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
  – Recursive list generation via \texttt{genlist} (can make iterative)
  – General iteration via \texttt{iterate}
Factorial Revisited

(\texttt{define (fact-rec n)}
 (\texttt{if (= n 0)}
  1
 (\texttt{* n (fact-rec (- n 1))}))))

Invocation Tree

- pending multiplication is nontrivial glue step

- divide

- glue

\texttt{(fact-rec 4): 24}

\texttt{(fact-rec 3): 6}

\texttt{(fact-rec 2): 2}

\texttt{(fact-rec 1): 1}

\texttt{(fact-rec 0): 1}
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`.

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>
Iterative factorial: tail recursive version

<table>
<thead>
<tr>
<th>Iteration Rules:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• next num is previous num minus 1.</td>
</tr>
<tr>
<td>• next ans is previous num times previous ans.</td>
</tr>
</tbody>
</table>

(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, 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))

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

Invocation Tree:

(fact-iter 4)
  (fact-iter 4 1)
    (fact-iter 3 4)
      (fact-iter 2 12)
        (fact-iter 1 24)
          (fact-iter 0 24)
            no glue!

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
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

(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): # 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)
...
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 `ans` is previous `num` times previous `ans`.

```python
def fact_while(n):
    num = n
    ans = 1  # Declare/initialize local state variables
    while (num > 0):
        ans = num * ans  # Calculate product and decrement num
        num = num - 1
    return ans  # Don’t forget to return answer!
```

8-15
while loop factorial: Execution Land

Execution frame for fact_while(4)

<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>
</tbody>
</table>

num = n
ans = 1

while (num > 0):
    ans = num * ans
    num = num - 1

return ans

<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!

```
(define (fact-tail num ans)
  (if (= num 0)
      ans
      (fact-tail (- num 1) (* num ans)))))
```
Relating Tail Recursion and while loops

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

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

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

(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
  )))

fib(4): 3
  fib(3): 2
    fib(2): 1
      fib(1): 1
        fib(1): 1
          fib(0): 0
    fib(1): 1
      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>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 ... ))

(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
```

```python
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 = 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
```
Iterative list summation

L → 6 → 3 → 22 → 5 → Ø

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>
Capturing list iteration via `my-foldl`

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

> (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) (+ (* 10 res) n)) 0 (list 7 2 4))
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 first list: '}(5 6 7)
Iterative vs Recursive List Reversal

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

(define (reverse-rec xs)
  (foldr (flip2 snoc) null xs))

(define (snoc ys x)
  (foldr cons (list x) ys))
What does this do?

(define (whatisit f xs)
  (foldl (λ (x listSoFar)
           (cons (f x) listSoFar))
        null
    xs))
(define (genlist next done? seed)
  (if (done? seed)
      null
      (cons seed
        (genlist next done? (next seed)))))

> (genlist (λ (n) (- n 1))
  (λ (n) (= n 0))
  5)

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

Because of the pending conses, this genlist is not iterative (but we’ll see soon how to make it iterative)
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)
iterate

(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 1)))
Your Turn

(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 \texttt{let} to introduce local names

\begin{verbatim}
(define (fact-let n)
  (iterate (\lambda (num&prod)
    (let ([num (first num&prod)]
      [prod (second num&prod)])
      (list (- num 1) (* num prod))))
  (\lambda (num&prod) (\leq (first num&prod) 0))
  (\lambda (num&prod) (second num&prod))
  (list n 1)))
\end{verbatim}
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]))
    (list n 1)))
```
**apply and iterate-apply**

```scheme
> ((λ (a b c) (+ (* a b) c)) 2 3 4)
10

> (apply (λ (a b c) (+ (* a b) c)) (list 2 3 4))
10

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

(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

(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>
An Iterative Version of genlist

(define (genlist-iter next done? seed)
    (iterate (λ (elts) (cons (next (first elts)) elts))
             (λ (elts) (done? (first elts)))
             (λ (elts) (reverse (rest elts)))
             ; Eliminate done seed & reverse list
             (list seed)))

Example: How does this work?

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