Tail Recursion

+tail.rkt

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

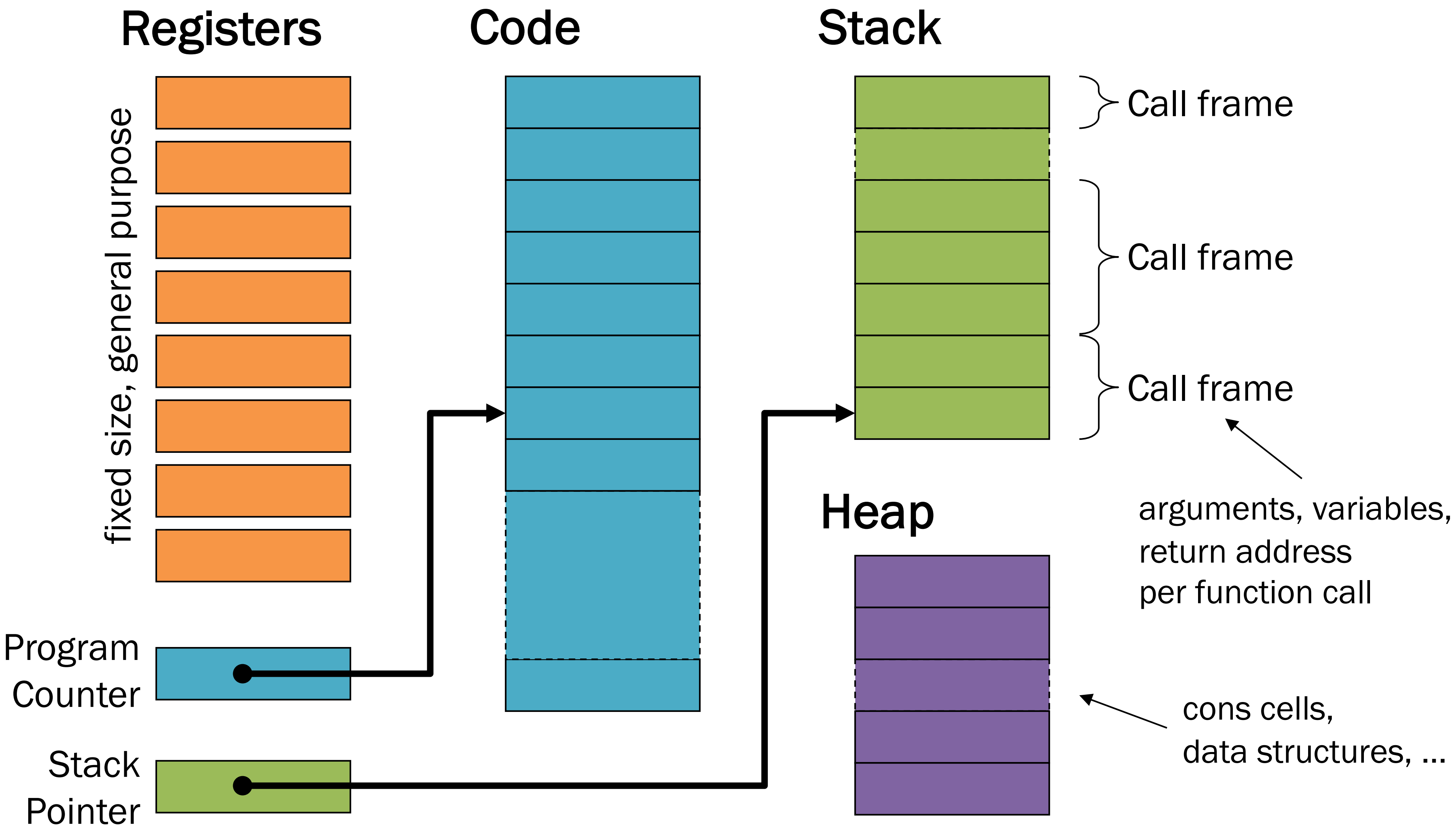
```
(define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
```

How efficient is this implementation?

Space: O(   )

Time: O(   )

CS 240-style machine model



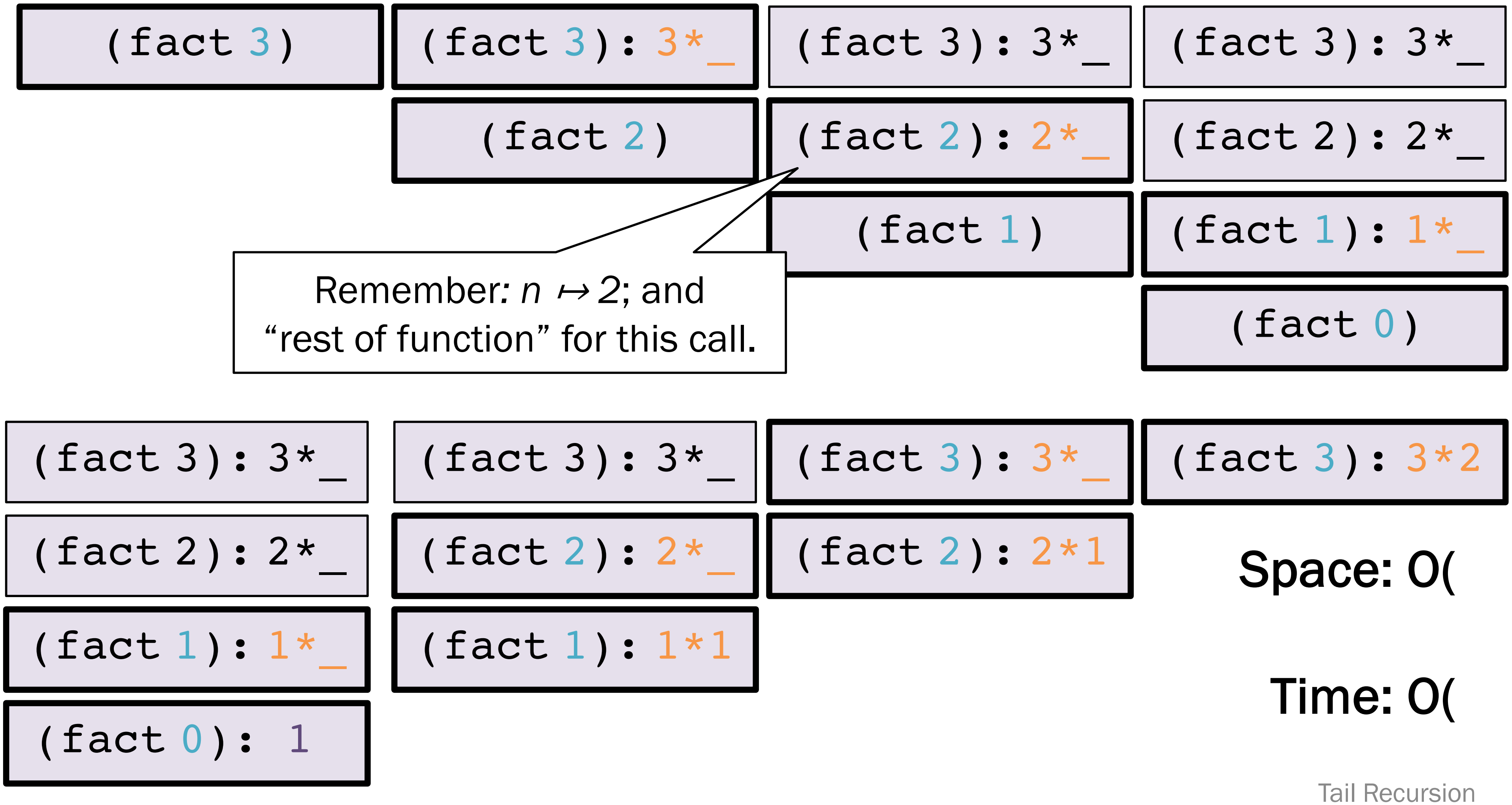
arguments, variables,
return address
per function call

cons cells,
data structures, ...

Evaluation example

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

Call stacks at each step



```
(fact 3)
```

```
(fact 3): 3*_
(fact 2)
```

```
(fact 3): 3*_
(fact 2): 2*_
(fact 1)
```

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

```
(fact 3): 3*_
(fact 2): 2*_
(fact 1): 1*_
(fact 0)
```

```
(fact 3): 3*_
(fact 2): 2*_
(fact 1): 1*_
(fact 0):  1
```

```
(fact 3): 3*_
(fact 2): 2*_
(fact 1): 1*1
```

```
(fact 3): 3*_
(fact 2): 2*1
```

```
(fact 3): 3*2
```

Space: O(   )

Time: O(   )

Naturally recursive factorial

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

Compute result so far **after/from** recursive call.

Base case returns base result.

Recursive case returns result so far.

Compute remaining argument **before/for** recursive call.

Tail recursive factorial

Accumulator parameter provides result so far.

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

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

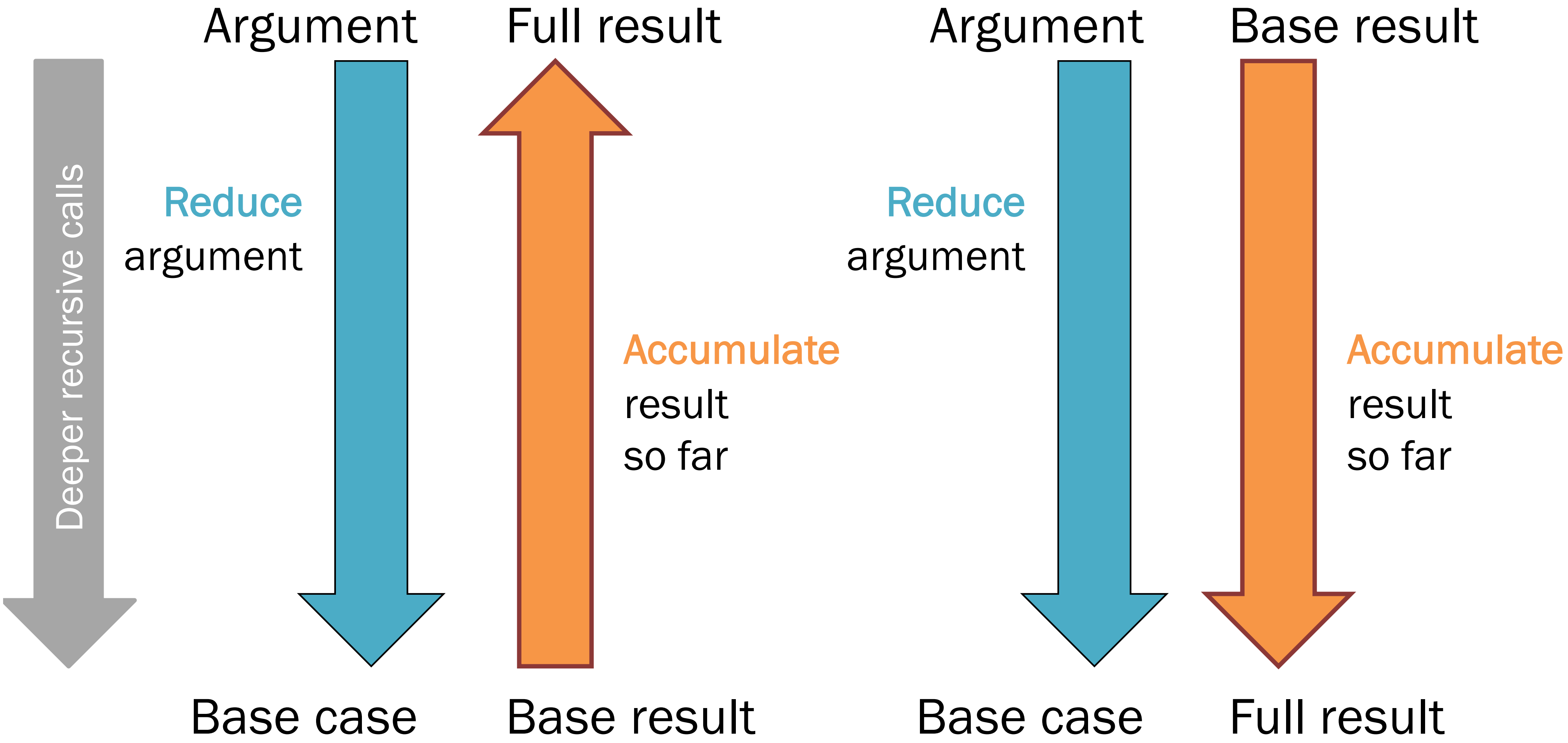
Base case returns full result.

Recursive case returns full result.

Compute remaining argument **before/for** recursive call.

Initial accumulator provides base result.

Common patterns of work



Deeper recursive calls

Natural recursion:

Argument

Reduce argument

Base case

Full result

Accumulate result so far

Base result

Tail recursion:

Argument

Reduce argument

Base case

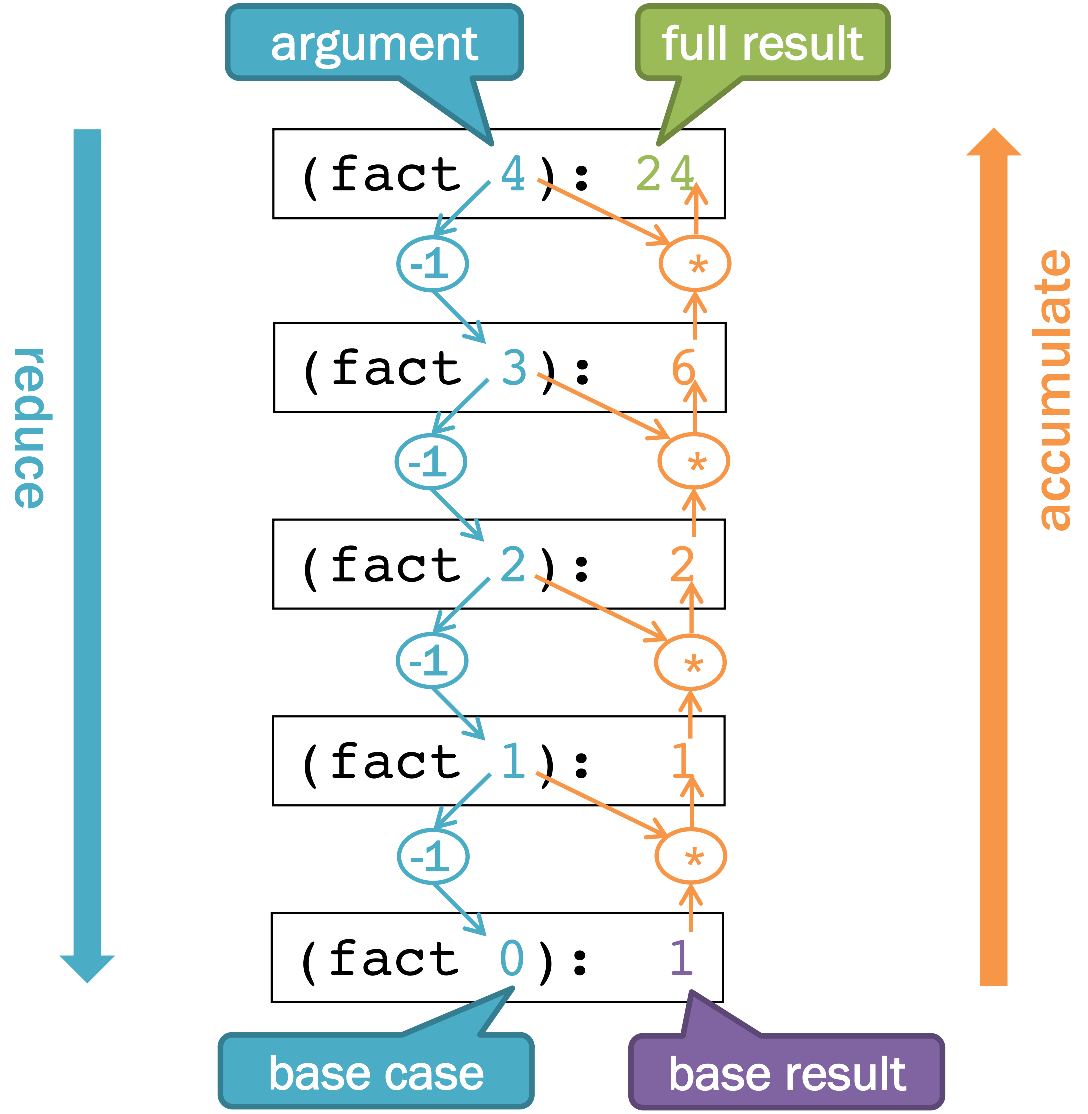
Base result

Accumulate result so far

Full result

Natural recursion

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



argument

full result

reduce

accumulate

(fact 4): 24

-1

*

(fact 3): 6

-1

*

(fact 2): 2

-1

*

(fact 1): 1

-1

*

(fact 0): 1

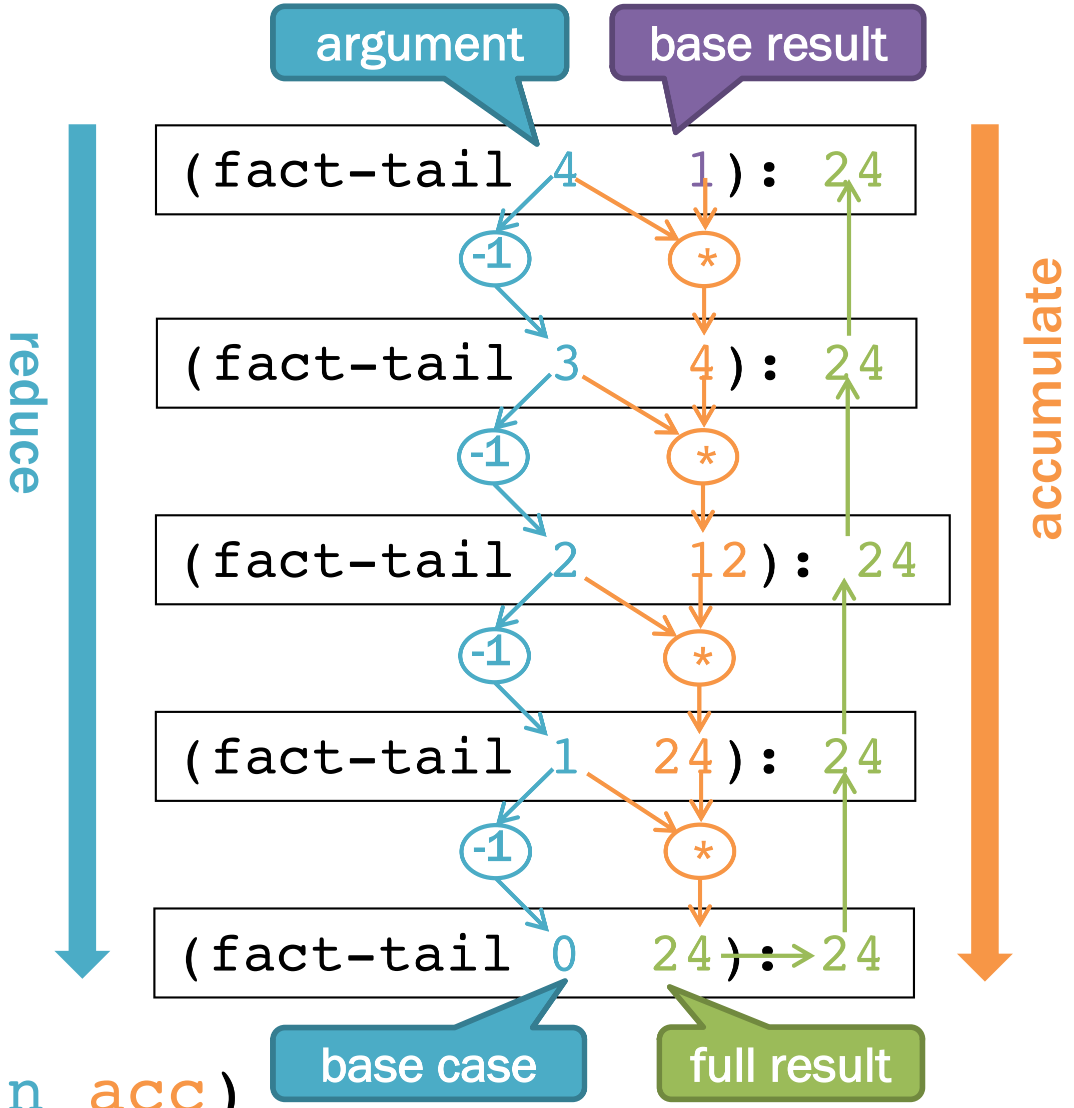
base case

base result

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

Tail recursion

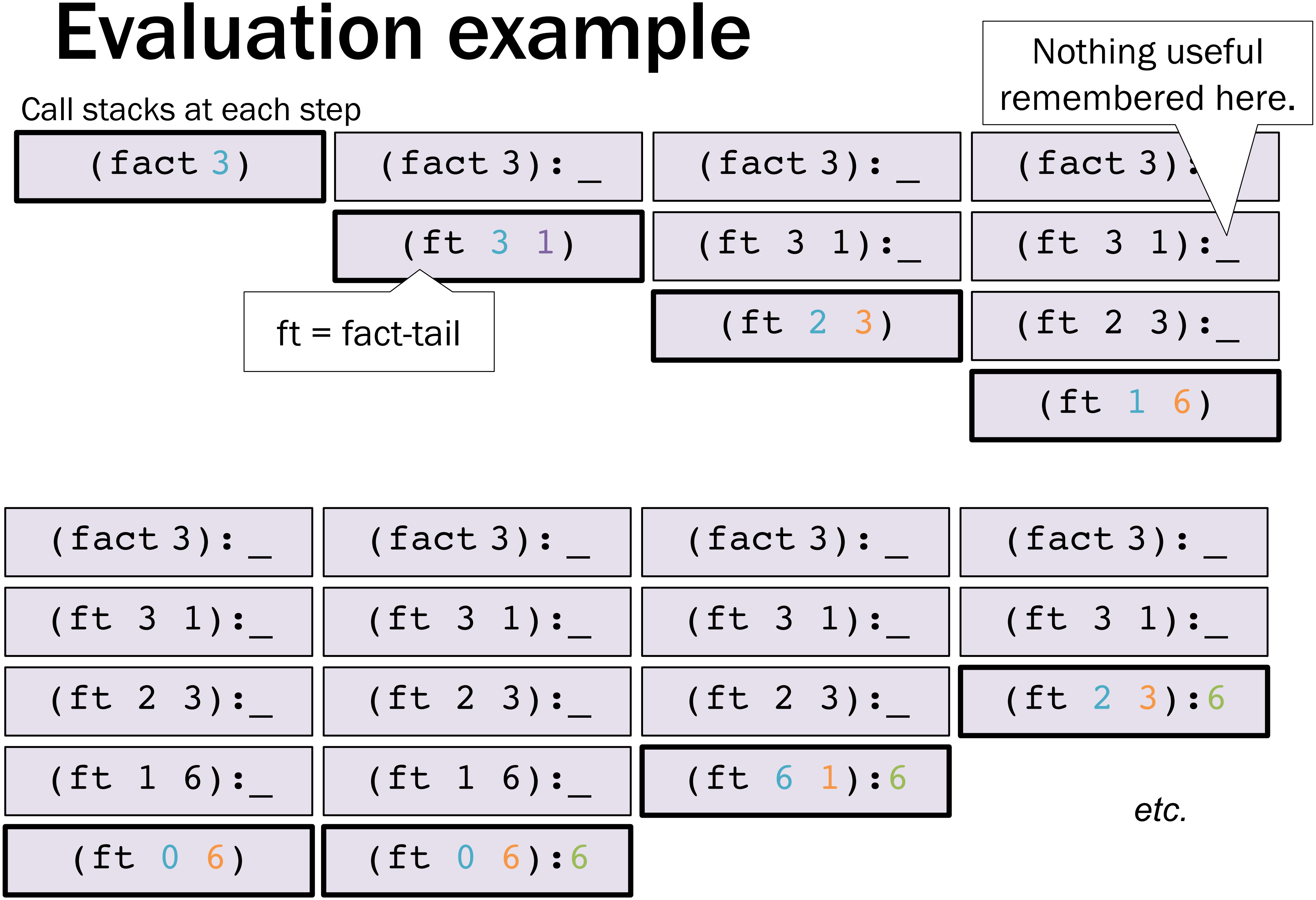
Recursive case:
Compute recursive argument
in terms of argument and
accumulator.



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

Evaluation example

Call stacks at each step



(fact 3)

(fact 3): _
(ft 3 1)

ft = fact-tail

(fact 3): _
(ft 3 1):_
(ft 2 3)

(fact 3): _
(ft 3 1):_
(ft 2 3):_
(ft 1 6)

Nothing useful remembered here.

(fact 3): _
(ft 3 1):_
(ft 2 3):_
(ft 1 6):_
(ft 0 6)

(fact 3): _
(ft 3 1):_
(ft 2 3):_
(ft 1 6):_
(ft 0 6):6

(fact 3): _
(ft 3 1):_
(ft 2 3):_
(ft 6 1):6

(fact 3): _
(ft 3 1):_
(ft 2 3):6

etc.

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))
```

Space: O(   )

Time: O(   )

`(fact 3)` `(ft 3 1)` `(ft 2 3)` `(ft 1 6)` `(ft 0 6)`

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

Tail call intuition:
"nothing left for caller to do after call",
"callee result is immediate caller result"

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:

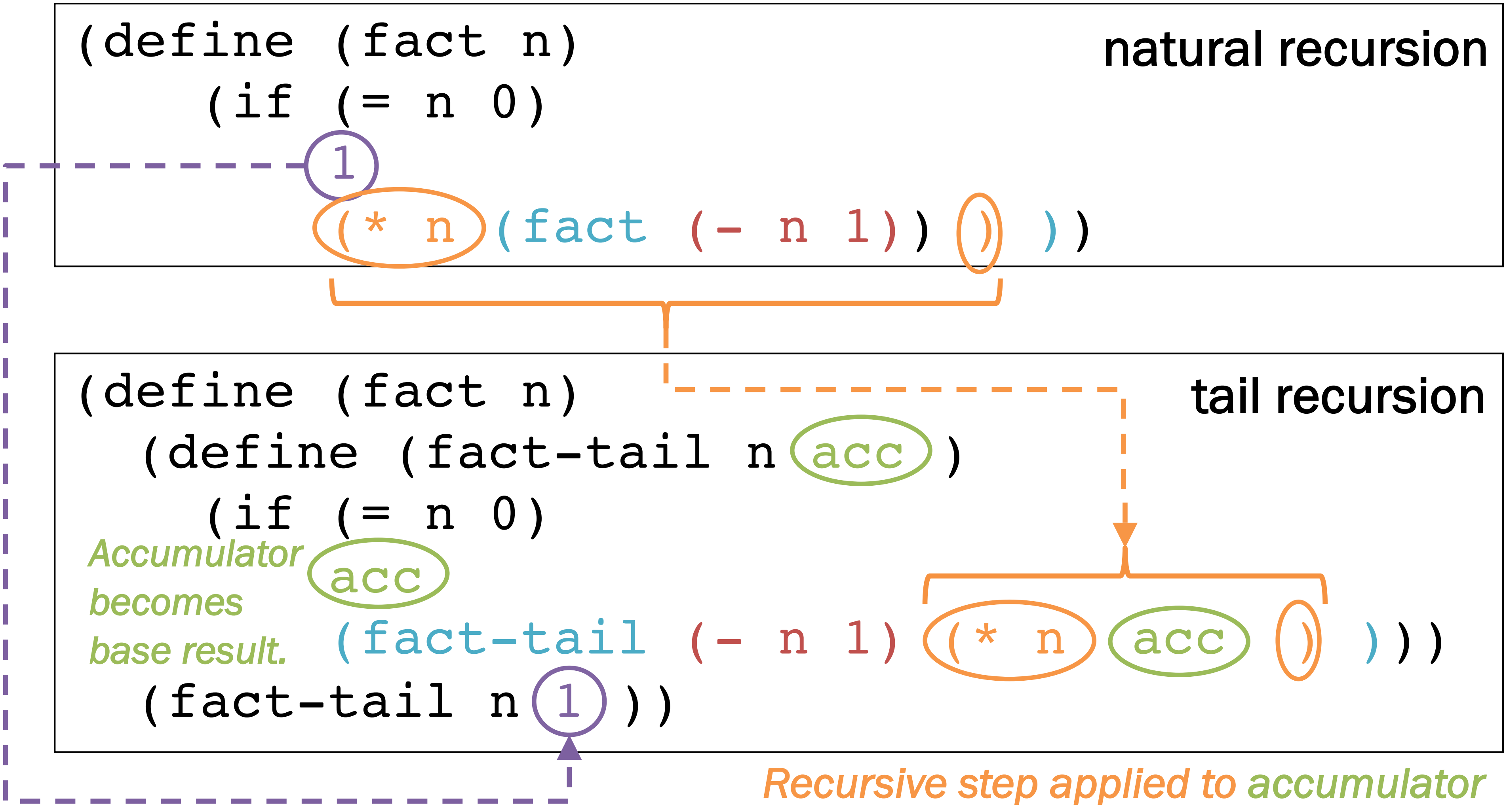
- If a non-lambda expression is not in tail position, then no subexpressions are.
- Critically, in a function call expression `(e1 e2)`,
  subexpressions `e1` and `e2` are **not** in tail position.

A ***tail call*** is a function call in *tail position.*

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

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).



natural recursion

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

tail recursion

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

Accumulator becomes base result.

Base result becomes initial accumulator.

Recursive step applied to accumulator instead of recursive 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)
```

(order matters)

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))))))
  (rev xs null))
```

✗

```
(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.

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

O(   )

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

What about map, filter?

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.

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

```
(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))
```

recursive calls

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
```

loop iterations

What must we inspect to

Identify dependences between ________.

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))
```

recursive calls

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
```

loop iterations

HOF HOF

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`.

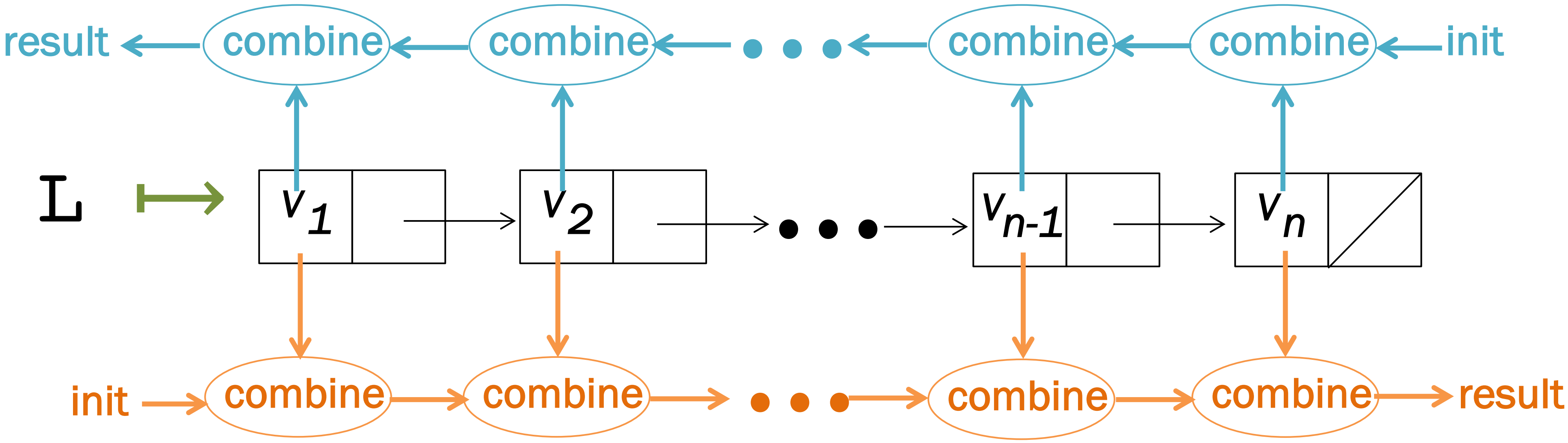
- `(foldr f init (list 1 2 3))`
  computes `(f 1 (f 2 (f 3 init)))`

- `(foldl f init (list 1 2 3))`
  computes `(f 3 (f 2 (f 1 init)))`

Folding geometry

Natural recursion

`(foldr combine init L)`



`(foldl combine init L)`

Tail recursion

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.