```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 6.821 Problem Set #4
;;;
;;;
;;; FLEX and FLAT interpreters:
;;; FLEX = (CBV FL) - recursion
;;; FLAT = (CBV FL) - recursion - (free vars in procedures) + tuples
;;;
;;; In both languages, all primitive operators are accessed as
;;; primops, e.g., (primop + ...), (primop left ...), etc.
;;;
;;; Author: Brian
;;; Created: 10/1/94
;;; Adapted from Lyn's ps4.fx (1992) and fl-naming.scm (1994)
;;; Revisions:
;;;        10/4: Fixed left and right primops.
;;;        10/7: Changed parse-common to use parse in all cases.
;;;        10/11: Added symbol to keyword list.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; PROBLEM SET CODE begins with DATATYPES:
;;;        ignore the initial patching for #u


;;;------------------------------------------------------------------------
;;; 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)
```

```
36. the-unit)))

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

38. ;;; Constructors for handling UNIT

39. (define unit->sexp
40. (make-constructor
41. (lambda () #u)
42. (lambda (sexp succ fail)
43. (if (unit? sexp)
        a. (succ)
        b. (fail)))))

44. (define a-unit unit->sexp)


45. ;;;----------------------------------------------------------------------------
46. ;;; SYM-SEXP is a synonym for SYMBOL->SEXP (and should replace it in
47. ;;; future versions of Scheme+)

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

```
;;;----------------------------------------------------------------------
;;; DATATYPES

(define-datatype exp
($lit exp)
($var-ref sym)
($proc sym exp)         ; In FLAT, sym can be only free var in exp
($call exp exp)
($if exp exp exp)
($let (listof sym) (listof exp) exp)      ; FLAT only
($pair exp exp)
($primop primitive (listof exp))
;; Tuples
($tuple (listof exp))                     ; FLAT only
($tuple-ref exp int)                      ; FLAT only
($tuple? exp)                             ; FLAT only
($tuple-length exp)                       ; FLAT only
($tuple-append exp exp)                   ; FLAT only
;; Top-level FLAT program -- only used by the LIFTer
($program-flat (listof sym) (listof exp) exp)
)

;;; NOTE: LET expressions are represented by
;;;   * a list of the indentifiers
;;;   * a list of the expressions for those identifiers
;;;   * a body expression
;;; A LET expression could also have been represented by
;;;   * a list of bindings that contain both an identifier and an expression
;;;   * a body expression
;;;

; Expressible Values
(define-datatype exp-val
(val->exp-val val)
(error->exp-val string)
)

(define-datatype value
(unit->val)
(int->val  int)
(bool->val bool)
(sym->val  sym)
(procedure->val (-> (value) exp-val))
(pair->val value value)
(tuple->val     (listof value))   ; FLAT only
```

90.)

```
;;;----------------------------------------------------------------------
;;; EVALUATOR

;;; Curried evaluator performs all dispatches on syntactic types first.
;;; Eval: exp -> env -> exp-val

;;; In this implementation, FLEX-EVAL and FLAT-EVAL are the same.
;;; However, FLEX-EVAL should be given an EXP parsed by FLEX-PARSE
;;; and FLAT-EVAL should be given an EXP parsed by FLAT-PARSE.

(define (flex-eval exp) (eval-exp exp))
(define (flat-eval exp) (eval-exp exp))

        (define (flex-eval-empty exp) ((flex-eval exp) the-empty-environment))
        (define (flat-eval-empty exp) ((flat-eval exp) the-empty-environment))

        (define (eval-exp exp)
        (match exp
        (($lit xval)           (eval-literal xval))
        (($var-ref v)          (eval-var-ref v))
        (($proc formal body)    (eval-proc formal body))
        (($call rator rand)     (eval-call rator rand))
        (($if test then else)   (eval-if test then else))
        (($let names exps body) (eval-let names exps body))
        (($pair left right)     (eval-pair left right))
        (($primop prim args)    (eval-primop prim args))
        (($tuple exps)          (eval-tuple exps))
        (($tuple-ref exp index) (eval-tuple-ref exp index))
        (($tuple? exp)     (eval-tuple? exp))
        (($tuple-length exp)    (eval-tuple-length exp))
        (($tuple-append exp1 exp2)       (eval-tuple-append exp1 exp2))
        (($program-flat names exps body) (eval-program names exps body))
        (_ (error
        (string-append
    a.  "FLEX/FLAT-EVAL doesn't know how to handle:\n"
    b.  (write-sexp-to-string (flat-unparse exp)))))
        ))

        (define (eval-literal exp-val)
        (lambda (env) exp-val))

        (define (eval-var-ref v)
        (lambda (env) (lookup v env)))

        (define (eval-proc formal body)
```

```
126.        (let ((body-meaning (eval-exp body)))
127.        (lambda (env)
128.        (val->exp-val
129.        (procedure->val
130.        (lambda (val)
    a.  (body-meaning (extend-env formal val env))))))))

131.        (define (eval-call rator rand)
132.        (let ((rator-meaning (eval-exp rator))
133.        (rand-meaning  (eval-exp rand)))
134.        (lambda (env)
135.        (with-procedure (rator-meaning env)
136.        (lambda (p)
    a.  (with-value (rand-meaning env)
    b.  p))))))

137.        (define (eval-if test then else)
138.        (let ((test-meaning (eval-exp test))
139.        (then-meaning (eval-exp then))
140.        (else-meaning (eval-exp else)))
141.        (lambda (env)
142.        (with-boolean (test-meaning env)
143.        (lambda (b)
    a.  (if b (then-meaning env) (else-meaning env)))))))

144.        (define (eval-let names exps body)
145.        (let ((exp-meanings (map eval-exp exps))
146.        (body-meaning (eval-exp body)))
147.        (lambda (env)
148.        (with-values (map (lambda (m) (m env)) exp-meanings)
149.        (lambda (values)
    a.  (body-meaning (extend-env-by-list
                1.  names
                2.  values
                3.  env)))))))

150.        (define (eval-pair left right)
151.        (let ((left-meaning  (eval-exp left))
152.        (right-meaning (eval-exp right)))
153.        (lambda (env)
154.        (with-value (left-meaning env)
155.        (lambda (left)
    a.  (with-value (right-meaning env)
    b.  (lambda (right)
    c.  (val->exp-val
            i.  (pair->val left right)))))))))
```

```
156.    (define (eval-primop prim args)
157.    (match prim
158.    ((make-primitive name n proc)
159.    (let ((arg-meanings (map eval-exp args)))
160.    (lambda (env)
        a. (with-values (map (lambda (m) (m env)) arg-meanings)
        b. (lambda (vals)
        c. (apply proc vals))))))))

161.    (define (eval-tuple components)
162.    (let ((component-meanings (map eval-exp components)))
163.    (lambda (env)
164.    (with-values (map (lambda (m) (m env)) component-meanings)
165.    (lambda (values)
        a. (val->exp-val (tuple->val values)))))))

166.    (define (eval-tuple-ref tuple-exp index)
167.    (let ((tuple-meaning (eval-exp tuple-exp)))
168.    (lambda (env)
169.    (with-tuple (tuple-meaning env)
170.    (lambda (elts)
        a. (if (and (>= index 0)
                i. (< index (length elts)))
        b. (val->exp-val (list-ref elts index))
        c. (error->exp-val "TUPLE-REF: Index out of range")))))))

171.    (define (eval-tuple? exp)
172.    (let ((tuple-meaning (eval-exp exp)))
173.    (lambda (env)
174.    (with-value (tuple-meaning env)
175.    (lambda (v)
        a. (val->exp-val
        b. (bool->val
        c. (match v
        d. ((tuple->val _) #t)
        e. (_ #f)))))))))

176.    (define (eval-tuple-length exp)
177.    (let ((tuple-meaning (eval-exp exp)))
178.    (lambda (env)
179.    (with-tuple (tuple-meaning env)
180.    (lambda (elts)
        a. (val->exp-val
        b. (int->val (length elts))))))))
```

```
(define (eval-tuple-append tuple-exp1 tuple-exp2)
(let ((tuple1-meaning (eval-exp tuple-exp1))
(tuple2-meaning (eval-exp tuple-exp2)))
(lambda (env)
(with-tuple (tuple1-meaning env)
(lambda (elts1)
  (with-tuple (tuple2-meaning env)
  (lambda (elts2)
  (val->exp-val
    (tuple->val (append elts1 elts2)))))))))

(define (eval-program names exps body)
(let ((exp-meanings (map eval-exp exps))
(body-meaning (eval-exp body)))
(lambda (env)
(letrec ((new-env
    (lambda (var)
    (letrec ((loop
        (lambda (vs ms)
        (cond
            ((null? vs) (lookup var env))
            ((same-var? var (car vs))
            ((car ms) new-env))
            (else (loop (cdr vs) (cdr ms)))))))
    (loop names exp-meanings)))))
;; Ensure that they are all defined
(with-values (map (lambda (a-meaning) (a-meaning new-env))
        exp-meanings)
  (lambda (ignore)
  (body-meaning new-env)
  ))))))
```

```
;;;----------------------------------------------------------------------
;;; ENVIRONMENTS

;;; This injects the output so that normal and unbound lookups both give
;;; EXP-VALUEs
(define extend-env
(lambda (var1 value env)
(lambda (var2)
(if (same-var? var1 var2)
    (val->exp-val value)
(lookup var2 env)))))

(define lookup
(lambda (var env) (env var)))

(define the-empty-environment
(lambda (var) (error->exp-val
    (string-append "Unbound variable: "
        (symbol->string var)))))

(define same-var? eq?)

(define (extend-env-by-list vars vals env)
(if (null? vars)
env
(extend-env (car vars) (car vals)
        (extend-env-by-list (cdr vars) (cdr vals) env))))
```

```
212.        ;;;----------------------------------------------------------------------
213.        ;;; Auxiliary procedures

214.        ;; with-value: (-> (exp-val (-> (value) exp-val)) exp-val)
215.        (define (with-value exp-val return)
216.        (match exp-val
217.        ((val->exp-val val) (return val))
218.        ((error->exp-val _) exp-val)
219.        ))

220.        ;; NOTE: This conflicts with Scheme's WITH-VALUES!
221.        ;;
222.        ;; with-values: (-> (exp-vals (-> (values) exp-val)) exp-val)
223.        (define (with-values exp-vals return)
224.        (if (null? exp-vals)
225.        (return '())
226.        (with-value (car exp-vals)
227.        (lambda (val)
228.        (with-values (cdr exp-vals)
       a.  (lambda (vals)
       b.  (return (cons val vals))))))))

229.        ;; with-integer: (-> (exp-val (-> (int) exp-val)) exp-val)
230.        (define (with-integer exp-val return)
231.        (with-value exp-val
232.        (lambda (val)
233.        (match val
234.        ((int->val n) (return n))
235.        (_  (error-with-val
       a.  "Non-integer occurs in position where a integer is expected: "
       b.  val))
236.        ))))

237.        ;; with-boolean: (-> (exp-val (-> (bool) exp-val)) exp-val)
238.        (define (with-boolean exp-val return)
239.        (with-value exp-val
240.        (lambda (val)
241.        (match val
242.        ((bool->val n) (return n))
243.        (_  (error-with-val
       a.  "Non-integer occurs in position where a integer is expected: "
       b.  val))
244.        ))))

245.        ;; with-procedure: (-> (exp-val (-> (procedure) exp-val)) exp-val)
```

```
246.        (define (with-procedure exp-val return)
247.        (with-value exp-val
248.        (lambda (val)
249.        (match val
250.        ((procedure->val p) (return p))
251.        (_  (error-with-val
        a.  "Non-procedure occurs in position where a procedure is expected: "
        b.  val))
252.        ))))

253.        ;; with-tuple:
254.        ;;          (-> (exp-val (-> ((listof exp-val)) exp-val)) exp-val)
255.        (define (with-tuple exp-val return)
256.        (with-value exp-val
257.        (lambda (val)
258.        (match val
259.        ((tuple->val exps) (return exps))
260.        (_ (error-with-val
        a.  "Non-tuple occurs where a tuple is expected: "
        b.  val))
261.        ))))
```

```
;;;----------------------------------------------------------------------
;;; Primitives: stored in a table to use for looking up primops and
;;;        building initial flk environment.

(define-datatype primitive
(make-primitive sym n-args proc))

(define *flk-primitives-table* '())

(define (add-prim! primitive)
(set! *flk-primitives-table*
(cons primitive *flk-primitives-table*))
unspecific)

(define (lookup-primop sym succ fail)
(let loop ((lib *flk-primitives-table*))
(if (null? lib)
(fail)
(match (car lib)
  a. ((make-primitive prim-name _ _)
  b. (if (eq? sym prim-name)
       i. (succ (car lib))
      ii. (loop (cdr lib))))
  c. (_ (loop (cdr lib)))))))


(define (define-general-primitive sym nargs proc)
(add-prim! (make-primitive sym nargs proc)))

(define (define-typed-primitive sym scheme-proc arg-list return-
  construtor)
(define-general-primitive
sym
(length arg-list)
(add-types sym scheme-proc arg-list return-construtor)))

(define (define-predicate sym obj->value)
(define-general-primitive
sym 1
(lambda (val)
(val->exp-val
(bool->val
(match val
  a. ((obj->value _) #t)
  b. (_ #f)))))))
```

```
(define (define-logical-primitive sym n-args scheme-proc)
(define-general-primitive
sym n-args
(add-types sym
  a. scheme-proc
  b. (make-list n-args bool->val)
  c. bool->val)))

(define (define-arithop-primitive sym scheme-proc)
(define-general-primitive
sym 2
(add-types sym scheme-proc (list int->val int->val) int->val)))

(define (define-arithop-error-at-0 sym scheme-proc)
(define-general-primitive
sym 2
(add-type-checks sym
    i. (lambda (x y)
   ii. (if (= y 0)
          1. 'divide-by-zero-error
          2. (scheme-proc x y)))
  iii. (list int->val int->val)
  iv. (lambda (result)
   v. (if (eq? result 'divide-by-zero-error)
          1. (error->exp-val "Divide by zero")
          2. (val->exp-val (int->val result)))))))

(define (define-arithop-relate sym scheme-proc)
(define-general-primitive
sym 2
(add-types sym scheme-proc (list int->val int->val) bool->val)))


(define (add-types sym scheme-proc arg-list result->val)
(add-type-checks sym
    i. scheme-proc
   ii. arg-list
  iii. (lambda (x) (val->exp-val (result->val x)))))

(define (add-type-checks prim-name scheme-proc arg-types result->exp-
  val)
(lambda arg-vals
(define (check-types types vals return)
(if (null? types)              ;;; Assume NARGS-checking is done by EVAL-
  PRIMOP
```

a. (return '())
b. (match (car vals)
c. (((car types) fst) ;; Types is a list of constructors
d. (check-types (cdr types)
    i. (cdr vals)
        1. (lambda (rest)
        2. (return (cons fst rest)))))
e. (_ (error->exp-val
    i. (string-append "Type error in application of primitive: "
        a. (symbol->string prim-name)))))))

313. (check-types arg-types
    i. arg-vals
    ii. (lambda (untagged-args)
    iii. (result->exp-val (apply scheme-proc untagged-args))))))

```
;;;----------------------------------------------------------------------
;;; Primitive Handlers

(define (fl/unit? val)
(val->exp-val
(bool->val
(match val
((unit->val) #t)
(_ #f)))))

(define (fl/pair? val)
(val->exp-val
(bool->val
(match val
((pair->val _ _) #t)
(_ #f)))))

(define (fl/pair-selector op)
(lambda (val)
(match val
;; left and right are values, so inject
((pair->val left right) (val->exp-val (op left right)))
(_ (error-with-val "pair selector applied to non-pair" exp-val)))))

(define fl/left  (fl/pair-selector (lambda (left right) left)))
(define fl/right (fl/pair-selector (lambda (left right) right)))
```

```
336.    ;;;----------------------------------------------------------------------
337.    ;;; Put primitives in the table

338.    ;; Predicates
339.    (define-general-primitive 'unit? 1 fl/unit?)
340.    (define-predicate 'boolean?   bool->val)
341.    (define-predicate 'integer?   int->val)
342.    (define-predicate 'symbol?    sym->val)
343.    (define-predicate 'procedure? procedure->val)
344.    (define-general-primitive 'pair? 1 fl/pair?)

345.    ;; Logical Primitives
346.    (define-logical-primitive 'not?   1 not)
347.    (define-logical-primitive 'and?   2 (lambda (x y) (and x y)))
348.    (define-logical-primitive 'or?    2 (lambda (x y) (or  x y)))
349.    (define-logical-primitive 'bool=? 2 (lambda (x y) (if x y (not y))))

350.    ;; Arithmetic Primitives
351.    (define-arithop-primitive  '+   +)
352.    (define-arithop-primitive  '-   -)
353.    (define-arithop-primitive  '*   *)
354.    (define-arithop-error-at-0 '/   quotient)
355.    (define-arithop-error-at-0 'rem remainder)

356.    ;; Arithmetic Relations
357.    (define-arithop-relate '=  =)
358.    (define-arithop-relate '/= (lambda (x y) (not (= x y))))
359.    (define-arithop-relate '<  <)
360.    (define-arithop-relate '<= <=)
361.    (define-arithop-relate '>  >)
362.    (define-arithop-relate '>= >=)

363.    ;; Symbols
364.    (define-typed-primitive 'sym=? eq? (list sym->val sym->val) bool->val)

365.    ;; Pairs
366.    (define-general-primitive 'left  1 fl/left)
367.    (define-general-primitive 'right 1 fl/right)
```

```
;;;----------------------------------------------------------------------
;;; SYMBOL SETS

(define the-empty-set '())

(define set-empty? null?)

(define (set->list set) set)

(define (list->set lst) lst)

(define set-member?
(lambda (elt set)
(cond ((null? set) #f)
  a. ((eq? elt (car set)) #t)
  b. (else (set-member? elt (cdr set))))))

(define set-adjoin
(lambda (elt set)
(if (set-member? elt set)
set
(cons elt set))))

(define set-choose car)

(define set-rest cdr)

(define set-singleton (lambda (elt) (list elt)))

(define set-union
(lambda (s1 s2)
(cond ((set-empty? s1) s2)
  a. ((set-member? (set-choose s1) s2)
  b. (set-union (set-rest s1) s2))
  c. (else (set-adjoin (set-choose s1)
                       1. (set-union (set-rest s1) s2))))))

(define set-intersection
(lambda (s1 s2)
(cond ((set-empty? s1) the-empty-set)
  a. ((set-member? (set-choose s1) s2)
  b. (set-adjoin (set-choose s1)
                 i. (set-intersection (set-rest s1) s2)))
  c. (else (set-intersection (set-rest s1) s2)))))
```

```
391.    (define set-difference
392.    (lambda (s1 s2)
393.    (cond ((set-empty? s1) the-empty-set)
    a. ((set-member? (set-choose s1) s2)
    b. (set-difference (set-rest s1) s2))
    c. (else (set-adjoin (set-choose s1)
                    1. (set-difference (set-rest s1) s2))))))

394.    (define mapunion
395.    (lambda (proc lst)
396.    (if (null? lst)
397.    '()
398.    (set-union (proc (car lst))
                i. (mapunion proc (cdr lst))))))

399.    (define (set-subset? s1 s2)
400.    (every? (lambda (elt) (set-member? elt s2))
    a. (set->list s1)))
```

```
401.    ;;;----------------------------------------------------------------------
402.    ;;; PARSING

403.    (define (parse-common parse sexp language-string)
404.    (match sexp
405.    ((unit->sexp)                ($lit (val->exp-val (unit->val))))
406.    ((bool->sexp b)              ($lit (val->exp-val (bool->val b))))
407.    ((int->sexp i)               ($lit (val->exp-val (int->val i))))
408.    (`(SYMBOL ,(sym->sexp s))    ($lit (val->exp-val (sym->val s))))
409.    ((sym->sexp s)               ($var-ref s))
410.    (`(PROC ,(sym->sexp formal) ,body) ($proc formal (parse body)))
411.    (`(CALL ,rator ,rand)        ($call (parse rator) (parse rand)))
412.    (`(IF ,test ,then ,else)     ($if (parse test)
                                        i. (parse then)
                                       ii. (parse else)))
413.    (`(PAIR ,left ,right)        ($pair (parse left) (parse right)))
414.    (`(PRIMOP ,(sym->sexp op) ,@args)  (parse-primop op args parse))
415.    (_ (error (string-append "PARSE: Unknown " language-string "
    expression!")
        a. sexp))
416.    ))

417.    (define (parse-primop op args parser)
418.    (lookup-primop op
419.    (lambda (prim)
420.    (match prim
421.    ((make-primitive name n proc)
        a. (if (= n (length args))
        b. ($primop prim (map parser args))
        c. (error "PARSE: Primop applied to wrong number of arguments: "
                i. (list op args))))))
422.    (lambda () (error "PARSE: Unknown primop!" op))))

423.    (define (flex-parse sexp)
424.    (if (flex-sugar? sexp)
425.    (flex-parse (flex-desugar sexp))
426.    (parse-common flex-parse sexp "FLEX")))

427.    (define (flat-parse-program sexp)
428.    (match sexp
429.    (`(PROGRAM ,(list->sexp bindings) ,body)
430.    ;; Bindings are mutually recursive and at top-level so okay if they
431.    ;; have free variables.
432.    (let ((old-flag check-free-variables?))
433.    (set! check-free-variables? #f)
```

```
434.        (let ((bound-expressions
    a.  (map (compose flat-parse binding->val) bindings)))
            i.  (set! check-free-variables? old-flag)
           ii.  ($program-flat (map binding->var bindings)
                        a.  bound-expressions
                        b.  (flat-parse body)))))
435.        (_ (flat-parse sexp))))

436.        (define (flat-parse sexp)
437.        (if (flat-sugar? sexp)
438.        (flat-parse (flat-desugar sexp))
439.        (match sexp
440.        ;; Check free-variables restriction
441.        (`(PROC ,(sym->sexp formal) ,body) (flat-parse-proc formal body))
442.        ;; Let is primitive in FLAT
443.        (`(LET ,bindings ,body)          (flat-parse-let bindings body))
444.        ;; Tuples
445.        (`(TUPLE ,@components)           ($tuple (map flat-parse components)))
446.        (`(TUPLE-REF ,tuple ,(int->sexp index))
    a.  ($tuple-ref (flat-parse tuple) index))
447.        (`(TUPLE? ,exp)                  ($tuple? (flat-parse exp)))
448.        (`(TUPLE-LENGTH ,exp)            ($tuple-length (flat-parse exp)))
449.        (`(TUPLE-APPEND ,exp1 ,exp2)     ($tuple-append (flat-parse exp1)
                                                a.  (flat-parse exp2)))
450.        (_ (parse-common flat-parse sexp "FLAT"))
451.        )))

452.        (define (flat-parse-proc formal body)
453.        ;; Embed restriction check for abstractions in parser
454.        (let ((abst ($proc formal (flat-parse body))))
455.        (if (or (flat? abst) (not check-free-variables?))
456.        abst
457.        (error
    a.  (string-append
    b.  "FLAT-PARSE: Not a legal FLAT abstraction\n(contains the free
        variables "
    c.  (string-append
    d.  (with-output-to-string
    e.  (lambda () (display (list->sexp (set->list (free-vars abst))))))
    f.  (string-append
    g.  "):\n"
    h.  (with-output-to-string
    i.  (lambda () (display (flat-unparse abst)))))))))))

458.        (define check-free-variables? #t)
```

```
459.    (define (flat-parse-let bindings body)
460.    ($let (map binding->var bindings)
461.    (map (compose flat-parse binding->val) bindings)
462.    (flat-parse body)))
```

```
;;;----------------------------------------------------------------------
;;; UNPARSING

(define (make-unparser unparse language-name)
(lambda (exp)
(match exp
(($lit (val->exp-val (unit->val)))   (unit->sexp))
(($lit (val->exp-val (bool->val b))) (bool->sexp b))
(($lit (val->exp-val (int->val i)))  (int->sexp i))
(($lit (val->exp-val (sym->val s)))  `(symbol ,s))
(($var-ref s)         (sym->sexp s))
(($proc formal body)   `(PROC ,(sym->sexp formal) ,(unparse body)))
(($call rator rand)    `(CALL ,(unparse rator) ,(unparse rand)))
(($if test then else)  `(IF ,(unparse test) ,(unparse then) ,(unparse else)))
(($pair left right)    (unparse-pair (unparse left) (unparse right)))
(($primop (make-primitive name _ _) args) `(PRIMOP ,name ,@(map
    unparse args)))
(_ (error
        a. (string-append "UNPARSE -- unknown " language-name " expression.")))
)))

(define (unparse-pair left right)
(match right
(`#u           `(list ,left))
(`(LIST ,@elts) `(list ,left ,@elts))
(_             `(PAIR ,left ,right))))

(define flex-unparse (make-unparser (lambda (exp) (flex-unparse exp))
                                    a. "FLEX"))

(define flat-unparse
(let ((recur (make-unparser (lambda (exp) (flat-unparse exp))
                                    a. "FLAT")))
(lambda (exp)
(match exp
(($let names exps body) `(LET ,(map list names (map flat-unparse exps))
                                    a. ,(flat-unparse body)))
(($tuple components)       `(TUPLE ,@(map flat-unparse components)))
(($tuple-ref tuple index)  `(TUPLE-REF ,(flat-unparse tuple)
                                        i. ,index))
(($tuple? exp)             `(TUPLE? ,(flat-unparse exp)))
(($tuple-length exp)       `(TUPLE-LENGTH ,(flat-unparse exp)))
(($tuple-append exp1 exp2) `(TUPLE-APPEND ,(flat-unparse exp1)
                                              1. ,(flat-unparse exp2)))
(($program-flat names exps body)
```

a. `(PROGRAM ,(map list names (map flat-unparse exps))
    i. ,(flat-unparse body)))
497. (_ (recur exp))))))

```
;;;-----------------------------------------------------------------------
;;; DESUGARING
;;;
;;; Build an environment mapping sugar keyword to a sexp-transform.

(define (flex-sugar? sexp) (sugar? sexp flex-keywords))
(define (flat-sugar? sexp)
;; LET is in the FLAT-kernel
(match sexp
(`(LET ,(a-list _) ,_) #f)
(_ (sugar? sexp flat-keywords))))

(define flex-keywords '(proc call if pair primop symbol))
(define flat-keywords (append
    i. flex-keywords
    ii. '(let tuple tuple-ref tuple? tuple-length tuple-append)))

(define (flex-desugar sexp) (desugar sexp))
(define (flat-desugar sexp) (desugar sexp))

(define *sugar-keywords* '())

(define *sugar-env*
(lambda (keyword)
(error "Syntax Error: unbound sugar keyword" keyword)))

(define (sugar? sexp keywords)
(match sexp
(`(lambda ,(a-list _) ,_) #t)      ;; curried abstraction
(`(,(a-symbol sym) ,@_) (or (memq sym *sugar-keywords*)
                            a. (not (memq sym keywords))))
(`(,operator ,@operands) #t)   ;; application
(_ #f)))

(define (desugar sexp)
;; The standard environment is handled differently than in desugaring rules
((lookup (keyword sexp) *sugar-env*) sexp))

(define (keyword sexp)
(match sexp
(`(,(a-symbol keyword) ,@_) (if (memq keyword *sugar-keywords*)
                                a. keyword
                                b. implicit-call-tag))
(`(,operator ,@operands)  implicit-call-tag)
(_ (error "KEYWORD: unrecognized syntax" sexp))
```

```
530.        ))

531.        (define (define-sugar keyword transformer)
532.        (if (null? (memq keyword *sugar-keywords*))
533.        (set! *sugar-keywords* (cons keyword *sugar-keywords*)))
534.        ;; Extend environment
535.        (let ((old-env *sugar-env*))
536.        (set! *sugar-env* (lambda (sym)
                    1. (if (eq? sym keyword)
                    2. transformer
                    3. (lookup sym old-env))))))

537.        (define-sugar 'list
538.        (lambda (sexp)
539.        (match sexp
540.        (`(LIST) #u)
541.        (`(LIST ,first ,@rest)
542.        `(PAIR ,first (LIST ,@rest)))
543.        (_ (error "DESUGAR-LIST: invalid syntax" sexp)))))

544.        (define-sugar 'quote
545.        (lambda (sexp)
546.        (match sexp
547.        (`(QUOTE ,item)
548.        (match item
        a. ((bool->sexp b) item)
        b. ((int->sexp  n) item)
        c. ((sym->sexp s) `(SYMBOL ,s))
        d. ((list->sexp lst)
        e. `(LIST ,@(map (lambda (elt) `(QUOTE ,elt)) lst)))
        f. (_ (error "DESUGAR-QUOTE: invalid syntax" sexp))))
549.        (_ (error "DESUGAR-QUOTE: invalid syntax" sexp)))))

550.        (define-sugar 'lambda
551.        (lambda (sexp)
552.        (match sexp
553.        (`(LAMBDA () ,body)
554.        `(PROC ,(fresh-var) ,body))
555.        (`(LAMBDA (,a-formal) ,body)
556.        `(PROC ,a-formal ,body))
557.        (`(LAMBDA (,first ,@rest) ,body)
558.        `(PROC ,first
        a. (LAMBDA (,@rest)
                i. ,body)))
559.        (_ (error "DESUGAR-LAMBDA: invalid syntax" sexp)))))
```

```
560.    (define implicit-call-tag (list '*implicit-call*))

561.    (define-sugar implicit-call-tag
562.    (lambda (sexp)
563.    (match sexp
564.    (`(,operator)
565.    `(CALL ,operator #u))
566.    (`(,operator ,one-arg)
567.    `(CALL ,operator ,one-arg))
568.    (`(,operator ,first-arg ,@rest)
569.    `((CALL ,operator ,first-arg) ,@rest))
570.    (_ (error "DESUGAR-IMPLICIT-CALL: invalid syntax" sexp)))))

571.    (define-sugar 'cond
572.    (lambda (sexp)
573.    (match sexp
574.    (`(COND) #u)
575.    (`(COND (ELSE ,default))
576.    default)
577.    (`(COND (ELSE ,default) ,@rest)
578.    (error "DESUGAR-COND: else not last clause" sexp))
579.    (`(COND (,test ,consequent) ,@rest)
580.    `(IF ,test ,consequent (COND ,@rest)))
581.    (_ (error "DESUGAR-COND: invalid syntax" sexp)))))

582.    (define-sugar 'and
583.    (lambda (sexp)
584.    (match sexp
585.    (`(AND) (bool->sexp #t))
586.    (`(AND ,first ,@rest)
587.    `(IF ,first (AND ,@rest) #f))
588.    (_ (error "DESUGAR-AND: invalid syntax" sexp)))))

589.    (define-sugar 'or
590.    (lambda (sexp)
591.    (match sexp
592.    (`(OR) (bool->sexp #f))
593.    (`(OR ,first ,@rest)
594.    `(IF ,first #t (OR ,@rest)))
595.    (_ (error "DESUGAR-OR: invalid syntax" sexp)))))

596.    ;; Only in FLEX
597.    (define-sugar 'let
598.    (lambda (sexp)
599.    (match sexp
600.    (`(LET (,@bindings) ,body)
```

```
601.        ;; Syntax of bindings enforced by binding selectors
602.        `((LAMBDA ,(list->sexp (map binding->var bindings)) ,body)
      a.    ,@(map binding->val bindings)))
603.        (_ (error "DESUGAR-LET: invalid syntax" sexp)))))

604.        (define binding->var
605.        (lambda (sexp)
606.        (match sexp
607.        (`(,(sym->sexp var) ,_) var)
608.        (_ (error "BINDING->VAR: Not a binding!" sexp)))))

609.        (define binding->val
610.        (lambda (sexp)
611.        (match sexp
612.        (`(,_ ,val) val)
613.        (_ (error "BINDING->VAL: Not a binding!" sexp)))))
```

```
614.    ;;;----------------------------------------------------------------------
615.    ; FRESH-VAR
616.    ; Generate a new variable.

617.    ; Fresh variables are of the form `[VAR-n]', where n is the next integer
618.    ; from the counter maintained by FRESH-VAR.  The name is surrounded
619.    ; by square brackets ---  these are illegal in FL identifiers but not
620.    ; in FLK identifiers. (No check is performed here to ensure this
621.    ; constraint holds on FL identifiers; but we are helped by the fact that
622.    ; the Scheme reader doesn't recognize '[' and ']'.)

623.    (define fresh-var
624.    (let ((counter 1))
625.    (lambda ()
626.    (let ((val counter))
627.    (set! counter (+ counter 1))
628.    (string->symbol (string-append "[var-" (number->string val) "]"))))))
```

```
629.    ;;;-------------------------------------------------------------------------
630.    ;;; FREE VARIABLES

631.    (define (free-vars exp)
632.    (match exp
633.    (($lit _) the-empty-set)
634.    (($var-ref id) (set-singleton id))
635.    (($proc formal body) (set-difference (free-vars body)
                                    i.  (set-singleton formal)))
636.    (($call rator rand) (set-union (free-vars rator)
                                a.  (free-vars rand)))
637.    (($if test consequent alternate)
638.    (set-union (free-vars test)
            i.  (set-union (free-vars consequent)
                    1.  (free-vars alternate))))
639.    (($primop op args) (mapunion free-vars args))
640.    (($pair exp1 exp2) (set-union (free-vars exp1) (free-vars exp2)))
641.    (($let ids exps body)
642.    (set-union (mapunion free-vars exps)
            i.  (set-difference (free-vars body)
                                a.  (list->set ids))))
643.    (($tuple components)      (mapunion free-vars components))
644.    (($tuple-ref tuple _)     (free-vars tuple))
645.    (($tuple? exp)            (free-vars exp))
646.    (($tuple-length exp)      (free-vars exp))
647.    (($tuple-append exp1 exp2) (set-union (free-vars exp1) (free-vars exp2)))
648.    (($program-flat ids exps body)
649.    (set-difference (set-union (mapunion free-vars exps)
                                a.  (free-vars body))
           ii.  (list->set ids)))
650.    ))

651.    ;;; Checks that a FLAT expression is legal ---
652.    ;;; i.e., all abstractions are closed

653.    (define (non-scoped? exp) (flat? exp))

654.    (define (flat? exp)
655.    (match exp
656.    (($lit _) #t)
657.    (($var-ref _) #t)
658.    (($proc formal body)
659.    (and (flat? body)
    a.  (set-subset? (free-vars body) (set-singleton formal))))
660.    (($call rator rand) (and (flat? rator) (flat? rand)))
```

```
661.    (($if test consequent alternate) (and (flat? test)
                                              i.  (flat? consequent)
                                             ii.  (flat? alternate)))
662.    (($primop op-name args)    (every? flat? args))
663.    (($pair exp1 exp2)         (and (flat? exp1) (flat? exp2)))
664.    (($let ids exps body)      (and (every? flat? exps) (flat? body)))
665.    (($tuple components)       (every? flat? components))
666.    (($tuple-ref tuple _)      (flat? tuple))
667.    (($tuple? exp)             (flat? exp))
668.    (($tuple-length exp)       (flat? exp))
669.    (($tuple-append exp1 exp2) (and (flat? exp1) (flat? exp2)))
670.    (($program-flat ids exps body) #f)     ;; Kludge...
671.    ))
```

```
672.    ;;;----------------------------------------------------------------------
673.    ;;; READ-EVAL-PRINT LOOP

674.    (define (make-repl prompt parse eval)
675.    (lambda ()
676.    (let loop ((env the-empty-environment))
677.    (newline)
678.    (newline)
679.    (write-string prompt)
680.    (let ((sexp (read)))
681.    (newline)
682.    (cond ((eq? sexp 'quit) 'done)
        a. ((define-sexp? sexp)
              i. (let ((exp-meaning (eval (parse (definition-value sexp))))
             ii. (name (definition-name sexp)))
            iii. (match (exp-meaning env)
             iv. ((val->exp-val v)
              v. (begin
             vi. (display
            vii. (string-append ";Updating " (symbol->string name)
                                    i.  " --> "     (value->string v)))
           viii. (loop (extend-env name v env))))
             ix. (error-val
              x. (begin
             xi. (display (unparse-exp-value error-val))
            xii. (loop env))))))
        b. (else (display (unparse-exp-value ((eval (parse sexp)) env)))
              i. (loop env)))))))

683.    (define flex-repl (make-repl "flex=> " flex-parse flex-eval))

684.    (define flat-repl (make-repl "flat=> " flat-parse-program flat-eval))

685.    (define false-symbol (string->symbol "#f"))

686.    (define (unparse-exp-value exp-val)
687.    (match exp-val
688.    ((error->exp-val string)
689.    (string->sexp (string-append "[FLEX/FLAT Error: " string "]")))
690.    ((val->exp-val v) (unparse-value v))))

691.    (define (unparse-value val)
692.    (match val
693.    ((unit->val)   #u)
694.    ((int->val  n) (int->sexp n))
```

```
695.    ((bool->val b) (if b (bool->sexp b) false-symbol))
696.    ((sym->val s) `',(sym->sexp s))
697.    ((pair->val left right) (unparse-pair (unparse-value left)
                                              i.  (unparse-value right)))
698.    ((procedure->val p) `procedure)
699.    ((tuple->val vals) `(tuple ,@(map unparse-value vals)))
700.    ))


701.    (define (error-with-val error-string val)
702.    (error->exp-val
703.    (string-append error-string "\n\t" (value->string val))))

704.    (define (value->string val)
705.    (with-output-to-string
706.    (lambda () (display (unparse-value val)))))
```

```
;;;----------------------------------------------------------------------
;;; UTILITIES

(define (every? pred lst)
(if (null? lst)
#t
(and (pred (car lst))
a. (every? pred (cdr lst)))))

(define (compose f g)
(lambda (x) (f (g x))))

(define (define-sexp? sexp)
(match sexp
(`(define ,(a-symbol name) ,value-exp) #t)
(_ #f)))

(define (definition-name def)
(match def
(`(define ,(a-symbol name) ,value-exp) name)
(_ (error (string-append "DEFINITION-NAME -- not a definition")))))

(define (definition-value def)
(match def
(`(define ,(a-symbol name) ,value-exp) value-exp)
(_ (error (string-append "DEFINITION-VALUE -- not a definition")))))
```

```
;;;----------------------------------------------------------------------
;;; TESTING

(define (test-translate translator)
(begin
(analyze-it translator (get-input "test=> "))
#u))

(define (analyze-it translator flex-exp)
(let ((flat-exp (translator flex-exp))
(validinput (set-empty? (free-vars flex-exp))))
(newline)
(newline)
(write-string "Input expression (FLEX): ")
(pp (flex-unparse flex-exp))
(newline)
(warn-on-unbound-vars "INPUT" validinput)
(newline)
(write-string "Translated expression (FLAT): ")
(pp (flat-unparse flat-exp))
(newline)
(warn-on-bogus-translator validinput flat-exp)
flat-exp))

(define (get-input prompt)
(newline)
(write-string prompt)
(flex-parse (read)))

(define (test-loop translator)
(letrec
((testloop
(lambda ()
  a. (begin
  b. (newline)
  c. (let ((flex-exp (get-input "testloop=> ")))
  d. (let ((flat-exp (analyze-it translator flex-exp)))
        i. (begin
       ii. (newline)
      iii. (write-string "Input expression value: ")
       iv. (pp (unparse-exp-value (flex-eval-empty flex-exp)))
        v. (newline)
       vi. (newline)
      vii. (write-string "Translated expression value: ")
     viii. (pp (unparse-exp-value (flat-eval-empty flat-exp)))
```

```
                ix.  (newline)
                 x.  (testloop))))))))
(testloop)))

(define (warn-on-unbound-vars string validinput)
(if (not validinput)
(begin
(newline)
(write-string "-------------------------------------------------\n")
(write-string (string-append
                 i.  "*** " string
                ii.  " EXPRESSION CONTAINS UNBOUND VARIABLES!
                     ***\n"))
(write-string "-------------------------------------------------\n")
)
#u))

(define (warn-on-bogus-translator validinput flat-exp)
(if (and validinput (not (non-scoped? flat-exp)))
(begin
(newline)
(write-string "-----------------------------------------\n")
(write-string "*** TRANSLATOR DOESN'T WORK ON THIS CASE!
***\n")
(write-string "(Some PROCs contain free variables!)\n")
(write-string "-----------------------------------------\n")
)
#u))
```

```
;;;----------------------------------------------------------------------
;;; LIFTer

;lift: flat-exp -> flat-exp
(define (lift flat-exp)
(walk flat-exp (lambda (new-exp ids procs)
    i. ($program-flat ids procs new-exp))))

(define (walk exp return)
(match exp
(($proc formal body)
(let ((id (fresh-var)))
(walk body
  a. (lambda (new-body ids procs)
       i. (return ($var-ref id)
      ii. (cons id ids)
     iii. (cons ($proc formal new-body) procs))))))
(($call rator rand)   (walk-list (list rator rand)     $call return))
(($if test consq alt) (walk-list (list test consq alt) $if   return))
(($pair left right)   (walk-list (list left right)     $pair return))
(($let ids exps body) (walk-list (cons body exps)
                        a. (lambda new-list
                             i. ($let ids
                            ii. (cdr new-list)
                           iii. (car new-list)))
                        b. return))
(($primop prim exps)  (walk-list exps
                        a. (lambda new-exps
                             i. ($primop prim new-exps))
                        b. return))
(($tuple exps)        (walk-list exps
                        a. (lambda new-exps ($tuple new-exps))
                        b. return))
(($tuple-ref exp index) (walk-list (list exp)
                             i. (lambda (new-exp)
                            ii. ($tuple-ref new-exp index))
                           iii. return))
(($tuple? exp)       (walk-list (list exp) $tuple? return))
(($tuple-length exp) (walk-list (list exp) $tuple-length return))
(($tuple-append exp1 exp2) (walk-list (list exp1 exp2)
                             i. $tuple-append
                            ii. return))
(_ (return exp '() '()))
))
```

```
798.        (define (walk-list listof-exps constructor return)
799.        (let loop ((listof-exps listof-exps)
      a.    (return (lambda (new-list ids procs)
                 i.   (return (apply constructor new-list) ids procs))))
800.        (if (null? listof-exps)
801.        (return '() '() '()))
802.        (walk (car listof-exps)
      a.    (lambda (new-car car-ids car-procs)
                 i.   (loop (cdr listof-exps)
                ii.   (lambda (new-cdr cdr-ids cdr-procs)
                           1.   (return (cons   new-car   new-cdr)
                                     a.   (append car-ids   cdr-ids)
                                     b.   (append car-procs cdr-procs)))))))

803.        (define (translate-and-lift translator)
804.        (lambda (sexp)
805.        (flat-unparse (lift (translator (flex-parse sexp))))))

806.        (define (lift-flat sexp)
807.        (flat-unparse (lift (flat-parse sexp))))


808.        ;; eg. (lift-flat '(call (proc x x) 3))
809.        ;; ==> (program (([var-4] (proc x x))) (call [var-4] 3))

810.        (define (lift-loop-on-flex)
811.        (let loop ()
812.        (newline)
813.        (newline)
814.        (write-string "lift-flex=> ")
815.        (let* ((sexp (read))
      a.    (flex-exp   (flex-parse sexp))
      b.    (validinput (set-empty? (free-vars flex-exp)))
      c.    (lifted-exp (lift flex-exp))
      d.    (validlift  (set-empty? (free-vars lifted-exp)))
      e.    )
816.        (newline)
817.        (newline)
818.        (write-string "Input expression: ")
819.        (pp (flex-unparse flex-exp))
820.        (warn-on-unbound-vars "INTPUT" validinput)
821.        (newline)
822.        (newline)
823.        (write-string "Lifted expression: ")
824.        (pp (flat-unparse lifted-exp))
825.        (warn-on-unbound-vars "LIFTED" validlift)
```

```
826.        (newline)
827.        (newline)
828.        (write-string "Input expression value: ")
829.        (pp (unparse-exp-value (flex-eval-empty flex-exp)))
830.        (newline)
831.        (newline)
832.        (write-string "Lifted expression value: ")
833.        (pp (unparse-exp-value (flat-eval-empty lifted-exp)))
834.        (loop))))

835.        (define (lift-loop-with-translate translator)
836.        (let loop ()
837.        (newline)
838.        (newline)
839.        (write-string "lift-trans=> ")
840.        (let* ((sexp (read))
    a.  (flex-exp   (flex-parse sexp))
    b.  (validinput (set-empty? (free-vars flex-exp)))
    c.  (flat-exp   (translator flex-exp))
    d.  (lifted-exp (lift flat-exp))
    e.  (validlift  (set-empty? (free-vars lifted-exp)))
    f.  )
841.        (newline)
842.        (newline)
843.        (write-string "Input expression: ")
844.        (pp (flex-unparse flex-exp))
845.        (warn-on-unbound-vars "INTPUT" validinput)
846.        (newline)
847.        (newline)
848.        (write-string "Translated expression: ")
849.        (pp (flat-unparse flat-exp))
850.        (warn-on-bogus-translator validinput flat-exp)
851.        (newline)
852.        (newline)
853.        (write-string "Lifted expression: ")
854.        (pp (flat-unparse lifted-exp))
855.        (warn-on-unbound-vars "LIFTED" validlift)
856.        (newline)
857.        (newline)
858.        (write-string "Input expression value: ")
859.        (pp (unparse-exp-value (flex-eval-empty flex-exp)))
860.        (newline)
861.        (newline)
862.        (write-string "Translated expression value: ")
863.        (pp (unparse-exp-value (flat-eval-empty flat-exp)))
864.        (newline)
```

```
865.        (newline)
866.        (write-string "Lifted expression value: ")
867.        (pp (unparse-exp-value (flat-eval-empty lifted-exp)))
868.        (loop))))
```