Tail Calls

July 21, 2022
Laryn Qi

Announcements

Scheme Lists Practice

Dynamic Scope

Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was *defined*

Dynamic scope: The parent of a frame is the environment in which a procedure was *called*

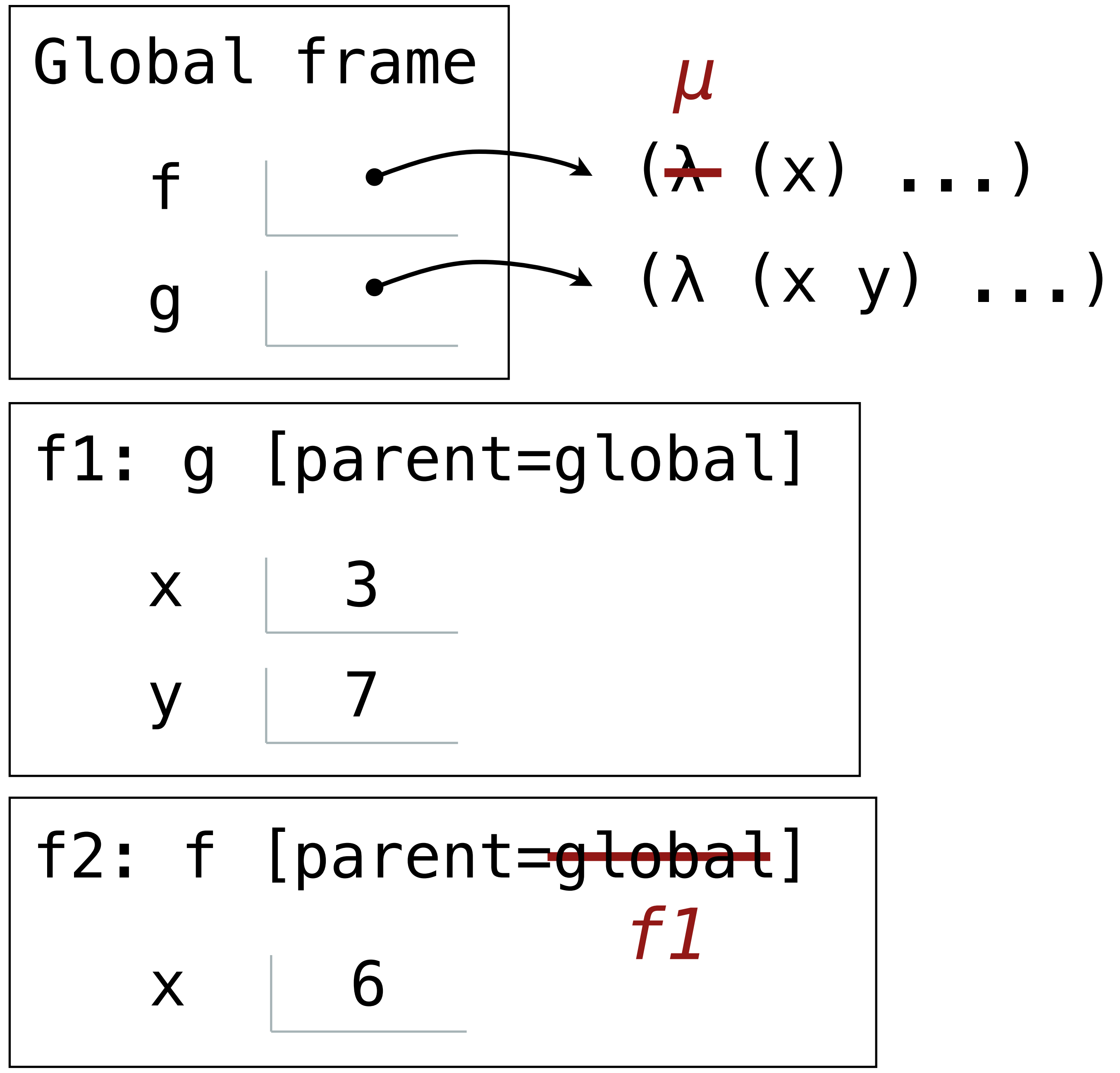
Special form to create dynamically scoped procedures (you will implement **mu** special form in Project 4 Scheme)

mu

(define f (~~lambda~~ (x) (+ x y)))

(define g (lambda (x y) (f (+ x x))))

(g 3 7)



Lexical scope: The parent for f's frame is the global frame

Error: unknown identifier: y

Dynamic scope: The parent for f's frame is g's frame

13

Space

Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

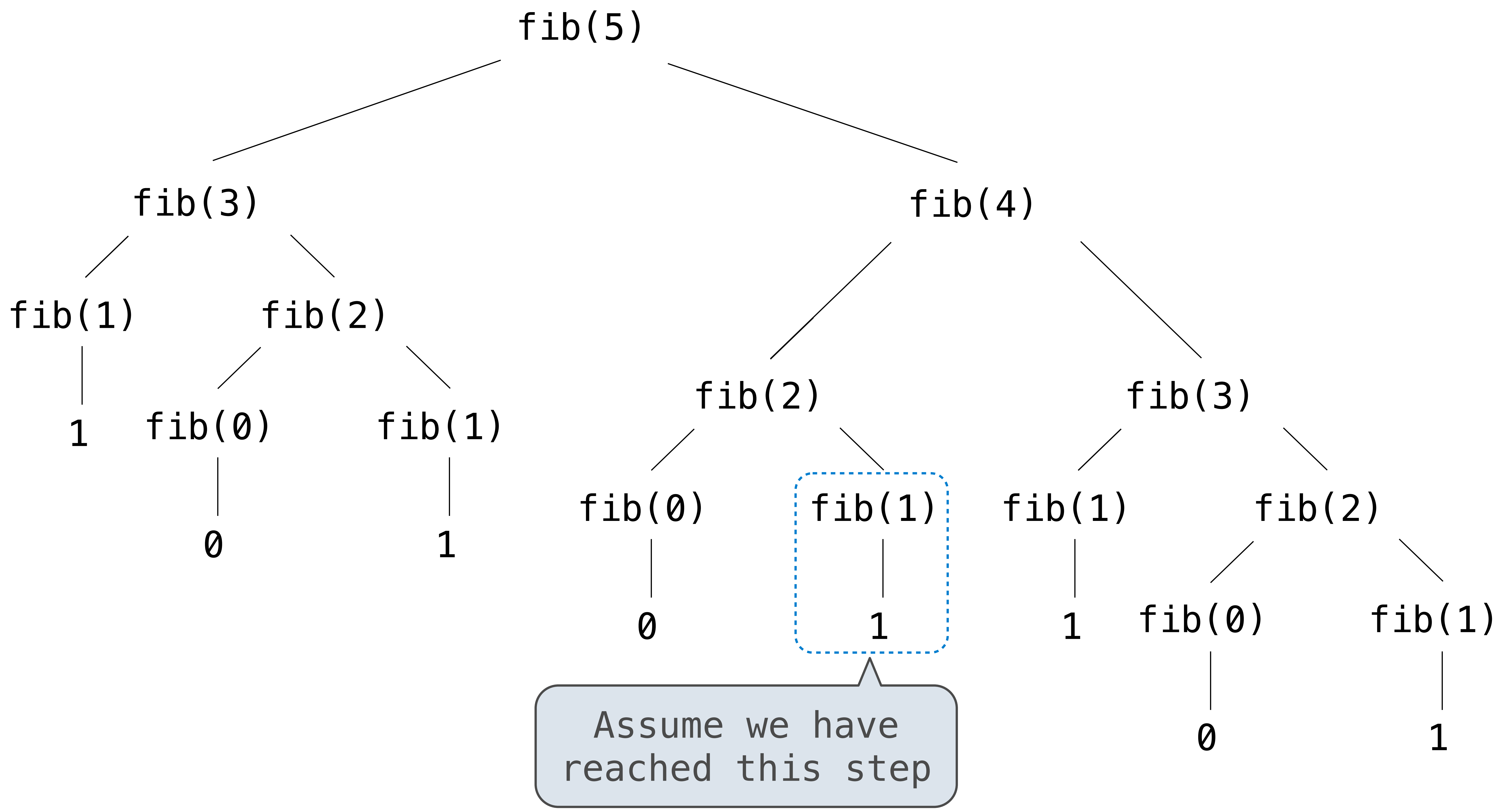
Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

Active environments:

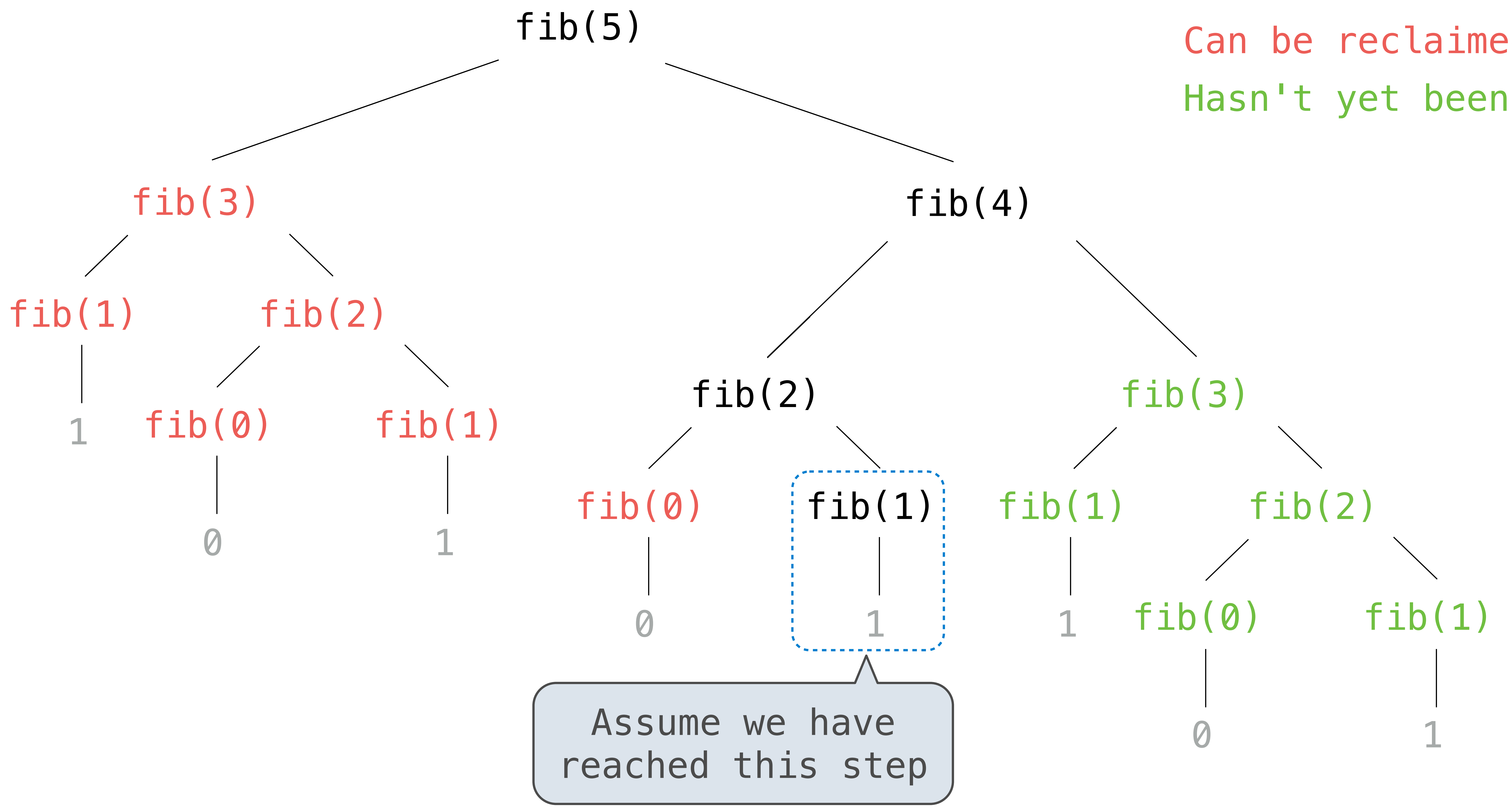
- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

Fibonacci Space Consumption



Fibonacci Space Consumption

Has an active environment
Can be reclaimed
Hasn't yet been created



fib takes **linear space.**

(Demo)

pythontutor.com/composingprograms.html#code=def%20fib%28n%29%3A%0A%20%20%20%20if%20n%20%3D%3D%200%20or%20n%20%3D%3D%201%3A%0A%20%20%20%20%20%20%20%20return%20n%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20return%20fib%28n-2%29%20%2B%20fib%28n-1%29%0A%20%20%20%20%20%20%20%20%0Afib%286%29&mode=display&origin=composingprograms.js&cumulative=false&py=3&rawInputLstJSON=[]&curInstr=1

Tail Recursion

Functional Programming

All functions are pure functions.

No re-assignment and no mutable data types.

Name-value bindings are permanent.

Advantages of functional programming:

- The value of an expression is independent of the order in which sub-expressions are evaluated
- Sub-expressions can safely be evaluated in parallel or only on demand (lazily) (Demo)
- **Referential transparency:** The value of an expression does not change when we substitute one of its subexpression with the value of that subexpression

But... no for/while statements! Can we make recursion efficient? Yes!

Recursion and Iteration in Python

In Python, recursive calls always create new active frames.

`fact_k(n, k)` computes: `n! * k`

```
def fact_k(n, k):
    if n == 0:
        return k
    else:
        return fact_k(n - 1, n*k)
```

```
def fact_k(n, k):
    while n > 0:
        n, k = n - 1, k * n
    return k
```

Time	Space
Linear	Linear
Linear	Constant

Tail Recursion

From the Revised[7] Report on the Algorithmic Language Scheme:

> "Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

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

Should use resources like

```python
def fact_k(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

How? Eliminate the middleman!

Time	**Space**
Linear	Constant

(Demo)

pythontutor.com/composingprograms.html#code=def%20factorial%28n,%20k%29%3A%0A%20%20%20%20if%20n%20%3D%3D%200%3A%0A%20%20%20%20%20%20%20%20return%20k%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20return%20factorial%28n-1,%20k*n%29%0A%20%20%20%20%0Afactorial%284,%2010%29&mode=display&origin=composingprograms.js&cumulative=true&py=3&rawInputLstJSON=[]&curInstr=0

Tail Calls

Tail Calls, Tail Contexts, Tail Recursion

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter should support an unbounded number of active tail calls using only a constant amount of space.

A tail call is a call expression in a ***tail context***:

- The last body sub-expression in a **lambda** expression (or procedure definition)
- Sub-expressions 2 & 3 in a tail context **if** expression
- All non-predicate sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

A recursive procedure is tail recursive if ***all*** of its recursive calls are tail calls

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

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

Example: Length of a List

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

Not a tail context

A call expression is not a tail call if more computation is still required in the calling procedure

Linear recursive procedures can often be re-written to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n)) ) )
  (length-iter s 0) )
```

Recursive call is a tail call

Break

Tail Recursion Practice

Tail Recursion Examples

Which Procedures are Tail Recursive?

Which of the following procedures run in constant space?

```scheme
;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
           -1
           (length (cdr s))) ) )

;; Return whether s has any repeated elements.
(define (has-repeat2 s)
  (if (null? s)
      #f
      (if (contains (cdr s) (car s))
          #t
          (if (has-repeat2 (cdr s))
            #t
            #f                          )   )   ) )
```

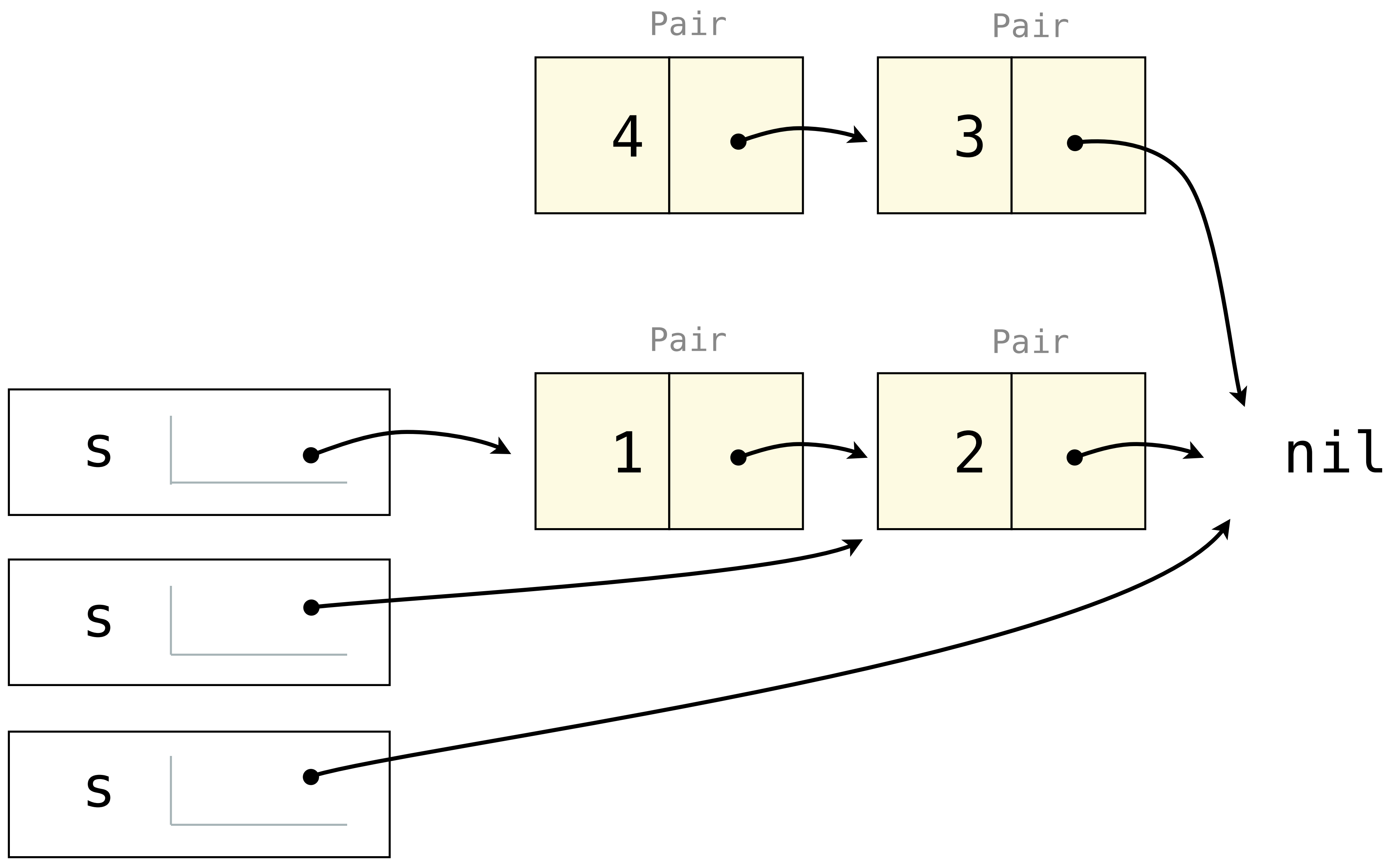
```scheme
;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      #f
      (if (= v (car s))
          #t
          (contains (cdr s) v))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      #f
      (if (contains (cdr s) (car s))
          #t
          (has-repeat (cdr s))) ) )
```

Tail Recursion with Scheme Lists

```scheme
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))) ) )
```

```scheme
(map (lambda (x) (- 5 x)) (list 1 2))
```




```scheme
(define (map procedure s)
  (define (map-reverse s m)
    (if (null? s)
        m
        (map-reverse (cdr s)
                     (cons (procedure (car s))
                           m)) ) )
  (reverse (map-reverse s nil)))
```

```scheme
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s)
                      (cons (car s) r)) ) )
  (reverse-iter s nil))
```

Tail Recursion Techniques

Base case should return the complete answer (rather than a partial solution).

Define a helper with an extra parameter to keep track of progress so far.

Sketch an iterative solution (e.g. in Python) – names that are iteratively updated need to be tracked as function arguments in recursion.

Verify all recursive calls are tail calls.

(Demo)

Tail Call Optimization

Who'da Thunk?

Thunk: An expression wrapped in an argument-less function.

```
thunk1 = lambda: 2 * (3 + 4)

thunk2 = lambda: add(2, 4)

thunk1()

thunk2()
```

Known as **Unevaluated** objects in the Scheme project.

Trampolining

Trampoline: A loop that iteratively invokes thunk-returning functions.

```
def trampoline(f, *args):
  v = f(*args)
  while callable(v):
    v = v()
  return v
```

The function needs to be thunk-returning.

```
def fact_k_thunked(n, k):
  if n == 0:
    return k
  return lambda: fact_k_thunked(n - 1, n * k)
```

```
trampoline(fact_k_thunked, 3, 1)
```

This way of executing the factorial function uses a constant number of frames.

Trampolining can simulate tail call optimization in unoptimized languages (e.g. Python).

(Demo)

https://pythontutor.com/composingprograms.html#code=def%20trampoline%28f,%20*args%29%3A%0A%20%20%20%20v%20%3D%20f%28*args%29%0A%20%20%20%20while%20callable%28v%29%3A%0A%20%20%20%20%20%20%20%20v%20%3D%20v%28%29%0A%20%20%20%20return%20v%0A%0Adef%20fact_thunked%28n,
%20k%29%3A%0A%20%20%20%20if%20n%20%3D%3D%200%3A%0A%20%20%20%20%20%20%20%20return%20k%0A%20%20%20%20return%20lambda%3A%20fact_thunked%28n%20-%201,%20n%20*%20k%29%0A%20%20%20%20%0Atrampoline%28fact_thunked,%203,%201%29&cumulative=true&mode=edit&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D