61A Lecture 21

Monday, October 17

Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

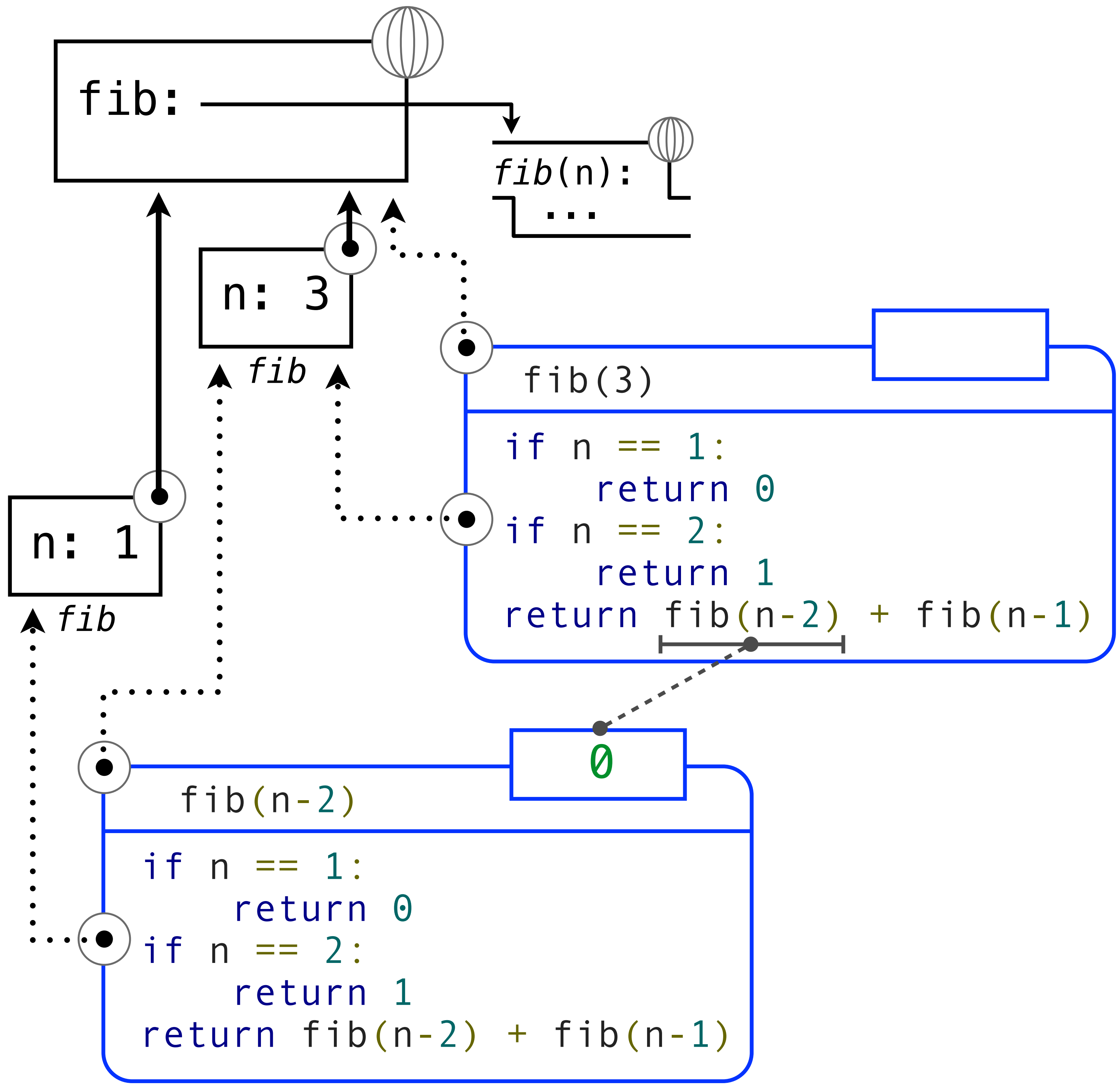
Values and frames referenced by active environments are kept.

Memory used for other values and frames can be reclaimed.

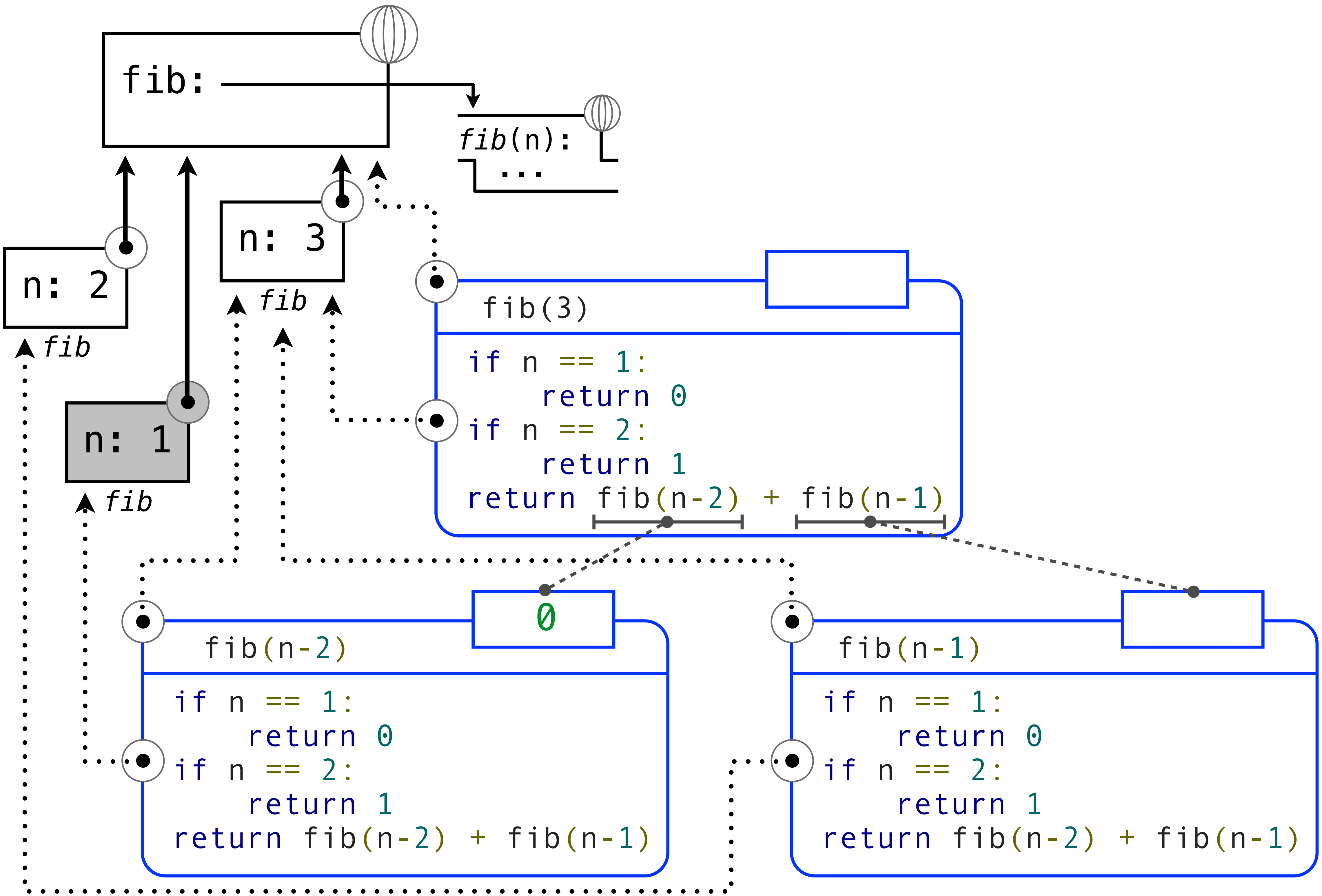
Active environments:

- The environment for the current expression being evaluated
- Environments for calls that depend upon the value of the current expression
- Environments associated with functions referenced by active environments

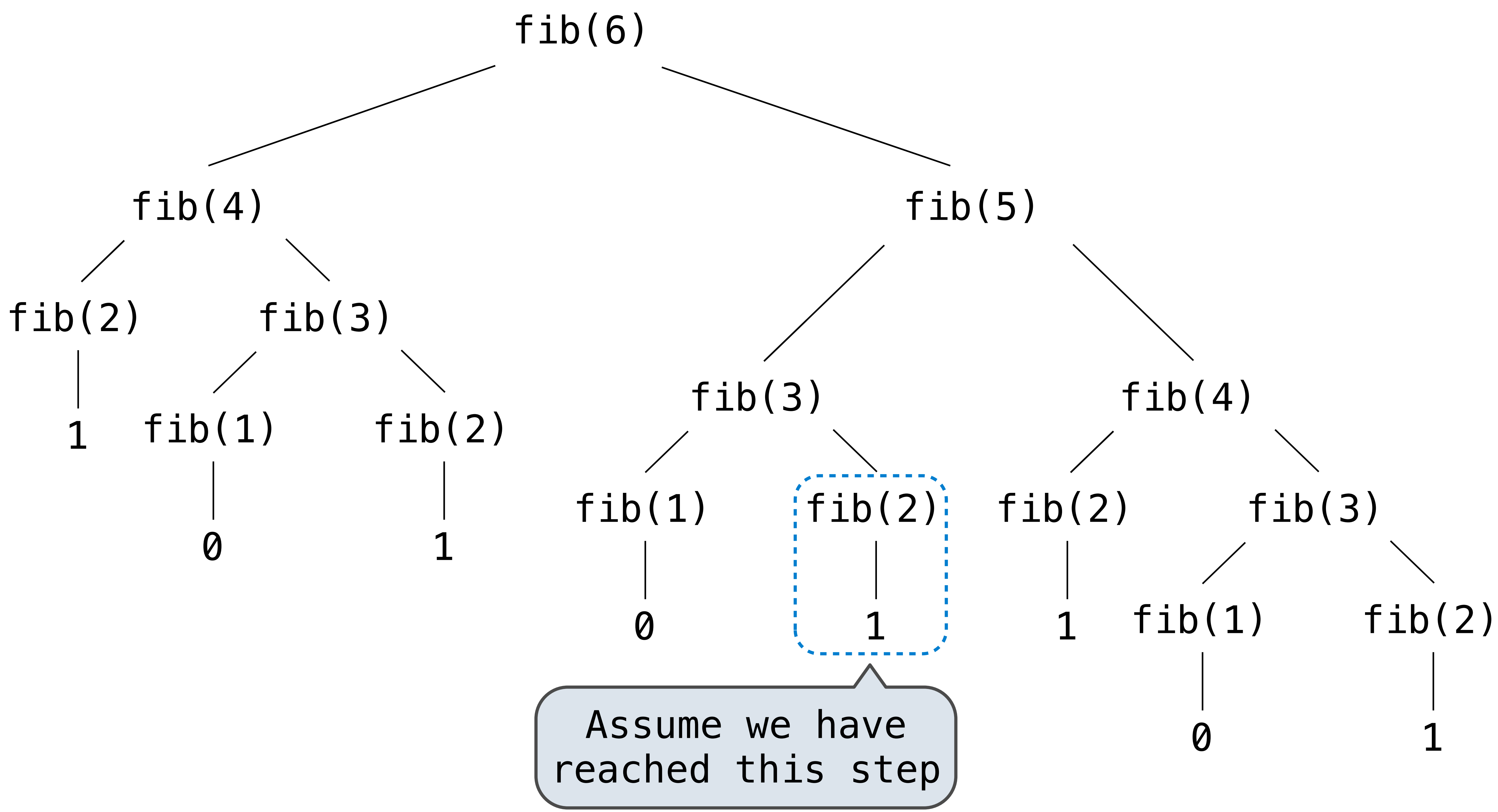
Fibonacci Environment Diagram



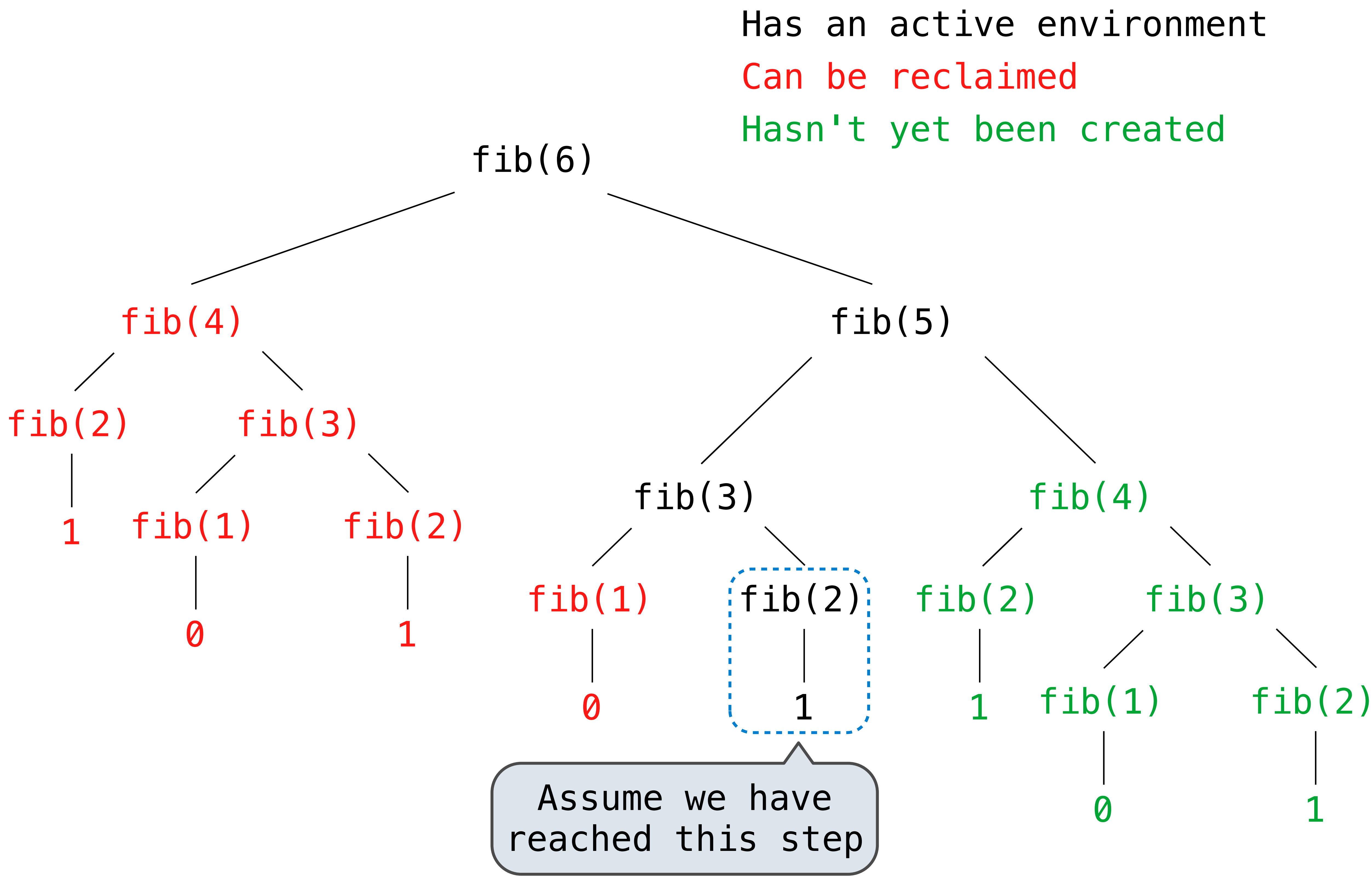
Fibonacci Environment Diagram



Fibonacci Memory Consumption

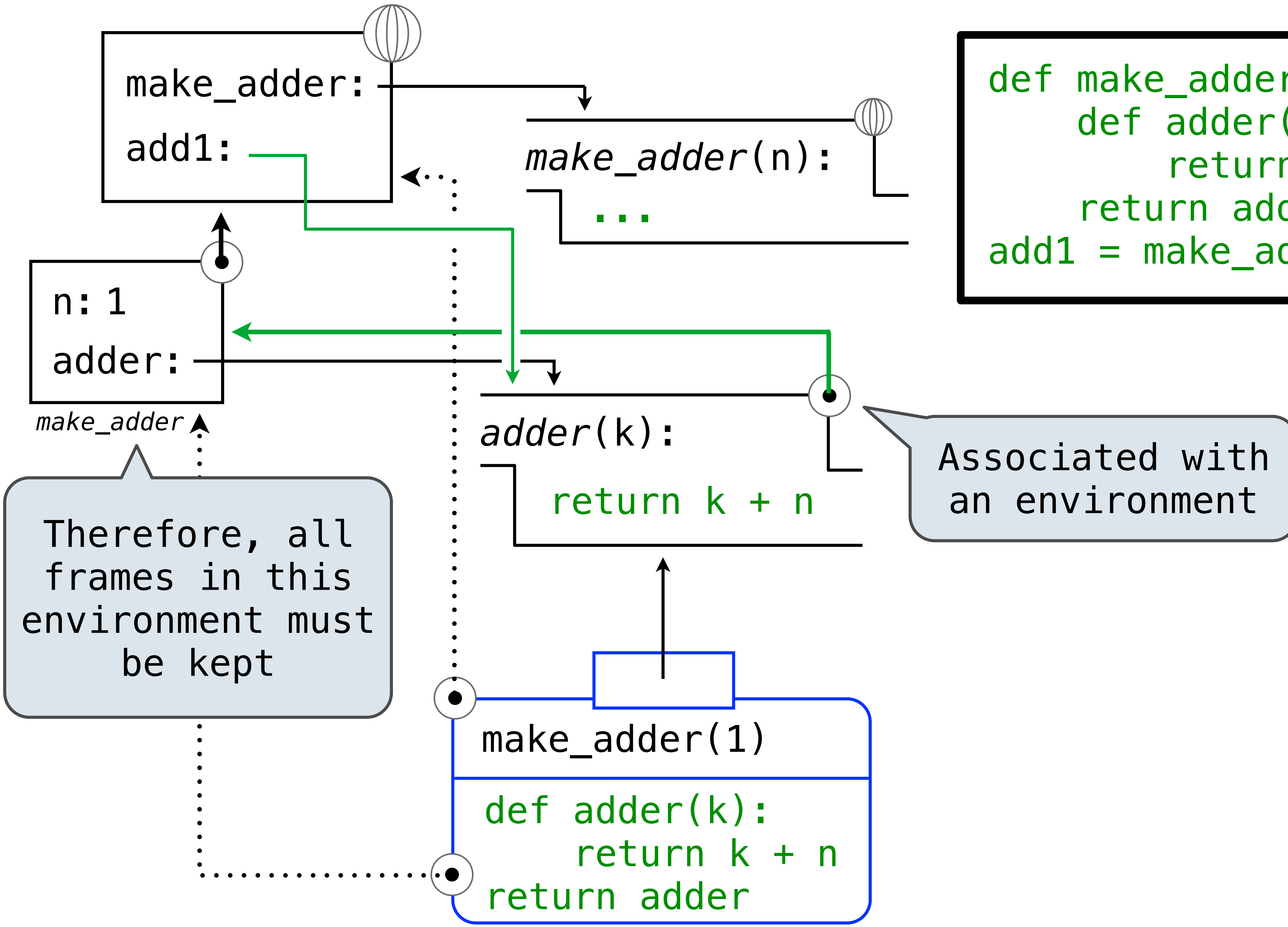


Fibonacci Memory Consumption



Active Environments for Returned Functions

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
add1 = make_adder(1)
```



Monday, October 17, 2011

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

n: size of the problem

R(n): Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of ***n***.

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

Code	Time	Space
`fib_iter`	$\Theta(n)$	$\Theta(1)$
`fib` (memoized)	$\Theta(n)$	$\Theta(n)$

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
         prev, curr = curr, prev + curr
    return curr
```

```
@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These functions scale well.

Doubling the problem increments resources needed.

$\Theta(1)$ Constant. The problem size doesn't matter.

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

```
def square(x):
    return x*x

def fast_exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

	Time	Space
`exp`	$\Theta(n)$	$\Theta(n)$
`fast_exp`	$\Theta(\log n)$	$\Theta(\log n)$

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

```
def square(x):
    return x*x

def fast_exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```