Class 11

• Warmup
  • Second, swap
  • Right-max: recurrence?
  • Function composition

• Natural number recursion example

• Bignum introduction

• Two-argument recursion, four flavors
  • Both arguments reduce
  • One argument reduces; other remains the same
  • One argument reduces; other grows (!)
Warmup

• Two *nonrecursive* programs
Second

;; return the second item in an a list
;; of length at least two.
;;
Second

;; return the second item in an a list
;; of length at least two.
;; second: ???  -->  ???
Second

;; return the second item in an a list
;; of length at least two.
;; second: 'a list -> 'a
Second

;; return the second item in an a list
;; of length at least two.
;; second: 'a list -> 'a
;;
;;
;;
;;
;;
(define (second alod) ...)
Second

;; return the second item in an a list
;; of length at least two.
;; second: 'a list -> 'a

;; input: alod, a list containing at least two items
;; output: the second item in alod
(define (second alod) ...)
Swap

(swap (list 1 2)) => (list 2 1)
;; Swap the items in a list of length two
;; swap: 'a list -> 'a list
;; input:
;;    alod, a list containing exactly two items
;; output: a list with the same two elements in
;;     the
;; opposite order
(define (swap alod) …)
Warmup 2: natural number recursion

;; zlist : num -> num list
;;
;; NB: a natural number is either
;; zero
;; the successor of (i.e., one greater than) a natural number
;; nothing else is a natural number
;; input:
;; output:
Warmup 2: natural number recursion

;; zlist : num -> num list
;;
;; NB: a natural number is either
;;    zero
;;    the successor of (i.e., one greater than) a natural number
;;    nothing else is a natural number
;; input: n, a natural number
;; output: a num list containing n copies of the number 0
(define (zlist n)
  (cond
    [(zero? n) ...]
    [(> n 0) ...]))
Warmup 2: natural number recursion

;; zlist : num -> num list
;;
;; NB: a natural number is either
;;    zero
;;    the successor of (i.e., one greater than) a natural number
;;    nothing else is a natural number
;; input: n, a natural number
;; output: a num list containing n copies of the number 0
(define (zlist n)
  (cond
   [(zero? n) empty]
   [(> n 0) (cons 0 (zlist (- n 1)))]))
Lab

• **We** provide two extra procedures: `pred` and `succ?` D
  • The first one gives you the natural number just before n (i.e., n-1), the *predecessor*
  • Second one tests whether a natural number has a predecessor, i.e., whether it's a *successor* (i.e., nonzero)
  • Template ends up looking more like the one for lists:

```scheme
(define (proc n)
  (cond
    [(zero? n) empty]
    [(succ? n) (... (proc ... (pred n) ...))])
)```
Bignum

• Longer project
• Work in pairs
• Design check
• Read project handout *now*, find partner
• Main problem: write
  \((\text{bignum+ bignum1 bignum2})\)

which is a two-argument recursive procedure
• A compact version is about 12-15 lines
Two-argument recursion

• Almost every procedure today will work on num lists
  • A num list is either empty, or (cons item lst), where item is a num and lst is a num list.
  • Nothing else is an num list.
  • [Now we don't need to do that part of the design recipe]

• Examples
  (define ex0 empty)
  (define ex1 (list 1 3))
  (define ex2 (list 1 4 7))
  (define ex3 (list 2 4 9))

• Most procedures will use two arguments
Warmup: two argument recursion, list + atomic

• We wrote "improve", which converted (list 3 7 10) to (list 17 17 17)

• Recursion diagram:
  Orig. in.: (list 3 7 10)
    Rec. In: (list 7 10)
    Rec. out: (list 17 17)
  Cons on a 17 to the recursive output?
  Orig out (list 17 17 17)
Warmup: two argument recursion, list + atomic

(define (improve alon)
  (cond
    [(empty? alon)  empty]
    [(cons? alon)  (cons 17 (improve (rest alon))))]))
New: (**better** alon num)

- Better produces a new list of the same length as alon, but with each item replaced with num:
- `(better (list 3 7 10) 14)` produces `(list 14 14 14)`
- Recursion diagram:
  Orig. in.: `(list 3 7 10), 14`
  - Rec. In: `(list 7 10), 14`
  - Rec. out: `(list 14 14)`
  Cons on a "num" to the recursive output?
  Orig. out: `(list 14 14 14)`
New: (better alon num)

(define (better alon num)
  (cond
    [(empty? alon)   empty]
    [(cons? alon)   (cons num (better (rest alon) num))])))
General rules for **natural num-and-list** recursion

• Recursive input is often the *same* natnum, and "rest" of the list (e.g., *better*)

• Alternatives:
  1. reduce natnum by 1, keep the list the same; base case is when natnum is zero
  2. reduce natnum by 1, take "rest" of the list (i.e., reduce both)
  3. reduce natnum by 1, somehow *enlarge* the list
  4. ...

• Example of case 2: to find the *n*th item in a list (of at least *n* items)...
  • find the *(n-1)*st item in the *rest* of the list!

• This weekend's lab: natural-number recursion
Generalities

• More arguments means more complexity
• More arguments means more things you might recur on... and just about any possibility might arise.
• More arguments means more structures in the main "cond":
  • empty, empty
  • cons, empty
  • empty, cons
  • cons, cons
• For two-list recursion, three main possibilities for recursive input: shrink both, shrink one, shrink-one-grow-one
Example proc: "zip"

; zip: (num list) * (num list) -> (num list) list
; Input: lists alon and alon2, of the same length
; Output: a list consisting of "pairs" (two item lists), the first output
; pair containing the first item in each list; the second output pair
; containing the second item for each list, and so on.

(check-expect (zip (list 1 2 5) 
    (list 1 3 5))
  (list (list 1 1)
    (list 2 3)
    (list 5 5)))

(check-expect (zip empty empty) empty)
(check-expect (zip (list 4) (list 4)) (list (list 4 4)))
Decision time: on which input(s) to recur?

Original inputs: (list 1 2 5)    (list 1 3 5)
  Recursive inputs: ???   ???
  Recursive result:

Original result: (list 1 5)
Decision time

Original inputs: (list 1 2 5)  (list 1 3 5)
Recursive inputs: ???  ???

• Three basic choices
  • Shorten the first list; leave other unchanged
  • Shorten the second list; leave first unchanged
  • Shorten both lists

• More exotic choice
  • Shorten first list, lengthen the second (!)
  • Base case: first list is empty (but second could be huge!)

• For this problem: which one?
  • Shorten both; otherwise recursive input doesn't meet input spec (equal-length lists)
Decision time

Original inputs: (list 1 2 5)  (list 1 3 5)

Recursive inputs: (list 2 5)   (list 3 5)

Recursive result: (list (list 2 3)
                  (list 5 5))

Original result: (list (list 1 1) (list 2 3) (list 5 5))
Decision time

Original inputs: (list 1 2 5)  (list 1 3 5)
Recursion inputs: (list 2 5)  (list 3 5)
Recursive result: (list (list 2 3)
                (list 5 5))

Cons a pair onto the recursive result
Pair contains "first" of each of the input lists

Original result: (list (list 1 1) (list 2 3) (list 5 5))
Write the call-structure and template

(define (zip alon alon2)
  (cond
    [???
      • Wait...what are the possible kinds of input?
      • Each of alon and alon2 could be empty or cons!)
Write the call-structure and template

(define (zip alon alon2)
  (cond
   [(and (empty? alon) (empty? alon2)) …]
   [(and (empty? alon) (cons? alon2)) …]
   [(and (cons? alon) (empty? alon2)) …]
   [(and (cons? alon) (cons? alon2)) …]))

• For our particular procedure, where the lengths must be the same, two of these never occur
Write the call-structure and template

(define (zip alon alon2)
  (cond
    [(and (empty? alon) (empty? alon2)) …]
    [(and (cons? alon) (cons? alon2)) …])

• What goes in the … places?
  • For the first, the base case
  • For the second, something involving first and rest of the two lists
  • The "rests" are involved in a recursive call (because of our diagram)
Write the code!

(define (zip alon alon2)
  (cond
   [(and (empty? alon) (empty? alon2)) ...]
   [(and (cons? alon) (cons? alon2))
    ... (first alon) ... (first alon2) ... (zip (rest alon) (rest alon2))]]

Cons a pair, consisting of "first" of each of the inputs, onto the recursive result

• Work with your neighbor to fill in the second case. (The first is empty)
Write the code!

```
(define (zip alon alon2)
  (cond
    [(and (empty? alon) (empty? alon2)) empty]
    [(and (cons? alon) (cons? alon2))
      (cons (list (first alon) (first alon2))
        (zip (rest alon) (rest alon2))))])
```
New Problem! Which list is longer?

; Input: two num lists alon1 and alon2
; Output: a boolean, true if alon1 is longer than alon2, 
; false if they're the same length or alon2 is longer.

(define (longer? alon1 alon2)
  ...
)

• Test cases?
A not-so-good answer

• Compute length of each list, then compare the numbers
• I'll explain why it's not so good a little later
Which list is longer?

; Input: two num lists alon and alon2
; Output: a boolean, true if alon is longer than alon2,
; false if they're the same length or alon2 is longer.

(define (longer? alon alon2)
    ...)

(check-expect (longer? (list 1 2 5) (list 1 3)) true)
(check-expect (longer? (list 1 2) (list 2 4 6)) false)
(check-expect (longer? empty empty) false)
(check-expect (longer? (list 4) empty) true)
(check-expect (longer? empty (list 4)) false)
Decision time

Original inputs: (list 1 2 5)   (list 1 3)
Recursive inputs: ???  ???
Recursive result: ???

Original result: true
Decision time

Original inputs: (list 1 2 5) (list 1 3)
Recursive inputs: (list 2 5) (list 3)
Recursive result: true

Copy recursive result?

Original result: true
Decision time

Original inputs: (list 1 2) (list 2 4 6)
  Recursive inputs: (list 2) (list 4 6)
  Recursive result: false

Copy recursive result?

Original result: false
Another diagram!

Original inputs: (list 4), empty
  Recursive inputs: empty, NA
  Recursive result:
  This must be another base case!
If first list empty: false
If both empty: false
If first cons, second empty: true
Otherwise recur on both and copy recursive result

Original result: true
Which list is longer?

; Input: two num lists alon1 and alon2
; Output: a boolean, true if alon1 is longer than alon2,
; false if they're the same length or alon2 is longer.

(define (longer? alon1 alon2)
  ...)

(check-expect (longer? (list 1 2 5) (list 1 3)) true)
(check-expect (longer? (list 1 2) (list 2 4 6)) false)
(check-expect (longer? empty empty) false)
(check-expect (longer? (list 4) empty) true)
(check-expect (longer? empty (list 4)) false)

• Template?
Which list is longer?

; Input: two num lists alon1 and alon2
; Output: a boolean, true if alon1 is longer than alon2,
; false if they're the same length or alon2 is longer.

(define (longer? alon1 alon2)
  (cond
    [(and (empty? alon1) (empty? alon2)) ...]
    [(and (cons? alon1) (empty? alon2)) ... (first alon1)... (rest alon1) ... ]
    [(and (empty? alon1) (cons? alon2)) ... (first alon2)... (rest alon2) ... ]
    [(and (cons? alon1) (cons? alon2)) ... (first alon1)... (rest alon1) ...
      (first alon2)... (rest alon2) ... ]
  )

• there's likely to be a recursive call in the last line, and possibly the two before it.
(define (longer? alon1 alon2)
  (cond
   [(and (empty? alon1) (empty? alon2)) false]
   [(and (cons? alon1) (empty? alon2)) true]
   [(and (empty? alon1) (cons? alon2)) false]
   [(and (cons? alon1) (cons? alon2)) (longer? (rest alon1) (rest alon2))])))
What we've seen

• Two examples of "shorten both lists"
• No examples of "shorten one, keep other the same" (less common, but there's one on the HW)
• We're about to see shorten-one, lengthen-one
One more challenge: list reversal

- Define the problem by examples, for expedience:
  (check-expect (my-reverse (list 1 2 3)) (list 3 2 1))
  (check-expect (my-reverse empty) empty)
  (check-expect (my-reverse (list 1)) (list 1))
;; my-reverse: (num list) -> (num list)
;; input: alod, a list of num
;;
;; output: a list that’s contains the same items as alod,
;; but in reverse order

(check-expect (my-reverse (list 1 2 3)) (list 3 2 1))
(check-expect (my-reverse empty) empty)
(check-expect (my-reverse (list 1)) (list 1))
Recursive diagram

Original input: (list 1 2 5)

Recursive inputs: (list 2 5)
  Recursive result: (list 5 2)

Stick (first alon) on as the last item in the list?
Could write another proc (a helper) to do this...

Original result: (list 5 2 1)
Bad idea

• This approach to reversing a list ends up being slow and complicated
• Turns out there's an easier way
• Involves *generalizing* the problem
  • Oddly, in CS, sometimes it's easier to solve a more general problem than a more specific one!
  • Finding the right "generalized problem" requires insight and experience, alas.
Reverse-with-tail

• > (reverse-with-tail (list 1 2 3) (list 4 5)) (list 3 2 1 4 5)
• > (reverse-with-tail (list 1 2 3) empty) (list 3 2 1)

;; reverse-with-tail: (num list) * (num list) -> (num list)
;; input: alon, a list of numbers
;; tail, another list of numbers
;; output: the elements of alon, in reverse order, followed by
;; those of tail, in their original order
Why is this a good idea?

• If we can write reverse-with-tail, then reverse becomes easy:

(define (reverse alon)
  (reverse-with-tail alon empty))
Recursive diagram for reverse-with-tail

Original input: (list 1 2 5) (list 3 4)
Recursive inputs: (list 2 5) ???
  Recursive result: (list 5 2 1 3 4)

just copy the recursive result!

Original result: (list 1 5 2 3 4)
Big idea

• from alon, tail, recur to (rest alon), (cons (first alon) tail)
• First list gets shorter, so we eventually hit a base case (first list empty)
• Second list grows!
(define (reverse-with-tail alon tail)
  (cond
    [(and (empty? alon) (empty? tail)) empty]
    [(and (empty? alon) (cons? tail)) tail]
    [(and (cons? alon) (empty? tail))
      (reverse-with-tail (rest alon)
                           (cons (first alon) tail))]
    [(and (cons? alon) (cons? tail))
      (reverse-with-tail (rest alon)
                           (cons (first alon) tail))])))
Simplify

(define (reverse-with-tail alon tail)
  (cond
   [(and (empty? alon) (empty? tail)) empty]
   [(and (empty? alon) (cons? tail)) tail]
   [(and (cons? alon) (empty? tail))
     (reverse-with-tail (rest alon) (cons (first alon) tail))]
   [(and (cons? alon) (cons? tail))
     (reverse-with-tail (rest alon) (cons (first alon) tail))])))
(define (reverse-with-tail alon tail)
  (cond
   [(and (empty? alon) (empty? tail)) empty]
   [(and (empty? alon) (cons? tail)) tail]
   [(cons? alon)
     (reverse-with-tail (rest alon) (cons (first alon) tail))])))
(define (reverse-with-tail alon tail)
  (cond
   [(and (empty? alon) (empty? tail)) empty]
   [(and (empty? alon) (cons? tail)) tail]
   [(cons? alon)
     (reverse-with-tail (rest alon) (cons (first alon) tail))])))
Simplify

(define (reverse-with-tail alon tail)
  (cond
   [(and (empty? alon) (empty? tail)) tail]
   [(and (empty? alon) (cons? tail)) tail]
   [(cons? alon)
    (reverse-with-tail (rest alon)
       (cons (first alon) tail))])
)
Simplify

(define (reverse-with-tail alon tail)
  (cond
   [((empty? alon) tail]
   [(cons? alon)
     (reverse-with-tail (rest alon) (cons (first alon) tail))]))}
Simplify

(define (reverse-with-tail alon tail)
  (cond
    [(empty? alon) tail]
    [(cons? alon) (reverse-with-tail (rest alon) (cons (first alon) tail))]))
Simplify

(define (reverse-with-tail alon tail)
  (cond
    [(empty? alon) tail]
    [(cons? alon) (reverse-with-tail (rest alon)
                                (cons (first alon) tail))]))

• Really just a recursive program based on the structure of a *single* list, alon
• Hence only two cases: the empty? case and the cons? case
Today's topics

• Natural number recursion example
• Bignum introduction
• Two-argument recursion, four flavors
  • Both arguments reduce
  • One argument reduces; other remains the same (2 cases)
  • One argument reduces; other grows (!)
• Zip
• Longer
• Reverse
• Generalizing to simplify: a profound CS strategy