Nested lists
Structuring data using mutual recursion

Mutual recursion arises when complex relationships among data result in cross references between data definitions.

The number of data definitions can be greater than two. Structures and lists may also be used.

In each case:
• create templates from the data definitions and
• create one function for each template.
Other uses of general trees

We can generalize from allowing only two arithmetic operations and numbers to allowing arbitrary functions and variables.

In effect, we have the beginnings of a Racket interpreter.

But beyond this, the type of processing we have done on arithmetic expressions can be applied to tagged hierarchical data, of which a Racket expression is just one example.

Organized text and Web pages provide other examples.
'(chapter
  (section
    (paragraph "This is the first sentence."
     "This is the second sentence."))
    (paragraph "We can continue in this manner."))
  (section ...)
...
...)
(webpage
  (title "CS 135: Designing Functional Programs")
  (paragraph "For a course description, "
    (link "click here" "desc.html"")
    " . Enjoy the course!"
  )
  (horizontal-line)
  (paragraph "(Last modified yesterday.)")
)
Nested lists

We have discussed flat lists (no nesting):
'(a 1 "hello" x)
and lists of lists (one level of nesting):
'(((1 "a") (2 "b"))
We now consider nested lists (arbitrary nesting):
'(((1 (2 3)) 4 (5 (6 7 8) 9)))
It is often useful to visualize a nested list as a tree, in which the leaves correspond to the elements of the list, and the internal nodes indicate the nesting:

'((1 (2 3)) 4 (5 (6 7 8) 9 ()))
Examples of nested lists:

```
empty
'(4 2)
'((4 2) 3 (4 1 6))
'((3) 2 () (4 (3 6)))
```

Each non-empty tree is a list of subtrees.

The first subtree in the list is either

- a single leaf (not a list) or
- a subtree rooted at an internal node (a list).
Data definition for nested lists

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

This can be generalized to generic types: (Nest-List-X)
Template for nested lists

The template follows from the data definition.

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)))]))
The function count-items

;;; count-items: Nest-List-Num -> Nat
(define (count-items nln)
  (cond
   [(empty? nln) 0]
   [(number? (first nln))
    (+ 1
     (count-items (rest nln)))]
   [else
    (+ (count-items (first nln))
      (count-items (rest nln)))]))
CQ 7:

(define (count-items nln)
  (cond
   [(empty? nln) 0]
   [(number? (first nln))
    (+ 1 (count-items (rest nln)))]
   [else
    (+ (count-items (first nln))
       (count-items (rest nln)))]))

What is the result of applying
(count-items '((7 5) 3 () (2 (6 4)) (1)))?

A. 4
B. 5
C. 6
D. 7
E. 8
CQ 7:

\[
(\text{define} \ (\text{count-items} \ \text{nln})
\]
\[
(\text{cond}
\]
\[
[\text{(empty? nln)} \ 0]
\]
\[
[\text{(number? (first nln))}
\]
\[
(+ 1 \ (\text{count-items} \ (\text{rest nln})))]
\]
\[
[\text{else}
\]
\[
(+ \ (\text{count-items} \ (\text{first nln}))
\]
\[
(\text{count-items} \ (\text{rest nln})))]]
\]

What is the result of applying \( (\text{count-items} \ '((7 5) 3 () (2 (6 4)) (1))) \)?

A. 4  
B. 5  
C. 6  
D. 7  
E. 8
Condensed trace of count-items

(count-items '((10 20) 30))
=> (+ (count-items '((10 20)) (count-items '(30)))
=> (+ (+ 1 (count-items '(20))) (count-items '(30)))
=> (+ (+ 1 (+ 1 (count-items '()))) (count-items '(30)))
=> (+ (+ 1 (+ 1 0)) (count-items '(30)))
=> (+ (+ 1 1) (count-items '(30)))
=> (+ 2 (count-items '(30)))
=> (+ 2 (+ 1 (count-items '())))
=> (+ 2 (+ 1 0))
=> (+ 2 1)
=> 3
Flattening a nested list

flatten produces a flat list from a nested list.
;; (flatten lst) produces a single-level list with all
;; the elements of lst.
;; flatten: Nest-List-Num -> (listof Num)
(check-expect (flatten '(1 2 3)) '(1 2 3))
(check-expect (flatten '(((1 2 3) (a b))) '(1 2 3 a b))
(define (flatten lst)
    ...))

We make use of the built-in Racket function append.
(append '(1 2) '(3 4)) => '(1 2 3 4)
;; flatten: Nest-List-Num -> (listof Num)
(define (flatten lst)
  (cond
   [(empty? lst) empty]
   [(number? (first lst))
    (cons (first lst) (flatten (rest lst)))]
   [else
    (append (flatten (first lst))
            (flatten (rest lst)))]))
Condensed trace of flatten

(flatten '((10 20) 30))
⇒ (append (flatten '(10 20)) (flatten '(30)))
⇒ (append (cons 10 (flatten '(20))) (flatten '(30)))
⇒ (append (cons 10 (cons 20 (flatten '()))) (flatten '(30)))
⇒ (append (cons 10 (cons 20 empty)) (flatten '(30)))
⇒ (append (cons 10 (cons 20 empty)) (cons 30 (flatten '())))
⇒ (append (cons 10 (cons 20 empty)) (cons 30 empty))
⇒ (cons 10 (cons 20 (cons 30 empty)))
Goals of this module

• You should be familiar with tree terminology.
• You should understand the data definitions for binary trees, binary search trees, evolutionary trees, and binary arithmetic expressions.
• You should understand how the templates are derived from those definitions, and how to use the templates to write functions that consume those types of data.
• You should understand the definition of a binary search tree and its ordering property.
• You should be able to write functions which consume binary search trees, including those sketched (but not developed fully) in lecture.

• You should be able to develop and use templates for other binary trees, not necessarily presented in lecture.

• You should understand the idea of mutual recursion for both examples given in lecture and new ones that might be introduced in lab, assignments, or exams.

• You should be able to develop templates from mutually recursive data definitions, and to write functions using the templates.