Class 19

review
combinatorics
bignum
ReasonML rapid tour, day 2
• Reason so far:
• Basic types: int, float, string, bool, list(int), list('a), function types [more to come]
• Builtins: +, etc.; +., etc., float_of_int (and int_of_float, and string_of_bool and ...), &&, |, ! (boolean and, or, not)
• "let" instead of (define...); each "let" opens a new environment.
• Arithmetic is "infix"; other functions are applied by writing f(x) or f(x,y)
• Type-ascription: (x:int) means "x is an expression whose value has type int"
• Lists: [], [1,2,3], [3, ... [4, 5]] "conses" 3 onto [4,5] to produce [3,4,5].
  • Lists are monotype lists – all items must be the same type.
  • List types are written "list(int)" rather than "int list".
• Tuples: (true, "my string") has type "(bool, string)" (which in Racket we would have written "bool * string")
Combinatorial Programs
Combinatorial programs

• Used to *count* or *enumerate* things
  • "all triples of ints between 0 and 10 whose sum is no more than 7"
  • "all increasing sequences of numbers between 0 and 20"
  • "The number of ways to partition a regular n-gon into triangles"
  • ...

• Today we're going to look at one in detail:
  • Stars-and-stripes: "Find all strings containing only "*" and "-", and which contain exactly \( n \) stars and \( n \) stripes."

• Input: \( n \), a natural number
• Output: a list of strings
This *isn’t* going to be about code!

- But to illustrate the goals, I'll use a Racket-style design-recipe:
;; ss: num -> (string list)
;; input: an integer n
;; output: a list of all strings made up from exactly
;; n asterisks and n dashes

(ss 0) => empty
(ss 1) => (list "*-*" "-**") [in some order]
(ss 2) => (list "**--" "*-*-" "*-*-" "-**-" "-*-*" "--**") [in some order]
What next?

• Draw a recursive diagram!
OI: 2
RI: 1
RO: *-, -*

? stick a star and a stripe into every possible intermediate position?

OO: **--, *-*-, *--*, -**-, -*-*, --**
Ideas?

• Recursive result doesn't actually seem very helpful
• Have to "put an extra star and an extra stripe" in every possible position
• Ends up producing duplicates, too
• Solution: as we've seen before, we can make the problem harder, so the recursive result can be more useful!
Revised stars-n-stripes problem: more general

• Given n, k, create all strings containing n “*”s and k “-”s, in any order.
• (sns 2 0) => (list “**”)
• (sns 2 1) => (list “-**” “*-*” “**-”)

Recursive diagram

0I: 1 2
RI  0  2
RO  --

idea: stick a star in every possible “slot”

00: *--  -*-  --*
Another diagram

OI: 0 2
  RI 0 1
  RO –
  idea: you can’t stick a stripe in every slot - duplicates!
00: --
Some ideas

• Idea: Go ahead and produce duplicates, but filter them out later.
  • Slow.
• Idea: Maybe there are many base-cases, like 0 0, 0 n, and n 0.
A surprisingly general approach to (beginning) combinatorial problems is this:

- Divide the things you're trying to produce (or count) into two disjoint piles and work on each one
- Disjoint means “Not sharing any items”
- Sometimes we divide into 3 or 4 piles, but 2 is often enough
- Being certain they’re disjoint is essential, and sometimes tricky.
- An application of “divide and conquer”, a general idea in CS
How can we divide these into disjoint groups? (preferably of about equal size)
Divide into piles based on the “starting letter”!

Groups are obviously disjoint.
Continuing with the plan

• Divide the things your trying to produce (or count) into two *disjoint* piles and **work on each one**

• Q: How can we produce these? **-- -** *-* - *--*

• A: By prepending a "*" on each of these: *-- -** *--*

• How can we produce *those*?
  • A *recursive* call!

• **Summary**
  • Make *two* recursive calls, reducing the number of stars in one, the number of stripes in the other
  • Prepend a * on each item in the first result; prepend a - on each item in the second result
  • Append the two resulting lists
• Summary
  • Make two recursive calls, reducing the number of stars in one, the number of stripes in the other
  • Prepend a * on each item in the first result; prepend a – on each item in the second result
  • Append the two resulting lists

• Quiz: what should be the base-case(s) for this recursion (i.e., what inputs n and k, and what outputs)?
• Base cases!

(define (sns n k)
  (cond
   [[(zero? n) (list (replicate "-" k))]
    [[(zero? k) (list (replicate "*" n))]
     [(and (succ? n) (succ? k) (sns-helper n k))]]

(define (sns-helper n k)
  (let
   ([(alos1 (map (lambda (x) (string-append "*" x)) (sns (- n 1) k))
     (alos2 (map (lambda (x) (string-append "-" x)) (sns n (- k 1))))
     (append alos1 alos2))]

(define (replicate str n)
  (cond
   [[(zero? n) ""]
    [(succ? n) (string-append str (replicate str (- n 1)))]))


; sns-helper: int*int -> (str list)
; Inputs:
;   n, a positive int, saying how many stars
;   k, a positive int, saying how many stripes
; output:
;   a complete list (with no duplicates) of all strings containing
;   only * and -, where each string has exactly n stars and k stripes
(define (sns-helper n k)
  (let
    (((alos1 (map (lambda (x) (string-append "*" x)) (sns (- n 1) k)))

    (alos2 (map (lambda (x) (string-append "-" x)) (sns n (- k 1))))))))
  (append alos1 alos2)))

Check-expects here
(define (sns n k)
  (cond
    [(and (zero? n) (zero? k))  empty]
    [(zero? n) (list (repeat "-" k))]
    [(zero? k) (list (repeat "*" n))]
    [(and (succ? n) (succ? k))  (sns-helper n k)])))
It’s still wrong!

• Can you think of a string that contains no characters other than *s and -s, and which contains zero of each of these?
• What should (sns 0 0) be?
  (list "")
• Have to change the check-expect, too!
A more compact solution

(define (sub1 n) (- n 1))
(define (sns num-stars num-stripes)
  (cond
    [(and (zero? num-stars) (zero? num-stripes)) (list "" )]
    [(and (zero? num-stars) (succ? num-stripes))
      (map (lambda (x) (string-append "-" x)) (sns 0 (sub1 num-stripes)))]
    [(and (succ? num-stars) (zero? num-stripes))
      (map (lambda (x) (string-append "*" x)) (sns (sub1 num-stars) 0))]
    [(and (succ? num-stars) (succ? num-stripes))
      (append
        (map (lambda (x) (string-append "*" x)) (sns (sub1 num-stars) num-stripes))
        (map (lambda (x) (string-append "-" x)) (sns num-stars (sub1 num-stripes))))])))
Review

• Small (but big) ideas
  • Your base case is the only one involved in every possible invocation of your procedure; get it right!
  • Making a recursive problem more general gives you the possibility of getting more from your recursive result!
  • Sometimes there are multiple base cases
Return to Reason
• Reason so far:
• Basic types: int, float, string, bool, list(int), list('a), function types [more to come]
• Builtins: +, etc.; +., etc., float_of_int (and int_of_float, and string_of_bool and ...), &&, ||, ! (boolean and, or, not)
• "let" instead of (define...); each "let" opens a new environment.
• Arithmetic is "infix"; other functions are applied by writing f(x) or f(x,y)
• Type-ascription: (x:int) means "x is an expression whose value has type int"
• Lists: [], [1,2,3], [3, ... [4, 5]] "conses" 3 onto [4,5] to produce [3,4,5].
  • Lists are monotype lists – all items must be the same time.
  • List types are written "list(int)" rather than "int list".
• Tuples: (true, "my string") has type "(bool, string)" (which in Racket we would have written "bool * string")
File inclusion

• Doesn't happen in ReasonML

• A "build" process figures out which files to use; you just tell it what things you need.

• Example:

• open List;

• Once you've done this, List.rev becomes available to you by just writing "rev"

• If you don't open a module, Reason's build-system will still find it (usually), but you'll need to use "qualified names", as in

List.rev([1,2,3]);
Applying the OCaml we've seen
Bignum solution (in ocaml)

[omitted]
A bit more OCaml
Contains17?

• Remember this proc?

(define (contains17? aloi)
 (cond
   [(empty? aloi) false]
   [(cons? aloi) (or (= 17 (first aloi))
                   (contains17? (rest aloi))))])
A builtin type for handling success/failure cases!

type ‘a option = Some of ‘a | None

• Intended use:
  • “None” means something like “I didn’t find an answer” or “your item isn’t in the data”, etc.
  • “Some x” means “I found an answer, and it was x”

• Requires new type-signature for most functions

f: int -> int becomes
f: int -> int option

For recursive procs, adds slight complexity