Homework 5. Listdiffs

A *listdiff* is a pair whose car is *L* and whose cdr is `eq?` to either *L*, or to `(cdr` *L*`)`, or to `(cdr (cdr` *L*`))`, etc. The cdr of a listdiff need not be a list; it may be any object.

A listdiff *D* represents the prefix of `(car` *D*`)` that precedes `(cdr` *D*`)`. For example, suppose `ils` is the improper list `(a e i o u . y)`. Then `(cons ils ils)` returns an empty listdiff, `(cons ils (cdr (cdr ils)))` returns a listdiff with the same elements as the list `(a e)`, and `(cons (cdr ils) 'y)` returns a listdiff with the same elements as `(e i o u)`. Conversely, neither `(cons '() ils)` nor `(cons ils (append '(a e i o u) 'y))` returns a listdiff.

Listdiffs are intended to be efficient representations for sublists. Normally if you want to represent a sublist of length *N* nondestructively, you must invoke `cons` *N* times to copy the *N* pairs in the sublist. However, with a listdiff you must invoke `cons` only once, to create the pair whose car is the first element of the sublist and whose cdr is the first element after the end of the sublist.

Define the following procedures, which are specified using the entry format described in R6RS §6.2, except that the label "procedure" is omitted. All but the last three procedures have semantics similar to the standard Scheme procedures for lists; for example, `cons-ld` is to listdiffs as `cons` is to lists.

`(null-ld?` *obj*`)`
: Return `#t` if *obj* is an empty listdiff, `#f` otherwise.

`(listdiff?` *obj*`)`
: Return `#t` if *obj* is a listdiff, `#f` otherwise.

`(cons-ld` *obj listdiff*`)`
: Return a listdiff whose first element is *obj* and whose remaining elements are *listdiff*. (Unlike `cons`, the last argument cannot be an arbitrary object; it must be a listdiff.)

`(car-ld` *listdiff*`)`
: Return the first element of *listdiff*. It is an error if *listdiff* has no elements. ("It is an error" means the implementation can do anything it likes when this happens, and we won't test this case when grading.)

`(cdr-ld` *listdiff*`)`
: Return a listdiff containing all but the first element of *listdiff*. It is an error if *listdiff* has no elements.

`(listdiff` *obj* …`)`
: Return a newly allocated listdiff of its arguments.

`(length-ld` *listdiff*`)`
: Return the length of *listdiff*.

`(append-ld` *listdiff* …`)`
: Return a listdiff consisting of the elements of the first *listdiff* followed by the elements of the other *listdiffs*. The resulting listdiff is always newly allocated, except that it shares structure with the last argument. (Unlike `append`, the last argument cannot be an arbitrary object; it must be a listdiff.)

`(assq-ld` *obj alistdiff*`)`
: *alistdiff* must be a listdiff whose members are all pairs. Find the first pair in *alistdiff* whose car field is `eq?` to *obj*, and return that pair; if there is no such pair, return `#f`.

`(list->listdiff` *list*`)`
: Return a listdiff that represents the same elements as *list*.

`(listdiff->list` *listdiff*`)`
: Return a list that represents the same elements as *listdiff*.

`(expr-returning` *listdiff*`)`
: Return a Scheme expression that, when evaluated, will return a copy of *listdiff*, that is, a listdiff that has the same top-level data structure as *listdiff*. Your implementation can assume that the argument listdiff contains only booleans, characters, numbers, and symbols.

Your implementation of the above routines must be free of side effects. Also, except for `expr-returning`, your implementation should avoid using unnecessary storage: e.g., it should minimize the number of calls to `cons`, and when it uses recursion it should when possible use tail recursion. Returned values may share storage with arguments; they need not copy their arguments.

The expressions that `expr-returning` returns should avoid generating objects that are not used in the final result, and should restrict themselves to the following tiny subset of Scheme:

- Numerical constants, character constants, and boolean constants.
- `(quote` *s*`)` where *s* is a symbol, is `()`, or is one of the just-mentioned constants.
- Calls to `cons`, `car`, `cdr`, and `list`.

Submit

Submit one file:

`hw5.scm`
: should contain your procedure definitions. Your implementation should work on the SEASnet installation of `racket`, the Scheme implementation installed on SEASnet.

Here are a few examples:

```
(define ils (append '(a e i o u) 'y))
(define d1 (cons ils (cdr (cdr ils))))
(define d2 (cons ils ils))
(define d3 (cons ils (append '(a e i o u) 'y)))
(define d4 (cons '() ils))
(define d5 0)
(define d6 (listdiff ils d1 37))
(define d7 (append-ld d1 d2 d6))
(define e1 (expr-returning d1))

(listdiff? d1)                          ===>  #t
(listdiff? d2)                          ===>  #t
(listdiff? d3)                          ===>  #f
(listdiff? d4)                          ===>  #f
(listdiff? d5)                          ===>  #f
(listdiff? d6)                          ===>  #t
(listdiff? d7)                          ===>  #t

(null-ld? d1)                           ===>  #f
(null-ld? d2)                           ===>  #t
(null-ld? d3)                           ===>  #f
(null-ld? d6)                           ===>  #f

(car-ld d1)                             ===>  a
(car-ld d2)                             ===>  error
(car-ld d3)                             ===>  error
(car-ld d6)                             ===>  (a e i o u . y)

(length-ld d1)                          ===>  2
(length-ld d2)                          ===>  0
(length-ld d3)                          ===>  error
(length-ld d6)                          ===>  3
(length-ld d7)                          ===>  5

(define kv1 (cons d1 'a))
(define kv2 (cons d2 'b))
(define kv3 (cons d3 'c))
(define kv4 (cons d1 'd))
```

```
(define d8 (listdiff kv1 kv2 kv3 kv4))
(eq? (assq-ld d1 d8) kv1)              ===>  #t
(eq? (assq-ld d2 d8) kv2)              ===>  #t
(eq? (assq-ld d1 d8) kv4)              ===>  #f

(eq? (car-ld d6) ils)                  ===>  #t
(eq? (car-ld (cdr-ld d6)) d1)          ===>  #t
(eqv? (car-ld (cdr-ld (cdr-ld d6))) 37)===>  #t
(equal? (listdiff->list d6)
        (list ils d1 37))              ===>  #t
(eq? (list-tail (car d6) 3) (cdr d6))  ===>  #t

(listdiff->list (eval e1))             ===>  (a e)
(equal? (listdiff->list (eval e1))
        (listdiff->list d1))           ===>  #t
```

$Id: hw5.html,v 1.48 2016/11/17 20:45:25 eggert Exp $