Module 7: Processing two lists or numbers

Readings: HtDP, section 17.

Processing two lists simultaneously

We now move to using recursion in a complicated situation, namely writing functions that consume two lists (or two data types, each of which has a recursive definition).

Following Section 17 of the textbook, we will distinguish three different cases, and look at them in order of complexity.

One goal of this module is to learn how to choose among cases.

Case 1: a list going along for the ride

my-append consumes two lists list1 and list2, and produces the list with empty in list1 replaced by list2.

The function append is a built-in function in Racket that consumes two or more lists. Both (append (list 3) my-list) and (cons 3 my-list) produce the same list. Our style preference is to use cons rather than append when the first list has length one.

Do we need to process the individual elements of list1?

What about the individual elements of list2?

```
(define (my-list-fun alist)
  (cond
    [(empty? alist) ...]
    [else ...(first alist) ...
            ...(my-list-fun (rest alist)) ...]))

(define (my-alongforride-fun list1 list2)
  (cond
    [(empty? list1) ...]
    [else ...(first list1) ...
            ...(my-alongforride-fun (rest list1) list2) ...]))
```

The function my-append

;; my-append: (listof Any) (listof Any) → (listof Any)

;; Examples:

(check-expect (my-append empty (list 1 2)) (list 1 2))

(check-expect (my-append (cons 1 empty) (cons 2 empty))

(cons 1 (cons 2 empty)))

(define (my-append list1 list2)

  (cond [(empty? list1) list2]

        [else (cons (first list1) (my-append (rest list1) list2))]))

Condensed trace of my-append

(my-append (cons 1 (cons 2 empty))

                         (cons 3 (cons 4 empty)))

$\Rightarrow$ (cons 1 (my-append (cons 2 empty)

                          (cons 3 (cons 4 empty))))

$\Rightarrow$ (cons 1 (cons 2 (my-append empty

                              (cons 3 (cons 4 empty)))))

$\Rightarrow$ (cons 1 (cons 2 (cons 3 (cons 4 empty))))

Case 2: processing in lockstep

total-value determines the total value of items sold, given prices of items and numbers of each item.

Example: the total of prices $(1\ 2\ 3)$ and numbers $(4\ 5\ 6)$ is $1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32$.

We can store the items (and numbers) in lists, so $(1\ 2\ 3)$ becomes (list 1 2 3).

Key observation: the lists have the same length and items in a given position correspond to each other.

```
;; (total-value pricelist numlist) produces the total value of items
;;    with prices in pricelist and numbers in numlist.
;; total-value: (listof Num) (listof Nat) → Num
;; requires: Values in pricelist >= 0
;;              pricelist and numlist have the same length
;; Examples:
(check-expect (total-value empty empty) 0)
(check-expect (total-value (list 2) (list 3)) 6)
(check-expect (total-value (list 2 3) (list 4 5)) 23)
```

Developing a template

pricelist is either empty or a cons, and the same is true of numlist, yielding four options of empty/non-empty lists.

However, because the two lists must be the same length, (empty? pricelist) is true if and only if (empty? numlist) is true.

Out of the four possibilities, two are invalid for proper data.

The template is thus simpler than in the general case.

```
(define (total-value pricelist numlist)
  (cond
    [(empty? pricelist) ... ]
    [else
        ... (first pricelist) ...  (first numlist) ...
        ... (total-value (rest pricelist) (rest numlist)) ... ]))
```

Extracting the lockstep template

(define (my-lockstep-fun list1 list2)

  (cond

    [(empty? list1) ...]

    [else

        ... (first list1) ... (first list2) ...

        ... (my-lockstep-fun (rest list1) (rest list2)) ...]))

Completing total-value

```
(define (total-value pricelist numlist)
  (cond
    [(empty? pricelist) 0 ]
    [else
      (+
        (* (first pricelist) (first numlist))
        (total-value (rest pricelist) (rest numlist)))]))
```

Condensed trace of total-value

(total-value (cons 2 (cons 3 empty))

                (cons 4 (cons 5 empty)))

$\Rightarrow$ (+ ($*$ 2 4) (total-value (cons 3 empty) (cons 5 empty)))

$\Rightarrow$ (+ 8 (total-value (cons 3 empty) (cons 5 empty)))

$\Rightarrow$ (+ 8 (+ ($*$ 3 5) (total-value empty empty)))

$\Rightarrow$ (+ 8 (+ 15 (total-value empty empty)))

$\Rightarrow$ (+ 8 (+ 15 0))

$\Rightarrow$ (+ 8 15)

$\Rightarrow$ 23

Case 3: processing at different rates

If the lists being consumed, list1 and list2, are of different lengths, all four possibilities for their being empty/nonempty are possible:

(and (empty? list1) (empty? list2))

(and (empty? list1) (cons? list2))

(and (cons? list1) (empty? list2))

(and (cons? list1) (cons? list2))

Exactly one of these is true for a given pair of lists, but all possibilities must be included in the template.

The template so far:

```
(define (my-two-list-fn list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) ...]
    [(and (empty? list1) (cons? list2)) ...]
    [(and (cons? list1) (empty? list2)) ...]
    [(and (cons? list1) (cons? list2)) ...]))
```

The first case is a **base case**. The second and third may or may not be.

Refining the template

(define (my-two-list-fn list1 list2)

  (cond

    [(and (empty? list1) (empty? list2)) . . .]

    [(and (empty? list1) (cons? list2)) . . . (first list2) . . . (rest list2) . . .]

    [(and (cons? list1) (empty? list2)) . . . (first list1) . . . (rest list1) . . .]

    [(and (cons? list1) (cons? list2)) ???]))

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 several different possible natural recursions for the last

cond answer ??? :

... (first list2) ... (my-two-list-fn list1 (rest list2)) ...

... (first list1) ... (my-two-list-fn (rest list1) list2) ...

... (first list1) ... (first list2) ... (my-two-list-fn (rest list1) (rest list2)) ...

We write all of these down, realizing that not all will be used, and

eliminate unnecessary ones in reasoning about any particular

function.

```
(define (my-twolist-fn list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) ...]
    [(and (empty? list1) (cons? list2))
     ... (first list2) ... (my-twolist-fn empty (rest list2)) ...]
    [(and (cons? list1) (empty? list2))
     ... (first list1) ... (my-twolist-fn (rest list1) empty)...]
    [(and (cons? list1) (cons? list2))
     ... (first list2) ... (my-twolist-fn list1 (rest list2)) ...
     ... (first list1) ... (my-twolist-fn (rest list1) list2) ...
     ... (first list1) ... (first list2)...
     ... (my-twolist-fn (rest list1) (rest list2))...]))
```

Example: merging two sorted lists

We wish to design a function merge that consumes two lists, each of distinct elements sorted in ascending order, and produces one sorted list containing all elements.

For example:

(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.

(merge empty empty) ⇒ empty

(merge empty

      (cons 2 empty)) ⇒ (cons 2 empty)

(merge (cons 1 (cons 3 empty))

      empty) ⇒ (cons 1 (cons 3 empty))

(merge (cons 1 (cons 4 empty))

      (cons 2 empty)) ⇒ (cons 1 (cons 2 (cons 4 empty)))

(merge (cons 3 (cons 4 empty))

      (cons 2 empty)) ⇒ (cons 2 (cons 3 (cons 4 empty)))

Reasoning about merge

If list1 and list2 are both nonempty, what is the first element of the merged list?

It is the smaller of (first list1) and (first list2).

If (first list1) is smaller, then the rest of the answer is the result of merging (rest list1) and list2.

If (first list2) is smaller, then the rest of the answer is the result of merging list1 and (rest list2).

```
(define (merge list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) empty]
    [(and (empty? list1) (cons? list2)) list2]
    [(and (cons? list1) (empty? list2)) list1]
    [(and (cons? list1) (cons? list2))
      (cond
        [(< (first list1) (first list2))
            (cons (first list1) (merge (rest list1) list2))]
        [ else
            (cons (first list2) (merge list1 (rest list2)))])]))
```

Condensed trace of merge

(merge (cons 3 (cons 4 empty))

       (cons 2 (cons 5 (cons 6 empty))))

$\Rightarrow$ (cons 2 (merge (cons 3 (cons 4 empty))

         (cons 5 (cons 6 empty))))

$\Rightarrow$ (cons 2 (cons 3 (merge (cons 4 empty)

            (cons 5 (cons 6 empty)))))

$\Rightarrow$ (cons 2 (cons 3 (cons 4 (merge empty

              (cons 5 (cons 6 empty))))))

$\Rightarrow$ (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty)))))

Consuming a list and a number

In Module 6, we saw how to use structural recursion on natural numbers as well as lists.

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

Example: computing the sublist starting at the nth occurrence of some symbol.

The sublist of (list 'd 'z 'z 'h 'z 'd) starting at the 3rd occurrence of 'z is (list 'z 'd).

The function nth-occur-suffix

;; (nth-occur-suffix sym n alist) produces sublist starting at nth

;;   occurrence of sym in alist.

;; nth-occur-suffix: Sym Nat (listof Sym) $\rightarrow$ (listof Sym)

;; requires: n $>=$ 1

;; Examples:

(check-expect (nth-occur-suffix 'a 1 empty) empty)

(check-expect (nth-occur-suffix 'a 1 (list 'a 'c)) (list 'a 'c))

(check-expect (nth-occur-suffix 'a 2 empty) empty)

(check-expect (nth-occur-suffix 'a 2 (list 'a 'b 'a 'c)) (list 'a 'c))

(define (nth-occur-suffix sym n alist) . . .)

Developing the template

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

(and (= n 1) (empty? alist))

(and (= n 1) (cons? alist))

(and (> n 1) (empty? alist))

(and (> n 1) (cons? alist))

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

```
(define (my-list-and-num-fun n alist)
  (cond
     [(and (= n 1) (empty? alist)) …]
     [(and (= n 1) (cons? alist))
      … (first alist) … (my-list-and-num-fun 1 (rest alist)) …]
     [(and (> n 1) (empty? alist))
      … (my-list-and-num-fun (sub1 n) empty) … ]
     [(and (> n 1) (cons? alist))
      … (first alist) … (my-list-and-num-fun n (rest alist)) …
      … (my-list-and-num-fun (sub1 n) alist) …
      … (first alist) … (my-list-and-num-fun (sub1 n) (rest alist)) …]))
```

7: Processing two lists or numbers

Reasoning about nth-occur-suffix

What should happen when:

- n=1 and alist is empty?

- n=1 and alist is not empty?

- n>1 and alist is empty?

- n>1 and alist is not empty?

```
(define (nth-occur-suffix sym n alist)
  (cond
    [(and (= n 1) (empty? alist)) empty]
    [(and (= n 1) (cons? alist))
     (cond [(equal? sym (first alist)) alist]
           [else (nth-occur-suffix sym n (rest alist))])]
    [(and (> n 1) (empty? alist)) empty]
    [(and (> n 1) (cons? alist))
     (cond [(equal? sym (first alist))
            (nth-occur-suffix sym (sub1 n) (rest alist))]
           [else (nth-occur-suffix sym n (rest alist))])]))
```

Simplified:

```
(define (nth-occur-suffix sym n alist)
  (cond [(empty? alist) empty]
        [(= n 1) ; alist is nonempty
         (cond [(equal? sym (first alist)) alist]
               [else (nth-occur-suffix sym n (rest alist))])]
        [else ; alist is nonempty, n > 1
         (cond [(equal? sym (first alist))
                (nth-occur-suffix sym (sub1 n) (rest alist))]
               [else (nth-occur-suffix sym n (rest alist))])]))
```

Condensed trace of nth-occur-suffix

(nth-occur-suffix 'z 3

            (cons 'd (cons 'z (cons 'z

                   (cons 'h (cons 'z (cons 'd empty)))))))

$\Rightarrow$ (nth-occur-suffix 'z 3

         (cons 'z (cons 'z (cons 'h (cons 'z (cons 'd empty))))))

$\Rightarrow$ (nth-occur-suffix 'z 2 (cons 'z (cons 'h (cons 'z (cons 'd empty)))))

$\Rightarrow$ (nth-occur-suffix 'z 1 (cons 'h (cons 'z (cons 'd empty))))

$\Rightarrow$ (nth-occur-suffix 'z 1 (cons 'z (cons 'd empty)))

$\Rightarrow$ (cons 'z (cons 'd empty))

Midpoints of pairs of posns

Suppose we have two lists of posns and wish to find a list of midpoints between each pair of corresponding posns.

Which template should we use?

Testing list equality

We can apply the templates we have created to the question of deciding whether or not two lists of numbers are equal.

;; list=?: (listof Num) (listof Num) $\rightarrow$ Bool

Which template is most appropriate?

7: Processing two lists or numbers

Applying the general two list template

```
(define (list=? list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) ...]
    [(and (empty? list1) (cons? list2))
     ... (first list2) ... (list=? empty (rest list2)) ...]
    [(and (cons? list1) (empty? list2))
     ... (first list1) ... (list=? (rest list1) empty) ...]
    [(and (cons? list1) (cons? list2)) ??? ]))
```

7: Processing two lists or numbers

Reasoning about list equality

Two empty lists are equal.

If one list is empty and the other is not, the lists are not equal.

If two nonempty lists are equal, then their first elements are equal, and their rests are equal.

The natural recursion in this case is

(list=? (rest list1) (rest list2))

The function list=?

```
(define (list=? list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) true]
    [(and (empty? list1) (cons? list2)) false]
    [(and (cons? list1) (empty? list2)) false]
    [(and (cons? list1) (cons? list2))
      (and (= (first list1) (first list2))
           (list=? (rest list1) (rest list2)))]))
```

Another approach to the problem

Another way of viewing the problem comes from the observation that if the lists are equal, they will have the same length.

We can then use the structure of one list in our function, checking that the structure of the other list matches.

This implies the use of the lockstep template.

Here is the result of applying the lockstep template.

```
(define (list=? list1 list2)
  (cond
    [(empty? list1) ... ]
    [else
        ... (first list1) ... (first list2) ...
        ... (list=? (rest list1) (rest list2))... ]))
```

Reasoning about list equality again

If the first list is empty, the answer depends on whether or not the second list is empty.

If the first list is not empty, then the following should all be true:

- the second list must be nonempty

- the first elements must be equal

- the rests must be equal

Notice the importance of order of checking.

7: Processing two lists or numbers

```
(define (list=? list1 list2)
  (cond
    [(empty? list1) (empty? list2) ]
    [else
      (and (cons? list2)
            (and (= (first list1) (first list2))
                 (list=? (rest list1) (rest list2))))]))
```

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 nonatomic 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 three approaches to designing functions that consume two lists (or a list and a number, or two numbers) and know which one is (or ones are) suitable in a given situation.