```
;;;----------------------------------------------------------------------------
;;; Update to NODE.SCM abstractions to handle FUNREC:
;;;
;;;  (funrec ((I1 (lambda (I*) E_1))
;;;           ...
;;;           (In (lambda (I*) E_n)))
;;;    E_body)
;;;

(define (funrec-node? node)
(eq? (node-type node) 'funrec))

(define (make-funrec names lams body)
`(FUNREC ,(map (lambda (name lam) `(,name ,lam))
                  i. names
                 ii. lams)
         b. ,body))

(define (funrec-names node)
(map first (second node)))

(define (funrec-lambdas node)
(map second (second node)))

(define (funrec-body node)
(third node))
```

```
(define *special-forms*
'(program lambda call if set! begin quote primop let
define ; Only should be a top-level
define-global global-ref global-set! ; Introduced by globalizing
call-generic call-closure            ; Closure conversion
if-zero if-non-zero constant         ; Data-conversion
integer boolean char string
unspecific null
error syscall
code call-code
funrec              ; ***
))

(define (subnodes node)
(cond
((leaf-node? node) '())
((lambda-node? node) (list (lambda-body node)))
; ((simple-let-node? node)
;  (cons (simple-let-body node) (simple-let-defs node)))
((let-node? node)
(cons (let-body node) (let-defs node)))
((application-node? node) (application-subexps node))
((primop-node? node) (primop-args node))
((syscall-node? node) (syscall-args node))
((assignment-node? node) (list (assignment-body node)))
((program-node? node)
(cons (program-body node) (map definition-body node)))

((funrec-node? node)
(cons (funrec-body node) (funrec-lambdas node)))

(else (node-subexps node))))
```

```
(define (subnode-map fn node)
;;
;; For compound (non-leaf) nodes, return a new compound node in which
;; FN has been applied to each subnode. Has no effect on leaf nodes.
;;
(cond
((leaf-node? node) node)
((lambda-node? node)
(make-lambda (lambda-formals node)
    (fn (lambda-body node))))
((let-node? node)
(make-let (let-names node)
    (map fn (let-defs node))
    (fn (let-body node))))
((assignment-node? node)
(new-assignment node (fn (assignment-body node))))
((primop-node? node)
(make-primop (primop-op node)
    (map fn (primop-args node))))
((syscall-node? node)
(make-syscall (syscall-op node)
    (map fn (syscall-args node))))
((program-node? node)
(let ((defs (program-defs node)))
(make-program
(map new-definition defs (map (compose fn definition-body) defs))
(fn (program-body node)))))
;;; *** NEW ***
((funrec-node? node)
(make-funrec (funrec-names node)
        (map fn (funrec-lambdas node))
        (fn (funrec-body node))))
;; ***********
(else (make-node (node-keyword node)
        (map fn (node-subexps node))))
))
```

```
(define (subnode-map-receive fn node leaf receive)
;;
;; Generalized version of SUBNODE-MAP that allows the return of
;; multiple results in a recursive tree accumulation over a node tree.
;; At any node, applies RECEIVE to:
;;
;;   (i) A node-making procedure specialized for the node that
;;       expects new subnodes as arguments.
;;  (ii) A rest arg that is the result of applying FN to all of the
;;       subnodes. In general,   FN will return a compound structure only
;;       one component of which is the new node.
;;
;; Since leaf nodes have no subnodes, the LEAF procedure is applied
;; to leaf nodes to generate the appropriate base case for the
;; recursive tree accumuation.
;;
(cond
((leaf-node? node)
(receive (lambda (ignore) node)
    a. (leaf node)))
((lambda-node? node)
(receive (lambda (body) (make-lambda (lambda-formals node) body))
    a. (fn (lambda-body node))))
((let-node? node)
(apply receive
(lambda (new-body . new-defs)
    a. (make-let (let-names node) new-defs new-body))
    b. (map fn (cons (let-body node) (let-defs node)))))
        ((assignment-node? node)
        (receive (lambda (body) (new-assignment node body))
    a. (fn (assignment-body node))))
        ((primop-node? node)
        (apply receive
        (lambda new-args (make-primop (primop-op node) new-args))
        (map fn (primop-args node))))
        ((syscall-node? node)
        (apply receive
        (lambda new-args (make-syscall (syscall-op node) new-args))
        (map fn (syscall-args node))))
        ((program-node? node)
        (let ((defs (program-defs node)))
        (apply receive
    a. (lambda (new-body . new-def-bodies)
            i. (make-program (map new-definition defs new-def-bodies)
                1. new-body))
```

```
        b. (map fn (cons (program-body node)
                         1. (map define-body defs))))))
113.        ;; *** NEW ***
114.        ((funrec-node? node)
115.        (apply receive
        a. (lambda (new-body . new-lambdas)
        b. (make-funrec (funrec-names node)
                         1. new-lambdas
                         2. new-body))
        c. (map fn (cons (funrec-body node) (funrec-lambdas node)))))
116.        ;; ***********
117.        (else
118.        (apply receive
119.        (lambda new-subnodes (make-node (node-keyword node) new-subnodes))
120.        (map fn (node-subexps node))))
121.        ))
```

```
(define (rewrite vars rewrite-ref rewrite-set! node)
;;
;; A simple substitution routine.
;; For each X in the set VARS of variable names:
;;   (i) replace every reference to X in NODE by the result of
;;       (REWRITE-REF X).
;;  (ii) replace every (SET! X <body>) in NODE by the result of
;;       (REWRITE-SET! X <rewritten-body>)
;;
;; Neither REWRITE-REF and REWRITE-SET! should return nodes with names
;; that might be captured by enclosing lambdas.
;;
(let walk ((vars vars)
(node node))
(cond
((set-empty? vars) node)  ;; Optimization
((and (var-node? node) (set-member? (var-name node) vars))
(rewrite-ref (var-name node)))
((and (set!-node? node) (set-member? (set!-name node) vars))
(rewrite-set! (set!-name node)
(walk vars (set!-body node))))
((lambda-node? node)
(let ((formals (lambda-formals node)))
(make-lambda formals
(walk (set-difference vars (list->set formals))
(lambda-body node)))))
((let-node? node)
(let ((names (let-names node)))
(make-let names
(map (lambda (def) (walk vars def))
(let-defs node))
(walk (set-difference vars (list->set names))
(let-body node)))))
((program-node? node)
(let* ((defs (program-defs node))
(names (map definition-names defs))
(new-vars (set-difference vars (list->set names))))
(make-program
(map (lambda (def)
(new-definition def
(walk new-vars (definition-body def))))
defs)
(walk new-vars (program-body node)))))
;; *** NEW ***
```

```
151.    ((funrec-node? node)
152.    (let ((new-vars (set-difference vars
                                a.  (list->set (funrec-names node)))))
153.    (make-funrec (funrec-names node)
                 i.  (map (lambda (def) (walk new-vars def))
                              1.  (funrec-lambdas node))
                ii.  (walk new-vars (funrec-body node)))))
154.    ;; **********
155.    (else (subnode-map (lambda (n) (walk vars n)) node))
156.    )))
```

```
(define (free-vars node)
(cond
((var-node? node) (set-singleton (var-name node)))
((assignment-node? node)
(set-union (set-singleton (assignment-name node))
    i. (free-vars (assignment-body node))))
((lambda-node? node)
(set-difference (free-vars (lambda-body node))
    i. (list->set (lambda-formals node))))
((let-node? node)
(set-union (map-union free-vars (let-defs node))
    i. (set-difference (free-vars (let-body node))
            a. (list->set (let-names node)))))
((program-node? node)
(set-difference
(set-union
(map-union free-vars (map define-body (program-defs node)))
(free-vars (program-body node)))
(list->set (map define-name (program-defs node)))))
;; *** NEW ***
((funrec-node? node)
(set-difference
(set-union (map-union free-vars (funrec-lambdas node))
    i. (free-vars (funrec-body node)))
(list->set (funrec-names node))))
;; **********
(else (map-union free-vars (subnodes node)))
))

(define (free-mutables node)
;;
;; New function (not in NODE.SCM).
;; Finds all free vars in node that are assigned via SET!
;;
(cond
((var-node? node) the-empty-set)
((assignment-node? node)
(set-union (set-singleton (assignment-name node))
    i. (free-mutables (assignment-body node))))
((lambda-node? node)
(set-difference (free-mutables (lambda-body node))
    i. (list->set (lambda-formals node))))
((let-node? node)
(set-union (map-union free-mutables (let-defs node))
    i. (set-difference (free-mutables (let-body node))
```

a. (list->set (let-names node)))))
193. ((program-node? node)
194. (set-difference
195. (set-union
196. (map-union free-mutables (map define-body (program-defs node)))
197. (free-mutables (program-body node)))
198. (list->set (map define-name (program-defs node)))))
199. ;; *** NEW ***
200. ((funrec-node? node)
201. (set-difference
202. (set-union (map-union free-mutables (funrec-lambdas node))
i. (free-mutables (funrec-body node)))
203. (list->set (funrec-names node))))
204. ;; **********
205. (else (map-union free-mutables (subnodes node)))
206.))

```
;;;----------------------------------------------------------------------
;;; New implementation of closures supporting %CLOSURE-SHIFT to work

(define closure-tag '(closure))

(define (%closure . elts)
(vector closure-tag 0 (apply vector elts)))

(define (%closure-ref closure index)
(closure-check-index closure index)
(vector-ref (vector-ref closure 2)
    i. (+ index (vector-ref closure 1))))

(define (%closure-set! closure index new)
(closure-check-index closure index)
(vector-set! (vector-ref closure 2)
    i. (+ index (vector-ref closure 1))
   ii. new))

(define (closure-check-index closure index)
(let ((elts (vector-ref closure 2))
(real-index (+ index (vector-ref closure 1))))
(if (or (< real-index 0)
a. (>= real-index (vector-length elts)))
(error "CLOSURE: index out of range -- " index))))

(define (%closure-shift closure offset)
;;
;; Effectively returns a pointer into the middle of the closure.
;; Note that the result shares structure with the input.
;;
(let ((new-offset (+ offset (vector-ref closure 1)))
(elts (vector-ref closure 2)))
(if (or (< new-offset 0)
a. (>= new-offset (vector-length elts)))
(error "CLOSURE-SHIFT: Offset out of range -- " (list closure offset))
(vector closure-tag new-offset elts))))

(define (%closure? obj)
(and (vector? obj)
(= (vector-length obj) 3)
(eq? (vector-ref obj 0) closure-tag)))
```

```
;;;----------------------------------------------------------------------
;;; Extension to DESUGAR.SCM to catch assignments to FUNREC names
;;; (which are illegal).

(define-sugar 'funrec
(lambda (exp)
(define (lambda-exp? exp)
(and (list? exp)
     (>= (length exp) 3)
     (eq? (car exp) 'lambda)))
(define (check-lambda exp)
(if (not (lambda-exp? exp))
    (error "FUNREC: non-lambda expression" exp)
    exp))
(let ((bindings (second exp))
     (body-exps (cddr exp)))
(let ((names (map first bindings))
     (lams (map (compose check-lambda second) bindings)))
(let ((new-lams (map desugar lams))
     (new-body (make-desugared-begin
                  (map desugar body-exps))))
     (let ((illegal-mutables
               (set-intersection
                (list->set names)
                (map-union free-mutables
                           (cons new-body new-lams)))))
          (if (not (set-empty? illegal-mutables))
              (error "SYNTAX ERROR: FUNREC contains illegal assignments"
               illegal-mutables)
              `(FUNREC ,(map (lambda (name lam) `(,name ,lam))
                             names
                             new-lams)
                       ,new-body))))))))

;;;----------------------------------------------------------------------
;;; GLOBALIZE and ASSIGNMENT CONVERSION phases don't need to change.

;;;----------------------------------------------------------------------
;;; CPS-CONVERSION phase:

;;; Modify CPS to dispatch to CPS-FUNREC (below)
(define (cps node mcont)
;; MCONT here is a "meta-continuation" that maps a lettable value
;; (i.e., syntactic class W) into a syntactic continuation.
```

```
257.        (cond
258.        ((leaf-node? node) (mcont node))
259.        ((lambda-node? node) (cps-lambda node mcont))
260.        ((let-node? node) (cps-let node mcont))
261.        ((application-node? node) (cps-application node mcont))
262.        ((conditional-node? node) (cps-conditional node mcont))
263.        ((assignment-node? node) (cps-assignment node mcont))
264.        ((primop-node? node) (cps-primop node mcont))
265.        ((syscall-node? node) (cps-syscall node mcont))
266.        ((program-node? node) (cps-program node mcont))
267.        ((funrec-node? node) (cps-funrec node mcont))
268.        ;; ((begin-node? node) (cps-begin node mcont)) ; No longer supported
269.        (else (error "CPS: Don't know how to handle node:" node))))

270.        (define (cps-funrec node mcont)
271.        ;; Patterned after CPS-PROGRAM:
272.        (cps-list (funrec-lambdas node)
    a.  (lambda (Vs) ;; Guaranteed to be Vs because all are lambdas
    b.  (make-funrec (funrec-names node)
                1.  Vs
                2.  (cps (funrec-body node) mcont)))))
```

```
;;;----------------------------------------------------------------------
;;; Extension to RUNTIME.SCM to make FUNREC desugar into a LETREC within Scheme

; The local version
(define-syntax define-syntax-global
(macro (name expander)
`(begin
(define-syntax ,name ,expander)
(syntax-table-define system-global-syntax-table ',name ,expander))))

; The exported version
(syntax-table-define system-global-syntax-table
'define-syntax-global
(macro (name expander)
`(begin
(define-syntax ,name ,expander)
(syntax-table-define system-global-syntax-table ',name ,expander))))

(define-syntax-global define-macro-global
(macro (pattern . body)
`(DEFINE-SYNTAX-GLOBAL ,(car pattern)
(MACRO ,(cdr pattern) ,@body))))

(define-macro-global (funrec bindings . body)
`(LETREC ,bindings ,@body))
```

```
;;;----------------------------------------------------------------------
;;; Names for compiler passes

(define ->desugar (cascade initialize desugar abbreviate pp))

(define ->globalize (cascade initialize desugar globals/wrap abbreviate
  pp))

(define ->assign (cascade initialize desugar globals/wrap assignment-
  convert
                    1. abbreviate pp))

(define ->cps (cascade initialize desugar globals/wrap assignment-convert
                 i. cps-convert abbreviate pp))

;; Note: the following passes don't include an ORDER-CONVERT at the
  end,
;; but they could.

(define ->closures (cascade initialize desugar globals/wrap assignment-
  convert
                    1. cps-convert closurize abbreviate pp))

(define ->closures/no-cps
(cascade initialize desugar globals/wrap assignment-convert
    a. closurize abbreviate pp))

(define ->lift (cascade initialize desugar globals/wrap assignment-convert
                    1. cps-convert closurize lift-convert
                    2. abbreviate pp))

(define ->data (cascade initialize desugar globals/wrap assignment-convert
                    1. cps-convert closurize lift-convert
                    2. data-convert data-unconvert abbreviate pp))
```

```
307.    ;;;----------------------------------------------------------------------
308.    ;;; Code for the even/odd example:

309.    (define even/odd
310.    '(funrec ((even? (lambda (a) (if (= 0 a)
                                    a.  #t
                                    b.  (odd? (- a 1)))))
        b.  (odd? (lambda (b) (if (= 0 b)
                                    a.  #f
                                    b.  (even? (- b 1))))))
        c.  (even? 2)))
```