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 containing all the values in list1 followed by all the values in list2.

It effectively creates a new list from list1 by replacing its empty list with list2.

Do we need to process the individual elements of list1?

What about the individual elements of list2?

```
;; list-template: (listof Any) → Any
(define (list-template alist)
  (cond [(empty? alist) ...]
        [else (... (first alist) ...
                   (list-template (rest alist)) ...)]))

;; alongforride-template: (listof Any) (listof Any) → Any
(define (alongforride-template list1 list2)
  (cond [(empty? list1) ...]
        [else (... (first list1) ...  (alongforride-template (rest list1) list2) ...)]))
```

alongforride-template is a slight variation of extra-list-info-template in M05.

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 (list 1) (list 2 3)) (list 1 2 3))

(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 (list 1 2) (list 3 4))

$\Rightarrow$ (cons 1 (my-append (list 2) (list 3 4)))

$\Rightarrow$ (cons 1 (cons 2 (my-append empty (list 3 4))))

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

$\Rightarrow$ (list 1 2 3 4)

Built-in append

The Racket function append is a built-in function like my-append.

Unlike our version, append may consume two or more lists.

You may use append on assignments, unless told otherwise.

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.

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.

Extracting the lockstep template

```
;; lockstep-template: (listof Any) (listof Any) → Any
;; requires: list1 and list2 are the same length
(define (lockstep-template list1 list2)
  (cond
    [(empty? list1) ... ]
    [else
      (... (first list1) ... (first list2) ...
           (lockstep-template (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 (list 2 3) (list 4 5))

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

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

$\Rightarrow$ (+ 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:

```
;; twolist-template: (listof Any) (listof Any) → Any
(define (twolist-template 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 situation is a **base case**.

Refining the template

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

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) ... (twolist-template list1 (rest list2)) ...
    (first list1) ... (twolist-template (rest list1) list2) ...
    (first list1) ... (first list2)
    (twolist-template (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 (twolist-template list1 list2)
  (cond
    [(and (empty? list1) (empty? list2)) ...]
    [(and (empty? list1) (cons? list2))
     (... (first list2) ... (twolist-template empty (rest list2)) ...)]
    [(and (cons? list1) (empty? list2))
     (... (first list1) ... (twolist-template (rest list1) empty)...)]
    [(and (cons? list1) (cons? list2))
     (... (first list2) ... (twolist-template list1 (rest list2)) ...
          (first list1) ... (twolist-template (rest list1) list2) ...
          (first list1) ... (first list2)...
          (twolist-template (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)) $\Rightarrow$ (list 1 2 4 6 8 10 12)

We need more examples to be confident of how to proceed.

(merge empty empty) $\Rightarrow$ empty

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

(merge (cons 1 (cons 3 empty))
empty) $\Rightarrow$ (cons 1 (cons 3 empty))

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

(merge (cons 3 (cons 4 empty))
(cons 2 empty)) $\Rightarrow$ (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]
    [(< (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))))
⇒ (cons 2 (merge (cons 3 (cons 4 empty))
                 (cons 5 (cons 6 empty))))
⇒ (cons 2 (cons 3 (merge (cons 4 empty)
                         (cons 5 (cons 6 empty)))))
⇒ (cons 2 (cons 3 (cons 4 (merge empty
                                 (cons 5 (cons 6 empty))))))
⇒ (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty)))))
```

A new data type Symbol (Sym)

Racket allow us to define and use **symbols** with meaning to us.

Symbols have a special marker to indicate that they are not function names or strings.

Syntax rule: a **symbol** starts with a single quote ’ followed by something obeying the rules for identifiers.

The symbol ’blue is a value like 5, but is more limited computationally.

Using symbols

- Symbols can be used to avoid using numbers or strings to represent a small collection of values, such as colours, names of planets, or movie genres.
- Symbols can be compared using the functions symbol=? and equal?.

```
(define favourite 'yellow)
(symbol=? favourite 'red) ⇒ false
```

Limitations: Symbols cannot be compared or processed in any other way other than checking for equality.

Symbols versus strings

Symbols

- are **atomic** (indivisible) data, like numbers
- can be compared very efficiently for equality
- should be used when only a small set of values are needed

Strings

- are **compound** data, as a string is a sequence of characters
- have lots of built-in functions
- should be used when a wide range of values are possible

Using Symbols

```
;; create-posn: Posn Sym → Posn
(define (create-posn p action)
  (cond [(symbol=? action 'rev)
         (make-posn (posn-y p) (posn-x p))]
        [(symbol=? action 'abs)
         (make-posn (abs (posn-x p)) (abs (posn-y p)))]
        [(symbol=? action 'neg)
         (make-posn (- (posn-x p)) (- (posn-y p)))]
        [else p]))
```

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.

Problem: Determining if *elem* appears at least *n* times in a list.

Example: Does 'z appear at least 3 times in (list 'd 'z 'z 'h 'z 'd)?
Does 'a appear at least 2 times?

The function at-least?

```
;; (at-least? n elem lst) produces true if elem appears at
;;    least n times in lst, and false if not.
;; at-least?: Nat Any (listof Any) → Bool
;; Examples:
(check-expect (at-least? 0 'a empty) true)
(check-expect (at-least? 3 "hi" empty) false)
(check-expect (at-least? 2 'red (list 'red 'blue 'red 'green)) true)
(check-expect (at-least? 1 7 (list 5 4 0 5 3)) false)

(define (at-least? n elem lst) ...)
```

Developing the template

The recursion will involve processing the parameters n and lst. The parameter elem is ”along for the ride”, once again giving four possibilities:

```
(and (= n 0) (empty? lst))
(and (= n 0) (cons? lst))
(and (> n 0) (empty? lst))
(and (> n 0) (cons? lst))
```

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

```
(define (list-and-nat-template n lst)
  (cond
    [(and (= n 0) (empty? lst)) ...]
    [(and (= n 0) (cons? lst))
     (... (first lst) ... (list-and-nat-template 0 (rest lst)) ...)]
    [(and (> n 0) (empty? lst))
     (... (list-and-nat-template (sub1 n) empty) ... )]
    [(and (> n 0) (cons? lst))
     (... (first lst) ... (list-and-nat-template n (rest lst)) ...
          (list-and-nat-template (sub1 n) lst) ...
          (first lst) ...
          (list-and-nat-template (sub1 n) (rest lst)) ...)]))
```

Reasoning about at-least?

What should happen when:

- n=0 and lst is empty?
- n=0 and lst is not empty?
- n>0 and lst is empty?
- n>0 and lst is not empty?

```
(define (at-least? n elem lst)
  (cond
    [(and (= n 0) (empty? lst)) true]
    [(and (= n 0) (cons? lst)) true]
    [(and (> n 0) (empty? lst)) false]
    [(and (> n 0) (cons? lst))
     (cond [(equal? elem (first lst))
            (at-least? (sub1 n) elem (rest lst))]
           [else (at-least? n elem (rest lst))])]))
```

Simplified:

```
(define (at-least? n elem lst)
  (cond [(= n 0) true]
        [(empty? lst) false]
        ;; otherwise, lst is nonempty and n > 0
        [(equal? elem (first lst))
         (at-least? (sub1 n) elem (rest lst))]
        [else
         (at-least? n elem (rest lst))]))
```

Condensed trace of at-least?

(at-least? 3 'z (list 'd 'z 'z 'h 'z 'd))

⇒ (at-least? 3 'z (list 'z 'z 'h 'z 'd))

⇒ (at-least? 2 'z (list 'z 'h 'z 'd))

⇒ (at-least? 1 'z (list 'h 'z 'd))

⇒ (at-least? 1 'z (list 'z 'd))

⇒ (at-least? 0 'z (list 'd))

⇒ true

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?

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))
     (... (first list2) ... (list=? list1 (rest list2)) ...
          (first list1) ... (list=? (rest list1) list2) ...
          (first list1) ... (first list2)... (list=? (rest list1) (rest list2))...)]))
```

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]
    [else
      (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

Note that the order in which conditions are checked is very important.

```
(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.