# 61A Lecture 9

Friday, September 19

# Announcements

# Announcements

- Midterm 1 is on Monday 2/9 from 7pm to 9pm

# Announcements

- Midterm 1 is on Monday 2/9 from 7pm to 9pm
  - Go to a room based on the first letter of your @berkeley.edu email:
    145 Dwinelle (A–B, #), 155 Dwinelle (C–K), & 1 Pimentel (L–Z)

# Announcements

- Midterm 1 is on Monday 2/9 from 7pm to 9pm
  - Go to a room based on the first letter of your @berkeley.edu email:
    145 Dwinelle (A–B, #), 155 Dwinelle (C–K), & 1 Pimentel (L–Z)
  - HKN review session on Saturday 2/7 (2050 VLSB 1–4)

# Announcements

- Midterm 1 is on Monday 2/9 from 7pm to 9pm
  - Go to a room based on the first letter of your @berkeley.edu email:
    145 Dwinelle (A–B, #), 155 Dwinelle (C–K), & 1 Pimentel (L–Z)
  - HKN review session on Saturday 2/7 (2050 VLSB 1–4)
  - Review sessions on Sunday 2/8 (1 Pimentel 1–2:30 & 2:30–4)

# Announcements

- Midterm 1 is on Monday 2/9 from 7pm to 9pm
  - Go to a room based on the first letter of your @berkeley.edu email:
    145 Dwinelle (A–B, #), 155 Dwinelle (C–K), & 1 Pimentel (L–Z)
  - HKN review session on Saturday 2/7 (2050 VLSB 1–4)
  - Review sessions on Sunday 2/8 (1 Pimentel 1–2:30 & 2:30–4)
  - Info: cs61a.org/exams/midterm1.html

# Announcements

- Midterm 1 is on Monday 2/9 from 7pm to 9pm
  - Go to a room based on the first letter of your @berkeley.edu email:
    145 Dwinelle (A–B, #), 155 Dwinelle (C–K), & 1 Pimentel (L–Z)
  - HKN review session on Saturday 2/7 (2050 VLSB 1–4)
  - Review sessions on Sunday 2/8 (1 Pimentel 1–2:30 & 2:30–4)
  - Info: cs61a.org/exams/midterm1.html
- No lecture on Monday 2/9

# Announcements

- Midterm 1 is on Monday 2/9 from 7pm to 9pm
  - Go to a room based on the first letter of your @berkeley.edu email:
    145 Dwinelle (A–B, #), 155 Dwinelle (C–K), & 1 Pimentel (L–Z)
  - HKN review session on Saturday 2/7 (2050 VLSB 1–4)
  - Review sessions on Sunday 2/8 (1 Pimentel 1–2:30 & 2:30–4)
  - Info: cs61a.org/exams/midterm1.html
- No lecture on Monday 2/9
- No lab or office hours on Tuesday 2/10 or Wednesday 2/11

# Announcements

- Midterm 1 is on Monday 2/9 from 7pm to 9pm
  - Go to a room based on the first letter of your @berkeley.edu email:
    145 Dwinelle (A–B, #), 155 Dwinelle (C–K), & 1 Pimentel (L–Z)
  - HKN review session on Saturday 2/7 (2050 VLSB 1–4)
  - Review sessions on Sunday 2/8 (1 Pimentel 1–2:30 & 2:30–4)
  - Info: cs61a.org/exams/midterm1.html
- No lecture on Monday 2/9
- No lab or office hours on Tuesday 2/10 or Wednesday 2/11
- Optional Hog strategy contest concludes Wednesday 2/18 @ 11:59pm

# Abstraction

# Functional Abstractions

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument.

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square.

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number.

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul.

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

```
def square(x):
    return pow(x, 2)
```

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

```
def square(x):
    return pow(x, 2)
```

```
def square(x):
    return mul(x, x-1) + x
```

# Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

```
def square(x):
    return pow(x, 2)
```

```
def square(x):
    return mul(x, x-1) + x
```

If the name "square" were bound to a built-in function, sum_squares would still work identically.

# Choosing Names

## Choosing Names

Names typically don’t matter for correctness

***but***

they matter a lot for composition

# Choosing Names

Names typically don’t matter for correctness

***but***

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

# Choosing Names

Names typically don’t matter for correctness

***but***

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

# Choosing Names

Names typically don’t matter for correctness

***but***

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

# Choosing Names

Names typically don’t matter for correctness

***but***

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

# Choosing Names

Names typically don't matter for correctness

***but***

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (print), their behavior (triple), or the value returned (abs).

# Choosing Names

Names typically don't matter for correctness

***but***

they matter a lot for composition

| From: | To: |
| --- | --- |
| | |

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (print), their behavior (triple), or the value returned (abs).

# Choosing Names

Names typically don't matter for correctness

***but***

they matter a lot for composition

| **From:** | **To:** |
| --- | --- |
| `true_false` | `rolled_a_one` |

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (print), their behavior (triple), or the value returned (abs).

# Choosing Names

Names typically don't matter for correctness

***but***

they matter a lot for composition

| **From:** | **To:** |
| --- | --- |
| `true_false` | `rolled_a_one` |
| `d` | `dice` |

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (print), their behavior (triple), or the value returned (abs).

# Choosing Names

Names typically don't matter for correctness

***but***

they matter a lot for composition

| **From:** | **To:** |
| --- | --- |
| `true_false` | `rolled_a_one` |
| `d` | `dice` |
| `helper` | `take_turn` |

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (print), their behavior (triple), or the value returned (abs).

# Choosing Names

Names typically don't matter for correctness

***but***

they matter a lot for composition

| **From:** | **To:** |
|---|---|
| `true_false` | `rolled_a_one` |
| `d` | `dice` |
| `helper` | `take_turn` |
| `my_int` | `num_rolls` |

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (print), their behavior (triple), or the value returned (abs).

# Choosing Names

Names typically don't matter for correctness

***but***

they matter a lot for composition

| **From:** | **To:** |
|---|---|
| `true_false` | `rolled_a_one` |
| `d` | `dice` |
| `helper` | `take_turn` |
| `my_int` | `num_rolls` |
| `l, I, O` | `k, i, m` |

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (print), their behavior (triple), or the value returned (abs).

# Which Values Deserve a Name

**Reasons to add a new name**

## Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

*Meaningful parts of complex expressions:*

# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

*Meaningful parts of complex expressions:*

```
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

*Meaningful parts of complex expressions:*

```
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = sqrt(square(b) - 4 * a * c)
x = (-b + discriminant) / (2 * a)
```

# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

*Meaningful parts of complex expressions:*

```
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = sqrt(square(b) - 4 * a * c)
x = (-b + discriminant) / (2 * a)
```

**More Naming Tips**

# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

*Meaningful parts of complex expressions:*

```
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = sqrt(square(b) - 4 * a * c)
x = (-b + discriminant) / (2 * a)
```

**More Naming Tips**

- Names can be long if they help document your code:

```
average_age = average(age, students)
```

is preferable to

```
# Compute average age of students
aa = avg(a, st)
```

# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

*Meaningful parts of complex expressions:*

```
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = sqrt(square(b) - 4 * a * c)
x = (-b + discriminant) / (2 * a)
```

**More Naming Tips**

- Names can be long if they help document your code:

```
average_age = average(age, students)
```

is preferable to

```
# Compute average age of students
aa = avg(a, st)
```

# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

*Meaningful parts of complex expressions:*

```
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = sqrt(square(b) - 4 * a * c)
x = (-b + discriminant) / (2 * a)
```

**More Naming Tips**

- Names can be long if they help document your code:

```
average_age = average(age, students)
```

is preferable to

```
# Compute average age of students
aa = avg(a, st)
```

- Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.

```
n, k, i - Usually integers
x, y, z - Usually real numbers
f, g, h - Usually functions
```

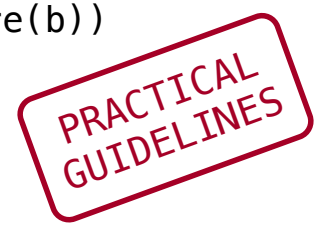
# Which Values Deserve a Name

**Reasons to add a new name**

*Repeated compound expressions:*

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```




*Meaningful parts of complex expressions:*

```
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
discriminant = sqrt(square(b) - 4 * a * c)
x = (-b + discriminant) / (2 * a)
```

**More Naming Tips**

- Names can be long if they help document your code:

```
average_age = average(age, students)
```

is preferable to

```
# Compute average age of students
aa = avg(a, st)
```

- Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.

```
n, k, i - Usually integers
x, y, z - Usually real numbers
f, g, h - Usually functions
```

# Testing

# Test-Driven Development

# Test-Driven Development

Write the test of a function before you write the function.

## Test-Driven Development

Write the test of a function before you write the function.

*A test will clarify the domain, range, & behavior of a function.*

## Test-Driven Development

Write the test of a function before you write the function.

*A test will clarify the domain, range, & behavior of a function.*

*Tests can help identify tricky edge cases.*

## Test-Driven Development

Write the test of a function before you write the function.

*A test will clarify the domain, range, & behavior of a function.*

*Tests can help identify tricky edge cases.*

Develop incrementally and test each piece before moving on.

# Test-Driven Development

Write the test of a function before you write the function.

*A test will clarify the domain, range, & behavior of a function.*

*Tests can help identify tricky edge cases.*

Develop incrementally and test each piece before moving on.

*You can't depend upon code that hasn't been tested.*

# Test-Driven Development

Write the test of a function before you write the function.

*A test will clarify the domain, range, & behavior of a function.*

*Tests can help identify tricky edge cases.*

Develop incrementally and test each piece before moving on.

*You can't depend upon code that hasn't been tested.*

*Run your old tests again after you make new changes.*

## Test-Driven Development

Write the test of a function before you write the function.

*A test will clarify the domain, range, & behavior of a function.*

*Tests can help identify tricky edge cases.*

Develop incrementally and test each piece before moving on.

*You can't depend upon code that hasn't been tested.*

*Run your old tests again after you make new changes.*

Bonus idea: Run your code interactively.

## Test-Driven Development

Write the test of a function before you write the function.

*A test will clarify the domain, range, & behavior of a function.*

*Tests can help identify tricky edge cases.*

Develop incrementally and test each piece before moving on.

*You can't depend upon code that hasn't been tested.*

*Run your old tests again after you make new changes.*

Bonus idea: Run your code interactively.

*Don't be afraid to experiment with a function after you write it.*

## Test-Driven Development

Write the test of a function before you write the function.

*A test will clarify the domain, range, & behavior of a function.*

*Tests can help identify tricky edge cases.*

Develop incrementally and test each piece before moving on.

*You can't depend upon code that hasn't been tested.*

*Run your old tests again after you make new changes.*

Bonus idea: Run your code interactively.

*Don't be afraid to experiment with a function after you write it.*

*Interactive sessions can become doctests. Just copy and paste.*

## Test-Driven Development

Write the test of a function before you write the function.

*A test will clarify the domain, range, & behavior of a function.*

*Tests can help identify tricky edge cases.*

Develop incrementally and test each piece before moving on.

*You can't depend upon code that hasn't been tested.*

*Run your old tests again after you make new changes.*

Bonus idea: Run your code interactively.

*Don't be afraid to experiment with a function after you write it.*

*Interactive sessions can become doctests. Just copy and paste.*

(Demo)

# Decorators

# Function Decorators

(Demo)

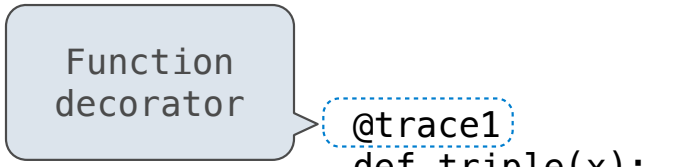
# Function Decorators

(Demo)

```
@trace1
def triple(x):
    return 3 * x
```

# Function Decorators

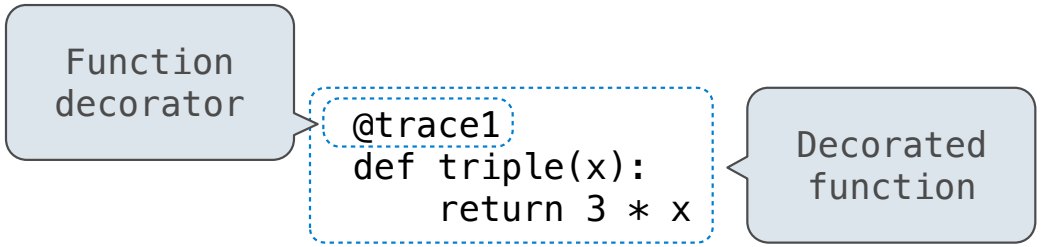
(Demo)




```
@trace1
def triple(x):
    return 3 * x
```

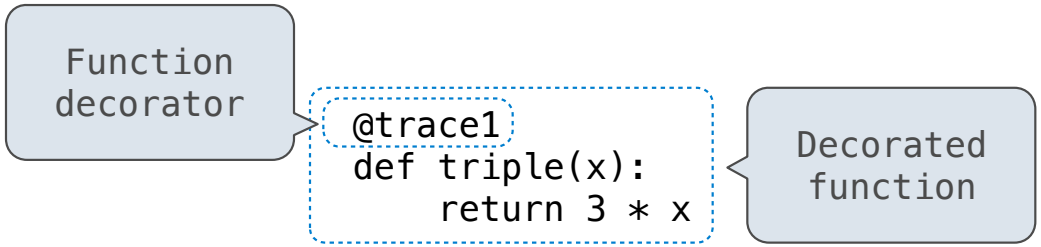
# Function Decorators

(Demo)



# Function Decorators

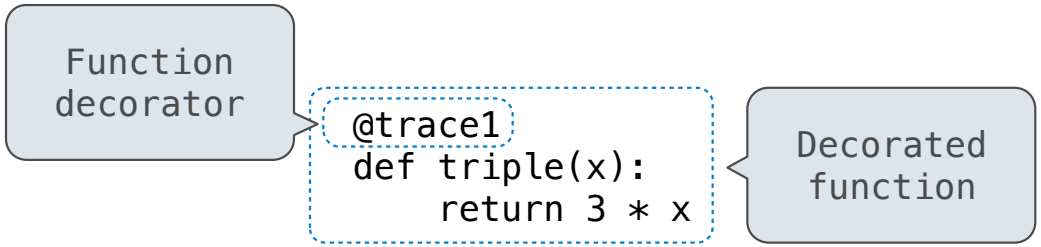
(Demo)




*is identical to*

# Function Decorators

(Demo)

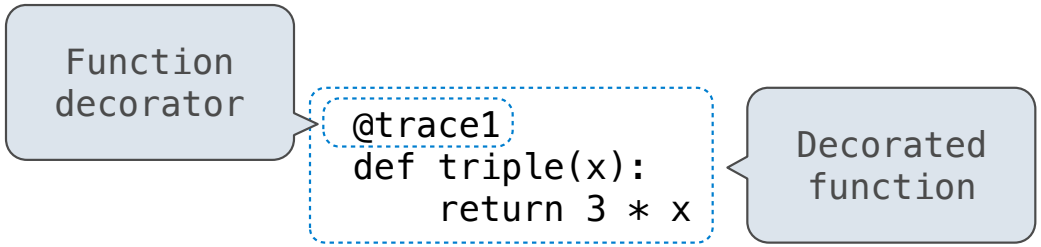



*is identical to*

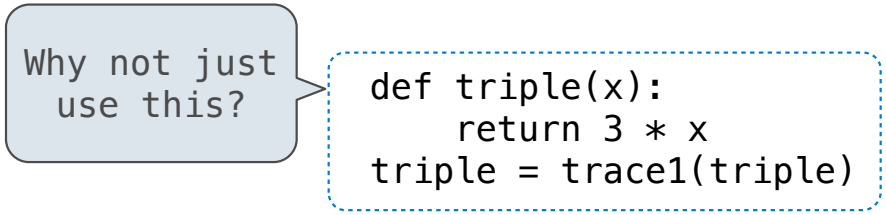
```
def triple(x):
    return 3 * x
triple = trace1(triple)
```

# Function Decorators

(Demo)




*is identical to*



# Currying

# Function Currying

# Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

# Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

# Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

# Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

(Demo)

# Function Currying

```
def make_adder(n):
    return lambda k: n + k
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

(Demo)

**Curry:** Transform a multi-argument function into a single-argument, higher-order function

# Review

# What Would Python Print?

# What Would Python Print?

```
The print function returns None.  It also displays its arguments
(separated by spaces) when it is called.
```

# What Would Python Print?

The print function returns None.  It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
| --- | --- | --- |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
| --- | --- | --- |
| 5 | 5 | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
| --- | --- | --- |
| 5 | 5 | 5 |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
| --- | --- | --- |
| `5` | `5` | `5` |
| `print(5)` | | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
| --- | --- | --- |
| `5` | `5` | `5` |
| `print(5)` | `None` | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
| --- | --- | --- |
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |

None

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |

`None`

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |

`None`

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

`delay(delay)()(6)()`

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |

`print(print(5))` → `None`

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

`delay(delay)()(6)()`

Names in nested def statements can refer to their enclosing scope

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | | |

`print(5)` → None

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | | `delayed` |

None

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | | `delayed`<br>`delayed` |

None

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | | `delayed`<br>`delayed`<br>`6` |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | `6` | `delayed`<br>`delayed`<br>`6` |

None

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | `6` | `delayed`<br>`delayed`<br>`6` |
| `print(delay(print)()(4))` | | |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | `6` | `delayed`<br>`delayed`<br>`6` |
| `print(delay(print)()(4))` | | `delayed` |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | `6` | `delayed`<br>`delayed`<br>`6` |
| `print(delay(print)()(4))` | | `delayed`<br>`4` |

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | `6` | `delayed`<br>`delayed`<br>`6` |
| `print(delay(print)()(4))` | | `delayed`<br>`4`<br>`None` |

None

# What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul
def square(x):
    return mul(x, x)
```

A function that takes any argument and returns a function that returns that arg

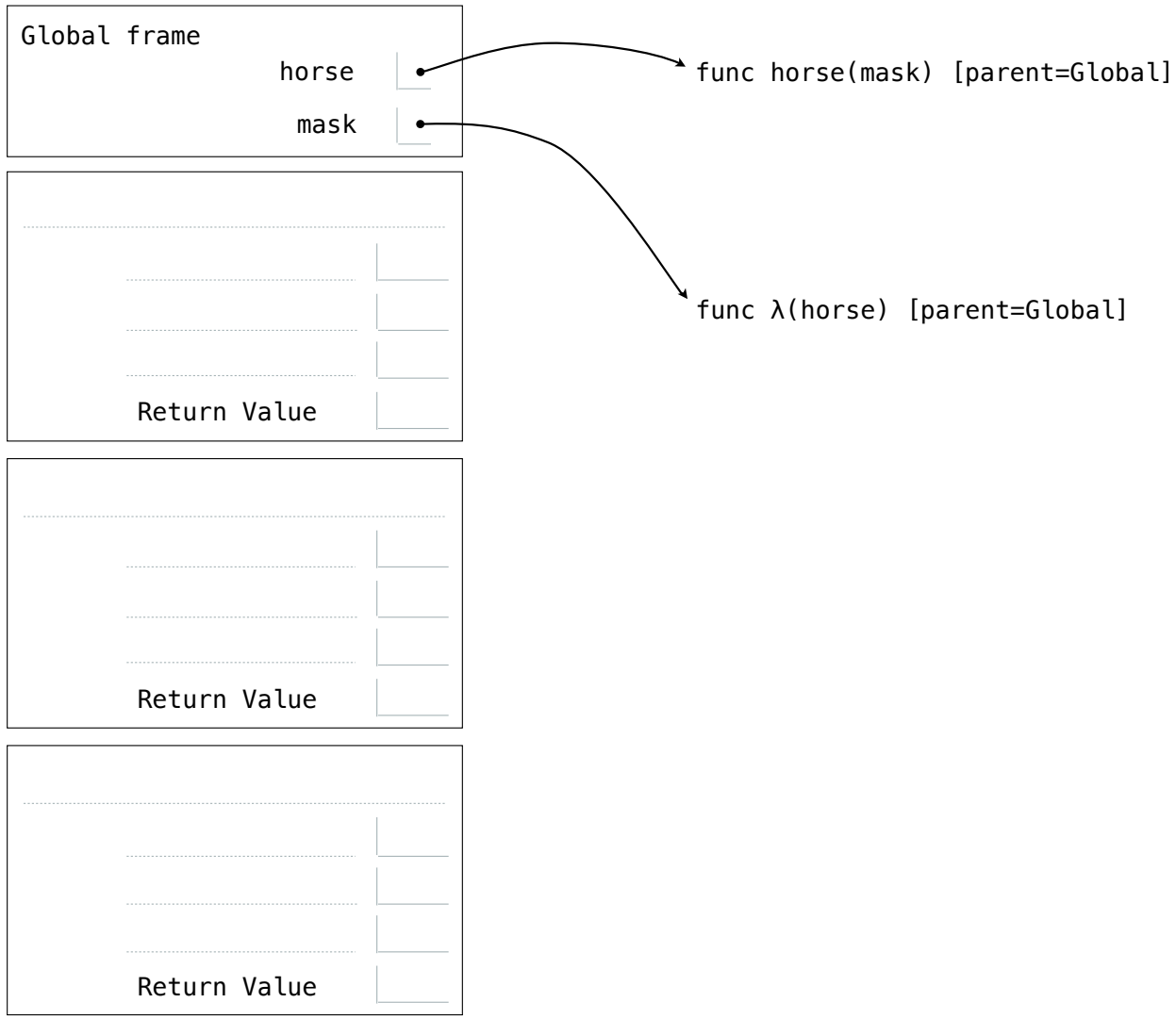
```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

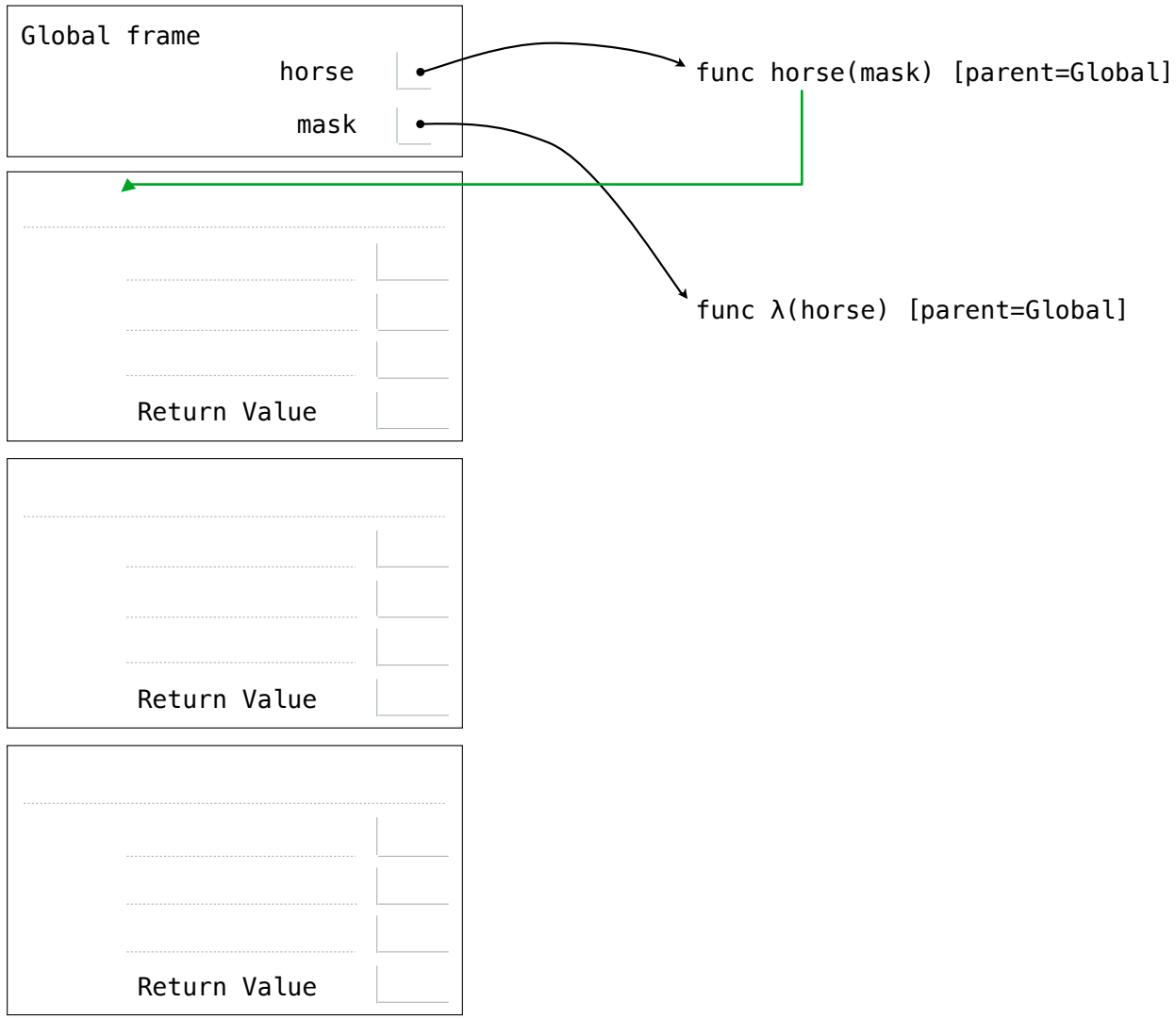
| This expression | Evaluates to | Interactive Output |
|---|---|---|
| `5` | `5` | `5` |
| `print(5)` | `None` | `5` |
| `print(print(5))` | `None` | `5`<br>`None` |
| `delay(delay)()(6)()` | `6` | `delayed`<br>`delayed`<br>`6` |
| `print(delay(print)()(4))` | `None` | `delayed`<br>`4`<br>`None` |

None

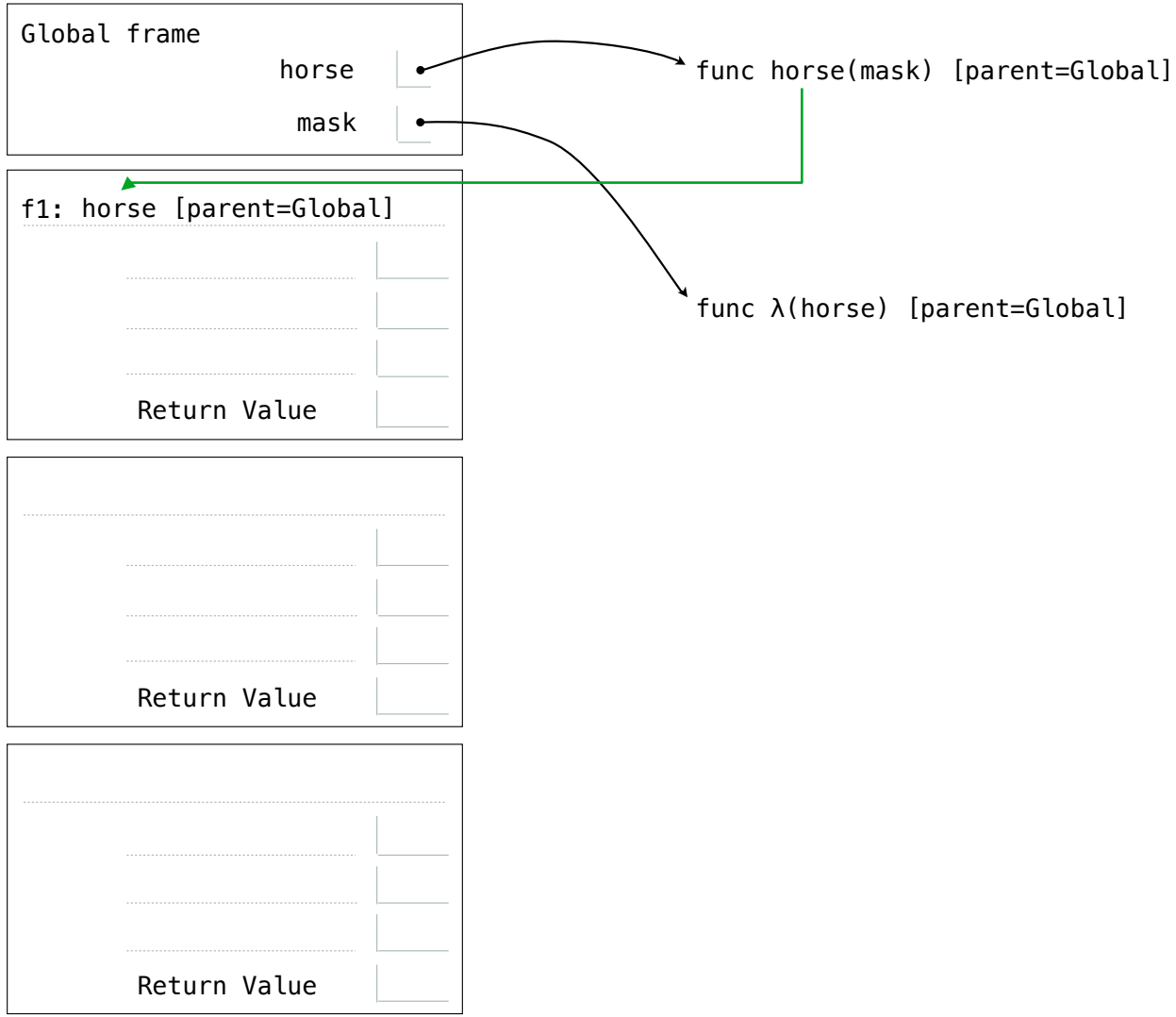
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



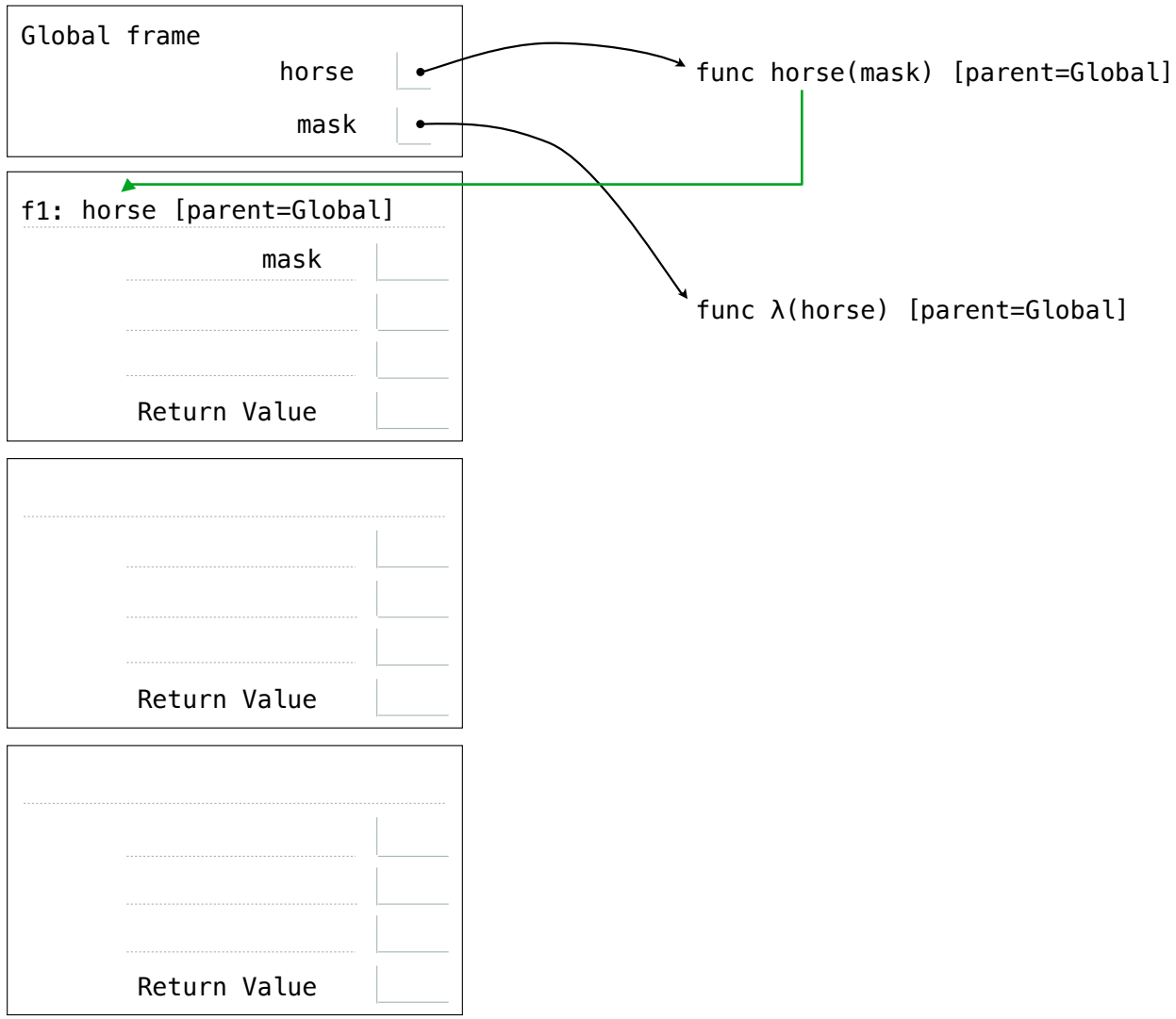
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



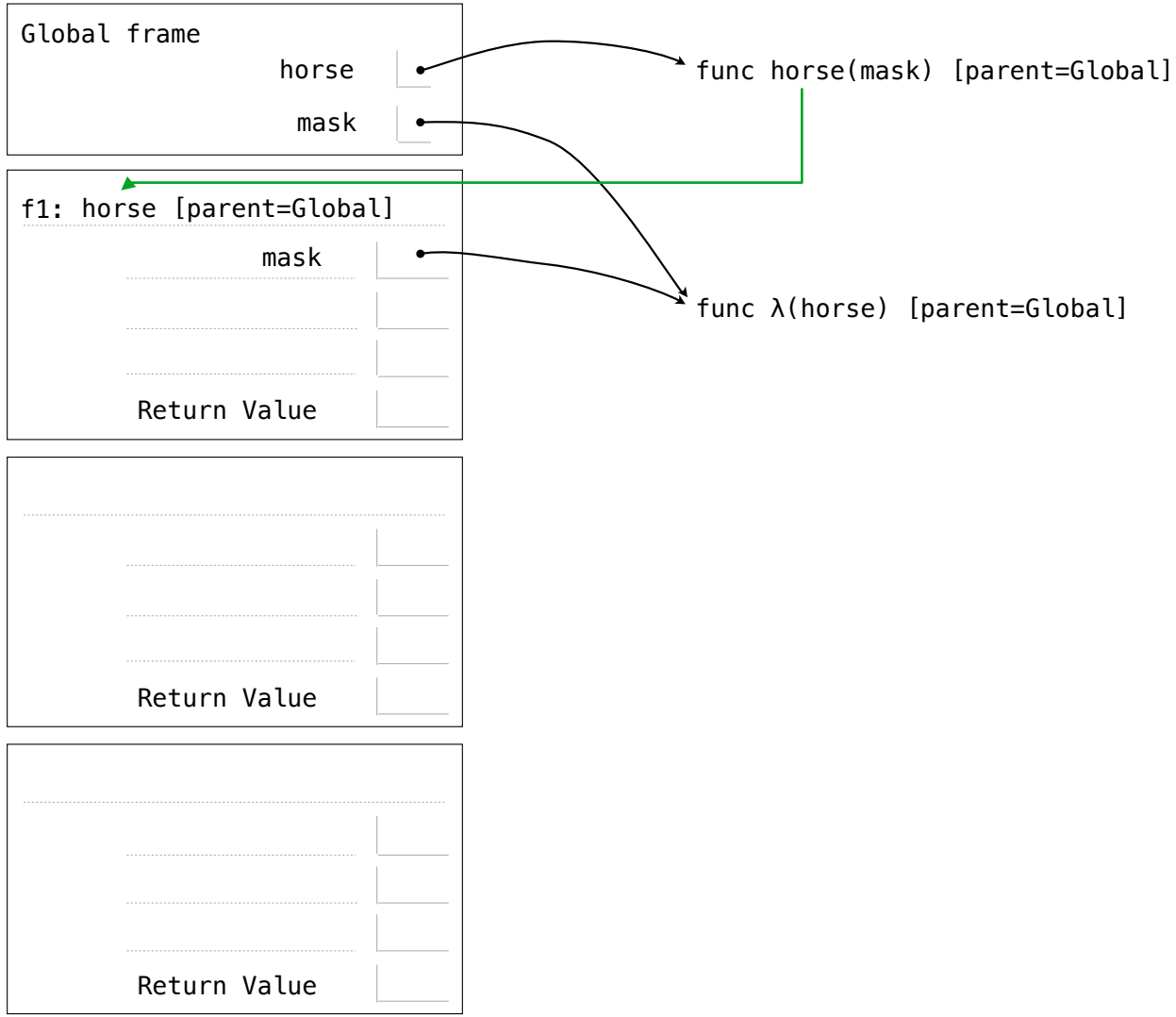
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



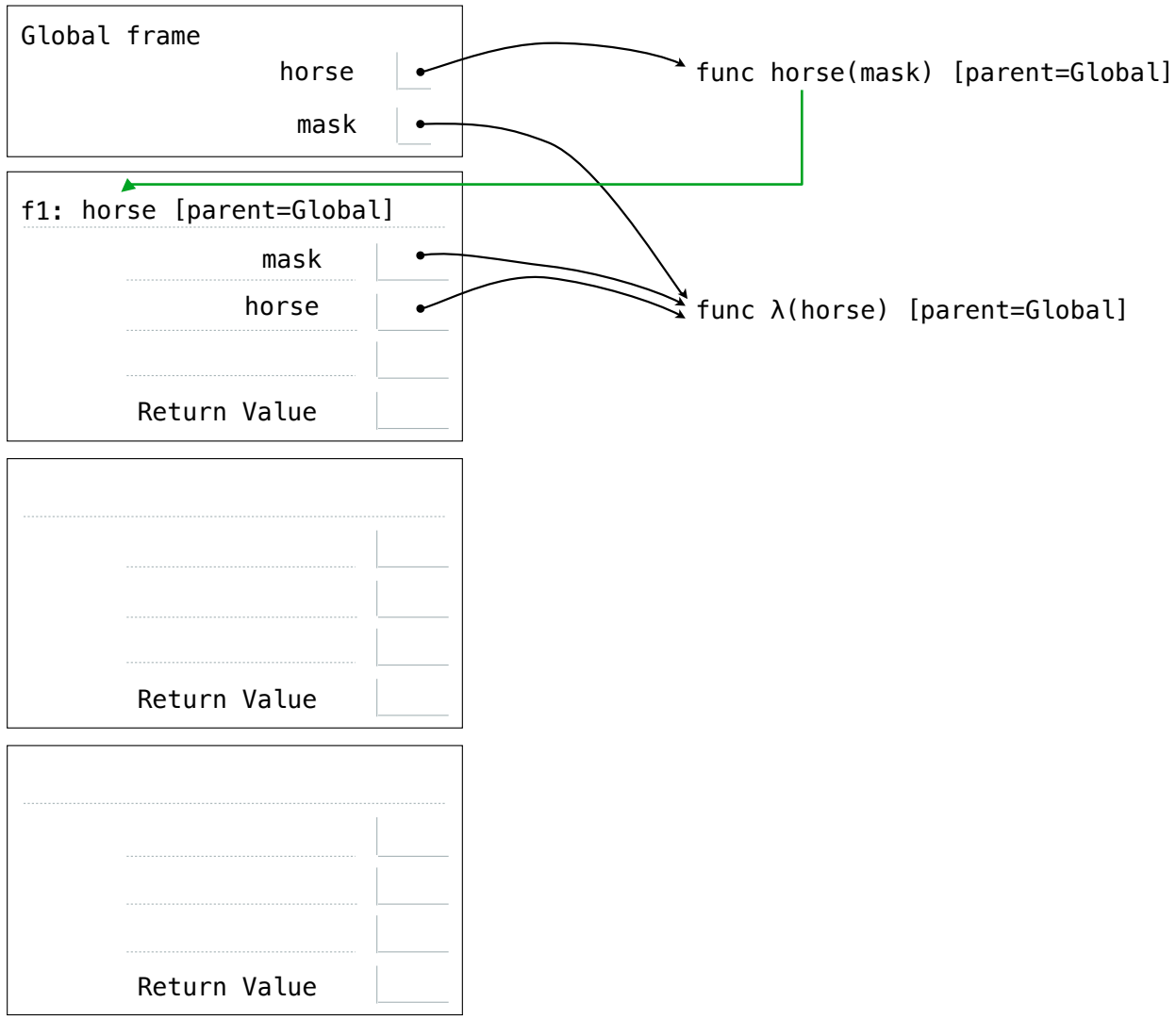
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



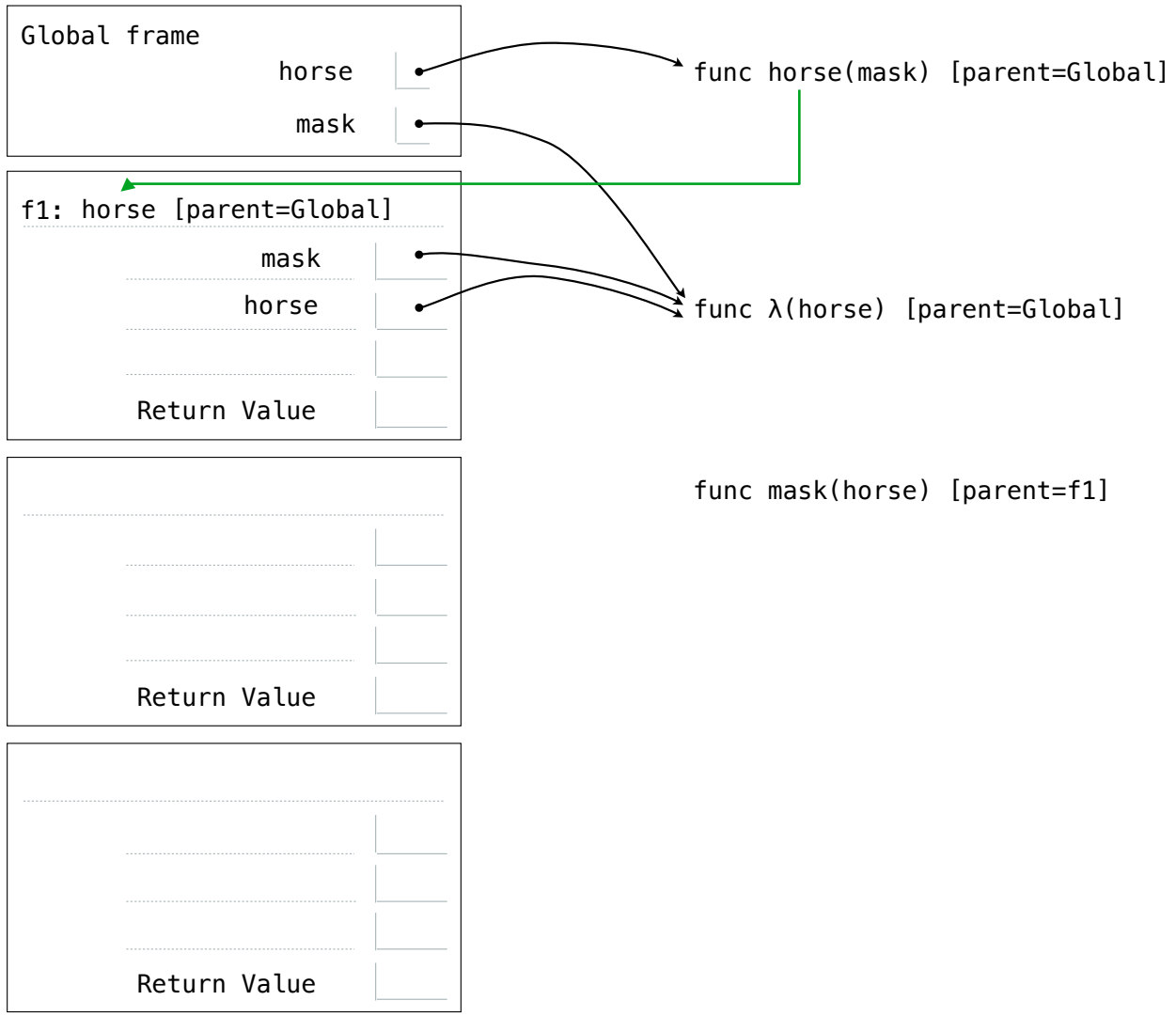
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



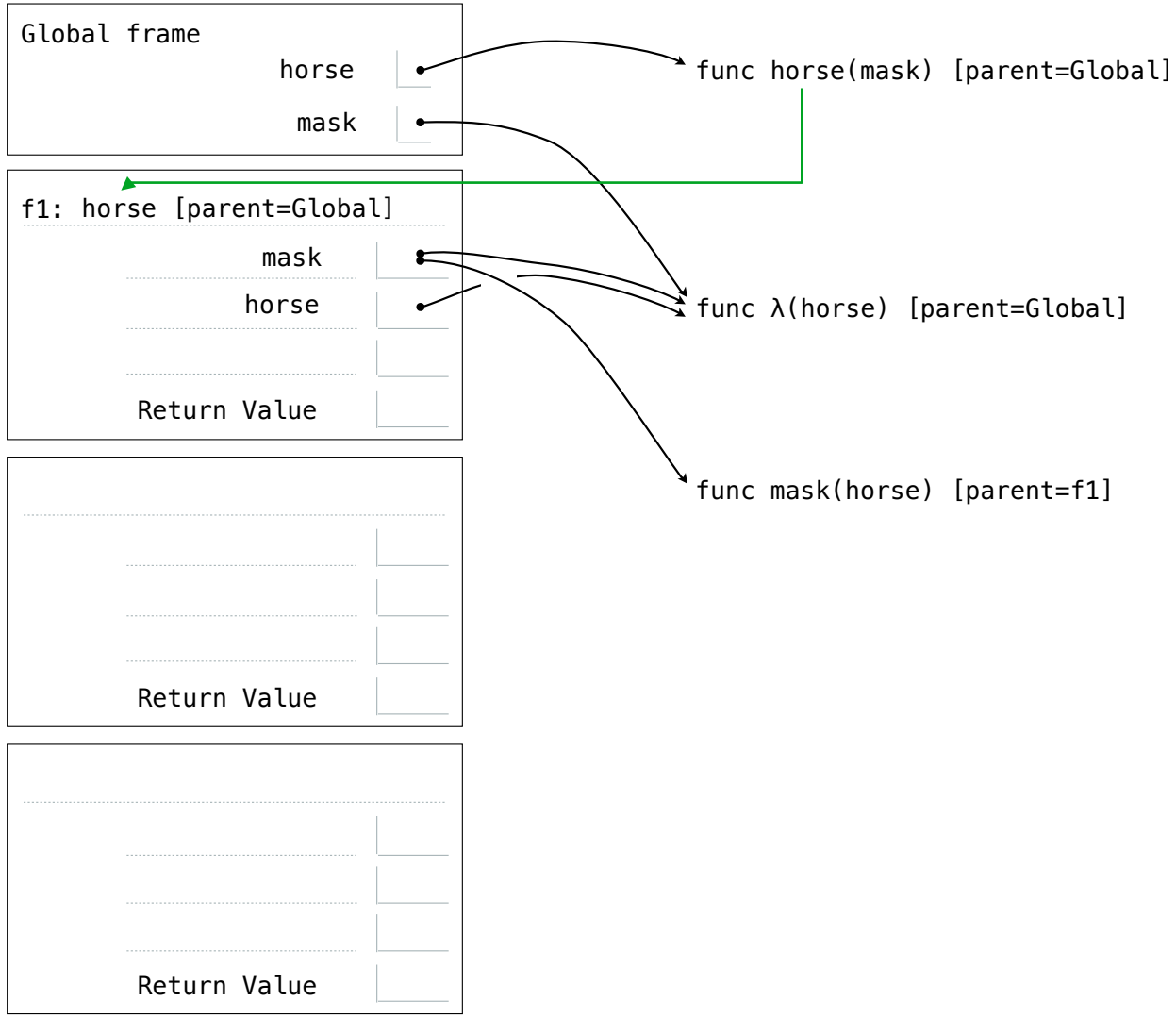
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



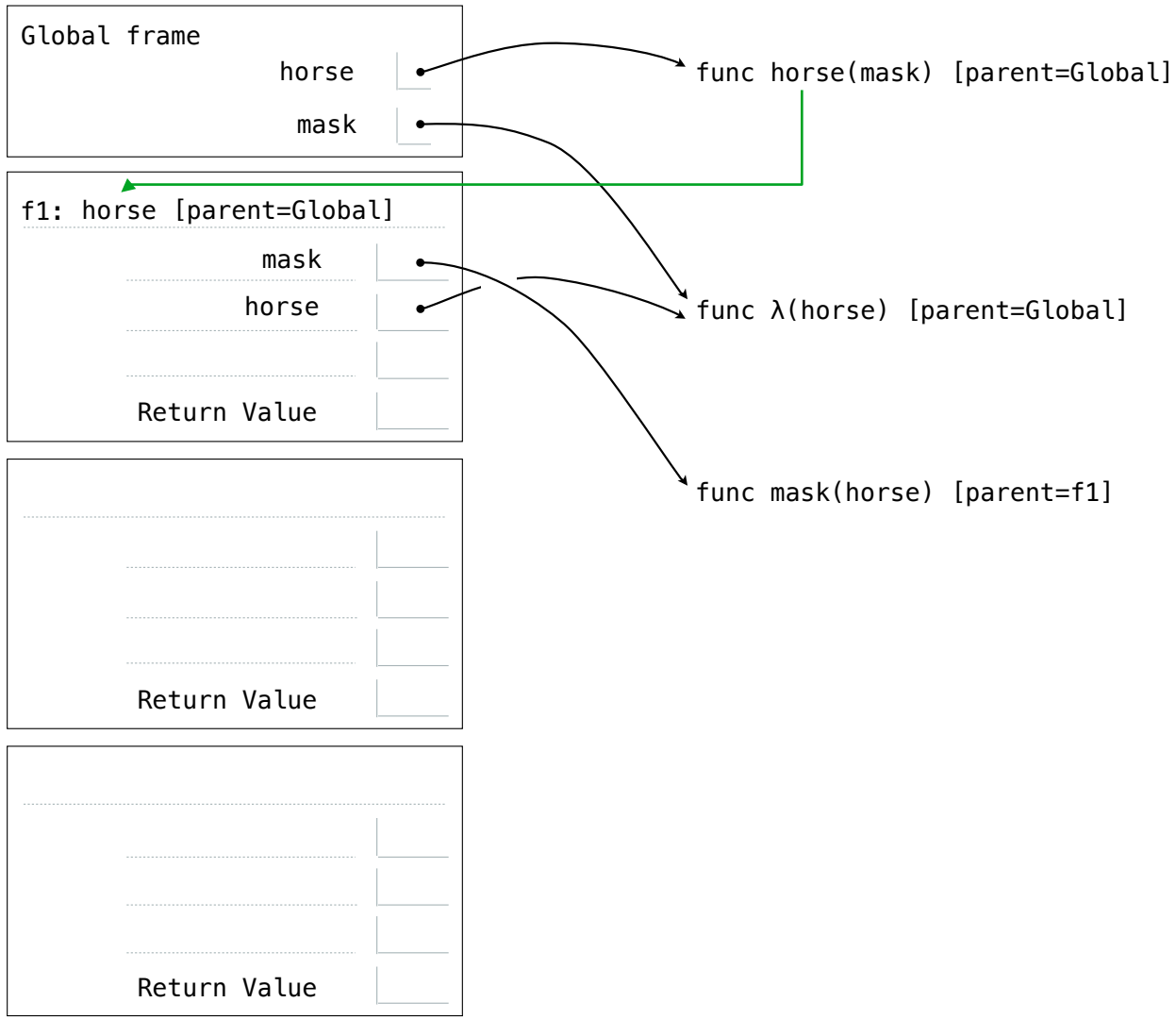
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



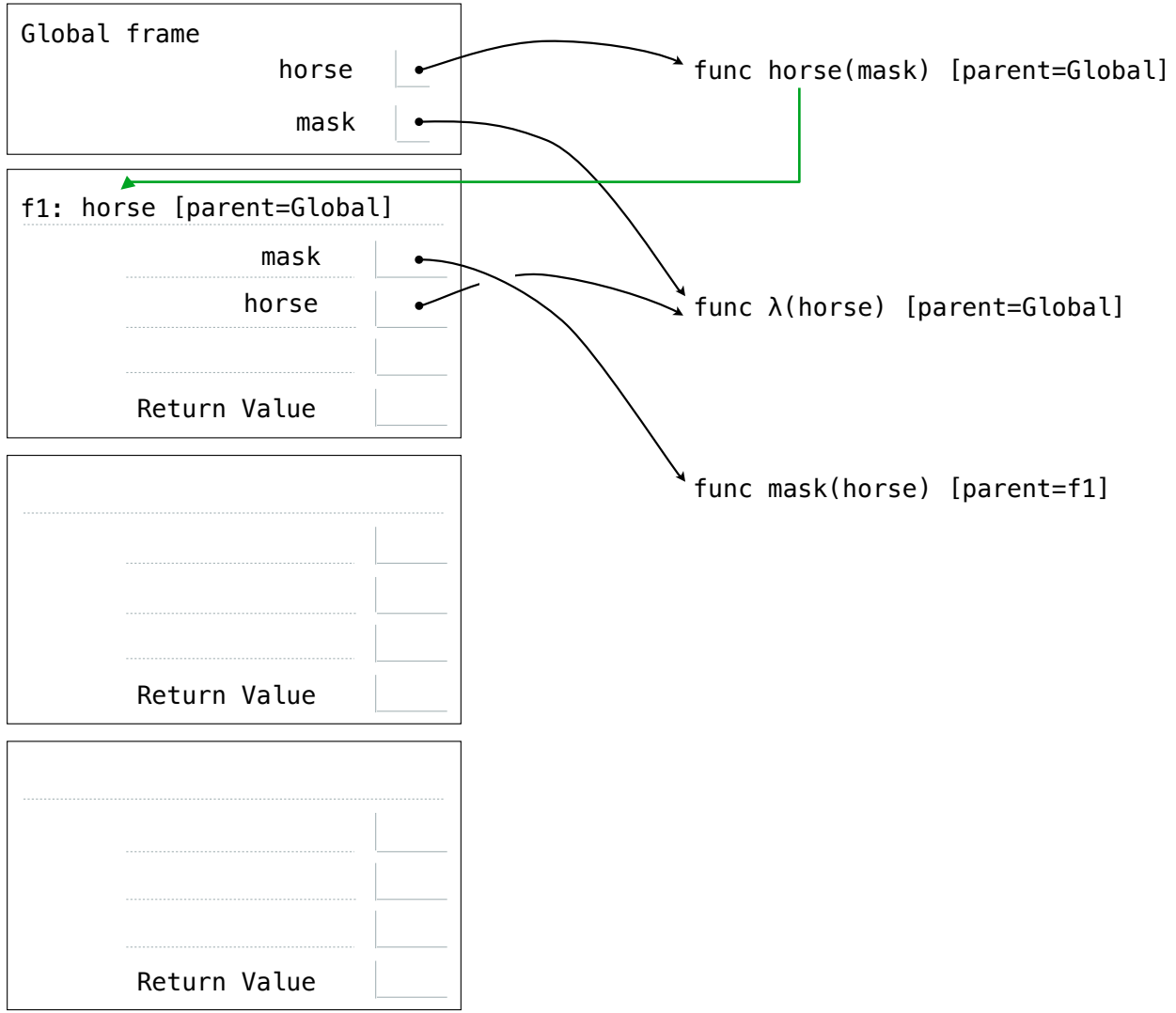
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```



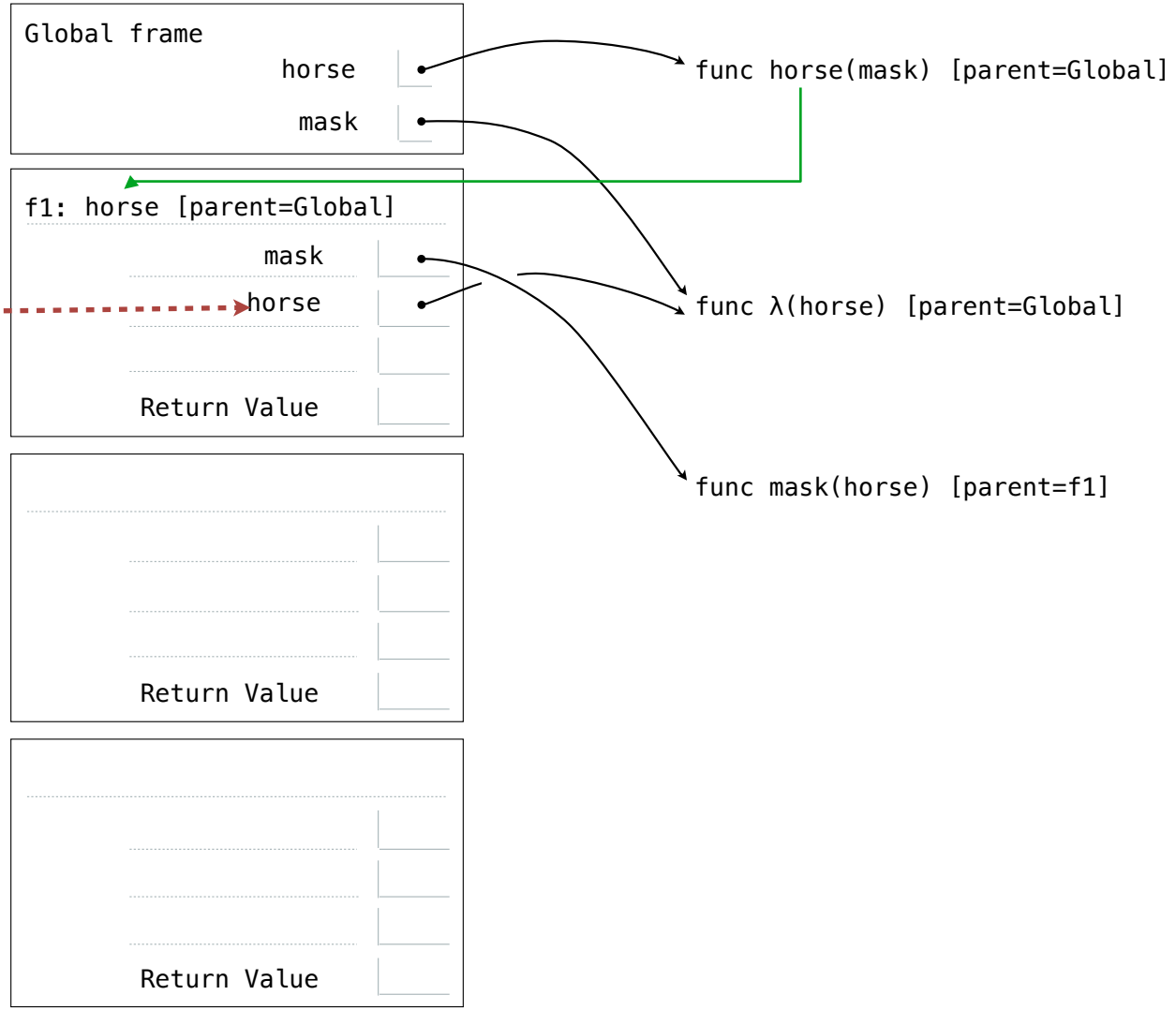
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```

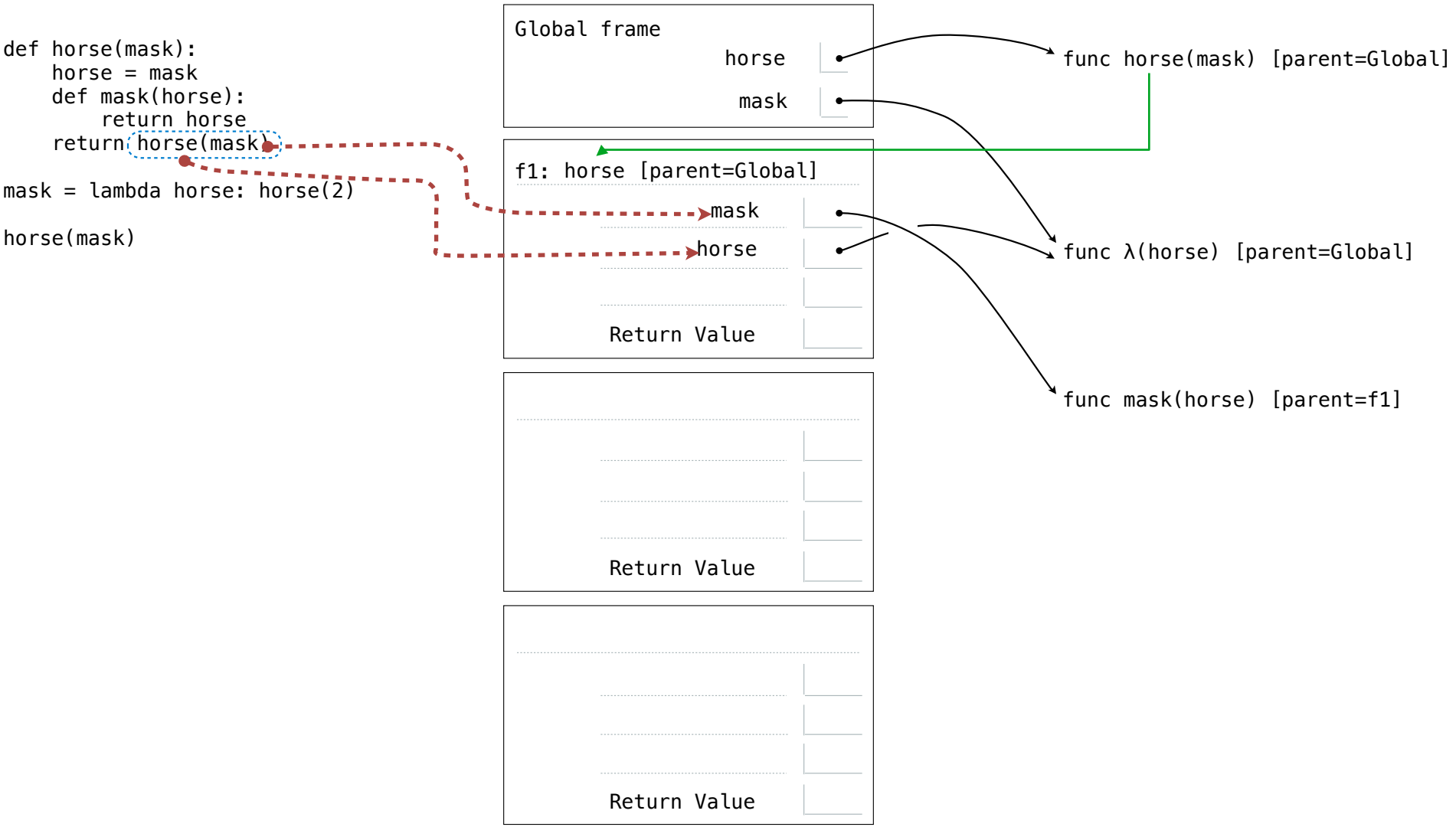


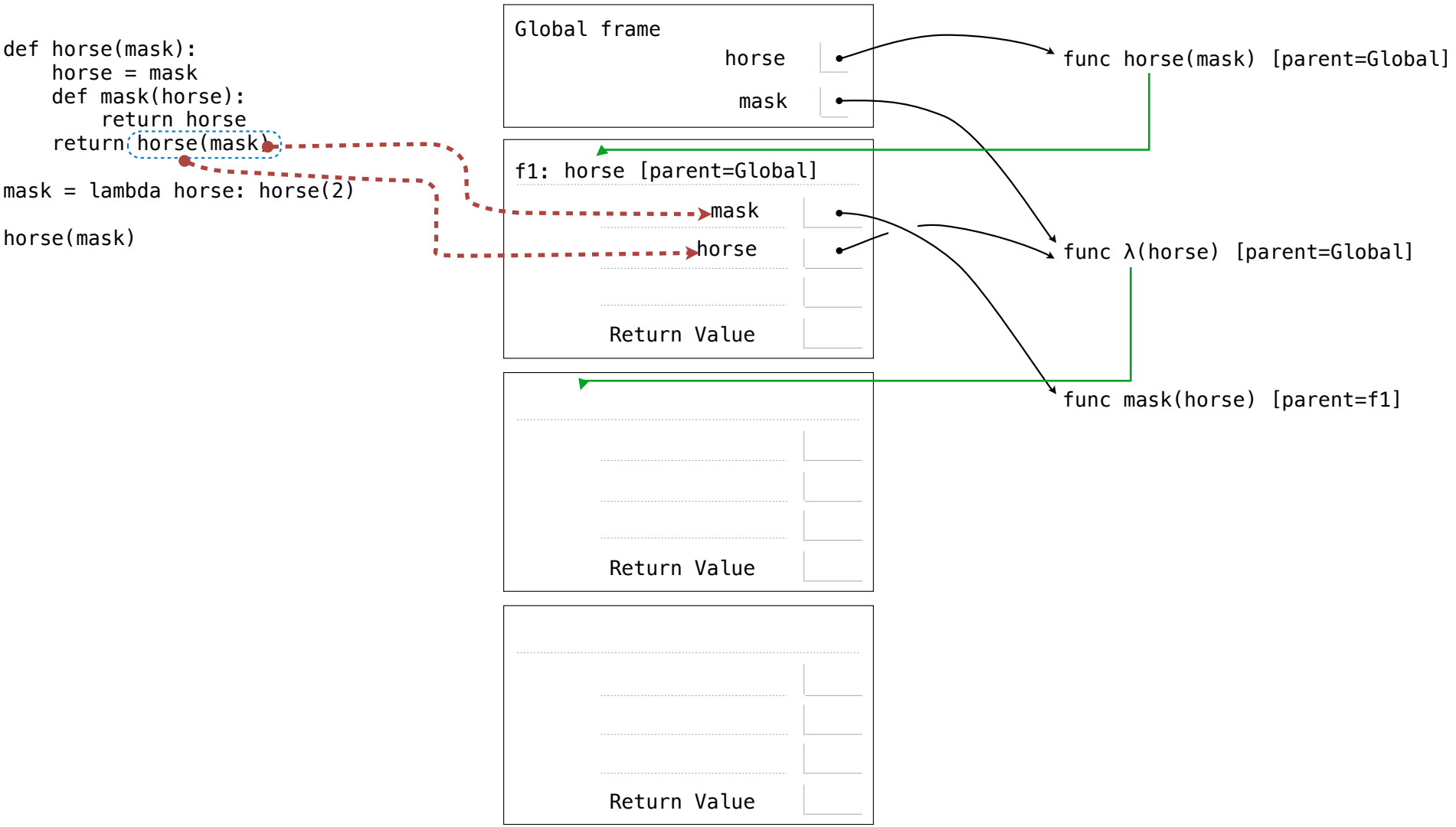
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```

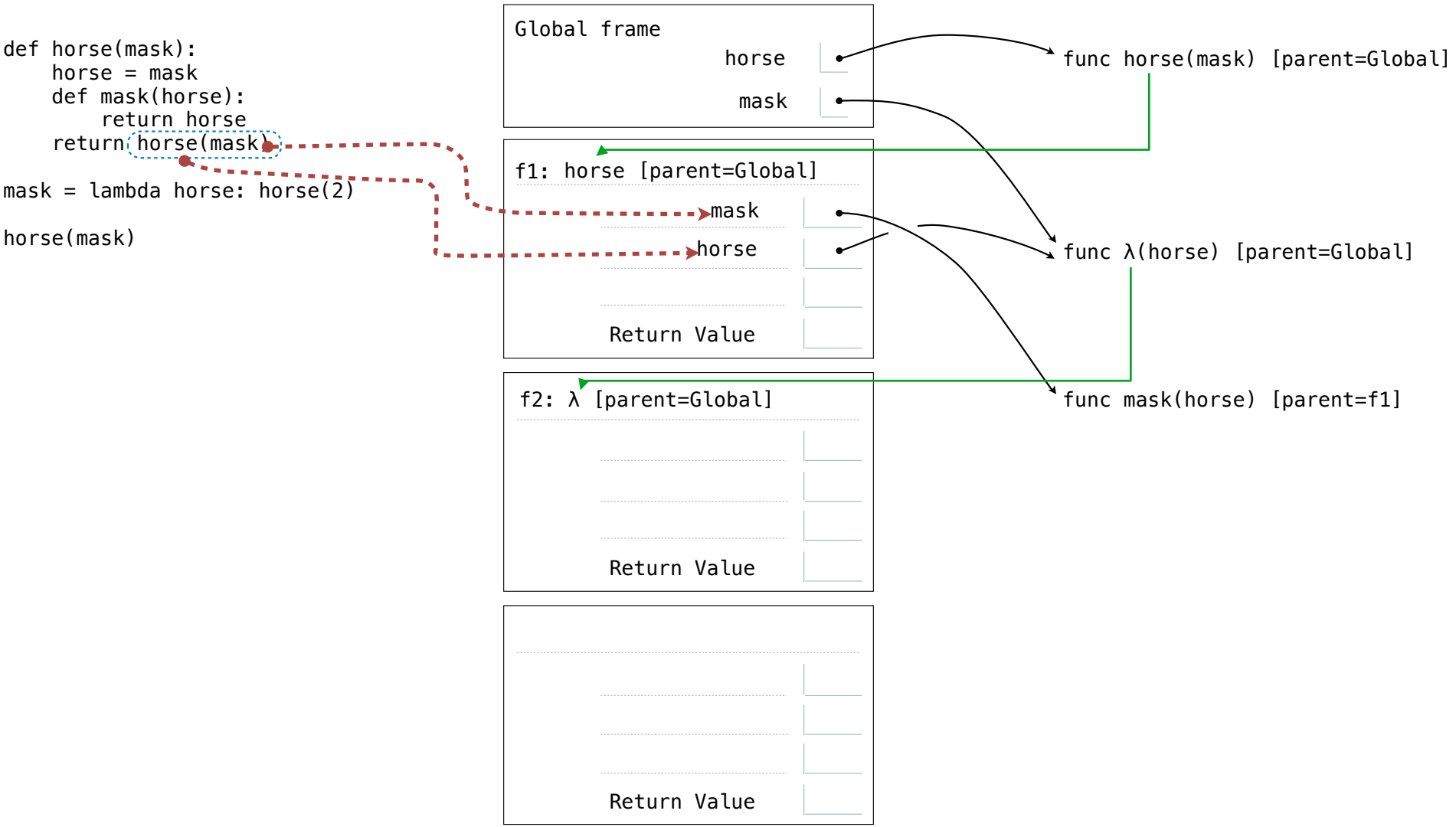


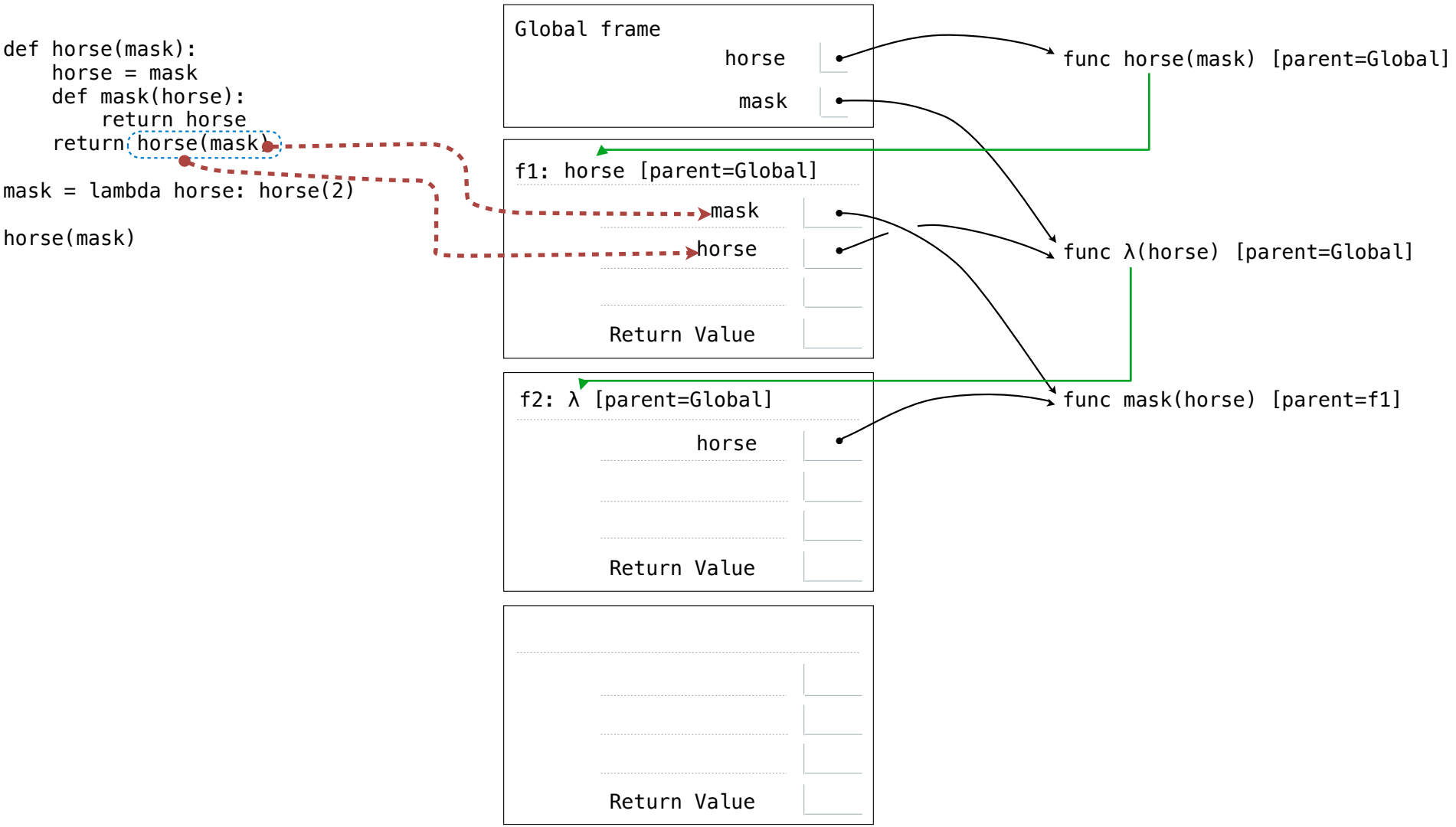
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
```

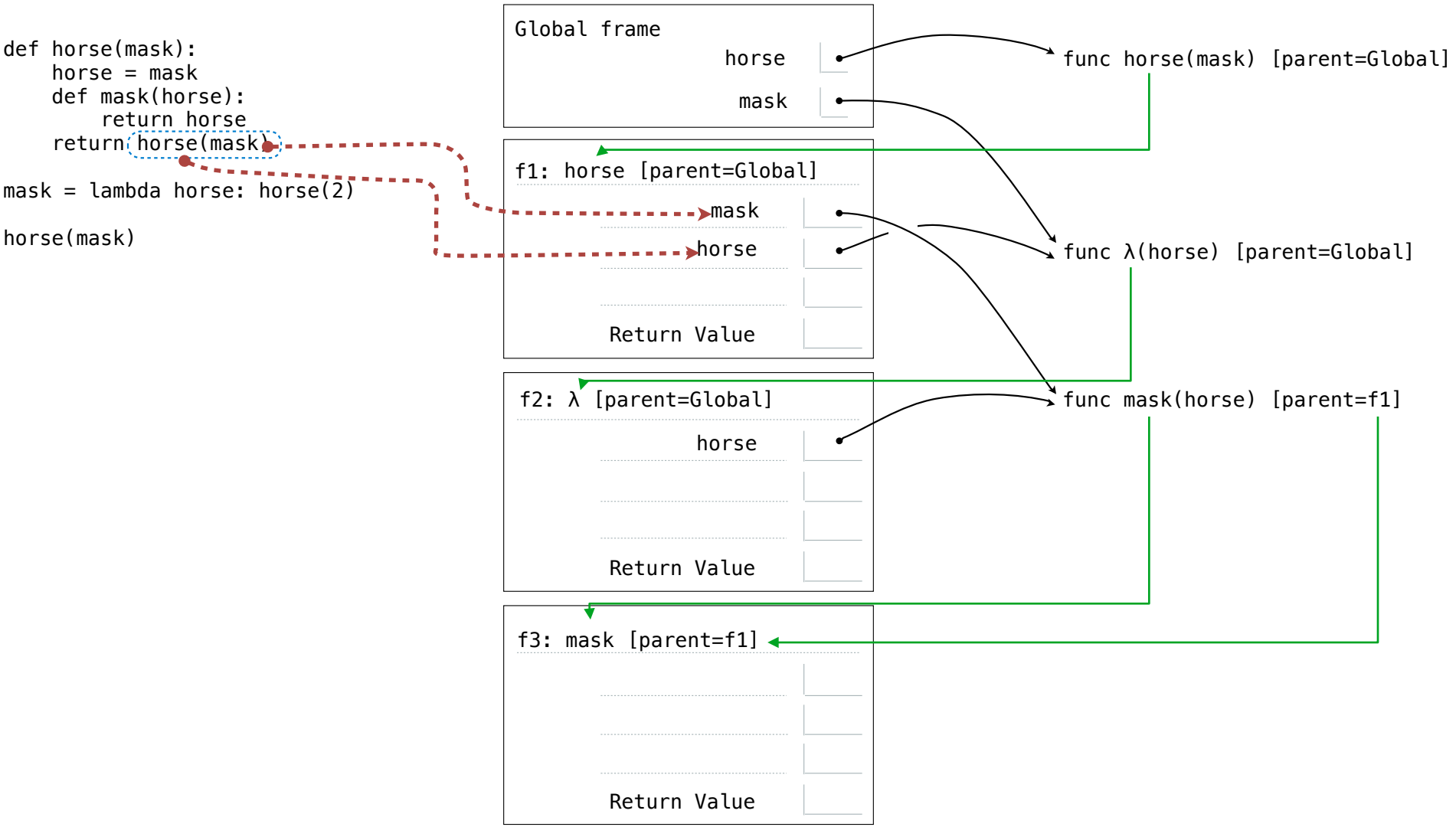


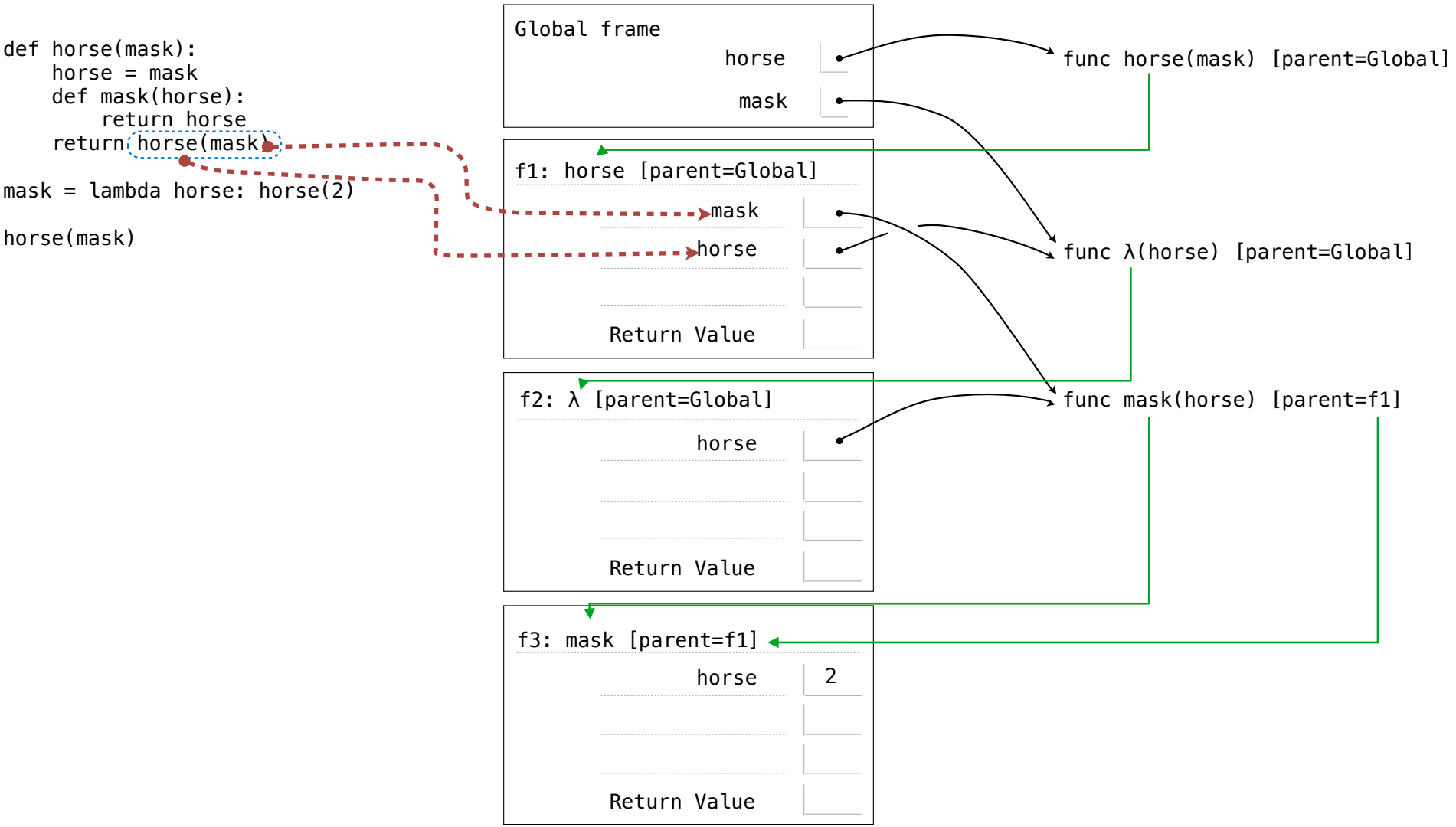

def horse(mask):
horse = mask
def mask(horse):
return horse
return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
Global frame
horse
mask
func horse(mask) [parent=Global]
f1: horse [parent=Global]
mask
horse
Return Value
func λ(horse) [parent=Global]
func mask(horse) [parent=f1]
Return Value
Return Value

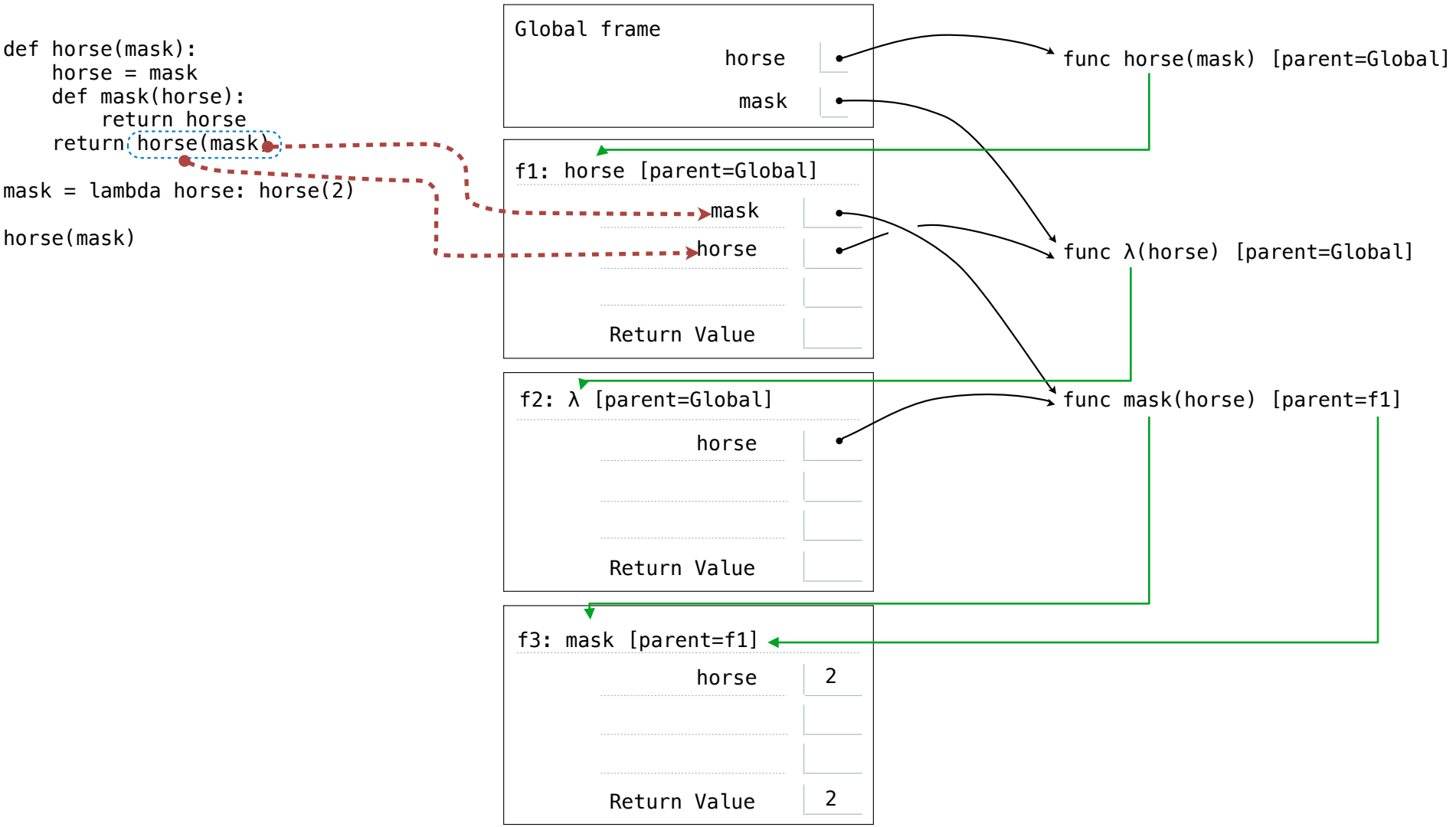

def horse(mask):
horse = mask
def mask(horse):
return horse
return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
Global frame
horse
mask
func horse(mask) [parent=Global]
f1: horse [parent=Global]
mask
horse
Return Value
func λ(horse) [parent=Global]
func mask(horse) [parent=f1]
Return Value
Return Value

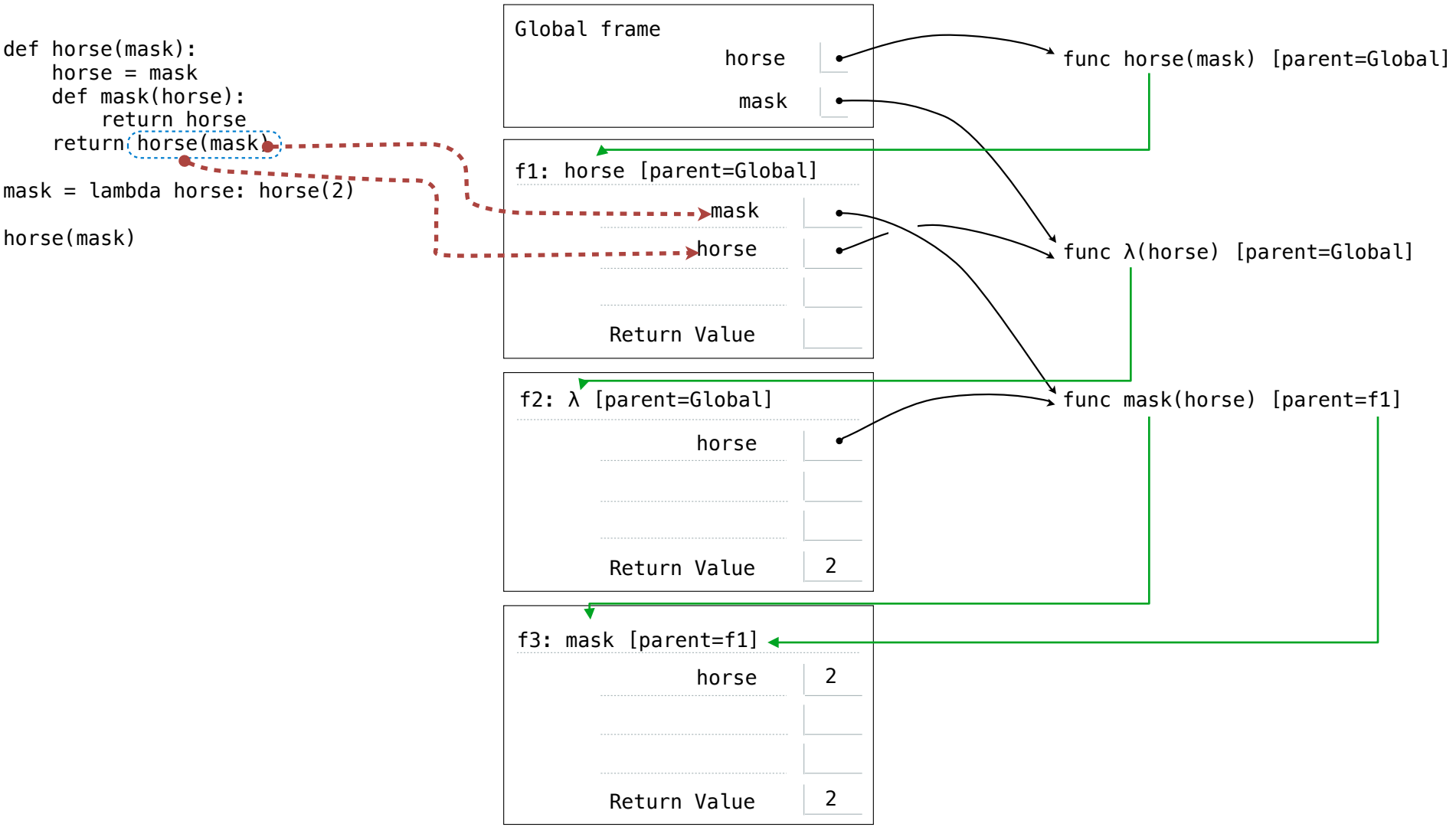

def horse(mask):
horse = mask
def mask(horse):
return horse
return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
Global frame
horse
mask
func horse(mask) [parent=Global]
f1: horse [parent=Global]
mask
horse
Return Value
func λ(horse) [parent=Global]
f2: λ [parent=Global]
Return Value
func mask(horse) [parent=f1]
Return Value

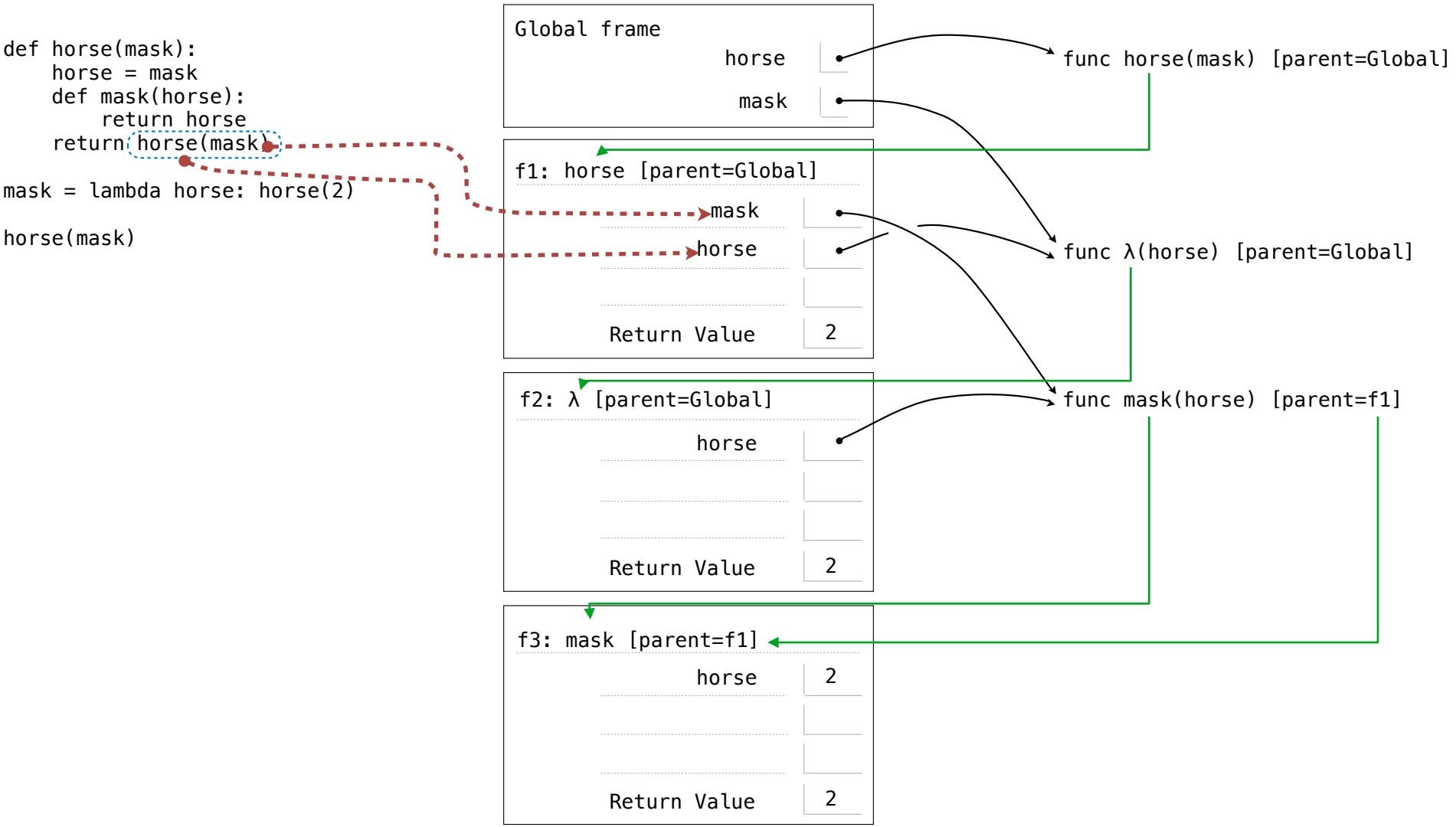

def horse(mask):
horse = mask
def mask(horse):
return horse
return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
Global frame
horse
mask
func horse(mask) [parent=Global]
f1: horse [parent=Global]
mask
horse
Return Value
func λ(horse) [parent=Global]
f2: λ [parent=Global]
horse
Return Value
func mask(horse) [parent=f1]
Return Value


def horse(mask):
horse = mask
def mask(horse):
return horse
return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
Global frame
horse
mask
func horse(mask) [parent=Global]
f1: horse [parent=Global]
mask
horse
Return Value
func λ(horse) [parent=Global]
f2: λ [parent=Global]
horse
Return Value
func mask(horse) [parent=f1]
f3: mask [parent=f1]
Return Value


def horse(mask):
horse = mask
def mask(horse):
return horse
return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
Global frame
horse
mask
func horse(mask) [parent=Global]
f1: horse [parent=Global]
mask
horse
Return Value
func λ(horse) [parent=Global]
f2: λ [parent=Global]
horse
Return Value
func mask(horse) [parent=f1]
f3: mask [parent=f1]
horse
2
Return Value


def horse(mask):
horse = mask
def mask(horse):
return horse
return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
Global frame
horse
mask
func horse(mask) [parent=Global]
f1: horse [parent=Global]
mask
horse
Return Value
func λ(horse) [parent=Global]
f2: λ [parent=Global]
horse
Return Value
func mask(horse) [parent=f1]
f3: mask [parent=f1]
horse
2
Return Value
2


def horse(mask):
horse = mask
def mask(horse):
return horse
return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
Global frame
horse
mask
func horse(mask) [parent=Global]
f1: horse [parent=Global]
mask
horse
Return Value
func λ(horse) [parent=Global]
f2: λ [parent=Global]
horse
Return Value
2
func mask(horse) [parent=f1]
f3: mask [parent=f1]
horse
2
Return Value
2


def horse(mask):
horse = mask
def mask(horse):
return horse
return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
Global frame
horse
mask
func horse(mask) [parent=Global]
f1: horse [parent=Global]
mask
horse
Return Value
2
func λ(horse) [parent=Global]
f2: λ [parent=Global]
horse
Return Value
2
func mask(horse) [parent=f1]
f3: mask [parent=f1]
horse
2
Return Value
2