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 \texttt{foldl}
- Other useful abstractions
  - General iteration via \texttt{iterate} and \texttt{iterate-apply}
  - General iteration via \texttt{genlist} and \texttt{genlist-apply}

Factoring Revisited

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

 Invocation Tree

State Variables:
- \texttt{num} is the current number being processed.
- \texttt{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 \texttt{num} is previous \texttt{num} minus 1.
- next \texttt{prod} is previous \texttt{num} times previous \texttt{prod}.

Small-Step Semantics

```
((fact-rec 4))
⇒ ((λ fact-rec 4))
⇒ (* 4 ((λ fact-rec 3)))
⇒ (* 4 (* 3 ((λ fact-rec 2))))
⇒ (* 4 (* 3 (* 2 ((λ fact-rec 1)))))
⇒ (* 4 (* 3 (* 2 (* 1 ((λ fact-rec 0)))))
⇒ (* 4 (* 3 (* 2 (* 1 1))))
⇒ (4 (* 3 (* 2 1)))
⇒ (4 (* 3 2))
⇒ (4 6)
⇒ 24
```
**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))))
```

**Invocation Tree**
```
(fact-tail 4)
(fact-tail 3 4)
(fact-tail 2 12)
(fact-tail 1 24)
(fact-tail 0 24)
```

**Small-Step Semantics**
```
(fact-tail 4)
  => ((λ_fact-tail 4))
  => ((λ_fact-tail 3 4))
  => ((λ_fact-tail 2 12))
  => ((λ_fact-tail 1 24))
  => (fact-tail 0 24)
```

**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>
```

**Tail-recursive factorial: Dynamic execution**
```
(define (fact-tail num prod)
  (if (= num 0)
      prod
      (fact-tail (- num 1) (* num prod))))
```

```
> (inc-rec 1000000) ; 10^6
1000001
> (inc-rec 10000000) ; 10^7
```

**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 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)

> `inc_rec(n)`
9  return 1
10  else:
---> 11  return 1 + inc_rec(n - 1)
```plaintext
RuntimeError: maximum recursion depth exceeded
```

**inc_iter/inc_tail in Racket**

```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?

**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)

> `inc_iter(n)`
9  return 1
10  else:
---> 11  return 1 + inc_tail(n - 1, 1)
```plaintext
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

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

**next num** is previous num minus 1.

**next prod** is previous num times previous 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>3</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>0</td>
<td>1</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>
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!

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

**Note:**
tail recursion doesn’t have this gotcha!

## 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) )
(define (fib-tail n i 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 = fib_i_plus_1 + 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!)

```
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
```

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 (+ fibi fibi+1))))
    (fib-tail 0 0 1))
```
### Iterative List Summation

#### 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>

#### Code snippet

```scheme
(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 `my-foldl`

#### `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`
- `> (my-foldl cons null (list 7 2 4))`
  - `'(4 2 7)`
- `> (my-foldl (λ (n res) (+ (* 3 res) n)) 0 (list 10 -4 5 2))`
  - `251`

These examples showcase `my-foldl`'s capability to capture list iteration, similar to `foldl` and `foldr`, but with an additional flexibility in combining elements of the list.
**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?

How about stack depth?

**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

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

def toInt(digits):
    i = 0
    for d in digits:
        i = 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.

iterate

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

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)))

least-power-geq

(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?

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)))
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) (second num&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) (match num&prod
      [(list num prod) prod]))
    (list n 1)))
\end{verbatim}

Racket’s \textit{apply}

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

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

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

\textit{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) (match num&prod
      [(list num prod) prod]))
    (list n 1)))
\end{verbatim}
iterate-apply: fib and gcd

```scheme
(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))

(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)))
```

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))))

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

(define (halves num)
  (genlist
   (lambda (n) (* n 2))
   ; next
   (lambda (n) (> n 100))
   ; done?
   #f
   ; keepDoneValue?
   1))
```

Simple genlist examples

What are the values of the following calls to genlist?

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

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

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

(genlist (lambda (n) (* n 2))
  (lambda (n) (> n 100))
  #f
  1)
```

Iteration/Tail Recursion 41

Iteration/Tail Recursion 42

Iteration/Tail Recursion 43

Iteration/Tail Recursion 44
Using genlist to generate iteration tables

```scheme
(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))
        #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))
```

Your turn: sum-list iteration table

```scheme
(define (sum-list-table ns)
  (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)))
        #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))
```

genlist can collect iteration table column!

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

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

genlist-apply: a kinder, gentler genlist

```scheme
(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:

```scheme
(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))))
```

Moral: ask yourself the question
“Can I generate this list as the column of an iteration table?”
(define (partial-sums-between lo hi)
  (map second
    (genlist-apply
      ; next
      (λ (num sum) (list (+ num 1) (+ num sum)))
      ; done?
      (λ (num sum) (> num hi))
      ; 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 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))
  #t
  5)