WELLESLEY

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

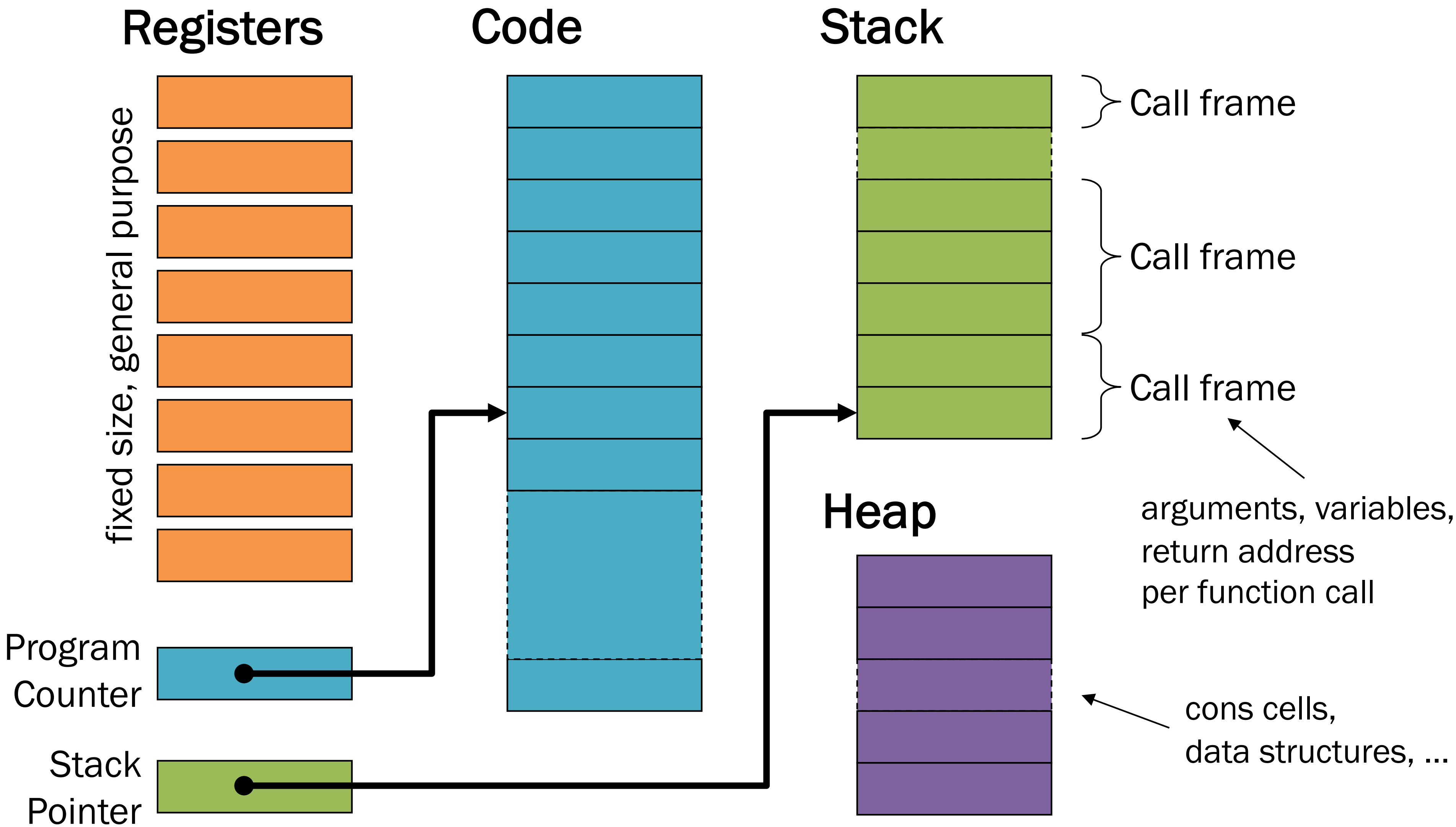
```
(define (fact n)
     (if (= n 0)
         1
         (* n (fact (- n 1)))))
```

How efficient is this implementation?

Space: O(   )

Time: O(   )

CS 240-style machine model



Example

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

(fact 3)

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

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

Remember: $n \mapsto 2$; 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)))))
```

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))
```

Accumulator parameter provides result so far.

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

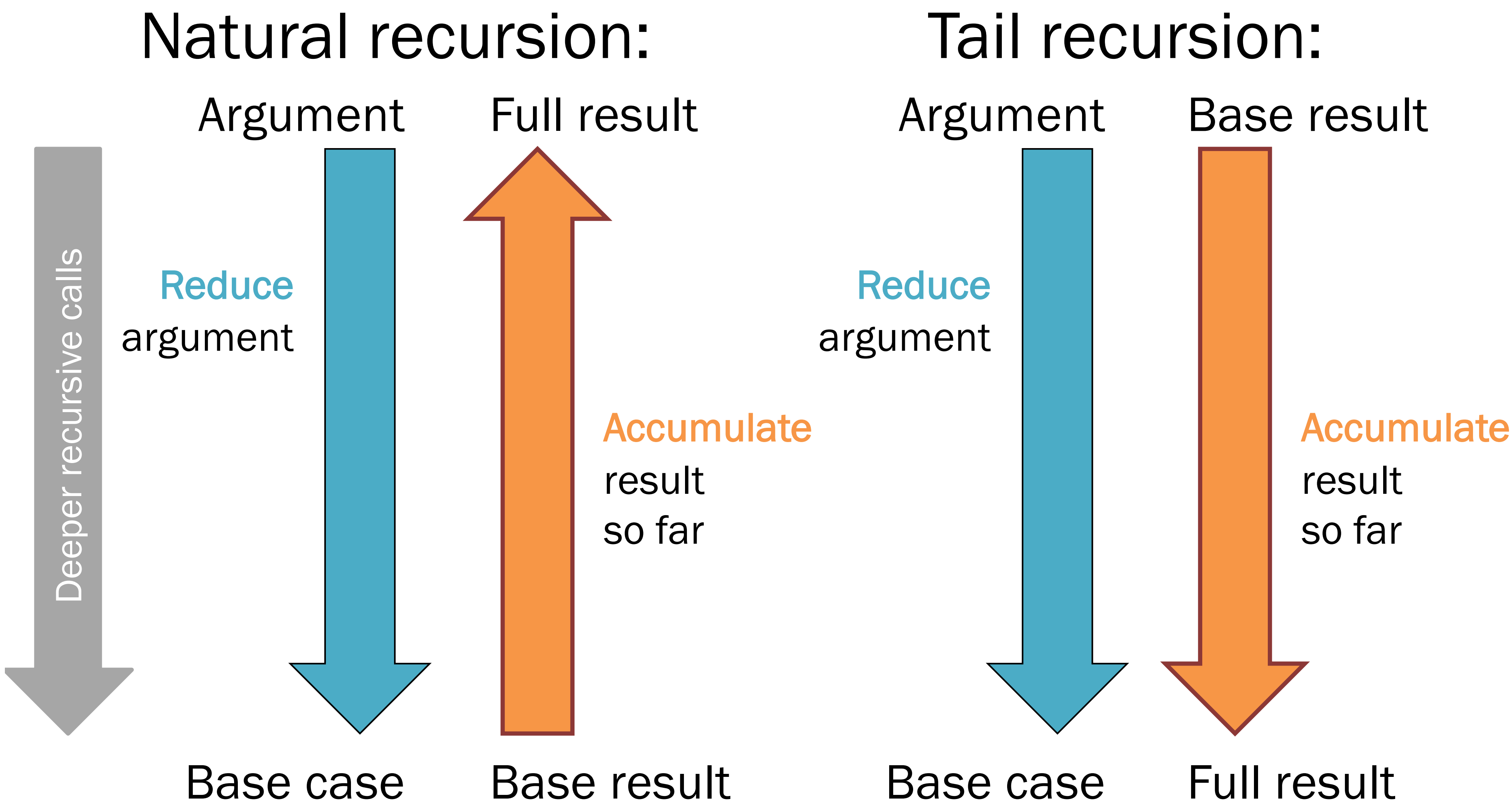
Base case returns full result.

Recursive case returns full result.

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

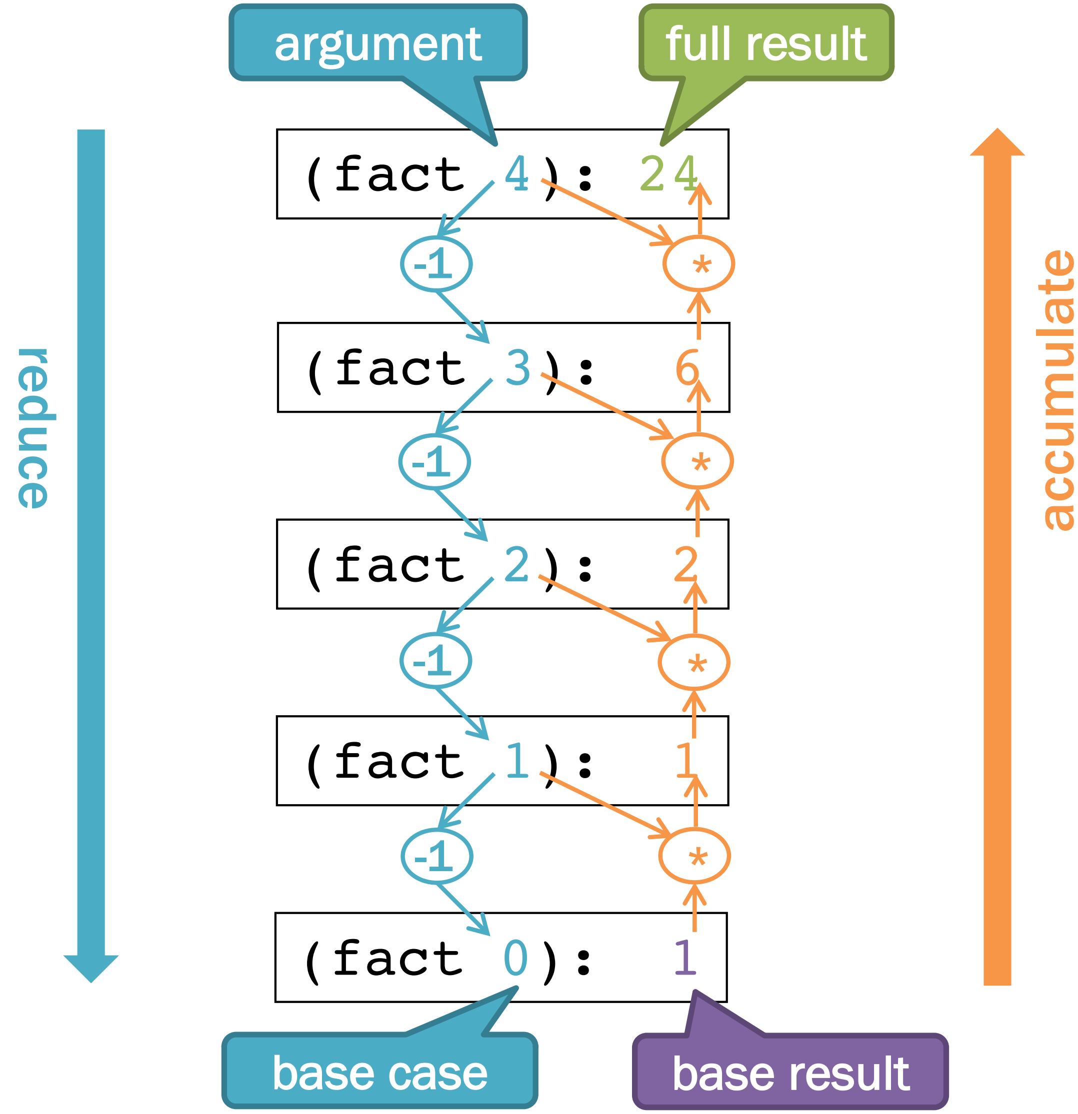
Initial accumulator provides base result.

Common patterns of work



Natural recursion

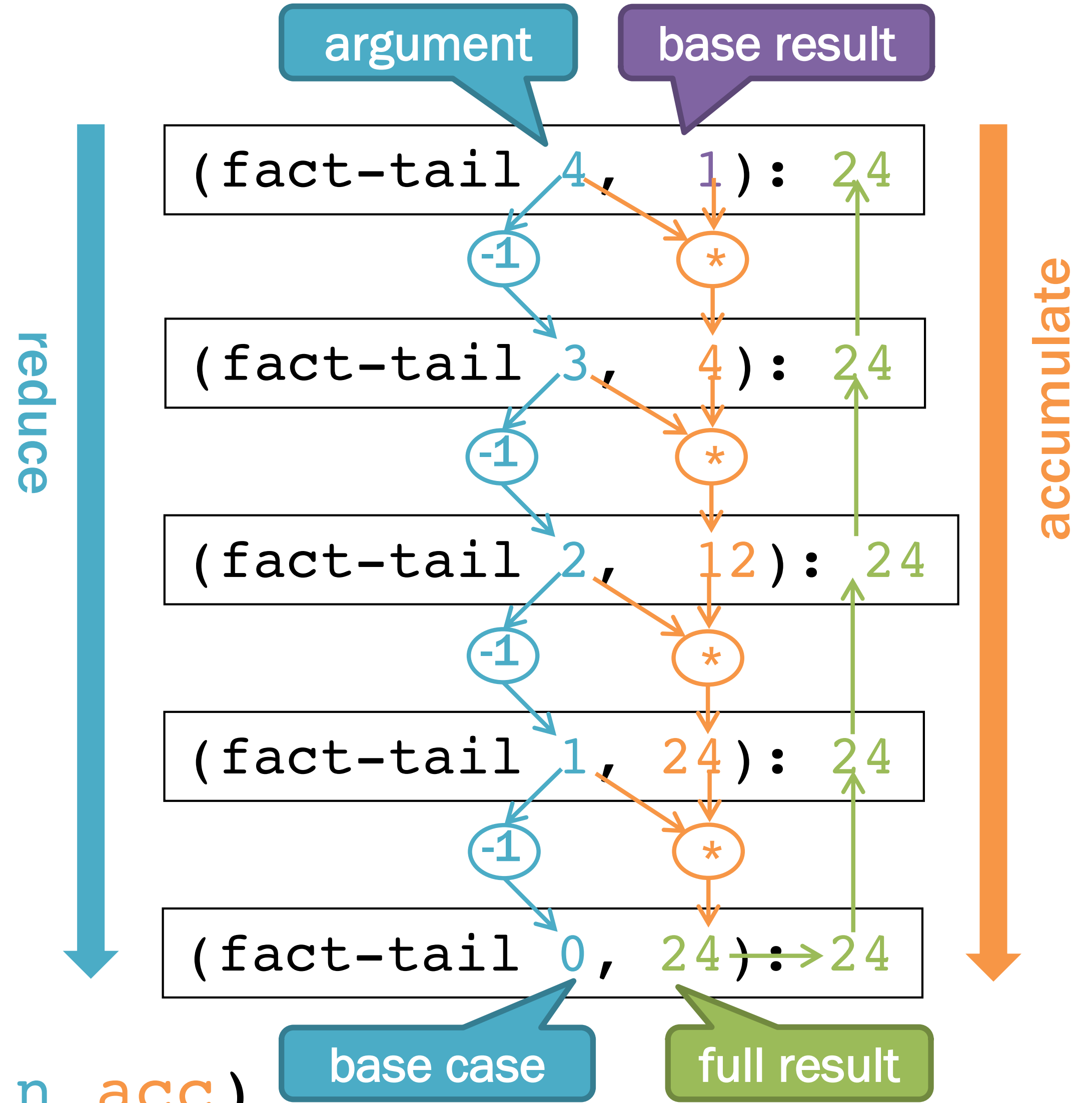
Recursive case:
Compute result
in terms of argument and
accumulated recursive result.




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

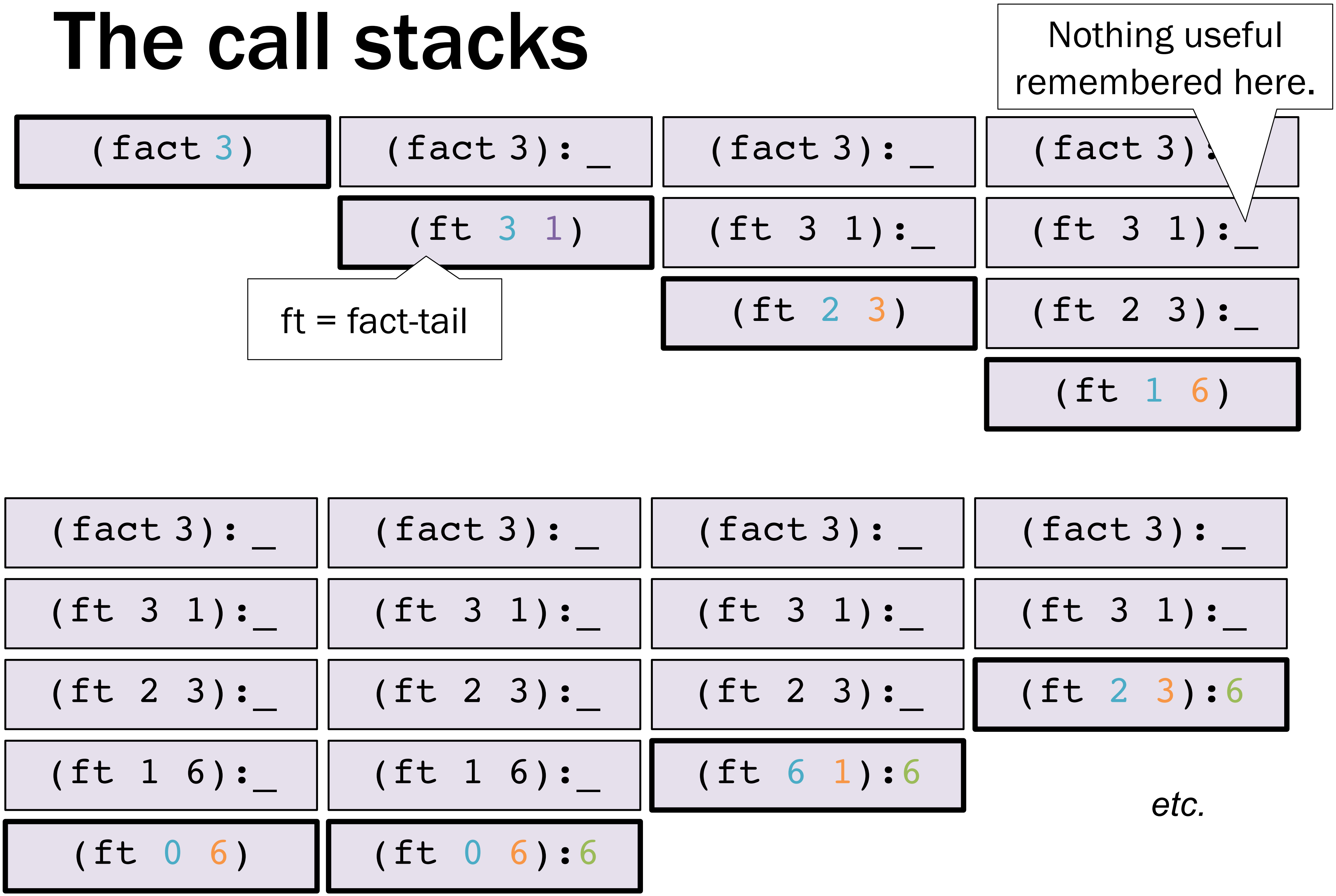
Tail recursion

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))
```

The call stacks



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))
```

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 recursion transformation

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.

Recursive step applied to accumulator instead of recursive result.

Base result becomes initial accumulator.

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

```
(define (rev xs)
```

```
(define (rev xs)
```

- 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*n/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 call intuition:
“nothing left for caller to do”,
“callee result is immediate caller result”

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.*

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)))))
```

recursive calls

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))
```

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)))))
```

recursive calls

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))
```

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`.

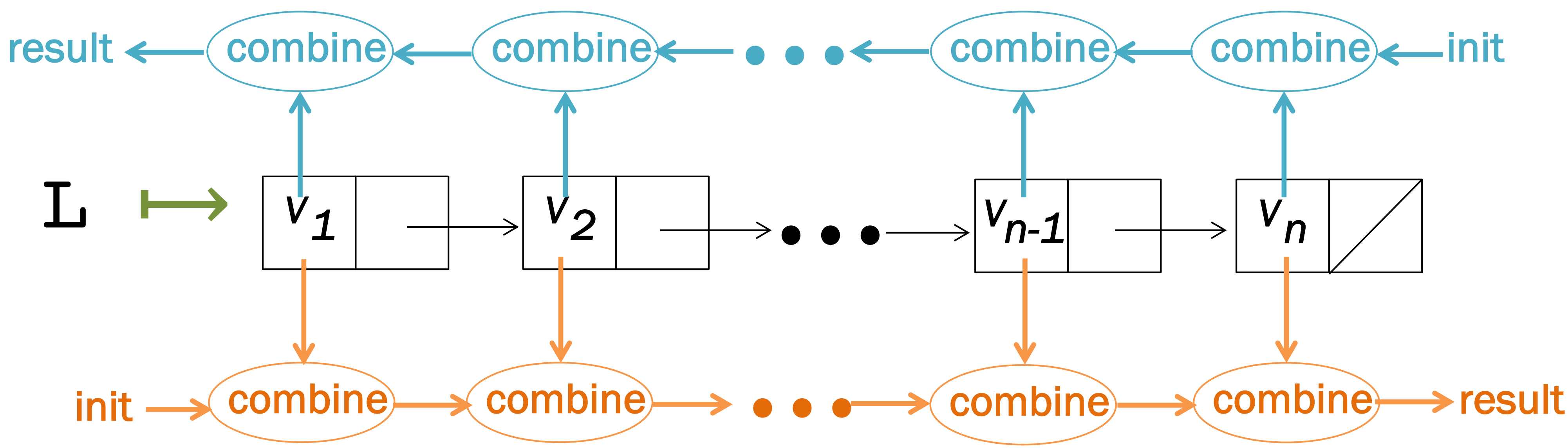
- `(foldr f init (list 1 2 3))`
  computes `(f 1 (f 2 (f 3 init)))`

- `(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

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.