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)))))

(factor-rec 4):

Small-Step Semantics

\[
\begin{align*}
((\text{fact-rec} \ 4) & \Rightarrow ((\lambda \text{fact-rec} \ 4)) \\
& \Rightarrow (* \ 4 ((\lambda \text{fact-rec} \ 3))) \\
& \Rightarrow (* \ 4 (* \ 3 ((\lambda \text{fact-rec} \ 2)))) \\
& \Rightarrow (* \ 4 (* \ 3 (* \ 2 ((\lambda \text{fact-rec} \ 1))))) \\
& \Rightarrow (* \ 4 (* \ 3 (* \ 2 (* \ 1 ((\lambda \text{fact-rec} \ 0))))) \\
& \Rightarrow (* \ 4 (* \ 3 (* \ 2 (* \ 1 1)))) \\
& \Rightarrow (* \ 4 (* \ 3 (* \ 2 1))) \\
& \Rightarrow (* \ 4 (* \ 3 2)) \\
& \Rightarrow (* \ 4 6) \\
& \Rightarrow 24
\end{align*}
\]

Invocation Tree

Idea: multiply on way down

An iterative approach to factorial

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 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/

Small-Step Semantics

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

Tail-recursive factorial: Dynamic execution

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

```
((fact-iter 4)
  ((lambda (fact-iter 4))
    ((lambda (fact-tail 1 4))
      ((lambda (fact-tail 3 4))
        ((lambda (fact-tail 2 12))
          ((lambda (fact-tail 1 24))
            ((lambda (fact-tail 0 24))
              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**.

```
; 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**

```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)

```
RuntimeError: maximum recursion depth exceeded
```

**inc_iter/inc-tail in Racket Solutions**

```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?**

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**

```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
```

**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

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**

**Execution frame for fact_while(4)**

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

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?

```
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!

Relating Tail Recursion and while loops

```
def fact_iter(n):
    (define (fact-tail num prod)
     (if (= num 0)
           ans
           (fact-tail (- num 1) (* num prod))))

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

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

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:

<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 Solutions

Flesh out the missing parts

```racket
(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?

```python
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
```

```python
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 = 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)

```python
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:

```python
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.

```racket
(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

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

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

### Iteration table

<table>
<thead>
<tr>
<th>nums</th>
<th>sumSoFar</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(6 3 -5 7)</td>
<td>0</td>
</tr>
<tr>
<td>'(3 -5 7)</td>
<td>6</td>
</tr>
<tr>
<td>'(-5 7)</td>
<td>9</td>
</tr>
<tr>
<td>'(7)</td>
<td>4</td>
</tr>
<tr>
<td>'()</td>
<td>11</td>
</tr>
</tbody>
</table>

### Capturing list iteration via `my-foldl`

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

### `my-foldl` Examples

```
> (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 \texttt{foldl} Function
Folds over Any Number of Lists

\begin{verbatim}
> (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)
\end{verbatim}

Iterative vs Recursive List Reversal Solutions

\begin{verbatim}
(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))
\end{verbatim}

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

How about stack depth?

\textbf{Ans:}

\begin{itemize}
  \item \texttt{reverse-iter}: exactly n conses, none pending; \text{O}(1) stack space.
  \item \texttt{snoc}: exactly n+1 conses, all pending; \text{O}(n) stack space
  \item \texttt{reverse-rec}: quadratic (\text{O}(n^2)) conses, all pending; \text{O}(n^2) stack space
\end{itemize}

What does this do? Solutions

\begin{verbatim}
(define (whatisit f xs)
  (foldl (λ (x listSoFar)
  (cons (f x) listSoFar))
  null
  xs)))
\end{verbatim}

\textbf{Ans:} It performs the "reverse map" of function \texttt{f} on list \texttt{xs}.

E.g., \texttt{(whatisit (λ (n) (* n 3)) '(7 2 4))} \Rightarrow '(12 6 21)

To perform a regular map, change \texttt{foldl} to \texttt{foldr}!

Tail Recursion Review 1 Solutions

\begin{verbatim}
# Euclid’s algorithm
def gcd(a,b):
  while b != 0:
    temp = b
    b = a \% b
    a = temp
  return a
\end{verbatim}

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

\begin{tabular}{|c|c|}
  \hline
  a & b \\
  \hline
  42 & 72 \\
  72 & 42 \\
  42 & 30 \\
  30 & 12 \\
  12 & 6 \\
  6 & 0 \\
  \hline
\end{tabular}

\begin{verbatim}
(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b)))))
\end{verbatim}
Tail Recursion Review 2 Solutions

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

---

iterate

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

For example:

```racket
(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)))
```

For example:

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

; Soln 1

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

; Soln 2

```racket
(define (least-power-geq base threshold)
  (iterate (λ (exp) (+ exp 1))
           (λ (exp) (>= (expt base exp) threshold))
           (λ (exp) (expt base exp))
           (list n 1))
```

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.

What do These Do?

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

---

least-power-geq

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

; Soln 2

```racket
(define (least-power-geq base threshold)
  (iterate (λ (exp) (+ exp 1))
           (λ (exp) (>= (expt base exp) threshold))
           (λ (exp) (expt base exp))
           (list n 1))
```

> `(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 \textit{let} to introduce local names

\begin{verbatim}
(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) prod)
        (list n 1)))
\end{verbatim}

Using \textit{match} to introduce local names

\begin{verbatim}
(define (fact-match n)
  (iterate (λ (num prod)
             (match num prod
                     [(list num prod)
                      (list (- num 1) (* num prod))])
         (λ (num prod) (<= num 0))
         (λ (num prod) prod)
        (list n 1)))
\end{verbatim}

Racket’s \texttt{apply}

\begin{verbatim}
(define (avg a b)
  (/ (+ a b) 2))
\end{verbatim}

apply takes (1) a function and (2) a single argument that is a \texttt{list of values} and returns the result of applying the function to the values.

iterate-apply: a kinder, gentler iterate

\begin{verbatim}
(define (iterate-apply next done? finalize state)
  (if (apply done? state)
      (apply finalize state)
      (iterate-apply next done? finalize (apply next state)))))
\end{verbatim}

\begin{verbatim}
(define (fact-iterate-apply n)
  (iterate-apply (λ (num prod)
                  (match num prod
                        [(list num prod)
                         (list (- num 1) (* num prod))])
               (λ (num prod) (<= num 0))
               (λ (num prod) prod)
              (list n 1)))
\end{verbatim}

\begin{verbatim}
(step | num | prod)
      | 1   | 4   |
      | 2   | 3   |
      | 3   | 2   |
      | 4   | 1   |
      | 5   | 0   |
\end{verbatim}
iterate-apply: fib and gcd Solutions

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

```scheme
(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
)
```

Creating lists with genlist

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

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 2)) (λ (n) (> n 100)) #t 1)`
  - '(1 2 4 8 16 32 64 128)

- `(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)) #f 1)`
  - '(1 2 4 8 16 32 64)

- `(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)

genlist: my-range and halves Solutions

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

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

```
> (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)
              (let ([nums (first nums&ans)]
                    [sum (second nums&ans)])
               (list (rest nums) [sum (second nums&ans)])
               (λ (nums&sum) (null? (first nums&sum)))
               (list ns 0))
    (λ (nums&sum) (null? (first nums&sum)))
    (null? (first nums&sum)))
    (list ns 0)))
```

```
> (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))
```

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

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)))
               (list ns 0))))
```

```
> (partial-sums '(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))
```

Example:

```
(define (partial-sums ns)
  (map second
    (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))
      (null? (first nums&sum)))
    (list ns 0))))
```

genlist-apply: a kinder, gentler genlist
(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)

---

(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 '()))

Example: How does this work?

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