7⟩ ::= ⟨initial⟩ ⟨subsequent⟩* | ⟨peculiar identifier⟩
| ⟨vertical line⟩ ⟨symbol element⟩* ⟨vertical line⟩
⟨initial-57⟩ ::= ⟨letter⟩ | ⟨special initial⟩
⟨initial-6⟩ ::= ⟨constituent⟩ | ⟨special initial⟩ | ⟨inline hex escape⟩
⟨letter⟩ ::= `a` | `b` | `c` | ... | `z` | `A` | `B` | `C` | ... | `Z`
⟨constituent-6⟩ ::= ⟨letter⟩ | ⟨Unicode letter⟩

According to R$^6$RS, §4/2/1, a ⟨Unicode letter⟩ is any character whose Unicode scalar value is greater than 127, and whose Unicode general category is: Lu, Ll, Lt, Lm, Lo, Mn, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co. I can't cope with that.

There is no ⟨constituent-7⟩. Instead the formal syntax provides only for Ascii. R$^7$RS says Scheme implementations *may* permit the use of Unicode characters in identifiers, provided that the character general category is one of the above or Mc, Me, Nd, or is U+200C or U+200D, that is a ⟨Unicode subsequent⟩ or ZERO WIDTH (NON-)JOINER.

It will be left to the underlying system to decide whether to reject a specific character in an identifier. This program will be as liberal as possible.

The intricate grammar is contradicted by exceptions: §7.1.1 "+i, -i, and ⟨infnan⟩ are numbers"; §2.1 "An identifier is any sequence . . . that does not have a prefix which is a valid number".

Can we say
⟨word⟩ ::= ⟨word with delim⟩/⟨delimiter⟩

⟨id with delim⟩ ::= ⟨ordinary character⟩*⟨delimiter⟩
⟨identifier⟩ ::= ⟨word⟩ & ¬ (⟨number⟩ ⟨ordinary character⟩)
?
⟨special initial-7⟩ ::= `!` | `$` | `%` | `&` | `*` | `/` | `:` | < | `=` | > | `?` | `@` | ^ | `_` | ∼
⟨special initial⟩ ::= ⟨special initial-7⟩

These are just the extended alphabetic characters without "`+ - .`". That is, without the ⟨special subsequent⟩s. R7RS § 2.3 says of these special subsequents: "These are used in numbers, and can also occur anywhere in an identifier." I think it should say "anywhere after the first character in an identifier."
⟨special subsequent⟩ ::= `+` | `-` | `.`
⟨subsequent-6⟩ ::= ⟨initial⟩ | ⟨digit⟩ | ⟨Unicode subsequent⟩ | ⟨special subsequent⟩
⟨subsequent-57⟩ ::= ⟨initial⟩ | ⟨digit⟩ | ⟨special subsequent⟩

A ⟨Unicode subsequent⟩ is any character whose Unicode general category is: Mc, Me, or Nd, that is, any spacing combining Mark, enclosing Mark, or Numeric decimal digit.

With these definitions (**define** *@name* 'joe) and (**define** *name* '(your uncle)) what is
'(refer to ,@ *name* , *@name* as ,*@name*) `'(refer to ,@ name , @name as ,@name)`?
Guile says: (refer to your uncle joe as your uncle).
⟨digit⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨hex digit⟩ ::= ⟨digit⟩ | `a` | `A` | `b` | `B` | `c` | `C` | `d` | `D` | `e` | `E` | `f` | `F`
⟨inline hex escape⟩ ::= `\x` ⟨hex scalar value⟩`;`
⟨hex scalar value⟩ ::= ⟨hex digit⟩$^+$
⟨peculiar identifier-5⟩ ::= `+` | `-` | `...`
⟨peculiar identifier-6⟩ ::= `+` | `-` | `...` | `->` ⟨subsequent⟩*
⟨peculiar identifier-7⟩ ::= ⟨explicit sign⟩ | ⟨explicit sign⟩ ⟨sign subsequent⟩ ⟨subsequent⟩*
| ⟨explicit sign⟩ . ⟨dot subsequent⟩* ⟨subsequent⟩*
| . ⟨dot subsequent⟩* ⟨subsequent⟩*
⟨boolean⟩ ::= `#t` | `#T` | `#f` | `#F`
⟨boolean-7⟩ ::= `#t` | `#f` | `#true` | `#false`
⟨character⟩ ::= `#\`⟨any character⟩ | `#\`⟨character name⟩ | `#\x`⟨hex scalar value⟩
⟨character name-5⟩ ::= `space` | `newline`
⟨character name-6⟩ ::= `nul` | `esc` | `linefeed` | `vtab` | `page`
| `alarm` | `backspace` | `delete` | `return` | `tab` | ⟨character name-5⟩
⟨character name-7⟩ ::= `null` | `escape`
| `alarm` | `backspace` | `delete` | `return` | `tab` | ⟨character name-5⟩
⟨string⟩ ::= `"` ⟨string element⟩* `"`
⟨string element-6⟩ ::= ⟨any character other than `"` or `\`⟩
| `\a` | `\b` | `\t` | `\n` | `\v` | `\f` | `\r` | `\"` | `\\`
| `\`⟨intraline whitespace⟩* ⟨line ending⟩ ⟨intraline whitespace⟩*
| ⟨inline hex escape⟩
⟨string element-7⟩ ::= ⟨any character other than `"` or `\`⟩
| `\"` | `\\`| `\`⟨vertical line⟩
| `\`⟨intraline whitespace⟩* ⟨line ending⟩ ⟨intraline whitespace⟩*
| ⟨inline hex escape⟩
⟨mnemonic escape⟩ ::= `\a` | `\b` | `\t` | `\n` | `\r`
⟨short escape⟩ ::= ⟨mnemonic escape⟩ | `\"` | `\|` | `\\`

**Questions and Comments** The *R7RS Small Errata*[15] says:

> In section 7.1.1 (Lexical structure), the escape sequence `\|` is not shown as permitted in ⟨string element⟩. The list in Section 6.7 shows that it is equivalent to ⟨vertical line⟩. Similarly, the escape sequences `\"` and `\\`should be allowed in ⟨symbol element⟩. This makes the same escape sequences valid in both strings and symbols.

but R$^7$RS called "corrected" and dated February 13, 2021 (See [16]) has the change to ⟨string element⟩ but not the corresponding change to ⟨symbol element⟩.

R$^7$RS [17, §7.1] says: `#\space` and `#\Space` are distinct. Presumably one of those is a name for an Ascii space `#\x20`, what is the other one? Are they two distinct names for the same character? What would be the point of that? Maybe it means that `#\space` is a space, since ⟨character name⟩::=`space`, but `#\Space` is an error since it is distinct and not defined anywhere.

More clues are in §6.6: "if ⟨character⟩ in `#`⟨character⟩ is alphabetic, then any character immediately following ⟨character⟩ can not be one that can appear in an identifier". I want to say a character constant must be followed by a delimiter. The alternative is that it can be any ⟨odd character⟩ as defined above. We might want `#\lambda`= $\lambda$ but `#\Lambda`= $\Lambda$.

Does R$^7$RS require the *read* procedure to accept (and ignore) comments in a ⟨datum⟩?

## 8.2 Lexeme Lists

必也正名乎 —孔子

Most imperative is to rectify names.
— Confucius[1, XIII (Zi Lu) §3]

**Version 2.1** This is saved from a previous version, see the next section for more current information.

To reconstruct the original source file from the Scheme and its atmospherics, the atmosphere may also include things like the numeric format. Numeric formats are as in Common Lisp formats.

The scheme data is in the form of a list of ⟨datum⟩s found in the lexeme lists. If the input is a Scheme program encoded as a list of lexemes, the scheme data is that program encoded as ⟨datum⟩s.

The layout is in the form of a list of ⟨layout-datum⟩s. A ⟨layout-datum⟩ is one of the following forms:

| **Layout** | **Scheme** |
|---|---|
| (Number . ⟨layout-number⟩) | ⟨number⟩ |
| (String . ⟨layout-string⟩) | ⟨string⟩ |
| (Id . ⟨layout-id⟩) | ⟨identifier⟩ |
| (List . ⟨layout-datum⟩*) | ( ⟨datum⟩* ) |
| (Abbrev . ⟨layout-datum⟩*) | (⟨abbreviatable⟩ ⟨datum⟩* ) |
| (Dot . ⟨layout-datum⟩) | . ⟨datum⟩ |
| (NewLine ⟨line-number⟩ ) | nothing |
| (Semicolon ⟨number⟩ . ⟨string⟩ ) | nothing |
| (WhtSpc ⟨spacewidth⟩ ⟨column number⟩ TabSet?) | nothing |

Note that all ⟨layout-datums⟩ begin with a capitalized word.

The layout for a single ⟨lexeme datum⟩, such as ⟨layout-id⟩ or ⟨layout-number⟩ comprises instructions for printing a single lexeme. A atmosphere for a list or abbreviation comprises a list of instructions for printing each of the items in the corresponding list.

A ⟨spacewidth⟩ starts as a single number, which is the number of blanks needed to make the space, but it is changed by the *re-indent* procedure to be a pair

(⟨tab count⟩ . ⟨extra blanks⟩)

where the ⟨tab count⟩ is a count of LaTeX tabs (not to be confused with Ascii `#\tab`).

**Version 3** This is not done yet (2023-01-09), it is notes on how it should be. —Still not done (2024-05-05), both the program code and these notes are changing. Hope they agree soon! —Better read and fix this (2025-01-23), then make code do it.

Any ordinary Scheme implementation will read the ⟨data⟩[6] in a file while ignoring all atmosphere. We want to save the the comments, remarks, newlines, and indentation.

[6] ⟨data⟩::=⟨datum⟩*

These will all be stored as blocks, each of which contains a block type, initial whitespace, and a list of lexemes. Something equivalent to[7] the original source code should be obtained by converting the lexeme lists in the blocks to character strings.

When an Ascii encoded Scheme program is first read by TEX←Scm, it is stored in lists of lexemes, which include both tokens and purely atmospheric lexemes. The tokens encode the data and the atmospheric lexemes encode the atmosphere.

The tokens encode ⟨data⟩, which is the same[8] as what would be obained by the *read* procedure applied to the Ascii Scheme file.

The atmospheric lexemes are called ⟨atmo⟩s; ⟨atmo⟩s are to atmosphere as tokens are to data.

In Ascii encoded Scheme, the atmosphere is distributed over the ⟨data⟩, which is what would be returned by the *read* procedure applied to the Ascii Scheme. Similarily, the tokens and atmos are mixed in any stored lexeme lists.

The lexeme types that can be read from a file are: Id, Number, NewLine, WhtSpc, Abbrev, Semicolon, Char, String, Bool, Open, VOpen, Close, Dot, LexError, EOF,

Lexeme lists are broken into blocks, which are contiguous pieces big enough to be processed as a unit — by Scheme, LATEX, TEX←Scm, or a person as the case may be. The three block types are therefore, ⟨data⟩, ⟨comments⟩, and ⟨remarks⟩. White space, line breaks, and indentation may be scattered through the lexeme lists of all three types of block.

The blocks are encoded as lists that begin with one of the symbols BkDatum, BkComment, BkRemark, or BkEof, followed by an indication of any preceding white space and the indentation level to be used, and finally a lexeme list.

The lexeme lists are not nested; they are just linear lists. Parentheses, both open and close, are each one lexeme. An exception is that a single remark has a lexeme list inside it, just as a comment block has a list of strings.

A ⟨lexeme⟩ is one of the following forms:

| **Lexeme** | **Ascii** | **Datum** |
|---|---|---|
| (Number . ⟨string⟩) | | ⟨number⟩ |
| (String . ⟨strings⟩) | | ⟨string⟩ |
| (Id . ⟨string⟩) | | ⟨identifier⟩ |
| (Char . ⟨string⟩) | | ⟨identifier⟩ |
| (Boolean . ⟨string⟩) | | ⟨identifier⟩ |
| (Open ) | | “(” |
| (VOpen ) | | “#(” |
| (Close ) | | “)” |
| (Dot ) | | “.” |
| (Abbrev . ⟨datum-atm⟩*) | | (⟨abbreviated⟩ ⟨datum⟩* ) |
| (NewLine ⟨line-number⟩ ) | | |
| (WhtSpc ⟨spacewidth⟩ ⟨column number⟩ TabSet?) | | |
| (SemiSharp ⟨datum⟩* ) | | |
| (Comment⟨number of semicolons⟩ ⟨indentation⟩ . ⟨lines⟩ ) | ⟨lines⟩ | |

A Scheme program is composed of files that contain
⟨scm file contents⟩::=(⟨atmo⟩ | ⟨token⟩)$^+$
⟨scm file contents⟩::=⟨block⟩$^+$
⟨directive⟩::=⟨compiler directive⟩ | ⟨dialog directive⟩
⟨datum/atmos⟩ ::= ⟨datum with atmosphere⟩

Note that ⟨data/atmos⟩::=(⟨scm file contents⟩) , that is, if parentheses are put around the contents of a Scheme file the result is a datum with atmosphere.

Just as with *call/cc*, “/” is short for “w/”, which is short for “with”. The computer, under control of the dialog moderator, should not modify the program data or the comments, but it might

[7] See §9.2 for more on “equivalent to”.
[8] See §9.2 for more on “the same”.

modify dialog directives.

A Lexeme list unzips into a list of ⟨datum⟩s, called the program data, which is what the Scheme system sees, and a list of ⟨atmospherics⟩, which is the embedded atmosphere. The atmospherics are encoded as a data, of course, but it is no longer in the form of a flat list of lexems, but rather it is generally the same shape as the program data. (—Why do it that way rather than keep it as a flat list? We might want to take sub-data and corresponding sub-atmosphere.)

## 8.3 Change Log, History, Known Bugs, and Plans

**Change Log**

| Ver. | Date | Init. | Description |
|---|---|---|---|
| 1.0 | 2010-Nov-22 | KW | The *read-tokens*, *write-TeX*, and *write-scheme* procedures are able to process this file. |
| 1.1 | 2010-Dec-05 | KW | Now uses the `scheme` environment and gets rid of explicit line breaks (double backslashes) in the TEX file. This fixes all error messages like "No line to end here". |
| 1.2 | 2010-Dec-15 | KW | *zip* and *unzip* work and are inverse! |
| 1.3 | 2010-Dec-21 | KW | Put on web. |
| 1.4 | 2010-Jan-14 | KW | It can process itself again, this time using the identifier sorting procedures to print keywords, variables, and symbols each in their own special font. Put on web again. |
| 1.5 | 2012-12-14 | KW | Version 1.1 turned out to be a bad idea. The `scheme` environment no longer uses `obeylines`. Instead it uses tabbing, and so we need explicit linebreaks. |
| 1.6 | 2015-10-16 | KW | Rebuild. Change title. Fix string escapes \t, \n. Use *eval* instead of *load* and implement "show" and "show-TeX". |
| 1.7 | 2015-10-16 | KW | Write re-indent. Change NL-indent and Space to NewLine and TabSpc (RSN? LxSpcToCol). |
| 1.8 | 2016-02-04 | KW | This version processes `poly.scm` version 1.0. Put it on web. |
| 1.9 | 2017-07-09 | KW | Change name to `texscm`. Save "working" version. |
| 1.9 | 2017-08-12 | KW | Added noeval to *run-fmt* flag and "\"" in *read-string-contents* so it can process version 2.0 |
| 2.0 | 2017-??-?? | KW | Give up exact reproduction of source in favor of idempotence. Make single space delimiters implicit. |
| 2.x | 2018-11-15 | KW | It no longer works. But I'm working on it. |
| 2.x | 2018-11-23 | KW | I don't even remember how it was supposed to work. I'm starting over. |
| 2.1 | 2022-10-03 | KW | This version can process itself. I will use it to make a section of notes. change \vb to \vbn to avoid clash with `preamble.tex`. |
| 2.2 | 2022-10-10– – 2023-01-08 | KW | Version dates are now creation––last change. I will only make a new numbered version if the current version is worth saving. This version now uses *read-line* instead of *read-char*. It uses sharp-bar comments and can process itself. I will save it and start a new version. |
| 2.3 | 2023-01-08 – –2024-04-03 | KW | Start work on ;# dialog directives. 2024-03-23: Fixed Bug1. Copied back as version $2.3-2i$ . (See page 92 below.) |
| $2.3-1/n$ | 2023-01-08– – 2023 | KW | Make changes here and copy back to version 2.3. when it works. |
| $2.3-1/4$ | 2025-03-30 | KW | Saved as `texscmv23.scm`. The "main program" has been split into two. One still works much as before, the other reads and writes blocks, but does nothing. |

| | | | |
|---|---|---|---|
| $3-1/7$ | 2025-06-23 | KW | Saved as `texscmv3.scm`. The "Vanishing Remark" Bug (See §**??**.) seems to be fixed. Now working on version $3-1/8$ |
| $3-1/9$ | 2025-11-10 | KW | Saved as `history/texscmv23`. This is the last version to use the *ts:layout* procedure. Deprecated code will be removed in future versions. |
| $3-1/10$ | 2025-11-10– –2026-02-20 | KW | Saved as `texscm3-20260220`. Remark blocks mostly work. Probably needs polishing, but it can process `poly3` and itself. |

**History** This section is included in all versions of TEX←`Scm`. If you are reading an old version of the program which has been rebuilt recently, some of the following will have been written in the future. That will be obvious in the Change Log, because the entries are dated, but plans change without date.

2025-11-17: Version 3 is under construction. This is the first to come with a user manual. Both the program and the manual are changing; when they converge it will be verson 3.0, but until then the version number will end in a minus sign and reciprocal of an integer, like $3-1/3$. The denominator will increase as we approach 0. I've gone to hex. Version $3-1/A$ is current best. Stopping at $3-1/F$ and reusing.

Old versions did this:
(**define** *program-title* "`\\TeX$\\leftarrow$\{\\tt␣Scm\}`")
(**define** *program-file-name* "`texscmv22`")
(**define** *program-version* "`2.2`")

and explained it:

> Because the program to be printed is read and evaluated (so that the output can be printed too) definitions like that of *program-file-name* may overwrite some that seem to come later. So the definition above may happen later than the **set!** of the same variable in the *fmt* procedure below. This an odd bug or feature.
>
> —This worked by redefining the variable in the TEX←`Scm` program itself. This should not even be possible. In any case, it is not portable[9]. The `noeval` option should be used when TEX←`Scm` processes itself. Even though the program has no expressions with values that need to be printed, without this option the definitions will be evaluated, with the result that the object program overwrites the compiled and running program with new definitions of its top-level procedures. Those procedures will be replaced by new copies of themselves, and so TEX←`Scm` will actually work! It will, however, be *very* slow since it accomplishes the opposite of just-in-time compilation, replacing the compiled procedures with data that must be interpretted. I think there may be cases in which it would not even work at all, due to unexpected update of a variable.

We no longer rely upon this behavior. The version 3 manual says "`;#:=` ⟨symbol⟩ ⟨value⟩" will set the value of a known symbol, and *program-title* is known.

On the other hand I don't know how to change that behaviour. We must follow several rules to avoid rock-⊥.

- The input program should not set or define any identifiers that begin with tt ts:.
- The TEX←`Scm` version that makes a reply file appends its subversion to the file name. It `\input` its own version.
- To experiment and test, make a `xx:` copy and replace with `ts:` when it works.

**Known Bugs** Actually, I don't remember what I thought I knew when I wrote these. Check them out and fix the list.

[9] See ¶9.2 on page 98 for more on environments.

- 2025-05-30 *eval-blk* does not work if block contains **quote**, even if inside **define** and therefore not actually evaluated.
- 2025-04-05 line-buf eof

```
/home/kwright/srckaw/texscm/./texscm.scm:639:32:
       In procedure read-lexemes-on-line:
       In procedure string-length:
        Wrong type argument in position 1 (expecting string): #<eof>
\begin{verbatim}

\item New in 2.2: tabs in input break indentation of output.
      --- fixed 2022-12-11
\item String escapes do not work.
      In particular $\pmb{\backslash}${\tt "} prints as {\tt "}.
      --- fixed 2022-11-11
\item BugTbTk there is an extra TabTick in \TeX\ output of line
  \begin{verbatim}
      (if #t (set! ts:read-block read-block)  )  #| Test it! |# #| BugTbTk |#
```

- Special characters in identifiers.
- Peculiar identifiers are broken; so are numbers.
- blank lines in input are not blank lines in TEX output, instead they cause extra indentation of the next line.
- Datum comments do not work. Nesting of #— does not work Semicolon comments on the same line cause "Missing $ inserted". Semicolon comments inside ⟨datum⟩ gives

```
! Missing \endcsname inserted.
    <to be read again>
        \global
 l.795 \hs{4.0mm}\=\begin
                       {scheme}%
```

- Quasi-quotation and dotted list are not re-formatted correctly.
- Backquotes containing vectors like `#((0 ,(- b) 1)). Maybe no vector constants work.
- Macros! In particular, quasisyntax abbreviations.
- Lambda with atomic parameter.—Is that fixed? What was wrong?
- Syntax errors in input program crash it horribly.
- In scheme code, lines with blanks or tabs but no visible characters mess up indentation badly.
- (*dialog*) does not read comments following last Scheme form.
- What to do about transput to standard files in evaluated ⟨expressions⟩.
- Fonts (identifier sorts) are wrong after unquote.

Some strange things. I meant to write `(+ 1 k)` I accidentally wrote `(+1 k)`. After processing that printed as (1 *k*). How to add one? I think `1+` is not an identifier, but `++` is. It's not? Can we say + can be ⟨initial⟩ but then digits are not ⟨subsequents⟩? I would like to use identifiers " `++ +- --` ".

I meant to write `(do (...)(...)  (ts:read-char))` but wrote `(do (...)(...)  ts:read-char)` It is obvious, but usually we want to call the procedure, not just mention it. (The `(...)`s were confusing.)

**Numbered Bugs**

**Bug1:** sharp-bar does not work with the bar-sharp at start of next line.
— 2024-02-17(Sat) I think I fixed that. The problem was adding one in *find-index* instead of *read-line-to-bar-sharp* where it was called. The two bugs canceled each other and worked in almost all cases. Fixing only the first made an infinite loop.

**Bug2:** In file `tstread.scm` an extra pair of parentheses

```
(with-input-from-file "tstest.txt"
  (lambda ()
    (let loop ((lx (ts:read-block))
               (ls '()) )
      (if ((or (eof-object? lx)(eq? (lxm-type lx) 'Empty))) #|Bug2 was here|#
          (reverse ls)
          (loop (ts:read-block) (cons lx ls))   )))))
```

took all day to find, because the error message

```
Backtrace:
In unknown file:
          12 (apply-smob/0 #<thunk 7f188cd0d2e0>)
...
In ice-9/eval.scm:
    619:8 10 (_ #(#(#<directory (guile-user) 7f188cd12c80>)))
...
In unknown file:
           5 (eval (show (test-read-block "(* 2 pi)")) #<directory (g...>)
...
Exception thrown while printing backtrace:
In procedure frame-local-ref: Argument 2 out of range: 1

ice-9/eval.scm:279:15: Wrong type to apply: #f
```

**Bug3:** There is no named let*, but trying to use it crashes the texscm program.

### 8.3.1 Two Pass Bug

Using procedure *ts:layout* the value of a lexeme list may be shown differently from what was just defined. Here is a short demonstration of that

```
(define twopassdemo
  (read-lxms-from-lines
   "(define␣pi␣3.14)"))
(show twopassdemo)
⇒((NewLine 240) (Open) (Id . "define") (WhtSpc 1 8) (Id . "pi") (WhtSpc 1 11)‖
‖  (Number . "3.14") (Close) (NewLine 241))
```

Nothing remarkable there. Let's try exactly the same thing again:

```
(define twopassdemo
  (read-lxms-from-lines
   "(define␣pi␣3.14)"))
(show twopassdemo)
⇒((NewLine 250) (Open) (Kw . "define") (WhtSpc 1 8) (Id . "pi") (WhtSpc 1 11)‖
‖  (Number . "3.14") (Close) (NewLine 251))
```

Note that it now says `(Kw . "define")` rather than `(Id . "define")` as it was the first time. Why is it different this time? The answer is revealed below!

It is because *ts:layout* saves results as lexeme lists and actually displays them at the end, after all data in the file have been evaluated. Destructive update can change the saved results before they are displayed.

It is no surprise that if I do

(*set-car!* (*caddr twopassdemo*) '`Kw`)

then

(show *twopassdemo*)

```
⇒((NewLine 250) (Open) (Kw . "define") (WhtSpc 1 8) (Id . "pi") (WhtSpc 1 11)‖
‖  (Number . "3.14") (Close) (NewLine 251))
```

the modification shows up in the displayed result, but the modification should not be seen *before* it is done. This is the two-pass bug.

On the other hand, if an identifier is used before it is defined, it should be printed in the font determined by the definition. Therefore blocks must be saved to be printed at the end of a binding region.

This: `./texscm tstblocks includable verbose > scratchtstblocks.txt` shows correct answers, but they don't get displayed.

For a while, I thought I might be modifying a constant, and tried to prevent that.

(**define** *iddef* (*cons* '`Id` "`define`"))
(show *iddef*)

```
⇒(Id . "define")
```

Make a lexeme list.

(**define** *twotoget*
  (*list* '(Open) *iddef* '(Id . "`pi`") '(Number . "`3.14`") '(Close)
    '(Id . "`pi`") ))
(show *twotoget*)

```
⇒((Open) (Id . "define") (Id . "pi") (Number . "3.14") (Close) (Id . "pi"))
```

but it made no difference from

(**define** *twotoget*
  (*list* '(Open) '(Id . "`define`") '(Id . "`pi`") '(Number . "`3.14`") '(Close)
    '(Id . "`pi`") ))
(show *twotoget*)

```
⇒((Open) (Id . "define") (Id . "pi") (Number . "3.14") (Close) (Id . "pi"))
```

Note that it *is* an error to modify the ⟨literal⟩ quotation `'(Id . "define")`, but that was not the cause of the problem.

In other news:

This turned up when I mistyped the first definition of this section:

This `(define iddef (cons ''Id "define"))` on 2025-04-08 with V2.3−1/4 gets

```
In unknown file:
         3 (eval (define pi-def (read-lexdatum lx-port)) #<directory (...>)
In /home/kwright/srckaw/texscm/./texscm.scm:
2347:10  1 (getall-outlex #<procedure 7f04d80cf6a0 at /home/kwrigh...>)
 127:28  0 (lxm-type _)

/home/kwright/srckaw/texscm/./texscm.scm:127:28: In procedure lxm-type:
In procedure car:
     Wrong type argument in position 1 (expecting pair): #<unspecified>
make: *** [Makefile:76: tstblocks-x.tex] Error 1
```

. . . so that should be checked. Maybe it goes away when I delete old code.

Procedure *make-reply* puts a re-made copy of the source code into `*-r.scm`.

2025-04-04(Fri) At this time

```
diff texscm.scm texscm-r.scm
```

shows that they differ mostly by line breaks and white space. The process is not even idempotent. The second run loses even more line breaks. I should fix that, but. . . :
2025-04-16(Wed) working on other general stuff, but now it's almost idempotent. The differences `diff texscm.scm texscm-r.scm` are: (1) trailing blanks are lost, (2) tabs changed to spaces, (3) one line comments immediately before a datum are moved.

2025-11-17: The current version is idempotent:

```
$ diff -s texscm3.scm texscm3-r.scm
Files texscm3.scm and texscm3-r.scm are identical
```

**Plans** There are several problems in choosing font that need thought.

Proper choice of font will need information from macro expansion. Probably this means that macro expansion must be done by the Dialog Manager with the underlying Scheme system used to run fully expanded code. That will break the ability (never actual, possibly impossible) to print code written for any underlying Scheme system.

We need editorial remarks to force changes. Maybe the macro expander can put them in, maybe we edit by human hand.

What if an identifier is used in one file which loads or includes (or is loaded by or included by) the file in which it is defined?

Read a line at a time. Scan back and forth as needed to find tokens. Use *substring* to copy string name from line to token. Done—2022-12-17.

Scan both comments and Scheme forms for occurrences of \vb{. . .} and replace some with special things, e.g. change `number->string` to *number→string* wherever it occurs.

There is column offset for input tabs (#\tab).

Still needs something (long copies?) to paste lines together for \ \n. A string lexeme, like a comment, contains a list of strings. A string layout contains what's needed.?.

> Nested comments will be tricky because sharp (#) is a special character in both TEX and Scheme. It must be escaped for TEX and must not be for Scheme. Also bar looks wrong except in typewritter font. Can we use **\verbatim**?

R7RS has both *substring* and *read-line* in the base library. R6RS and R5RS only *substring*. Guile has *substring* by default, but *read-line* only after (**use-modules** `(ice-9 rdelim)`)

Think about this `;#`(**if** *verbose* => *skip*).

# 9 Appendix C: First Draft TEX←Scm Manual

## 9.1 Preface

It might be said that the first draft of this program was written around 1995 when I was paid to write Pascal programs. I admired the typeset programs [7, 8] that Knuth wrote under the slogan "Literate Programming" [5, 6], but I was paid to write working programs, not beautiful programs. Still, I wrote a very simple program that read a Pascal program and produced a TEX input that listed the program with keywords in bold-face and comments typeset by TEX.

It is simple to do that with Pascal, because there is a small set of keywords fixed by the language definition. It is far more complicated to do it with Scheme, because macro definitions create new keywords. There is no code left from that so-called "first draft", but there is a design principle that I am following:

(1) The source code is a Scheme program. No preprocessing is needed before compiling and running the program.

A wise grad student once said to me "If you don't touch it, you can't mess it up." But this program is intended to facilitate a dialog. The computer must be allowed to have its say, but must not be allowed to mess it up.

(2) Despite talk of adding remarks to the program, the original (Scheme) source code is not modified. Instead, it is copied to the reply with changes made and remarks added there.

Despite the name "dialog", Plato's dialogues include more than two participants. Of course there may be more than one human programmer working on the program, but there also might be more than one computer program. "The computer" can refer to TEX←Scm or the underlying Scheme system. Some remarks are made by Scheme and recorded by TEX←Scm.

## 9.2 Introduction

Not a compiler, not a pretty printer, TEX←Scm is a new kind of program. It is a Dialog Moderator. It converts a Scheme program into a TEX document which contains the text of the program shown with different fonts for variables, keywords, and symbols. The defining occurrence of an identifier is underlined[10]. The comments in the program are interpreted as TEX commands. That is, it is just Ascii text which the LATEX program can process into a nicely typeset document.

In addition to the program itself, the output of (parts of) the program is shown, in the same way as the text itself. The result looks like a transcript of a REPL session, but with better typography.

A line that begins with "`;#`" is (part of) a remark. Any Scheme system will treat this as a comment. It will not affect the meaning of the program itself, which remains syntactically correct Scheme (assuming it ever was).

Both comments and remarks combine into blocks if they occur on subsequent lines with the same indentation as the first. In the case of comments, this just means that the entire block of consecutive comments is sent to LATEX as a unit, possibly with the result that all of them are combined into a single paragraph in which line breaks are inserted and justification done by LATEX.

In the case of remarks, the TEX←Scm program might make alterations. Since the intention is that reply will replace the original program it must be equivalent, at minimum, in the sense that the changes are idempotent, so running TEX←Scm on the reply should result in a identical copy. The reply to the reply is identical to the reply.

On the other hand, the point of running TEX←Scm at all is that it acts as a REPL, displaying the result of evaluating a block of code as a reply remark. If the correct reply is already there in the block of remarks it is not repeated. An incorrect reply remark will be corrected, not by removing the incorrect reply, but by adding a further remark which complains and displays the computed result.

Each remark begins with a symbol immediately after the "`;#`" with no white space between.

In the case of a remark extended across several lines, each subsequent line must begin with "`;#␣`", indented to the same column as the first "`;#`".

[10] At least it should be. Implementation not started.

After the "`;#`" in that first line of the remark there is an identifier which determines the meaning of all the rest. The most important of these identifiers is "`=>`". There is no whitespace allowed here, so that "`;#=>`" appears as a unit.

— Change that to "`?`" that is "`;#?`" appears as a unit.

So if a line begins with "`;#␣`" then (part of) a remark is contained in the remainder of the line, ending at the end of the line, like any comment. The Scheme program continues on the next line as if nothing happened.

What the TEX←`Scm` program does with it, depends upon what immediately follows the "Semi-Sharp". If it is not a blank to continue the remark on the previous line, then it must be one of the following:

- `;#=>` ⟨datum⟩ or `;#?` ⟨datum⟩
  Print the result of evaluating the preceding ⟨expression⟩ (i.e. part of the Scheme program). The given ⟨datum⟩ is the proposed result. If the ⟨datum⟩ comprises several lines, each line must begin with `;#` followed by white space. If the result computed by evaluating the preceding expression is *equal?* to the proposed result then nothing happens, the `;#=>` ⟨datum⟩ remains in the program text[11]. If the ⟨datum⟩ computed by evaluation differs from the proposed result then the proposed result still remains as is, but it is followed by remarks inserted by TEX←`Scm`. The newly inserted remarks consists of a warning message followed by the computed value of the preceding ⟨expression⟩.

  For example:
  `(* 6 7)`
  `;#?` 42
  remains unchanged in the Scheme file and prints as
  (* 6 7)
  ⇒ 42

  On the other hand:
  `(* 6 7)`
  `;#?` 48
  is updated to something like:
  `(* 6 7)`
  `;#?` 48
  `;#!` {\bf Warning! } proposed result differs from computed:
  `;#!!` 42
  and prints as
  (* 6 7)
  ⇒ 42
  ⇒**? Warning!** proposed result differs from computed:
  ⇒**!** 48
  The (human) programmer should understand the reason for the discrepency, delete the incorrect ⟨datum⟩, and make the correct one the the new proposed result. Atmospherics in the proposed result may be edited while doing this.

  One might want to write:
  `(* 6 7);#=>`
  all on one line. This could conflict with a definition of "block" that requires blocks to consist of whole lines, or we could say that the trailing remark is atmosphere in the data block and that responding to it is part of evaluating the data.

- `;#??` or? `;#?t ;#?;`
  The result of evaluating the preceding ⟨exression⟩ should be not a general datum, but a (list of) string(s), which should be sent to LATEX as is.

[11] Maybe add a reply remark that says result was checked

`;# ;` ⟨character⟩*
Following lines contain those strings as the body of one-semicolon comments. Those strings are sent directly to LaTeX.

- `;#:=` ⟨symbol⟩ ⟨value⟩
The ⟨symbol⟩ must be one of a list which is meaningful to the dialog manager. For example:
`;#:=` *title* "`\\TeXScm`"
`;#:=` *author* "`Keith Wright`"
`;#:=` *cmnt-width* "`8cm`"
`;#:=` *cmnt-width* (*auto*)
Let's call these "editorial remarks" because they do not pertain to the results of the program, but to the way they are presented.

- `;#?s`, `;#?d`, `;#?s`
result should be printed as Scheme, data, or TeX, respectively.

- `;#?sm`
result should be printed as Scheme with macros expanded.

Inside a comment, Scheme code is marked `;#'` . This is formatted for printing as Scheme, but of course it is not evaluated. If I write inside a comment:
`This is a semisharp-quote ;#'(define pi 3.14159) faked for demo`
I should see "This is a semisharp-quote #'(**define** *pi* 3.14159) faked for demo". Note `#'`⟨datum⟩ is the R6RS abbreviation for (`syntax` ⟨datum⟩). See R6RS [18, §4.3.5].

## • Appendix D: Dialog: Random Toughts

you can not see the Vision 'till
you've redesigned your eyes

If we say that European philosophy "consists of a series of footnotes to Plato"[12], then we must admit that the Platonic "dialogs" are not verbatim transcripts. They are ideal dialogs, recalled with mistakes forgotten, and with later interpolations and corrections.

The Scheme program `poly.scm` can be loaded and evaluated. That is part of the process of producing the document `poly.ps` which contains both the source code in `poly.scm` and its output (in this case Platonic polyhedra).

The other part of producing the document is giving `poly.scm`, as input, to the program in the file `texscm.scm`. In addition to carefully evaluating the input program to produce its output, it gathers the input program and its output into one single typeset document.

Suppose we want to show the program in a different way?
(integrate (lambda(x) (exp (- (* x x)))))
`;#|->`mything
$$\longmapsto \int_{-\infty}^{+\infty} e^{-x^2}\,\mathrm{d}x$$
In the default case mything=eval. The less ambitious might want a way to write code to translate a bar-delimited identifier to a bit of TeX. `|\Omega|` $\longmapsto \Omega$. Or maybe show the result of macro-expansion.

There are two features needed; a way to get the previous program part as a ⟨datum⟩, and a way to apply an arbitrary user specifed procedure to that datum.

There could be source line numbers printed in the left margin, and page numbers of printed document inserted in the source as `;#page` ⟨decimal number⟩.

There should be a variant of `;#=>` ⟨datum⟩ that puts the result in a file instead of inline. Track changes but don't print. Whether or not to print is independant of whether the data is in a file or inline.

[12] Whitehead [22, Part II,Ch.I§I,p.63] says it does. (This footnote was added two days after the quotation, which itself was corrected.)

• **Equivalence** The reply *-r.scm should be "equivalent" to the original *.scm program but obviously can not be identical as Ascii. —Why not identical copy of lines in a data block with additional lines of remarks? — That's not how version 3 works, trailing blanks are deleted; Ascii \t tabs may become spaces. What else?

• **Environments** The Scheme eval procedure needs an environment, but R$^7$RS does not say much about how they must work. The Guile manual seem half done on this point. The main unanswered question is what happens when `(eval '(define z 0) env)`? Experiment indicates that `z` is added to `env` so `(eval 'z env)` is 0. What if `env` were produced by `(define env old-env)`? What is `(eval 'z old-env)`?

• **Digging in the Feature Pile**

> Programming languages should be designed
> not by piling feature on top of feature...
> — R*RS

The usual way to construct a program in a "batch" language such as **C** is to put the text of the program into a file, then to compile and execute it, look at the output, change the program and repeat.

An "interactive" language can be run in a REPL, a "Read-Eval-Print-Loop". The programmer enters an expression, the computer reads it, evaluates it, prints its value, and the process repeats. This can be great fun, but when it is done where is the program?

In a similar manner this program, although it takes the form of a Scheme program written by a programmer with results calculated by a computer, is actually more of a fixed point reached by a read-evaluate-print-edit loop, which involves both the programmer and the computer repeatedly making changes.

The usual `Makefile` mechanism has the disadvantage that fixing a spelling error or adding a few words of explanation in the comments of a lower-level library could result in re-compilation of the world.

Preserve the distinction between loading the half-completed progam and appending vs editing to correct. Not all assignments are done to correct mistakes, those that are should just be edits.

Can Programming be Liberated from the von Neumann Style? Assignments are for big changes; otherwise it's just a function.

Are macros all expanded before the program starts running, or is each macro call expanded as needed? If you need to know, then macros are poorly designed. Expanding a macro freezes all variables it uses. A frozen variable can not be assigned, its definition must be edited to change it. Early expansion is like compilation; as-needed expansion is like JIT compilation.

**Keywords:** Many versions of `Scheme` have a feature they call "keywords"; the following SRFIs propose variations:

**srfi-88:** Keyword objects by Marc Feeley (Final) proposes keyword objects that end with a colon

**srfi-89:** Optional positional and named parameters by Marc Feeley (Final)

**srfi-177:** Portable Keyword Arguments by Lassi Kortela (withdrawn)

All these seem, not explictly, to treat a keyword as a notation for a new expressed value (in the sense of R$^7$RS §7.2.2) which can be passed to a procedure, which then at run-time is responsible for parsing them out and subsituting the following argument for the correct parameter in the body of the procedure. Can we then do this: `(make-window (if sideways #:width #:height) 200)`?

In the Scheme Reports (R$^5$RS,R$^7$RS) the word "keyword" is in the index and points to §4.3 Macros, where we read: "Program-defined expression types have the syntax (⟨keyword⟩ ⟨datum⟩ ...) where ⟨keyword⟩ is an identifier...".

These keywords, like those in R$^7$RS should be part of macro expansion. In particular literals in a macro definition are keywords.

As [2] says, the binding structure is part of the macro's shape, therefore binding occurrences are underlined. Is every binding occurrence of an identifier bound by the smallest containing macro call? What if a literal keyword in one macro definition is the same identifier as one already bound? Does it depend upon whether it is already a keyword or identifier?

The macro expander must produce a list of binding occurences when given a macro definition. Underlining must be done in the text of a macro call. While developing, give it some hints in dialog manager commands.

In R$^7$RS [17, §5.5] the description of records is rather sketchy. It is, in fact, an almost word for word copy of SRFI-9, without the implementation. Under **Pairs and Lists** [17, §6.4,p40] it says that a pair is a record structure. This may be more help to understand records than pairs. Several times it says "record structures... do not have datum representations". That's an unpleasant surprise and a show-stopper for any use of records in this program.

An infinite loop is a side-effect.

A Scheme file can have directives, such as `#!fold-case`, which are treated as comments, except that they affect the operation of the compiler. That is, they do not denote data or algorithms, but affect the way a text is interpretted at a very early stage. It is tempting to generalize this to `#!`⟨identifier⟩, and use these to give directives to TEX←`Scm`.

I don't do that [13], because TEX←`Scm` is not the compiler. The compiler could break if confronted with an unknown directive. Instead, we use `;#`. Any Scheme implementation will ignore the rest of the line, but TEX←`Scm` will treat it as directives. Just as `#;` will "comment out" the following ⟨datum⟩, `;#` comments it in. The layout directive (dialog directive?) is the rest of the line. If `;#` is followed by white space, the the rest of the line is taken as a continuation of the previous directive, if any.

There is a `*.scm` file which contains contributions from both the programmer and the computer. The directive `;#=>` goes between an expression and a suggested answer. The computer evaluates the expression and replaces the suggested answer with the computed answer, taking layout from the suggested answer. If the computed answer is not the same as the suggested, a warning is added.

The directive `;#->` goes after an expression. The computer evaluates the expression, the result is a list of strings, which replace all lines up to next `;#<-`. When processed by TEX←`Scm`, that means that it goes straight through to LATEX.

The computer can add directives, which serve to record results which might be expensive to re-compute, or to communicate between different programs.

Scheme expression (*dialog* "`file-name`") means load the file, but remember it and reload if it changes.

The `prog.dat` file can be created by simply reading the program (with *read*) and writing it back out. The lexeme lists are needed to make `prog.tex`. Why do we need to unzip and zip?

# References

[1] Confucius, *Analects*, (≃450BC) in *The Four Books, with English translation and notes by James Legge*, Unknown publisher, Hong Kong (1861) and The Chinese Book Company, Shanghai (no date) also in *Confucius* second revised edition Clarendon Press, Oxford (1893) and Dover (1971)

[2] David Herman & Mitchell Wand, *A Theory of Hygienic Macros*, ESOP 2008 & Springer-Verlag LNCS 4960 pp.48–62 (2008)

[3] Aaron Hsu, *ChezWEB User's Guide*, gopher://gopher.sacrideo.us/1chezweb

[4] Richard Kelsey, et. al. *Revised$^5$ Report on the Algorithmic Language Scheme*,

[5] Donald E. Knuth, *Literate Programming*, Computer Journal 27(1):97–111, 1984.

[6] Donald E. Knuth, *Literate Programming*, CSLI, 1992. CSLI Lecture notes no. 27.

[13] TEX←`Scm` might need to look at some compiler directives, for example to print the program differently if `#!fold-case` is in effect.

[7] Donald E. Knuth, *TEX: The Program (Volume B of Computers and Typesetting)*, Addison-Wesley, 1986, ISBN 0-201-13438-1

[8] Donald E. Knuth, *Metafont: The Program (Volume D of Computers and Typesetting)*, Addison-Wesley, 1986, ISBN 0-201-13438-1

[9] Helmut Kopka and Patrick W. Daly, *A Guide to LATEX, third edition*, Addison Wesley (1999)

[10] B. O. Peirce, *A Short Table of Integrals*, Ginn and Company (1929)

[11] . Plato, *The Collected Dialogs*, Bollingen Series LXXI, Princeton University Press

[12] Norman Ramsey, *Literate programming: Weaving a language-independent web*, Communications of the ACM, 32(9):1051–1055, 1989.

[13] Norman Ramsey *The noweb Hacker's Guide*, Princeton University, 1992. Revised 08/1994.

[14] John D. Ramsdell *SchemeWEB – WEB for Lisp. Simple support for literate programming in Lisp.* The MITRE Corporation, 1994

[15] `https://small.r7rs.org/wiki/R7RSSmallErrata/`

[16] Alex Shinn, John Cowen, and Arthur A. Gleckler (eds.),

[17] Alex Shinn, John Cowen, and Arthur A. Gleckler (eds.), *Revised$^7$ Report on the Algorithmic Language Scheme*,

[18] Michael Sperber, et. al. *Revised$^6$ Report on the Algorithmic Language Scheme* Cambridge University Press ISBN: 9780521193993 (2010) or http://www.r6rs.org/

[19] Dorai Sitaram, *SLaTeX*, 1991, 1999
`http://www.ccs.neu.edu/~dorai/slatex/slatex.tar.gz`

[20] Michael Spivak, *Calculus on Manifolds*, Benjamin/Cummings Publishing (1965)

[21] Fransisci Vietæ, *Variorum de rebus mathematicis responsorum* (1593)

[22] Alfred North Whitehead, *Process and Reality*, Macmillan Company (1929)