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!
• 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

Idea: multiply on way down

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

Iteration Table:

<table>
<thead>
<tr>
<th>step</th>
<th>num</th>
<th>ans</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 ans` is previous `num` times previous `ans`.

An iterative approach to factorial
Iterative factorial: tail recursive version

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

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

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

---

Tail-recursive factorial: invocation tree

```
; Here, 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 ans)
  (if (= num 0)
      ans
      (fact-tail (- num 1) (* num ans))))
```

**Invoke Tree:**

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

**Iteration Table:**

<table>
<thead>
<tr>
<th>step</th>
<th>num</th>
<th>ans</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>

---

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
```

---

![Message Board Image]
**inc_rec in Python**

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

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

In [17]: inc_rec(1000)

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

In [18]: inc_rec(10) => 11
In [19]: inc_rec(100) => 101

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

**inc_iter/inc_tail in 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?

```
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)

```
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)
```

**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 inc-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 `ans` is previous `num` times previous `ans`.

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

Don’t forget to return `ans`!

while loop factorial: Execution Land

**Execution frame for fact_while(4)**

```
num   ans
----- ----
4   1
3   12
2   24
1
0
```

**Execution Land:**

<table>
<thead>
<tr>
<th>step</th>
<th>num</th>
<th>ans</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>
Gotcha! Order of assignments in loop body

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

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

**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

```python
(def fact_iter n)
(fact-tail n 1)
(def fact_tail num ans)
(if (= num 0)
    ans
    (fact-tail (- num 1) (* num ans)))
```

While not done, update variables

When done, return ans

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>fib_i</th>
<th>fib_i_plus_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 fib_i fib_i_plus_1)
  (if (= i n)
      fib_i
      (fib-tail n
        (+ i 1)
        fib_i_plus_1
        (+ fib_i fib_i_plus_1))))
```

Gotcha! Assignment order and temporary variables

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

```
def fib_for1(n):
defib_i= 0
defib_i_plus_1 = 1
for i in range(n):
defib_i = fib_i_plus_1
defib_i_plus_1 = fib_i + fib_i_plus_1
return fib_i
```

```
def fib_for2(n):
defib_i= 0
defib_i_plus_1 = 1
for i in range(n):
defib_i_plus_1 = fib_i + fib_i_plus_1
defib_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):
defib_i= 0
defib_i_plus_1 = 1
for i in range(n):
defib_i_prev = fib_i
defib_i = fib_i_plus_1
defib_i_plus_1 = fib_i_prev + fib_i_plus_1
return fib_i
```

2. Use simultaneous assignment:

```
def fib_for_fixed2(n):
defib_i= 0
defib_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 fib_i fib_i_plus_1)
    (if (= i n)
        fib_i
        (fib-tail (+ i 1)
          fib_i_plus_1
          (+ fib_i fib_i_plus_1)))))
  (fib-tail n 0 0 1)
```

Iteration/Tail Recursion 21

Iteration/Tail Recursion 22

Iteration/Tail Recursion 23

Iteration/Tail Recursion 24
**Iterative list summation**

Iterative list summation via `my-foldl`

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

**Iteration table**

<table>
<thead>
<tr>
<th>L</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(6 3 -22 5)</td>
<td>0</td>
</tr>
<tr>
<td>'(3 -22 5)</td>
<td>6</td>
</tr>
<tr>
<td>'(-22 5)</td>
<td>9</td>
</tr>
<tr>
<td>'(5)</td>
<td>-13</td>
</tr>
<tr>
<td>'()</td>
<td>-8</td>
</tr>
</tbody>
</table>

**my-foldl Examples**

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

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

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

> (my-foldl (λ (n res) (+ (* 10 res) n))
  0
  (list 7 2 4))
724 ; An example of Horner’s method
  ; for polynomial evaluation
```

**Built-in Racket foldl Function**

Folds over Any Number of Lists

```scheme
> (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 ; 2*5 + 3*6 + 4*7

> (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

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

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

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

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
• reverse-rec: quadratic (O(n^2)) conses
• snoc: exactly n+1 conses

What does this do?

(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!

Your Turn

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

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

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

How could we return just the exponent rather than the base raised to the exponent?
What do These Do?

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

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

mystery1 returns the list of ints from 0 up to and including n.
E.g., (mystery1 5) => '(0 1 2 3 4 5)
mystery2 calculates the log-base-2 of n by determining how many times
n can be divided by 2 before reaching 1. E.g.
(mystery2 32) => 5

Using let to introduce local names

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

Using match to introduce local names

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

Racket's apply

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

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

apply takes (1) a function and (2) a single argument that is a
list of values and returns the result of applying the function
to the values.
**iterate-apply:** a kinder, gentler iterate

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

**Your Turn**

```scheme
(define (fib-iterate-apply n)
  (iterate-apply (λ (i fib_i fib_i_plus_1) (list (+ i 1) fib_i_plus_1 (+ fib_i fib_i_plus_1))) ; next
                  (λ (i fib_i fib_i_plus_1) (= i n)) ; done?
                  (λ (i fib_i fib_i_plus_1) fib_i) ; finalize
                  (list 0 0 1) ; initial state
 ))
```

**Creating lists with genlist**

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

**Simple genlist examples**

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

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

*not iterative as written, but next function gives iterative "flavor"*
Your Turn

(my-range lo hi)

> (my-range 10 20)
'(10 11 12 13 14 15 16 17 18 19)

> (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)

Using genlist to generate iteration tables

(define (fact-table n)
  (genlist (λ (num&ans)
    (let ((num (first num&ans))
          (ans (second num&ans)))
      (list (- num 1) (* num ans))))
    (λ (num&ans) (<= (first num&ans) 0))
    #t
    (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 1814400) (1 3628800) (0 3628800))

Your Turn: sum-list iteration table

(define (sum-list-table ns)
  (genlist (λ (nums&ans)
    (let ((nums (first nums&ans))
          (ans (second nums&ans)))
      (list (rest nums) (+ (first nums) ans))))
    (λ (nums&ans) (null? (first nums&ans)))
    #t
    (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)

Using genlist to generate iteration tables

(define (fact-table n)
  (genlist (λ (num&ans)
    (let ((num (first num&ans))
          (ans (second num&ans)))
      (list (- num 1) (* num ans))))
    (λ (num&ans) (<= (first num&ans) 0))
    #t
    (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 1814400) (1 3628800) (0 3628800))

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))))
```

Your turn: partial-sums-between

```
(define (partial-sums-between lo hi)
  (map second
   (genlist-apply
    ; Flesh out parts
    )))

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

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

Iterative Version of genlist

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

Example: How does this work?

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

Iterative Version of genlist-apply

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

Iteration/Tail Recursion 45
(define (fact-table-apply-iter n)
 (genlist-apply-iter
  (λ (num ans) (list (- num 1) (* num ans)))
  (λ (num ans) (<= num 0))
  #t
  (list n 1)))