Iteration via Tail Recursion in Racket

- 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)))))

Invocation Tree

pending multiplication is nontrivial glue step

divide glucose

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:

<table>
<thead>
<tr>
<th>step</th>
<th>num</th>
<th>prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>24</td>
</tr>
</tbody>
</table>

Iteration Rules:
- next num is previous num minus 1.
- next prod is previous num times previous prod.
Iterative factorial: tail recursive version

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

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

<table>
<thead>
<tr>
<th>step</th>
<th>num</th>
<th>prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>24</td>
</tr>
</tbody>
</table>

Tail-recursive factorial: invocation tree

```
;; 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))
```

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

<table>
<thead>
<tr>
<th>Invocation Tree:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fact-iter 4)</td>
</tr>
<tr>
<td>(fact-tail 4 1)</td>
</tr>
<tr>
<td>(fact-tail 3 4)</td>
</tr>
<tr>
<td>(fact-tail 2 12)</td>
</tr>
<tr>
<td>(fact-tail 1 24)</td>
</tr>
<tr>
<td>(fact-tail 0 24)</td>
</tr>
</tbody>
</table>

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
```

```
Evaluation timed out after 1 second, so evaluation will not continue.
```

The program ran out of memory.

1. Show this dialog next time
2. Increase memory limit to 656 megabytes
3. OK
### inc_rec in Python

```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)
...: /Users/fturbak/Desktop/lyn/courses/cs251-archive/cs251-s16/slides-lyn-s16/racket-tail/iter.py
In
inc_rec(n)
9         return 1
10     else:
---> 11         return 1 + inc_rec(n - 1)
12 # inc_rec(10) => 11
13 # inc_rec(100) => 101

```

RuntimeError: maximum recursion depth exceeded
```

### inc_iter/inc_tail in Racket

```racket
(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?**

### Why the Difference?

- **Python** pushes a stack frame for every call to `inc_tail`. When `inc_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!

---

### inc_iter/int_tail in Python

```python
def inc_iter(n):
    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
```

---

### Why the Difference?

- **Python** pushes a stack frame for every call to `inc_tail`. When `inc_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 441
October 1977

DEBUNKING THE 'EXPENSIVE PROCEDURE CALL' MYTH
or, PROCEDURE CALL IMPLEMENTATIONS CONSIDERED HARMFUL
or, LAMBDA: THE ULTIMATE GOTO
by
Guy Lewis Steele Jr., a.k.a. "The Great Quux"

Guy Lewis Steele

- 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!

```python
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.

```python
def fact_while(n):
    num = n
    prod = 1

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

Don’t forget to return answer!

**while loop factorial: Execution Land**

```
<table>
<thead>
<tr>
<th>step</th>
<th>num</th>
<th>prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>24</td>
</tr>
</tbody>
</table>
```

Declare/initialize local state variables
Calculate product and decrement num
Gotcha! Order of assignments in loop body

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

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

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

Note: tail recursion doesn’t have this gotcha!

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

Relating Tail Recursion and while loops

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

Recursive Fibonacci

```scheme
(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:

<table>
<thead>
<tr>
<th>n</th>
<th>i</th>
<th>fibi</th>
<th>fibi+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>21</td>
<td>34</td>
</tr>
</tbody>
</table>
Iterative Fibonacci in Racket

Flesh out the missing parts

```
(define (fib-iter n) )

(define (fib-tail n i 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
```

```
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
```

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)))
    )
```

```
Iterative List Summation

\[
\begin{array}{c|c|c|c|c|c|c}
\text{nums} & \text{result} \\
\hline
\text{'(6 3 -22 5)} & 0 \\
\text{'(3 -22 5)} & 6 \\
\text{'(-22 5)} & 9 \\
\text{'(5)} & -13 \\
\text{'(0)} & -8 \\
\end{array}
\]

\begin{verbatim}
(define (sumList-iter L)
  (sumList-tail L 0))

(define (sumList-tail nums sumSoFar)
  (if (null? nums)
      sumSoFar
      (sumList-tail (rest nums) (+ (first nums) sumSoFar))))
\end{verbatim}

Capturing list iteration via my-foldl

\[
\begin{array}{c|c|c|c|c|c|c}
\text{initial} & \text{combine} & \text{combine} & \cdots & \text{combine} & \text{combine} & \text{initial} \\
\hline
v_1 & v_2 & \cdots & v_{n-1} & v_n & \text{-} & \text{-} \\
\end{array}
\]

\[
\begin{verbatim}
(define (my-foldl combine resultSoFar xs)
  (if (null? xs)
      resultSoFar
      (my-foldl combiner (combine (first xs) resultSoFar) (rest xs))))
\end{verbatim}

\begin{verbatim}
> (my-foldl + 0 (list 7 2 4))
0
> (my-foldl * 1 (list 7 2 4))
0
> (my-foldl cons null (list 7 2 4))
null
> (my-foldl (λ (n res) (+ (* 3 res) n)) 0 (list 10 -4 5 2))
23
\end{verbatim}

Foldr vs Foldl

my-foldl Examples
**Built-in Racket foldl Function**

Folds over Any Number of Lists

\[ > (\text{foldl} \ cons \ \text{null} \ (\text{list} \ 7 \ 2 \ 4)) \]
\[ '4 \ 2 \ 7 \]

\[ > (\text{foldl} \ (\lambda \ (a \ b \ res) \ (+ \ (* \ a \ b) \ res)) \ 0 \n \ (\text{list} \ 2 \ 3 \ 4) \n \ (\text{list} \ 5 \ 6 \ 7)) \]
\[ 56 \]

\[ > (\text{foldl} \ (\lambda \ (a \ b \ res) \ (+ \ (* \ a \ b) \ res)) \ 0 \n \ (\text{list} \ 1 \ 2 \ 3 \ 4) \n \ (\text{list} \ 5 \ 6 \ 7)) \]

**ERROR:** foldl: given list does not have the same size as the first list: '5 6 7

**Iterative vs Recursive List Reversal**

\[
\begin{align*}
\text{(define (reverse-iter xs)} & \\
& (\text{foldl} \ cons \ \text{null} \ xs))
\end{align*}
\]

\[
\begin{align*}
\text{(define (snoc x ys)} & \\
& (\text{foldr} \ cons \ (\text{list} \ x) \ ys))
\end{align*}
\]

\[
\begin{align*}
\text{(define (reverse-rec xs)} & \\
& (\text{foldr} \ \text{snoc} \ \text{null} \ xs))
\end{align*}
\]

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

**What does this do?**

\[
\begin{align*}
\text{(define (whatisit f xs)} & \\
& (\text{foldl} \ (\lambda \ (x \ \text{listSoFar}) \n \ (\text{cons} \ (f \ x) \ \text{listSoFar})) \n \ \text{null} \n \ \text{xs}))
\end{align*}
\]

**Tail Recursion Review 1**

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

```python
# 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 `toInt([1,7,2,9])`
2. Translate Python `toInt` into Racket tail recursion.
3. Translate Python `toInt` into Racket `foldl`.

```python
def toInt(digits):
    i = 0
    for d in digits:
        i = 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`.

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

(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)))
```

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

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

```
(define (least-power-geq base threshold)
    (iterate                        ; next
        ; done?
        ; finalize
        ; initial state
))
```

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

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)))
```

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

How could we return just the exponent rather than the base raised to the exponent?
Using **let** to introduce local names

```scheme
(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))
    (list n 1)))
```

Using **match** to introduce local names

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

**Racket’s apply**

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

```scheme
> (avg 6 10)
8
```

```scheme
> (apply avg '(6 10))
8
```

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

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

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

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

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

**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.

<table>
<thead>
<tr>
<th>step</th>
<th>num</th>
<th>prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>24</td>
</tr>
</tbody>
</table>
iterate-apply: fib and gcd

```
(define (fib-iterate-apply n)
  (iterate-apply
   (lambda (i fibi fibi+1)
    ; next
    (list (+ i 1) fibi+1 (+ fibi fibi+1)))
   (lambda (i fibi fibi+1) (= i n))
   ; done?
   (lambda (i fibi fibi+1) fibi)
   ; finalize
   (list 0 0 1)); init state)
```

```
(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)
```

Simple genlist examples

What are the values of the following calls to genlist?

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

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

Creating lists with genlist

```
(seed V1 next V2 next Vpenult next Vdone?
  Vdone?
  Vdone?
  Vdone?)

(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"

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

It's your turn

Iteration/Tail Recursion 41

genlist: my-range and halves

What are the values of the following calls to genlist?

```
(my-range lo hi)
  > (my-range 10 15)
  '(10 11 12 13 14)
  > (my-range 20 10)
  '()
```

```
(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)
```

It's your turn

Iteration/Tail Recursion 42

Iteration/Tail Recursion 43

Iteration/Tail Recursion 44
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))
  )
)
```

<table>
<thead>
<tr>
<th>step</th>
<th>num</th>
<th>prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>24</td>
</tr>
</tbody>
</table>

> (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 21259360) (1 3628800))

Your turn: sum-list iteration table

```
(define (sum-list-table ns)
  (genlist
    ; next
    (λ (nums sum)
      (let ((nums (first nums ans))
        (sum (second nums ans)))
        (list (rest nums) (+ (first nums) sum))))
    ; done?
    (λ (nums sum) (null? nums))
  )
  ; keepDoneValue?
)
```

```
> (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-apply (λ (nums ans)
                  (list (rest nums) (+ (first nums) ans)))
                  (λ (nums ans) (null? (first nums ans)))
                  #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))))
```

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**

```scheme
(define (partial-sums-between lo hi)
  (map second
    (genlist-apply
      ; next
      ; done?
      ; keepDoneValue?
      ; seed
    )))
)
```

```scheme
> (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-apply**

```scheme
(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 '()))
)
```

**Iterative Version of genlist**

```scheme
;; 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 (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 '()))
)
```

**Example: How does this work?**

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

**Iteration/Tail Recursion**