Types of recursion

Readings: none.

In this module:

- a glimpse of non-structural recursion

Structural vs. general recursion

All of the recursion we have done to date has been **structural**.

The templates we have been using have been derived from a data definition and specify the form of the recursive application.

We will continue to use primarily structural recursion for several more lecture modules.

But here we'll take a brief peek ahead at more general forms of recursion.

(Pure) structural recursion

Recall from Module 05:

In (pure) structural recursion, all arguments to the recursive function application (or applications, if there are more than one) are either:

- unchanged, or
- *one step* closer to a base case according to the data definition

The limits of structural recursion

```
;; (max-list lon) produces the maximum element of lon
;; max-list: (listof Num) → Num
;; requires: lon is nonempty
(define (max-list lon)
  (cond [(empty? (rest lon))  (first lon)]
        [(> (first lon) (max-list (rest lon))) (first lon)]
        [else  (max-list (rest lon))]))
```

There may be two recursive applications of max-list.

The code for max-list is correct.

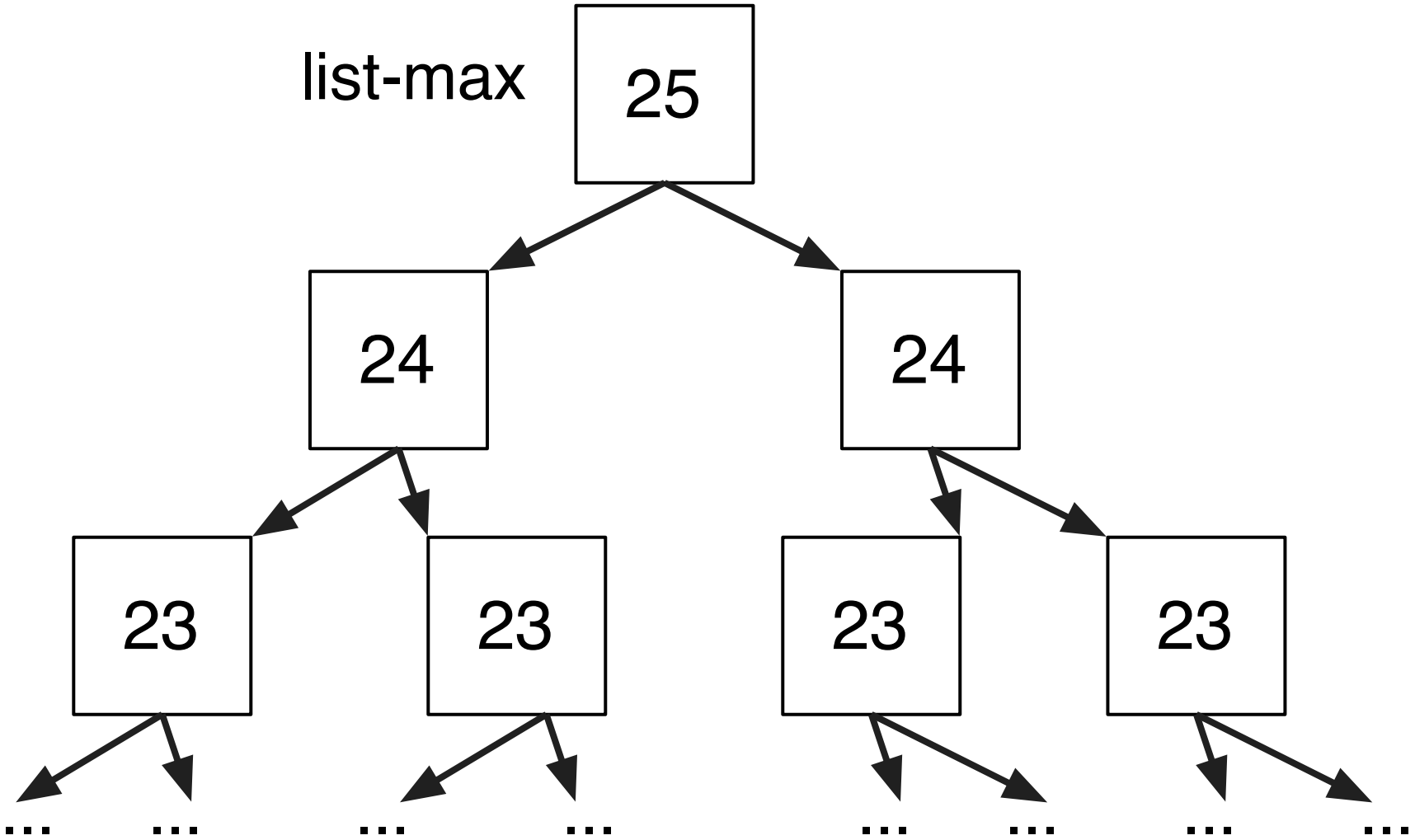
But computing (max-list (countup-to 1 25)) is very slow.

Why?

The initial application is on a list of length 25.

There are two recursive applications on the rest of this list, which is of length 24.

Each of those makes two recursive applications.


list-max
25
24
24
23
23
23
23
...
...
...
...
...
...
...
...

Intuitively, we can find the maximum of a list of numbers by scanning it, remembering the largest value seen so far.

Computationally, we can pass down that largest value seen so far as a parameter called an **accumulator**.

This parameter accumulates the result of prior computation.

This results in the code on the next slide.

```
;; (max-list/acc lon max-so-far) produces the largest
;;      of the maximum element of lon and max-so-far
;; max-list/acc: (listof Num) Num → Num
(define (max-list/acc lon max-so-far)
  (cond [(empty? lon) max-so-far]
        [(> (first lon) max-so-far)
         (max-list/acc (rest lon) (first lon))]
        [else (max-list/acc (rest lon) max-so-far)]))
```

```
(define (max-list2 lon)
  (max-list/acc (rest lon) (first lon)))
```

Now even `(max-list2 (countup-to 1 2000))` is fast.

```
(max-list2 (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
⇒ (max-list/acc (cons 2 (cons 3 (cons 4 empty))) 1)
⇒ (max-list/acc (cons 3 (cons 4 empty)) 2)
⇒ (max-list/acc (cons 4 empty) 3)
⇒ (max-list/acc empty 4)
⇒ 4
```

This technique is known as **structural recursion with an accumulator**, or sometimes **accumulative recursion**.

It is more difficult to develop and reason about such code, which is why purely structural recursion is preferable if it is appropriate.

HtDP discusses it much later than we are doing (after material we cover in lecture module 10) but in more depth.

Structural vs. accumulative recursion

In **(pure) structural recursion**, all arguments to the recursive function application (or applications, if more than one) are either:

- unchanged, or
- *one step* closer to a base case

In **structural recursion with an accumulator**, arguments are as above, plus one or more accumulators containing partial answers. The accumulatively recursive function is a helper function, and a wrapper function sets the initial value of the accumulator(s).

Another example: reversing a list

```
;; my-reverse: (listof Any) → (listof Any)
(define (my-reverse lst) ; using pure structural recursion
  (cond [(empty? lst) empty]
        [else (append (my-reverse (rest lst)) (list (first lst)))]))
```

Intuitively, append does too much work in repeatedly moving over the produced list to add one element at the end.

This has the same worst-case behaviour as insertion sort.

Reversing a list with an accumulator

```
(define (my-reverse lst)  ; wrapper function
  (my-rev/acc lst empty))

(define (my-rev/acc lst acc)  ; helper function
  (cond [(empty? lst) acc]
        [else (my-rev/acc (rest lst)
                          (cons (first lst) acc))]))
```

A condensed trace

(my-reverse (cons 1 (cons 2 (cons 3 empty))))

⇒ (my-rev/acc (cons 1 (cons 2 (cons 3 empty))) empty)

⇒ (my-rev/acc (cons 2 (cons 3 empty)) (cons 1 empty))

⇒ (my-rev/acc (cons 3 empty) (cons 2 (cons 1 empty)))

⇒ (my-rev/acc empty (cons 3 (cons 2 (cons 1 empty))))

⇒ (cons 3 (cons 2 (cons 1 empty)))

Example: greatest common divisor

In Math 135, you learn that the Euclidean algorithm for GCD can be derived from the following identity for $m > 0$:

$$gcd(n, m) = gcd(m, n \bmod m)$$

We also have $gcd(n, 0) = n$.

We can turn this reasoning directly into a Racket function.

```
;; (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm
;; euclid-gcd: Nat Nat → Nat
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

This function does not use structural or accumulative recursion.

The arguments in the recursive application were **generated** by doing a computation on m and n.

The function euclid-gcd uses **generative recursion**.

Once again, functions using generative recursion are easier to get wrong, harder to debug, and harder to reason about.

We will return to generative recursion in a later lecture module.

Structural vs. accumulative vs. generative recursion

In **(pure) structural recursion**, all arguments to the recursive function application (or applications, if there are more than one) are either unchanged, or *one step* closer to a base case.

In **accumulative recursion**, parameters are as above, plus one or more parameters containing partial answers.

In **generative recursion**, parameters are freely calculated at each step. (Watch out for correctness and termination!)

Goals of this module

You should be able to recognize uses of pure structural recursion, accumulative recursion, and generative recursion.