Goals of this tutorial

You should be able to...

• write mutually recursive functions based on the relevant templates.
• write functions that process nested lists based on the relevant templates

Group problem - BoolExp

In lecture, we saw that an arithmetic expression could be represented by the structure AINode. It could also be represented by nested lists. In this group problem, we will use the following data definition for a boolean expression:

```scheme
;; A boolean expression (BoolExp) is one of:
;; * (anyof 'true 'false)
;; * (cons (anyof 'and 'or) (listof BoolExp))
```

Some examples include:

`
'true
'(or true (and true true false))
`
Group problem - mutually recursive templates

;; A boolean expression (BoolExp) is one of:
;; * (anyof 'true 'false)
;; * (cons (anyof 'and 'or) (listof BoolExp))

Write a template function for each of BoolExp and (listof BoolExp).

Group problem - count-subexpressions

;; A boolean expression (BoolExp) is one of:
;; * (anyof 'true 'false)
;; * (cons (anyof 'and 'or) (listof BoolExp))

Using your templates, write a function count-subexpressions that consumes a BoolExp and produces the number of subexpressions in that boolean expression:

(count-subexpressions '(or true (and true true false))) ⇒ 6
(count-subexpressions 'true) ⇒ 1
(count-subexpressions '(or (or true (and false false) (and true true false)) true (and false false)))
⇒ 14

Group problem - is-bexp?

;; A boolean expression (BoolExp) is one of:
;; * (anyof 'true 'false)
;; * (cons (anyof 'and 'or) (listof BoolExp))

Write a function is-bexp? that consumes an input and produces true if it is a BoolExp:
Group problem - is-bexp?

(is-bexp? ()) ⇒ false
(is-bexp? (or true (and true true false))) ⇒ true
(is-bexp? true) ⇒ false
(is-bexp? (or)) ⇒ true

Group problem - mutually recursive templates

For this example, we are going to use general trees to store employee information for an organization. Consider the following definitions:

(define-struct orgnode (name employees))

;; An OrgChart is a (make-orgnode Str (listof OrgChart))
;; requires: the names are unique

(define cs135-staff (make-orgnode "Karen"
(list (make-orgnode "Byron"
(list (make-orgnode "Anna"
(list (make-orgnode "Ben" ()
(make-orgnode "Viv" ()
(make-orgnode "Charlie" ()
(make-orgnode "Paul"
(list (make-orgnode "Jason"
(make-orgnode "Bill"
(make-orgnode "Jimmy"))))))))))))

Group problem - mutually recursive templates

(define cs135-staff (make-orgnode "Karen"
(list (make-orgnode "Byron"
(list (make-orgnode "Anna"
(list (make-orgnode "Ben" ()
(make-orgnode "Viv" ()
(make-orgnode "Charlie" ()
(make-orgnode "Paul"
(list (make-orgnode "Jason"
(make-orgnode "Bill"
(make-orgnode "Jimmy"))))))))))))
Group problem - mutually recursive templates

Here is what cs135-staff looks like:

```
   Karen
   ├── Byron
   |    ├── Anne
   |    └── Ben
   ├── Charlie
   |    └── Jason
   └── Paul
        └── Yao
    └── Jesse
         └── Bill
             └── Jimmy
```

Group problem - mutually recursive templates

Write a template function for each of OrgChart and (listof OrgChart).

Group Problem - list-employees-at

Using your templates, write a function list-employees-at that consumes a positive integer, n, and an OrgChart. It produces a list of all the employees at level n in the OrgChart, where the root node is at level 1. The list may be in any order.

```
(list-employees-at 1 cs135-staff) ⇒ ('"Karen"

(list-employees-at 3 cs135-staff) ⇒ ('"Anne" "Jason" "Jesse"

(list-employees-at 5 cs135-staff) ⇒ ()
```
Group Problem - build-chain

Write a function `build-chain` that consumes a list of strings representing a chain hierarchy from the highest command to lowest command chain, and produces the OrgChart that represents this hierarchy.

```lisp
(build-chain '("Karen" "Josh")) =>

Karen
  |
  Josh

(build-chain '("Jesse" "Jimmy" "Bill")) =>

Jesse
  |
  Jimmy
    |
    Bill
```
Group Problem - add-chain-of-command
Write a function add-chain-of-command that consumes a non-empty list of strings, chain, and an OrgChart. chain represents a chain of command of employees in the OrgChart, starting from the topmost employee in the OrgChart. The function should produce the same OrgChart but with any new employees in chain added to it, in that order. Note that adding the chain may not replace any employees already in the OrgChart and it may not create any duplicate employees.

Hint: An important step is when you reach a point where no employees match the top of the chain that you have at the moment and you wish to append the head of the chain to the list of the employees; you will use build-chain for that step.

Group Problem - add-chain-of-command
(add-chain-of-command ("Karen" "Josh") cs135-staff) ⇒

Group Problem - add-chain-of-command
(add-chain-of-command ("Karen" "Byron" "Anne" "Yao") cs135-staff) ⇒

Group Problem - add-chain-of-command
(add-chain-of-command ("Karen" "Byron" "Anne" "Yao") cs135-staff) ⇒
Group Problem - add-chain-of-command

(add-chain-of-command ("Karen" "Paul") cs135-staff) =>

(Karen
  Byron Charlie Paul
  Anne Jason Jesse
  Ben Yao Bill Jimmy)

(Optional) Group problem - Nested Lists
Recall the data definition for Nest-List-Num and its template:

;; A nested list of numbers (Nest-List-Num) is one of:
;; * empty
;; * (cons Num Nest-List-Num)
;; * (cons Nest-List-Num Nest-List Num)

;; nest-lst-template: Nest-List-Num -> Any
(define (nest-lst-template lst)
  (cond [(empty? lst) . . . ]
        [(number? (first lst))
         . . . (first lst) . . . (nest-lst-template (rest lst)) . . . ]
        [else
         . . . (nest-lst-template (first lst)) . . .
         (nest-lst-template (rest lst)) . . . ]))

(Optional) Group problem - Nested Lists
Using the previous template, write a function sum-nest that consumes a nested list of numbers and produces the sum of all the numbers in it.

(sum-nest '()) => 0

(sum-nest '((3) 2) (4 (3 6))) => 18

(sum-nest '((4 2) 3 (4 1.5 6))) => 20.5