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

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

Invocation Tree

Idea: multiply on way down

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>

An iterative approach to factorial

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

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

O(1) time complexity for tail recursive version of factorial.
Iterative factorial: tail recursive version

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

\[
\text{(define (fact-tail num ans )}
\]
\[
\text{(if (= num 0)}
\]
\[
\text{ans}
\]
\[
\text{(fact-tail (- num 1) (* num ans)))}
\]

;; 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!
\[
\text{(define (fact-iter n)}
\]
\[
\text{(fact-tail n 1))}
\]

---

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!
\[
\text{(define (fact-tail num ans )}
\]
\[
\text{(if (= num 0)}
\]
\[
\text{ans}
\]
\[
\text{(fact-tail (- num 1) (* num ans)))}
\]

Invocation Tree:

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

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
\[
\text{(define (inc-rec n)}
\]
\[
\text{(if (= n 0)}
\]
\[
\text{1}
\]
\[
\text{(+ 1 (inc-rec (- n 1))))}
\]

> (inc-rec 1000000) ; 10^6
1000001

> (inc-rec 10000000) ; 10^7

---

The evaluation threshold is no longer warning, no evaluation can take place until the next execution.
The program ran out of memory.
Show this dialog next time: Increase stack limit to 16 megabytes:  Off
def inc_rec (n):
    if n == 0:
        return 1
    else:
        return 1 + inc_rec(n - 1)

inc_rec(100)
Out[16]: 101

inc_rec(1000)

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

Why the Difference?

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)

inc_iter(100)
Out[19]: 101

inc_iter(1000)

RuntimeError: maximum recursion depth exceeded

def (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?
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

```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

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

```
step | num | ans |
-----|-----|-----|
1    | 4   | 1   |
2    | 3   | 4   |
3    | 2   | 12  |
4    | 1   | 24  |
5    | 0   | 24  |
```
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_while(n):
    num = n
    ans = 1
    while (num > 0):
        num = num - 1
        ans = num * ans
    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, 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

```scheme
(define (fib-iter n)
  (fib-tail ... ))

(define (fib-tail n i fib_i fib_i_plus_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

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)

```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.

```scheme
(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))
```
### Iterative List Summation

#### Iteration table

<table>
<thead>
<tr>
<th>nums</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>

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

#### Iteration table

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>...</th>
<th>vn-1</th>
<th>vn</th>
</tr>
</thead>
<tbody>
<tr>
<td>initval</td>
<td>combine</td>
<td>combine</td>
<td>...</td>
<td>combine</td>
</tr>
</tbody>
</table>

#### `my-foldl` Examples

```scheme
> (my-foldl + 0 (list 7 2 4))
> (my-foldl * 1 (list 7 2 4))
> (my-foldl cons null (list 7 2 4))
> (my-foldl (λ (n res) (+ (* 3 res) n)) 0 (list 10 -4 5 2))
```
**Built-in Racket foldl Function**
Folds over Any Number of Lists

> (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**

(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?

**What does this do?**

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

**Tail Recursion Review 1**

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

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

1. Create an iteration table for gcd(42, 72)
2. Translate Python gcd into Racket tail recursion.
3. Translate Python toInt into Racket foldl.
# Euclid’s algorithm

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

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

```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 (ns) (<= (first ns) 0))
   (λ (ns) (second num&prod))
   (list n l))))
```

## Your Turn

```racket
(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?
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) 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))]
      [prod prod]))
    (λ (num&prod) (<= num 0))
    (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
```

iterate-apply: a kinder, gentler iterate

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

(define (fib-iterate-apply n)
 (iterate-apply ??? ; next
 ??? ; done?
 ??? ; finalize
 ???; initial state ))

<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>

(define (gcd-iterate-apply a b)
 (iterate-apply ??? ; next
 ??? ; done?
 ??? ; finalize
 ???; initial state ))

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>72</td>
</tr>
<tr>
<td>72</td>
<td>42</td>
</tr>
<tr>
<td>42</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Simple genlist examples

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

(genlist (λ (n) (- n 1))
 (λ (n) (= n 0))
 #f
 5)

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

(genlist (λ (n) (* n 2))
 (λ (n) (> n 100))
 #f
 1)

Creating lists with genlist

V1

next

V2

next

Vpenult

next

Vdone?

seed

done?

#f

Keep VDone only if keepDoneValue? Is true

not iterative as written, but next function gives iterative `flavor`

Simple genlist examples

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)))
  #t
  (list n 1)))
```

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)))
  #t
  (list ns 0)))
```

### 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&ans)
               (let ((nums (first nums&ans)))
                 (ans (second nums&ans))))
               (list (rest nums) (+ (first nums) ans)))
    #t
    (list ns 0))))

```
> (partial-sums '(7 2 5 8 4))
'(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))))

(define (partial-sums ns)
  (map second
    (genlist-apply
      (λ (nums&ans)
       (let ((nums (first nums&ans)))
         (ans (second nums&ans))))
       (list (rest nums) (+ (first nums) ans)))
       #t
       (list ns 0))))
```

Example:

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

Iteration/Tail Recursion 47

Iteration/Tail Recursion 48
**Your turn: partial-sums-between**

```scheme
(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**

```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 (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?

```scheme
(genlist-iter (λ (n) (quotient n 2))
  (λ (n) (<= n 0))
  #t
  (list n 1))
```