sicp-ex-2.27

<< Previous exercise (2.26) | Index | Next exercise (2.28) >>

jz

```
;; A value for testing.
(define x (list (list 1 2) (list 3 (list 4 5))))

;; My environment doesn't have nil.
(define nil '())


;; Here's reverse for reference:
(define (reverse items)
  (define (rev-imp items result)
    (if (null? items)
        result
        (rev-imp (cdr items) (cons (car items) result))))
  (rev-imp items nil))

;; Usage:
(reverse x)


;; Deep reverse.  Same as reverse, but when adding the car to the
;; result, need to check if the car is a list.  If so, deep reverse
;; it.

;; First try:
(define (deep-reverse items)
  (define (deep-rev-imp items result)
    (if (null? items)
        result
        (let ((first (car items)))
          (deep-rev-imp (cdr items)
                   (cons (if (not (pair? first))
                             first
                             (deep-reverse first))
                         result)))))
  (deep-rev-imp items nil))

;; Usage:
(deep-reverse x)


;; Works, but it's a bit hard to read?  Refactoring:

(define (deep-reverse-2 items)
  (define (deep-rev-if-required item)
    (if (not (pair? item))
        item
        (deep-reverse-2 item)))
  (define (deep-rev-imp items result)
    (if (null? items)
        result
        (deep-rev-imp (cdr items)
                      (cons (deep-rev-if-required (car items))
                            result))))
  (deep-rev-imp items nil))

;; Usage:
(deep-reverse-2 x)
```

Here's Eli Bendersky's code, translated into Scheme. It's pretty sharp, and better than my own since it's more concise:

```
(define (eli-deep-reverse lst)
  (cond ((null? lst) nil)
        ((pair? (car lst))
         (append
          (eli-deep-reverse (cdr lst))
          (list (eli-deep-reverse (car lst)))))
        (else
         (append
          (eli-deep-reverse (cdr lst))
          (list (car lst))))))

(eli-deep-reverse x)
```

This works for me:

```
(define (deep-reverse x)
  (if (pair? x)
      (append (deep-reverse (cdr x))
              (list (deep-reverse (car x))))
      x))
```

A solution that uses `reverse` to do the work:

```
(define (deep-reverse t)
  (if (pair? t)
      (reverse (map deep-reverse t))
      t))
```

shyam

Another solution without append

```
(define (deep-reverse items)
  (define (iter items result)
    (if (null? items)
        result
        (if (pair? (car items))
            (let ((x (iter (car items) ())))
              (iter (cdr items) (cons x result)))
            (iter (cdr items) (cons (car items) result)))))
  (iter items ()))
```

varoun

Solution that is a simple modification of reverse

```
(define (deep-reverse tree)
  (define (iter t result)
    (cond ((null? t) result)
          ((not (pair? (car t)))
           (iter (cdr t) (cons (car t) result)))
          (else
           (iter (cdr t) (cons (deep-reverse (car t)) result)))))
  (iter tree '()))

#|
> (deep-reverse '((1 2) (3 4)))
'((4 3) (2 1))
> (deep-reverse '(1 2 (3 4) 5 (6 (7 8) 9) 10))
'(10 (9 (8 7) 6) 5 (4 3) 2 1)
>
|#
>>>
```

meteorgan

there is another solution. it may be simpler.

```
(define (deep-reverse li)
  (cond ((null? li) '())
        ((not (pair? li)) li)
        (else (append (deep-reverse (cdr li))
                      (list (deep-reverse (car li)))))))
```

Daniel-Amariei

Took me a while to implement it without append and reverse.

```
(define (deep-reverse L)
  (define (rev L R)
    (cond ((null? L) R)
          ((not (pair? (car L))) (rev (cdr L)
                                      (cons (car L) R)))
          (else (rev (cdr L)
                     (cons (rev (car L) '())
                                R))))))
  (rev L '()))

(define x '((1 2) (3 4)))
(deep-reverse x) ;; ((4 3) (2 1))
```

atrika

Solution with no use of `append`. Would be nice to have a full iterative process, but this problem is naturally recursive.

```
 (define (deep-reverse l)
   (define (update-result result picked)
     (cons (if (pair? picked) (deep-reverse picked) picked) ;; recursive
process
           result))

   (define (iter source result)
     (if (null? source)
         result
         (iter (cdr source) (update-result result (car source)))))

   (iter l '()))


 ;; testing
 (deep-reverse '(1 2 (a b c (d1 d2 d3)) 4)) ;; returns '(4 ((d3 d2 d1) c b a) 2
1)
```

adam

My solution which is very similar to the original.

```
 (define (deep-reverse items)
   (define (try-deep item)
     (if (not (list? item))
         item
         (iter item '())))

   (define (iter old new)
     (if (null? old)
         new
         (iter (cdr old)
               (cons (try-deep (car old)) new))))

   (iter items '()))

 ;; Testing
 (define x (list (list 1 2) (list 3 (list 4 5)) (list (list 2 3) 3)))
 ;; ((1 2) (3 (4 5)) ((2 3) 3))
 (deep-reverse x)
 ;; ((3 (3 2)) ((5 4) 3) (2 1))
```

yves

I though I'll found my version. Very close to some version here tough, but using map for clarity. It looks like it is working. May I be wrong somewhere?

```
 (define (deep-reverse tree)
   (cond ((null? tree) nil)
         ((not (pair? tree)) tree)
         (else (map deep-reverse (reverse tree)))))
```

```
(display (deep-reverse (list (list 1 2) (list 3 4))))
(newline)
(display (deep-reverse (list (list 1 (list 5 6)) (list 3 4))))
(newline)
```

Lambdalef

The most briefly solution

This solution only works with list of lists

```
(define (deep-reverse l)
(reverse (map reverse l)))
```

DeepDolphin

This solution doesn't require to check for the end of list.

```
          (define (deep-reverse lst)
        (if (not (pair? lst))
            lst
            (append (deep-reverse (cdr lst))
                    (list (deep-reverse (car lst))))))

;;Testing
(define x (list (list 1 2) (list 3 (list 4 (list 5 6 7) (list 8 9 10) 11))))
(deep-reverse (deep-reverse x))
;; ((1 2) (3 (4 (5 6 7) (8 9 10) 11)))
```

joshwarrior

```
          (define (reverse item)
            (define (reverse-iter item result)
      (if (null? item)
          result
          (reverse-iter (cdr item) (cons (car item) result))))
    (reverse-iter item nil))

 (define (deep-reverse item)
   (define (deep-reverse-iter item result)
     (cond ((null? item) result)
           ((pair? (car item)) (deep-reverse-iter (cdr item) (cons (deep-
reverse (car item)) result)))
           (else (deep-reverse-iter (cdr item) (cons (car item) result)))))
   (deep-reverse-iter item nil))

 ;;Testing
 > (define x (list (list 1 2) (list 3 4)))
 > x
 (mcons (mcons 1 (mcons 2 '())) (mcons (mcons 3 (mcons 4 '())) '()))
 > (reverse x)
 (mcons (mcons 3 (mcons 4 '())) (mcons (mcons 1 (mcons 2 '())) '()))
 > (deep-reverse x)
 (mcons (mcons 4 (mcons 3 '())) (mcons (mcons 2 (mcons 1 '())) '()))
 > (deep-reverse (list 1 2 3))
 (mcons 3 (mcons 2 (mcons 1 '())))
```

```
> (deep-reverse (list (list 1 2 3)))
(mcons (mcons 3 (mcons 2 (mcons 1 '()))) '())
```