



Tail Calls and Tail Recursion

CIS352 — Spring 2021

Kris Micinski



((lambda (x) x) ((lambda (y) y) 5))

((lambda (x) x) 5)

5

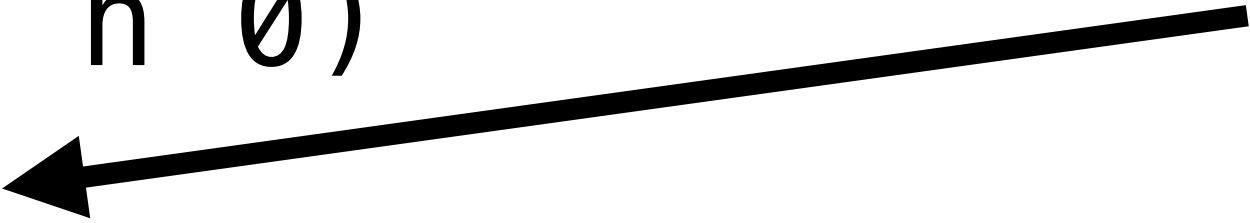
Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

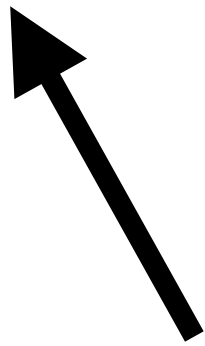
Defines **base case**



Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

and **inductive / recursive** case



Calculating factorial in Racket

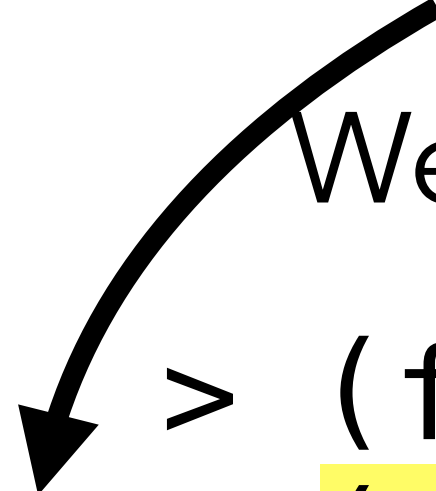
```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as “substitution”

```
> (factorial 2)
```

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as "substitution"

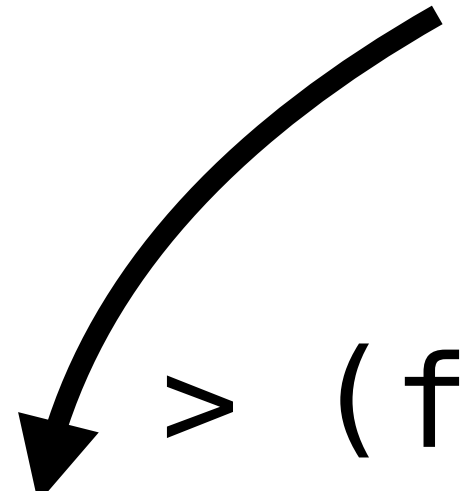


```
> (factorial 2)
= (if (= 2 0)
      1
      (* 2 (factorial (sub1 2))))
```

Copy defn, substitute for argument n

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

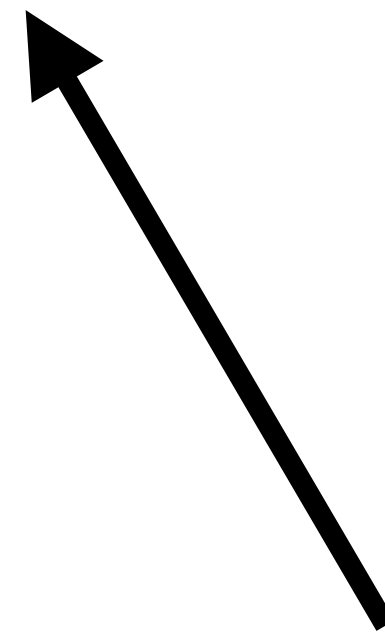
We can think of recursion as "substitution"



```
> (factorial 2)
= (if (= 2 0)
      1
      (* 2 (factorial (sub1 2))))
= (if #f 1 (* 2 (factorial (sub1 2))))
= (* 2 (factorial (sub1 2)))
= (* 2 (factorial 1))
= (* 2 (if ...))
```



```
...  
= (* 2 (if (= 2 0)  
           1  
           (* n (factorial (sub1 2)))))  
= (* 2 (factorial 1))  
= ...  
= (* 2 (* 1 1))  
= (* 2 1)  
= 2
```



Notice we're building a big stack of calls to *

Tail Calls

- Unlike calls in general, ***tail calls*** do not affect the stack:
 - Tail calls *do not grow* (or shrink) the stack.
 - They are more like a goto/jump than a normal call.

Tail Position

- A subexpression is in ***tail position*** if it's:
 - The last subexpression to run, whose return value is also the value for its parent expression
 - In `(let ([x rhs]) body)`; body is in *tail position*...
 - In `(if grd thn els)`; thn & els are in *tail position*...

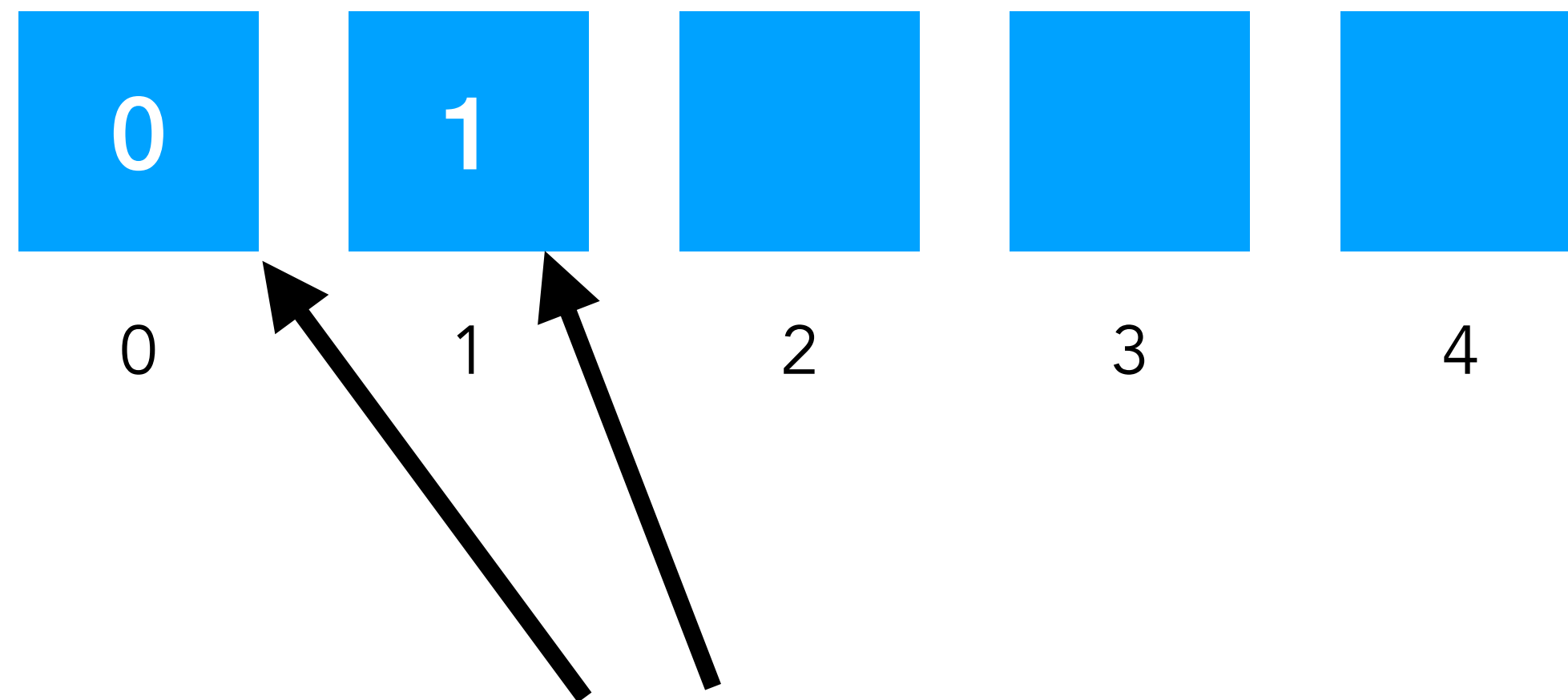
Tail Recursion

- A function is ***tail recursive*** if all recursive calls in tail position
- Tail-recursive functions are analogous to loops in imperative langs

Tail calls / tail recursion

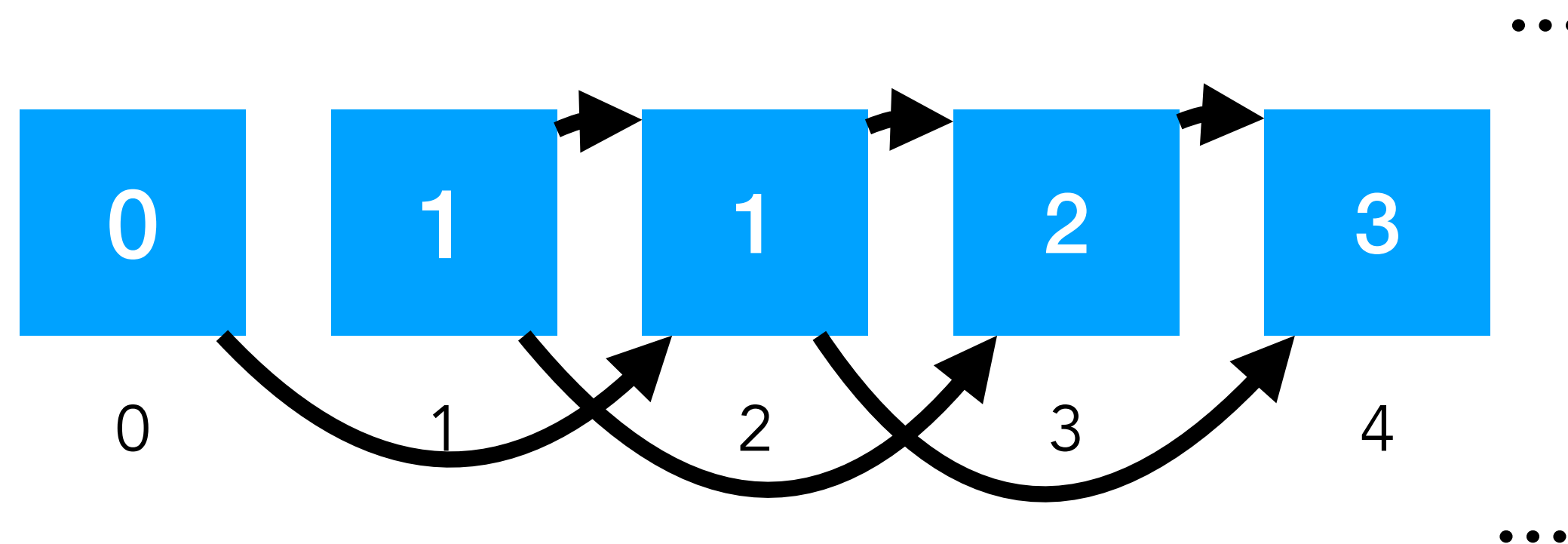
- Unlike calls in general, **tail calls** do not affect the stack:
 - Tail calls *do not grow* (or shrink) the stack.
 - They are more like a goto/jump than a normal call.
- A function is **tail recursive** if all recursive calls in tail position
- Tail-recursive functions are analogous to loops in imperative langs

Instead, use ***dynamic programming***:
design a recursive solution top-down, but implement
as a bottom-up algorithm!

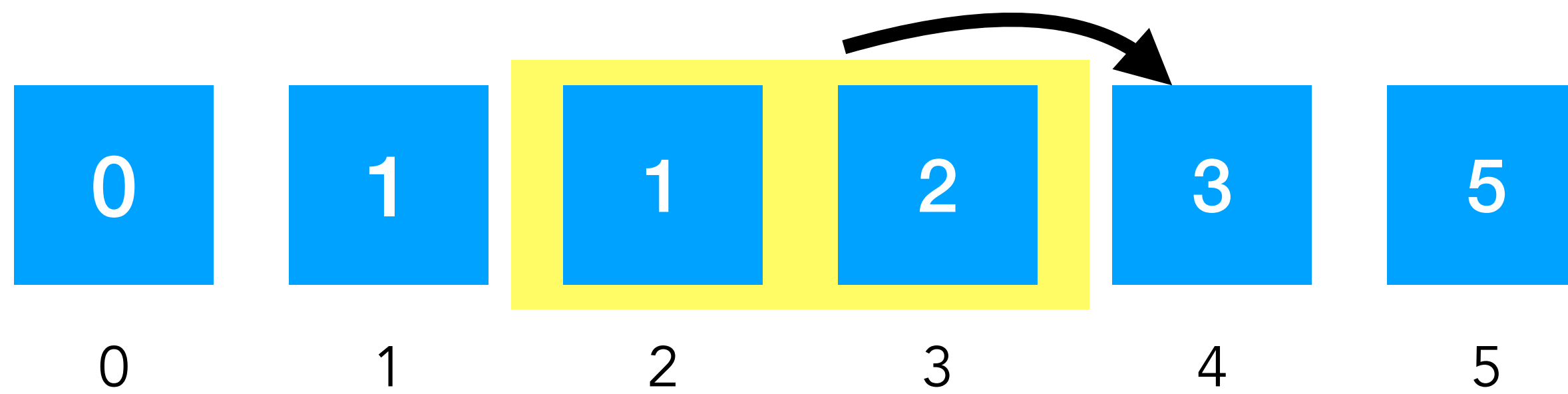


Start with first two, then build up

Instead, use ***dynamic programming***:
design a recursive solution top-down, but implement as a
bottom-up algorithm!



Key idea: only need to look at **two most recent** numbers



Accumulate via arguments

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Exercise



```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))
```

```
(define (fib n) (fib-h n 0 1))
```

Question: what is the runtime complexity of `fib`?

Exercise



```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))
```

```
(define (fib n) (fib-h n 0 1))
```

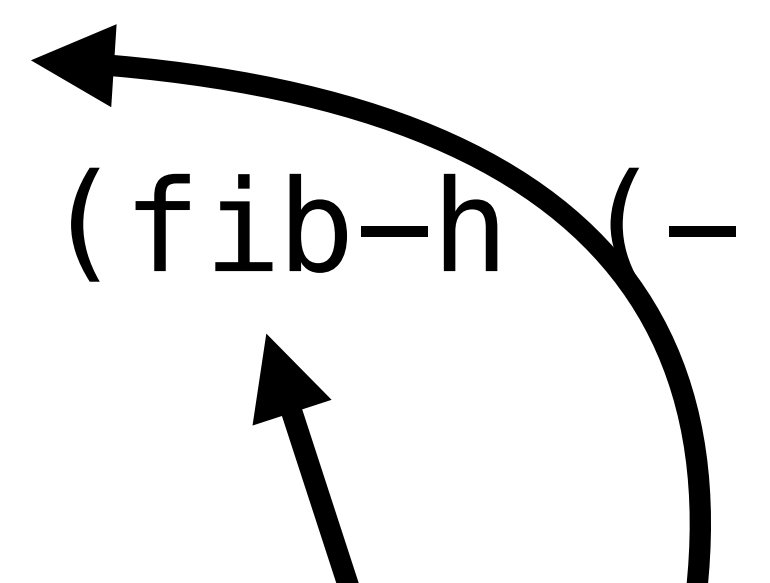
Answer: $O(n)$, fib-helper runs from n to 0

Consider how `fib-h` executes

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))
```

```
(define (fib n) (fib-h n 0 1))
```

```
(fib-helper 3 0 1)
= (if (= 3 0) 0 (fib-h (- 3 1) 1 (+ 0 1)))
= ...
= (fib-h 2 1 1)
= (if (= 2 0) 1 (fib-h (- 2 1) 1 (+ 1 1)))
= ...
= (fib-h 1 1 2)
```



Notice that we don't get the "stacking" behavior:
recursive calls don't grow the stack

This is because `fib-h` is **tail recursive**

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

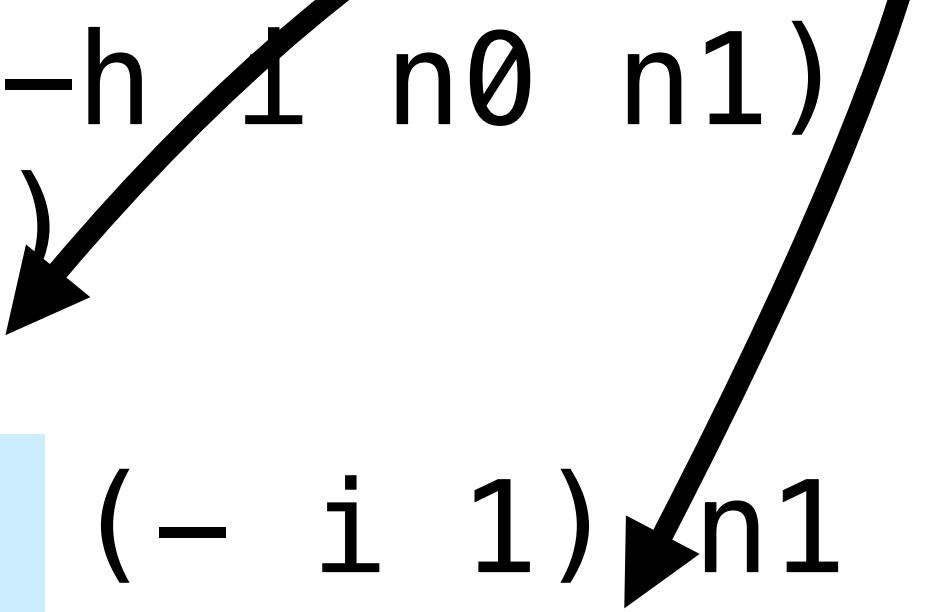
Intuitively: a callsite is in **tail-position** if it is the **last thing** a function will do before exiting

(We call these **tail calls**)

This is because `fib-h` is **tail recursive**

Both of these are tail calls

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))
```



```
(define (fib n) (fib-h n 0 1))
```

Intuitively: a callsite is in **tail-position** if it is the **last thing** a function will do before exiting

(We call these **tail calls**)

Tail calls / tail recursion

- Unlike calls in general, **tail calls** do not affect the stack:
 - Tail calls *do not grow* (or shrink) the stack.
 - They are more like a goto/jump than a normal call.
- A subexpression is in **tail position** if it's the last subexpression to run, whose return value is also the value for its parent expression:
 - In (let ([x rhs]) body); body is in *tail position*...
 - In (if grd thn els); thn & els are in *tail position*...
- A function is **tail recursive** if all recursive calls in tail position
- Tail-recursive functions are analogous to loops in imperative langs

Exercise



Which of the following is tail recursive?

```
(define (length-0 l)
  (if (null? l)
      0
      (+ 1 (length-0 (cdr l)))))
```

```
(define (length-1 l n)
  (if (null? l)
      n
      (length-1 (cdr l) (+ n 1))))
```

Exercise



Answer

```
(define (length-0 l)
  (if (null? l)
      0
      (+ 1 (length-0 (cdr l)))))
```

Not tail recursive
Adds (+ 1 _) operation to stack

```
(define (length-1 l n)
  (if (null? l)
      n
      (length-1 (cdr l) (+ n 1))))
```

Is tail recursive!
Call to length-1 in tail position