# Discussion 04

List mutation, Orders of growth, Nonlocal

# How to figure out orders of growth

1. Walk through the function.
   - What does it do?
   - How does it change its input?
   - When does it stop iterating/recursion?

2. After you get an expression for the order of growth, simplify it!
   - Remove any constants
   - Remove smaller terms

# Techniques

**Draw a tree:**

1. Start with the first function call.
2. Then draw a branch for each recursive call from that function call. (We used this technique to figure out recursive_fib)

**Draw a graph:**

1. Figure out what should be on the x axis.
2. Start with the largest input and determine how the input changes for each recursive call.
3. Find the area.
4. Simplify

# 1.2 #3

```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

# 1.2 #3

```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

What does this function do?

# 1.2 #3

```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

What does this function do?

Finds the nth fibonacci number. Try running through this function with a small input.

# 1.2 #3

```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

What does this function do?

Finds the nth fibonacci number. Try running through this function with a small input.

How many times do we iterate through the while loop?

# 1.2 #3

```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

What does this function do?

Finds the nth fibonacci number. Try running through this function with a small input.

How many times do we iterate through the while loop?

We keep iterating while i is less than n. i starts as 0, and increments by 1 with each iteration. So there must be n iterations.

# 1.2 #3

```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

What does this function do?

Finds the nth fibonacci number. Try running through this function with a small input.

How many times do we iterate through the while loop?

We keep iterating while i is less than n. i starts as 0, and increments by 1 with each iteration. So there must be n iterations.

How much work do we do at each iteration?

# 1.2 #3

```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

What does this function do?

Finds the nth fibonacci number. Try running through this function with a small input.

How many times do we iterate through the while loop?

We keep iterating while i is less than n. i starts as 0, and increments by 1 with each iteration. So there must be n iterations.

How much work do we do at each iteration?

We reassign two values inside the while loops and increment i. This takes a constant amount of work.

# 1.2 #3

```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

What does this function do?

Finds the nth fibonacci number. Try running through this function with a small input.

How many times do we iterate through the while loop?

We keep iterating while i is less than n. i starts as 0, and increments by 1 with each iteration. So there must be n iterations.

How much work do we do at each iteration?

We reassign two values inside the while loops and increment i. This takes a constant amount of work.

Multiply and simplify!

# 1.2 #3

```
def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev
```

What does this function do?

Finds the nth fibonacci number. Try running through this function with a small input.

How many times do we iterate through the while loop?

We keep iterating while i is less than n. i starts as 0, and increments by 1 with each iteration. So there must be n iterations.

How much work do we do at each iteration?

We reassign two values inside the while loops and increment i. This takes a constant amount of work.

Multiply and simplify!

So we have $\Theta(n)$ total iterations, and $\Theta(1)$ work at each iteration.
This gives us $\Theta(1 * n)$ total amount of work we must do.

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

What does this function do?

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

What does this function do?

Divides n by 2 at every iteration and accumulates the sum of those numbers in total

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

What does this function do?

Divides n by 2 at every iteration and accumulates the sum of those numbers in total

How many iterations of the while loop are there?

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

What does this function do?

Divides n by 2 at every iteration and accumulates the sum of those numbers in total

How many iterations of the while loop are there?

This is a bit tricky. Since we continue halving the input from n to 0 or 1, we know that there are **log(n)** iterations. But why?

# Why halving implies log

We know that at each iteration, we halve the input.

# Why halving implies log

We know that at each iteration, we halve the input.

$$n \longrightarrow n/2 \longrightarrow n/4 \longrightarrow n/8 \longrightarrow \quad \dots \quad \longrightarrow 1$$

# Why halving implies log

We know that at each iteration, we halve the input.

$$n \longrightarrow n/2 \longrightarrow n/4 \longrightarrow n/8 \longrightarrow \quad \ldots \quad \longrightarrow 1$$

Each of these circles represents a function call. We need to determine how many function calls there are if we keep halving n until it becomes 1. In other words, how many times can I divide n by 2 to get 1?

# Why halving implies log

We know that at each iteration, we halve the input.

$$n \longrightarrow n/2 \longrightarrow n/4 \longrightarrow n/8 \longrightarrow \quad \dots \quad \longrightarrow 1$$

Each of these circles represents a function call. We need to determine how many function calls there are if we keep halving n until it becomes 1. In other words, how many times can I divide n by 2 to get 1?

$$n/2^0 \longrightarrow n/2^1 \longrightarrow n/2^2 \longrightarrow n/2^3 \longrightarrow \quad \dots \quad \longrightarrow 1$$

# Why halving implies log

We know that at each iteration, we halve the input.

$$n \longrightarrow n/2 \longrightarrow n/4 \longrightarrow n/8 \longrightarrow \dots \longrightarrow 1$$

Each of these circles represents a function call. We need to determine how many function calls there are if we keep halving n until it becomes 1. In other words, how many times can I divide n by 2 to get 1?

$$n/2^0 \longrightarrow n/2^1 \longrightarrow n/2^2 \longrightarrow n/2^3 \longrightarrow \dots \longrightarrow 1$$

In other other words, notice that in each hop, we just raise 2 to a higher power. How many times do we need to raise 2 to get to 1? We get the following equation:

# Why halving implies log

We know that at each iteration, we halve the input.

$$n \longrightarrow n/2 \longrightarrow n/4 \longrightarrow n/8 \longrightarrow \ldots \longrightarrow 1$$

Each of these circles represents a function call. We need to determine how many function calls there are if we keep halving n until it becomes 1. In other words, how many times can I divide n by 2 to get 1?

$$n/2^0 \longrightarrow n/2^1 \longrightarrow n/2^2 \longrightarrow n/2^3 \longrightarrow \ldots \longrightarrow 1$$

In other other words, notice that in each hop, we just raise 2 to a higher power. How many times do we need to raise 2 to get to 1? We get the following equation:

$$n/2^k = 1$$

# Why halving implies log

We know that at each iteration, we halve the input.

$$n \longrightarrow n/2 \longrightarrow n/4 \longrightarrow n/8 \longrightarrow \quad \dots \quad \longrightarrow 1$$

Each of these circles represents a function call. We need to determine how many function calls there are if we keep halving n until it becomes 1. In other words, how many times can I divide n by 2 to get 1?

$$n/2^0 \longrightarrow n/2^1 \longrightarrow n/2^2 \longrightarrow n/2^3 \longrightarrow \quad \dots \quad \longrightarrow 1$$

In other other words, notice that in each hop, we just raise 2 to a higher power. How many times do we need to raise 2 to get to 1? We get the following equation:

$$n/2^k = 1$$

Now do math to find k.

# Why halving implies log

We know that at each iteration, we halve the input.

$$n \longrightarrow n/2 \longrightarrow n/4 \longrightarrow n/8 \longrightarrow \quad \dots \quad \longrightarrow 1$$

Each of these circles represents a function call. We need to determine how many function calls there are if we keep halving n until it becomes 1. In other words, how many times can I divide n by 2 to get 1?

$$n/2^0 \longrightarrow n/2^1 \longrightarrow n/2^2 \longrightarrow n/2^3 \longrightarrow \quad \dots \quad \longrightarrow 1$$

In other other words, notice that in each hop, we just raise 2 to a higher power. How many times do we need to raise 2 to get to 1? We get the following equation:

$$n/2^k = 1$$

Now do math to find k.

$$\begin{aligned} n &= 2^k \\ \log(n) &= \log(2^k) \\ \log(n) &= k*\log(2) \\ \log(n) &= k \end{aligned}$$

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

What does this function do?

Divides n by 2 at every iteration and accumulates the sum of those numbers in total

How many iterations of the while loop are there?

This is a bit tricky. Since we continue halving the input from n to 0 or 1, we know that there are **log(n)** iterations. But why?

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

What does this function do?

Divides n by 2 at every iteration and accumulates the sum of those numbers in total

How many iterations of the while loop are there?

This is a bit tricky. Since we continue halving the input from n to 0 or 1, we know that there are **log(n)** iterations. But why?

How much work is done at each iteration?

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

What does this function do?

Divides n by 2 at every iteration and accumulates the sum of those numbers in total

How many iterations of the while loop are there?

This is a bit tricky. Since we continue halving the input from n to 0 or 1, we know that there are **log(n)** iterations. But why?

How much work is done at each iteration?

Like the last problem, we only do two operations inside the loop, so its constant time!

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

What does this function do?

Divides n by 2 at every iteration and accumulates the sum of those numbers in total

How many iterations of the while loop are there?

This is a bit tricky. Since we continue halving the input from n to 0 or 1, we know that there are **log(n)** iterations. But why?

How much work is done at each iteration?

Like the last problem, we only do two operations inside the loop, so its constant time!

Multiply the numbers!

# 1.2 #4

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

What does this function do?

Divides n by 2 at every iteration and accumulates the sum of those numbers in total

How many iterations of the while loop are there?

This is a bit tricky. Since we continue halving the input from n to 0 or 1, we know that there are **log(n)** iterations. But why?

How much work is done at each iteration?

Like the last problem, we only do two operations inside the loop, so its constant time!

Multiply the numbers!

$\Theta(1 * \log(n)) = \Theta(\log(n))$

# 1.2 #5

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

# 1.2 #5

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

What does this function do?

# 1.2 #5

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

What does this function do?

Plug in some values for n and walk through the function. The function keeps decrementing the input until it reaches a number that is divisible by 7. At this point it returns 0. Then it goes back up the recursive calls and adds 1 for each recursive call. So this function just returns a number between 0 and 6 that is equivalent to n in mod 7

# 1.2 #5

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

What does this function do?

Plug in some values for n and walk through the function. The function keeps decrementing the input until it reaches a number that is divisible by 7. At this point it returns 0. Then it goes back up the recursive calls and adds 1 for each recursive call. So this function just returns a number between 0 and 6 that is equivalent to n in mod 7

How many recursive calls are there?

# 1.2 #5

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

What does this function do?

Plug in some values for n and walk through the function. The function keeps decrementing the input until it reaches a number that is divisible by 7. At this point it returns 0. Then it goes back up the recursive calls and adds 1 for each recursive call. So this function just returns a number between 0 and 6 that is equivalent to n in mod 7

How many recursive calls are there?

This function will call itself at most 6 times. Why?

# 1.2 #5

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

What does this function do?

Plug in some values for n and walk through the function. The function keeps decrementing the input until it reaches a number that is divisible by 7. At this point it returns 0. Then it goes back up the recursive calls and adds 1 for each recursive call. So this function just returns a number between 0 and 6 that is equivalent to n in mod 7

How many recursive calls are there?

This function will call itself at most 6 times. Why?

How much work is done in each recursive call?

# 1.2 #5

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

What does this function do?

Plug in some values for n and walk through the function. The function keeps decrementing the input until it reaches a number that is divisible by 7. At this point it returns 0. Then it goes back up the recursive calls and adds 1 for each recursive call. So this function just returns a number between 0 and 6 that is equivalent to n in mod 7

How many recursive calls are there?

This function will call itself at most 6 times. Why?

How much work is done in each recursive call?

Constant!

# 1.2 #5

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

What does this function do?

Plug in some values for n and walk through the function. The function keeps decrementing the input until it reaches a number that is divisible by 7. At this point it returns 0. Then it goes back up the recursive calls and adds 1 for each recursive call. So this function just returns a number between 0 and 6 that is equivalent to n in mod 7

How many recursive calls are there?

This function will call itself at most 6 times. Why?

How much work is done in each recursive call?

Constant!

Whats the final order of growth?

# 1.2 #5

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

What does this function do?

Plug in some values for n and walk through the function. The function keeps decrementing the input until it reaches a number that is divisible by 7. At this point it returns 0. Then it goes back up the recursive calls and adds 1 for each recursive call. So this function just returns a number between 0 and 6 that is equivalent to n in mod 7

How many recursive calls are there?

This function will call itself at most 6 times. Why?

How much work is done in each recursive call?

Constant!

Whats the final order of growth?

$\Theta(1)$

# 2.1 #1, #2

What is wrong with the code?

```
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```

```
def adder(x):
    def add(y):
        nonlocal x, y
        x += y
        return x
    return add
adder(2)(3)
```

# 2.1 #1, #2

What is wrong with the code?

```
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```

Nonlocal looks in parent frames, until it gets to global. If it gets to the global frame without finding the variable, it will say “I have no idea where this variable is” and Error.

```
def adder(x):
    def add(y):
        nonlocal x, y
        x += y
        return x
    return add
adder(2)(3)
```

# 2.1 #1, #2

What is wrong with the code?

```
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```

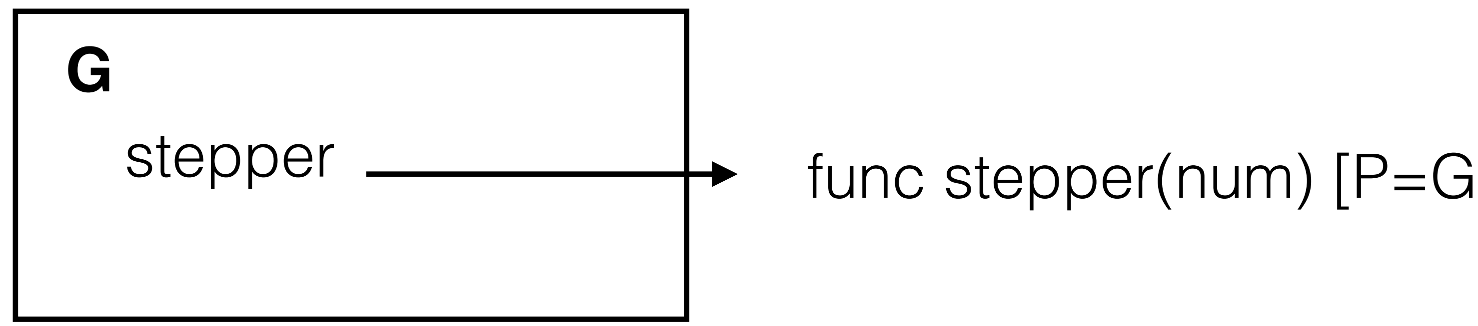
Nonlocal looks in parent frames, until it gets to global. If it gets to the global frame without finding the variable, it will say “I have no idea where this variable is” and Error.

```
def adder(x):
    def add(y):
        nonlocal x, y
        x += y
        return x
    return add
adder(2)(3)
```

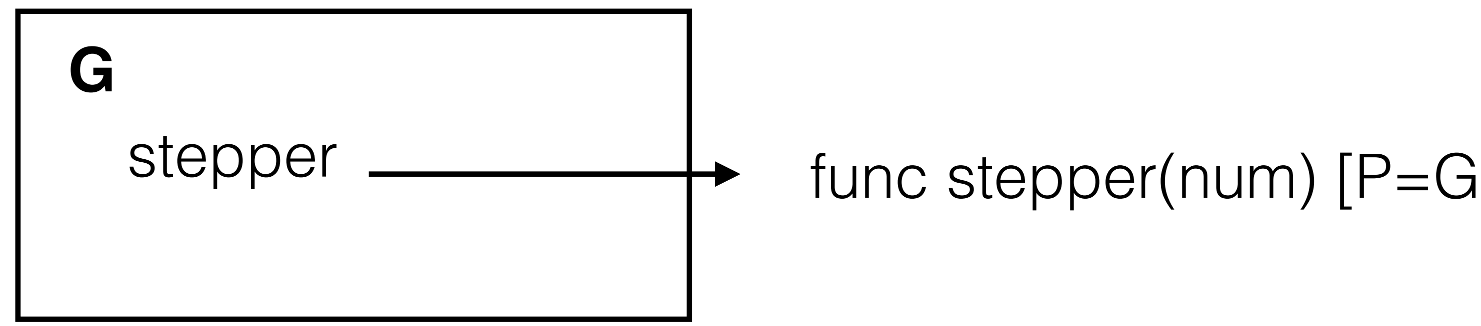
y isn’t defined in the parent! It’s passed into add, so it’s a local variable.

```python
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

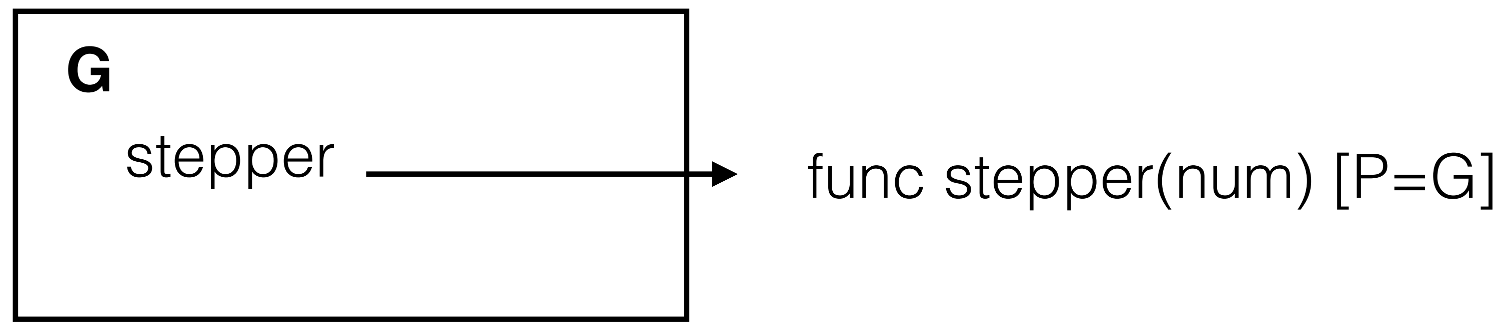
```python
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```



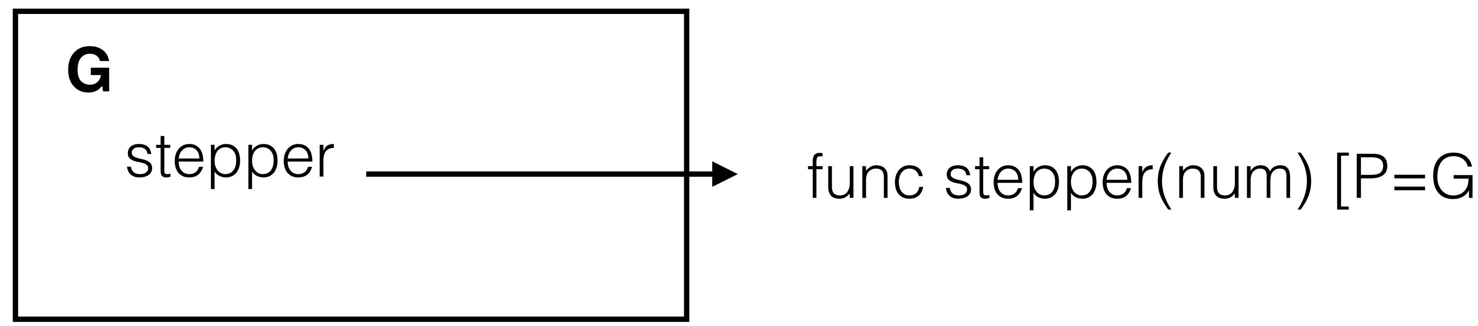
```python
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```



```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

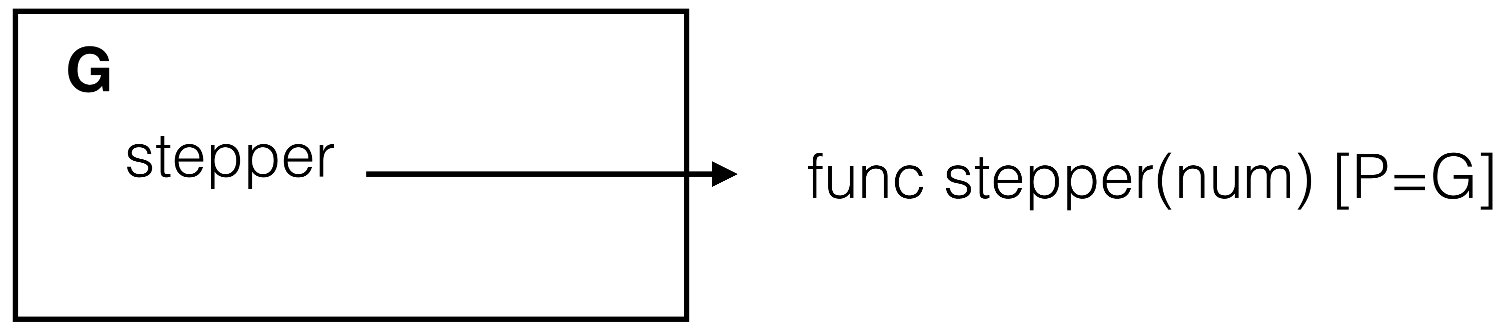


```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```



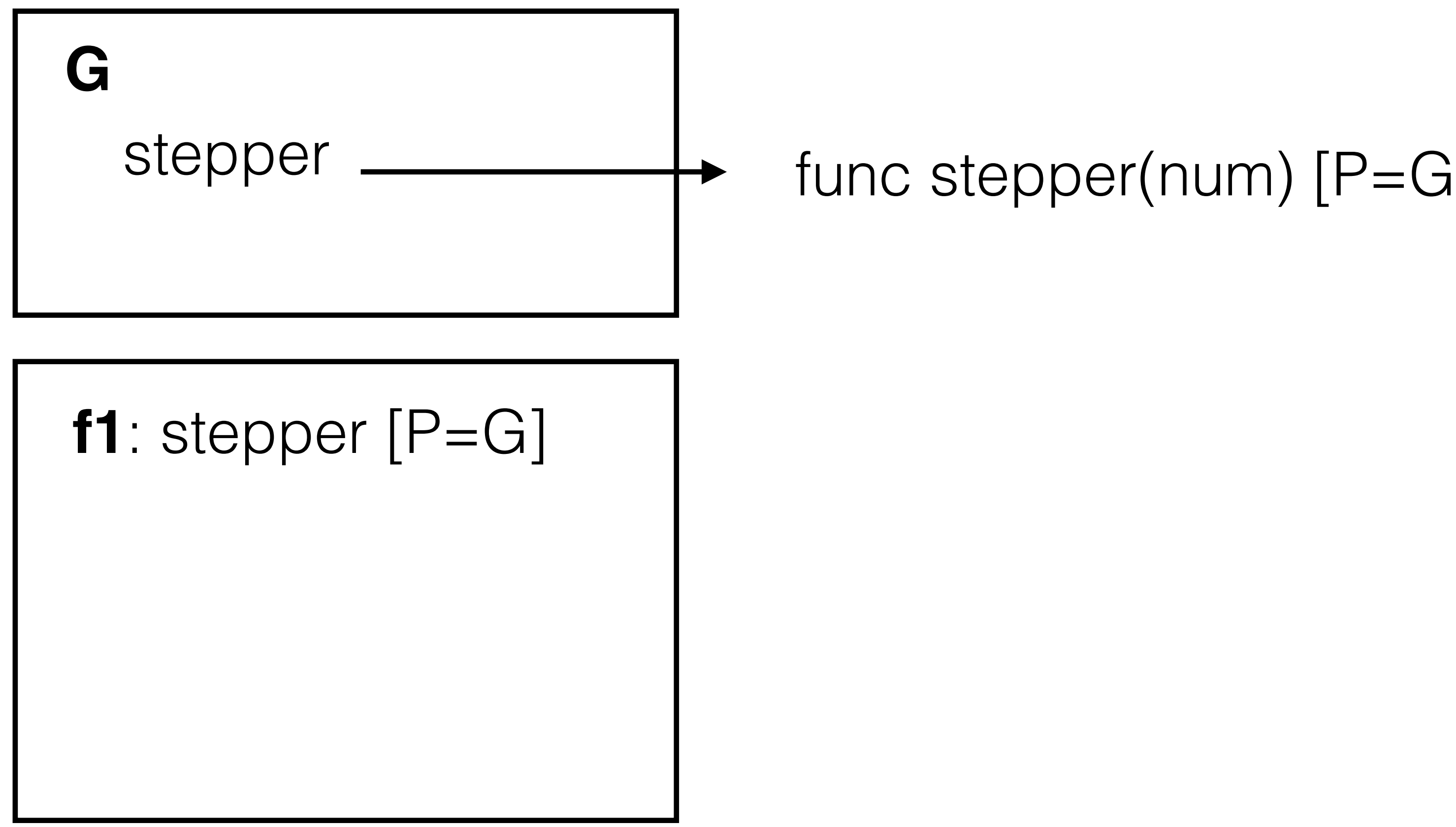
```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

f1



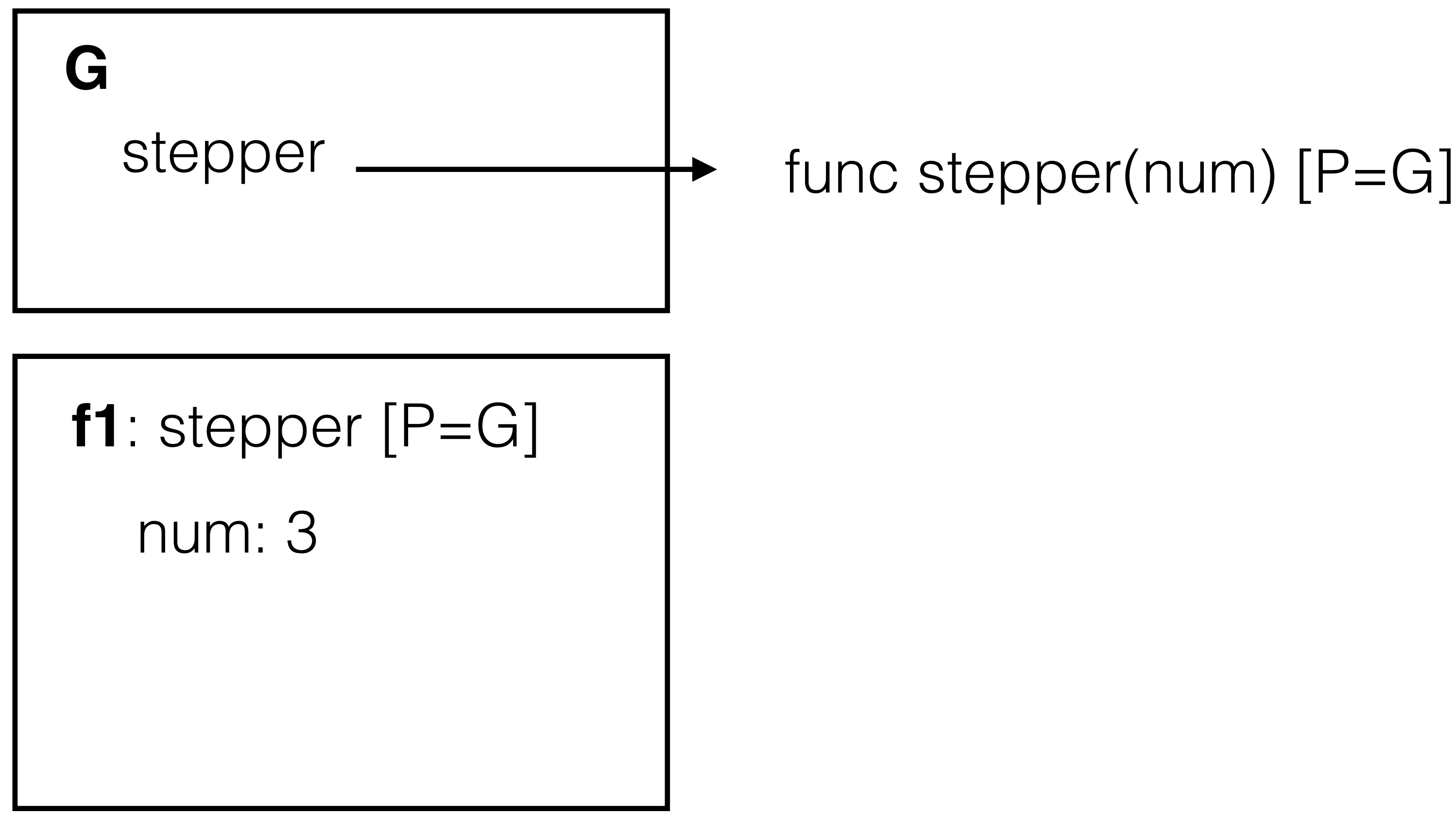
```python
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

f1

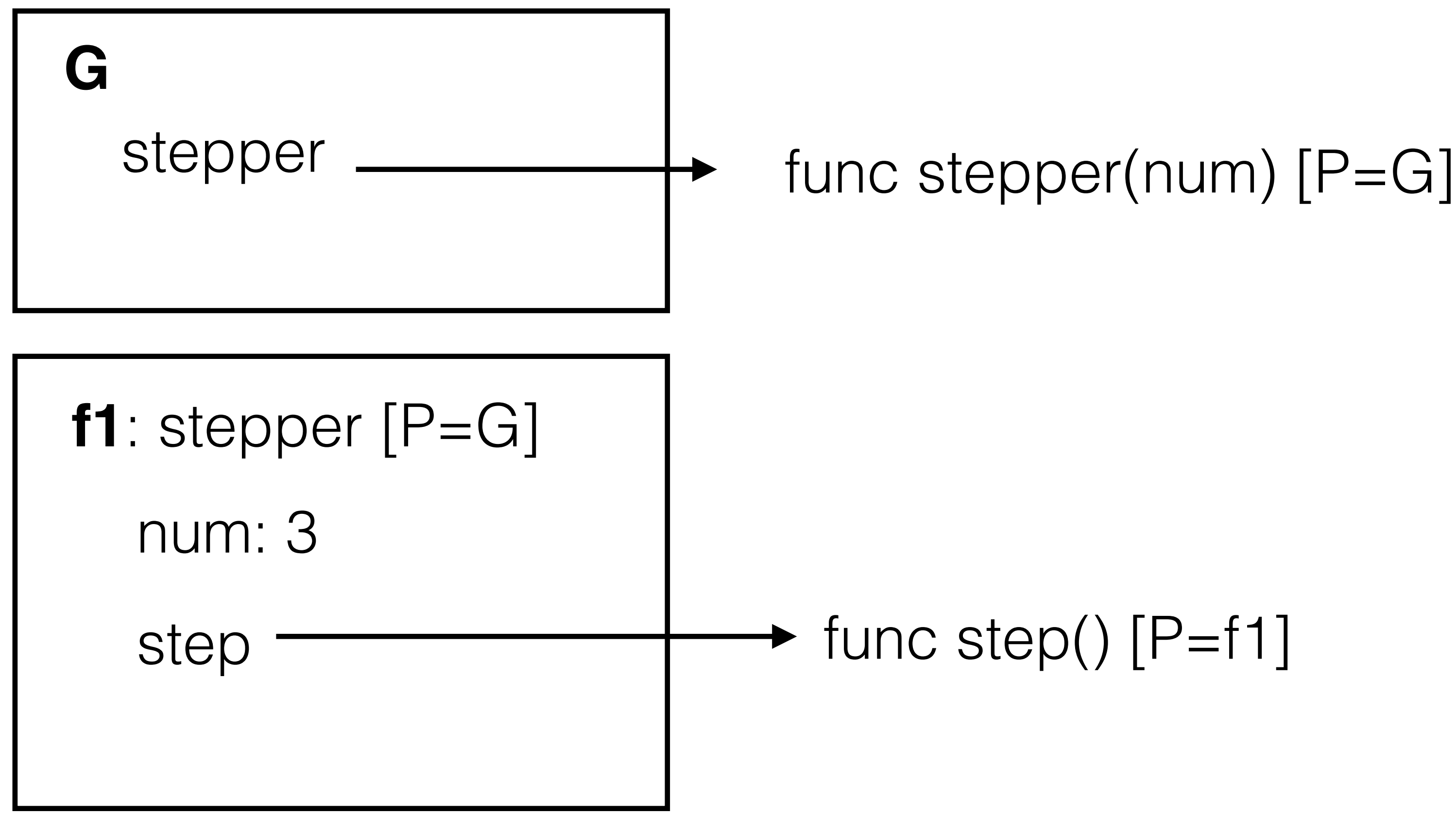


```python
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

f1

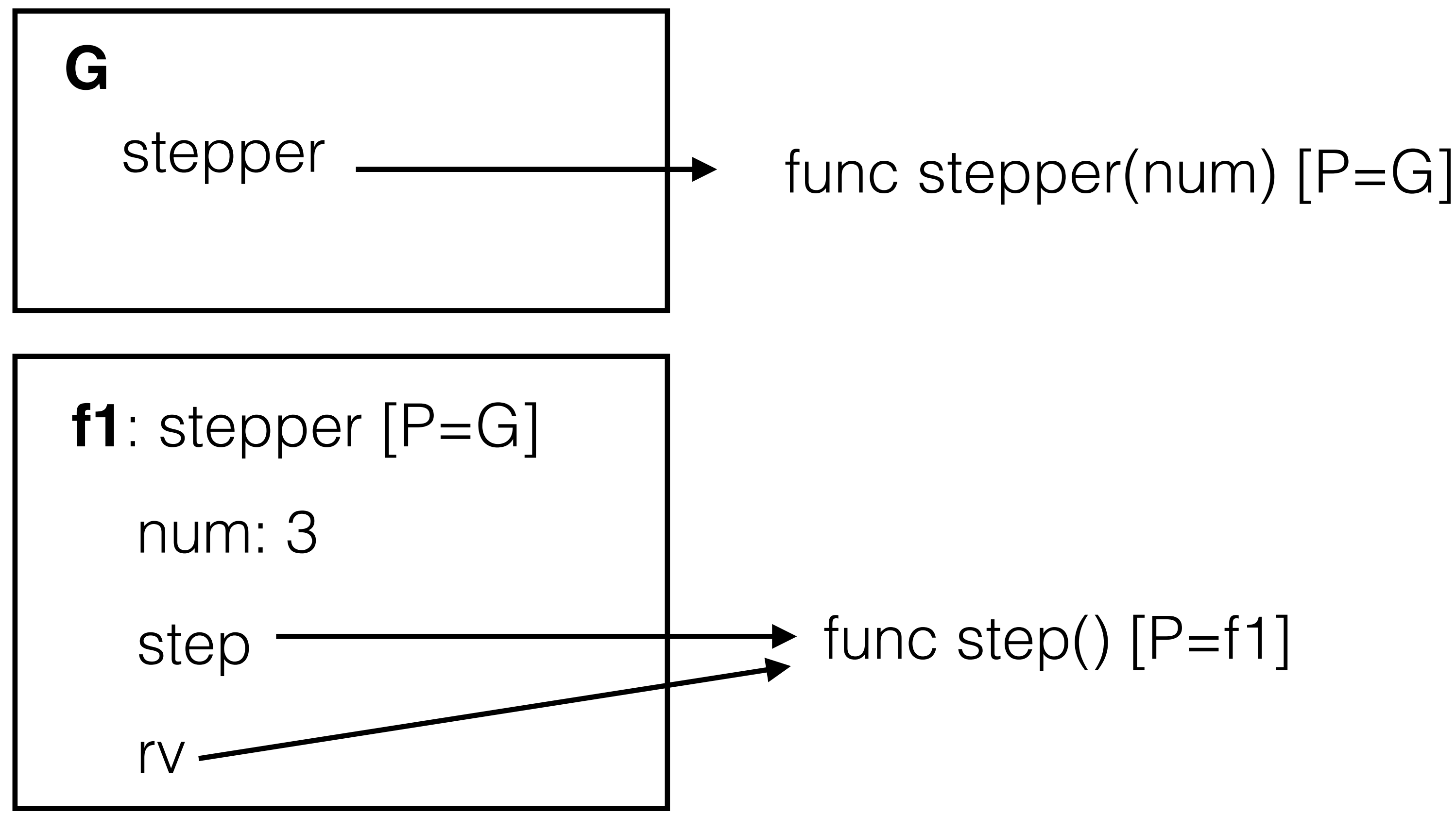


```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)   f1
s()
s()
```



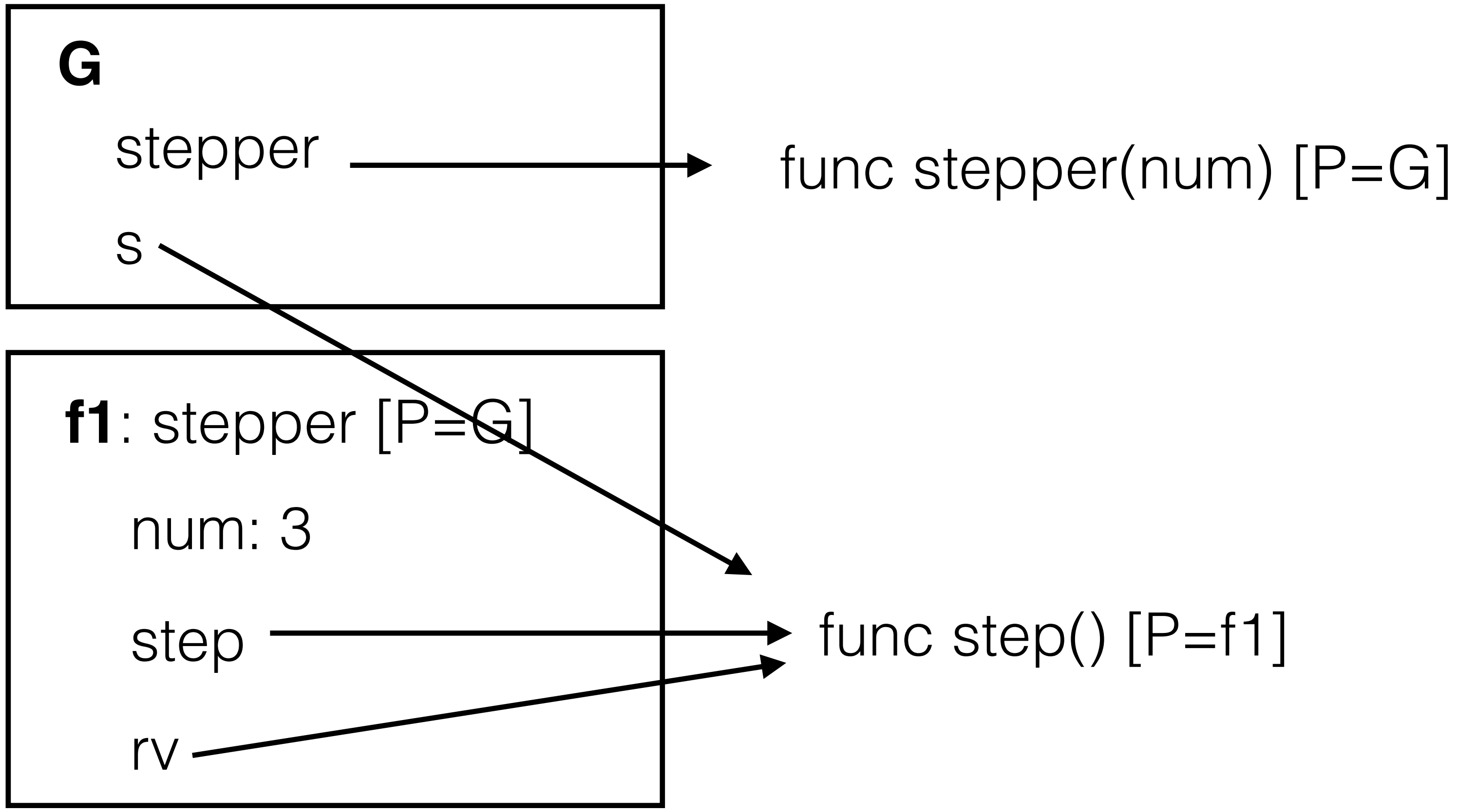
```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

f1



```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

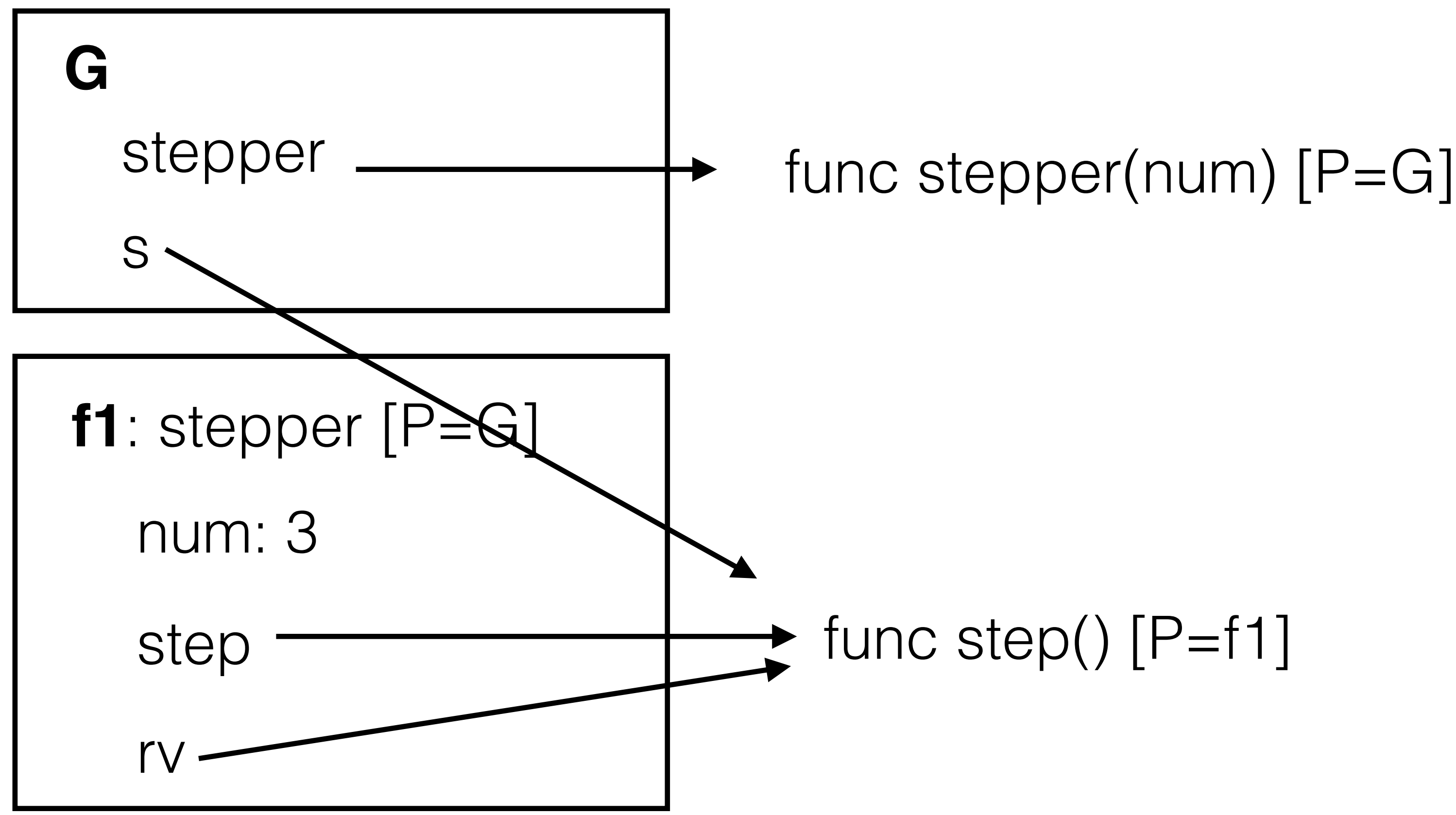
f1



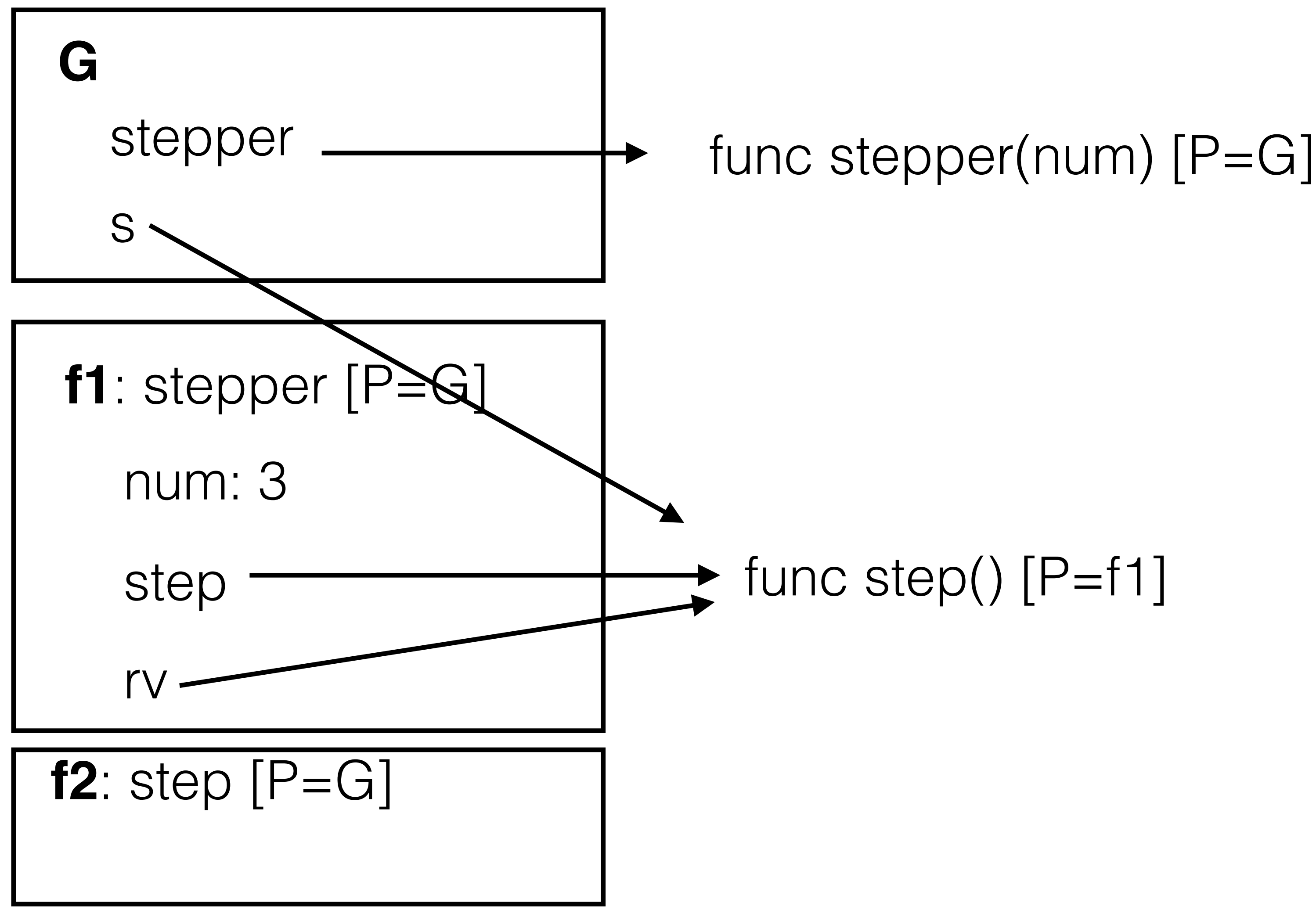
```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

f1

f2



```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)  f1
s() f2
s()
```

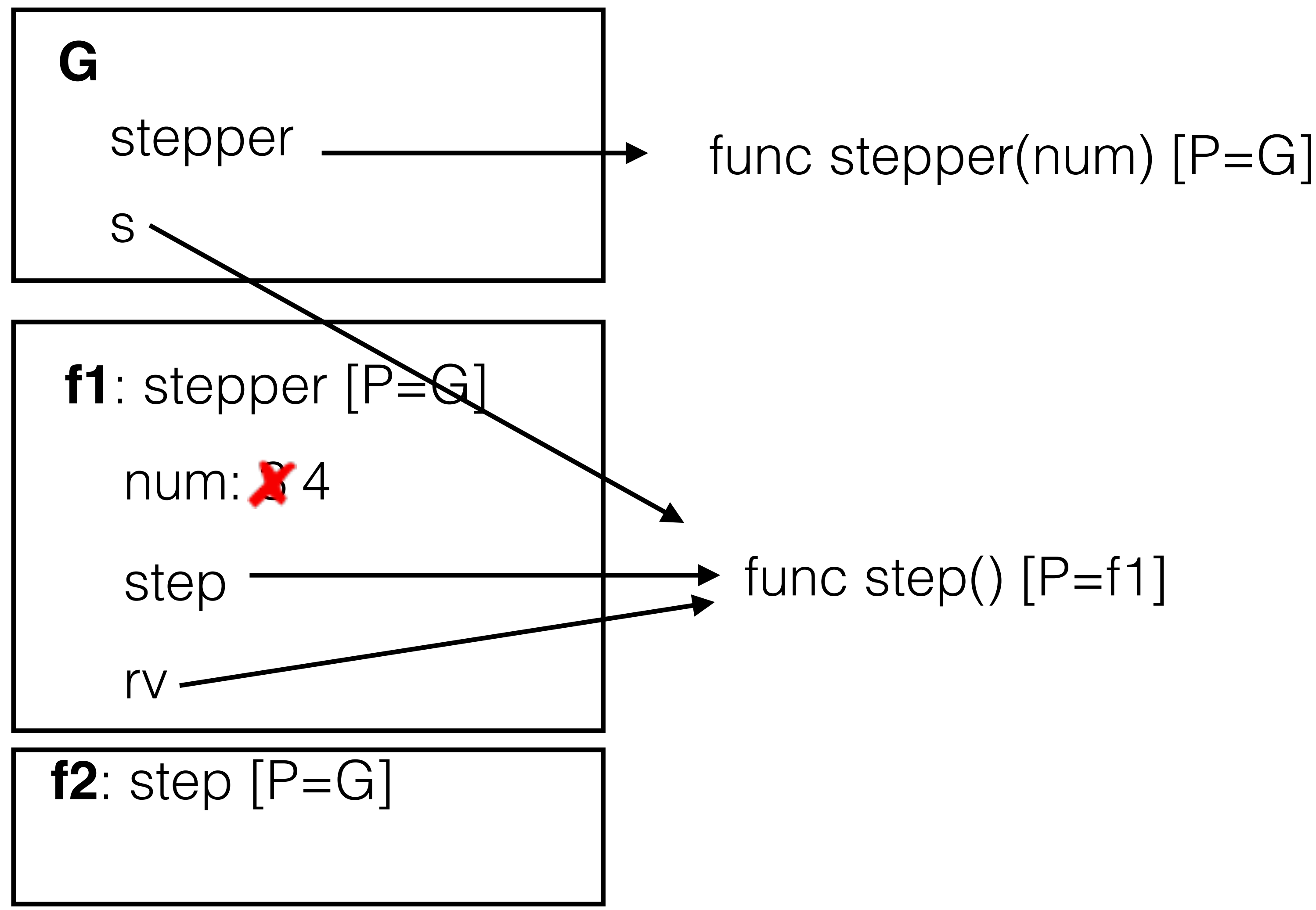



nonlocal num! (so the name `num` cannot appear anywhere on the left hand side in this frame!)

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

f1

f2



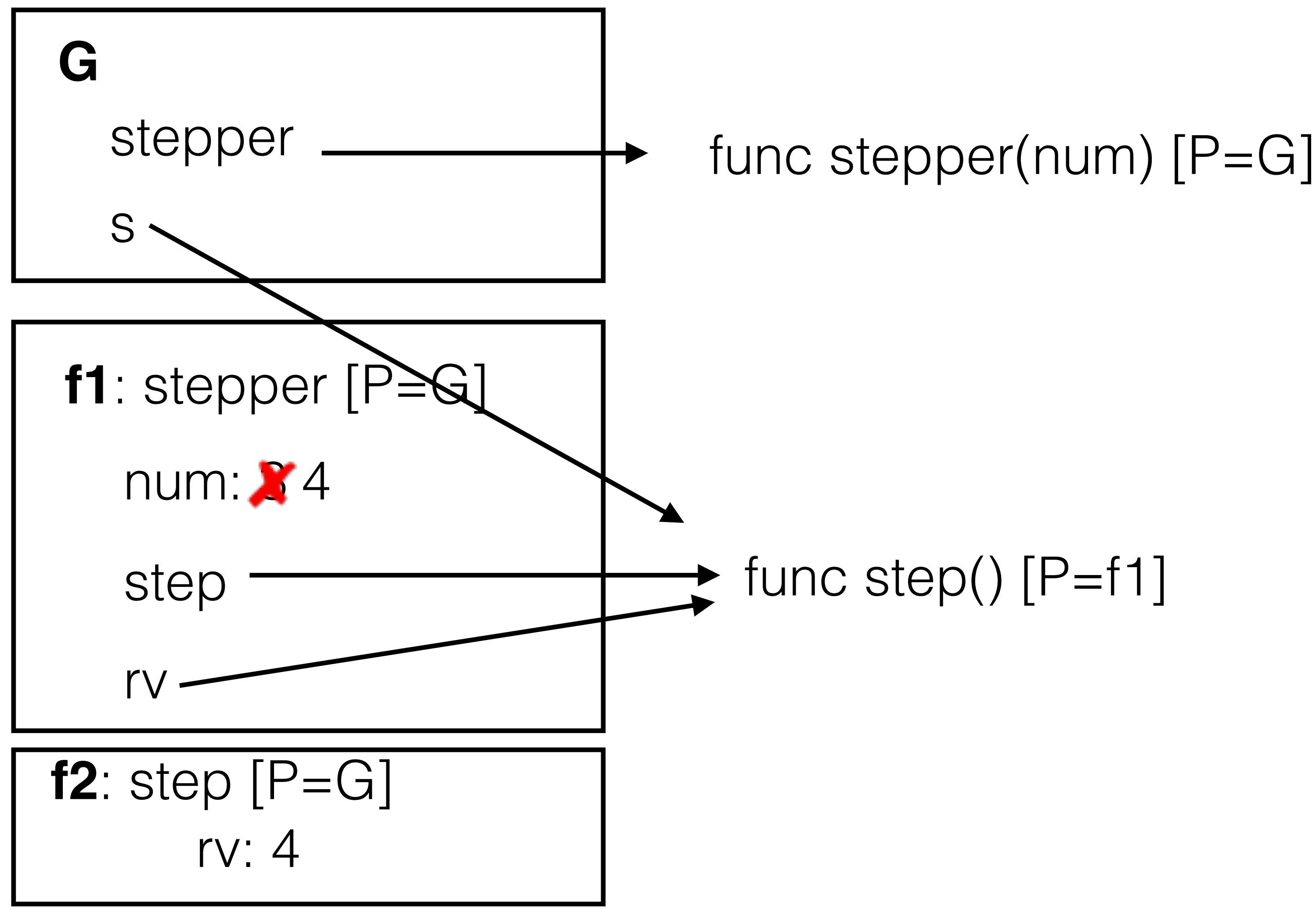

nonlocal num! (so the name `num` cannot appear anywhere on the left hand side in this frame!)

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

f1

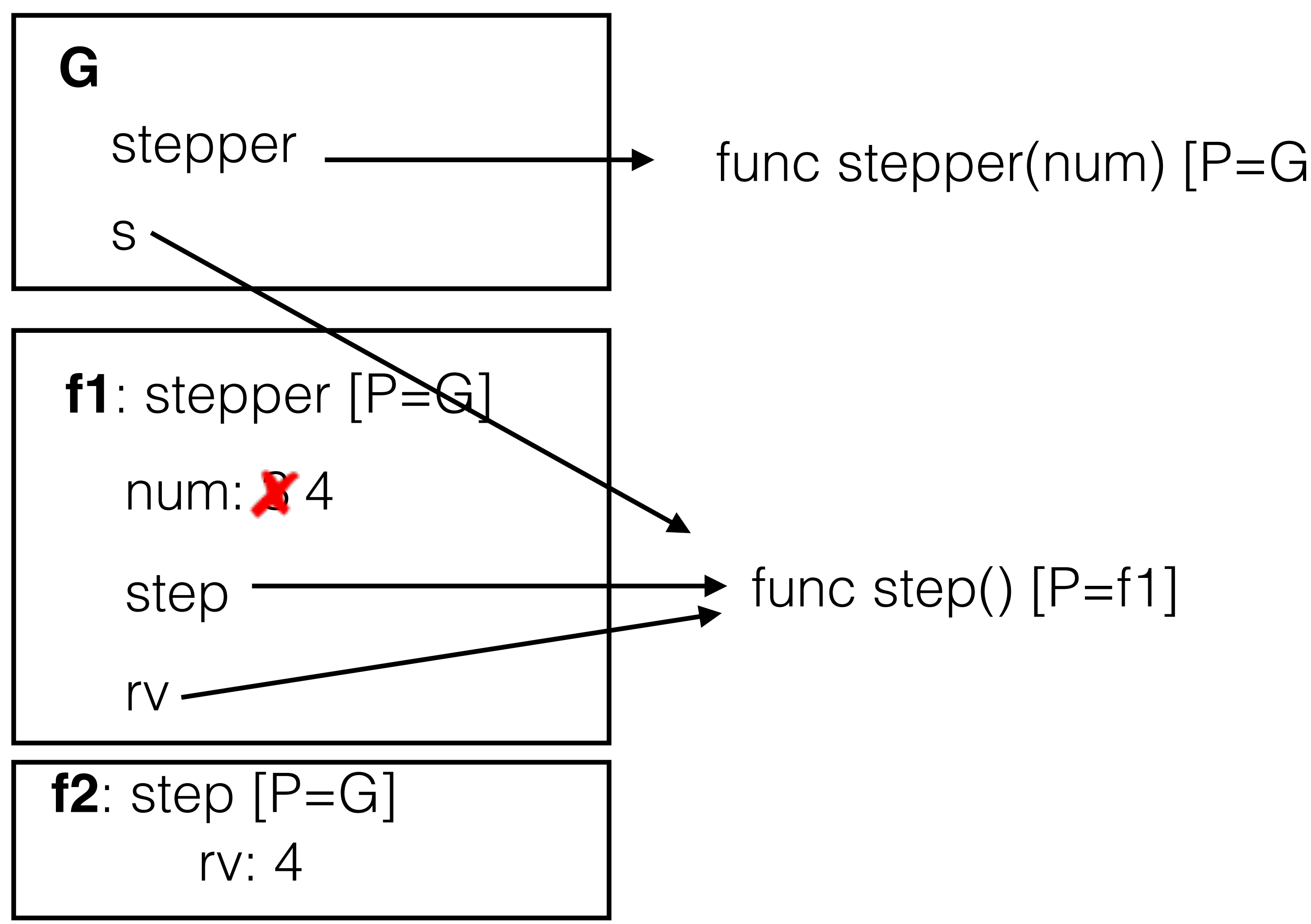
f2

nonlocal num! (so the name `num` cannot appear anywhere on the left hand side in this frame!)



```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3) f1
s() f2
s() f3
```

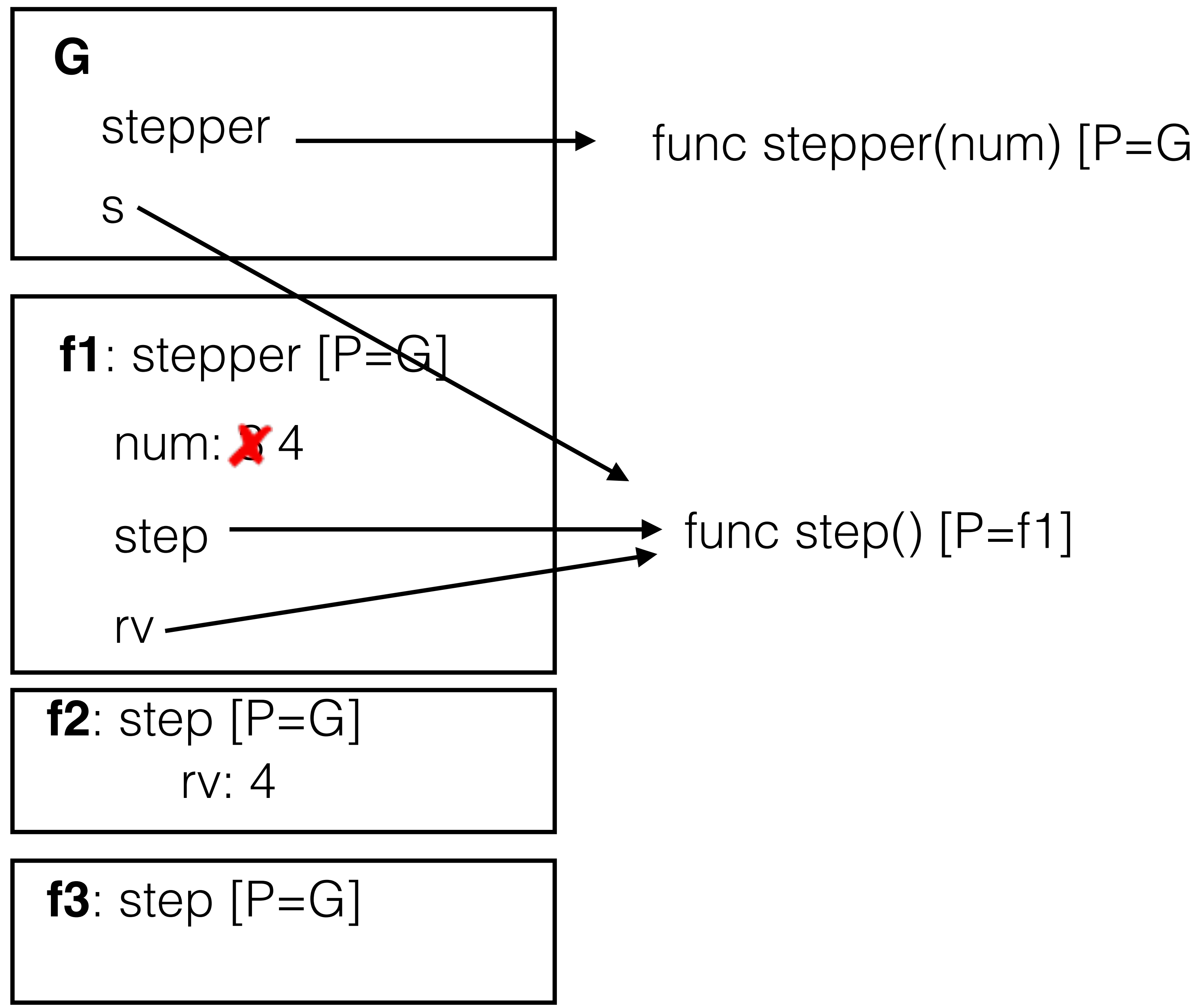
nonlocal num! (so the name `num` cannot appear anywhere on the left hand side in this frame!)



```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)  f1
s()  f2
s()  f3
```

nonlocal num! (so the name `num` cannot appear anywhere on the left hand side in this frame!)

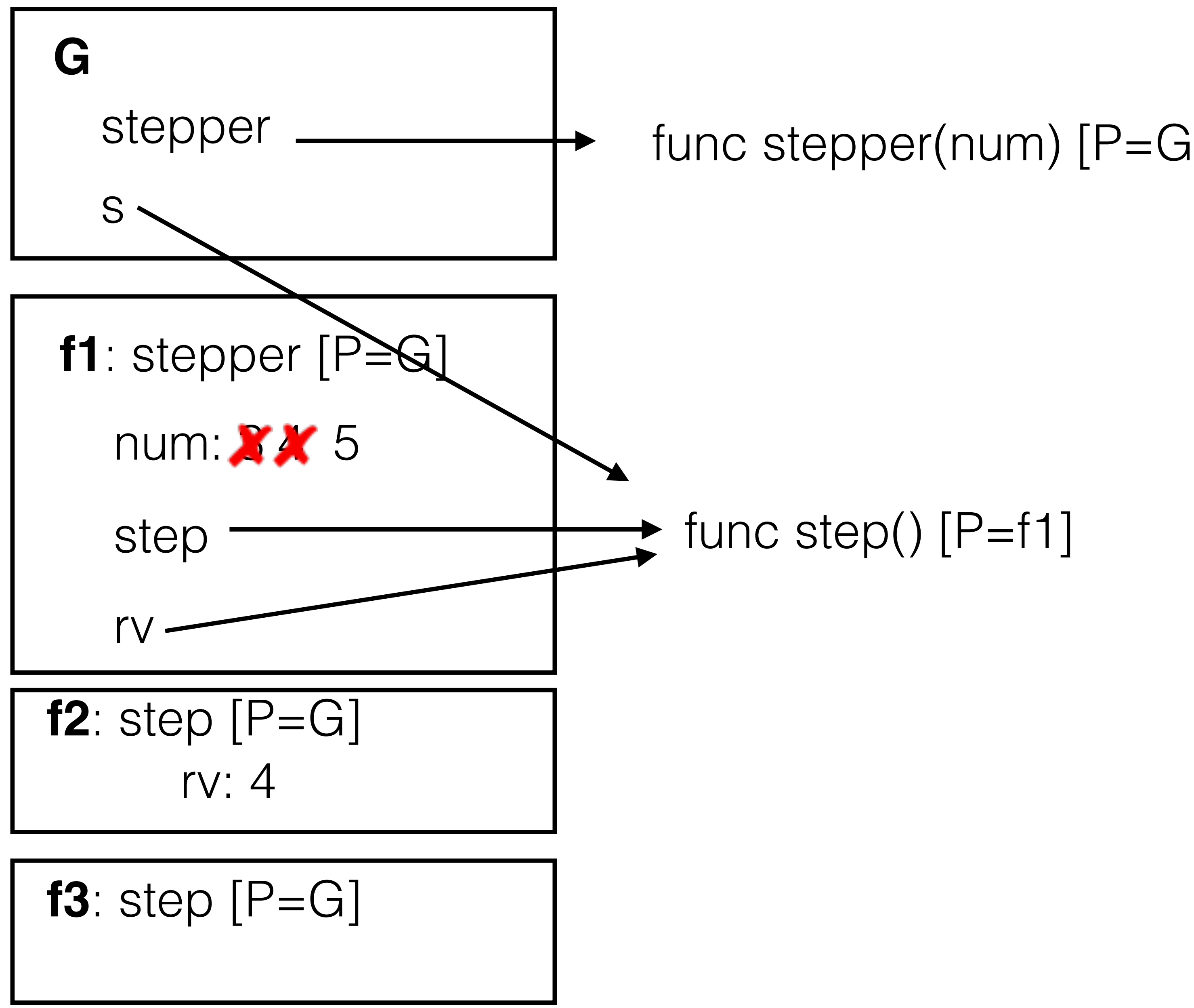
Again, nonlocal num!



```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)   f1
s()  f2
s()  f3
```

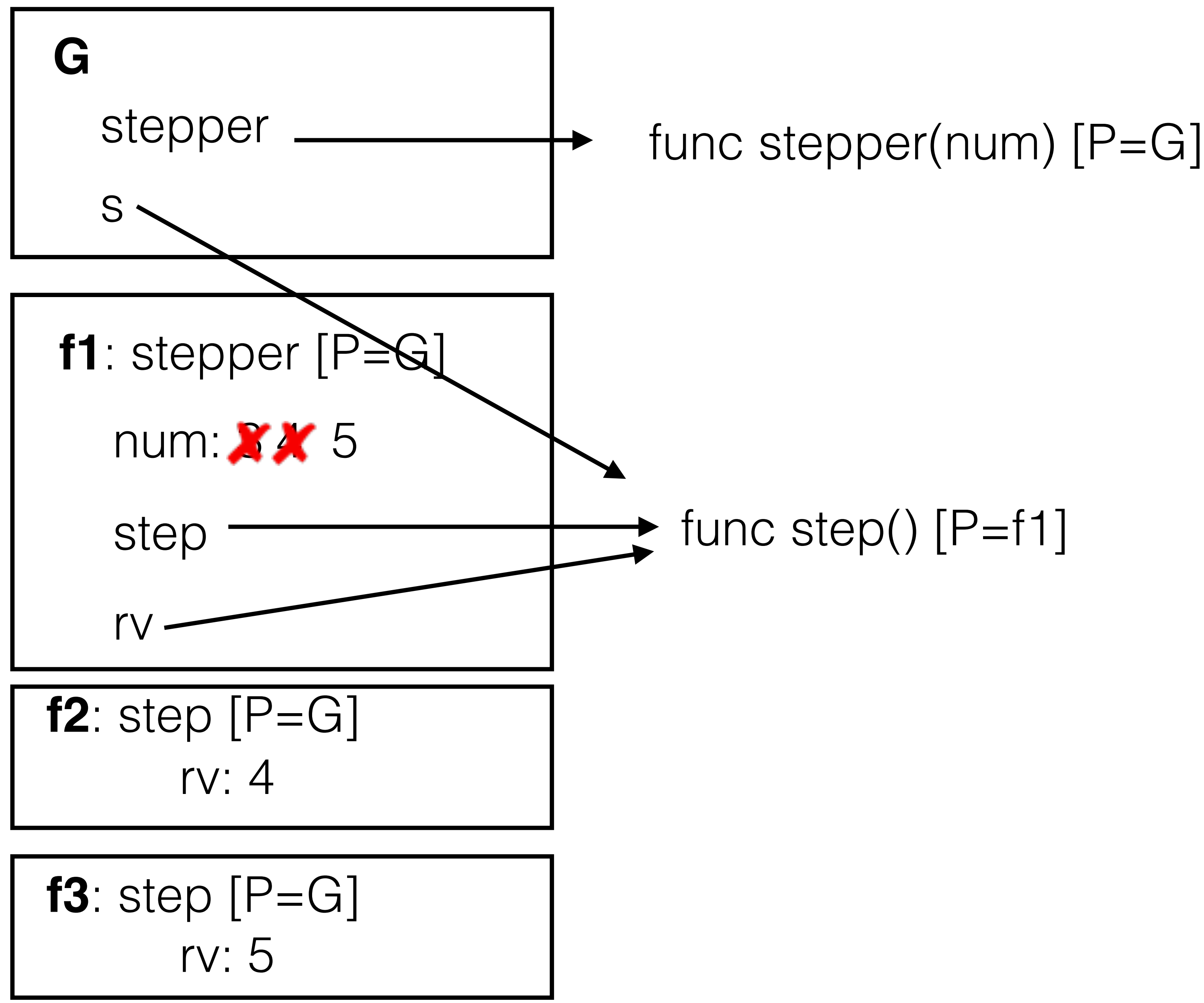
nonlocal num! (so the name `num` cannot appear anywhere on the left hand side in this frame!)

Again, nonlocal num!



```python
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

f1 f2 f3




nonlocal num! (so the name `num` cannot appear anywhere on the left hand side in this frame!)

Again, nonlocal num!