Fall 2014 **Programming Project #3** CMSC 420

Hanan Samet

Program the following 11 functions in LISP. Make sure you test them thoroughly. Pay particular attention to the efficiency of your solutions. Test data will be mailed to you. Turn in a run that includes both a pretty printing of your functions and the execution of said functions on the test data.

1. Ordered lists of numbers (with duplicates):

   (a) Write a function `mergelists[x, y]` which takes two ordered lists `x` and `y`, and makes one ordered list from them, for example,

   `mergelists['(2 3 4), '(1 4)] = (1 2 3 4 4).`

   Your algorithm should run in time proportional to the number of elements in the two lists.

   (b) Using `mergelists`, write a function `sortlist[l]` which takes an unordered list `l` and makes an ordered list of it, for example,

   `sortlist['(1 7 3 5 3)] = (1 3 3 5 7).`

   For an initial list of n elements, your algorithm should run in $O(n \log n)$ time and not $O(n^2)$ time.

   (c) Write a predicate `dup[l]` which indicates if any atom occurs more than once in an unordered list `l`, for example,

   `dup['(1 3 5 3)] = t`

   The algorithm should be as efficient as `sortlist`. Make sure you compare numbers with `equal` and not `eq` (`eq` will generally not return `t` for two equal numbers if they are sufficiently large.)

2. Lists of lists of numbers:

   (a) Write a function `countlists[l]` which counts the number of top level lists in a list of lists, for example,

   `countlists['((1 2) (1 3) (1 4))] = 3.`

   (b) Lexical ordering on lists of numbers is a binary relation defined by:

   ```
   lex-lt[x, y] = if null[x] or null[y] then null[x]
                  else if car[x]=car[y] then lex-lt[cdr[x], cdr[y]]
                  else car[x]<car[y]
   ```

   Write a function `duplist_of_lists[l]` which returns `t` if any of the lists in a list of lists `l` are identical. `duplist_of_lists` should be as efficient as `sortlist`.

   (c) Given a list of numbers, there are several ways to obtain a list of all permutations of these numbers. For example, the set of permutations of `'(1 2 3)` is `'((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))`. Note that a list of n numbers has $n!$ permutations. There are no duplications in the list and all sublists have the same number of elements. One strategy, though not very efficient, is to take out each element, say `a`, in turn from the list, permute the rest, and then attach `a` to the front of each permutation. Write a function `permute[l]` to implement this method.

3. S-expressions of numbers:

   (a) A cons_tree of a nonempty list x containing no nils, is defined as

```
cons_tree[x] =
  if null[cdr[x]] then car[x]
    if x has 2^n elements then
      cons[cons_tree[first 2^(n-1) elements of x]
           cons_tree[second 2^(n-1) elements of x]]
    else cons_tree[append[x, '(nil . nil)]].
```

   For example,

```
cons_tree['(1 2 3 4 5)]
  = (((1 . 2) . (3 . 4)) . ((5 . nil) . (nil . nil))).
```

   Write a function make_cons_tree[l] that sorts an unordered list of numbers l and then returns the corresponding cons_tree, for example,

```
make_cons_tree['(3 2 1)] = ((1 . 2) . (3 . nil))
                         = ((1 . 2) 3)
```

   (b) The natural way to write make_cons_tree is to follow the definition closely. This can be rather inefficient due to the use of operations that repeatedly scan the list in a top-down manner in order to construct lists whose lengths are powers of two. Write a function squeeze[l] that returns the cons_tree of a list l, but doesn't use any operations that compute the length of a list. Thus you will do it in a bottom-up manner. Assume that the list is already sorted. [Hint: your function should take time $O(n)$.]

   (c) Write a predicate cons_treep[s] that determines whether or not an arbitrary s-expression s is a cons_tree. For example,

```
cons_treep['(nil . 3)] = cons_treep['(3 . nil)] = nil.
```

   The first one is nil because all occurrences of nil must be at the end, while the second is nil because the true cons_tree for a set consisting of just one atom is the atom itself.

   (d) Write a function bin[n] that returns the list of 1's and 0's that correspond to the binary representation of an integer n, for example,

```
bin[6] = (1 1 0).
```

   (e) Write a function kthleast[x, k] which takes a cons_tree x and an integer k as arguments and returns the k^{th} least element of x (nil if no such element exists). For example,

```
kthleast['((1 . 2) . (5 . nil)), 3] = 5.
```

   [Hint: look at the binary representation of k-1.]