;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; CS 442 Assignment 2

;; Holden Li - h55li - 20300403

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Question 1

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define make-abs (lambda (var body) (list 'fun var body)))

(define make-app (lambda (rator rand) (list rator rand)))

(define abs? (lambda (expr)

(and (list? expr) (= (length expr) 3) (eqv? 'fun (car expr)))))

(define app? (lambda (expr)

(and (list? expr) (= (length expr) 2))))

(define var? symbol?)

(define var-of cadr)

(define body-of caddr)

(define rator-of car)

(define rand-of cadr)

;; Dynamic Sub

(define substd (lambda (e x expr)

(cond

((abs? expr)

(if (eqv? x (var-of expr))

expr ;; [e/x](fun x e1) = (fun x e1)

(make-abs ;; [e/x](fun y e1) = (fun y [e/x]e1)

(var-of expr) (substd e x (body-of expr))

)

)

)

((app? expr) ;; [e/x](e1 e2) = ([e/x]e1 [e/x]e2)

(make-app (substd e x (rator-of expr))

(substd e x (rand-of expr))

)

)

((var? expr)

(if (eqv? x expr)

e ;; [e/x]x = e

expr ;; [e/x]y = y

)

)

(else expr) ;; Error! Just return the expr

)

))

;; Check if var is a free variable in expr

(define free-in?

(lambda (var expr)

(cond

((abs? expr)

(if (eqv? var (var-of expr))

#f

(free-in? var (body-of expr))

)

)

((app? expr)

(or (free-in? var (rator-of expr))

(free-in? var (rand-of expr))

)

)

((var? expr)

(if (eqv? var expr)

#t

#f

)

)

(else #f) ;; Error! Just return true...

)

)

)

;; Return a "new" var

;; Currently just returns 'NEW assuming expr does not contain this var

(define newSuffix 0)

(define new-var-for

(lambda (expr)

(set! newSuffix (+ newSuffix 1))

(string->symbol (string-append "new" (number->string newSuffix)))

)

)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;

;; Substitution

;; - Implements Static Binding

(define subst (lambda (e x expr)

(cond

((abs? expr)

(if (eqv? x (var-of expr))

expr ;; [e/x](fun x e1) = (fun x e1)

(if (free-in? (var-of expr) e)

(let ((z (new-var-for (body-of expr))))

(make-abs ;; [e/x](fun y e1) = (fun z [e/x][z/y]e1)

z (subst e x (subst z (var-of expr) (body-of expr)))

)

)

(make-abs ;; [e/x](fun y e1) = (fun y [e/x]e1)

(var-of expr) (subst e x (body-of expr))

)

)

)

)

((app? expr) ;; [e/x](e1 e2) = ([e/x]e1 [e/x]e2)

(make-app (subst e x (rator-of expr))

(subst e x (rand-of expr))

)

)

((var? expr)

(if (eqv? x expr)

e ;; [e/x]x = e

expr ;; [e/x]y = y

)

)

(else expr) ;; Error! Just return the expr

)

))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Reduce

(define reduce

(lambda (expr)

(if (reducible? expr)

(reduce (reduction-step expr))

expr

)

)

)

(define reducible?

(lambda (expr)

(cond

((var? expr) #f)

((abs? expr) (reducible? (body-of expr)))

((app? expr)

(cond

((var? (rator-of expr)) (reducible? (rand-of expr)))

((abs? (rator-of expr)) #t)

((app? (rator-of expr))

(or (reducible? (rator-of expr)) (reducible? (rand-of expr)))

)

)

)

(else #f) ;; Error!

)

)

)

(define reduction-step

(lambda (expr)

;(display "#")(display expr)(newline)

(cond

((var? expr) expr)

((abs? expr) (make-abs (var-of expr) (reduction-step (body-of expr))))

((app? expr)

;(display "#app:")(display (rator-of expr))(display ":")(display (rand-of expr))(newline)

(cond

((var? (rator-of expr))

(make-app (rator-of expr) (reduction-step (rand-of expr)))

)

((abs? (rator-of expr))

;(display "!abs")(display expr)(newline)

(reduction-step (subst (rand-of expr) (var-of (rator-of expr)) (body-of (rator-of expr))))

)

((app? (rator-of expr))

;(display "!app")(display expr)(newline)

(if (reducible? (rator-of expr))

(make-app (reduction-step (rator-of expr)) (rand-of expr))

(make-app (rator-of expr) (reduction-step (rand-of expr)))

)

)

)

)

(else expr) ;; Error! Just return the expr

)

)

)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Parse Lambda

; Get rid of redundant paraentheses at the start of a list

(define strip-car

(lambda (expr)

(cond

((and (list? expr) (= 1 (length expr)))

(strip-car (car expr)))

(else expr)

)

)

)

; Get rid of extra parentheses in an expression

(define strip-expr

(lambda (expr)

(strip-car (strip expr))

)

)

; The actual work of strip-expr

(define strip

(lambda (expr)

(cond

; list

((and (not (null? expr)) (list? expr))

;(display "#list")(display expr)(newline)

(cond

; car is a list

((list? (car expr))

;(display "#nested: ")(display (car expr))(newline)

(cons (strip-expr (car expr)) (strip (cdr expr)))

)

; else

(else

(cons (car expr) (strip (cdr expr)))

)

)

)

; variable or null

(else expr)

)

)

)

; Add parentheses to abs

(define paren-abs

(lambda (expr)

;(display "#")(display expr)(newline)

(cond

((and (not (null? expr)) (list? expr)) ; non-empty list

(cond

; car is abs

((eqv? 'fun (car expr))

;(display "#abs ")(display (cdr (cdr expr)))(newline)

(if (= 1 (length (cdr (cdr expr))))

; body of abs doesn't need paren

(cons (car expr) ; 'fun

(cons (car (cdr expr)) ; variable

(paren-abs (cdr (cdr expr))))) ; expr

; body of abs needs paren

(cons (car expr) ; 'fun

(cons (car (cdr expr)) ; variable

(list (paren-abs (cdr (cdr expr)))))) ; expr

)

)

; cdr is abs

((and (< 1 (length expr)) (eqv? 'fun (car (cdr expr))))

;(display "#cdr ")(display (car expr))(newline)

(cons (paren-abs (car expr)) (list (paren-abs (cdr expr))))

)

; else

(else

(cons (paren-abs (car expr)) (paren-abs (cdr expr)))

)

)

)

; variable or null

(else expr)

)

)

)

; Add parentheses to app

(define paren-app

(lambda (expr)

(cond

((and (not (null? expr)) (list? expr)) ; non-empty list

(cond

; abs

((eqv? 'fun (car expr))

(cons 'fun

(cons (car (cdr expr)) ; variable

(paren-app (cdr (cdr expr)))) ; body

)

)

; app that can be paranthesized

((< 2 (length expr))

(paren-app

(cons (list (paren-app (car expr))

(paren-app (car (cdr expr))))

(cdr (cdr expr)))

)

)

; else

(else

(cons (paren-app (car expr)) (paren-app (cdr expr)))

)

)

)

(else expr)

)

)

)

; Add parentheses

(define parenthesize

(lambda (expr)

(paren-app (paren-abs expr))

)

)

; parse-lambda

(define parse-lambda

(lambda (expr)

(parenthesize (strip-expr expr))

)

)

; interpret

(define (interpret E) (reduce (parse-lambda E)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Question 1 Part D

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; INPUT:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define l-true

'(fun x fun y x))

(define l-false

'(fun x fun y y))

(define l-if

'(fun b fun t fun f b t f))

(define l-cons

'(fun h fun t fun s s h t))

(define l-car

'(fun l l fun x fun y x))

(define l-cdr

'(fun l l fun x fun y y))

(define l-nil

(append '(fun s) l-true))

(define l-null?

(append '(fun l l fun h fun t) l-false))

(define l-incr

'(fun n (fun s s ! n)))

(define l-decr l-cdr)

(define l-Y

'(fun f (fun x f (x x)) (fun x f (x x))))

(define Y-sum

`(fun s fun x fun y

,l-if (,l-null? x) ; if

y ; then

(s ; else

(,l-decr x)

(,l-incr y)

)

)

)

(define l-sum (list l-Y Y-sum))

(define Y-fib

`(fun s fun x

; if 0

,l-if (,l-null? x)

; then

(,l-nil)

; if 1

(,l-if (,l-null? (,l-decr x))

(,l-incr ,l-nil)

; else

(,l-sum (s (,l-decr x))

(s (,l-decr (,l-decr x))))

)

)

)

(define l-fib (list l-Y Y-fib))

; fib 0 = 0

(interpret `(,l-fib ,l-nil))

; fib 1 = 1

(interpret `(,l-fib (,l-incr ,l-nil)))

; fib 2 = 1

(interpret `(,l-fib

(,l-incr (,l-incr ,l-nil))

)

)

; fib 3 = 2

(interpret `(,l-fib

(,l-incr (,l-incr (,l-incr ,l-nil)))

)

)

; fib 5 = 5

(interpret `(,l-fib

(,l-incr (,l-incr (,l-incr (,l-incr (,l-incr ,l-nil)))))

)

)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; OUTPUT:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(fun s (fun x (fun y x)))

(fun s ((s !) (fun s (fun x (fun y x)))))

(fun s ((s !) (fun s (fun x (fun y x)))))

(fun s ((s !) (fun s ((s !) (fun s (fun x (fun y x)))))))

(fun s ((s !) (fun s ((s !) (fun s ((s !) (fun s ((s !) (fun s ((s !) (fun s (fun x (fun y x)))))))))))))