Iteration via Tail Recursion in Racket
What is iteration?

Racket has no loops, and yet can express iteration.

- Tail recursion

Tail recursive list processing via \texttt{foldl}

- General iteration via \texttt{iterate} and \texttt{iterate-app}

- General iteration via \texttt{iterate} and \texttt{iterate-app}

Other useful abstractions

Overview

- Tail recursion!

How can that be?

- What is iteration?
(define (fact-rec n)
  (if (= n 0)
      1
      (* n (fact-rec (- n 1)))))

(fact-rec 4): 24
(fact-rec 3): 6
(fact-rec 2): 2
(fact-rec 1): 1
(fact-rec 0): 1
An iterative approach to factorial.

Iteration Rules:

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

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>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Iterative/Tail Recursion

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

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

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

Iteration Rules:

- next prod is previous num times previous prod.
- next num is previous num minus 1.
- prod is the product of all numbers already processed.
- num is the current number being processed.

State Variables:

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

Stopping condition:

\[
(\text{fact-tail (fact-tail-n 1))}
\]

Tail call (no pending operations) expresses iteration rules
Tail-recursive factorial:

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

Define (fact-tail n) = (fact-tail (–n) (* n prod))

Small-Step Semantics

```
define (fact-tail n) (**fact-tail (n \ 1) (* n prod))
```
The essence of iteration in Racket
Extremely silly and inefficient recursive incrementing function for testing Racket stack memory limits

\[
\text{\texttt{inc-rec}}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
1 + (\text{\texttt{inc-rec}}(\text{\texttt{inc-rec}}(n - 1))) & \text{otherwise}
\end{cases}
\]

Racket

\[
> \text{\texttt{inc-rec}}(10000000) \quad : \quad 10^7
\]

Eventually run out of stack space.
In [16]: inc_rec(100)
Out[16]: 101

In [17]: inc_rec(1000)

RuntimeError: maximum recursion depth exceeded
Will inc-iterate ever run out of memory?

\[
\begin{align*}
10000001 \\
\times (10000001) &> 10^7 \\
10000001 \\
\times (10000000) &> 10^8
\end{align*}
\]

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

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

```
Racket:
```
> (inc-iterate 10000000) ; 10^7
10000001
> (inc-iterate 100000000) ; 10^8
10000001
> (inc-iterate 1000000000) ; 10^9
```

\( \text{inc-iterate/\text{inc-tail}} \) in Racket
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)

In [19]: inc_iter(100)
Out[19]: 101
In [19]: inc_iter(1000)
……
RuntimeError: maximum recursion depth exceeded

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

```python
def inc_iter(n):
    if n == 0:
        return 1, resultSoFar + 1
    else:
        resultSoFar = inc_iter(n - 1, resultSoFar)
        return resultSoFar

inc_iter(100)
```

```python
inc_iter/ inc_tail in Python
```
Why the Difference?

When iter-tail(0,4) returns 4, no unnecessarily stacked frames need to be popped.

When iter-tail(0,4) returns 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.

Python pushes a stack frame for every call to iter-tail. When iter-tail(0,4) returns 4, no unnecessarily stacked frames need to be popped!
Origins of Tail Recursion

Guy Lewis Steele Jr., 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

• Looping constructs are unnecessary; tail recursive calls are a more general
  and elegant way to express iteration.

• Function calls should not push stack; subexpression evaluation should!

• One of the most important but least appreciated CS papers of all time

Guy Lewis Steele Jr., a.k.a. "The Great Quux"

A.I. Memo 443

October 1977

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
What to do in Python (and most other languages)?
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

• next num is previous num minus 1.
• next prod is previous num times previous prod.

Don't forget to return answer!
while loop factorial:

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

**Execution Frame for fact_while(4)**

<table>
<thead>
<tr>
<th>num</th>
<th>prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

**Iteration/Tail Recursion**
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

\[ \text{def fact-while}(n): \]
\[ \text{num} = n \]
\[ \text{prod} = 1 \]
\[ \text{while } (\text{num} > 0): \]
\[ \text{prod} = \text{num} \times \text{prod} \]
\[ \text{num} = \text{num} - 1 \]
\[ \text{return } \text{prod} \]

\[ \text{define (fact-tail n prod)} \]
\[ \text{if } (\text{num} = 0) \]
\[ \text{return } \text{prod} \]
\[ \text{prod} = \text{num} \times \text{prod} \]
\[ \text{num} = \text{num} - 1 \]
\[ \text{return } \text{prod} \]
Recursive Fibonacci

\[
\begin{align*}
\text{fib}(0) & : 1 \\
\text{fib}(1) & : 1 \\
\text{fib}(2) & : 2 \\
\text{fib}(3) & : 3 \\
\text{fib}(4) & : 5 \\
\end{align*}
\]

\[
\text{fib}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\text{fib}(n-1) + \text{fib}(n-2) & \text{if } n > 1 
\end{cases}
\]

\[
\begin{align*}
\text{define} & \quad \text{fib-rec} \quad n \\
& \quad \text{return rabbit pairs at month } n \\
& \quad \text{if } (n < 2) \quad \text{assume } n \geq 0 \\
& \quad \text{if } (n > 2) \\
& \quad \text{return rabbit pairs at month } n \\
& \quad \text{fib-rec} \quad (- n 1) + \text{fib-rec} \quad (- n 2) \\
\end{align*}
\]
The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

**Iteration/Recursion**

<table>
<thead>
<tr>
<th>( \text{Iter} )</th>
<th>( n )</th>
<th>( i )</th>
<th>( n_0 )</th>
<th>( n_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>13</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>21</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Iteration leads to a more efficient Fib
Iterative Fibonacci in Racket

Define (fib-iter n)

Define (fib-tail n i fibi fibi+1)

(Fib-tail (fib-iter n))

Flesh out the missing parts
Gotcha: assignment order and temporary variables

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

Or can use simultaneous assignment in languages that have it.

```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

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
```

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

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

2. Use simultaneous assignment:

```
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
```

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

(fib-tail 0 0 1)

Local fib-tail can be defined locally within fib-iter. Since n remains constant, don't need it as an argument.

Local fib-tail function in fib-iter.
(define (sum-list-iter L)
  (if (null? L)
      sumSoFar
      (sum-list-iter (rest nums) (+ (first nums) sumSoFar))))

<table>
<thead>
<tr>
<th>Iteration</th>
<th>List</th>
<th>Sum</th>
<th>nums</th>
<th>sumSoFar</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>()</td>
<td>0</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1</td>
<td>(7)</td>
<td>7</td>
<td>(7)</td>
<td>(7)</td>
</tr>
<tr>
<td>2</td>
<td>(-5 7)</td>
<td>-5</td>
<td>(-5 7)</td>
<td>(-5 7)</td>
</tr>
<tr>
<td>3</td>
<td>(3 -5 7)</td>
<td>3</td>
<td>(3 -5 7)</td>
<td>(3 -5 7)</td>
</tr>
<tr>
<td>4</td>
<td>(6 3 -5 7)</td>
<td>6</td>
<td>(6 3 -5 7)</td>
<td>(6 3 -5 7)</td>
</tr>
</tbody>
</table>

Iteration/Tail Recursion
\[
\begin{align*}
\text{my-foldl} & \ (\text{combine} \ (\text{first} \ (xs \ \text{resultSoFar}) \ \text{resultSoFar}) \\
& \quad \text{my-foldl} \ \text{combine} \\
& \quad \text{resultSoFar} \\
& \quad (\text{null?} \ (xs)) \\
& \quad \text{definite (my-foldl} \ \text{combine resultSoFar} \\
& \quad \text{xs})
\end{align*}
\]

Capturing list iteration via \text{my-foldl}
foldr vs foldl

nullval

V

combine

V

combine

V

combine

initval

combine

combine

combine

nullval

combine

combine

combine

combine

iterate vs foldl
Examples

```lisp
(my-foldl + 0 (list 7 2 4)) 13
(my-foldl * 1 (list 7 2 4)) 56
(my-foldl - 0 (list 7 2 4)) 9
(my-foldl cons nil (list 7 2 4)) '(4 2 7)
(my-foldl (λ (n res) (+ (* 3 res) n)) 0 (list 10 -4 5 2)) 251
```

**An example of Horner's method for polynomial evaluation**
Built-in Racket `foldl` function

Folds over Any Number of Lists

Same design decision as in `map` and `foldr`

ERROR: `foldl`: given list does not have the same size as the first list: `(5 6 7)`

`(foldl 5 6 7)`

`(foldl 2 3 4)`

0

`(( (a b res) + (a b res) res) a b res)`

56

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

ERROR: `foldl`: given list does not have the same size as the first list: `(5 6 7)`

29
Iterative vs Recursive List Reversal

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

How about stack depth?

Iterative/Full Recursion

(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))
What does this do?

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

Iteration/Tail Recursion
Tail Recursion Review

1. Create an iteration table for $\text{gcd}(42, 72)$.

2. Translate Python $\text{gcd}$ into Racket tail recursion.

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

```racket
return a
def gcd(a, b):
    while b != 0:
        temp = b
        b = a % b
        a = temp
    return a
```

```
return a
def gcd(a, b):
    while b != 0:
        temp = b
        b = a % b
        a = temp
    return a
```
Tail Recursion Review 2

1. Create an iteration table for

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Python</th>
<th>Racket</th>
<th>Tail Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Translate Python `toInt` into Racket `foldl`.

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

3. Translate Python `toInt` into Racket tail recursion.

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

Racket Tail

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Python</th>
<th>Racket</th>
<th>Tail Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(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:

(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:

(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)))

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

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

(least-power-geq 3 100)

(define (least-power-geq base threshold)
  ; initial state
  ; initialize
  ; done?
  ; iterate
  (iterate (least-power-geq base threshold))
What do These Do?

```scheme
(define (mystery1 n)
  (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)))
```

**Iteration/Tail Recursion**
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]
            (λ (num&prod) (iterate (λ (num&prod) (define (fact-match n))))))))))))
  (list n 1))
```

Iteration/Tail Recursion
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.

Racket's apply:

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

```racket
> (avg 6 10)
8
> (apply avg '(6 10))
8
> (define (λ (a b c) (+ (* a b) c)) (λ (a b c) (+ a b c) (λ (a b c) (+ a b c) 2 3 4)))
> (apply (λ (a b c) (+ (* a b) c)) (list 2 3 4))
10
> (define (λ (a b c) (+ (* a b) c)) (λ (a b c) (+ a b c) 2 3 4))
> (apply (λ (a b c) (+ (* a b) c)) (list 2 3 4))
10
```

Racket's apply's
iterate-apply:

\[
\text{iterate-apply: } \text{a kinder, gentler iterate-apply}
\]

\[
\begin{array}{|c|c|}
\hline
\text{num} & \text{prod} \\
\hline
2 & 0 \\
2 & 1 \\
12 & 2 \\
4 & 3 \\
4 & 4 \\
\hline
\end{array}
\]

\[
\text{fact-iterate-apply: } n \\
\]

\[
\begin{array}{|c|c|}
\hline
\text{step} & \text{prod} \\
\hline
5 & 1 \\
4 & 2 \\
3 & 3 \\
2 & 4 \\
1 & 4 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{iterate-apply:} & \text{fact-iterate-apply:} \\
\hline
\text{initialize state} & \text{initialize state} \\
\text{iterate-apply:} & \text{iterate-apply:} \\
\text{next done? finalize state} & \text{next done? finalize state} \\
\text{iterate-apply:} & \text{iterate-apply:} \\
\text{next done? finalize state} & \text{next done? finalize state} \\
\hline
\end{array}
\]
(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
Creating lists with `genlist`
Simple genlist examples

\[
\begin{align*}
\text{genlist } & (\lambda (n) (- n 1)) (\lambda (n) (= n 0)) \texttt{#t} 5) \\
\text{genlist } & (\lambda (n) (* n 2)) (\lambda (n) (> n 100)) \texttt{#t} 1) \\
\text{genlist } & (\lambda (n) (* n 2)) (\lambda (n) (> n 100)) \texttt{#f} 1) \\
\text{genlist } & (\lambda (n) (= n 0)) (\lambda (n) (> n 100)) \texttt{#f} 1) \\
\end{align*}
\]

What are the values of the following calls to genlist?

\[
\begin{align*}
((0 \quad u \quad =) \quad (u) \quad v) \\
((1 \quad u \quad =) \quad (u) \quad v) \\
((1 \quad u \quad -) \quad (u) \quad v) \\
((0 \quad u \quad =) \quad (u) \quad v) \\
((1 \quad u \quad -) \quad (u) \quad v) \\
\end{align*}
\]

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

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

### Example Usage
- `(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)`
- `(halves 0)`
  - `(0)`
- `(halves 1)`
  - `()`
Using `genlist` to generate iteration tables

```
(define (fact-table n)
  (genlist (λ (num&prod)
    (let ((num (first num&prod))
          (prod (second num&prod)))
      (list (- num 1) (* num prod))))
    (λ (num&prod) (<= (first num&prod) 0))
    #t)
    (fact-table 10))
```

```
<table>
<thead>
<tr>
<th>num</th>
<th>prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

Iteration/Tail Recursion

```
(define (test n i)
  (cond ((<= i n) (test (prod i num) (i+ num 1)))
        ((< i n) (list i i))
        (else (list num (prod num))))
```

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

< (fact-table 4)>
((4 1) (3 4) (2 12) (1 24) (0 24))

< (fact-table 4)>
((3 1) (2 2) (1 4) (0 4))

< (fact-table 4)>
((2 1) (1 2) (0 2))

< (fact-table 4)>
((1 1) (0 1))
```

```
Your turn: sum-list iteration table

```
(define (sum-list-table ns)
  (genlist (λ (nums sum)
               (let [{
                 nums (first nums)
                 ans (1)
               }]
               (list rest nums (+ sum (first nums))))
               (λ (nums sum)
                 (done?
                   (null? (first nums))))
               keepDoneValue?
               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))
```

Iteration/Tail Recursion
“Can I generate this list as the column of an iteration table?”

Moral: ask yourself the question

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

```
(define (partial-sums ns)
  (map second
       (sum (first ns ans))
              ; rest
       (list rest ns ans)
              ; first
       (genlist (remainder ns ans)
                (map second
                     (define (partial-sums-arrays ns)
                       (define table
                         (define table-rows (map first ns))
                         (define table-columns (map second ns))
                         (define table-rows-columns (map table-rows table-columns))
                         table-rows-columns)))))
```

```
(define (partial-sums-arrays ns)
  (define table
    (define table-rows (map first ns))
    (define table-columns (map second ns))
    (define table-rows-columns (map table-rows table-columns))
    table-rows-columns))
```

genlist can collect iteration table column!
Example:

```scheme
((apply (λ (next seed) cons seed) seed) (apply (λ (done? seed) "Kinder, gentler, gentler" done? next done? keepDoneValue?) (apply (λ (nums ans) (+ (first nums) ans)) (map second (define (partial-sums ns) (map second (genlist-apply (λ (nums ans) (null? nums)) (λ (nums ans) (null? nums)))) ns)))))
```
\[
\text{partial-sums-between}\lo\hi = \text{map second}\left(\text{genlist-apply}\left(\lambda\text{num sum};\text{next}\left(\lambda\text{num sum};\text{done?}\left(\lambda\text{num};\gt\text{num hi})\right)\right)\left(\text{list lo 0}\right)\right)\right)
\]

\[
\begin{align*}
\text{define}\ (\text{partial-sums-between}\lo\hi) = \\
\text{map second}\left(\text{genlist-apply}\left(\lambda\text{num sum};\text{next}\left(\lambda\text{num sum};\text{done?}\left(\lambda\text{num};\gt\text{num hi})\right)\right)\left(\text{list lo 0}\right)\right)\right)
\end{align*}
\]

> (partial-sums-between 3 7)
'(0 3 6 10 15 21 28 36 45 55)

> (partial-sums-between 1 10)
'(0 1 3 6 10 15 21 28 36 45 55)
Iterative Version of \texttt{gentlist}

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

Example: How does this work?
```

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