Today’s topics

• Predicates
• Natural Number recursion
• Recursion Diagrams
• List recursion
• A first glance at the design recipe
Predicates: things that test stuff

(> 3 5) : tests whether 3 is greater than 5
(= 4 7) : tests for equality of numbers
(zero? 5) : test to see whether the number 5 is zero or not.

All of these are functions that produce booleans; such functions are called predicates. Their names often end with a question mark.

Other important ones:
(empty? myList) : test to see whether the list is the empty list
(cons? myList) : test to see whether the list is a nonempty list
What predicates/builtin can we use?

• In general, each time a builtin is introduced in class, you can use it in the homework thereafter.
  • This includes generalizations ... having seen “>”, you can also use “<“.
• So far: +, -, *, /, zero?, cons, car, cdr, >, =
• Note: “car” and “cdr” are classic names for “the first item in a list” and “the whole list except for the first item”. Scheme/Racket also allow the names “first” and “rest”, which I’ll use.
  • They are synonyms for car and cdr, respectively.
Natural Numbers

• *For CS17, the natural numbers* are 0, 1, 2, ... [some books start at 1]
• Predicates for natural numbers: zero? and succ?
  - \((\text{succ} \? \ n)\) : test to see if the number \(n\) is a natural number other than 0
  - \((\text{pred} \ n)\) : if \(n\) is a successor, return the predecessor of \(n\)

  • *Note: succ? and pred are not procedures which are actually included in the Racket language. However, they are helpful procedures for explaining recursion.*

• For every natural number, either zero? or succ? returns true, but not both!
• We’re going to write programs that consume natural numbers and produce ... something
  - Other natural numbers
  - Real numbers
  - Lists of numbers
  - ...
Natural numbers in programming languages

- Few programming languages have a notion of “natural number” built in; instead they have “integers”, often shortened to “int”
- We’ll generally say what kinds of data our programs consume and produce
  - Mostly stuck with what the language provides rather than what we want
- For instance, we might write
  ```
  ;; sum: int list -> int
  ```
  to indicate that “sum” is a procedure that consumes a list of integers and produces a single integer.
- Allowable type-names will get refined as we go along.
What about natural numbers?

• If we want to say that a procedure consumes only natural numbers, how do we say that?
  • In the “type signature” we use “int”
  • In the comments, we say “only natural numbers allowed”
  • It’s a compromise
Harmonic numbers

• Phil wrote a function H that computed harmonic numbers.
• He left out part of the code that I always include – comments
• His code:
  (define H
    (lambda (n)
      (cond
        [(zero? n) 0]
        [#true (+ (/ 1 n) (H (- n 1)))]))))
My version

;;; H: int -> real
;;; input: a natural number, n
;;; output: the nth harmonic number, 1 + 1/2 + ... + 1/n.
;;;         For n = 0, output is 0.
(define H
  (lambda (n)
    (cond
     [(zero? n) 0]
     [#true  (+ (/ 1 n) (H (- n 1)))]))
)
What do the comments do?

• They help humans read your program
• They give you something to write while you’re suffering with not knowing how to get started
• Racket processing completely ignores them
Is the harmonic number procedure correct?

• If we evaluate \((H \ 0)\), what should we get?
• What about \((H \ 1)\)? \((H \ 3)\)?
<try it out>
• Seems to work
Restructure the program a tiny bit

• Every natural number is either
  • Zero or
  • The successor of exactly one natural number
    • 4 is the successor of 3, for instance
  • If \( n \) is a successor, then \( (\text{pred } n) \) produces its predecessor
    • \( (\text{pred } 4) \) evaluates to 3

• A natural way to write a program that uses natural number recursion is to split into cases based on this
My version (restructured)

;;; data:
;;;   natural numbers are 0 or successors
;;;     examples: 0, 1, 17
;;;   real numbers are things on the number line
;;;     examples: 0, -2.5, 73.042
;;; H: int -> real
;;; input: a natural number, n
;;; output: the nth harmonic number, 1 + ½ + ... + 1/n.
;;;         For n = 0, output is 0.
(define H
  (lambda (n)
    (cond
      [(zero? n)  0]
      [(succ? n) (+ (/ 1 n) (H (pred n)))])))
More stuff you should do

• Write down some examples of what you expect the procedure to produce.

• You should do this before writing the procedure!

• Why?
  • Because you have human failings
  • Programming is a fundamentally human activity
  • We design our programming habits to take advantage of human strengths and avoid human weaknesses
My version (restructured, 2)

;; data:
;;   natural numbers are 0 or successors
;;   examples: 0, 1, 17
;;   real numbers are things on the number line
;;   examples: 0, -2.5, 73.042
;; H: int -> real
;; input: a natural number, n
;; output: the nth harmonic number, 1 + ½ + ... + 1/n.
;;   For n = 0, output is 0.
(define H
  (lambda (n)
    (cond
     [(zero? n) 0]
     [(succ? n) (+ (/ 1 n) (H (pred n)))])))
;; (H 0): expect 0
;; (H 1): expect 1
;; (H 4): expect 1 + ½ + 1/3 + ¼ = 25/12 ~ 2.083
Summorial

• For natural numbers, the “summorial” is defined informally by “summorial of n is $1 + 2 + \ldots + n$”, the sum of the first n numbers.

• What’s the summorial of zero?
  • Ambiguous or unclear from the informal definition.

• More formal (and recursive) description:
  • Summorial of 0 is 0
  • Summorial for nonzero n: the sum of n with the summorial of the predecessor of n.

• Example: summorial of 1 is the sum of 1 with the summorial of the predecessor of 1
  • summorial of 1 is the sum of 1 with the summorial of 0
  • summorial of 1 is the sum of 1 with 0
  • summorial of 1 is 1.

• Talk with your neighbor and compute, by hand, the summorial of 0, 1, 2, 3, 4.
Let’s get ready to write summorial

• A few of you may already have figured it out
  • Feel free to sit in quiet smugness
  • Learn about the general method I’m introducing, because it’ll be your lifeline later!

• We have some example cases:
  • \((\text{summorial } 0)\) is 0
  • \((\text{summorial } 1)\) is 1
  • \((\text{summorial } 2)\) is 3
  • \((\text{summorial } 3)\) is 6
  • \((\text{summorial } 4)\) is 10
Recursion Diagrams: a helper for recursive code

• Five parts
  • Overall (or “original”) input
  • Recursive input
  • Recursive output
  • Ideation space
  • Overall output

• We fill these in *before writing any programs at all*

• We do so in a specific order: overall input and output, then recursive input and output, then ideation space material

• Best to start with the header things, though, just to be clear
Summorial header

;; data:
;;   natural numbers are 0 or successors
;;   examples: 0, 1, 17
;; summorial : int -> int
;; input: a natural number, n
;; output: the sum of the first n numbers, 1 + 2 + ... + n
;;         For n = 0, output is 0.
Summorial footer: some test cases

;; data:
;;   natural numbers are 0 or successors
;;   examples: 0, 1, 17
;; summorial : int -> int
;; input: a natural number, n
;; output: the sum of the first n numbers, 1 + 2 + ... + n
;;         For n = 0, output is 0.
...
;; (summorial 0) should be 0
;; (summorial 2) should be 3
;; (summorial 4) should be 10
A recursion diagram for (summorial 4)

Start with this:

Original input:
  Recursive input:
    Recursive output:

Original output:
A recursion diagram for (summorial 4)

Fill in the original input

Original input: 4
  Recursive input:
  Recursive output:

Original output:
A recursion diagram for (summorial 4)

Fill in the original output, using the description of the procedure

Original input: 4
Recursive input:
Recursive output:

Original output: 10
A recursion diagram for (summorial 4)

Figure out a recursive input --- for natural number recursion, the *predecessor of the original input*.

Original input: 4  
  Recursive input: 3  
  Recursive output:

Original output: 10
A recursion diagram for (summorial 4)

Figure out a recursive output --- using the description of the procedure

Original input: 4
  Recursive input: 3
  Recursive output: 6

Original output: 10
A recursion diagram for (summorial 4)

Try to see how to get from recursive output to original output – toss in lots of ideas

Original input: 4
  Recursive input: 3
  Recursive output: 6

Original output: 10
A recursion diagram for (summorial 4)

Try to see how to get from recursive output to original output – toss in lots of ideas

Original input: 4
  Recursive input: 3
  Recursive output: 6

Add 4 to recursive output?
Add original input to recursive output?
Multiply by 10/6?

Original output: 10
Which idea do we use?

• Make another diagram with different data --- perhaps with original input 3.
A recursion diagram for (summorial 3)

Original input: 3
  Recursive input: 2
  Recursive output: 3

Original output: 6

Add 4 to recursive output?
Add original input to recursive output?
Multiply by 10/6?
Now we’re ready to write a program

- The overall structure is exactly the same as the structure for \( H \):
  - Two cases, when \( n \) is zero, or when \( n \) is a successor
  - The 0 case is easy (“the base case”) and the answer is just written in
  - The successor case almost always involves using the same procedure, applied to the predecessor of \( n \).

\[
\text{(define summorial} \\
\text{ (lambda} (n) \\
\text{ (cond} \\
\text{ \quad [(zero? } n) \ldots] \\
\text{ \quad [(succ? } n) (\ldots (summorial (pred n)))]}))
\]
Completing the program (1)

• Fill in the answer for the base case:
  
  `(define summorial
   (lambda (n)
     (cond
       [(zero? n) 0]
       [(succ? n) (... (summorial (pred n)))])))`
Completing the program (2)

- Fill in the answer for the recursive case:
  
  ``` Scheme
  (define summorial
    (lambda (n)
      (cond
       [(zero? n) 0]
       [(succ? n) (+ n (summorial (pred n)))])))
  ```
The whole program

;; data:
;;   natural numbers are 0 or successors
;;   examples: 0, 1, 17
;; summorial : int -> int
;; input: a natural number, n
;; output: the sum of the first n numbers, 1 + 2 + ... + n
;;     For n = 0, output is 0.
(define summorial
  (lambda (n)
    (cond
      [(zero? n) 0]
      [(succ? n) (+ n (summorial (pred n)))])))

;; (summorial 0) should be 0
;; (summorial 2) should be 3
;; (summorial 4) should be 10
Quiz: write \texttt{square-sum}

- Input: a natural number \( n \)
- Output: the sum of the squares of integers up to \( n \), i.e., \( 1*1 + 2*2 + \ldots + n*n \)
- When \( n \) is 0, the output is zero.
- Two test-cases
  - \((\texttt{square-sum 0})\) should be 0
  - \((\texttt{square-sum 3})\) should be \( 1*1 + 2*2 + 3*3 = 1 + 4 + 9 = 14 \).
Quiz: write square-sum (just the green part)

;; data:
;;    natural numbers are 0 or successors
;;    examples: 0, 1, 17
;; square-sum: int -> int
;; input: a natural number, n
;; output: the sum of the first n squares, $1^2 + 2^2 + \ldots + n^2$
;;        For n = 0, output is 0.
(define square-sum
  (lambda (n)
    (cond
     [(zero? n) 0]
     [(succ? n) (square-sum (pred n))]))))

;; (square-sum 0) should be 0
;; (square-sum 3) should be 14
Solution

;;; data:
;;;   natural numbers are 0 or successors
;;;   examples: 0, 1, 17
;;; square-sum: int -> int
;;; input: a natural number, n
;;; output: the sum of the first n squares, 1*1 + 2*2 + ... + n*n
;;;       For n = 0, output is 0.
(define square-sum
  (lambda (n)
    (cond
      [(zero? n) 0]
      [(succ? n) (+ (* n n) (square-sum (pred n)))])))

;; (square-sum 0) should be 0
;; (square-sum 3) should be 14
More interesting recursion: lists as data

• For now, I’ll stick with “monotype” lists, where all the items in the lists are the same type
• In fact, I’ll stick with just lists of integers to teach you list recursion.
Building lists: the basics

• There are exactly two ways to construct lists
  • `empty`, which evaluates to the empty list
  • `(cons item my-list)`, which takes an existing list, and makes a larger list with the new item as its first element, and then all the elements of my-list following it.
  • You’ve learned about `quote`, but secretly `quote` is just using `empty` and/or `cons`.
  • There are associated predicates `empty?` and `cons?`.
The defining rules for lists

• \((first\ (cons\ \text{item}\ \text{my-list}))\) evaluates to \text{item}
• \((rest\ (cons\ \text{item}\ \text{my-list}))\) evaluates to \text{my-list}

• Examples:
  • \((first\ (cons\ 1\ empty))\) => 1
  • \((rest\ (cons\ 1\ empty))\) => empty
  • \((first\ (cons\ 1\ (cons\ 4\ empty)))\) => 1
  • \((rest\ (cons\ 1\ (cons\ 4\ empty)))\) => \((cons\ 4\ empty)\)
Recursion on lists

• Natural numbers came in two types: zero, or successors
• Lists come in two types: empty, and cons-lists
• Natural-number recursion programs have a cond with two cases
• So do list-recursion programs!
• Let’s write a program that computes the length of a list of integers.
Header first!

;;; data:
;;;  an int list is either
;;;    - empty or
;;;    - (cons n aloi) where n is an integer and aloi is an int list
;;; len: int list -> int
;;; input: an int-list, aloi
;;; output: the number of items in aloi; if aloi is empty, this is 0.
an int list is either
- empty or
- (cons n aloi) where n is an integer and aloi is an int list

len: int list -> int

input: an int-list, aloi
output: the number of items in aloi; if aloi is empty, this is 0.

(len empty) should be 0
(len (cons 17 empty)) should be 1
(len (cons 3 (cons 17 empty))) should be 2
“Template” code for list recursion: two cases!

;; data:
;; an int list is either
;; - empty or
;; - (cons n aloi) where n is an integer and aloi is an int list
;; len: int list -> int
;; input: an int-list, aloi
;; output: the number of items in aloi; if aloi is empty, this is 0.
(define len
  (lambda (aloi)
    (cond
      [(empty? aloi) ???]
      [(cons? aloi) (??? (len ??? (first aloi) ??? (rest aloi))))]))
;; (len empty) should be 0
;; (len (cons 17 empty)) should be 1
;; (len (cons 3 (cons 17 empty))) should be 2
Draw a recursive diagram for the overall input

\((\text{cons} \ 3 \ (\text{cons} \ 17 \ \text{empty}))\)

Original input: \((\text{cons} \ 3 \ (\text{cons} \ 17 \ \text{empty}))\)
- Recursive input: (hint: \textit{usually the “rest” of the original input!})
- Recursive output:

Original output:
Solution (with some possible ideas)

Original input: \((\text{cons} \ 3 \ (\text{cons} \ 17 \ \text{empty}))\)
  Recursive input: \((\text{cons} \ 17 \ \text{empty})\)
  Recursive output: 1

Original output: 2

Always answer "2"?
Add 1 to the recursive output?
Write the code: base case

;; data:
;;   an int list is either
;;     - empty or
;;     - (cons n aloi) where n is an integer and aloi is an int list
;; len: int list -> int
;; input: an int-list, aloi
;; output: the number of items in aloi; if aloi is empty, this is 0.
(define len
  (lambda (aloi)
    (cond
      [(empty? aloi) 0]
      [(cons? aloi) (??? (len ??? (first aloi) ??? (rest aloi)))]]))

;; (len empty) should be 0
;; (len (cons 17 empty)) should be 1
;; (len (cons 3 (cons 17 empty))) should be 2
Write the code: recursive case

;;; data:
;;;   an int list is either
;;;     - empty or
;;;     - (cons n aloi) where n is an integer and aloi is an int list
;;; len: int list -> int
;;; input: an int-list, aloi
;;; output: the number of items in aloi; if aloi is empty, this is 0.
(define len
  (lambda (aloi)
    (cond
      [(empty? aloi) 0]
      [(cons? aloi) (+ 1 (len (rest aloi)))])))

;; (len empty) should be 0
;; (len (cons 17 empty)) should be 1
;; (len (cons 3 (cons 17 empty))) should be 2
The length procedure

• The thing we just wrote – `len` – is actually a builtin in Racket
• But it’s called “`length`” instead
• You can use it from now on
  • But you’ll seldom need to --- avoid it if you can!
  • Example: you could check if the length of a list is zero, or you could just use
    `(empty? my-list)`
  • The latter is much preferred, for reasons you’ll soon learn
A challenge

• Can you write \texttt{list-sum}, which adds up all the ints in an int list?
• For the empty list, it should return 0 (because I say so)
Discussion

• Why all this header/footer nonsense?
• Why write tests before writing the program?
• Will we always use the templates you showed us today for every natural-number--recursion or list-recursion program we write?