Tail Recursion

Naturally recursive factorial

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

How efficient is this implementation?

Space: \(O(\quad)\)

Time: \(O(\quad)\)

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
### Tail Recursion Example

Evaluation example:

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

Call stacks at each step:

- `(fact 3)`
- `(fact 2)`
- `(fact 1)`
- `(fact 0)`

Remember: `n ↦ 2`; and “rest of function” for this call.

Space: `O( )`

Time: `O( )`

### Naturally Recursive Factorial

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

Base case returns base result. Recursive case returns result so far.

Compute result so far after/from recursive call.

Space: `O( )`

Time: `O( )`

### Tail Recursive Factorial

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

Initial accumulator provides base result.

Accumulator parameter provides result so far.

Base case returns full result.

Recursive case returns full result.

Compute remaining argument before/for recursive call.

### Common Patterns of Work

#### Natural Recursion:
- Argument
- Full result

#### Tail Recursion:
- Argument
- Base result

- Reduce argument
- Accumulate result so far
- Base case
- Full result
Natural recursion

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

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

Tail recursion

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

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

Evaluation example

Call stacks at each step

Nothing useful remembered here.

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

Tail-call optimization

Space: O(  )

Time: O(  )

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 function is tail-recursive if it uses a recursive tail call.

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)
  ;\text{Transforming non-commutative steps}
  (define (reverse-natural-slow xs)
    (if (null? xs)
        null
        (append (reverse-natural-slow (cdr xs)) (list (car xs))))
  (define (reverse-tail-just-kidding xs)
    (define (rev xs acc)
      (if (null? xs)
          acc
          (rev (cdr xs) (append acc (list (car xs)))))))
  (define (reverse-tail-slow xs)
    (define (rev xs acc)
      (if (null? xs)
          acc
          (rev (cdr xs) (append (list (car xs)) acc))))
    (rev xs null)))
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))))
  (rev xs null))

(define (reverse-tail-good xs)
  ; mutually tail recursive
  (define (even n)
    (or (zero? n) (odd (- n 1))))
  (define (odd n)
    (or (not (zero? n)) (even (- n 1))))

  ; tail recursive
  (define (even2 n)
    (cond [[= 0 n] #t]
          [[= 1 n] #f]
          [#t (even2 (- n 2))]))

  (even n)
```

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!

Tail recursion ≠ accumulator pattern

```
; mutually tail recursive
(define (even n)
  (or (zero? n) (odd (- n 1))))
(define (odd n)
  (or (not (zero? n)) (even (- n 1))))

; tail recursive
(define (even2 n)
  (cond [[= 0 n] #t]
        [[= 1 n] #f]
        [#t (even2 (- n 2))]))
```

- 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.

Identify dependences between ________.

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

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

```
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  ; Python: loop iteration with mutation
```

7) 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!

Tail recursion ≠ accumulator pattern

```
; mutually tail recursive
(define (even n)
  (or (zero? n) (odd (- n 1))))
(define (odd n)
  (or (not (zero? n)) (even (- n 1))))

; tail recursive
(define (even2 n)
  (cond [[= 0 n] #t]
        [[= 1 n] #f]
        [#t (even2 (- n 2))]))
```

- 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.

Identify dependences between ________.

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

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

```
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  ; Python: loop iteration with mutation
```
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))
```

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

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
```

---

**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

```
(foldr Combine Init L)
```

```
result Combine Combine ... Combine Combine Init
L ➔ v1 ➔ v2 ➔ ... ➔ vn-1 ➔ vn
Init Combine Combine ... Combine Combine Result
```

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.