CS61A Exam Prep 10

FA21 Final Q4a

Implement `repeated-call`, a procedure that takes an `operator` expression and a list of `operand` expressions. It returns a repeated call for the `operator` and `operands`. If `operands` is `nil`, the result is the `operator` expression.

```
;;; Construct a repeated call expression from an operator and a list of operands.
;;;
;;; scm> (repeated-call 'f '(2 3 4))
;;; (((f 2) 3) 4)
;;; scm> (repeated-call '(f 2) '(3 4))
;;; (((f 2) 3) 4)
;;; scm> (repeated-call 'f nil)
;;; f
(define (repeated-call operator operands)
    (if (null? operands)
        operator
        (repeated-call (list operator (car operands)) (cdr operands))))
                       (a)              (b)            (c)
```

link

(cons

(list

'(

(f 2) (3 4)

((f 2) 3) (4)

(list operator (car operands))

((f 2) 3) 4) ()

(a) (4.0 points)

Implement the procedure `cons` that behaves just like the built-in `cons` when called on a value `x` and a (possibly empty) list `s`. You may not write `cons` in your solution.

```
(define (cons x s) ( _____ _____ _____ ))
;                    (a)   (b)   (c)
```

append (list x) s

FA 20 Final Q5a

i. (1.0 pt) Which of these could fill in blank (a)?

- ~~list~~
- append
- if
- map
- lambda

(cons a b)

link(a, b)

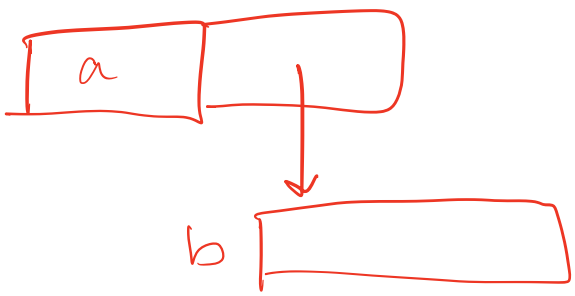
(list a b)

ii. (2.0 pt) Fill in blank (b).

(list x) '(x) (quote x)

iii. (1.0 pt) Which of these could fill in blank (c)? Check all that apply.

- [x] s
- [ ] (cdr s)
- [ ] (car s)
- [ ] (list s)



FA19 Final Q7a

Uh oh! Someone evaluated `(define * +)`. Now `(* 3 2)` evaluates to 5 instead of 6! Let's fix it.
Important: Answer all questions on this page without calling the built-in multiplication procedure.

(a) (3 pt) Implement `mulxy`, which multiplies integers `x` and `y`. **Hint:** `(- 2)` evaluates to -2.

```
;; multiply x by y (without using the * operator).
;; (mulxy 3 4) -> 12          ; 12 = 3 + 3 + 3 + 3
;; (mulxy (- 3) (- 4)) -> 12  ; 12 = - ( -3 + -3 + -3 + -3 )
(define (mulxy x y)

  (cond ((< y 0) (- _____ ))

        ((= y 0) 0)

        (else ( _____ x (mulxy x _____ )))))
```

x = -3, y = -4 } -12

-3 4 = -12

(mulxy x (- y))

(+ x (mulxy x (+ y 1)))

+

(- y 1)

FA19 Final Q7b

(b) (2 pt) Implement `mul-expr`, which takes an expression `e` that contains only calls to `*` and numbers. It returns the normal value of `e` under a Scheme interpreter with an unmodified `*` operator that multiplies. You may call the `mul` procedure defined below.
Important: Fill each blank with only a single symbol.

```
;; Multiply together a list of numbers.
;; (mul '(2 3 4 2)) -> 48
(define (mul s) (reduce mulxy s))

;; Evaluate an expression with only calls to * and numbers.
;; (mul-expr '(* (* 1 2) (* 3 (* 4 1 1) 2))) -> 48
(define (mul-expr e)

  (if (number? e) e

      (_____ (_____ _____ (_____ e)))))
```

(1 2 3 4)

2

24

(* 1 2)

(1 2)

(mul (cdr e))

mul

map

mul-expr

cdr

(2 24)

FA19 Final Q7c

(c) (5 pt) Implement `*-to-mul`, which takes any expression `e`. It returns an expression like `e`, but with all calls to `*` replaced with calls to `mul`. Note that `*` takes an arbitrary number of arguments, while `mul` always takes exactly one argument: a list of numbers. You should account for this difference.

```
;; Convert all calls to * to calls to mul in expression e.
;; (eval (*-to-mul '(* 1 (+ 2 3) (+ 4 5 (* 6 1))))) -> 75
(define (*-to-mul e)
  (if (not (list? e)) e

    (let ((op _____________ ) (rest _____________________________________________ ))

      (if (equal? op '*)

          (list ____________________________________________________________________ )

          (cons op rest)))))
```

go.cs61a.org/melanie-feedback