# The final rules

CS135 Lecture 10

# Rules of recursion (L06 version)

1. Change one argument closer to termination while recurring. No other arguments can change.
2. When recurring on a natural number use **`(sub1 n)`** and test termination with **`zero?`**
3. When recurring on a list use **`(rest lst)`** and test termination with **`empty?`**’

In this version, the argument in #1 is always the same argument and no other arguments can change.

# Finalizing the rules of recursion

In this lecture we will make two final extensions to our rules:

1. We will allow recursion on more than one argument. Arguments involved in recursion should either be unchanged or get closer to the base case. At least one argument should be closer to the base case on each recursive call.
2. Arguments that aren't involved in recursion can change.

We'll made one extension at a time…

# L10.00 Indexing a list

# The *n*th item in a list

```
;; Produce the nth item in a list
;; index: Nat (listof Any) -> Any
(define (index n lst) ...)
```

Since *n* is a natural number, it's simplest to start indexing from 0:

```
(check-expect (index 4 (list 0 1 2 3 4 5)) 4)
(check-expect (index 2 (list 'a 'b 'c 'd 'e)) 'c)
```

What to do if *n* is ≥ than the length of the list? Maybe just produce **`empty`**?

```
(check-expect (index 100 (list 0 1 2 3 4 5)) empty)
(check-expect (index 100 empty) empty)
```

# Key observation

The solution to

```
(index 2 (list 'a 'b 'c 'd 'e))
```

is the same as the solution to

```
(index 1 (list 'b 'c 'd 'e))
```

is the same as the solution to

```
(index 0 (list 'c 'd 'e))
```

On each recursive call we `sub1` from *n* and recurse on the `rest` of the list.

Eventually either *n* = 0 or the list is empty. What then?

# Recursion on a natural number and a list

If more than one argument changes we need to explicitly check all base cases.
So the body of our index function will look like one of these:

```
(define (index n lst)
  (cond [(empty? lst) ...]
        [(zero? n) ...]
        ...)

(define (index n lst)
  (cond [(zero? n) ...]
        [(empty? lst) ...]
        ...)
```

*Which one?*

# Recursion on two arguments

Let's think about what we do for each base case and decide

```
(define (index n lst)
  (cond [(empty? lst) empty]
        [(zero? n) (first lst)]
        ...)

(define (index n lst)
  (cond [(zero? n) (first lst)]
        [(empty? lst) empty]
        ...)
```

*You have all the pieces to finish the function. Try to put them together yourself.*

# Rules of recursion (almost the final version)

1. Change at least one argument closer to termination while recurring
2. When recurring on a natural number use `(sub1 n)` and test termination with `zero?`
3. When recurring on a list use `(rest lst)` and test termination with `empty?`

# L10.01 Recursion on two lists

# Vectors

In mathematics, a ***vector*** is a set of numbers arranged in a specific order, often used to represent a point in space or a direction and magnitude.

For example, <3, 5, 2> is a vector with three elements. Each element corresponds to a coordinate or a value in a specific dimension.

In Racket, we represent a vector as a list of numbers:

```
(list 3 5 2)
(list -10.6 14.8 23.7 0.03 111.8)
```

For convenience, we define the empty vector with no entries, as **`empty`**.

# Dot Product

A ***dot product*** is a way of multiplying two vectors.

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.

$$\langle 1\ 2\ 3 \rangle \cdot \langle 4\ 5\ 6 \rangle = 1 \times 4 + 2 \times 5 + 3 \times 6 = 4 + 10 + 18 = 32$$

```
(check-expect (dot-product (list 1 2 3) (list 4 5 6))32)
```

*What do we do if the lists are of different lengths?*

# Recursion on two lists

If more than one argument changes we need to explicitly check all base cases.

```
;; compute the dot product of two vectors
;; dot-product: (listof Num) (listof Num) -> Num
(define (dot-product lst1 lst2)
  (cond [(empty? lst1) 0]
        [(empty? lst2) 0]
        ...)
```

# Recursion on two lists

Change at least one argument closer to termination while recurring:

```
;; compute the dot product of two vectors
;; dot-product: (listof Num) (listof Num) -> Num
(define (dot-product lst1 lst2)
  (cond [(empty? lst1) 0]
        [(empty? lst2) 0]
        [else (+ (* (first lst1) (first lst2))
                 (dot-product (rest lst1) (rest lst2)))]))
```

In this case both change. This is called recursion in “lockstep”.

The **`index`** function is an example of a natural number and list in lockstep.

# Merging two sorted lists.

Design a function merge that consumes two lists of numbers. Each consumed list is sorted in non-descending order. Produced list is in non-descending order.

```
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: lists must be sorted in increasing order
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1] ...))
(check-expect (merge (list 2 4 6 8) (list 1 3 5 7))
              (list 1 2 3 4 5 6 7 8))
(check-expect (merge (list 1 2 2 3 3) (list 1 2 3 3))
              (list 1 1 2 2 2 3 3 3 3))
```

# Merging two sorted lists

If neither list is empty, there are three cases:

Case #1 `[(< (first lst1) (first lst2))`

Case #2 `[(> (first lst1) (first lst2))`

Case #3 `[else ;; the first elements are equal`

# Case #1 `[(< (first lst1) (first lst2)`

```
(merge (list 1 3 3 7)(list 2 3 9 11))
```

```
[(< (first lst1) (first lst2))
 (cons (first lst1) (merge (rest lst1) lst2))]
```

```
(cons 1 (merge (list 3 3 7)(list 2 3 9 11))
```

# Case #2 `[(> (first lst1) (first lst2)`

```
(merge (list 3 3 7)(list 2 3 9 11))
```

```
[(> (first lst1) (first lst2))
 (cons (first lst2) (merge lst1 (rest lst2)))]
```

```
(cons 2 (merge (list 3 3 7)(list 3 9 11)))
```

# Case #3 `[else`

```
(merge (list 3 3 7)(list 3 9 11))
```

```
[else (cons (first lst1)
            (cons (first lst2)
                  (merge (rest lst1) (rest lst2))))]
```

```
(cons 3 (cons 3 (merge (list 3 7)(list 9 11)))
```

# Merging two sorted lists.

```
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: lists must be sorted in increasing order
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(< (first lst1) (first lst2))
         (cons (first lst1) (merge (rest lst1) lst2))]
        [(> (first lst1) (first lst2))
         (cons (first lst2) (merge lst1 (rest lst2)))]
        [else (cons (first lst1)
                    (cons (first lst2)
                          (merge (rest lst1) (rest lst2))))]))
```

# L10.02 Accumulators

# Reversing a list in lecture L09

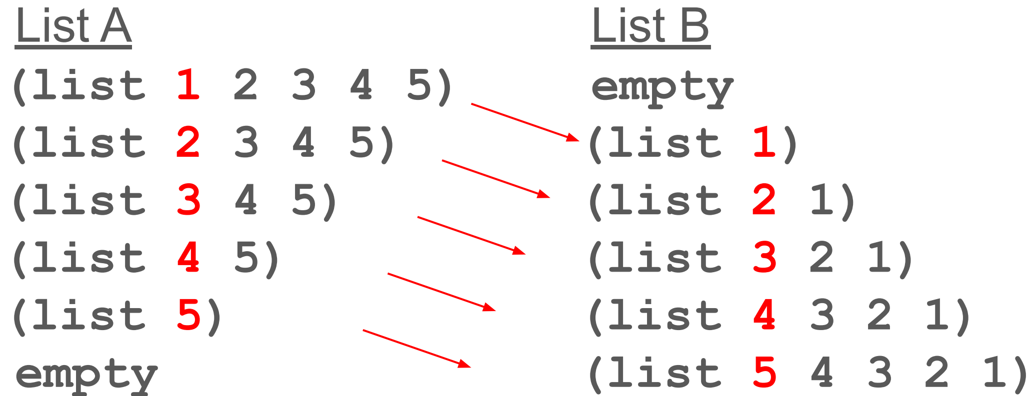
```
(define (attach element lst)
  (cond [(empty? lst) (cons element empty)]
        [else (cons (first lst) (attach element (rest lst)))]))

(define (rev lst)
  (cond [(empty? lst) empty]
        [else (attach (first lst) (rev (rest lst)))]))
```

Steps to reverse a list of length $n = 3.5n^2 + 6.5n + 32 = O(n^2)$

*Can we do better?*

# Reversing a list

Reverse would straightforward if we could build a list during recursion.




Each step of the recursion we remove an element from List A and add it to List B.

# Reversing a list

We can build a list during recursion if we allow arguments not involved in recursion to change. In this case, we will have a second argument that will increase.

```
(define (rev/accumulate lst accumulator)
  (cond [(empty? lst) accumulator]
        [else (rev/accumulate
                (rest lst)
                (cons (first lst) accumulator))]))
```

We still use **`(rest lst)`** and test termination with **`empty?`** so we know the function will terminate. The argument that increases is called an “accumulator”.

# Reversing a list

In the stepper, we see:

```
(rev/accumulate (list 1 2 3 4 5) empty) ⇒ … ⇒
(rev/accumulate (list 2 3 4 5) (list 1)) ⇒ … ⇒
(rev/accumulate (list 3 4 5) (list 2 1)) ⇒ … ⇒
(rev/accumulate (list 4 5) (list 3 2 1)) ⇒ … ⇒
(rev/accumulate (list 5) (list 4 3 2 1)) ⇒ … ⇒
(rev/accumulate empty (list 5 4 3 2 1))
```

The base case is: `[(empty? lst) accumulator]`

The base case produces the accumulator.

# Wrapping it up

Accumulators need to be initialized (in this case to **`empty`**). This initialization happens in a wrapper function.

```
(define (rev/accumulate lst accumulator)
  (cond [(empty? lst) accumulator]
        [else (rev/accumulate
                (rest lst)
                (cons (first lst) accumulator))]))
(define (rev lst)
  (rev/accumulate lst empty))
```

We have already seen other examples of wrapper functions this term.

# Efficiency

`(rev empty)` ⇒ 5 steps

`(rev (list 1))` ⇒ 12 steps

`(rev (list 1 2))` ⇒ 19 steps

`(rev (list 1 2 3))` ⇒ 26 steps

…

`(rev (list 1 2 3 4 5 6 7 8 9 10))` ⇒ 75 steps

Total steps (where $n$ is the length of the list): $7n + 5 = O(n)$, i.e., linear.

# Built-in `reverse`

DrRacket has a built-in function **`reverse`** that you can now use as appropriate.

Like the built-in functions **`length`** and **`append`**, you should think of the efficiency of **`reverse`** as linear in the size of the consumed list even though in the stepper it only takes one step.

Recursion using an accumulator (“accumulative recursion”) can be tricky. If you write a function where the answer comes out “backwards” it can be better to **`reverse`** the answer in a wrapper, rather than trying get an accumulator working.

Be super-careful to avoid **`reverse`** in the body of a recursive function.

# Rules of recursion (final version)

1. Change at least one argument closer to termination while recurring
2. When recurring on a natural number use **`(sub1 n)`** and test termination with **`zero?`**
3. When recurring on a list use **`(rest lst)`** and test termination with **`empty?`**
4. Arguments that aren’t involved in recursion can change.

# Lecture 10 Summary

# L10: You should know

- The final “rules of recursion”. You must stick to these rules for the rest of the term unless we say otherwise.
- Recursion on a natural number and a list.
- Recursion on two lists, including the “lockstep” pattern.
- Accumulators and accumulative recursion.
- How to reverse a list using an accumulator.

# L10: Allowed constructs

Newly allowed constructs:
**`reverse`**
Rules of Recursion (final version)

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ;
abs acos add1 and append asin atan check-expect check-within
cond cons cons? cos define e else empty empty? exp expt
false first inexact? integer? length list list? log max min
not number? or pi quotient rational? remainder rest second
sin sqr sqrt sub1 symbol? symbol=? tan third true zero?
listof Any anyof Bool Int Nat Num Rat Sym
```