

CSCI 301, Lab # 2

Fall 2018

Goal: The purpose of this lab is write some code using lists. All procedures should be written using `car` and `cdr` and recursion to traverse lists. Do not use `map`, write out the recursive function needed to get the job done. You may use `append` (see below).

Due: Your program, named `lab02.rkt`, must be submitted to Canvas before midnight, Monday, Oct 15.

Unit tests: At a minimum, your program must pass the unit tests found in the file `lab02-test.rkt`. Place this file in the same folder as your program, and run it; there should be no output.

Finding subsets: Suppose we want to procedurally find all the subsets of a given set, $A = \{1, 2, 3\}$, the power set, $\mathcal{P}(A)$.

One way to think of this is to break the subsets into two groups by picking a single element of A , for example, 1, and dividing $\mathcal{P}(A)$ into subsets that have 1 in them, and subsets that don't.

Call the ones that don't have 1 in them A_0 and the ones that do, A_1 . In our example, we have:

$$\begin{aligned} A_0 &= \{\emptyset, \{2\}, \{3\}, \{2, 3\}\} \\ A_1 &= \{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\} \end{aligned}$$

Note that the sets in A_1 are just the sets in A_0 with a 1 added to them.

The power set is just the union of these two:

$$\mathcal{P}(A) = A_0 \cup A_1$$

Note that we now have a recursive definition of the power set:

$$\mathcal{P}(A) = \begin{cases} \{\emptyset\} & \text{if } A = \emptyset \\ A_0 \cup A_1 & \text{otherwise} \end{cases}$$

where A_0 are all the subsets of a set without one of the elements of A , and A_1 are all the subsets with that element. But remember, A_1 is just A_0 with the one element added back to each subset, so both sets are defined in terms of A_0 .

Program: We'll use the above ideas to write a Scheme program to create sublists of a list.

```
> (sublists '())
'()
> (sublists '(1 2))
'() (2) (1) (1 2)
> (sublists '(1 2 3))
'() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3)
> (sublists '(1 2 3 4))
'() (4) (3) (3 4) (2) (2 4) (2 3) (2 3 4) (1) (1 4) (1 3) (1 3 4) (1 2) (1 2 4) (1 2 3) (1 2 3 4)
```

This procedure should be easy, given the above insights. If the list is empty, the value is simple. If the list `ls` is not empty, then find the sublists of `(cdr ls)`. Save this list in a local variable. (This represents the set A_0 .) Call another procedure to add the `(car ls)` to each of the lists in this set. Call this procedure `distribute`. It works like this:

```
> (distribute 7 '((1 2 3) (4 5) (1 1 1)))  
'((7 1 2 3) (7 4 5) (7 1 1 1))
```

Note that the order returned by `distribute` will depend on whether you use tail-recursion. (Why?)
Now just append the two lists to get the final result.

Sorting the results: The results we get are not very satisfying as regards their order. Clearly, the second order here is better than the first:¹

```
> (sublists '(1 2 3 4))  
'(( ) (4) (3) (3 4) (2) (2 4) (2 3) (2 3 4) (1) (1 4) (1 3) (1 3 4) (1 2) (1 2 4) (1 2 3) (1 2 3 4))  
> (subsets '(1 2 3 4))  
'(( ) (1) (2) (3) (4) (1 2) (1 3) (1 4) (2 3) (2 4) (3 4) (1 2 3) (1 2 4) (1 3 4) (2 3 4) (1 2 3 4))
```

We can get this simply by sorting the results from the `sublists` procedure. Scheme has a builtin sorting function, which takes a two-place boolean operator to decide how to sort:

```
> (sort '(3 5 2 9 1) <)  
'(1 2 3 5 9)  
> (sort '(3 5 2 9 1) >)  
'(9 5 3 2 1)
```

So all you have to do is write some two-place boolean operators that, first, sorts by length of the list, and then, within lists of the same length, sorts by elements. For example, the function `element-ordered?` returns `#t` if the lists are the same, or the first differing element is smaller in the first list, and `#f` otherwise:

```
> (element-ordered? '(4 7 9) '(4 7 9))  
#t  
> (element-ordered? '(1 3 5) '(1 3 4))  
#f  
> (element-ordered? '(1 3 5 8) '(1 3 6 7))  
#t
```

And another function, `length-ordered?`, which returns `#t` if the first list is shorter, `#f` if the first list is longer, and the result of `element-ordered?` if they are the same length.

¹Note: although I call this new procedure “`subsets`,” it only sorts the lists. It does not remove duplicates, *etc.*

Putting these together gives such spectacular results as this:

```
> (subsets '(1 2 3 4 5))  
'()  
  (1)  
  (2)  
  (3)  
  (4)  
  (5)  
 (1 2)  
 (1 3)  
 (1 4)  
 (1 5)  
 (2 3)  
 (2 4)  
 (2 5)  
 (3 4)  
 (3 5)  
 (4 5)  
 (1 2 3)  
 (1 2 4)  
 (1 2 5)  
 (1 3 4)  
 (1 3 5)  
 (1 4 5)  
 (2 3 4)  
 (2 3 5)  
 (2 4 5)  
 (3 4 5)  
 (1 2 3 4)  
 (1 2 3 5)  
 (1 2 4 5)  
 (1 3 4 5)  
 (2 3 4 5)  
 (1 2 3 4 5))
```