Iteration via Tail Recursion in Racket

Overview

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

Factorial Revisited

An iterative approach to factorial

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

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

**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>

**Idea:** multiply on way down

SOLUTIONS

CS251 Programming Languages
Fall 2018, Lyn Turbak
Department of Computer Science
Wellesley College
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))

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.

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

;;;; 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>

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

The evaluation thread is no longer running, so no evaluation can take place until the next execution. The program ran out of memory.

Increase memory limit to 256 megabytes [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**

```scheme
(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?

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

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
```

While loop factorial: Execution Land

Execution frame for fact_while(4)

<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>

Don’t forget to return answer!
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!

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

Note: tail recursion doesn’t have this gotcha!

## Relating Tail Recursion and while loops

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

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

```
(define (fib 4) : 3
  (define fib(4) : 3
    (define fib(3) : 2
      (define fib(2) : 1
        (define fib(2) : 1
          (define fib(1) : 1
            (define fib(0) : 0
              +))))
    +))
```

---

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)
  (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

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

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

(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 \texttt{my-foldl}

\[
\begin{array}{llll}
\text{v}_1 & \rightarrow & \text{v}_2 & \rightarrow \cdots \rightarrow \text{v}_{n-1} & \rightarrow \text{v}_n \\
\text{initval} & \rightarrow \text{combine} & \rightarrow \text{combine} & \rightarrow \cdots \rightarrow \text{combine} & \rightarrow \text{nullval}
\end{array}
\]

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

\textbf{my-foldl} Examples

\[
\begin{align*}
> & \ (\text{my-foldl} \ + \ 0 \ (\text{list} \ 7 \ 2 \ 4)) \\
& 13 \\
> & \ (\text{my-foldl} \ * \ 1 \ (\text{list} \ 7 \ 2 \ 4)) \\
& 56 \\
> & \ (\text{my-foldl} \ \text{cons} \ \text{null} \ (\text{list} \ 7 \ 2 \ 4)) \\
& '(4 \ 2 \ 7) \\
> & \ (\text{my-foldl} \ (\lambda \ (n \ res) \ (+ \ (*3 \ res) \ n)) \ 0
\ (\text{list} \ 10 -4 5 2)) \\
& 251 ; = 10*3^3 + -4*3^2 + 5*3^1 + 2*3&0 \\
& \text{An example of Horner's method} \\
& \text{for polynomial evaluation}
\end{align*}
\]
**Built-in Racket foldl Function**

Folds over Any Number of Lists

```racket
> (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)
```

---

**Iterative vs Recursive List Reversal**

```racket
(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?

**Ans:**
- reverse-iter: exactly n conses
- snoc: exactly n+1 conses
- reverse-rec: quadratic (O(n^2)) conses
```

---

**What does this do?**

```racket
(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**

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

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

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

---

**Iteration/Tail Recursion 29**

**Iteration/Tail Recursion 30**

**Iteration/Tail Recursion 31**

**Iteration/Tail Recursion 32**
Tail Recursion Review 2

```python
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.

<table>
<thead>
<tr>
<th>digits</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,7,2,9]</td>
<td>0</td>
</tr>
<tr>
<td>[7,2,9]</td>
<td>1</td>
</tr>
<tr>
<td>[2,9]</td>
<td>17</td>
</tr>
<tr>
<td>[]</td>
<td>172</td>
</tr>
</tbody>
</table>

```
(digits i)
[(1,7,2,9) 0]
[(7,2,9) 1]
[(2,9) 17]
[[] 172]
```

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

Evaluate the following:
```
(define (toInt digits)
    (foldl (λ (d i) (+ (* 10 i) d)) 0 digits))
```

Iteration/Tail Recursion 34

```
(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)))
```

Iteration/Tail Recursion 35

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

; Soln 2
(define (least-power-geq base threshold)
    (iterate (λ (exp) (+ exp 1))
        (λ (exp) (> 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
```

Iteration/Tail Recursion 36

```
(define (least-power-geq base threshold)
    (iterate (λ (ns) (cons (- (first ns) 1) ns))
        (λ (ns) (<= (first ns) 0))
        (λ (ns) ns)
        (list n)))
```

```
(define (least-power-geq base threshold)
    (iterate (λ (ns) (cons (quotient (first ns) 2) ns))
        (λ (ns) (<= (first ns) 1))
        (λ (ns) (- (length ns) 1))
        (list n)))
```

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)))
```

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

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

```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))
               (λ (num&prod) (second num&prod))
               (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))]
                     (λ (num&prod) (<= num 0))
                     (λ (num&prod) prod)
                     (list n 1)))
```

---

Racket’s **apply**

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

```scheme
> (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**

```scheme
(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 prod]
     ([list (- num 1) (* num prod)])
     (λ (num&prod) (<= num 0))
     (λ (num&prod) prod)
     ([list num prod] prod))
   (list n 1)))
```

---

---

---

---

---

---

---

---

---

---

---

---

---

---
iterate-apply: fib and gcd

```scheme
(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
 ))
```

```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
 ))
```

Simple genlist examples

What are the values of the following calls to genlist?

```scheme
(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)
```

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))))
)
```

genlist: my-range and halves

```scheme
(define (my-range-genlist 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
 ));)
```

What are the values of the following calls to genlist?

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

> (my-range 20 10)
'()

> (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)
```
Using genlist to generate iteration tables

\[
\text{(define (fact-table n)}
\text{  (genlist (λ (num&prod))}
\text{    (let ((num (first num&ans)))
\text{      (prod (second num&ans)))})
\text{    (list (- num 1) (* num prod))))
\text{  (λ (num&prod) (<= (first num&prod) 0))})
\text{#t}
\text{  (list n 1))})
\]

> (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 2419200) (1 10080000))

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&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?”

Your turn: sum-list iteration table

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

> (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-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:

(let ((nums ans)
      (list (rest nums) (+ (first nums) ans)))
  (λ (nums ans) (null? nums))
  #t
  (list ns 0))))

It's your turn
**partial-sums-between**

```scheme
(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**

```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)
      (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**

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

---