CS 135 Winter 2019

Tutorial 8: Mutual recursion
Goals of this tutorial

You should be able to...

- write templates for mutually recursive functions.
- write mutually recursive functions 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:

;;; 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

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

(is-bexp? 111) ⇒ false
Group Problem - Tries

A Trie is a general tree where each node in the Trie stores a single character (for the purposes of this question, we will only be working with lower-case alphabetic characters, and the character #\$ that denotes the end of a word), and 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 definitions and example:

```
(define-struct node (letter children))
;; A Trie is one of:
;; * (make-node #\$ empty); or
;; * (make-node Char (listof Trie))
;; requires: #\a <= letter <= #\z and children is non-empty
;; each node’s letter is distinct from the letters of its siblings
```
Group Problem - Tries

\(\text{define } \text{d-trie} = \left(\begin{array}{c}
\text{make-node } \# \text{d} \left(\begin{array}{c}
\text{list } \left(\begin{array}{c}
\text{make-node } \# \text{o} \\
\text{list } \left(\begin{array}{c}
\text{make-node } \# \text{g} \\
\text{list } \left(\begin{array}{c}
\text{make-node } \# \text{$ empty}) \\
\text{make-node } \# \text{$ empty}) \\
\text{make-node } \# \text{v} \\
\text{list } \left(\begin{array}{c}
\text{make-node } \# \text{e} \\
\text{list } \left(\begin{array}{c}
\text{make-node } \# \text{$ empty}) \\
\text{make-node } \# \text{$ empty}) \\
\text{make-node } \# \text{a} \\
\text{list } \left(\begin{array}{c}
\text{make-node } \# \text{y} \\
\text{list } \left(\begin{array}{c}
\text{make-node } \# \text{$ empty}) \\
\text{make-node } \# \text{$ empty})
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)\)

In this example, \text{d-trie} stores the words dog\$, do\$, dove\$, and day\$. Note that we use a ’$’ as a special marker to conveniently mark the end of a word to make searching and storing words easier.
Group Problem - Mutually Recursive Templates

Write template functions for a Trie, and for a (listof Trie). This should help you plan out the structure of your code for the following questions.
Group Problem - 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 "$".

\[
\text{(count-words d-trie) } \Rightarrow 4
\]

\[
\text{(count-words (make-node #\$ empty)) } \Rightarrow 1
\]
Group Problem - contains-word?

Write a function contains-word? that consumes a string, and a Trie and determines if the string corresponds to a word stored in the Trie. For example,

(contains-word? "dove\$" d-trie) ⇒ true

(contains-word? "daze\$" d-trie) ⇒ false