

; enumeration

```
(define (enumerate-interval low high)
  (if (> low high)
      '()
      (cons low (enumerate-interval (+ low 1) high))))
```

No need to discuss!

```
(define (enumerate-tree tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                        (enumerate-tree (cdr tree))))))
```

; previously called this
fringe

Let's look at 2 versions of postcondition for
enumerate-tree. The first:

; post₁: returns list of the leaves of tree.
(no mention of the order of these)

; post₂: returns the list of the leaves of tree, in
the same order as they occur in tree.

I want to look at proofs appropriate for each version

Both are tree recursions, so we know we need to carry
out tree inductions.

IH₁: the recursive calls each return a list of leaves,
one of the leaves in (car tree) and the
other of the leaves in (cdr tree).

IS₁: given two lists, we see that append is the right combiner, since $(\text{append } l_1 \ l_2)$ is just the concatenation of l_1 and l_2 .

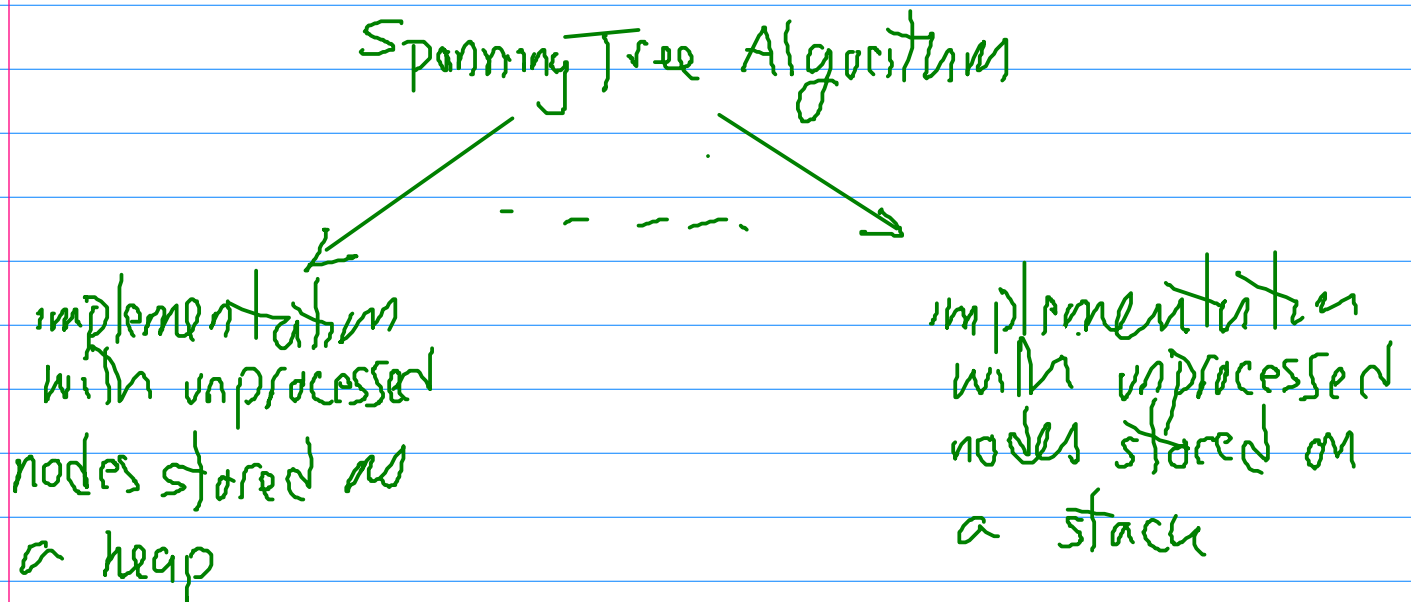
But the program as written actually satisfies the stronger specification given with Post_2 : the reason for this, of course, is that append preserves the element-wise order of its arguments.

So the question is: can the program be made more efficient (or more clean) if we used a combiner other than append?

For example, if we require that a tree not have duplicate leaves, so that the fringe is a set instead of a multiset, perhaps some version of set-union could be more efficient than append. Or, without making this restriction, perhaps multiset union could be useful.

The point I am making is that the program is overspecified relative to post_1 - suggesting that one wants to consider alternatives.

This kind of thing happens frequently — consider, for example, the gap between many graph algorithms and their implementations



One could say that when the algorithm states, at a high level of abstraction

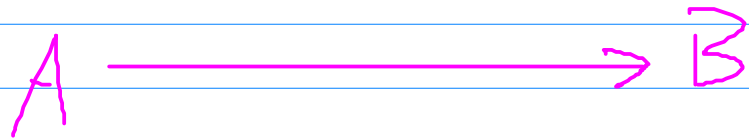
"select an unprocessed node"

that it actually doesn't care how this selection is done. That the algorithm is non-deterministic.

Non-determinism is actually used by all of us in thinking — at a high level — about computational processes. Consider, for example, the process

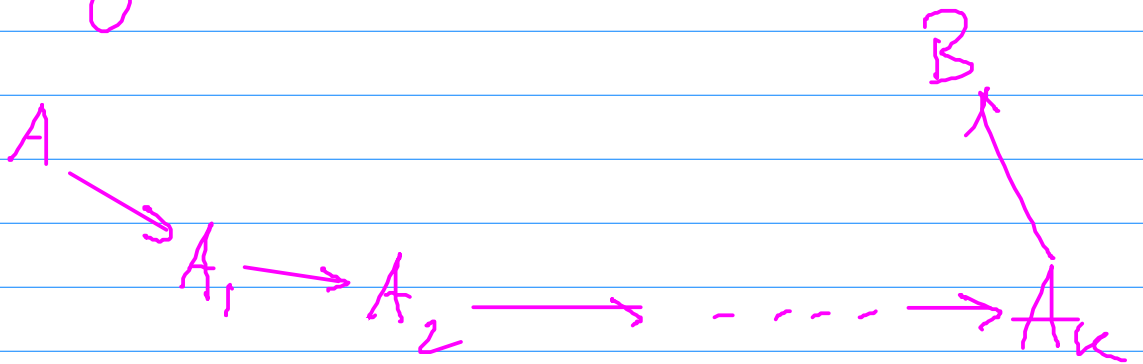
of sending email from point A to point B.

High-level view

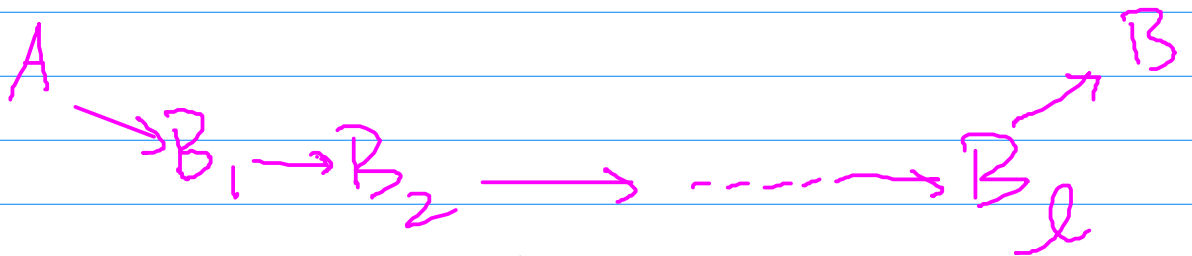


In reality, we have no idea (without using your favorite net utilities) what hops the message makes

on a day
with light
loading of
the net



But maybe



an entirely different set at another time.

Yet the behavior — at the top level — appears deterministic.

The convenience of This kind of abstraction is evident — and very much like the kind we seek as functional programmers.

So: are non-deterministic programming languages a thing? YES — eg, see (part of)

Ch 4. in A&S, where they build an interpreter for one. See also the A&S discussion of amb, which was used by McCarthy (the inventor of LISP) for non-deterministic programming. As the next example suggests, what appears non-det. at one level must be deterministic at a lower level. So amb is actually nothing more than a one-word trigger for BFS. You'll see some amazingly elegant solutions to puzzles, using amb, in A&S —

; signal processing approach to some problems

```
(define tree-2 (list 1 (list (list 2 3) (list 4 5)) (list (list 6))))
```

see the morning notes →

```
(define (sum-odd-square tree)
  (accumulate +
    0
    (map square
      (filter odd?
        (enumerate-tree tree))))))
```

```
(define (fib n)
  (define (aux curr prev count)
    (if (= count 0)
        prev
        (aux (+ curr prev) curr (- count 1))))
  (aux 1 0 n))
```

out-fibs row

```
(define (even-fibs n)
  (accumulate cons
    '()
    (filter even?
      (map fib
        (enumerate-interval 0 n)))))
```

$(0 \ 1 \ 2 \ \dots \ n) \xrightarrow[\text{fib}]{\text{map by}} (0 \ 1 \ 2 \ \dots \ \text{nth-fib-}\#)$
filter by even

$(0 \ 2 \ \dots \ \text{just even fibs out to } n)$
accumulate cons ??

$(0 \ 2 \ \dots \ \text{list is unchanged})$

This is the returned value
proving that the
program returns the
list consisting of
precisely the even
fib #s up to
fib(n).

Why is this a proof? There is
no recursion or iteration at the top
level — in fact, we have already

proved the correctness of map,
filter, accumulate → SO THERE
IS NO NEED TO DO IT AGAIN!

All one needs to do is to describe the data
flow (after checking that the composed functions
are compatible), perhaps with a diagram
(as I have done).

ie, nested for-loops in Scheme

; nested mappings

; given a positive integer n, find all ordered pairs of distinct positive integers i and j,
; where $1 \leq i < j \leq n$

(define (ordered-pairs-of-distinct-integers n)

(accumulate append

'()

(map (lambda (i) (map (lambda (j) (list i j))

(enumerate-interval (+ i 1) n)))

(enumerate-interval 1 (- n 1))))

call this
(f i)

Realize that — once again — there is no top-level recursion or looping. So our interest is directed at the data flow, as for the previous example.

so that the code appears

(map (lambda (i) (f i))

(enumerate-interval 1 (- n 1)))

It's easy to see that the flow is

$(1\ 2 \dots n-1) \mapsto ((f\ 1)\ (f\ 2) \dots (f\ (-n\ 1)))$

Now we look at $(f\ i)$ — eg

$(f\ 1) = ((1\ 2)\ (1\ 3) \dots (1\ n))$

$(f\ 2) = ((2\ 3)\ (2\ 4) \dots (2\ n))$

and so on. So when this completes,

you have $(f\ i)$ replaced by
 $((i\ i+1)\ (i\ i+2)\ \dots\ (i\ n))$

and thus the intermediate output is
a list of lists of pairs

$$\begin{aligned} &((1\ 2)\ \dots\ (1\ n)) \\ &\quad ((2\ 3)\ \dots\ (2\ n)) \\ &\quad \vdots \\ &\quad ((n-1\ n)) \end{aligned}$$

Problem arises for a list of pairs, not a list
of lists of pairs — so we flatten
using the usual accumulate — append
combination.

; flatmap allows a simplification

```
(define (flatmap proc seq)
  (accumulate append '() (map proc seq)))
```

```
(define (ordered-pairs-of-distinct-integers n)
  (flatmap (lambda (i)
    (map (lambda (j) (list i j))
      (enumerate-interval (+ i 1) n))))
  (enumerate-interval 1 (- n 1))))
```

; flatmap turns out to be quite useful - next we use it to compute
; all permutations of a set S

; for example, the permutations of {1,2,3} are given --
; first, list all permutations with 1 in the first position
; next, list all permutations with 2 in the first position
; finally, list all permutations with 3 in the first position

```
(define (permutations s)
  (if (null? s)
    (list '())
    (flatmap
      (lambda (x)
        (map (lambda (p) (cons x p))
          (permutations (remove x s))))
      s)))
```

; where

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item)))
    sequence))
```