Iteration via Tail Recursion in Racket

SOLUTIONS

CS251 Programming Languages
Spring 2019, Lyn Turbak

Department of Computer Science
Wellesley College

Overview

- What is iteration?
- Racket has no loops, and yet can express iteration. How can that be?
  - Tail recursion!
- Tail recursive list processing via `foldl`
- Other useful abstractions
  - General iteration via `iterate` and `iterate-apply`
  - General iteration via `genlist` and `genlist-apply`

Factorial Revisited

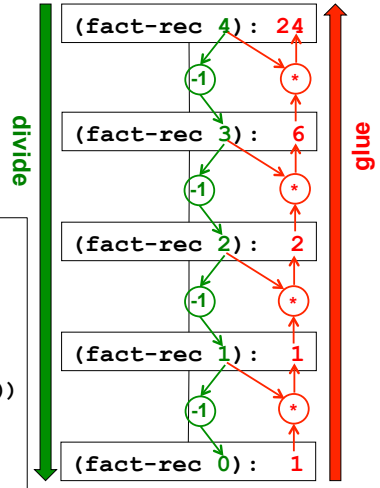
```
(define (fact-rec n)
  (if (= n 0)
      1
      (* n (fact-rec (- n 1)))))
```

pending multiplication is nontrivial glue step

Small-Step Semantics

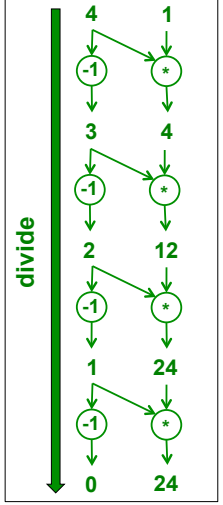
```
({fact-rec} 4)
⇒ {(λ_fact-rec 4)}
⇒* (* 4 {(λ_fact-rec 3)})
⇒* (* 4 (* 3 {(λ_fact-rec 2)}))
⇒* (* 4 (* 3 (* 2 {(λ_fact-rec 1)})))
⇒* (* 4 (* 3 (* 2 (* 1 {(λ_fact-rec 0)}))))
⇒* (* 4 (* 3 (* 2 {(* 1 1)})))
⇒ (* 4 (* 3 {(* 2 1)}))
⇒ (* 4 {(* 3 2)})
⇒ {(* 4 6)}
⇒ 24
```

Invocation Tree



An iterative approach to factorial

Idea: multiply on way down



State Variables:

- **`num` is the current number being processed.**
- **`prod` is the product of all numbers already processed.**

Iteration Table:

step	`num`	`prod`
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

Iteration Rules:

- **next `num` is previous `num` minus 1.**
- **next `prod` is previous `num` times previous `prod`.**

Iterative factorial: tail recursive version in Racket

State Variables:
- `num` is the current number being processed.
- `prod` is the product of all numbers already processed.

```
(define (fact-tail  num        prod     )
  (if (= num 0)
      prod
      (fact-tail (- num 1) (* num prod))))
```

stopping condition

tail call (no pending operations) expresses iteration rules

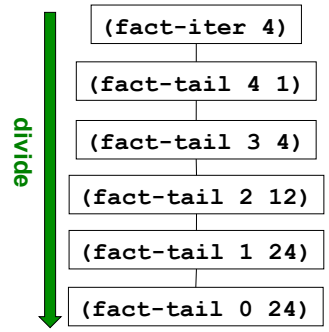
Iteration Rules:
- next `num` is previous `num` minus 1.
- next `prod` is previous `num` times previous `prod`.

```
;; Here, and in many tail recursions, need a wrapper
;; function to initialize first row of iteration
;; table. E.g., invoke (fact-iter 4) to calculate 4!
(define (fact-iter n)
  (fact-tail n 1))
```


Tail-recursive factorial: Dynamic execution

```
(define (fact-iter n)
  (fact-tail n 1))

(define (fact-tail num prod)
  (if (= num 0)
      prod
      (fact-tail (- num 1) (* num prod))))
```

Invocation Tree



no glue!

Small-Step Semantics

```
({fact-iter} 4)
⇒ {(λ_fact-iter 4)}
⇒ {(λ_fact-tail 4 1)}
⇒* {(λ_fact-tail 3 4)}
⇒* {(λ_fact-tail 2 12)}
⇒* {(λ_fact-tail 1 24)}
⇒* {(λ_fact-tail 0 24)}
⇒* 24
```

Iteration Table

step	num	prod
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

The essence of iteration in Racket

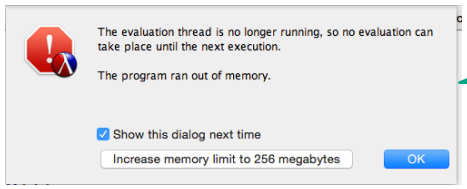
- A process is iterative if it can be expressed as a sequence of steps that is repeated until some stopping condition is reached.
- In divide/conquer/glue methodology, an iterative process is a recursive process with a single subproblem and no glue step.
- Each recursive method call is a tail call -- i.e., a method call with no pending operations after the call. When all recursive calls of a method are tail calls, it is said to be tail recursive. A tail recursive method is one way to specify an iterative process.

Iteration is so common that most programming languages provide special constructs for specifying it, known as loops.

`inc-rec` in Racket

```
; Extremely silly and inefficient recursive incrementing
; function for testing Racket stack memory limits
(define (inc-rec n)
  (if (= n 0)
      1
      (+ 1 (inc-rec (- n 1)))))
```

```
> (inc-rec 1000000) ; 10^6
1000001

> (inc-rec 10000000) ; 10^7
```



Eventually run out of stack space

inc_rec in Python

```
def inc_rec (n):
  if n == 0:
    return 1
  else:
    return 1 + inc_rec(n - 1)
```

```
In [16]: inc_rec(100)
Out[16]: 101

In [17]: inc_rec(1000)
...
in inc_rec(n)
    9       return 1
   10     else:
---> 11        return 1 + inc_rec(n – 1)

RuntimeError: maximum recursion depth exceeded
```

Very small maximum recursion depth (implementation dependent)

inc-iter/inc-tail in Racket Solutions

```
(define (inc-iter n)
  (inc-tail n 1))

(define (inc-tail num resultSoFar)
  (if (= num 0)
      resultSoFar
      (inc-tail (- num 1) (+ resultSoFar 1))))
```

```
> (inc-iter 10000000) ; 10^7
10000001

> (inc-iter 100000000) ; 10^8
100000001
```

Will `inc-iter` ever run out of memory?

It will not run out of stack memory. The only memory that matters here is the memory to represent large numbers.

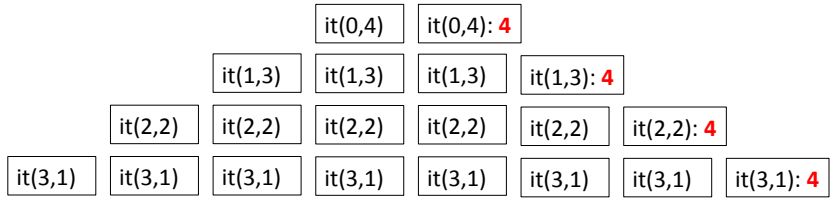
inc_iter/int_tail in Python

```
def inc_iter (n): # Not really iterative!
  return inc_tail(n, 1)

def inc_tail(num, resultSoFar):
  if num == 0:
    return resultSoFar
  else:
    return inc_tail(num - 1, resultSoFar + 1)
```

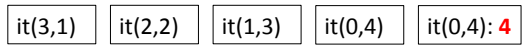
```
In [19]: inc_iter(100)
Out[19]: 101

In [19]: inc_iter(1000)
...
RuntimeError: maximum recursion depth exceeded
```

Although tail recursion expresses iteration in Racket (and SML), it does *not* express iteration in Python (or JavaScript, C, Java, etc.)

Why the Difference?



Python pushes a stack frame for every call to iter_tail. When iter_tail(0,4) returns the answer 4, the stacked frames must be popped even though no other work remains to be done coming out of the recursion.



Racket's *tail-call optimization* replaces the current stack frame with a new stack frame when a *tail call* (function call not in a subexpression position) is made. When iter-tail(0,4) returns 4, no unnecessarily stacked frames need to be popped!

Origins of Tail Recursion

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 443 October 1977

DEBUNKING THE "EXPENSIVE PROCEDURE CALL" MYTH

or, PROCEDURE CALL IMPLEMENTATIONS CONSIDERED HARMFUL

or, LAMBDA: THE ULTIMATE GOTO

by

Guy Lewis Steele Jr. *

Guy Lewis Steele
a.k.a. ``The Great Quux"

- One of the most important but least appreciated CS papers of all time
- Treat a function call as a GOTO that passes arguments
- Function calls should not push stack; subexpression evaluation should!
- Looping constructs are unnecessary; tail recursive calls are a more general and elegant way to express iteration.

What to do in Python (and most other languages)?

In Python, **must** re-express the tail recursion as a loop!

```
def inc_loop (n):
  resultSoFar = 0
  while n > 0:
    n = n - 1
    resultSoFar = resultSoFar + 1
  return resultSoFar
```

```
In [23]: inc_loop(1000) # 10^3
Out[23]: 1001

In [24]: inc_loop(10000000) # 10^8
Out[24]: 10000001
```

But Racket doesn't need loop constructs because tail recursion suffices for expressing iteration!

Iterative factorial: Python **`while`** loop version

Iteration Rules:
- next num is previous num minus 1.
- next prod is previous num times previous prod.

```
def fact_while(n):

  num = n
  prod = 1

  while (num > 0):
    prod = num * prod
    num = num - 1

  return prod
```

Declare/initialize local state variables

Calculate product and decrement num

Don't forget to return answer!

`while` loop factorial: Execution Land

Execution frame for fact_while(4)

```
num = n
prod = 1

while (num > 0):
  prod = num * prod
  num = num - 1

return prod
```

n	num	prod
4	~~4~~	~~1~~
	~~3~~	~~4~~
	~~2~~	~~12~~
	~~1~~	~~24~~
	0	24

step	num	prod
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

Gotcha! Order of assignments in loop body

What's wrong with the following loop version of factorial?

```
def fact_while(n):
  num = n
  prod = 1
  while (num > 0):
    num = num - 1
    prod = num * prod
  return prod
```

```
In [23]: fact_while(4)
Out[23]: 6
```

Moral: must think carefully about order of assignments in loop body!

Note: tail recursion doesn't have this gotcha!

```
(define (fact-tail  num        prod     )
  (if (= num 0)
      ans                  ⬇        ⬇
      (fact-tail (- num 1) (* num prod))))
```


Relating Tail Recursion and while loops

```
(define (fact-iter n)
  (fact-tail n 1))

(define (fact-tail num prod)
  (if (= num 0)
      prod
      (fact-tail (- num 1) (* num prod))))
```

Initialize variables

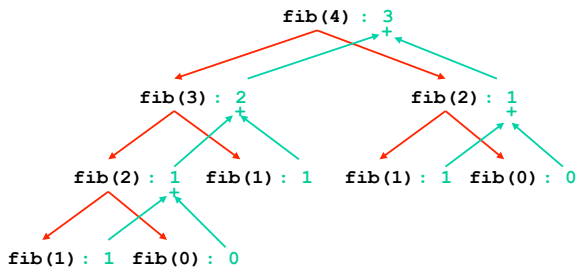
When done, return ans

```
def fact_while(n):
  num = n
  prod = 1
  while (num > 0):
    prod = num * prod
    num = num - 1
  return prod
```

While not done, update variables

Recursive Fibonacci

```
(define (fib-rec n) ; returns rabbit pairs at month n
  (if (< n 2) ; assume n >= 0
      n
      (+ (fib-rec (- n 1)) ; pairs alive last month
         (fib-rec (- n 2)) ; newborn pairs
         )))
```



Iteration leads to a more efficient Fib

The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

Iteration table for calculating the 8th Fibonacci number:

n	i	fibi	fibi+1
8	0	0	1
8	1	1	1
8	2	1	2
8	3	2	3
8	4	3	5
8	5	5	8
8	6	8	13
8	7	13	21
8	8	21	34

Iterative Fibonacci in Racket Solutions

Flesh out the missing parts

```
(define (fib-iter n)
  (fib-tail n 0 0 1)                )

(define (fib-tail n i fibi fibi+1)
  (if (= i n)
      fibi
      (fib-tail n
                (+ i 1)
                fibi+1
                (+ fibi fibi+1)))
  )
```


Gotcha! Assignment order and temporary variables

What's wrong with the following looping versions of Fibonacci?

```
def fib_for1(n):
  fib_i= 0
  fib_i_plus_1 = 1
  for i in range(n):
    fib_i = fib_i_plus_1
    fib_i_plus_1 = fib_i + fib_i_plus_1
  return fib_i
```

wrong fib_i

```
def fib_for2(n):
  fib_i= 0
  fib_i_plus_1 = 1
  for i in range(n):
    fib_i_plus_1 = fib_i + fib_i_plus_1
    fib_i = fib_i_plus_1
  return fib_i
```

wrong fib_i_plus_1

Moral: sometimes no order of assignments to state variables in a loop is correct and it is necessary to introduce one or more temporary variables to save the previous value of a variable for use in the right-hand side of a later assignment.

Or can use **simultaneous assignment** in languages that have it (like Python!)

Fixing Gotcha

1. Use a temporary variable (in general, might need n-1 such vars for n state variables

```
def fib_for_fixed1(n):
  fib_i= 0
  fib_i_plus_1 = 1
  for i in range(n):
    fib_i_prev = fib_i
    fib_i = fib_i_plus_1
    fib_i_plus_1 = fib_i_prev + fib_i_plus_1
  return fib_i
```

2. Use simultaneous assignment:

```
def fib_for_fixed2(n):
  fib_i= 0
  fib_i_plus_1 = 1
  for i in range(n):
    (fib_i, fib_i_plus_1) =\
       (fib_i_plus_1, fib_i + fib_i_plus_1)
return fib_i
```

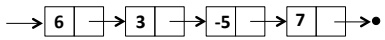

Local `fib-tail` function in `fib-iter`

Can define `fib-tail` locally within `fib-iter`.

Since `n` remains constant, don't need it as an argument to local fib-tail.

```
(define (fib-iter n)
  (define (fib-tail i fibi fibi+1)
    (if (= i n)
        fibi
        (fib-tail (+ i 1)
                  fibi+1
                  (+ fibi fibi+1))))
  (fib-tail 0 0 1)
  )
```

Iterative List Summation

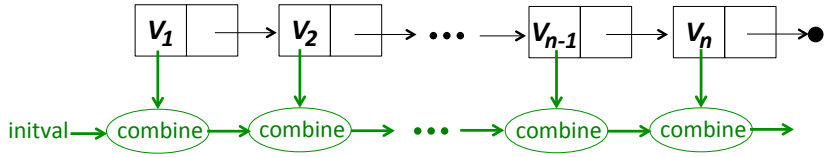


Iteration table

nums	sumSoFar
'(6 3 -5 7)	0
'(3 -5 7)	6
'(-5 7)	9
'(7)	4
'()	11

```
(define (sumList-iter L)
  (sumList-tail     L     0    ))

(define (sumList-tail nums sumSoFar)
  (if (null? nums)
      sumSoFar
      (sumList-tail (rest nums)
                    (+ (first nums) sumSoFar))))
```

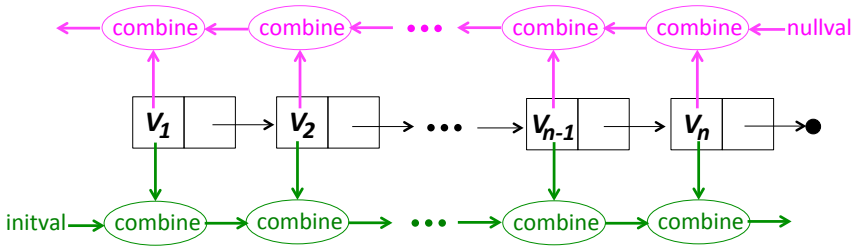

Capturing list iteration via `my-foldl`



(initval is the initial resultSoFar)

```
(define (my-foldl combine resultSoFar xs)
  (if (null? xs)
      resultSoFar
      (my-foldl combiner
                (combine (first xs) resultSoFar)
                (rest xs))))
```


`foldr` vs `foldl`



`my-foldl` Examples Solutions


```
> (my-foldl + 0 (list 7 2 4))
13

> (my-foldl * 1 (list 7 2 4))
56

> (my-foldl - 0 (list 7 2 4))
9; (- 4 (- 2 (- 7 0)))

> (my-foldl cons null (list 7 2 4))
'(4 2 7); (cons 4 (cons 2 (cons 7 0)))

> (my-foldl (λ (n res) (+ (* 3 res) n))
            0
            (list 10 -4 5 2))
251 ; = 10*3^3 + -4*3^2 + 5*3^1 + 2*3^0
    ; An example of Horner's method
    ; for polynomial evaluation of 10x^3 -4x^2 + 5x + 2
```

Built-in Racket `foldl` Function Folds over Any Number of Lists

```
> (foldl cons null (list 7 2 4))
'(4 2 7)

> (foldl (λ (a b res) (+ (* a b) res))
         0
         (list 2 3 4)
         (list 5 6 7))
56

> (foldl (λ (a b res) (+ (* a b) res))
         0
         (list 1 2 3 4)
         (list 5 6 7))
> ERROR: foldl: given list does not have the same
size as the first list: '(5 6 7)
```

Same design decision as in map and foldr

Iterative vs Recursive List Reversal Solutions

```
(define (reverse-iter xs)
  (foldl cons null xs))

(define (snoc x ys)
  (foldr cons (list x) ys))

(define (reverse-rec xs)
  (foldr snoc null xs))
```

How do these compare in terms of the number of conses performed for a list of length 100? 1000? n?

How about stack depth?

Ans:

- **`reverse-iter`: exactly n conses, none pending; O(1) stack space.**
- **`snoc`: exactly n+1 conses, all pending; O(n) stack space**
- **`reverse-rec`: quadratic (O(n^2)) conses, all pending; O(n^2) stack space**

What does this do? Solutions


```
(define (whatisit f xs)
  (foldl (λ (x listSoFar)
           (cons (f x) listSoFar))
         null
         xs)))
```

Ans: It performs the ``reverse map" of function `f` on list `xs`.

E.g., `(whatisit (λ (n) (* n 3)) '(7 2 4)) => '(12 6 21)`

To perform a regular map, change `foldl` to `foldr`!

Tail Recursion Review 1 Solutions


```
# Euclid's algorithm
def gcd(a,b):
    while b != 0:
        temp = b
        b = a % b
        a = temp
    return a
```

1. Create an iteration table for `gcd(42,72)`
2. Translate Python `gcd` into Racket tail recursion.

a	b
42	72
72	42
42	30
30	12
12	6
6	0

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Tail Recursion Review 2 Solutions


```
def toInt(digits):
    i = 0
    for d in digits:
        num = 10*i + d
    return i
```

1. Create an iteration table for `toInt([1,7,2,9])`
2. Translate Python `toInt` into Racket tail recursion.
3. Translate Python `toInt` into Racket `foldl`.

digits	i
[1,7,2,9]	0
[7,2,9]	1
[2,9]	17
[9]	172
[]	1729

```
(define (toInt digits)
  (define (toIntTail ds i)
    (if (null? ds)
        i
        (toIntTail (rest ds) (+ (* 10 i) d))))
  (toIntTail digits 0))
```

```
(define (toInt digits)
  (foldl (λ (d i) (+ (* 10 i) d))
         0
         digits))
```


iterate

```
(define (iterate next done? finalize state)
  (if (done? state)
      (finalize state)
      (iterate next done? finalize
               (next state))))
```

step	num	prod
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

For example:

```
(define (fact-iterate n)
  (iterate
    (λ (num&prod)
      (list (- (first num&prod) 1)
            (* (first num&prod)
               (second num&prod))))
    (λ (num&prod) (<= (first num&prod) 0))
    (λ (num&prod) (second num&prod))
    (list n 1)))
```

step	num&prod
1	'(4 1)
2	'(3 4)
3	'(2 12)
4	'(1 24)
5	'(0 24)

least-power-geq


```
; Soln 1
(define (least-power-geq base threshold)
  (iterate (λ (pow) (* base pow))
           (λ (pow) (>= pow threshold))
           (λ (pow) pow)
           1)) ; Initial power

; Soln 2
(define (least-power-geq base threshold)
  (iterate (λ (exp) (+ exp 1))
           (λ (exp) (>= (expt base exp) threshold))
           (λ (exp) (expt base exp)) ; To return just exponent,
                                     ; use exp here.
           0)) ; Initial exponent
```

```
> (least-power-geq 2 10)
16
> (least-power-geq 5 100)
125
> (least-power-geq 3 100)
243
```

In Soln 2, just return exp rather than (expt base exp)

In Soln 1, change state to list of (1) power and (2) exponent. exponent is initialized to 0 and is incremented at each step. In finalization step, return exponent.

How could we return just the exponent rather than the base raised to the exponent?

What do These Do?


```
(define (mystery1 n) ; Assume n >= 0
  (iterate (λ (ns) (cons (- (first ns) 1) ns))
           (λ (ns) (<= (first ns) 0))
           (λ (ns) ns)
           (list n)))

(define (mystery2 n)
  (iterate (λ (ns) (cons (quotient (first ns) 2) ns))
           (λ (ns) (<= (first ns) 1))
           (λ (ns) (- (length ns) 1))
           (list n)))
```

mystery1 returns the list of ints from 0 up to and including n.
E.g., (mystery1 5) => '(0 1 2 3 4 5)

mystery2 calculates the log-base-2 of n by determining how many times n can be divided by 2 before reaching 1. E.g.
(mystery2 32) => 5

Using `let` to introduce local names

```
(define (fact-let n)
  (iterate (λ (num&prod)
             (let ([num (first num&prod)]
                   [prod (second num&prod)])
               (list (- num 1) (* num prod))))
           (λ (num&prod) (<= (first num&prod) 0))
           (λ (num&prod) (second num&prod))
           (list n 1)))
```


Using `match` to introduce local names

```
(define (fact-match n)
  (iterate (λ (num&prod)
             (match num&prod
               [(list num prod)
                (list (- num 1) (* num prod))]))
           (λ (num&prod)
            (match num&prod
              [(list num prod) (<= num 0)]))
           (λ (num&prod)
             (match num&prod
               [(list num prod) prod]))
           (list n 1)))
```


Racket's `apply`

```
(define (avg a b)
  (/ (+ a b) 2))
```

```
> (avg 6 10)
8

> (apply avg '(6 10))
8

> ((λ (a b c) (+ (* a b) c)) 2 3 4)
10

> (apply (λ (a b c) (+ (* a b) c)) (list 2 3 4))
10
```

`apply` takes (1) a function and (2) a single argument that is a **list of values** and returns the result of applying the function to the values.

`iterate-apply:` a kinder, gentler `iterate`

```
(define (iterate-apply next done? finalize state)
  (if (apply done? state)
      (apply finalize state)
      (iterate-apply next done? finalize
                     (apply next state))))
```

```
(define (fact-iterate-apply n)
  (iterate-apply
    (λ (num prod)
      (list (- num 1) (* num prod)))
    (λ (num prod) (<= num 0))
    (λ (num prod) prod)
    (list n 1)))
```

step	num	prod
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

iterate-apply: fib and gcd Solutions

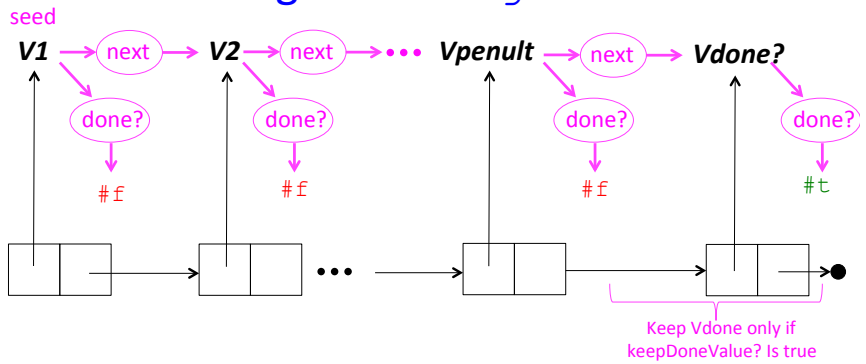

```
(define (fib-iterate-apply n)
  (iterate-apply
    (λ (i fibi fibi+1)                ; next
      (list (+ i 1) fibi+1 (+ fibi fibi+1))
    (λ (i fibi fibi+1) (= i n))       ; done?

    (λ (i fibi fibi+1) fib_i)         ; finalize

    (list 0 0 1)                      ; init state
    ))
```

n	i	fibi	fibi+1
8	0	0	1
8	1	1	1
8	2	1	2
8	3	2	3
8	4	3	5
8	5	5	8
8	6	8	13
8	7	13	21
8	8	21	34

```
(define (gcd-iterate-apply a b)
  (iterate-apply
    (lambda (a b)                     ; next
      (list b (remainder a b))
    (lambda (a b) (= b 0)             ; done?
    (lambda (a b) a)                  ; finalize
    (list a b)))                      ; init state
    ))
```

a	b
42	72
72	42
42	30
30	12
12	6
6	0

Creating lists with genlist




```
(define (genlist next done? keepDoneValue? seed)
  (if (done? seed)
      (if keepDoneValue? (list seed) null)
      (cons seed
            (genlist next done? keepDoneValue? (next seed)))))
```

not iterative as written, but `next` function gives iterative ``flavor''

Simple genlist examples Solutions

What are the values of the following calls to `genlist`?

```
(genlist (λ (n) (- n 1))
         (λ (n) (= n 0))
         #t
         5)
```

'(5 4 3 2 1 0)

```
(genlist (λ (n) (- n 1))
         (λ (n) (= n 0))
         #f
         5)
```

'(5 4 3 2 1)

```
(genlist (λ (n) (* n 2))
         (λ (n) (> n 100))
         #t
         1)
```

'(1 2 4 8 16 32 64 128)

```
(genlist (λ (n) (* n 2))
         (λ (n) (> n 100))
         #f
         1)
```

'(1 2 4 8 16 32 64)

genlist: my-range and halves Solutions


```
(my-range lo hi)

> (my-range 10 15)
'(10 11 12 13 14)

> (my-range 20 10)
'()
```

```
(define (my-range-genlist lo hi)
  (genlist
    (λ (n) (+ n 1))              ; next
    (λ (n) (>= n hi))            ; done?
    #f                  ; keepDoneValue?
    lo                           ; seed
    ))
```

```
(halves num)

> (halves 64)
'(64 32 16 8 4 2 1)

> (halves 42)
'(42 21 10 5 2 1)

> (halves -63)
'(-63 -31 -15 -7 -3 -1)
```

```
(define (halves num)
  (genlist
    (λ (n) (quotient n 2))       ; next
    (λ (n) (= n 0))              ; done?
    #f                  ; keepDoneValue?
    num                          ; seed
    ))
```

Using genlist to generate iteration tables

```
(define (fact-table n)
  (genlist (λ (num&prod)
             (let ((num (first num&ans))
                   (prod (second num&ans)))
               (list (- num 1) (* num prod))))
           (λ (num&prod) (<= (first num&prod) 0))
           #t
           (list n 1)))
```

step	num	prod
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

```
> (fact-table 4)
'((4 1) (3 4) (2 12) (1 24) (0 24))
> (fact-table 5)
'((5 1) (4 5) (3 20) (2 60) (1 120) (0 120))
```

```
> (fact-table 10)
'((10 1)
  (9 10)
  (8 90)
  (7 720)
  (6 5040)
  (5 30240)
  (4 151200)
  (3 604800)
  (2 1814400)
  (1 3628800)
  (0 3628800))
```


Your turn: sum-list iteration table Solutions


```
(define (sum-list-table ns)
  (genlist
    (λ (nums&sum)                                  ; next
      (let {[nums (first nums&ans)]
            [sum (second nums&ans)]}
        (list (rest nums)
              (+ sum (first nums)))))
    (λ (nums&sum)                                  ; done?
      (null? (first nums&sum)))
    #t                                             ; keepDoneValue?
    (list ns 0))                                   ; seed
    )
```

```
> (sum-list-table '(7 2 5 8 4))
'(((7 2 5 8 4) 0)
  ((2 5 8 4) 7)
  ((5 8 4) 9)
  ((8 4) 14)
  ((4) 22)
  (() 26))
```


`genlist` can collect iteration table column!

```
; With table abstraction
(define (partial-sums ns)
  (map second (sum-list-table ns)))
```

```
; Without table abstraction
(define (partial-sums ns)
  (map second
       (genlist (λ (nums&sum)
                  (let ((nums (first nums&ans))
                        (sum (second nums&ans)))
                    (list (rest nums) (+ (first nums) sum))))
                (λ (nums&sum) (null? (first nums&sum)))
                #t
                (list ns 0))))
```

```
> (partial-sums '(7 2 5 8 4))
'(0 7 9 14 22 26)
```

Moral: ask yourself the question
"Can I generate this list as the column of an iteration table? "

`genlist-apply`: a kinder, gentler `genlist`

```
(define (genlist-apply next done? keepDoneValue? seed)
  (if (apply done? seed)
      (if keepDoneValue? (list seed) null)
      (cons seed
            (genlist-apply next done? keepDoneValue?
                           (apply next seed)))))
```

Example:

```
(define (partial-sums ns)
  (map second
       (genlist-apply
         (λ (nums ans)
           (list (rest nums) (+ (first nums) ans)))
         (λ (nums ans) (null? nums))
         #t
         (list ns 0))))
```

partial-sums-between Solutions


```
(define (partial-sums-between lo hi)
  (map second
    (genlist-apply
      (λ (num sum)                          ; next
        (list (+ num 1) (+ num sum)))
      (λ (num sum)                          ; done?
        (> num hi))
      #t                          ; keepDoneValue?
      (list lo 0)                           ; seed
      )))
```

```
> (partial-sums-between 3 7)
'(0 3 7 12 18 25)

> (partial-sums-between 1 10)
'(0 1 3 6 10 15 21 28 36 45 55)
```


Iterative Version of genlist

```
;; Returns the same list as genlist, but requires only
;; constant stack depth (*not* proportional to list length)
(define (genlist-iter next done? keepDoneValue? seed)
  (iterate-apply
   (λ (state reversedStatesSoFar)
     (list (next state)
           (cons state reversedStatesSoFar)))
   (λ (state reversedStatesSoFar) (done? state))
   (λ (state reversedStatesSoFar)
     (if keepDoneValue?
         (reverse (cons state reversedStatesSoFar))
         (reverse reversedStatesSoFar)))
   (list seed '())))
```

Example: How does this work?

```
(genlist-iter (λ (n) (quotient n 2))
              (λ (n) (<= n 0))
              5)
```


Iterative Version of genlist-apply

```
(define (genlist-apply-iter next done? keepDoneValue? seed)
  (iterate-apply
   (λ (state reversedStatesSoFar)
     (list (apply next state)
           (cons state reversedStatesSoFar)))
   (λ (state reversedStatesSoFar) (apply done? state))
   (λ (state reversedStatesSoFar)
     (if keepDoneValue?
         (reverse (cons state reversedStatesSoFar))
         (reverse reversedStatesSoFar)))
   (list seed '())))
```