Topics

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

\[
\begin{align*}
\text{(define (fact n)} & \text{)} \\
\text{  \quad (if (= n 0))} & \text{1} \\
\text{  \quad (* n (fact (- n 1)))))}
\end{align*}
\]

How efficient is this implementation?

Space: \(O(n)\)

Time: \(O(n)\)
Evaluation example

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

Call stacks at each step

<table>
<thead>
<tr>
<th>(fact 3)</th>
<th>(fact 3): 3*_</th>
<th>(fact 3): 3*_</th>
<th>(fact 3): 3*_</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(fact 2)</td>
<td>(fact 2): 2*_</td>
<td>(fact 2): 2*_</td>
</tr>
<tr>
<td></td>
<td>(fact 1)</td>
<td>(fact 1): 1*_</td>
<td>(fact 1): 1*1</td>
</tr>
<tr>
<td></td>
<td>(fact 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Space: \( O() \)

Time: \( O() \)

Tail recursive factorial

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

Base case returns full result.

Accumulator parameter provides result so far.

Recursive case returns full result.

Accumulator parameter provides result so far.

Initial accumulator provides base result.

Common patterns of work

Natural recursion:
- Argument
- Full result
- Reduce argument
- Accumulate result so far

Tail recursion:
- Argument
- Full result
- Reduce argument
- Accumulate result so far
Natural recursion

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

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

Tail recursion

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

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

Evaluation example

Call stacks at each step

\[
\begin{align*}
\text{(fact 3) } & \quad \text{(fact 3): } & \quad \text{(fact 3): } & \quad \text{(fact 3): } \\
\text{(ft 3 1) } & \quad \text{(ft 3 1): } & \quad \text{(ft 3 1): } & \quad \text{(ft 3 1): } \\
\text{ft } = \text{ fact-tail} & \quad \text{(ft 2 3): } & \quad \text{(ft 2 3): } & \quad \text{(ft 2 3): } \\
\text{(fact 0) } & \quad \text{(fact 0) } & \quad \text{(fact 0) } & \quad \text{(fact 0) } \\
\text{(ft 0 6) } & \quad \text{(ft 0 6): } & \quad \text{(ft 0 6): } & \quad \text{(ft 0 6): } \\
\end{align*}
\]

Tail-call optimization

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

\[
\begin{align*}
\text{(fact 3) } & \quad \text{(ft 3 1) } & \quad \text{(ft 2 3) } & \quad \text{(ft 1 6) } & \quad \text{(ft 0 6)} \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{etc.}
\end{align*}
\]

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

Recursive definition of tail position:
- \( \text{In } (\lambda (x_1 \ldots x_n) \, e), \text{the body } e \text{ is in tail position.} \)
- If \((\text{if } e_1 \, e_2 \, e_3)\) is in tail position, then \(e_2 \text{ and } e_3\) are in tail position (but \(e_1\) is not).
- If \((\text{let } ([x_1 \, e_1] \ldots [x_n \, e_n]) \, e)\) 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 \((e_1 \, e_2)\), subexpressions \(e_1 \text{ and } e_2\) 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.

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-tail-onto xs acc)
    (if (null? xs)
      acc
      (sum-tail-onto (cdr xs) (+ (car xs) acc)))))
```

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).

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

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

Transforming non-commutative steps

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

;; Tail-recursive reverse: "just kidding"
(define (rev xs acc)
  (if (null? xs)
    acc
    (rev (cdr xs) (append acc (list (car xs))))))

;; Tail-recursive reverse: "slow"
(define (rev xs acc)
  (if (null? xs)
    acc
    (rev (cdr xs) (append (list (car xs)) acc))))
```

(order matters)

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

;; Tail-recursive reverse: "just kidding"
(define (rev xs acc)
  (if (null? xs)
    acc
    (rev (cdr xs) (append acc (list (car xs))))))

;; Tail-recursive reverse: "slow"
(define (rev xs acc)
  (if (null? xs)
    acc
    (rev (cdr xs) (append (list (car xs)) acc))))
```
The transformation is not always ideal.

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

(define (reverse-tail-good xs)
  (rev xs null))
```

Tail recursion ≠ accumulator pattern

```scheme
(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 ________.

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

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

(define fibi)
```

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 + fib_i_prev
    return fib_i
```
Identify dependences between ________.

Racket: immutable natural recursion

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

Racket: immutable tail recursion

```scheme
(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
```

Fold: iterator over recursive structures
(a.k.a. reduce, inject, …)

```scheme
(fold_ combine init list)
```

accumulates result by iteratively applying

```scheme
(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)))`

Fold code: tail.rkt

- foldr implementation
- foldl implementation
- using foldr/foldl
- bonus mystery folding puzzle

Folding geometry

Natural recursion

```
(foldr combine init L)
```

Tail recursion

```
(foldl combine init L)
```

Fold code: tail.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.