CS 135 Fall 2019

Tutorial 8: Mutual Recursion
Goals of this tutorial
You should be able to...

- Write templates for mutually recursive data types.
- Using mutually recursive templates to write functions.
Problem 1: Boolean Expressions

Boolean expressions can be represented by nested lists.
Consider the following data definition for a boolean expression:

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

;; Examples:
‘true

‘(or true (and true true false))

Write template functions for BoolExp and (listof BoolExp).
Problem 2: Count Sub-Expressions

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

```lisp
(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
```
Problem 2: Design Recipes

;; (count-subexpressions bexp) produces
;; the number of expressions in bexp
;; count-subexpressions: BoolExp → Nat

(define (count-subexpressions bexp) . . . )

;; (count-subexpressions-list lobexp) produces the
;; number of expressions in all bexps in lobexp
;; count-subexpressions-list: (listof BoolExp) → Nat

(define (count-subexpressions-list) . . . )
Problem 3: is-bexp?

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

;; Examples:
(is-bexp? '()) ⇒ false
(is-bexp? (or true (and true true false))) ⇒ true
(is-bexp? true) ⇒ false
(is-bexp? '(or)) ⇒ true
(is-bexp? '(or (true (and false false)) (and true false))) ⇒ false
(is-bexp? 111) ⇒ false
Problem 3: Example

\[ \text{or} \]
\[ \text{true} \] \[ \text{and} \] \[ \text{true} \] \[ \text{true} \] \[ \text{false} \]
Problem 3: is-bexp? Design Recipes

;; (is-bexp? bexp) Validates a bexp
;; is-bexp?: Any → Bool

(define (is-bexp? bexp) . . .)

;; (is-bexp-lst? lobexp) Validates a lobexp
;; is-bexp-lst?: Any → Bool

(define (is-bexp-lst? lobexp) . . .)
Problem 4: Tries

A Trie is a general tree where each node stores a single character and the character \#\$ denotes the end of a word. Each path from the root of the Trie to a leaf node would correspond to a word stored in the Trie. Consider the following data definition and example:

```
(define-struct node (letter children))
;; A Trie is one of:
;; * (make-node #\$ empty)
;; * (make-node Char (listof Trie))
;; requires: a ≤ letter ≤ z, children is non-empty
;; parents cannot have duplicate siblings
```
Problem 4: Tries Example

```
(define d-trie (make-node #\d
                              (list (make-node #\o (list (make-node #\g
                                                      (list (make-node #\$ empty)))))
                              (make-node #\v (list (make-node #\e
                                                      (list (make-node #\$ empty)))))
                              (make-node #\$ empty)))))

(make-node #\a (list (make-node #\y
                          (list (make-node #\$ empty)))))))))
```

d-trie stores the words **dog**, **do**, **dove**, and **day**.
Problem 4: Example
Problem 5: contains-word?

Using the provided templates write a function `contains-word?` that consumes a Trie, and a string and determines if the string corresponds to a word stored in the Trie.

(contains-word? d-trie "dove\$") ⇒ true
(contains-word? d-trie "daze\$") ⇒ false
Problem 5: Design Recipes

;; (contains-word/char? trie loc) Determine whether the given
;; trie contains all characters in the list of characters, loc.
;; contains-word/char?: Trie (listof Char) → Bool

(define (contains-word/char? trie loc) . . . )

;; (contains-word/list? lotrie loc) Determine whether one of the tries in
;; list of tries, lotrie contains all characters in list of characters, loc.
;; contains-word/list?: (listof Trie) (listof Char) → Bool

(define (contains-word/list? lotrie loc) . . . )
Problem 6: Count Words

Write a function `count-words` that consumes a Trie and determines the total numbers of words stored in the Trie.

We can consider a single node with a `#\$` letter to represent a Trie containing only the empty string "$".

\[
\begin{align*}
\text{(count-words d-trie)} & \Rightarrow 4 \\
\text{(count-words (make-node #\$ empty))} & \Rightarrow 1 
\end{align*}
\]
Problem 6: Design Recipe

;; (count-words trie) Produces the number of words in given trie.
;; count-words: Trie → Nat

(define (count-words trie) . . . )

;; (count-words/list lotrie) Produces the number
;; of words in a given list of tries, lotrie.
;; count-words/list: (listof Trie) → Nat

(define (count-words/list lotrie) . . . )