```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; recon.scm: TYPE RECONSTRUCTOR FOR Scheme/R
;;;
;;; To run this code, first load Scheme+, then this file.
;;;
;;; Evaluate
;;;
;;;    (recon <sexp>)
;;;
;;; to type-reconstruct the Scheme/R expression <sexp>
;;;
;;;


;;;--------------------------------------------------------------------------
;;; Magic for handling unit.  This should really be in a separate file, or
;;; be part of Scheme+.

(define-structure (unit-obj (print-procedure
                    (lambda (state struct)
                        (unparse-string state "#u")))))

;; THE-UNIT is the unique instance of the UNIT-OBJ structure
(define the-unit (make-unit-obj))

(define (unit? obj) (eq? obj the-unit))

;;; Changing parser to handle #u
(define parse-object/unit
(let ((discard-char (access discard-char
                    (->environment (find-package '(runtime parser))))))
(lambda ()
(discard-char)
the-unit)))

(parser-table/set-entry! system-global-parser-table
                    '("#u" "#U")
                    parse-object/unit)

;;; Constructors for handling UNIT

(define unit->sexp
(make-constructor
(lambda () #u)
(lambda (sexp succ fail)
```

```
(if (unit? sexp)
    (succ)
    (fail)))))

(define a-unit unit->sexp)


;;;----------------------------------------------------------------------------
;;; SYM-SEXP is a synonym for SYMBOL->SEXP (and should replace it in
;;; future versions of Scheme+)

(define sym->sexp (make-sexp-constructor 'sym symbol?))

;;;----------------------------------------------------------------------------
;;; CELLS in Scheme
(define cell-tag (list '*cell*))
(define (cell val) (list cell-tag val))
(define (cell? x) (and (list? x) (= (length x) 2) (eq? (car x) cell-tag)))
(define (^ x) (if (cell? x) (cadr x) (error "^: not a cell " x)))
(define (:= x y) (if (cell? x) (begin (set-car! (cdr x) y) #u)
                     (error "^: not a cell " x)))


;;;----------------------------------------------------------------------------
;;; DATATYPES

;;; Expressions

(define-datatype exp                    ; E ::=
(unit->exp)                         ;       Unit
(boolean->exp bool)                 ;     | Bool
(integer->exp int)                  ;     | Int
(string->exp string)                ;     | String
(symbol->exp sym)                   ;     | (symbol Sym)
(variable->exp sym)                 ;     | I
(lambda->exp (listof sym) exp)      ;     | (lambda (I*) E)
(call->exp exp (listof exp))        ;     | (E0 E*)
(if->exp exp exp exp)               ;     | (if E1 E2 E3)
(primop->exp primop (listof exp))   ;     | (primop O E*)
(let->exp (listof definition) exp)  ;     | (let ((I E)*) E0)
(letrec->exp (listof definition) exp)  ;     | (letrec ((I E)*) E0)
(set!->exp sym exp)                 ;     | (set! I E)
(begin->exp exp exp)                ;     | (begin E E)
;; **MODULES**
(module->exp (listof definition))   ;     | (module (define I E)*)
(with->exp (listof sym) exp exp)    ;     | (with (I*) E1 E2)
```

```
;; **MODULES**
)

(define-datatype definition                    ; (I E)
(make-definition sym exp))

(define (definition-name d)
(match d
((make-definition name value) name)))

(define (definition-value d)
(match d
((make-definition name value) value)))

;;; Types

(define-datatype type
(tvariable->type tvariable)              ; type variable
(base->type sym)                         ; (unit, bool, int, string, symbol)
(compound->type sym (listof type))       ; ->, list-of, etc.
(unknown->type)                          ; placeholder for unconstrained tvars
;; **MODULES**
(moduleof->type (listof sym) (listof type)) ; (moduleof (val I T)*)
;; **MODULES**
)

(define unit-type    (base->type 'unit))
(define boolean-type (base->type 'bool))
(define integer-type (base->type 'int))
(define string-type  (base->type 'string))
(define symbol-type  (base->type 'sym))


(define same-constructor? eq?)

;; **MODULES**
(define (same-field-names? names1 names2)
(if (null? names1)
(null? names2)
(if (null? names2)
    a. #f
    b. (and (eq? (car names1) (car names2))
        i. (same-field-names? (cdr names1) (cdr names2))))))
;; **MODULES**

(define (make-arrow-type arg-types body-type)
```

```
100.    (compound->type arrow-constructor
            i.  (cons body-type arg-types)))

101.    (define arrow-constructor '->)

102.    (define same-name? eq?)

103.    ;;; Type schemas

104.    (define-datatype tvar-or-schema
105.    (tvar->tvar-or-schema tvariable)
106.    (schema->tvar-or-schema schema))

107.    (define-datatype schema
108.    (make-schema (listof tvariable) type))

109.    (define (schema-generics s)
110.    (match s
111.    ((make-schema generics typ) generics)))

112.    (define (schema-type s)
113.    (match s
114.    ((make-schema generics typ) typ)))
```

```
115.    ;;;----------------------------------------------------------------------
116.    ;;; TYPE RECONSTRUCTION

117.    (define (reconstruct exp tenv)
118.    (match exp
119.    ((unit->exp)              unit-type)
120.    ((boolean->exp _)         boolean-type)
121.    ((integer->exp _)         integer-type)
122.    ((string->exp _)          string-type)
123.    ((symbol->exp _)          symbol-type)
124.    ((variable->exp var)      (reconstruct-variable var tenv))
125.    ((if->exp test con alt)   (reconstruct-if test con alt tenv))
126.    ((primop->exp primop args)  (reconstruct-primop primop args tenv))
127.    ((lambda->exp formals body) (reconstruct-lambda formals body tenv))
128.    ((call->exp op args)      (reconstruct-call op args tenv))
129.    ((let->exp defs body)     (reconstruct-let defs body tenv))
130.    ((letrec->exp defs body)  (reconstruct-letrec defs body tenv))
131.    ((set!->exp id exp)       (reconstruct-set! id exp tenv))
132.    ((begin->exp exp1 exp2)   (reconstruct-begin exp1 exp2 tenv))
133.    ;; **MODULES**
134.    ((module->exp defs)       (reconstruct-module defs tenv))       ; ***
135.    ((with->exp vars mod body)  (reconstruct-with vars mod body tenv)) ; ***
136.    ;; **MODULES**
137.    ))

138.    (define (reconstruct-variable var tenv)
139.    (let ((tvar-or-schema (tlookup tenv var)))
140.    (match tvar-or-schema
141.    ((tvar->tvar-or-schema tvar)
142.    (tvariable->type tvar))
143.    ((schema->tvar-or-schema schema)
144.    (instantiate-schema schema)))))

145.    (define (reconstruct-if test con alt tenv)
146.    (begin (unify! (reconstruct test tenv)
            i. boolean-type)
        b. (let ((con-type (reconstruct con tenv))
            i. (alt-type (reconstruct alt tenv)))
        c. (begin (unify! con-type alt-type)
            i. con-type))))

147.    (define (reconstruct-lambda vars body tenv)
148.    (let ((new-tvars (map new-tvariable vars)))
149.    (make-arrow-type
150.    (map tvariable->type new-tvars)
```

```
151.        (reconstruct body
                i. (extend-by-tvariables tenv vars new-tvars)))))

152.        (define (reconstruct-call op args tenv)
153.        (let ((arg-types (map (lambda (arg) (reconstruct arg tenv))
                        1. args))
154.        (result-type (tvariable->type (new-tvariable 'result))))
155.        (begin (unify! (reconstruct op tenv)
                i. (make-arrow-type arg-types result-type))
        b. result-type)))

156.        (define (reconstruct-primop op args tenv)
157.        (let ((arg-types (map (lambda (arg) (reconstruct arg tenv))
                        1. args))
158.        (result-type (tvariable->type (new-tvariable 'result))))
159.        ;; Get the type of the operator from the standard type env:
160.        (begin (unify! (reconstruct-variable op standard-type-environment)
                i. (make-arrow-type arg-types result-type))
        b. result-type)))

161.        (define (reconstruct-let defs body tenv)
162.        (reconstruct body
                i. (extend-by-schemas
               ii. tenv
              iii. (map definition-name defs)
               iv. (map (lambda (def)
                v. (compute-schema
                        1. (reconstruct (definition-value def) tenv)
                        2. tenv))
               vi. defs))))

163.        (define (reconstruct-letrec defs body tenv)
164.        (let ((names (map definition-name defs)))
165.        (let ((tvars (map new-tvariable names)))
166.        (let ((dummy-tenv (extend-by-tvariables tenv names tvars)))
167.        (let ((types (map (lambda (def)
                        1. (reconstruct (definition-value def) dummy-tenv))
                        2. defs)))
        b. (begin (for-each-2 unify!
                        1. (map tvariable->type tvars)
                        2. types)
               ii. (reconstruct body
                        1. (extend-by-schemas
                                a. tenv
                                b. names
                                c. (map (lambda (type)
```

d. (compute-schema type tenv))
e. types)))))))))

```
168.    (define (for-each-2 proc lst1 lst2)
169.    (if (null? lst1)
170.    #u
171.    (begin
172.    (proc (car lst1) (car lst2))
173.    (for-each-2 proc (cdr lst1) (cdr lst2)))))

174.    ;;; Note: the use of UNIFY!-LIST rather than FOR-EACH-2 fails to
175.    ;;; correctly type (or find a type error in) the following example:
176.    ;;; (recon '(letrec ((a (lambda () 3))
177.    ;;;                 (b (if (a) 1 2)))
178.    ;;;           4))


179.    (define (reconstruct-set! id exp tenv)
180.    (begin (unify! (reconstruct-variable id tenv)
```

i. (reconstruct exp tenv))

b. unit-type))

```
181.    (define (reconstruct-begin exp1 exp2 tenv)
182.    ;; Be sure to check type-safety of 1st expression:
183.    (begin (reconstruct exp1 tenv)
```

a. (reconstruct exp2 tenv)))

```
;;;---------------------------------------------------------------------------
;;; TYPE SCHEMAS

(define (compute-schema type tenv)  ;Function GEN from handout
(make-schema (generic-tvariables type tenv)
    i. type))

; NOTE: generic-tvariables looks not only at tvariables in the
; given type, but also at tvariables in the leaves of

; the fully unwound version of the given type.  This interacts with
; a similar unwinding at instantiation time to appropriately handle
; generalization.  There is potential confusion in that the returned
; list may contain types that are not manifestly in TYPE but are in
; the fully unwound tree associated with it.

(define (generic-tvariables type tenv) ;Compute FTV(type) - FTE(tenv)
(match (prune type)
((tvariable->type tvar)
(if (generic-tvariable? tvar tenv)
a. (list tvar)
b. (null)))
((compound->type _ operands)
(letrec ((loop (lambda (ops tvars)
    i. (if (null? ops)
        1. tvars
        2. (loop (cdr ops)
            a. (union (generic-tvariables (car ops) tenv)
                i. tvars))))))
(loop operands (null))))
;; **MODULES**
((moduleof->type _ fields)
(letrec ((loop (lambda (flds tvars)
    i. (if (null? flds)
        1. tvars
        2. (loop (cdr flds)
            a. (union (generic-tvariables (car flds) tenv)
                i. tvars))))))
(loop fields (null))))
;; **MODULES**
((base->type _) (null))
(_ (error "This shouldn't happen!" (unparse-type type)))))

(define (union l1 l2)
(cond ((null? l1) l2)
```

```
211.        ((null? l2) l1)
212.        ((in-tvariable-list? (car l1) l2) (union (cdr l1) l2))
213.        (else (cons (car l1) (union (cdr l1) l2)))))

214.        (define (in-tvariable-list? tvar tvar-list)
215.        (if (null? tvar-list)
216.        #f
217.        (if (same-tvariable? tvar (car tvar-list))
        a.  #t
        b.  (in-tvariable-list? tvar (cdr tvar-list)))))

218.        ; Instantiate a type schema on a fresh set of type variables.
219.        ; [This corresponds to Cardelli's "FreshType".]

220.        (define (instantiate-schema schema)
221.        (substitute-into-type
222.        (map (lambda (g)
        a.  (tvariable->type (new-tvariable (tvariable-name g))))
223.        (schema-generics schema))
224.        (schema-generics schema)
225.        (schema-type schema)))

226.        ; [The following corresponds to Cardelli's "Fresh"; note the call to prune.]

227.        ; Note that this unwinds TYPE out to the leaves when doing the
    substitution;
228.        ; this guarantees that we don't miss any substitutions because type itself
229.        ; isn't fully unwound.

230.        (define (substitute-into-type types tvars type)
231.        (let ((type (prune type)))
232.        (match type
233.        ((tvariable->type tvar)
234.        (letrec ((loop (lambda (ts tvars)
                    1.  (if (null? ts)
                    2.  type
                    3.  (if (same-tvariable? tvar (car tvars))
                            a.  (car ts)
                            b.  (loop (cdr ts) (cdr tvars)))))))
        b.  (loop types tvars)))
235.        ((base->type _) type)
236.        ((compound->type c args)
237.        (compound->type c (map (lambda (arg)
                            a.  (substitute-into-type types tvars arg))
                    2.  args)))
238.        ((moduleof->type names args)
```

```
239.        (moduleof->type names (map (lambda (arg)
                                    a.  (substitute-into-type types tvars arg))
                                    b.  args)))
240.        (_ (error "This shouldn't happen" (unparse-type type))))))
```

```
241.  ;;;----------------------------------------------------------------------
242.  ;;; TYPE ENVIRONMENTS.
243.  ;
244.  ;  Environments can be extended in either of two ways:
245.  ;    extend-by-tvariables   should be used by lambda and letrec to bind
246.  ;      variables to type variables
247.  ;    extend-by-schemas  should be used by let and letrec to bind variables
248.  ;      to type schemas
249.  ;
250.  ;  Once constructed, there are two operations one can perform on a
251.  ;  type environment:
252.  ;    tlookup : tenv * var -> (tvar + schema)
253.  ;      does the usual thing.
254.  ;    generic-tvariable? : tvar * tenv -> bool
255.  ;      returns true iff tvar is not free in the type of any var bound in tenv.

256.  (define-datatype type-environment
257.  (make-type-env tlookup-proc generic-tvariable?-proc))

258.  (define (tenv-lookup te)
259.  (match te
260.  ((make-type-env lookup generic?) lookup)))

261.  (define (tenv-generic? te)
262.  (match te
263.  ((make-type-env lookup generic?) generic?)))

264.  (define (extend-by-tvariables outer-tenv vars tvars)
265.  (extend-tenv outer-tenv
          i.  vars
          ii.  (map tvar->tvar-or-schema tvars)
          iii.  (lambda (tvar)
          iv.  ;; tvar is an unconstrained type variable.
          v.  (letrec ((loop (lambda (tvars)
                                a.  (if (null? tvars)
                                b.  (generic-tvariable? tvar outer-tenv)
                                c.  (if (occurs-in-type?
                                          i.  tvar
                                          ii.  (tvariable->type (car tvars)))
                                          iii.  ;; (same-tvariable? tvar (car tvars))
                                          iv.  #f
                                          v.  (loop (cdr tvars)))))))
          vi.  (loop tvars)))))

266.  (define (extend-by-schemas outer-tenv vars schemas)
```

```
(extend-tenv outer-tenv
  i. vars
  ii. (map schema->tvar-or-schema schemas)
  iii. (lambda (tvar)
  iv. (generic-tvariable? tvar outer-tenv))))

;Students' code should not call this
(define (extend-tenv outer-tenv vars typas generic-tvariable?-proc)
(make-type-env
(lambda (var)
(letrec ((loop (lambda (vars typas)
    i. (if (null? vars)
        1. (tlookup outer-tenv var)
        2. (if (same-variable? var (car vars))
        3. (car typas)
        4. (loop (cdr vars) (cdr typas)))))))
(loop vars typas)))
generic-tvariable?-proc))

(define empty-type-environment
(make-type-env
(lambda (var) (error "Unbound variable: " (sym->sexp var)))
(lambda (tvar) #t)))

(define (tlookup tenv var)
((tenv-lookup tenv) var))

(define same-variable? eq?)

(define (generic-tvariable? tvar tenv)
((tenv-generic? tenv) tvar))

; Proving the correctness of this implementation of GENERIC-
  TVARIABLE?
; is tricky.
```

```
286.    ;;;----------------------------------------------------------------------
287.    ;;; TYPE VARIABLES

288.    ; A type variable is implemented as a record that contains a cell.  The
289.    ; global substitution is realized as the collective contents of the
290.    ; cells for all type variables.

291.    (define-datatype tvariable
292.    (make-tvariable sym int (cellof type)))          ; id gennum cell

293.    (define (tvariable-name tvar)
294.    (match tvar
295.    ((make-tvariable name _ _) name)))

296.    (define (tvariable-uid tvar)
297.    (match tvar
298.    ((make-tvariable _ uid _) uid)))

299.    (define (tvariable-cell tvar)
300.    (match tvar
301.    ((make-tvariable _ _ c) c)))

302.    (define tvariable-counter (cell 0))

303.    (define (reset-tvariable-counter!)
304.    (:= tvariable-counter 0))

305.    (define (new-tvariable id)
306.    (begin (:= tvariable-counter (+ (^ tvariable-counter) 1))
    a.  (make-tvariable id (^ tvariable-counter) (cell unknown-type))))

307.    (define (tvariable-binding tvar)
308.    (^ (tvariable-cell tvar)))

309.    (define (extend-substitution! tvar binding)
310.    (begin (:= (tvariable-cell tvar) binding)
    a.  #t))

311.    (define (same-tvariable? tvar1 tvar2)
312.    (= (tvariable-uid tvar1) (tvariable-uid tvar2)))

313.    (define unknown-type (unknown->type))

314.    (define (tvariable->symbol tvar)
315.    (string->symbol
```

316. (string-append "?" (symbol->string (tvariable-name tvar))
    i. "-" (number->string (tvariable-uid tvar)))))

```
317.        ;;;----------------------------------------------------------------------
318.        ;;; UNIFICATION
319.        ;;;
320.        ;;;  Has side effects.
321.        ;;;  Generates an error if there is no unification.

322.        (define (unify! type1 type2)
323.        (if (unify!-internal type1 type2)
324.        #u
325.        (error "Type clash:  "
        a. (unparse-type type1) (unparse-type type2))))

326.        (define (unify!-internal type1 type2)
327.        (let ((type1 (prune type1))
328.        (type2 (prune type2)))
329.        ;; Now if a type is a variable, it will be unbound
330.        (match type1
331.        ((tvariable->type v1)
332.        (match type2
        a. ((tvariable->type v2)
        b. (if (same-tvariable? v1 v2)
        c. #t
        d. (extend-substitution! v1 type2)))
        e. (_
        f. (if (occurs-in-type? v1 type2)
        g. #f                          ;Circularity
        h. (extend-substitution! v1 type2)))))
333.        ((base->type c1)
334.        (match type2
        a. ((tvariable->type v2)
        b. (extend-substitution! v2 type1))
        c. ((base->type c2)
        d. (same-name? c1 c2))
        e. (_ #f)))
335.        ((compound->type con1 args1)
336.        (match type2
        a. ((tvariable->type v2)
        b. (if (occurs-in-type? v2 type1)
        c. #f
        d. (extend-substitution! v2 type1)))
        e. ((compound->type con2 args2)
        f. (if (same-constructor? con1 con2)
        g. (unify!-list args1 args2)
        h. #f))
        i. (_ #f)))
```

```
((moduleof->type names1 args1)
(match type2
  ((tvariable->type v2)
  (if (occurs-in-type? v2 type1)
  #f
  (extend-substitution! v2 type1)))
  ((moduleof->type names2 args2)
  (if (same-field-names? names1 names2)
  (unify!-list args1 args2)
  #f))
  (_ #f))))))

(define (unify!-list types1 types2)
(if (null? types1)
(null? types2)
(if (null? types2)
  #f
  (if (unify!-internal (car types1) (car types2))
  (unify!-list (cdr types1) (cdr types2))
  #f))))

; Chase substitutions of tvariables until either a non-tvariable or an
; unbound tvariable is found.

(define (prune type)
(match type
((tvariable->type tvar)
(match (tvariable-binding tvar)
((unknown->type) type)
(other-type (prune other-type))))
(_ type)))

; Occurs check: prevent circular substitutions.

(define (occurs-in-type? tvar type)
(match (prune type)
((tvariable->type tvar2)
;; prune has guaranteed that tvar2 is unbound
(same-tvariable? tvar tvar2))
((compound->type c args)
(letrec ((loop (lambda (args)
    i. (if (null? args)
        1. #f
        2. (or (occurs-in-type? tvar (car args))
        3. (loop (cdr args)))))))
(loop args)))
```

```
((moduleof->type names args)
(letrec ((loop (lambda (args)
    i.  (if (null? args)
            1. #f
            2. (or (occurs-in-type? tvar (car args))
            3. (loop (cdr args)))))))
(loop args)))
(_ #f)))
```

```
;;;----------------------------------------------------------------------
;;; PARSING/UNPARSING -- old version with non-optimal strategy

; Parse a definition

(define (parse-definition sexp)
(match sexp
;; Allow Scheme-style definitions ...
(`(define (,name ,@args) ,body)
(make-definition (parse-formal name)
    i. (parse `(lambda ,args ,body))))
(`(define ,name ,value)
(make-definition (parse-formal name) (parse value)))
(_ (error "Invalid definition: " sexp))))

(define (parse-formal sexp)
(match sexp
((sym->sexp name)
(if ((member? eq?) name all-keywords)
a. (error "Attempt to use reserved word as variable name" sexp)
b. name))
(_ (error "Invalid variable name: " sexp))))

(define (parse-call operator operands)
(call->exp (parse operator)
a. (map parse operands)))

(define (parse-binding-spec bspec)
(match bspec
(`(,name ,value) (make-definition (parse-formal name) (parse value)))
(_ (error "Invalid binding specifier: " bspec))))

(define (syntax-error sexp)
(error "Invalid expression syntax: " sexp))


; Parse a single expression

(define (parse sexp)                    ; sexp -> exp
(match sexp
((unit->sexp)          (unit->exp))
((bool->sexp b)         (boolean->exp b))
((int->sexp n)         (integer->exp n))
((sym->sexp sym)        (variable->exp sym))
((string->sexp n)       (string->exp n))
```

```
397.        (`(,(sym->sexp head) ,@_)  ((parser-for-keyword head) sexp))
398.        ;; Procedure call is the default
399.        (`(,operator ,@operands)    (parse-call operator operands))
400.        (_ (error "Unrecognized expression " sexp))))

401.        (define-datatype parser-table
402.        (make-parser-table (listof sym)
                i.  (-> (sym) (-> (sexp) exp))))

403.        ; Expressions of the form (reserved-word ...)

404.        (define keyword-table
405.        (letrec
406.        ((keywords (cell (null)))
407.        (lookup (cell (lambda (head)
                i.  (lambda (sexp)
                        1.  ;; Procedure call is the default
                        2.  (match sexp
                        3.  (`(,operator ,@operands)
                        4.  (parse-call operator operands))
                        5.  (_ (error "KEYWORD TABLE: This shouldn't happen!"
                                a.  )))))))

408.        ;; DEFINE-KEYWORD is a function that defines a reserved word,
409.        ;; associating it with a function that can parse the named construct.

410.        (define-keyword
        a.  (lambda (keyword parser)
        b.  (let ((current-lookup (^ lookup)))
        c.  (begin (:= lookup
                        1.  (lambda (head)
                        2.  (if (eq? head keyword)
                        3.  parser
                        4.  (current-lookup head))))
                ii.  (:= keywords
                        1.  (cons keyword (^ keywords)))
               iii.  keyword)))))
411.        (begin
412.        ;; List of parsing functions.

413.        (define-keyword 'quote          ; (quote Name)
414.        (lambda (sexp)
        a.  (match sexp
        b.  (`(quote ,(sym->sexp name))
        c.  (symbol->exp name))
        d.  (_ (syntax-error sexp)))))
```

```
(define-keyword 'lambda        ; (lambda (I*) E)
(lambda (sexp)
  (match sexp
    (`(lambda (,@formals) ,body)
      (lambda->exp (map parse-formal formals)
                   (parse body)))
    (_ (syntax-error sexp)))))

(define-keyword 'if     ; (if E1 E2 E3)
(lambda (sexp)
  (match sexp
    (`(if ,test ,con ,alt)
      (if->exp (parse test)
               (parse con)
               (parse alt)))
    (_ (syntax-error sexp)))))

(define-keyword 'primop        ; (primop O E*)
(lambda (sexp)
  (match sexp
    (`(primop ,op ,@args)
      ;; Assume valid primop -- type reconstruction will verify
      ;; number of args.
      (primop->exp op (map parse args)))
    (_ (syntax-error sexp)))))

(define-keyword 'let    ; (let ((I E)*) E0)
(lambda (sexp)
  (match sexp
    (`(let (,@bspecs) ,body)
      (let->exp
       (map parse-binding-spec bspecs)
       (parse body)))
    (_ (syntax-error sexp)))))

(define-keyword 'letrec          ; (letrec ((I E)*) E0)
(lambda (sexp)
  (match sexp
    (`(letrec (,@bspecs) ,body)
      (letrec->exp
       (map parse-binding-spec bspecs)
       (parse body)))
    (_ (syntax-error sexp)))))

(define-keyword 'set!  ; (set! I E)
```

```
426.        (lambda (sexp)
        a. (match sexp
        b. (`(SET! ,(sym->sexp id) ,sexp)
        c. (set!->exp id (parse sexp)))
        d. (_ (syntax-error sexp)))))

427.        (define-keyword 'begin                  ; (begin E1 E2) + sugars
428.        (lambda (sexp)
        a. (match sexp
        b. (`(BEGIN) (unit->exp))
        c. (`(BEGIN ,sexp) (parse sexp))
        d. (`(BEGIN ,sexp1 ,sexp2) (begin->exp (parse sexp1) (parse sexp2)))
        e. (`(BEGIN ,sexp1 ,sexp2 ,@rest)
        f. (begin->exp (parse sexp1) (parse `(BEGIN ,sexp2 ,@rest))))
        g. (_ (syntax-error sexp)))))


429.        ;; Sugar

430.        ;; (and)       ==>  #t
431.        ;; (and E)     ==>  E
432.        ;; (and E0 E+)  ==>  (if E0 (and E+) #f)
433.        ;;
434.        (define-keyword 'and
435.        (lambda (sexp)
        a. (match sexp
        b. (`(and ,@exp-list)
        c. (parse (letrec ((recur (lambda (exps)
                                    a. (match exps
                                        i. ((null)          `#t)
                                        ii. (`(,exp)          exp)
                                        iii. ((cons first rest)
                                        iv. `(if ,first ,(recur rest) #f))))))
                    ii. (recur exp-list))))
        d. (_ (syntax-error sexp)))))


436.        ;; (or)       ==>  #f
437.        ;; (or E)     ==>  E
438.        ;; (or E0 E+)  ==>  (if E0 #t (or E+))
439.        ;;
440.        (define-keyword 'or
441.        (lambda (sexp)
        a. (match sexp
        b. (`(or ,@exp-list)
        c. (parse (letrec ((recur (lambda (exps)
```

```
                                    a. (match exps
                                        i. ((null)          `#f)
                                        ii. (`(,exp)          exp)
                                        iii. ((cons first rest)
                                        iv. `(if ,first #t ,(recur rest)))))))
                                ii. (recur exp-list))))
                        d. (_ (syntax-error sexp)))))

442.                    ;; (cond (E E)* (else E))
443.                    (define-keyword 'cond
444.                    (lambda (sexp)
                        a. (match sexp
                        b. (`(cond) (syntax-error sexp))
                        c. (`(cond (else ,default))
                        d. (parse default))
                        e. (`(cond (,predicate ,consequent) ,@clauses)
                        f. (parse `(if ,predicate
                                        1. ,consequent
                                        2. (cond ,@clauses))))
                        g. )))

445.                    ;; (list E*)
446.                    ;;
447.                    (define-keyword 'list
448.                    (lambda (sexp)
                        a. (match sexp
                        b. (`(list ,@exp-list)
                        c. (parse (letrec ((recur (lambda (exps)
                                    a. (match exps
                                        i. ((null) `(null))
                                        ii. ((cons first rest)
                                        iii. `(cons ,first ,(recur rest)))))))
                                ii. (recur exp-list))))
                        d. (_ (syntax-error sexp)))))

449.                    ;; Module stuff

450.                    (define-keyword 'module       ; (module (define I E)*)
451.                    (lambda (sexp)
                        a. (match sexp
                        b. (`(module ,@fspecs)
                        c. (module->exp
                        d. (map parse-definition fspecs)))
                        e. (_ (syntax-error sexp)))))

452.                    (define-keyword 'with ; (with (I*) E1 E2)
```

```
(lambda (sexp)
  a. (match sexp
  b. (`(with (,@formals) ,mod ,body)
  c. (with->exp (map parse-formal formals)
                1. (parse mod)
                2. (parse body)))
  d. (_ (syntax-error sexp)))))


;; Now return the whole parser table

(make-parser-table (^ keywords)
                1. (^ lookup))

)))

(define parser-for-keyword
(match keyword-table
((make-parser-table keywords parser-lookup) parser-lookup)))

(define all-keywords
(match keyword-table
((make-parser-table keywords parser-lookup) keywords)))
```

```
;;; Type expression parser

(define (parse-type sexp)
(match sexp
((sym->sexp sym) (base->type sym))
(`(-> (,@arg-types) ,result-type)
(compound->type arrow-constructor
    (cons (parse-type result-type)
        (map parse-type arg-types))))
;; **MODULES**
(`(moduleof ,@fields)
(parse-module-type fields))
;; **MODULES**
(`(,(sym->sexp name) ,@types)
(compound->type name (map parse-type types)))
(_ (error "Invalid type expression syntax " sexp))))

;; **MODULES**
(define (parse-module-type fields)
(if (null? fields)
(moduleof->type (null) (null))
(match (parse-module-type (cdr fields))
((moduleof->type names typs)
(match (car fields)
(`(val ,id ,typ)
(moduleof->type (cons id names) (cons (parse-type typ) typs)))
(_ (error "Invalid syntax in moduleof field entry "
    (car fields)))))
(_ (error  "PARSE-MODULE-TYPE: this shouldn't happen!"
    fields)))))
;; **MODULES**

; Type expression unparser

(define (unparse-type type)
(match (prune type)
((base->type sym) (sym->sexp sym))
((compound->type constructor operands)
(if (same-constructor? constructor arrow-constructor)
`(-> (,@(map unparse-type (cdr operands)))
,(unparse-type (car operands)))
`(,(sym->sexp constructor) ,@(map unparse-type operands))))
;; **MODULES**
((moduleof->type names operands)
`(moduleof
```

```
494.        ,@(map2 (lambda (id t) `(val ,(sym->sexp id) ,(unparse-type t)))
                i. names
               ii. operands)))
495.        ;; **MODULES**
496.        ((tvariable->type tvar)
497.        (sym->sexp (tvariable->symbol tvar)))
498.        ((unknown->type)
499.        '(*unknown*))))

500.        ; Parse a type schema (generic (I*) T)

501.        (define (parse-schema sexp)
502.        (match sexp
503.        (`(generic (,@names) ,type)
504.        (let ((names (map (lambda (name)
                              1. (match name
                              2. ((sym->sexp name) name)
                              3. (_ (error "Invalid type schema parameter: " name))))
                    ii. names)))
505.        (let ((tvars (map new-tvariable names)))
        a. (make-schema
        b. tvars
        c. (substitute-for-names (map tvariable->type tvars)
                                 a. names
                                 b. (parse-type type))))))
506.        (_ (make-schema (null) (parse-type sexp)))))

507.        ; substitute-for-names is a kludge, to be used only by initialization
508.        ; code.  Other ways to do this:
509.        ;
510.        ; (1) change the type parser to take an environment argument;
511.        ;
512.        ; (2) generalize substitute-into-type so that it; can substitute for
513.        ;     either names or tvars;
514.        ;
515.        ; (3) change the representation of schemas so that the generic variables
516.        ;     in the type are not tvars but rather names.

517.        (define (substitute-for-names types names type)
518.        (match type
519.        ((tvariable->type _) type)                ;shouldn't happen
520.        ((base->type name)
521.        (letrec ((loop (lambda (ts ns)
                    i. (if (null? ts)
                              1. type
                              2. (if (same-name? name (car ns))
```

```
                    3. (car ts)
                    4. (loop (cdr ts) (cdr ns)))))))
522.        (loop types names)))
523.        ((compound->type c args)
524.        (compound->type c (map (lambda (arg)
                    1. (substitute-for-names types names arg))
                    2. args)))
525.        ;; **MODULES**
526.        ((moduleof->type fieldnames args)
527.        (moduleof->type fieldnames (map (lambda (arg)
                                i. (substitute-for-names types names arg))
                            b. args)))
528.        ;; **MODULES**
529.        (_ (error "SUBSTITUTE-FOR-NAMES: This shouldn't happen! "
    a. (unparse-type type)))))

530.        (define (unparse-schema s)
531.        (match s
532.        ((make-schema tvars type)
533.        `(generic (,@(map sym->sexp (map tvariable->symbol tvars)))
                i. ,(unparse-type type)))))
```

```
534.    ;;;----------------------------------------------------------------------
535.    ;;; STANDARD TYPE ENVIRONMENT

536.    (define standard-type-bindings
537.    (list
538.    ; Arithmetic
539.    '(+ (-> (int int) int))
540.    '(- (-> (int int) int))
541.    '(* (-> (int int) int))
542.    '(/ (-> (int int) int))

543.    ; Relational
544.    '(= (-> (int int) bool))
545.    '(/= (-> (int int) bool))
546.    '(< (-> (int int) bool))
547.    '(> (-> (int int) bool))
548.    '(<= (-> (int int) bool))
549.    '(>= (-> (int int) bool))

550.    ; Logical
551.    '(and? (-> (bool bool) bool))
552.    '(or?  (-> (bool bool) bool))
553.    '(not? (-> (bool) bool))

554.    ; Symbols
555.    '(sym=? (-> (sym sym) bool))

556.    ; Strings
557.    '(string=? (-> (sym sym) bool))

558.    ; Lists
559.    '(cons   (generic (t) (-> (t (list-of t)) (list-of t))))
560.    '(car    (generic (t) (-> ((list-of t)) t)))
561.    '(cdr    (generic (t) (-> ((list-of t)) (list-of t))))
562.    '(null   (generic (t) (-> () (list-of t))))
563.    '(null?  (generic (t) (-> ((list-of t)) bool)))
564.    '(append (generic (t) (-> ((list-of t) (list-of t)) (list-of t))))
565.    ))

566.    (define standard-type-environment
567.    (extend-by-schemas empty-type-environment
            i.  (map (lambda (b)
                    1. (match b
                    2. (`(,(sym->sexp name) ,_) name)))
                    3. standard-type-bindings)
```

```
                ii. (map (lambda (b)
                        1. (match b
                        2. (`(,_ ,schema) (parse-schema schema))))
                        3. standard-type-bindings)))

568.        ;;;----------------------------------------------------------------------
569.        ;;; UTILITIES

570.        (define (member? pred)
571.        (lambda (elt lst)
572.        (letrec ((loop (lambda (lst)
                i. (if (null? lst)
                        1. #f
                        2. (if (pred elt (car lst))
                        3. #t
                        4. (loop (cdr lst)))))))
573.        (loop lst))))

574.        (define (map2 proc lst1 lst2)
575.        (if (or (null? lst1) (null? lst2))
576.        (null)
577.        (cons (proc (car lst1) (car lst2))
        a. (map2 proc (cdr lst1) (cdr lst2)))))



578.        ;;;----------------------------------------------------------------------
579.        ;;; TOP-LEVEL

580.        (define (recon sexp)
581.        (begin (reset-tvariable-counter!)
        a. (unparse-type (reconstruct (parse sexp)
                                a. standard-type-environment))))
```