Tail Recursion

+tail.rkt
Recursion is an elegant and natural match for many computations and data structures.

- Natural recursion with immutable data can be space-inefficient compared to loop iteration with mutable data.
- **Tail recursion** eliminates the space inefficiency with a simple, general pattern.
- Recursion over immutable data expresses iteration more clearly than loop iteration with mutable state.
- More higher-order patterns: fold
Naturally recursive factorial

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

How efficient is this implementation?

Space: O(n)

Time: O(n)
CS 240-style machine model

- Registers
- Stack
  - Stack
  - Program Counter
  - Stack Pointer
- Code
- Stack
  - Call frame
  - Call frame
  - Call frame
- Heap
  - arguments, variables, return address per function call
  - cons cells, data structures, ...
Evaluation example

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

Call stacks at each step:

- \((\text{fact } 3)\):
  - \((\text{fact } 3) : 3\times_\cdot\)
  - \((\text{fact } 3) : 3\times_\cdot\)
  - \((\text{fact } 3) : 3\times_\cdot\)
- \((\text{fact } 2)\):
  - \((\text{fact } 2) : 2\times_\cdot\)
  - \((\text{fact } 2) : 2\times_\cdot\)
  - \((\text{fact } 2) : 2\times_\cdot\)
- \((\text{fact } 1)\):
  - \((\text{fact } 1) : 1\times_\cdot\)
  - \((\text{fact } 1) : 1\times_\cdot\)
  - \((\text{fact } 1) : 1\times_\cdot\)
- \((\text{fact } 0)\):

Remember: \(n \rightarrow 2\); and “rest of function” for this call.

Space: \(O(\quad)\)

Time: \(O(\quad)\)

Tail Recursion 5
Naturally recursive factorial

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

Base case returns base result.
Recursive case returns result so far.
Compute result so far after/from recursive call.
Compute remaining argument before/for recursive call.
Tail recursive factorial

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

Base case returns full result.
Recursive case returns full result.
Initial accumulator provides base result.
Accumulator parameter provides result so far.
Compute remaining argument before/for recursive call.
Compute result so far before/for recursive call.
Common patterns of work

Natural recursion:
- Argument
- Full result

Tail recursion:
- Argument
- Base result
- Base case

Reduce argument
Accumulate result so far
Accumulate result so far

Deeper recursive calls
Recursive case:
Compute result in terms of argument and accumulated recursive result.

\[
\text{(define (fact } n) \text{)} \text{ if } (= n 0) \text{ 1 (* n (fact } (- n 1)))))
\]
Tail recursion

Recursive case:
Compute recursive argument in terms of argument and accumulator.

\[
\text{(define (fact \ n)} \n\begin{align*}
\text{  (define (fact-tail \ n \ acc)} \n\text{    (if (= \ n \ 0) \n        \text{      acc)} \n        \text{        (fact-tail (- \ n \ 1) (* \ n \ acc)))}) \n\text{      (fact-tail \ n \ 1))}
\end{align*}
\]
Evaluation example

Call stacks at each step

\[(\text{fact } 3)\]  \[\text{ft } 3 1\]  \[\text{ft } 2 3\]  \[\text{ft } 1 6\]  \[\text{ft } 0 6\]

\[(\text{fact } 3)\]: _  \[\text{ft } 3 1\]: _  \[\text{ft } 2 3\]: _  \[\text{ft } 1 6\]: _  \[\text{ft } 0 6\]: _

Nothing useful remembered here.

\[(\text{fact } 3)\]: _  \[\text{ft } 3 1\]: _  \[\text{ft } 2 3\]: _  \[\text{ft } 1 6\]: _  \[\text{ft } 0 6\]: _

ft = fact-tail

etc.

Tail Recursion
Tail-call optimization

(define (fact n)
  (define (fact-tail n acc)
    (if (= n 0)
        acc
        (fact-tail (- n 1) (* n acc))))
  (fact-tail n 1))

Language implementation recognizes tail calls.
  • Caller frame never needed again.
  • Reuse same space for every recursive tail call.
  • Low-level: acts just like a loop.

Racket, ML, most “functional” languages, but not Java, C, etc.
Tail position

Recursive definition of tail position:

– In `(lambda (x1 ... xn) e)`, the body `e` is in tail position.

– If `(if e1 e2 e3)` is in tail position, then `e2` and `e3` are in tail position (but `e1` is not).

– If `(let ([x1 e1] ... [xn en]) e)` is in tail position, then `e` is in tail position (but the binding expressions are not).

Note:

• If a non-lambda expression is not in tail position, then no subexpressions are.

• Critically, in a function call expression `(e1 e2)`, subexpressions `e1` and `e2` are not in tail position.

A tail call is a function call in tail position.

A recursive function is tail-recursive if and only if all of its recursive calls are tail calls.
Tail recursion transformation

Common pattern for transforming naturally recursive functions to tail-recursive form. Works for functions that do commutative operations (order of steps doesn't matter).

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

Base result becomes initial accumulator.

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

Recursive step applied to accumulator instead of recursive result.

Accumulator becomes base result.
Practice: use the transformation

;;; Naturally recursive sum
(define (sum-natural xs)
  (if (null? xs)
      0
      (+ (car xs) (sum-natural (cdr xs))))
)

;;; Tail-recursive sum
(define (sum-tail xs)
  (define (sum-onto xs acc)
    (if (null? xs)
        acc
        (sum-onto (cdr xs) (+ (car xs) acc))))

  (sum-onto xs 0))

)
(order matters)

**Transforming non-commutative steps**

```scheme
(define (reverse-natural-slow xs)
  (if (null? xs)
      null
      (append (reverse-natural-slow (cdr xs))
              (list (car xs)))))
```

```scheme
(define (reverse-tail-just-kidding xs)
  (define (rev xs acc)
    (if (null? xs)
        acc
        (rev (cdr xs) (append acc (list (car xs)))))))
  (rev xs null))
```

```scheme
(define (reverse-tail-slow xs)
  (define (rev xs acc)
    (if (null? xs)
        acc
        (rev (cdr xs) (append (list (car xs)) acc))))
  (rev xs null))
```

Tail Recursion
The transformation is not always ideal.

• Append-based recursive reverse is $O(n^2)$: each recursive call must traverse to end of list and build a fully new list.
  – $1 + 2 + \ldots + (n-1)$ is almost $n \times n / 2$
  – Moral: beware append, especially within recursion
• Tail-recursive reverse can avoid append in $O(n)$.
  – Cons is $O(1)$, done $n$ times.

```
(define (reverse-tail-slow xs)
  (define (rev xs acc)
    (if (null? xs)
        acc
        (rev (cdr xs) (append (list (car xs)) acc))))
  (rev xs null))

(define (reverse-tail-good xs)
  
  ...
```

$O(n^2)$
Tail recursion ≠ accumulator pattern

(define (even n)
  (cond [(= 0 n) #t]
        [(= 1 n) #f]
        [#t (even (- n 1))]))

• Tail recursion and the accumulator pattern are commonly used together. They are not synonyms.
  – Natural recursion may use an accumulator.
  – Tail recursion does not necessarily involve an accumulator.
Why tail recursion instead of loops with mutation?

1. Simpler language, but just as efficient.
2. Explicit dependences for easier reasoning.
   – Especially with HOFs like fold!
Identify dependences between ________. 

Racket: immutable natural recursion

```racket
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Racket: immutable tail recursion

```racket
(define (fib n)
  (define (fib-tail n fibi fibi+1)
    (if (= 0 n)
      fibi
      (fib-tail (- n 1) fibi+1 (+ fibi fibi+1)))
  (fib n 0 1))
```

Python: loop iteration with mutation

```python
def fib(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
```
Identify dependences between ________.

Python: loop iteration with mutation
```
def fib(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
```

Racket: immutable natural recursion
```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Racket: immutable tail recursion
```
(define (fib n)
  (define (fib-tail n fibi fibi+1)
    (if (= 0 n)
        fibi
        (fib-tail (- n 1) fibi+1 (+ fibi fibi+1)))
  (fib n 0 1))
```

Tail Recursion 21
Fold: iterator over recursive structures
(a.k.a. reduce, inject, ...)

(fold_ combine init list)
accumulates result by iteratively applying
(combine element accumulator)
to each element of the list and accumulator so far
(starting from init) to produce the next accumulator.

- (foldr f init (list 1 2 3))
  computes (f 1 (f 2 (f 3 init)))

- (foldl f init (list 1 2 3))
  computes (f 3 (f 2 (f 1 init)))
Folding geometry

Natural recursion

\[(\text{foldr } \text{combine } \text{init } L)\]

Tail recursion

\[(\text{foldl } \text{combine } \text{init } L)\]
Fold code: tail-practice.rkt

- foldr implementation
- foldl implementation
- using foldr/foldl
- bonus mystery folding puzzle
Super-iterators!

• Not built into the language
  – Just a programming pattern
  – Many languages have built-in support, often allow stopping early without resorting to exceptions

• Pattern separates recursive traversal from data processing
  – Reuse same traversal, different folding functions
  – Reuse same folding functions, different data structures
  – Common vocabulary concisely communicates intent

• map, filter, fold + closures/lexical scope = superpower
  – Later: argument function can use any “private” data in its environment.
  – Iterator does not have to know or help.