Iteration via Tail Recursion in Racket

CS251 Programming Languages
Fall 2016, Lyn Turbak
Department of Computer Science
Wellesley College

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}

Factorial Revisited

\begin{verbatim}
(define (fact-rec n)
  (if (= n 0)
      1
      (* n (fact-rec (- n 1)))))
\end{verbatim}

Invocation Tree

An iterative approach to factorial

State Variables:
• \texttt{num} is the current number being processed.
• \texttt{ans} is the product of all numbers already processed.

Iteration Table:

\begin{tabular}{|c|c|c|}
\hline
step & num & ans  \\
\hline
1   & 4   & 1    \\
2   & 3   & 4    \\
3   & 2   & 12   \\
4   & 1   & 24   \\
5   & 0   & 24   \\
\hline
\end{tabular}

Iteration Rules:
• next \texttt{num} is previous \texttt{num} minus 1.
• next \texttt{ans} is previous \texttt{num} times previous \texttt{ans}.
Iterative factorial: tail recursive version

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

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

The essence of iteration in Racket

- A process is \textit{iterative} if it can be expressed as a sequence of
  steps that is repeated until some stopping condition is reached.
- In \textit{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 \textit{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 \textit{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 \textit{loops}.

Tail-recursive factorial: invocation tree

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

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>

inc-rec in Racket

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

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

> (inc-rec 10000000) ; 10^7

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
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

```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)
...:

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

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 443
October 1977

DEBUNKING THE "EXPENSIVE PROCEDURE CALL" MYTH
or, PROCEDURE CALL IMPLEMENTATIONS CONSIDERED HARMFUL
or, LAMBDA: THE ULTIMATE GOTO
by
Guy Lewis Steele Jr.

Guy Lewis Steele
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
• 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 answer!

**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>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>24</td>
</tr>
</tbody>
</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>
```
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):
    num = n
    ans = 1
    while (num > 0):
        num = num - 1
        ans = num * ans
    return ans
```

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>

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/Tail Recursion 17

Iteration/Tail Recursion 18

Iteration/Tail Recursion 19

Iteration/Tail Recursion 20
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?

```
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
    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):
    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:

```
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]
```

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>

Iteration/Tail Recursion 21
Capturing list iteration via \texttt{my-foldl}

\begin{verbatim}
(define (my-foldl combiner resultSoFar xs)
  (if (null? xs)
      resultSoFar
      (my-foldl combiner
        (combiner (first xs) resultSoFar)
        (rest xs))))
\end{verbatim}

\textbf{my-foldl Examples}

\begin{verbatim}
> (my-foldl + 0 (list 7 2 4))
8

> (my-foldl * 1 (list 7 2 4))
14

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

> (my-foldl (λ (n res) (+ (* 10 res) n)) 0 (list 7 2 4))
40
\end{verbatim}

\textbf{Built-in Racket foldl Function}

Folds over Any Number of Lists

\begin{verbatim}
> (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

> (ERROR: foldl: given list does not have the same size as the first list: '(5 6 7))
\end{verbatim}

\textbf{Iterative vs Recursive List Reversal}

\begin{verbatim}
(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?
\end{verbatim}
What does this do?

\[
\text{(define (whatisit } f \text{ } xs) \\
(\text{foldl } (\lambda \text{ (x listSoFar)} \\
(\text{cons } (f \text{ } x) \text{ listSoFar})) \\
\text{null} \text{ } xs)))
\]

iterate

\[
\text{(define (iterate next done? finalize state)} \\
(\text{if } \text{ (done? state)} \\
(\text{finalize state}) \\
(\text{iterate next done? finalize} \\
(\text{next state}))))
\]

For example:

\[
\text{(define (fact-iterate } n) \\
(\text{iterate } (\lambda \text{ (num&prod)} \\
(\text{list } (- \text{ (first num&prod)} 1) \\
(* \text{ (first num&prod)} \\
(\text{second num&prod}))) \\
(\lambda \text{ (num&prod)} \text{ (<= } \text{ (first num&prod)} 0) \\
(\lambda \text{ (num&prod)} \text{ (second num&prod}) \\
(\text{list } n 1)))
\]

Your Turn

\[
\text{(define (least-power-geq } \text{base threshold)} \\
(\text{iterate } ??? \text{ ; next} \\
??? \text{ ; done?} \\
??? \text{ ; finalize} \\
??? \text{ ; initial state}) \\
\)
\]

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

What do These Do?

\[
\text{(define (mystery1 } n) ; Assume } n \geq 0 \\
(\text{iterate } (\lambda \text{ (ns)} \text{ (cons } (- \text{ (first ns)} 1) \text{ ns}) \\
(\lambda \text{ (ns)} \text{ (<= } \text{ (first ns)} 0) \\
(\lambda \text{ (ns)} \text{ ns}) \\
(\text{list } n))
\]

\[
\text{(define (mystery2 } n) \\
(\text{iterate } (\lambda \text{ (ns)} \text{ (cons } \text{ (quotient } \text{ (first ns)} 2) \text{ ns}) \\
(\lambda \text{ (ns)} \text{ (<= } \text{ (first ns)} 1) \\
(\lambda \text{ (ns)} (- \text{ (length ns)} 1)) \\
(\text{list } n)))
\]

How could we return just the exponent rather than the base raised to the exponent?
Using **let** to introduce global names

```scheme
(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)))
```

**Iteration/Tail Recursion**

Using **match** to introduce global names

```scheme
(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)))
```

**Iteration/Tail Recursion**

**apply and iterate-apply**

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

**Iteration/Tail Recursion**

**Your Turn**

```scheme
(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>

**Iteration/Tail Recursion**
Iterative Version of \texttt{genlist}

\begin{verbatim}
;; 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)
\end{verbatim}

\texttt{genlist-apply-iter} Example

\begin{verbatim}
(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)))
\end{verbatim}

Iterative Version of \texttt{genlist-apply}

\begin{verbatim}
(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 '()))
\end{verbatim}