

This lab is about ripping apart and stitching together Lisp-style lists. The procedures `down` and `up` are filched from Exercises 1.17 and 1.26 in *EOPL*. You may work on this lab singly or in pairs.

## 1. Building List: A Quick Review

The procedures `cons`, `list`, and `append` build lists as follows.

`(cons d l)`

Given a datum `d` and list `l`, then `cons` returns a new list with `d` as its car and `l` as its cdr.

`(list d1 ... dn)`

Given datums `d1` through `dn`, then `list` returns a new list made up of `d1 ... dn`.

`(append l1 ... ln)`

Given lists `l1` through `ln`, then `append` returns a new list consisting of the concatenation of the elements of `l1 ... ln`.

Examples:

```
> (cons 'a '(x y z))
```

```
(a x y z)
```

```
> (cons '(a b) '(a b))
```

```
((a b) a b)
```

```
> (list 1 2 3)
```

```
(1 2 3)
```

```
> (list 'a '(x y z))
```

```
(a (x y z))
```

```
> (list '(a b) '(a b))
```

```
((a b) (a b))
```

```
> (append 'a '(x y z))
```

```
✗append: expects argument of type
  <proper list>; given a
```

```
> (append '(a b) '(a b))
(a b a b)
```

## 2. Down to Business

Let us consider how to define a procedure `down` such that `(down lst)` wraps parentheses around each top-level element of `lst`. E.g.,

```
> (down '(1 2 3))
```

```
((1) (2) (3))
```

```
> (down '(a (more (complex)) object))
```

```
((a) ((more (complex))) (object))
```

As with most procedures that work on lists, the obvious approach is to try a recursive procedure. Taking that approach, we first have to determine what sorts of lists we may encounter and how to treat the different options.

In the case of `down`, all we know is that we will have a list. We need not worry about the types of the elements in the list, because we will treat them all the same: every item (be it a list or a number or ...) gets an extra pair of parentheses wrapped around it. Therefore, we only have to worry about whether or not the list is empty.

Empty lists are easy here: there aren't any elements to wrap in parentheses; hence, we can simply return the empty list. Thus, the following suffices for the shell of our procedure:

```
(define down
  (lambda (lst)
    (if (null? lst)
        '()
        'to-be-filled-in)))
```

This procedure returns `()` when called with the

empty list, and `(right now)` returns the symbol `to-be-filled-in` otherwise. This symbol can be viewed as a *stub*, which will eventually be filled in with the real code. Using stubs can help in writing and debugging procedures, especially recursive ones, because they help you isolate errors.

For nonempty lists, we have some more work to do. Let us start by thinking about a particular example—say, the list `(a b)`—to determine what to do. Remember that our goal is to end up with the list `((a) (b))`.

The general recursive approach involves three steps:

- (1) working on the first element of the list (`a`, in this case),
- (2) making a recursive call with the rest of the list (`(b)`, here), and then,
- (3) combining those results.

We consider these steps one at a time.

### STEP 1: Working on the first element

We want to transform `a` into `(a)`. From previous experience, we know that either `(list 'a)` or `(cons 'a '())` will do the right thing. More generally, for any list `lst`, `(list (car lst))` or `(cons (car lst) '())` will put parens around `lst`'s first element. Although this may be pretty easy to see directly, we could also try modifying our stub to check:

```
(define down
  (lambda (lst)
    (if (null? lst)
        '()
        (list (car lst)))))
```

If we evaluate `(down '(a b))` now, we get the list `(a)`; this tells us that we are doing the work on the

first element correctly. Of course, we still have to handle the rest of the list.

### STEP 2: Making a recursive call with the rest of the list

The recursive call itself is pretty standard here: we simply need to call `down` with `(cdr lst)` (i.e., the list `(b)` in our example). Assuming we have the stopping behavior correct (i.e., when `lst` is empty) and assuming we combine the first-of-list and rest-of-list results (still to be discussed) correctly, there is not much to do on the recursive call. When these assumptions hold (unfortunately, they do not hold yet), the result of `(down '(b))` will be the list `((b))`.

### STEP 3: Combining the results of steps 1 and 2

Finally, we need to combine the partial results `(a)` and `((b))` to yield the desired list `((a) (b))`. We have three main ways to combine list results: `cons`, `list`, and `append`. It is simple enough to try each of the following expressions to determine which one we need: `(cons '(a) '((b)))`, `(list '(a) '(b))`, and `(append '(a) '(b))`. In this case, `cons` is just what we want. We can now fill in our stub as follows:

```
(define down
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (list (car lst))
              (down (cdr lst))))))
```

Try the finished procedure on some test cases and verify that it works correctly. What happens if `cons`

is replaced by `list` or `append`? (You may want to try them out, as preparation for section 3's problems.)



## 3. Your Problems

**Problem 1.** Give box-and-arrow diagrams for the list structure produced by each of the following expressions.

- (a) `(cons '(a b) '(a b))`
- (b) `(list 1 2 3)`
- (c) `(list 'a '(x y z))`
- (d) `(list '(a b) '(a b))`
- (e) `(append '(a b) '(a b))`

For a sample box-and-arrow diagram, see Dybvig's Exercise 2.2.5 (url: [www.scheme.com/tspl3/start.html#g11](http://www.scheme.com/tspl3/start.html#g11)), an answer to which is: `'((a . b) ((c) d) ())`.

**Problem 2.** Consider the following definitions of `x`, `y`, and `z`:

```
(define x '(a b c) )
(define y '((d) ((e f) g)) )
(define z '((h (i) j)) )
```

For each of the following values, given an expression that evaluates to it, *where your expression must contain ONLY the following*:

- the procedures `car`, `cdr`, `caar`, `cadr`, ...
- the procedures `cons`, `list`, and `append`
- the variables `x`, `y`, and `z`

For example, `(d b c)` can be obtained by evaluating the expression `(append (car y) (cdr x))`.

- (a) `(c f)`
- (b) `i`
- (c) `(e . c)`
- (d) `(h d)`
- (e) `(g j b)`

**Problem 3.** Write a recursive procedure `up` such that `(up lst)` removes a pair of parentheses from each top-level element of `lst`. If a top-level element is not a list, it is included in the result, as is. (Note: The value of `(up (down lst))` is equivalent to `lst`, but `(down (up lst))` is not necessarily `lst`.) Here are some examples:

```
> (up '((1 2) () (3 4)) )
(1 2 3 4)
> (up '((x (y) ()) z) )
(x (y) () z)
```

Be sure to test it out on some interesting values. *Hint:* For this problem, you may find the procedure `append` as useful as `cons`. Also, if you find yourself using nested `ifs`, use a `cond` instead.

**What to hand in:**

- your drawings for Problem 1,
- runs of your answers for Problem 2, and
- your source code for `up` together with test runs that verify that your version of `up` works as required.

Put your assignment in the labeled bin in CST 3-212. You do not have to submit anything electronically for this lab.

*Also:* Groups should save some trees and submit one copy of the lab for the entire group.