Homework Assignment #1

Due Friday, October 13, 2000

(All assignments will be turned in at start of lecture.)

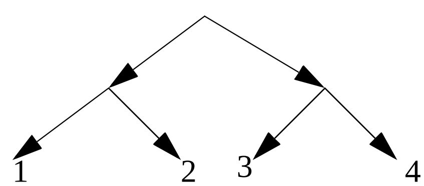
1. Lists allow fast access to the first element, but are slow to access the last element since the whole list is scanned; lists do not support efficient random access. Implement an alternative list data structure that supports fast access to the last element, but slow access to the first element, exactly opposite to standard lists. In particular, implement the following three functions: `rcons` (take a list and an element and return a new list with the element added to the *end*), `rcar` (take a list and return its *last* element), and `rcdr` (take a list and return a list of all elements but the *last*). It does not matter how your lists print out, only that they support `rcons`, `rcar`, and `rcdr` correctly and efficiently (each in constant time). Hint: the solution is very short!

2. In Scheme, binary trees can be represented using cons cells for internal nodes to hold pointers to the left subtree (in `car`) and right subtree (in `cdr`); empty subtrees can be represented with `nil`, leaf nodes with non-list values (like an integer). The following expression, for example, constructs a simple tree with four leaf nodes:

```
(cons (cons 1 2) (cons 3 4))    → ((1 . 2) 3 . 4)
```

(Scheme tries to print it as a list which it isn't.)

The tree looks like this:



The following higher-level functions can be used to work with trees represented this way:

```
(define (make-empty) ())
(define (make-leaf l) l)
(define (make-tree left right) (cons left right))

(define (internal-node? t) (pair? t))
(define (empty? t) (null? t))
(define (leaf? t) (not (or (internal-node? t) (empty? t))))
```

Using these functions, write a function `sum-tree` that takes a tree whose leaves are numbers and returns their sum. You don't have to handle trees whose leaves are not numbers.

Write `sum-tree` tail-recursively if possible, without using an auxiliary data structure whose size is proportional to the depth of the recursion. If not possible, explain why.

3. Write a `reduce-tree` function that works on trees rather than lists by visiting all the leaves of the tree in left-to-right order. Analogously to the list case, it should have three arguments: a binary reduction function; its identity element, returned in case of an empty tree; and the tree itself. Redefine the `sum-tree` function using `reduce-tree` in a minimal way.

4. Write a `map-tree` function that takes a unary function and a tree and constructs a new tree whose leaves are mapped from the old tree's leaves using the unary function. Give an interesting example of its use.

5. Recursive functions on lists follow a common pattern, as discussed in class. Implement a higher-order function, `process-list`, that abstracts this pattern. `process-list` should take a zero-argument function which is invoked to implement the base case, a two-argument function which is invoked on the head of the list and the result of recursively processing the rest of the list to implement the inductive case, and the list to be processed. Show how `sum`, `append`, `assoc`, `map`, and `reduce` can be reimplemented in terms of `process-list`.

6. Change the meta-circular Scheme interpreter from the course web page in the following ways:

- It should implement lexical instead of dynamic scoping. Demonstrate that your new interpreter correctly supports lexical scoping, and that the old interpreter implements dynamic scoping, on a few carefully chosen examples.
- It should support a `(set! name expr)` expression, which finds the current binding for `name` and modifies it to bind to the result of evaluating `expr` instead. Hint: the existing `lookup-binding` function should be helpful.
- It should support a `(begin expr...)` expression, which takes a non-empty list of expressions and evaluates them in turn, returning the value of the last expression.

Clearly mark you changes to the underlying interpreter in your answer.