CS 135
October 10, 2019
Lists of lists as 2D data
Two-dimensional data

Another use of lists of lists is to represent a two-dimensional table. For example, here is a multiplication table:

\[(\text{mult-table } 3 \ 4)\]

\[\Rightarrow (\text{list} \ (\text{list} \ 0 \ 0 \ 0 \ 0) \n\quad (\text{list} \ 0 \ 1 \ 2 \ 3) \n\quad (\text{list} \ 0 \ 2 \ 4 \ 6))\]

The $c^{th}$ entry of the $r^{th}$ row (numbering from 0) is $r \times c$.

We can write mult-table using two applications of the “count up” idea.
Two-dimensional data

Another use of lists of lists is to represent a two-dimensional table. For example, here is a multiplication table:

\[(\text{mult-table 3 4})\]

\[
=> (\text{list (list 0 0 0 0) (list 0 1 2 3) (list 0 2 4 6)})
\]
;;; (mult-table nr nc) produces multiplication table
;;; with nr rows and nc columns
;;; mult-table: Nat Nat -> (listof (listof Nat))
(define (mult-table nr nc)
  (rows-from 0 nr nc))

;;; (rows-from r nr nc) produces multiplication table,
;;;   rows r ... (nr-1)
;;; rows-from: Nat Nat Nat -> (listof (listof Nat))
(define (rows-from r nr nc)
  (cond
    [(>= r nr) empty]
    [else
     (cons (row r nc) (rows-from (add1 r) nr nc))])))
;; (row r nc) produces rth row of multiplication table of length nc
;; row: Nat Nat -> (listof Nat)
(define (row r nc)
  (cols-from 0 r nc))

;; (cols-from c r nc) produces entries c ... (nc-1) of rth row of multiplication table
;; cols-from: Nat Nat Nat Nat -> (listof Nat)
(define (cols-from c r nc)
  (cond
   [(>= cn c) empty]
   [else (cons (* r c) (cols-from (add1 c) r nc))])))
Processing two lists simultaneously
Processing two lists simultaneously

We now look at a more complicated recursion, namely writing functions which consume two lists (or two data types, each of which has a recursive definition).

Following the textbook, we will distinguish three different cases, and look at them in order of complexity.

The simplest case is when one of the lists does not require recursive processing.
Case 1: processing just one list

As an example, consider the function my-append.

;; (my-append lst1 lst2) appends lst2 to the end of lst1
;; my-append: (listof Any) (listof Any) -> (listof Any)
;; Examples:
(check-expect (my-append empty '(1 2)) '(1 2))
(check-expect (my-append '(3 4) '(1 2 5)) '(3 4 1 2 5))

(define (my-append lst1 lst2)
  ...)
(define (my-append lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [else (cons (first lst1)
                 (my-append (rest lst1) lst2))])))

The code only does simple recursion on lst1. The parameter lst2 is “along for the ride”. append is a built-in function in Racket.
A condensed trace

(my-append
  (cons 1 (cons 2 empty)) (cons 3 (cons 4 empty)))
⇒ (cons 1 (my-append
  (cons 2 empty) (cons 3 (cons 4 empty))))
⇒ (cons 1 (cons 2 (my-append empty
  (cons 3 (cons 4 empty))))))
⇒ (cons 1 (cons 2 (cons 3 (cons 4 empty))))
Case 2: processing in lockstep

To process two lists \( \text{lst1} \) and \( \text{lst2} \) in lockstep, they must be the same length, and are consumed at the same rate.

\( \text{lst1} \) is either \texttt{empty} or a \texttt{cons}, and the same is true of \( \text{lst2} \) (four possibilities in total).

However, because the two lists must be the same length, \((\texttt{empty? lst1}) \) is \texttt{true} if and only if \((\texttt{empty? lst2}) \) is \texttt{true}.

This means that out of the four possibilities, two are invalid for proper data.

The template is thus simpler than in the general case.
Lockstep template

\[
\text{(define (lockstep-template lst1 lst2)}
\begin{aligned}
\text{(cond)} \\
&[(\text{empty? lst1)} ...] \\
\text{[else (}} ...
&\text{...(first lst1)} \\
&\text{...(first lst2)} \\
&\text{(lockstep-template)} \\
&\text{(rest lst1)} \\
&\text{(rest lst2)))]\]
\end{aligned}
\]
Example: dot product

To take the dot product of two vectors, we multiply entries in corresponding positions (first with first, second with second, and so on) and sum the results.

Example: the dot product of \((1\ 2\ 3)\) and \((4\ 5\ 6)\) is

\[
1 \times 4 + 2 \times 5 + 3 \times 6 = 4 + 1 + 18 = 32
\]

We can store the elements of a vector in a list, so \((1\ 2\ 3)\) becomes \\
'(1 2 3).

For convenience, we define the empty vector with no entries, represented by empty.
The function dot-product

;;; (dot-product lon1 lon2) computes the dot product of
;;; vectors lon1 and lon2

;;; dot-product: (listof Num) (listof Num) -> Num
;;; requires: lon1 and lon2 are the same length

(check-expect (dot-product empty empty) 0)
(check-expect (dot-product '(2) '(3)) 6)
(check-expect (dot-product '(2 3) '(4 5)) 23)

(define (dot-product lon1 lon2)
  (...))
The function dot-product

\[
\text{(define (dot-product lon1 lon2)} \\
\text{(cond)} \\
\text{[(empty? lon1) 0]} \\
\text{[else (+ \}
\text{(* (first lon1) (first lon2)) \}
\text{(dot-product (rest lon1) (rest lon2)))]})]
\]
A condensed trace

\[
\text{(dot-product (cons 2 (cons 3 empty))}
\text{ (cons 4 (cons 5 empty)))}
\]
\[
\Rightarrow (+ 8 \text{(dot-product (cons 3 empty) (cons 5 empty))})
\]
\[
\Rightarrow (+ 8 (+ 15 \text{(dot-product empty empty)})))
\]
\[
\Rightarrow (+ 8 (+ 15 0))
\]
\[
\Rightarrow 23
\]
Case 3: processing 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:

\[(\text{and} \ (\text{empty?} \ \text{lst1}) \ (\text{empty?} \ \text{lst2}))\]
\[(\text{and} \ (\text{empty?} \ \text{lst1}) \ (\text{cons?} \ \text{lst2}))\]
\[(\text{and} \ (\text{cons?} \ \text{lst1}) \ (\text{empty?} \ \text{lst2}))\]
\[(\text{and} \ (\text{cons?} \ \text{lst1}) \ (\text{cons?} \ \text{lst2}))\]

Exactly one of these is true, but all must be tested in the template.
Base cases

The template so far

(define (twolist-template lst1 lst2)
  (cond
    [(and (empty? lst1) (empty? lst2)) ...]
    [(and (empty? lst1) (cons? lst2)) ...]
    [(and (cons? lst1) (empty? lst2)) ...]
    [(and (cons? lst1) (cons? lst2)) ...]))

The first case is a base case; the second and third may or may not be.
Template

The template so far

```
(define (twolist-template lst1 lst2)
  (cond
    [(and (empty? lst1) (empty? lst2)) ...]
    [(and (empty? lst1) (cons? lst2))
      (... (first lst2) ... (rest lst2) ...)]
    [(and (cons? lst1) (empty? lst2)) ...]
      (... (first lst1) ... (rest lst1) ...)]
    [(and (cons? lst1) (cons? lst2)) ???]])
```

The second and third cases may or may not require recursion. The fourth case definitely does, but its form is unclear.
Further refinements

There are many different possible natural recursions for the last `cond` answer:?

... (first lst2)
(twolist-template lst1 (rest lst2))

... (first lst1)
(twolist-template (rest lst1) lst2)

... (first lst1)
... (first lst2)
(twolist-template (rest lst1) (rest lst2))

We need to reason further in specific cases to determine which is appropriate.
Example: merging two sorted lists

We wish to design a function merge that consumes two lists.

Each list is sorted in ascending order (no duplicate values). merge will produce one such list containing all elements.

As an example:

\[
\text{merge (list 1 8 10) (list 2 4 6 12)}
\]

=> (list 1 2 4 6 8 10 12)

We need more examples to be confident of how to proceed.
Example: merging two sorted lists

(merge empty empty) => empty

(merge empty (list 2)) => (list 2)

(merge (list 1 3) empty) => (list 1 3)

(merge (list 1 4) (list 2)) => (list 1 2 4)

(merge (list 3 4) (list 2)) => (list 2 3 4)
Reasoning about merge

If lon1 and lon2 are both non-empty, what is the first element of the merged list?

It is the smaller of \((\text{first} \ lon1)\) and \((\text{first} \ lon2)\).

If \((\text{first} \ lon1)\) is smaller, then the rest of the answer is the result of merging \((\text{rest} \ lon1)\) and lon2.

If \((\text{first} \ lon2)\) is smaller, then the rest of the answer is the result of merging lon1 and \((\text{rest} \ lon2)\).
(define (merge lon1 lon2)
  (cond
   [((and (empty? lon1) (empty? lon2)) empty]
   [(and (empty? lon1) (cons? lon2)) lon2]
   [(and (cons? lon1) (empty? lon2)) lon1]
   [(and (cons? lon1) (cons? lon2))
      (cond
       [(< (first lon1) (first lon2))
          (cons
           (first lon1)
           (merge (rest lon1) lon2))]
       [else
          (cons
           (first lon2)
           (merge lon1 (rest lon2)))]))])
A condensed trace (with lists)

\[(\text{merge } (\text{list } 3 \ 4) (\text{list } 2 \ 5 \ 6))\]
\[\Rightarrow (\text{cons } 2 (\text{merge } (\text{list } 3 \ 4) (\text{list } 5 \ 6))))\]
\[\Rightarrow (\text{cons } 2 (\text{cons } 3 (\text{merge } (\text{list } 4) (\text{list } 5 \ 6))))\]
\[\Rightarrow (\text{cons } 2 (\text{cons } 3 (\text{cons } 4 (\text{merge } \text{empty} (\text{list } 5 \ 6))))))\]
\[\Rightarrow (\text{cons } 2 (\text{cons } 3 (\text{cons } 4 (\text{cons } 5 (\text{cons } 6 \text{ empty}))))))\]
Consuming a list and a number
Consuming a list and a number

We defined recursion on natural numbers by showing how to view a natural number in a list-like fashion.

We can extend our idea for computing on two lists to computing on a list and a number, or on two numbers.

A predicate “Does \textit{elem} appear at least \textit{n} times in this list?”

Example: “Does 2 appear at least 3 times in the list \texttt{(list 4 2 2 3 2 4)}?” produces \texttt{true}. 
The function at-least?

;; (at-least? n elem lst) determines if elem appears at least n times in lst.
;; at-least?: Nat Any (listof Any) -> Bool
(check-expect (at-least? 0 'red '(1 2 3)) true)
(check-expect (at-least? 3 "hi" empty) false)
(check-expect
  (at-least? 2 'red '(red blue red green)) true)
(check-expect
  (at-least? 3 'red '(red blue red green)) false)
(check-expect (at-least? 1 7 '(5 4 0 5 3)) false)

(define (at-least? n elem lst)
  (...))
Developing the code

The recursion will involve the parameters n and lst, once again giving four possibilities:

```lisp
(define (at-least? n elem lst)
  (cond
    [(and (zero? n) (empty? lst)) ...]
    [(and (zero? n) (cons? lst)) ...]
    [(and (> n 0) (empty? lst)) ...]
    [(and (> n 0) (cons? lst)) ...]])
```

Once again, exactly one of these four possibilities is true.
Developing the code

(define (at-least? n elem lst)
  (cond
    [(and (zero? n) (empty? lst)) ...]
    [(and (zero? n) (cons? lst)) ...]
    [(and (> n 0) (empty? lst)) ...]
    [(and (> n 0) (cons? lst)) ...])))

Found elem n times,
1st completely processed
Developing the code

\[(\text{define} \ (\text{at-least?} \ n \ \text{elem} \ \text{lst}) \ (\text{cond}) \ \text{...})\]

\[(\text{cond}) \ \text{...} \]

- \((\text{and} \ (\text{zero?} \ n) \ (\text{empty?} \ \text{lst}) \text{...})\)
- \((\text{and} \ (\text{zero?} \ n) \ (\text{cons?} \ \text{lst}) \text{...})\)
- \((\text{and} \ (> \ n \ 0) \ (\text{empty?} \ \text{lst}) \text{...})\)
- \((\text{and} \ (> \ n \ 0) \ (\text{cons?} \ \text{lst}) \text{...})\))

Found elem n times,
1st not completely processed
Developing the code

```
(define (at-least? n elem lst)
  (cond
    [(and (zero? n) (empty? lst)) ...]
    [(and (zero? n) (cons? lst)) ...]
    [(and (> n 0) (empty? lst)) ...]
    [(and (> n 0) (cons? lst)) ...]]))
```

Found elem less than n times, lst completely processed
Developing the code

(define (at-least? n elem lst)
  (cond
    [(and (zero? n) (empty? lst)) ...]
    [(and (zero? n) (cons? lst)) ...]
    [(and (> n 0) (empty? lst)) ...]
    [(and (> n 0) (cons? lst)) ...]))

Found elem less than n times, lst not completely processed
Developing the code

\[
\text{(define (at-least? n elem lst)}
\text{(cond}
\text{[(and (zero? n) (empty? lst)) ...]}
\text{[(and (zero? n) (cons? lst)) ...]}
\text{[(and (> n 0) (empty? lst)) ...]}
\text{[(and (> n 0) (cons? lst)) ...])})
\]

In which cases can we produce the answer without further processing?

In which cases do we need further recursive processing to discover the answer?

Which of the natural recursions should be used?
Improving at-least?

In working out the details for each case, it becomes apparent that some of them can be combined.

If \( n \) is zero, it does not matter whether \( \text{lst} \) is empty or not. Logically, every element always appears at least 0 times.

This leads to some rearrangement of the template, and eventually to the code that appears on the next slide.
Improved at-least?

\[
\text{(define (at-least? n elem lst)}
\begin{align*}
\text{cond} & \quad \text{[(zero? n) true]} \\
& \quad \text{[(empty? lst) false]} \\
& \quad \text{[(equal? (first lst) elem); \ text{lst is non-empty, n >= 1}]} \\
& \quad \quad \text{(at-least? (sub1 n) elem (rest lst))]}} \\
& \quad \text{[else (at-least? n elem (rest lst))]}))
\end{align*}
\]
Two condensed traces

(at-least? 3 'green '(red green blue))
=> (at-least? 3 'green '(green blue))
=> (at-least? 2 'green '(blue))
=> (at-least? 2 'green empty)
=> false

(at-least? 1 8 '(4 8 15 1 6 23 42))
=> (at-least? 1 8 '(8 15 1 6 23 42))
=> (at-least? 0 8 '(15 1 6 23 42))
=> true
Testing list equality

;; (list=? lst1 lst2) determines if lst1 and lst2 are equal
;; list=?: (listof Num) (listof Num) -> Bool
(define (list=? Lst1 lst2)
  (cond
   [(and (empty? lst1) (empty? lst2)) ...]
   [(and (empty? lst1) (cons? lst2)) (...
      ...((first lst2)
      ...((rest lst2)))]
   [(and (cons? lst1) (empty? lst2)) (...
      ...((first lst1)
      ...((rest lst1)))]
   [(and (cons? lst1) (cons? lst2)) ???]])
Reasoning about list equality

• Two empty lists are equal.
• If one list is empty and the other is not, they are not equal.
• If both are non-empty, then their first elements must be equal, and their rests must be equal.

The natural recursion in this case is \( \text{list=}? \ (\text{rest } \text{lst1}) \ (\text{rest lst2}) \)
(define (list =? lst1 lst2)
  (cond
    ([and (empty? lst1) (empty? lst2)] true]
    ([and (empty? lst1) (cons? lst2)] false]
    ([and (cons? lst1) (empty? lst2)] false]
    ([and (cons? lst1) (cons? lst2)]
      (and
       (= (first lst1) (first lst2))
       (list=? (rest lst1) (rest lst2)))]))

Some further simplifications are possible.
Built-in list equality

As you know, Racket provides the predicate equal? which tests structural equivalence. It can compare two atomic values, two structures, or two lists. Each of the non-atomic objects being compared can contain other lists or structures.

At this point, you can see how you might write equal? if it were not already built in. It would involve testing the type of data supplied, and doing the appropriate comparison, recursively if necessary.
Goals of this module

• You should understand the principle of insertion sort, and how the functions involved can be created using the design recipe.

• You should be able to use list abbreviations and quote notation for lists where appropriate.

• You should be able to construct and work with lists that contain lists.

• You should understand the three approaches to designing functions that consume two lists (or a list and a number, or two numbers) and know which one is suitable in a given situation.