Tail Recursion
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

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

Space: \( O(\ ) \)
Time: \( O(\ ) \)

How efficient is this implementation?
CS 240-style machine model

- **Registers**: fixed size, general purpose
- **Code**: Call frame
- **Stack**: Call frame
  - arguments, variables, return address per function call
- **Heap**: cons cells, data structures, ...
Example

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

\[
(\text{fact } 3) \quad (\text{fact } 3): 3* \quad (\text{fact } 3): 3* \quad (\text{fact } 3): 3* \\
(\text{fact } 2) \quad (\text{fact } 2): 2* \quad (\text{fact } 2): 2* \\
(\text{fact } 1) \quad (\text{fact } 1): 1* \\
(\text{fact } 0) : 1
\]

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

Space: \( O( ) \)

Time: \( O( ) \)
Naturally recursive factorial

\[
\text{(define (fact n)}
\]

\[
\text{(if (= n 0)}
\]

\[
1
\]

\[
(* n (fact (- n 1)))))
\]

Base case returns base result.

Recursive case returns result so far.

Compute result so far after/from recursive call.

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 result so far before/for recursive call.**
- **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

Deeper recursive calls

Base case
Base result
Base case
Full result
Natural recursion

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

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

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

\[
\begin{align*}
(fact-tail 4, 1): & 24 \\
(fact-tail 3, 4): & 24 \\
(fact-tail 2, 12): & 24 \\
(fact-tail 1, 24): & 24 \\
(fact-tail 0, 24): & 24
\end{align*}
\]

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

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

\[
\begin{align*}
& (\text{fact 3}) \_ \\
& (\text{fact 3}) \_ \\
& (\text{fact 3}) \_ \\
& (\text{fact 3}) \_ \\
& (\text{fact 3}) \_ \\
& (\text{fact 3}) \_ \\
& (\text{fact 3}) \_ \\
& \text{Nothing useful remembered here.} \\
& \text{etc.}
\end{align*}
\]
Optimization under the hood

```
(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 recursion transformation

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

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

Base result becomes initial accumulator.

Recursive step applied to accumulator instead of recursive result.
Example

(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs))))
)

(define (sum xs)
  (define (sum-tail xs acc)
    (if (null? xs)
      acc
      (sum-tail (cdr xs) (+ (car xs) acc))))
  (sum-tail xs 0))
Practice

Naturally recursive `rev` is $O(n^2)$: each recursive call must traverse to end of list and build a fully new list.
- $1+2+...+(n-1)$ is almost $n^2/2$
- Moral: beware append, especially within outer recursion

Tail-recursive `rev` is $O(n)$.
- Cons is $O(1)$, done $n$ times.

What about `map`, `filter`?
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*. 

Tail call intuition: “nothing left for caller to do”, “callee result is immediate caller result”
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
(define (fib n) Racket: immutable natural recursion
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2))))))
```

```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
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

Racket: immutable tail recursion

recursive calls

loop iterations
What must we inspect to 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 19
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

\[(\text{foldl} \ \text{combine} \ \text{init} \ L)\]
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
  – Next: argument function can use any “private” data in its environment.
  – Iterator does not have to know or help.