CS442 Assignment 1

Nissan Pow
20187246
npow

January 22, 2007

Question 1

Part A

```
(define (foldl f i L)
  (if (null? L)
      i
      (foldl f (f (car L) i) (cdr L))))


(define (foldr f i L)
  (if (null? L)
      i
      (f (car L) (foldr f i (cdr L)))))
```

Part B

1. (foldl + 0 L)
2. (foldl (lambda (x y) (+ y 1)) 0 L)
3. (foldl min (car L) L)
4. (foldl (lambda (x y) (or y (p x))) #f L)
5. (foldl (lambda (x y) (cons (f x) y)) '() L)
6. (foldr cons M L)

Part C

1. foldl is better to use since it uses tail recursion (foldl is called last), whereas in foldr, f is called last. And Scheme is optimized for tail-recursion.
2. The implementation listed is more efficient. Although the asymptotic runtimes are the same, the one listed will perform faster in the average case, since it can terminate immediately if an element is found.

3. 5.2.2 $[\![scc]\!] = \lambda n.\lambda s.\lambda z.(n\ s\ (s\ z))$
   5.2.3 $[\![times]\!] = \lambda m.\lambda n.\lambda s.n\ (m\ s)$
   5.2.4 $[\![pow]\!] = \lambda m.\lambda n.n\ m$
   5.2.5 $[\![sub]\!] = \lambda m.\lambda n.\ (n\ [\![prd]\!])\ m$

Question 2

Parts A and B

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; PART A
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Some functions for manipulating lambda expressions.
;;
;; (make-abs <var> <expr>) creates the encoding for an abstraction
;; (var-of <abs>) return the variable of an abstraction
;; (body-of <abs>) return the body of an abstraction
;;
;; (make-app <rator> <rand>) creates the encoding for an application
;; (rator-of <app>) return the function of an application
;; (rand-of <app>) return the argument of an application
;;
;; (abs? <expr>) predicate to identify abstractions
;; (app? <expr>) predicate to identify applications
;; (var? <expr>) predicate to identify variables
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define make-abs (lambda (var expr) (list 'fun var expr)))
(define make-app (lambda (rator rand) (list rator rand)))

(define abs? (lambda (expr)
  (and (list? expr) (= (length expr) 3) (eq? '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)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Substitution
;; - Implements Static Binding
;;
;; [e/x]y = y  (if x != y)
;; [e/x]x = e
;; [e/x](e1 e2) = ([e/x]e1 [e/x]e2)
;; [e/x](fun x e1) = (fun x e1)
;; [e/x](fun y e1) = (fun y [e/x]e1)  (if x != y, y not in FV[e])
;; [e/x](fun y e1) = (fun z [e/x][z/y]e1) (if x != y, y in FV[e], z is a "new" variable)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define subst (lambda (e x expr)
  (cond
    ((abs? expr)
     (if (eq? x (var-of expr))
       expr  ;; [e/x](fun x e1) = (fun x e1)
       (if (not (inlist? (var-of expr) (freevars e)))
         (make-abs (var-of expr) (subst e x (body-of expr)))
         (let ((newvar (nextvar)))
           (make-abs newvar (subst e x (subst newvar (var-of expr) (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 (eq? x expr)
       e   ;; [e/x]x = e
       expr  ;; [e/x]y = y
     )
    )
    (else expr)  ;; Error! Just return the expr
   )
))

;; returns a list containing FV[expr]
(define freevars
  (lambda (expr)
  (cond ((var? expr) (list expr))
    ((app? expr) (append (freevars (rator-of expr)) (freevars (rand-of expr))))
    ((abs? expr) (list-remove (var-of expr) (freevars (body-of expr))))
    (else ()))))

;; removes all occurences of x from L, and returns the new list
(define list-remove
```

```
  (lambda (x L)
  (if (null? L)
  '()
  (if (eq? (car L) x)
    (list-remove x (cdr L))
    (append (list (car L)) (list-remove x (cdr L)))))))

;; using npow[0,1,2,...] as the "new" variables
(define myvar "npow")
(define counter 0)

;; returns the next new variable
(define nextvar
  (lambda ()
  (set! counter (+ counter 1))
  (string->symbol (string-append myvar (number->string counter)))))

;; returns true if x is in L
;; else false
(define inlist?
  (lambda (x L)
  (if (null? L)
  #f
  (if (eq? x (car L))
    #t
    (inlist? x (cdr L))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; PART B
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define r1
  (lambda (expr)
  (cond
  ((var? expr) expr)
  ((abs? expr) (make-abs (var-of expr) (r1 (body-of expr))))
  ((app? expr) (cond
     ((var? (rator-of expr)) (make-app (rator-of expr) (r1 (rand-of expr))))
     ((abs? (rator-of expr)) (subst (rand-of expr)
                                    (var-of (rator-of expr))
                                    (body-of (rator-of expr))))
     ((app? (rator-of expr))
      (if (equal? (rator-of expr) (r1 (rator-of expr)))
        (make-app (rator-of expr) (r1 (rand-of expr)))
        (make-app (r1 (rator-of expr)) (rand-of expr))))
```

```
     (else expr)))
(else expr))))

(define reduce
  (lambda (expr)
  (if (equal? (r1 expr) expr)
  expr
  (reduce (r1 expr)))))

(define lc-true '(fun x (fun y x)))
(define lc-false '(fun x (fun y y)))
(define lc-if '(fun b (fun t (fun f ((b t) f)))))
(define lc-cons '(fun h (fun t (fun s ((s h) t)))))
(define lc-car `(fun l (l ,lc-true)))
(define lc-cdr `(fun l (l ,lc-false)))
(define lc-nil `(fun s ,lc-true))
(define lc-null? `(fun l (l (fun h (fun t ,lc-false)))))

;(equal? (reduce '((fun x (f x)) ((fun y (g y)) z))) '(f (g z)))
;(equal? (reduce '((fun x y) ((fun x (x x)) (fun x (x x))))) 'y)
;(equal? (reduce '(((fun x (fun y x)) z) w)) 'z)
;(equal? (reduce '(((fun x (fun y x)) y) w)) 'y)
;(equal? (reduce '((fun y z) w)) 'z)
;(equal? (reduce '((a b) c)) '((a b) c))

(define lc-one '(fun s ((s i) (fun s (fun x (fun y x))))))
(define lc-three '(fun s ((s i) (fun s ((s i) (fun s ((s i) (fun s (fun x (fun y x))))))))))

(define list1 `((,lc-cons a) ,lc-nil))
(define list2 `((,lc-cons a) ((,lc-cons b) ,lc-nil)))
(define list3 `((,lc-cons a) ((,lc-cons b) ((,lc-cons c) ,lc-nil))))
(define rev-list1 `((,lc-cons a) ,lc-nil))
(define rev-list2 `((,lc-cons b) ((,lc-cons a) ,lc-nil)))
(define rev-list3 `((,lc-cons c) ((,lc-cons b) ((,lc-cons a) ,lc-nil))))

(define cn-0 '(fun f (fun z z)))
(define cn-I '(fun f (fun z (f z))))
(define cn-II '(fun f (fun z (f (f z)))))
(define cn-III '(fun f (fun z (f (f (f z))))))

(define cn-plus '(fun m (fun n (fun s (fun z ((m s) ((n s) z)))))))
(define cn-mult '(fun m (fun n (fun f (n (m f))))))
(define cn-pow '(fun m (fun n (n m))))

(define cn-zero? `(fun m ((m (fun x ,lc-false)) ,lc-true)))
(define cn-zz `((,lc-cons ,cn-0) ,cn-0))
(define cn-ss `(fun p ((,lc-cons (,lc-cdr p)) ((,cn-plus ,cn-I) (,lc-cdr p)))))
```

```
(define cn-prd `(fun m (,lc-car ((m ,cn-ss) ,cn-zz))))
(define cn-sub `(fun m (fun n ((n ,cn-prd) m))))

(define F
  `(fun a
     (fun f
       (fun i
         (fun L
          (((,lc-if (,lc-null? L)) i) (((a f) ((f (,lc-car L)) i)) (,lc-cdr L))))))))
(define Y `(fun f ((fun x (f (x x))) (fun x (f (x x))))))
(define lc-foldl `(,Y ,F))
(define reverse `(fun L (((,lc-foldl ,lc-cons) ,lc-nil) L)))
(define lc-sum
  `(,Y (fun s
         (fun x
           (fun y
             (((,lc-if (,lc-null? x)) y) ((s (,lc-cdr x)) ((fun n (fun s ((s i) n))) y)))))))))
```

Part C

```
;No value

1 ]=> (define list1 `((,lc-cons a) ,lc-nil))

;Value: list1

1 ]=> (define rev-list1 `((,lc-cons a) ,lc-nil))

;Value: rev-list1

1 ]=> (equal? (reduce `(,reverse ,list1)) (reduce rev-list1))

;Value: #t

1 ]=> (define list2 `((,lc-cons a) ((,lc-cons b) ,lc-nil)))

;Value: list2

1 ]=> (define rev-list2 `((,lc-cons b) ((,lc-cons a) ,lc-nil)))

;Value: rev-list2

1 ]=> (equal? (reduce `(,reverse ,list2)) (reduce rev-list2))

;Value: #t
```

```
1 ]=> (define list3 `((,lc-cons a) ((,lc-cons b) ((,lc-cons c) ,lc-nil))))

;Value: list3

1 ]=> (define rev-list3 `((,lc-cons c) ((,lc-cons b) ((,lc-cons a) ,lc-nil))))

;Value: rev-list3

1 ]=> (equal? (reduce `(,reverse ,list3)) (reduce rev-list3))

;Value: #t

1 ]=> (transcript-off)
```

Question 3

1. $[\![sum]\!] = Y'(\lambda s.\lambda x.\lambda y.([\![zero?]\!]x)(\lambda z.y)(\lambda z.s([\![pred]\!]\ x)([\![succ]\!]\ y))\ z)$
2. Yes it should still work correctly. AOE uses η-expansion to "wrap" the results in a lambda expression, which NOR will reduce by η-reduction and produce the same results as in the original version of sum.
3. No. Consider $(\lambda x.\ ((\lambda y.\ x)\ z))$. Using NOR, this will reduce to $\lambda x.x$, but with AOE it will not reduce at all, since an abstraction is at the outermost level.