CS 135 Winter 2018

Tutorial 6: Recursion with multiple lists
Goals of this tutorial

You should be able to...

• process two lists in lockstep.
• process two lists at different rates.
• work with two-dimensional data represented by nested lists.
• recurse on natural numbers and lists at the same time.
Recursing in lockstep: create-al

;; An association list (AL) is one of:
;; * empty
;; * (cons (list Num Str) AL)
;; requires: the keys are unique

Write a function create-al that consumes a list of numbers and a list of strings, and produces an association list, formed by pairing up elements in the consumed lists which are at the same positions. When one of the lists is empty, pairs are no longer formed. You may assume that the consumed list of numbers contains no duplicate elements. For example:

(create-al (list 2 4 3) (list "a" "b" "c" "d"))
⇒ (list (list 2 "a") (list 4 "b") (list 3 "c"))
Review: Processing two lists at different rates

If the two lists lst1, lst2 being consumed are of different lengths, all four possibilities for their being empty/non-empty are possible:

(and (empty? lst1) (empty? lst2))
(and (empty? lst1) (cons? lst2))
(and (cons? lst1) (empty? lst2))
(and (cons? lst1) (cons? lst2))

Exactly one of these is true, but all must be tested in the template.
Recursing at different rates: union

Consider the following data definition for a NumSet:

;; A NumSet is a (listof Num)
;; requires: the numbers are strictly increasing

Write a function union that consumes two NumSets and produces a single NumSet containing all the numbers that are in either of the consumed sets.

Since the numbers are strictly increasing in a NumSet, all the numbers in an individual NumSet are also unique. You must make a single pass over the two lists, and not use any helper functions. The only list functions you may use are cons, cons?, empty, empty? and rest. For example:

(union (list 1 3 5 7) (list 2 3 6)) ⇒ (list 1 2 3 5 6 7)
Recursing on 2-d lists and natural numbers

Write a function, `add-to-first-n`, that consumes a number `to-add`, a natural number `n`, and a `(listof (listof Num))`, and adds the value of `to-add` to the first `n` elements in each sublist of the consumed `(listof (listof Num))`. For example:

```
(add-to-first-n 8 3 (list (list 8 6 3 4 9) (list 8 6) (list 0 5 2) (list 7 3 1 4) empty))
⇒ (list (list 16 14 11 4 9) (list 16 14) (list 8 13 10) (list 15 11 9 4) empty)
```
Review: Association Lists (Optional)

;; An association list (AL) is one of:

;; * empty

;; * (cons (list Num Str) AL)

;; my-al-fn: AL → Any
(define (my-al-fn alst)
  (cond
    [(empty? alst) . . .]
    [else (. . . (first (first alst)). . . ; first key
     . . . (second (first alst)). . . ; first value
     . . . (my-al-fn (rest alst)))]))
Group Problem (Optional): remove-al

Write a function, remove-al, that consumes an association list alst and a number k. It produces the same association list but with the key k removed. If k is not in the association list, there will be no changes to alst.

Note that keys in an association list are unique.
Group Problem (Optional): is-prefix-of?

Write a function, is-prefix-of?, that consumes a list of Sym called pattern, and another list of Sym called word. It produces true if the symbols in pattern occur in the same order at the start of word, and false otherwise.