```
;; This Scheme+ Version 1.2
;; Built on Wed Sep 14 16:47:44 EDT 1994
;; Includes:
;;    macros.scm
;;    constructor.scm
;;    datatype.scm
;;    match.scm
;;    top-level.scm

(declare (usual-integrations))

(define scheme+/version "1.2")

;;-------------------------------------------------
;; macros.scm
;;; ----------------------------------------------------------------------------
;;; Macro definitions

;;; Create the syntax table to be used by the compiler
(define scheme+/syntax-table
(make-syntax-table system-global-syntax-table))

;;; Scheme+ macros need to be visible both at run-time and a compile-time
;;;
;;; BJR: put define-scheme+-macro in user-initial-syntax-table so that
;;; we can load datatype.scm and match.scm independently.
;;;
(syntax-table-define
user-initial-syntax-table
'define-scheme+-macro
(macro (pattern . body)
`(begin
    (syntax-table-define
    user-initial-syntax-table
    ',(car pattern)
    (macro ,(cdr pattern) ,@body))
    (syntax-table-define
    scheme+/syntax-table
    ',(car pattern)
    (macro ,(cdr pattern) ,@body)))))

;;Also need it in this file so it's visible when we concat everything together.
(define-macro (define-scheme+-macro pattern . body)
`(begin
(syntax-table-define
    user-initial-syntax-table
```

```
        b.  ',(car pattern)
(macro ,(cdr pattern) ,@body))
(syntax-table-define
        a.  scheme+/syntax-table
        b.  ',(car pattern)
(macro ,(cdr pattern) ,@body))))

;;; Get real scheme value in expanded code
(define-scheme+-macro (get-real identifier)
`(access ,identifier ()))


;;; ---------------------------------------------------------------------
;;; Error handling macros - need to be visible both in this file
;;; and at run time.
;; BJR - These are now scheme procedures

; (define-macro (define-macro-both pattern . body)
;   `(begin
;      (define-macro ,pattern ,@body)
;      (syntax-table-define
;           user-initial-syntax-table
;           ',(car pattern)
;        (macro ,(cdr pattern) ,@body))))
; 
; (define-macro-both (syntax-error string . rest)
;   `(error (string-append "\nSCHEME+ SYNTAX ERROR: " ,string)
;           ,@(if rest rest '())))
; 
; (define-macro-both (runtime-error string . rest)
;   `((get-real error)
;     ((get-real string-append) "\nSCHEME+ RUN-TIME TYPE ERROR!\n"
;                               ,string)
;           ,@(if rest rest '())))

(define (scheme+/syntax-error msg . rest)
(error (apply error-string "\nSCHEME+ SYNTAX ERROR: " msg rest)

        a.  rest))

(define (scheme+/runtime-error msg . rest)
(error (apply error-string "\nSCHEME+ RUN-TIME TYPE ERROR!\n" msg rest)
        a.  rest))
```

```
;; From mini-fx/version-4.2.7.scm

(define (error-string str msg . objs )
(string-append
str
msg
"\n"
(objects->string objs)
"\n"
))


(define objects->string
(let ((separator "------------------------------------------------------------"))
(lambda (objs)
(if (null? objs)
    ""
    (string-append
    separator
    (apply
    string-append
    (map
    (lambda (obj)
        (string-append
        (with-output-to-string (lambda () (pp obj)))
        "\n"
        separator
        ))
    objs)))))))
```

```
;;-------------------------------------------------
;; constructor.scm
;;;-------------------------------------------------------------------------
;;; Constructors

;;; reistad 8/19/94 -- added quote and quasiquote s-expression consructors
;;;                    from mini-fx.scm (1991)



(define constructor-tag   '(constructor))

(define (scheme+/make-constructor bundler unbundler)
(make-apply-hook bundler
        (list constructor-tag unbundler)))
(define make-constructor scheme+/make-constructor)

(define (constructor? obj)
(and (apply-hook? obj)
(let ((extra (apply-hook-extra obj)))
    (and (list? extra)
    (= (length extra) 2)
    (eq? (first extra) constructor-tag)))))


(define (bundler obj)
(if (constructor? obj)
obj
(error "bundler: Not a constructor!" obj)))

(define (unbundler obj)
(if (constructor? obj)
(second (apply-hook-extra obj))
(error "unbundler: Not a constructor!" obj)))

;;;-------------------------------------------------------------------------
;;; Simulated LIST datatype

(define null
        (make-constructor
        (lambda () '())
        (lambda (obj succ fail)
        (if (null? obj)
    (succ)
    (fail)))))
```

```
104.        (define cons
105.        (make-constructor
106.        (access cons system-global-environment)
107.        (lambda (obj succ fail)
108.        (if (pair? obj)
        a.  (succ (car obj) (cdr obj))
        b.  (fail)))))

109.        ; (define null
110.        ;   (make-constructor
111.        ;    (lambda () '())
112.        ;    (lambda (obj succ fail)
113.        ;      (if *datatype-paranoid-match?*  ; See DATATYPE.SCM
114.        ;          (if (not (or (null? obj) (pair? obj)))
115.        ;              (error "Deconstructor for NULL given a non list")))
116.        ;      (if (null? obj)
117.        ;          (succ)
118.        ;          (fail)))))
119.        ;
120.        ; (define cons
121.        ;   (make-constructor
122.        ;    (access cons ())
123.        ;    (lambda (obj succ fail)
124.        ;      (if *datatype-paranoid-match?*  ; See DATATYPE.SCM
125.        ;          (if (not (or (null? obj) (pair? obj)))
126.        ;              (error "Deconstructor for CONS given a non list")))
127.        ;      (if (pair? obj)
128.        ;          (succ (car obj) (cdr obj))


129.        ; This can't work without the deconstructor being passed the
130.        ; number of args given to the deconstructor (or a datum from
131.        ; which that number could be constructed).
132.        ;
133.        ; (define list
134.        ;   (make-constructor
135.        ;    (access list ())
136.        ;    (lambda (obj succ fail)
137.        ;      (if (list? obj)
138.        ;          (apply succ obj)
139.        ;          (fail)))))

140.        ;;;----------------------------------------------------------------------
141.        ;;; Simulated S-EXPRESSION datatype
```

```
142.        (define (make-sexp-constructor sym pred)
143.        (make-constructor
144.        (lambda (sexp)          ; Constructor does a type check with PRED
145.        (if (pred sexp)
        a.  sexp
        b.  (error "Sexp constructor -- incorrect type:" sym sexp)))
146.        (lambda (sexp succ fail)   ; Deconstructor does a type check with PRED
147.        (if (pred sexp)
        a.  (succ sexp)
        b.  (fail)))))

148.        (define int->sexp  (make-sexp-constructor 'int  integer?))
149.        (define real->sexp (make-sexp-constructor 'real real?))
150.        (define bool->sexp (make-sexp-constructor 'bool boolean?))
151.        (define char->sexp (make-sexp-constructor 'char char?))
152.        (define string->sexp (make-sexp-constructor 'string string?))
153.        (define symbol->sexp (make-sexp-constructor 'symbol symbol?))
154.        (define list->sexp   (make-sexp-constructor 'list   list?))
155.        (define vector->sexp (make-sexp-constructor 'vector vector?))
```

```
;; reistad 8/24/94 -- added for orthogonality/uniformity with Scheme

(define (scheme+/make-check-type-constructor sym pred?)
(make-constructor
(lambda (obj) (if (pred? obj)
    i. obj
   ii. (error "Object was not of correct type:" sym obj)))
(lambda (obj succ fail)
(if (pred? obj)
  a. (succ obj)
  b. (fail)))))

(define a-number    (scheme+/make-check-type-constructor 'number
number?))
(define an-integer  (scheme+/make-check-type-constructor 'integer
integer?))
(define a-real      (scheme+/make-check-type-constructor 'real real?))
(define a-complex   (scheme+/make-check-type-constructor 'complex
complex?))
(define a-rational  (scheme+/make-check-type-constructor 'rational
rational?))
(define a-char      (scheme+/make-check-type-constructor 'char char?))
(define a-boolean   (scheme+/make-check-type-constructor 'boolean
boolean?))
(define a-symbol    (scheme+/make-check-type-constructor 'symbol
symbol?))
(define a-procedure (scheme+/make-check-type-constructor 'procedure
procedure?))
(define a-vector    (scheme+/make-check-type-constructor 'vector
vector?))
(define a-list      (scheme+/make-check-type-constructor 'list list?))
(define a-string    (scheme+/make-check-type-constructor 'string string?))
```

```
;;; reistad 8/19/94 -- taken from mini-fx (1991) for low level macro system
;;;
;;; Additional procedures for manipulating s-expressions with QUOTE,
;;; QUASIQUOTE, UNQUOTE, and UNQUOTE-SPLICING.

(define (scheme+/make-quote-constructor sym)
(make-constructor
(lambda (x) (list sym x))                ;; Constructor tags item
(let ((pred (lambda (thing)              ;; Deconstructor checks tag
    i.  (and (pair? thing)
   ii.  (eq? (car thing) sym)
  iii.  (pair? (cdr thing))
   iv.  (null? (cddr thing))))))
(lambda (thing succ fail)
(if (pred thing)
a. (succ (cadr thing))
b. (fail))))))

(define quoted->sexp          (scheme+/make-quote-constructor 'quote))
(define quasiquoted->sexp     (scheme+/make-quote-constructor
    'quasiquote))
(define unquoted->sexp        (scheme+/make-quote-constructor
    'unquote))
(define unquoted-splicing->sexp (scheme+/make-quote-constructor
    'unquote-splicing))
```

```
188.        ;;-----------------------------------------------
189.        ;; datatype.scm
190.        ;;;-------------------------------------------------------------------------
191.        ;;; DATATYPE.SCM
192.        ;;;
193.        ;;; Author: Lyn
194.        ;;; Log:
195.        ;;; * 8/19/94 - reistad: Eliminated use of define-syntax so define-datatype
196.        ;;;                  macro is now in match.scm.  Also dropped use of dtyp-desc
    as
197.        ;;;                  Scheme+ ingnores all type crap.
198.        ;;;       Define-Datatype now defines the constructor name to be a package
199.        ;;;         of both a bundler and unbundler as we've eliminated ~.  The
200.        ;;;       constructor is behaves just like the bundler but has a way to
201.        ;;;       get to the unbundler.
202.        ;;; * 8/11/94 - Updated
203.        ;;; * 6/20/94 - Created
204.        ;;;
205.        ;;; Notes:
206.        ;;; Implements datatypes for Scheme+. There is a design choice here that
207.        ;;; doesn't exist for Mini-FX: what should an unbundler do when not given
208.        ;;; a discriminant of the expected type? E.g., suppose BAR is a constructor
209.        ;;; for datatype FOO, and P is an instance of the CONS datatype. Should
210.        ;;;
211.        ;;;    ((unbundler bar) p <succeed> <fail>)
212.        ;;;
213.        ;;; signal an error, or merely call <fail>?  The latter is more in the
214.        ;;; "Scheme tradition", but the former provides superior error messages
215.        ;;; (otherwise, the only error is the largely unuseful "No pattern
    matched!").
216.        ;;; The latter makes sense if we view MATCH as being instantiated to a
217.        ;;; particular datatype before we apply it.
218.        ;;;
219.        ;;; Since its not clear which version is The Right Thing, I provide a flag
220.        ;;; *DATATYPE-PARANOID-MATCH?* that controls how this situation
    is handled.
221.        ;;; By default, this is #t -- i.e., all desconstructors in a given match
222.        ;;; are assumed to be for the same datatype.
223.        ;;;
224.        ;;;-------------------------------------------------------------------------

225.        ;; see NOTES above
226.        (define *datatype-paranoid-match?* #t)
```

```
227.        ;;;
    $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
    $$$$$$$$$
228.        ;;; DEFINE-DATATYPE Begin
229.        ;;;
230.        ;;; DEFINE-DATATYPE declares a sum-of-products data structure
231.        ;;; and creates the associated constructors.
232.        ;;;
233.        ;;; NOTE: the implementation below uses the same name for both the
    constructor
234.        ;;; and deconstructor.  (Actually uses application hooks to store the
235.        ;;; deconstructor on the constructor.) -- ACTUALLY, renamed so that
236.        ;;; always use constructor to refer to bundler/unbundler pair.
237.        ;;;
238.        ;;; E.g.
239.        ;;;
240.        ;;;   (define-datatype int-tree
241.        ;;;     (int-leaf int)
242.        ;;;     (int-node int-tree int-tree))
243.        ;;;
244.        ;;; expands into
245.        ;;;
246.        ;;;   (define int-leaf (make-constructor
247.        ;;;                             (lambda (i) ...)                ;; constructor
248.        ;;;                             (lambda (t succ fail) ...)))    ;; deconstructor
249.        ;;;   (define int-node (make-constructor
250.        ;;;                             (lambda (t1 t2) ...)            ;; constructor
251.        ;;;                             (lambda (t succ fail) ...)))    ;; deconstructor
252.        ;;;
253.        ;;; where
254.        ;;;
255.        ;;;   (int-leaf 0) -> #[int-tree:int-leaf 0]
256.        ;;;
257.        ;;;   (int-node (int-leaf 0) (int-leaf 1)) ->
258.        ;;;         #[int-tree:int-node
259.        ;;;             #[int-tree:int-leaf 0]
260.        ;;;             #[int-tree:int-leaf 1]]
261.        ;;;
262.        ;;; Here's a sample procedure using INT-TREE. Note the use of
263.        ;;; deconstructors in the patterns of MATCH (eventhough there's no ~s)
264.        ;;;
265.        ;;;   (define (int-tree-sum t)
266.        ;;;     (match t
```

```
267.        ;;;      ((int-leaf n) n)
268.        ;;;      ((int-node left right) (+ (tree-sum left) (tree-sum right)))))
269.        ;;;
270.        ;;;
271.        ;;; DEFINE-DATATYPE can be parameterized by type as well:
272.        ;;; (Yeah, but all types are ignored -- documentation only.
273.        ;;;
274.        ;;;  (define-datatype (tree t)
275.        ;;;    (leaf t)
276.        ;;;    (node (tree t) (tree t)))
277.        ;;;
278.        ;;;DROP THIS?
279.        ;;; These type parameters are (conceptually) parameters of the constructors
280.        ;;; and deselectors, but can be thought of as being removed by
281.        ;;; implicit projection.  E.g. the above "expands" into
282.        ;;;
283.        ;;;  ; Constructors
284.        ;;;  (define leaf (plambda (t) (lambda (i) ...)
285.        ;;;  (define node (plambda (t) (lambda (t1 t2) ...)
286.        ;;;  ; Deconstructors
287.        ;;;  (define leaf~ (plambda (t) (lambda (t success-cont fail-cont) ...))
288.        ;;;  (define node~ (plambda (t) (lambda (t success-cont fail-cont) ...))
289.        ;;;
290.        ;;; but the t's never have to be specified:
291.        ;;;
292.        ;;;  (define (tree-depth t)
293.        ;;;    (match t
294.        ;;;      ((leaf t) 0)
295.        ;;;      ((node left right) (1+ (max (tree-depth left) (tree-depth right))))
296.        ;;;
297.        ;;;      ((node left right) (1+ (max (tree-depth left) (tree-depth right))))))
298.        ;;;
299.        ;;;

300.        ;;; ----------------------------------------------------------------------
301.        ;;; Begin DEFINE-DATATYPE code

302.        ;;; ----------------------------------------------------------------------
303.        ;;; The DEFINE-DATATYPE macro

304.        ;;; There are two possible forms for DEFINE-DATATYPE:
305.        ;;;
306.        ;;; (define-datatype I (I D*)*)           ; Straightforward datatype
307.        ;;; (define-datatype (I I*) (I D*)*)      ; Parameterized datatype
308.        ;;;
309.        ;;; Here we ignore any params if they occur.
```

```
;;;
;;; In this implementation we ignore types altogether and simply simulate the
;;; run-time behavior of DEFINE-DATATYPE.


;;; Begin DEFINE-DATATYPE macro
(define-scheme+-macro (define-datatype name-or-name+params . sum-of-products)

(define (datatype-name header)
(cond
((symbol? header)      header)        ; (define-datatype I (I D*)*)
((and (pair? header)              ; (define-datatype (I I*) (I D*)*)
    a. (every? symbol? header))
(car name-or-name+params))
(else                                 ; otherwise is not a legal define-datatype
(scheme+/syntax-error "Ill-formed DEFINE-DATATYPE header: " header))
))

(define (datatype-clause-names-and-params clause-list receiver)
(if (null? clause-list)
(receiver '() '())
(let ((first-clause (car clause-list)))
    a. (if (and (list? first-clause)
            i. (>= (length first-clause) 1))
    b. (datatype-clause-names-and-params
            i. (cdr clause-list)
           ii. (lambda (names params)
          iii. (receiver
           iv. (cons (check-clause-name (car (car clause-list)))
                    1. names)
            v. (cons (map check-clause-param (cdr (car clause-list)))
                    1. params))))
    c. (scheme+/syntax-error
            i. (string-append
           ii. "Ill-formed DEFINE-DATATYPE clause.\n"
          iii. "(Clauses must be of the form: (<constructor-name> <type_1> ... <type_n>) ):\n"
           iv. )
            v. first-clause)))))

(define (check-clause-name obj)
(if (symbol? obj)
obj
```

```
330.        (scheme+/syntax-error
        a. "Non-symbolic constructor name within a DEFINE-DATATYPE
           clause:\n"
        b. obj)))

331.        ;; *** Need to have this do real error checking in future
332.        (define (check-clause-param type) type)

333.        (define (every? test lst)
334.        (if (null? lst)
335.        #t
336.        (and (test (car lst))
        a. (every? test (cdr lst)))))

337.        ; Main body of DEFINE-DATATYPE macro
338.        (let ((dname (datatype-name name-or-name+params)))
339.        (datatype-clause-names-and-params sum-of-products
340.        (lambda (clause-names clause-params)
341.        `(begin
        a. ;; Since Scheme+ has a single namespace, we must bind the datatype
        b. ;; name to *some* kind of object in order to preserve the semantics
        c. (DEFINE ,dname ,(make-datatype-descriptor dname)) ;;? clause-names
        d. ,@(map (lambda (cname cparams)
                i. `(DEFINE ,cname
               ii. (scheme+/make-constructor
                       1. (scheme+/make-datatype-bundler ',dname
                                                  a. ',cname
                                                  b. ,(length cparams))
                       2. (scheme+/make-datatype-unbundler ',dname
                                                  a. ',cname))))
              iii. clause-names
               iv. clause-params)
        e. ,dname))))
342.        )
343.        ;;; End DEFINE-DATATYPE macro


344.        ;;; ----------------------------------------------------------------------
345.        ;;; DEFDATATYPE is a synonym for DEFINE-DATATYPE

346.        (define-scheme+-macro (defdatatype name-or-name+params . sum-of-
    products)
347.        `(DEFINE-DATATYPE ,name-or-name+params ,@sum-of-products))
```

```
;; helper procedures

(define (scheme+/make-datatype-bundler dtyp-name cnstr-name nargs)
(lambda args
(if (= nargs (length args))
(make-datatype-instance dtyp-name dtyp-name cnstr-name args)
(error "Incorrect number of arguments given to bundler"
    i. cnstr-name
   ii. args))))
(define make-datatype-bundler scheme+/make-datatype-bundler)

(define (scheme+/make-datatype-unbundler dtyp-name cnstr-name)
(lambda (obj succ fail)
(if *datatype-paranoid-match?*  ; Could move this outside if
        1. ; wanted creation-time choice.
;; This case signals an error if OBJ isn't exactly right.
(scheme+/ensure-datatype dtyp-name cnstr-name obj)
obj)
(if (and (datatype-instance? obj)
a. (eq? dtyp-name (datatype-instance-descriptor obj))
b. (eq? cnstr-name (datatype-instance-constructor obj)))
(apply succ (datatype-instance-args obj))
(fail))))
(define make-datatype-unbundler scheme+/make-datatype-unbundler)

(define (scheme+/ensure-datatype dtyp-name cnstr-name obj)
(cond
((not (datatype-instance? obj))
(error (string-append "Unbundler "
            1. (symbol->string cnstr-name)
            2. " for datatype "
            3. (symbol->string dtyp-name)
            4. " applied to non-datatype instance: ")
b. obj))
((not (eq? dtyp-name (datatype-instance-descriptor obj)))
(error (string-append "Unbundler "
            1. (symbol->string cnstr-name)
            2. " for datatype "
            3. (object->string dtyp-name)
            4. " applied to instance of datatype "
            5. (object->string (datatype-instance-descriptor obj))
            6. ": ")
b. obj))
(else obj)))
```

```
372.    (define (print-datatype-descriptor state desc)
373.    (unparse-string state "#[datatype ")
374.    (unparse-object state (datatype-descriptor-name desc))
375.    (unparse-string state "]"))

376.    (define (print-datatype-instance state instance)
377.    (let ((type-name (datatype-instance-type-name instance))
378.    (constructor (datatype-instance-constructor instance))
379.    (args (datatype-instance-args instance)))
380.    (unparse-string state "#[")
381.    (unparse-object state type-name)
382.    (unparse-string state ":")
383.    (unparse-object state constructor)
384.    (for-each (lambda (arg)
                  i. (unparse-string state " ")
                  ii. (unparse-object state arg))
        b. args)
385.    (unparse-string state "]")
386.    ))

387.    (define (object->string obj)
388.    (with-output-to-string
389.    (lambda () (display obj))))


390.    ;;; Structures

391.    (define-structure (datatype-descriptor
                  i. (print-procedure print-datatype-descriptor))
392.    name)

393.    (define-structure (datatype-instance
                  i. (print-procedure print-datatype-instance))
394.    descriptor
395.    type-name
396.    constructor
397.    args)
```

```
;;-------------------------------------------------
;; match.scm
;;; DEFINE-DATATYPE and MATCH

;;; Log:
;;;
;;; reistad 8/19/94 -- Taken from /zu/lyn/6821/fall-91/code/mini-fx.scm
;;;          to replace original Scheme+ implementation with define-syntax.
;;;        Now use low level macros like in the past.
;;;        All global refs in macro output code protected with (get-real <id>)
;;;          which ensures that they cannot be shadowed by user definitions.
;;;
;;; reistad & lyn 8/23/94 (or thereabouts) Redesigned the MATCH desugaring
;;;      to look a lot nicer.  Previous version was verbose and
;;;      tended to shift to the right a lot.
;;;
;;; reistad & lyn 8/24/94 Designed and implemented a WITH-FAIL construct
;;;      that captures the failure continuation at the beginning of
;;;      a MATCH clause.  Brian extends to handle duplicated pattern
;;;      by doing a renaming prepass on the pattern and using WITH-FAIL.
;;;
;;; lyn 9/09/94 -- Fixed RENAME-IN-QUASIQUOTE to handle `_' correctly
;;;      within QUASIQUOTED patterns. They were getting renamed when
;;;      they shouldn't have been.
;;;
;;; lyn 9/10/94 -- Fixed RENAME-IN-QUASIQUOTE to correctly handle
;;;      + non-symbol atoms (e.g., null, numbers, strings, etc.)
;;;      + deconstructions (i.e., applications of constructors in
;;;        pattern position)
;;;    Previous version would try to rename these and barf if it
;;;      found duplicates!
;;;    I'm still somewhat confused because I don't understand why
;;;      renamer seems to get called twice on every pattern!


;;;
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$
;;; MATCH Begin

;;; This is not the most efficient possible strategy for implementing
;;; ML-style pattern matching.
```

```
434.        ;;;
435.        ;;; (match foo ((make-foo x y) ...) ...)
436.        ;;;   ==>  (~match-make-foo foo 2 (lambda (x y) ...) (lambda () ...))
437.        ;;;
438.        ;;; JAR sez: Don't try to understand the implementation of this macro
    without
439.        ;;; proper supervision.

440.        ;;; Here are the desugaring rules used below
441.        ;;;
442.        ;;; ------------------------------------------------------
443.        ;;; (match e (pat_1 e_1) ... (pat_n e_n))
444.        ;;;
445.        ;;; -> expand(e,
446.        ;;;           pat_1, ..., pat_n
447.        ;;;           e_1, ... e_n)
448.        ;;;
449.        ;;; ------------------------------------------------------
450.        ;;; expand(e,
451.        ;;;        pat_1, ..., pat_n
452.        ;;;        e_1, ... e_n)
453.        ;;;
454.        ;;;
455.        ;;; -> (error "Match -- no pattern matched"), n = 0
456.        ;;;
457.        ;;; -> (let ((id e))                ; id is fresh
458.        ;;;      expand-pattern(pat_1
459.        ;;;                     id
460.        ;;;                     e_1
461.        ;;;                     expand(id,
462.        ;;;                            pat_2, ... pat_n
463.        ;;;                            e_2, ..., e_n)))  else
464.        ;;;
465.        ;;; ------------------------------------------------------
466.        ;;;
467.        ;;; expand-pattern(pat, v, succ-exp, fail-exp)  ; Note v is always an id
468.        ;;;
469.        ;;; -> succ-exp, pat = _
470.        ;;;
471.        ;;; -> (if (=pat v) succ-exp fail-exp), pat a literal
472.        ;;;                              ; Note this is *only* place where failure
    expression
473.        ;;;                              ; can be evaluated, which makes sense, since
474.        ;;;                              ; it's the only place that can really detect
475.        ;;;                              ; a mismatch
476.        ;;;
```

```
;;; -> (let ((pat v)) succ-exp), pat a variable
;;;
;;; -> (e v (lambda (id_1 ... id_n) ; where id_i = pat_i, pat_i is a variable
;;;                                 ;             is fresh, otherwise
;;;          expand-sub-patterns(pat_1, ..., pat_n,
;;;                              id_1, ... id_n,
;;;                              succ-exp,
;;;                              fail-exp))
;;;        (lambda () ,fail-exp)), pat = (e pat_1 ... pat_n)
;;;
;;; ------------------------------------------------------
;;;
;;; expand-sub-patterns(pat_1, ..., pat_n,
;;;                     id_1, ... id_n,
;;;                     succ-exp,
;;;                     fail-exp)
;;;
;;; -> succ-exp, n = 0
;;;
;;; -> expand-pattern(pat_1
;;;                   id_1
;;;                   expand-sub-patterns(pat_2, ..., pat_n,
;;;                                       id_2, ..., id_n,
;;;                                       succ-exp,
;;;                                       fail-exp)
;;;                   (fail-exp))
;;;
;;;--------------------------------------------------------------------
;;; Example (note extra error checking and space-saving code
;;;          optimizations not necessarily implied by above rules:
;;;
;;;  (match x
;;;    ((foo~ a (bar~ b)) (list a b))
;;;    (_ '())))
;;;
;;;  -- desugars to -->
;;;
;;; (let
;;;  ((#fail-254 (lambda () ()))) ; Why isn't the second () quoted here?
;;;  (foo~
;;;   x
;;;   (lambda
;;;    #success-arg-256
;;;    (if
;;;     (not (= (*mini-fx-length* #success-arg-256) 2))
;;;     (*mini-fx-success-number-of-args-mismatch* '(a (bar~ b)) 2)
```

```
523.          ;;;    (apply
524.          ;;;     (lambda
525.          ;;;      (a #temp-255)
526.          ;;;      (bar~
527.          ;;;       #temp-255
528.          ;;;       (lambda
529.          ;;;        #success-arg-257
530.          ;;;        (if
531.          ;;;         (not (= (*mini-fx-length* #success-arg-257) 1))
532.          ;;;         (*mini-fx-success-number-of-args-mismatch* '(b) 1)
533.          ;;;         (apply (lambda (b) (list a b)) #success-arg-257)))
534.          ;;;       #fail-254))
535.          ;;;     #success-arg-256)))
536.          ;;;  #fail-254))
```

```
;;; ----------------------------------------------------------------------
;;; The MATCH macro

(define-scheme+-macro (match thing . clauses)

(define (expand thing clauses)
(if (null? clauses)
`(scheme+/runtime-error "MATCH -- no pattern matched to disc." ,thing)
(let ((clause (car clauses)))
(define (make-matcher disc clauses)
`(scheme+/match-clauses ,disc
                        '(match ,thing ,@clauses)
                        ,@(map (expand-clause disc) clauses)))
(if (not (pair? thing))
;; Optimization where discriminant is a literal or variable
(make-matcher thing clauses)
;; Name discriminate so that only evaluate once
(let ((thing-gensym (scheme+/gensym 'thing)))
`(LET ((,thing-gensym ,thing))
,(make-matcher thing-gensym clauses)))))))


(define (expand-clause thing)
(lambda (clause)
;; should return a failure acceptor
(let* ((failure (scheme+/gensym 'fail))
(make-failure-acceptor (lambda (body)
                         `(lambda (,failure) ,body))))
(cond ((with-fail? clause)
(make-failure-acceptor (expand-with-fail thing clause failure)))
((and (pair? clause)
(pair? (cdr clause))
(null? (cddr clause)))
(make-failure-acceptor
(expand-pattern-top (car clause) thing (cadr clause) failure)))
(else (scheme+/syntax-error "Invalid match clause syntax: " clause))))))

;; Original version without user-level capture of the failure continuation
;  (define (expand-clause thing clause fail-exp)
;    (if (and (pair? clause)
;             (pair? (cdr clause))
;             (null? (cddr clause)))
;        (expand-pattern (car clause) thing (cadr clause) fail-exp)
;        (scheme+/syntax-error "Invalid match clause syntax" clause)))
```

```
556.        ;; With-fail allows the user to capture the failure continuation.  Ie,
557.        ;; (match <foo>
558.        ;;   ((add e e) <body>))
559.        ;;
560.        ;; would be written:
561.        ;;
562.        ;; (match <foo>
563.        ;;   ((add e e.2)
564.        ;;    (with-fail
565.        ;;       (lambda (fail)
566.        ;;          (if (equal? e e.2)
567.        ;;              <body>
568.        ;;              (fail))))))
569.        ;;
570.        ;; Idea for new sugar on-top of with-fail (trevor)
571.        ;; (<pat> (where <exp_test> <exp_body))
572.        ;; ==> (<pat> (with-fail (lambda (fail) (if <exp_test> <exp_body>
    (fail)))))
573.        ;;
574.        (define (with-fail? clause)
575.        (and (pair? clause)
        a.  (pair? (cdr clause))
        b.  (null? (cddr clause))
        c.  (let ((body (cadr clause)))
        d.  (and (pair? body)
                 i.  (pair? (cdr body))
                ii.  (null? (cddr body))
               iii.  (eq? (car body) 'with-fail)))))

576.        (define (fail-body clause)
577.        (cadr (cadr clause)))

578.        (define (expand-with-fail thing clause fail-name)
579.        ;; Be sure to call expand-pattern-top to handle
580.        ;; duplicate pattern variables.
581.        (expand-pattern-top (car clause)
                    1.  thing
                    2.  `(scheme+/handle-with-fail ,(fail-body clause)
                                        1.  ,fail-name)
                    3.  fail-name))
582.        ;; End with-fail.


583.        ;; expand-pattern-top: entry point for pattern expansion
584.        ;;   First grovell pat to find duplicate pattern variables so we can
585.        ;;   use with-fail to make them do the right thing.
```

```
(define (expand-pattern-top pat thing succ-exp fail-name)
(with-values (lambda () (rename-pattern-variables pat empty-dict))
(lambda (new-pat dict)
(let ((dict (list-transform-negative dict used-once?)))
  (if (null? dict)
  (expand-pattern pat thing succ-exp fail-name)
  ;; Note: could just call expand-pattern directly as fail-name
  ;; is in scope:
  ;; (expand-pattern new-pat thing
  ;;              (if (and ,@(gen-check-dups dict))
  ;;                  ,succ-exp
  ;;                  (,fail-name))
  ;;              fail-name)
  (expand-with-fail
      thing
      (let ((fail-gensym (scheme+/gensym 'fail)))
      `(,new-pat (with-fail
                  (lambda (,fail-gensym)
                      (if (and ,@(gen-check-dups dict))
                      ,succ-exp
                      (,fail-gensym))))))
      fail-name))))))

(define (rename-pattern-variables pat dict)
;; Rename duplicate pattern variables and
;; use values to return new pattern and a dictionary of pattern variables
;; Eg. (mul (add e e) (var x)) ==> (values (add e e.2) ((e e.2) (x)))
(cond ((eq? pat '_) (values pat dict))
  ((symbol? pat)
  (if (in-dict? pat dict)
      (let ((new-var (scheme+/gensym pat)))
      (extend-entry! pat new-var dict)
      (values new-var dict))
      (values pat (add-entry pat dict))))
  ((not (pair? pat)) (values pat dict))     ;; constant pattern

  ((or (eq? (car pat) 'symbol)
      (eq? (car pat) 'quote))
  (values pat dict))
  ((eq? (car pat) 'quasiquote)
  (with-values (lambda () (rename-in-quasiquote (cadr pat) dict))
  (lambda (new-cadr dict)
      (values `(,'quasiquote ,new-cadr) dict))))
  (else
  ;; (constructor pat1 pat2 ... patn)
  (with-values (lambda () (rename-list (cdr pat) dict))
```

```
    l.  (lambda (patterns dict)
            i.  (values `(,(car pat) ,@patterns) dict))))
    m.  ))

595.    (define empty-dict '())
596.    (define (in-dict? x dict) (assoc x dict))
597.    (define (add-entry x dict) `((,x) ,@dict))
598.    (define (extend-entry! x new dict)
599.    (let ((entry (assoc x dict)))
600.    (set-cdr! entry (cons new (cdr entry)))
601.    ))
602.    (define (used-once? dict-entry)
603.    (= (length dict-entry) 1))

604.    (define (gen-check-dups dict)
605.    (if (null? dict)
606.    '()
607.    `(,(gen-ensure-equal (car dict)) ,@(gen-check-dups (cdr dict)))))
608.    (define (gen-ensure-equal lst)
609.    (let ((first (car lst)))
610.    `(and ,@(let loop ((rest (cdr lst)))
                i.  (if (null? rest)
               ii.  '()
              iii.  ;;; Lyn changed on 9/10/94 to install scheme+/equal?
               iv.  ; `(((get-real equal?) ,first ,(car rest))
                v.  ;   ,@(loop (cdr rest)))
               vi.  `((scheme+/equal? ,first ,(car rest))
              vii.  ,@(loop (cdr rest))))))))

611.    (define (rename-list lst dict)
612.    (if (null? lst)
613.    (values '() dict)
614.    (with-values (lambda () (rename-pattern-variables (car lst) dict))
    a.  (lambda (new-head dict)
    b.  (with-values (lambda () (rename-list (cdr lst) dict))
    c.  (lambda (new-tail dict)
            i.  (values (cons new-head new-tail) dict)))))))

615.    (define (rename-in-quasiquote body dict)
616.    (define (descend-quasi x level dict)
617.    (cond ((eq? x '_)                       ; Handle _ specially -- don't rename!
    a.  (values x dict))
    b.  ((symbol? x)            ; Level 0 symbols get added to dict
    c.  (if (= level 0)
            i.  (if (in-dict? x dict)
           ii.  (let ((tmp (scheme+/gensym x)))
```

```
            iii. (extend-entry! x tmp dict)
             iv. (values tmp dict))
              v. (values x (add-entry x dict)))
             vi. (values x dict)))
        d. ((not (pair? x))          ; Other atomic data (including empty list)
        e. (values x dict))          ; are treated as constants.
        f. ((and (not (null? (cdr x))) (null? (cddr x))) ; List of length 2
        g. (case (car x)
              i. ((unquote)
             ii. (if (= level 0)
            iii. (scheme+/syntax-error "unquote too deep")
             iv. (with-values (lambda ()
                                a. (descend-quasi (cadr x) (- level 1) dict))
              v. (lambda (new dict)
                   1. (values `(,'unquote ,new) dict)))))
             vi. ((unquote-splicing)
            vii. (if (= level 0)
           viii. (scheme+/syntax-error "unquote-splicing too deep")
             ix. (with-values (lambda ()
                                a. (descend-quasi (cadr x) (- level 1) dict))
              x. (lambda (new dict)
                   1. (values `(,'unquote-splicing ,new) dict)))))
             xi. ((quasiquote)
            xii. (with-values (lambda ()
                                a. (descend-quasi (cadr x) (+ level 1) dict))
           xiii. (lambda (new dict)
            xiv. (values `(,'quasiquote ,new) dict))))
             xv. ((quote)
            xvi. (with-values (lambda ()
                                a. (descend-quasi (cadr x) level dict))
          xvii. (lambda (new dict)
         xviii. (values `(,'quote ,new) dict))))
             xix. (else
              xx. (descend-quasi-deconstruction x level dict))))
        h. (else (descend-quasi-deconstruction x level dict))))
618.    (define (descend-quasi-deconstruction lst level dict)
619.    ;;
620.    ;; LST must be non-empty.
621.    ;;
622.    ;; If LEVEL is 0, then LST is a deconstruction and the car of LST
623.    ;; is a constructor. The constructor should not be renamed, but
624.    ;; its arguments should be.
625.    ;;
626.    ;; If LEVEL is not 0, treat LST as a regular list.
627.    ;;
628.    (if (= level 0)
```

```
        a. (with-values (lambda () (descend-quasi-list (cdr lst) level dict))
        b. (lambda (new-args dict)
        c. (values (cons (car lst) ; The constructor
                         1. new-args)
                    ii. dict)))
        d. (descend-quasi-list lst level dict)))
629.       (define (descend-quasi-list x level dict)
630.       (if (null? x)
        a. (values '() dict)
        b. (with-values (lambda () (descend-quasi (car x) level dict))
        c. (lambda (first dict)
        d. ;BJR: change to descend-quasi to handle dotted pairs
        e. ;(with-values (lambda() (descend-quasi-list (cdr x) level dict)))
        f. (with-values (lambda () (descend-quasi (cdr x) level dict))
                i. (lambda (rest dict)
               ii. (values (cons first rest) dict)))))))
631.       ; (trace-entry descend-quasi)
632.       ; (trace-entry descend-quasi-list)
633.       ; (trace-entry descend-quasi-deconstruction)
634.       (descend-quasi body 1 dict)
635.       )
636.       ;; End rename-pattern-variables


637.       ; succ-exp is an expressions
638.       ; fail-name is a symbol -- name of the current failure continuation
639.       (define (expand-pattern pat thing succ-exp fail-name)
640.       (let ((fail-exp `(,fail-name)))
641.       (cond ((eq? pat '_) succ-exp)
        a. ((symbol? pat)
        b. (if (eq? pat thing)
                i. succ-exp          ;Optimization for same variable that works
                            a. ;in conjunction with optimization for naming
                            b. ;success lambda args below
               ii. `(let ((,pat ,thing)) ,succ-exp)))

        c. ;; Don't make any assumptions about thing.
642.       ;             ((number? pat)
643.       ;              `(if ((get-real =) ,thing ,pat) ,succ-exp ,fail-exp))
644.       ;             ((boolean? pat)
645.       ;              `(if ((get-real eq?) ,thing ,pat) ,succ-exp ,fail-exp))
646.       ;             ((char? pat)
647.       ;              `(if ((get-real char=?) ,thing ,pat) ,succ-exp ,fail-exp))
648.       ;             ((string? pat)
649.       ;              `(if ((get-real string=?) ,thing ,pat) ,succ-exp ,fail-exp))
```

```
a.  ((number? pat)
b.  `(if ((get-real equal?) ,thing ,pat) ,succ-exp ,fail-exp))
c.  ((boolean? pat)
d.  `(if ((get-real equal?) ,thing ,pat) ,succ-exp ,fail-exp))
e.  ((char? pat)
f.  `(if ((get-real equal?) ,thing ,pat) ,succ-exp ,fail-exp))
g.  ((string? pat)
h.  `(if ((get-real equal?) ,thing ,pat) ,succ-exp ,fail-exp))

i.  ((not (pair? pat))
j.  (scheme+/syntax-error "unrecognized MATCH pattern: " pat))
k.  ((eq? (car pat) 'symbol)
l.  `(if ((get-real eq?) ,thing ,pat) ,succ-exp ,fail-exp))
m.  ((eq? (car pat) 'quote)
n.  (let ((pred (if (symbol? (cadr pat))
                    1. '(get-real eq?) ;Optimization
                    2. '(get-real equal?))))
        ii. `(if (,pred ,thing ,pat) ,succ-exp ,fail-exp)))
o.  ((eq? (car pat) 'quasiquote)
p.  (expand-pattern (expand-quasiquote (cadr pat) 0)
                    1. thing succ-exp fail-name))
q.  (else
r.  (expand-compound-pattern pat thing succ-exp fail-name)))))

650.    ; Expand a pattern of the form (op arg ...).
651.    ; op is assumed to be an injection or construction procedure.
652.    ; If it's not, you'll get a Scheme error of the form
653.    ; "unbound variable ~MATCH-OP".

654.    (define (expand-compound-pattern pat thing succ-exp fail-name)
655.    (let ((number-of-sub-patterns (length (cdr pat)))
a.  (success-arg-gensym (scheme+/gensym 'success-arg))
b.  (names (map (lambda (pat)
                    1. (if (and (symbol? pat) (not (eq? pat '_)))
                    2. pat              ;Optimization that works in conjunction
                            i. ;with variable case for EXPAND-
                               PATTERN
                    3. (scheme+/gensym 'temp)))
        ii. (cdr pat)))
c.  )
656.    `(
657.    scheme+/deconstruct-carefully
658.    ,(car pat)
659.    ',(cdr pat)
660.    ,number-of-sub-patterns
661.    ,thing
```

```
662.        (LAMBDA ,names
        a.  ,(let expand-sub-patterns ((pats (cdr pat))
                                       a.  (names names))
        b.  (if (null? pats)
                i.  succ-exp
               ii.  (expand-pattern (car pats)
                                       a.  (car names)
                                       b.  (expand-sub-patterns (cdr pats) (cdr names))
                                       c.  fail-name))))
663.        ,fail-name)))


664.        ; This is JAR's quasiquote pattern handler, which is exceptionally clever.
665.        ; It tries to avoid the desugaring into CONS~
666.        ; unless it absolutely has to.  Here are some examples:
667.        ;
668.        ; `(a b) -> '(a b)
669.        ; `(,a b) -> (list->sexp~ (cons~ a '(b)))
670.        ; `(a ,b) -> (list->sexp~ (cons~ 'a (cons~ b '())))
671.        ; `(,a ,b) -> (list->sexp~ (cons~ a (cons~ b '())))
672.        ; `(,a ,@b) -> (list->sexp~ (cons~ a b))


673.        ;;---------------------------------------------------------------------
674.        ;; LYN'S NOTES ON QUOTATION
675.        ;
676.        ; What I had in mind is expressed by the following rewrite rules
677.        ;
678.        ;   (quasiquote (unquote ?a)) => ?a
679.        ;
680.        ;   (quasiquote ((unquote-splicing ?a) ?extra ?rest ...) => Error! illegal ,@
681.        ;
682.        ;   (quasiquote ((unquote-splicing ?a)) => ?a
683.        ;
684.        ;   (quasiquote (unquote-splicing ?a)) => Error! illegal ,@
685.        ;
686.        ;   (quasiquote (?a . ?b)) => (cons (quasiquote ?a) (quasiquote ?b))
687.        ;
688.        ;   (quasiquote ?a) => (quote ?a)
689.        ;
690.        ; (These rules must be applied in the order shown to get the precedence
691.        ; right.)
692.        ;
693.        ; These rules treat UNQUOTE and UNQUOTE-SPLICING specially in
    the
694.        ; context of a quasiquote.  Granted, this can give some weird effects;
```

```
695.    ; the following are transcripts of the pattern matching implementation of
696.    ; Scheme+:
697.    ;
698.    ; ;;;-------------------------------------------------
699.    ;   (define x 17)
700.    ;   (define y '(23))
701.    ;
702.    ;   (match '(1 . 2)
703.    ;     (`(,x . ,y) (list y x)))
704.    ;   ;Value 56: (2 1)
705.    ;
706.    ;   (match '(1 . 2)
707.    ;     (`(,x unquote y) (list y x)))
708.    ;   ;Value 57: (2 1)
709.    ;
710.    ;   (match '(1 . 2)
711.    ;     (`(,x unquote y) (list y x)))
712.    ;   ;Value 57: (2 1)
713.    ;
714.    ;   (match '(1 unquote y)
715.    ;     (`(,x unquote y) (list y x)))
716.    ;   ;Value 58: (,y 1)
717.    ;
718.    ;   (match '(1 . 2)
719.    ;     (`(,x unquote y z) (list y x)))
720.    ;   ;No pattern matched!
721.    ;   ;To continue, call RESTART with an option number:
722.    ;
723.    ;   (match '(1 unquote y z)
724.    ;     (`(,x unquote y z) (list y x)))
725.    ;   ;Value 64: ((23) 1)  ; The (23) comes from definition of Y at top.
726.    ;
727.    ;   (match '(1 2)
728.    ;     (`(,x ,@y) (list y x)))
729.    ;   ;Value 75: ((2) 1)
730.    ;
731.    ;   (match '(1 2)
732.    ;     (`(,x . ,@y) (list y x)))
733.    ;   ;Illegal use of ,@ in pattern
734.    ;   ;To continue, call RESTART with an option number:
735.    ;   ; (RESTART 1) => Return to read-eval-print level 1.
736.    ;
737.    ;   (match '(1 2)
738.    ;     (`(,x unquote-splicing y) (list y x)))
739.    ;   ;Illegal use of ,@ in pattern
740.    ;   ;To continue, call RESTART with an option number:
```

```
741.        ;
742.        ;   (match '(1 unquote-splicing y z)
743.        ;     (`(,x unquote-splicing y z) (list y x)))
744.        ;   ;Value 76: ((23) 1)  ; The (23) comes from definition of Y at top.
745.        ;
746.        ;
747.        ; ;;;-----------------------------------------------
748.        ;
749.        ; The quirkiness is exhibited only with the symbols UNQUOTE and
750.        ; UNQUOTE-SPLICING in a QUASIQUOTE context; all other symbols
    behave as
751.        ; expected. This is OK (IMHO) because these are part of the quasiquote
752.        ; language.  Even more convincing, the above interpretation is consistent
753.        ; with what Scheme (at least MIT Scheme) does with QUASIQUOTE in a
754.        ; non-pattern context. Observe:
755.        ;
756.        ;   ;; Assume X is 17 and Y is (23)
757.        ;
758.        ;   `(,x . ,y)
759.        ;   ;Value 81: (17 23)
760.        ;
761.        ;   `(,x unquote y)
762.        ;   ;Value 78: (17 23)
763.        ;
764.        ;   `(,x unquote y z)
765.        ;   ;Value 77: (17 unquote y z)
766.        ;
767.        ;   `(,x ,@y)
768.        ;   ;Value 79: (17 23)
769.        ;
770.        ;   `(,x . ,@y)
771.        ;   ;Syntax error: ,@ in illegal context: y
772.        ;   ;To continue, call RESTART with an option number:
773.        ;
774.        ;   `(,x unquote-splicing y)
775.        ;   ;Syntax error: ,@ in illegal context: y
776.        ;   ;To continue, call RESTART with an option number:
777.        ;
778.        ;   `(,x unquote-splicing y z)
779.        ;   ;Value 80: (17 unquote-splicing y z)
780.        ;
781.        ;
782.        ; The only inconsistent usage is the treatment of ,@ in a pattern context
783.        ; to avoid backtracking.
784.        ;
785.        ;;-------------------------------------------------------------
```

```
;; NOTE: these procedures are all written with names using 'deconstructor'
;; which is an out-of-date term -- the correct terminology is 'unbunlder'.

(define (expand-quasiquote x level)
(descend-quasiquote x level finalize-quasiquote))

(define (descend-quasiquote x level return)
(cond ((not (pair? x))          ; Includes null?
(return 'quote x))
((and (not (null? (cdr x))) (null? (cddr x))) ; List of length 2
(case (car x)
((unquote)
(if (= level 0)
    (return 'unquote (cadr x))
    (descend-interesting x (- level 1) unquoted->sexp return)))
((unquote-splicing)
(if (= level 0)
    (return 'unquote-splicing (cadr x))
    (descend-interesting x (- level 1) unquoted-splicing->sexp return)))
((quasiquote)
(descend-interesting x (+ level 1) quasiquoted->sexp return))
((quote)
(descend-interesting x level quoted->sexp return))
(else
(descend-quasiquote-list x level return))))
(else (descend-quasiquote-list x level return))))

(define (descend-interesting x level inject return)
(descend-quasiquote (cadr x) level
(lambda (mode arg)
(if (eq? mode 'quote)
(return 'quote x)
(return 'unquote `(,inject ,(finalize-quasiquote mode arg)))))))

(define (descend-quasiquote-list x level return)
(descend-quasiquote-tail x level
(lambda (mode arg)
(if (eq? mode 'quote)
(return 'quote x)
;;BJR: list-sexp is identity and only needed for typing concerns
;; (return 'unquote `(,list->sexp ,arg))
(return 'unquote arg)))))

; Modified below to handle dotted pairs.
;   (define (descend-quasiquote-tail x level return)
```

```
802.        ;    (if (null? x)
803.        ;        (return 'quote x)
804.        ;        (descend-quasiquote-tail (cdr x) level
805.        ;          (lambda (cdr-mode cdr-arg)
806.        ;            (descend-quasiquote (car x) level
807.        ;              (lambda (car-mode car-arg)
808.        ;                  (cond ((and (eq? car-mode 'quote) (eq? cdr-mode 'quote))
809.        ;                         (return 'quote x))
810.        ;                        ((eq? car-mode 'unquote-splicing)
811.        ;                         (cond ((and (eq? cdr-mode 'quote) (null? cdr-arg))
812.        ;                                ;; (,@mumble)
813.        ;                                (return 'unquote car-arg))  ;Type must be a list!
814.        ;                               (else (scheme+/syntax-error "Illegal use of @ in
    a quasiquoted pattern."))))
815.        ;                        ; JAR allowed the following, but doesn't make sense
816.        ;                              ; without backtracking.
817.        ;                              ; (else
818.        ;                              ;  ;; (,@mumble ...)
819.        ;                              ;  (return 'unquote
820.        ;                              ;          `(,append-word ,car-arg
821.        ;                              ;                         ,(finalize-quasiquote
822.        ;                              ;                        cdr-mode cdr-arg))))
823.        ;                        (else
824.        ;                         (return 'unquote
825.        ;                                 ;; what is cons-word supposed to be?
826.        ;                                 `(,cons ,(finalize-quasiquote car-mode car-arg)
827.        ;                                         ,(finalize-quasiquote cdr-mode
    cdr-arg))))))))))))
```

```
828.        (define (descend-quasiquote-tail x level return)
829.        (if (null? x)
830.        (return 'quote x)
831.        ;; BJR&LYN: This handles dotted-pair too!
832.        (descend-quasiquote (cdr x) level
        a. (lambda (cdr-mode cdr-arg)
        b. (descend-quasiquote (car x) level
        c. (lambda (car-mode car-arg)
                i. (cond ((and (eq? car-mode 'quote) (eq? cdr-mode 'quote))
               ii. (return 'quote x))
              iii. ((eq? car-mode 'unquote-splicing)
               iv. (cond ((and (eq? cdr-mode 'quote) (null? cdr-arg))
                       1. ;; (,@mumble)
                       2. (return 'unquote car-arg)) ;Type must be a list!
                       3. (else (scheme+/syntax-error "Illegal use of @ in a
                          quasiquoted pattern."))))
                v. ((and (eq? cdr-mode 'unquote-splicing)
```

1. (not (pair? (cdr x))))
vi. ;; (foo . ,@x)
vii. (scheme+/syntax-error
1. "Illegal use of @ in a quasiquoted pattern."))
viii. ; JAR allowed the following, but doesn't make sense
ix. ; without backtracking.
x. ; (else
xi. ; ;; (,@mumble ...)
xii. ; (return 'unquote
xiii. ; `(,append-word ,car-arg
xiv. ; ,(finalize-quasiquote
xv. ; cdr-mode cdr-arg))))
xvi. (else
xvii. (return 'unquote
a. `(,cons ,(finalize-quasiquote car-mode car-arg)
i. ,(finalize-quasiquote cdr-mode cdr-arg))))))))))

833. (define (finalize-quasiquote mode arg)
834. (case mode
835. ((quote) `',arg)
836. ((unquote) arg)
837. ((unquote-splicing)
838. (scheme+/syntax-error ",@ in illegal context: " arg)) ;`,@x or ``,,@x or `(y . ,@x)
839. (else
840. (scheme+/syntax-error "quasiquote bug: " (list mode arg)))))

841. (define (non-atomic-deconstructors clauses)
842. (remove-duplicates
843. (mapcan (lambda (c)
i. (filter
ii. pair?
iii. (deconstructors (match-clause-pattern c))))
b. clauses)))

844. (define (deconstructors pat)
845. (cond ((not (pair? pat)) '())
a. ((quote? pat) '())
b. ((quasiquote? pat)
c. (mapcan deconstructors
i. (quasiquote-embedded-expressions (quasiquote-text pat))))
d. (else
e. (cons (car pat)
i. (mapcan deconstructors (cdr pat))))))

```
(define (filter pred lst)
(cond ((null? lst) '())
  ((pred (car lst))
  (cons (car lst) (filter pred (cdr lst))))
  (else (filter pred (cdr lst)))))

(define (mapcan proc lst)
(if (null? lst)
'()
(append (proc (car lst))
      (mapcan proc (cdr lst)))))

(define (remove-duplicates lst)
(if (null? lst)
'()
(let ((result (remove-duplicates (cdr lst))))
  (if (member (car lst) result)
  result
  (cons (car lst) result)))))

(define (quasiquote-embedded-expressions exp)
(let descend-quasiquote ((exp exp)
                         (level 0))
(cond ((not (pair? exp)) '())
  ((quote? exp) '())
  ((quasiquote? exp)
  (descend-quasiquote (quasiquote-text exp) (+ level 1)))
  ((unquote? exp)
  (if (= level 0)
      (list (unquote-text exp))
      (descend-quasiquote (unquote-text exp) (- level 1))))
  ((unquote-splicing? exp)
  (if (= level 0)
      (list (unquote-splicing-text exp))
      (descend-quasiquote (unquote-splicing-text exp) (- level 1))))
  (else (mapcan (lambda (e) (descend-quasiquote e level)) exp))
  )))

(define (clauses-deconstructor-subst clauses subst)
(map (lambda (clause)
  (cons (pattern-deconstructor-subst (car clause) subst)
      (cdr clause)))
  clauses))

(define (pattern-deconstructor-subst pat subst)
```

```
862.        (cond ((not (pair? pat)) pat)
        a.  ((quote? pat) pat)
        b.  ((quasiquote? pat)
        c.  (quasiquote-deconstructor-subst (quasiquote-text pat) subst))
        d.  (else (cons (subst (car pat))
                    i.  (map (lambda (p)
                                1.  (pattern-deconstructor-subst p subst))
                                2.  (cdr pat))))))

863.        (define (quasiquote-deconstructor-subst pat subst)
864.        (let descend-quasiquote ((exp pat)
                                1.  (level 0))
865.        (cond ((not (pair? exp)) exp)
        a.  ((quote? exp) exp)
        b.  ((quasiquote? exp)
        c.  (quasiquote-make
        d.  (descend-quasiquote (quasiquote-text exp) (+ level 1))))
        e.  ((unquote? exp)
        f.  (unquote-make
        g.  (if (= level 0)
                    i.  (pattern-deconstructor-subst
                   ii.  (unquote-text exp)
                  iii.  subst)
                   iv.  (descend-quasiquote (unquote-text exp) (- level 1)))))
        h.  ((unquote-splicing? exp)
        i.  (unquote-splicing-make
        j.  (if (= level 0)
                    i.  (pattern-deconstructor-subst
                   ii.  (unquote-splicing-text exp)
                  iii.  subst)
                   iv.  (descend-quasiquote (unquote-splicing-text exp) (- level 1)))))
        k.  (else (map (lambda (e) (descend-quasiquote e level)) ex)))))

866.        (define match-clause-pattern first)

867.        (define (predicate sym)
868.        (lambda (exp)
869.        (if (pair? exp)
        a.  (eq? (car exp) sym)
        b.  #f)))

870.        ; Quoting
871.        (define quote? (predicate 'quote))
872.        (define (quote-make text) (list 'quote text))
873.        (define quote-text second)
874.        (define quasiquote? (predicate 'quasiquote))
```

```
875.    (define (quasiquote-make text) (list 'quasiquote text))
876.    (define quasiquote-text second)
877.    (define unquote? (predicate 'unquote))
878.    (define (unquote-make text) (list 'unquote text))
879.    (define unquote-text second)
880.    (define unquote-splicing? (predicate 'unquote-splicing))
881.    (define (unquote-splicing-make text) (list 'unquote-splicing text))
882.    (define unquote-splicing-text second)


883.    ;; Main body of the MATCH macro
884.    (expand thing clauses)

885.    ) ; End MATCH macro expander
```

```
886.        (define (scheme+/match-clauses disc match-exp . rest)
887.        (let loop ((clauses rest))
888.        (if (null? clauses)
889.        (scheme+/runtime-error "MATCH -- no pattern matched to disc."
                                        a.  disc match-exp)
890.        ((car clauses)
        a.  (lambda ()
        b.  (loop (cdr clauses)))))))

891.        ;; Procedure for doing deconstruction out-of-line
892.        (define (scheme+/deconstruct-carefully
        a.  constr sub-patterns nargs thing succ-cont fail-cont)
893.        ((unbundler constr)
894.        thing
895.        (lambda success-args
896.        (if (not (= (length success-args) (length sub-patterns)))
        a.  (scheme+/success-number-of-args-mismatch sub-patterns
                                                    1.  success-args)
        b.  (apply succ-cont success-args)))
897.        fail-cont))


898.        (define (scheme+/handle-with-fail fail-acceptor fail-cont)
899.        (let ((token (list 'fail-token)))
900.        ;; fail-acceptor is the body of the with-fail
901.        ;; fail-cont is the failure continuation for this clause
902.        ;;OLD: fail-body is a failure acceptor, so just apply it to the
903.        ;;     current failure: (fail-acceptor fail-cont)
904.        ;;BJR: need to really abort clause so grab continuation
905.        (let ((return-value
        a.  (call-with-current-continuation
        b.  (lambda (k)
        c.  (fail-acceptor
                i.  ;; abort by returning failure continuation
               ii.  (lambda () (k token)))))))
906.        (if (eq? return-value token)
        a.  ;; we're aborting
        b.  (fail-cont)
        c.  return-value))))

907.        ;;; MATCH End
908.        ;;;
    $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
    $$$$$$$$
```

```
;;;
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$
;;; SCHEME+/GENSYM

(define scheme+/gensym
(let ((counter 0))
(lambda (sym)
(let ((result (string->symbol
    i.  ;; Symbols beginning with # aren't recognized by reader,
    ii. ;; so this can't conflict with a user-specified name.
    iii. (string-append "#"
                a. (symbol->string sym)
                b. "-"
                c. (number->string counter)))))
(set! counter (+ counter 1))
result))))
;;;
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$
```

```
(define (scheme+/success-number-of-args-mismatch patterns exps)
(scheme+/runtime-error
(string-append "the number of sub-patterns in a match clause ("
    (number->string (length patterns))
    ") is not equal to \nthe number of fields in a deconstructed datatype object ("
    (number->string (length exps))
    "). \nTried to match the following patterns and expressions:\n")
(list 'patterns: patterns 'exps: exps)))
```

```
;;-----------------------------------------------
;; top-level.scm
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Scheme+ Top-Level
;;;
;;; Author:       Lyn
;;; Creation Date: 6/23/94
;;; Log:
;;; * 8/19/94 (reistad): Reworked to use low level macros.
;;; * 8/11/94 (lyn): Adapted Mini-FX SF, CF, and path stuff to Scheme+
;;;   Also added an explicit Scheme+ interactive environment.
;;;
;;; Documentation:
;;; * Controls the integration of Scheme+ within MIT Scheme.
;;; * Based on the Mini-FX top level.
;;; * Ignores the deep issues of interactions between syntactic definitions
;;;   and first-class environments. (According to CPH, this is why
;;;   SYNTAX-RULES aren't standard in MIT Scheme.)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; History
;;;
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;------------------------------------------------------------------------
;;; TOP-LEVEL EVALUATION

;;; Loading files in Scheme+ requires special handling of unsyntaxed files.

(define (scheme+/load filename)
(load filename scheme+/interactive-environment scheme+/syntax-table))

(define (scheme+/eval exp env)
(eval (syntax exp scheme+/syntax-table) env))

;; Currently unsupported
;
(define (scheme+/sf filename . rest)
;; (sf/set-file-syntax-table! (->pathname filename) scheme+/syntax-table)
;; (apply sf filename rest)
(error "SF for Scheme+ files is currently unsupported")
)
```

```
962.        (define (scheme+/cf filename . rest)
963.        ;; (sf/set-file-syntax-table! (->pathname filename) scheme+/syntax-table)
964.        ;; (apply cf filename rest)
965.        (error "CF for Scheme+ files is currently unsupported")
966.        )


967.        ;;;----------------------------------------------------------------------------
968.        ;;; Installation of evaluator

969.        (define (scheme+/enter-top-level)

970.        ;; Change the emacs interface
971.        ;; Doesn't seem to work.
972.        ;; (scheme-runtime/install-mfx-emacs-interface!)

973.        ;; No longer needed -- reistad 8/19/94
974.        ;  (scheme-runtime/install-evaluator! (lambda (sexp env st)
975.        ;                                       (scheme+/eval sexp env)))

976.        (scheme-runtime/install-environment! scheme+/interactive-environment)
977.        (display (string-append
        a.  "\nYou are now typing at the Scheme+ interpreter (version "
        b.  scheme+/version
        c.  ")"
        d.  ))
978.        )


979.        (define (scheme+/exit-top-level)

980.        ;; No longer needed -- reistad 8/19/94
981.        ;  (scheme-runtime/install-default-evaluator!)
982.        (scheme-runtime/install-environment! user-initial-environment)
983.        (display "\nYou are now typing at the Scheme interpreter.")

984.        )

985.        ;; A new version of EQUAL? that provides equality checking of datatype
    instances
986.        ;;
987.        (define (scheme+/equal? obj1 obj2)
988.        (cond ((eq? obj1 obj2) #t)
989.        ((pair? obj1)
        a.  (if (pair? obj2)
```

```
        b. (and (scheme+/equal? (car obj1) (car obj2))
                i. (scheme+/equal? (cdr obj1) (cdr obj2)))
        c. #f))
990.    ((datatype-instance? obj1)
        a. (if (datatype-instance? obj2)
        b. (scheme+/datatype-instance-equal? obj1 obj2)
        c. #f))
991.    ((number? obj1)
        a. (if (number? obj2)
        b. (= obj1 obj2)
        c. #f))
992.    ((string? obj1)
        a. (if (string? obj2)
        b. (string=? obj1 obj2)
        c. #f))
993.    ((char? obj1)
        a. (if (char? obj2)
        b. (char=? obj1 obj2)
        c. #f))
994.    ((vector? obj1)
        a. (if (vector obj2)
        b. (and (= (vector-length obj1) (vector-length obj2))
                i. (let loop ((i (- (vector-length obj1) 1)))
               ii. (if (< i 0)
                       1. #t
                       2. (and (scheme+/equal? (vector-ref obj1 i)
                                               i. (vector-ref obj2 i))
                       3. (loop (- i 1))))))
        c. #f))
995.    (else #f)
996.    ))

997.    (define (scheme+/datatype-instance-equal? dinst1 dinst2)
        a. (and (eq? (datatype-instance-descriptor dinst1)
                i. (datatype-instance-descriptor dinst2))
        b. (eq? (datatype-instance-constructor dinst1)
                i. (datatype-instance-constructor dinst2))
        c. (scheme+/equal? (datatype-instance-args dinst1)
                       1. (datatype-instance-args dinst2))))

998.    ;; Handy synonyms

999.    (define (scheme+) (scheme+/enter-top-level))
1000.   (define (scheme)
1001.   (display "\nYou are already typing at the Scheme interpreter."))
```

```
1002.   (define (scheme+/pp . args)
1003.   ;; Version of PP that prints everything as code (nothing as tables).
1004.   (if (version-7.1.3?)
1005.   (apply pp args)
1006.   (fluid-let ((*pp-lists-as-tables?* #f))
1007.   (apply pp args))))

1008.   ;;;---------------------------------------------------------------------------
1009.   ;;; The Scheme+ Environment
1010.   ;;;
1011.   ;;; This creates a new environment on top of user-initial-environment
1012.   ;;; that contains a few Scheme+ specific bindings.

1013.   (define scheme+/interactive-environment
1014.   (make-environment
1015.   (define load scheme+/load)
1016.   (define eval scheme+/eval)
1017.   (define sf scheme+/sf)
1018.   (define cf scheme+/cf)
1019.   (define equal? scheme+/equal?)
1020.   (define user-initial-environment 'later)
1021.   (define global-eval
1022.   (lambda (exp) (eval exp scheme+/interactive-environment)))
1023.   (define scheme scheme+/exit-top-level)
1024.   (define scheme+
1025.   (lambda ()
1026.   (display "\nYou are already typing at the Scheme+ interpreter.")))
1027.   (define pp scheme+/pp)
1028.   ))

1029.   (eval `(set! user-initial-environment ,scheme+/interactive-environment)
1030.   scheme+/interactive-environment)
```

```
1031.   ;;;----------------------------------------------------------------------
1032.   ;;; Version Handling

1033.   ;;; Scheme version stuff

1034.   ; (define (runtime-system)
1035.   ;   (let ((answer '*))
1036.   ;     (begin
1037.   ;       (for-each-system!
1038.   ;        (lambda (sys)
1039.   ;           (if (string=? (system/name sys)
1040.   ;                         "Runtime")
1041.   ;               (set! answer (list (system/version sys)
1042.   ;                                  (system/modification sys))))))
1043.   ;       (if (eq? answer '*)
1044.   ;           (error "SCHEME+ INITIALIZATION ERROR: CAN'T FIND
    SCHEME RUNTIME VERSION")
1045.   ;           answer))))

1046.   (define (runtime-system)
1047.   (if (environment-bound? system-global-environment 'get-subsystem-
    version)
1048.   (get-subsystem-version "Runtime")
1049.   (let ((answer '*))
1050.   (begin
        a. (for-each-system!
        b. (lambda (sys)
        c. (if (string=? (system/name sys)
                    1. "Runtime")
               ii. (set! answer (list (system/version sys)
                           a. (system/modification sys))))))
        d. (if (eq? answer '*)
        e. (error "SCHEME+ INITIALIZATION ERROR: CAN'T FIND SCHEME
           RUNTIME VERSION")
        f. answer)))))


1051.   (define runtime-version first)
1052.   (define runtime-modification second)

1053.   (define (version-7.1.3?)
1054.   (equal? (runtime-system) '(14 104)))

1055.   (define (version-7.2?)
1056.   ;; This is a hack. This also returns #t for 7.3 systems.
```

```
1057.       ;; To discriminate for 7.3, explicitly test for it first!
1058.       ;   (or (equal? (runtime-system) '(14 155))
1059.       ;       (equal? (runtime-system) '(14 156))
1060.       ;       (equal? (runtime-system) '(14 157)))
1061.       (and (= (first (runtime-system)) 14)
1062.       (>= (second (runtime-system)) 155)))

1063.       (define (version-7.3?)
1064.       (and (= (first (runtime-system)) 14)
1065.       (>= (second (runtime-system)) 166)))

1066.       (define (version-error)
1067.       (error
1068.       (string-append
1069.       "SCHEME-+ VERSION ERROR\nScheme+ doesn't know how to handle
    runtime "
1070.       (number->string (runtime-version (runtime-system)))
1071.       "."
1072.       (number->string (runtime-modification (runtime-system)))
1073.       "\nContact 6821@psrg.lcs.mit.edu for help.")))

1074.       (define scheme-runtime/version-7.1.3/install-evaluator!
1075.       (let ((repl-env (->environment '(runtime rep))))
1076.       (lambda (evaluator) ; evaluator is SEXP x ENV x ST ->
1077.       (set! (access hook/repl-eval repl-env)
        a. ;; Version 7.1.3 evaluators take an extra first arg
        b. ;; that is ignored.
        c. (lambda (repl sexp env st)
        d. (evaluator sexp env st))))))

1078.       (define scheme-runtime/version-7.2/install-evaluator!
1079.       (let ((repl-env (->environment '(runtime rep))))
1080.       (lambda (evaluator) ; evaluator is SEXP x ENV x ST ->
1081.       (set! (access hook/repl-eval repl-env)
        a. evaluator))))

1082.       (define scheme-runtime/install-evaluator!
1083.       (cond
1084.       ((version-7.3?) scheme-runtime/version-7.1.3/install-evaluator!) ; kludge!
1085.       ((version-7.1.3?) scheme-runtime/version-7.1.3/install-evaluator!)
1086.       ((version-7.2?) scheme-runtime/version-7.2/install-evaluator!)
1087.       (else (version-error))))

1088.       ;;; This is written to be independent of version
1089.       (define scheme-runtime/install-default-evaluator!
1090.       (eval '(lambda () (set! hook/repl-eval default/repl-eval))
```

```
1091.       (->environment '(runtime rep))))


1092.       ;;; Patch for 7.1.3
1093.       (define (scheme-runtime/map proc . lsts)
1094.       (define (map proc . lsts)
1095.       (cond ((null? lsts) '())
        a.  ((null? (car lsts)) '())
        b.  ((null? (cdr lsts));; only one list to map over
        c.  (cons (proc (caar lsts))
                i.  (map proc (cdar lsts))))
        d.  (else
        e.  (cons (apply proc (map car lsts))
                i.  (apply map proc (map cdr lsts))))))
1096.       (apply map proc lsts))

1097.       (if (version-7.1.3?)
1098.       (set! map scheme-runtime/map)
1099.       'nop)


1100.       ;;; Environments

1101.       (define scheme-runtime/version-7.1.3/install-environment!
1102.       (eval '(lambda (env)
        a.  (let ((repl (nearest-repl))
                i.  (environment (->environment env)))
        b.  (set-repl-state/environment! (cmdl/state repl) environment)
        c.  (if (not (cmdl/parent repl))
                i.  (set! user-repl-environment environment))))
1103.       (->environment '(runtime rep))))

1104.       (define scheme-runtime/version-7.2/install-environment! ge)

1105.       (define scheme-runtime/install-environment!
1106.       (cond
1107.       ((version-7.1.3?) scheme-runtime/version-7.1.3/install-environment!)
1108.       ((version-7.2?) scheme-runtime/version-7.2/install-environment!)
1109.       (else (version-error))))

1110.       (define (scheme-runtime/fasload-file? pathname)
1111.       (let* ((port (open-input-file pathname))
        a.  (fasl-marker (peek-char port))
        b.  (result (and (not (eof-object? fasl-marker))
                i.  (= 250 (char->ascii fasl-marker)))))
1112.       (begin
```

```
1113.       (close-input-port port)
1114.       result)))


1115.       ;;; Dump/load

1116.       (define scheme-runtime/fasdump fasdump)
1117.       (define scheme-runtime/fasload fasload)


1118.       ;;; Path stuff

1119.       (define scheme+/scheme+-ext "scm")
1120.       (define scheme+/bin-ext "bin")
1121.       (define scheme+/com-ext "com")

1122.       ;; Get pathname of existing file ending in ".scm"
1123.       (define scheme-runtime/version-7.1.3/scheme+-pathname
1124.       (eval `(lambda (filename)
        a. (find-true-pathname (->pathname filename)
                                    a. '(,scheme+/scheme+-ext)))
1125.       (->environment '(runtime load))))

1126.       (define scheme-runtime/version-7.2/scheme+-pathname
1127.       (eval `(lambda (filename)
        a. (find-pathname filename '(,scheme+/scheme+-ext)))
1128.       (->environment '(runtime load))))

1129.       (define scheme-runtime/version-7.3/scheme+-pathname
1130.       (eval `(lambda (filename)
        a. (with-values (lambda ()
                        1. (find-pathname filename
                                        i. '((,scheme+/scheme+-ext
                                       ii. ,',load/internal))))
        b. (lambda (pathname loader) pathname)))
1131.       (->environment '(runtime load))))


1132.       (define scheme-runtime/scheme+-pathname
1133.       (cond
1134.       ((version-7.3?) scheme-runtime/version-7.3/scheme+-pathname)
1135.       ((version-7.2?) scheme-runtime/version-7.2/scheme+-pathname)
1136.       ((version-7.1.3?) scheme-runtime/version-7.1.3/scheme+-pathname)
1137.       (else (version-error))))

1138.       ;; Get pathname of existing file ending in either ".bin" or ".scm"
```

```
(define scheme-runtime/version-7.1.3/bin-pathname
(eval `(lambda (filename)
  (find-true-pathname (->pathname filename)
                      '(,scheme+/bin-ext
                        ,scheme+/scheme+-ext)))
(->environment '(runtime load))))

(define scheme-runtime/version-7.2/bin-pathname
(eval `(lambda (filename)
  (find-pathname filename '(,scheme+/bin-ext
                            ,scheme+/scheme+-ext)))
(->environment '(runtime load))))

(define scheme-runtime/version-7.3/bin-pathname
(eval `(lambda (filename)
  (with-values (lambda ()
                 (find-pathname filename
                                '((,scheme+/scheme+-ext
                                   ,',load/internal)
                                  (,scheme+/bin-ext
                                   ,',load/internal))))
    (lambda (pathname loader) pathname)))
(->environment '(runtime load))))

(define scheme-runtime/bin-pathname
(cond
((version-7.3?) scheme-runtime/version-7.3/bin-pathname)
((version-7.2?) scheme-runtime/version-7.2/bin-pathname)
((Version-7.1.3?) scheme-runtime/version-7.1.3/bin-pathname)
(else (version-error))))

;; Get pathname for existing file ending in ".scm" and return a pathname
;; for the same file, but ending in ".bin"
(define scheme-runtime/version-7.1.3/sf-name
(eval `(lambda (filename)
  (pathname->string
    (pathname-new-type
      (find-true-pathname (->pathname filename)
                          '(,scheme+/scheme+-ext))
      ,scheme+/bin-ext)))
(->environment '(runtime load))))

(define scheme-runtime/version-7.2/sf-name
(eval `(lambda (filename)
  (->namestring
    (pathname-new-type
```

```
        c. (find-pathname filename
                    1. '(,scheme+/scheme+-ext))
        d. ,scheme+/bin-ext)))
1161.       (->environment '(runtime load))))

1162.       (define scheme-runtime/version-7.3/sf-name
1163.       (eval `(lambda (filename)
        a. (->namestring
        b. (pathname-new-type
        c. (with-values (lambda ()
                    1. (find-pathname filename
                                i. '((,scheme+/scheme+-ext
                               ii. ,',load/internal))))
                ii. (lambda (pathname loader) pathname))
        d. "bin")))
1164.       (->environment '(runtime load))))


1165.       (define scheme-runtime/sf-name
1166.       (cond
1167.       ((version-7.3?) scheme-runtime/version-7.3/sf-name)
1168.       ((version-7.2?) scheme-runtime/version-7.2/sf-name)
1169.       ((version-7.1.3?) scheme-runtime/version-7.1.3/sf-name)
1170.       (else (version-error))))


1171.       ;; Get pathname for existing file ending in ".scm" or ".bin" and return a
1172.       ;; pathname for the same file, but ending in ".com"
1173.       (define scheme-runtime/version-7.1.3/cf-name
1174.       (eval `(lambda (filename)
        a. (pathname->string
        b. (pathname-new-type
        c. (find-true-pathname (->pathname filename)
                        a. '(,scheme+/scheme+-ext
                        b. ,scheme+/bin-ext))
        d. ,scheme+/com-ext)))
1175.       (->environment '(runtime load))))

1176.       (define scheme-runtime/version-7.2/cf-name
1177.       (eval `(lambda (filename)
        a. (->namestring
        b. (pathname-new-type
        c. (find-pathname filename '(,scheme+/scheme+-ext
                                i. ,scheme+/bin-ext))
        d. ,scheme+/com-ext)))
1178.       (->environment '(runtime load))))
```

```
(define scheme-runtime/version-7.3/cf-name
(eval `(lambda (filename)
(->namestring
(pathname-new-type
(with-values (lambda ()
                (find-pathname filename
                  '((,scheme+/scheme+-ext
                    ,',load/internal)
                    (,scheme+/bin-ext
                    ,',load/internal))))
  (lambda (pathname loader) pathname))
,scheme+/com-ext)))
(->environment '(runtime load))))

(define scheme-runtime/cf-name
(cond
((version-7.3?) scheme-runtime/version-7.3/cf-name)
((version-7.2?) scheme-runtime/version-7.2/cf-name)
((version-7.1.3?) scheme-runtime/version-7.1.3/cf-name)
(else (version-error))))


;; Compile a syntaxed file
(define (scheme-runtime/compile-bin-file . args)
(error "scheme-runtime/compile-bin-file not currently supported")
(cond ((version-7.2?)
(if (environment-bound? system-global-environment 'compile-bin-file)
compile-bin-file
(lambda (filename)
  (display "Compile not loaded, unable to compile: ")
  (display filename)
  (newline))))
(else (version-error)))
)


(define (scheme-runtime/file-modification-time file)
;; Return a 0 rather than null if file not present
(let ((t (file-modification-time file)))
(if (null? t) 0 t)))

;;-----------------------------------------------------
;; Start up Scheme+
(scheme+)
```