

Лекция 5

в которой мы заглянем в кишочки интерпретатора

Окружения для функций высшего порядка

Окружения позволяют использовать функции высшего порядка

Окружения позволяют использовать функции высшего порядка

Функции — это объекты первого класса. В Python функции являются значениями, они могут быть переданы как параметр, возвращены из функции, присвоены переменной.

Функция высшего порядка — функция, принимающая в качестве аргумента или возвращающая другую функцию.

Окружения позволяют использовать функции высшего порядка

Функции — это объекты первого класса. В Python функции являются значениями, они могут быть переданы как параметр, возвращены из функции, присвоены переменной.

Функция высшего порядка — функция, принимающая в качестве аргумента или возвращающая другую функцию.

Диаграммы окружения описывают работу функций высшего порядка!

Окружения позволяют использовать функции высшего порядка

Функции — это объекты первого класса. В Python функции являются значениями, они могут быть переданы как параметр, возвращены из функции, присвоены переменной.

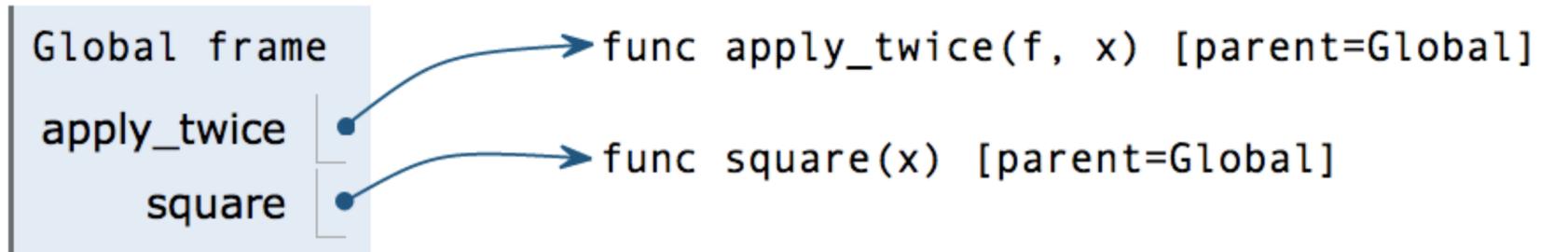
Функция высшего порядка — функция, принимающая в качестве аргумента или возвращающая другую функцию.

Диаграммы окружения описывают работу функций высшего порядка!

(Пример)

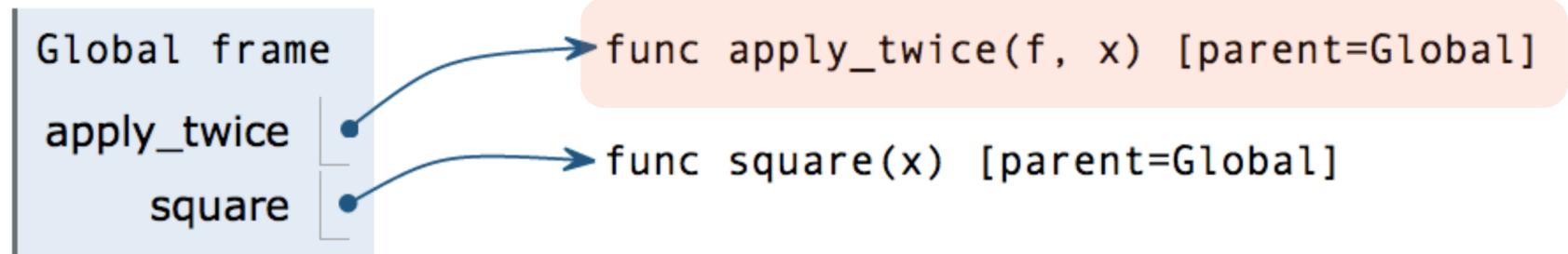
Значения формальных параметров могут быть функциями

```
1 def apply_twice(f, x):  
2     return f(f(x))  
3  
→ 4 def square(x):  
5     return x * x  
6  
→ 7 result = apply_twice(square, 2)
```



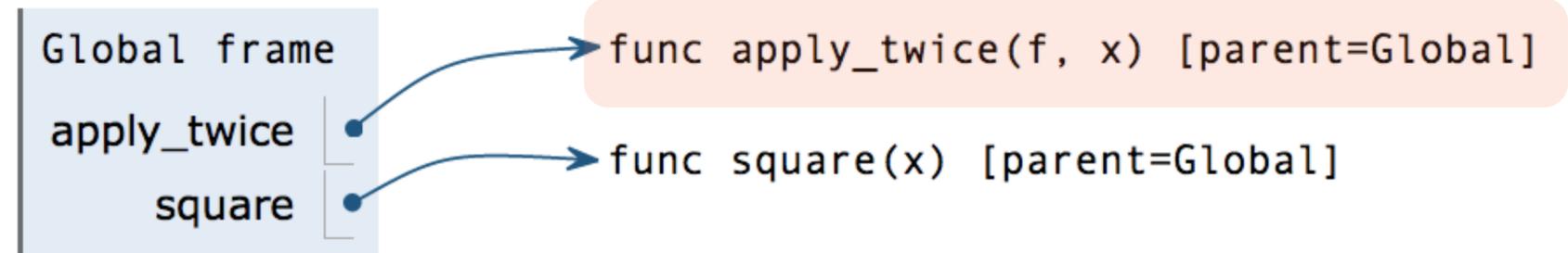
Значения формальных параметров могут быть функциями

```
1 def apply_twice(f, x):  
2     return f(f(x))  
3  
→ 4 def square(x):  
5     return x * x  
6  
→ 7 result = apply_twice(square, 2)
```



Значения формальных параметров могут быть функциями

```
1 def apply_twice(f, x):  
2     return f(f(x))  
3  
→ 4 def square(x):  
5     return x * x  
6  
→ 7 result = apply_twice(square, 2)
```

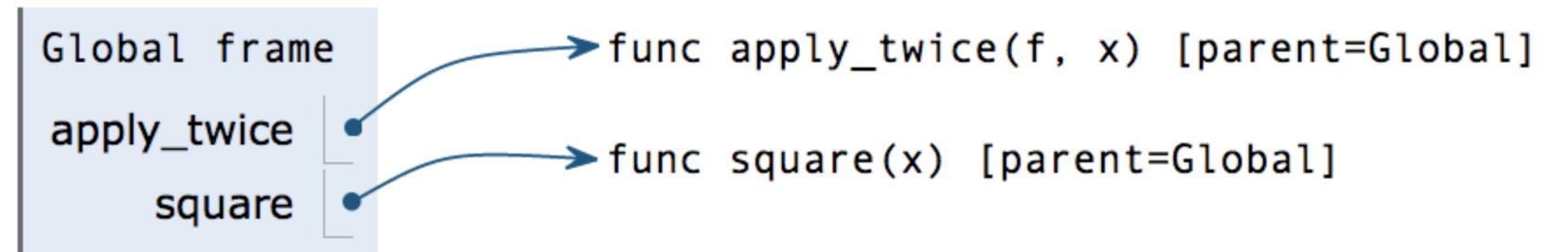


Вызов пользовательской функции:

- создаётся новый фрейм;
- формальные параметры (**f** и **x**) связываются с аргументами;
- выполняется тело **return f(f(x))**.

Значения формальных параметров могут быть функциями

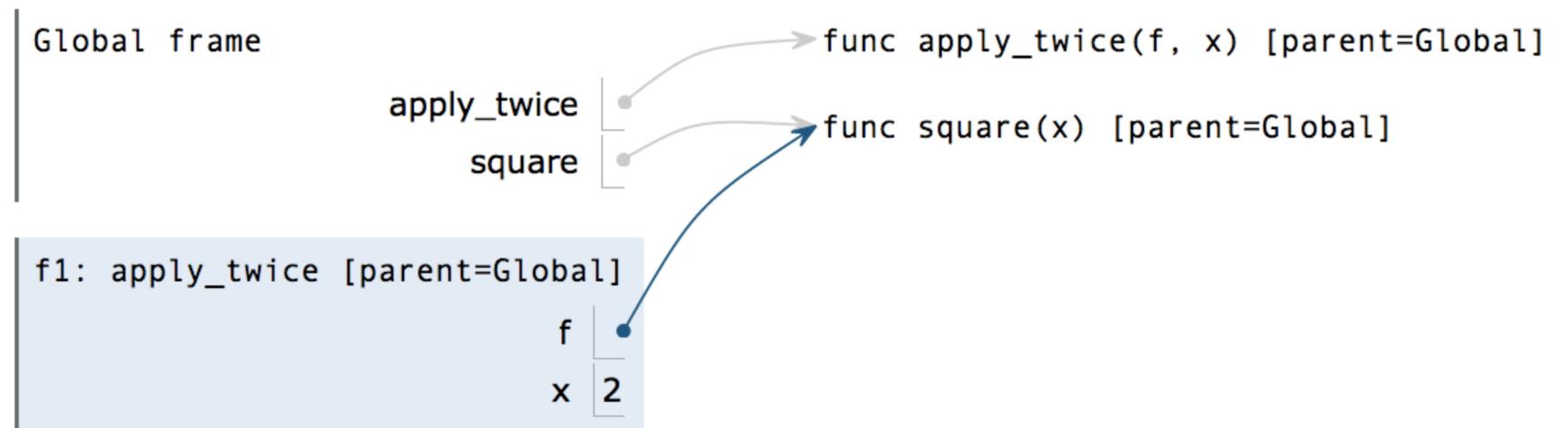
```
1 def apply_twice(f, x):  
2     return f(f(x))  
3  
→ 4 def square(x):  
5     return x * x  
6  
→ 7 result = apply_twice(square, 2)
```



Вызов пользовательской функции:

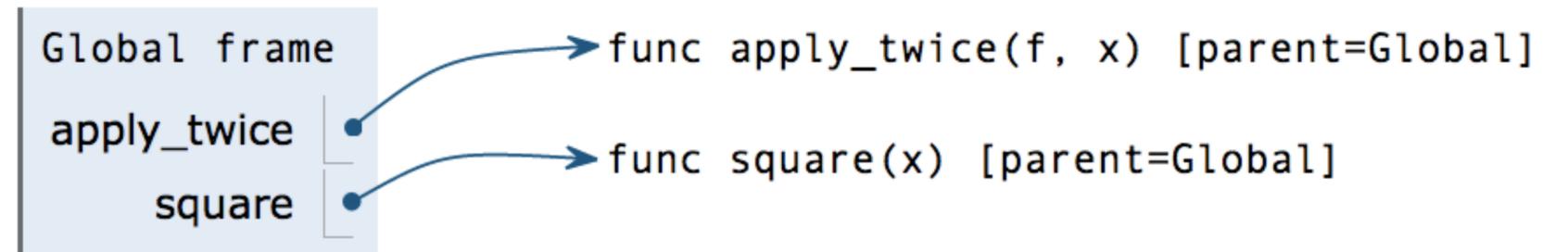
- создаётся новый фрейм;
- формальные параметры (**f** и **x**) связываются с аргументами;
- выполняется тело **return f(f(x))**.

```
→ 1 def apply_twice(f, x):  
→ 2     return f(f(x))  
3  
4 def square(x):  
5     return x * x  
6  
7 result = apply_twice(square, 2)
```



Значения формальных параметров могут быть функциями

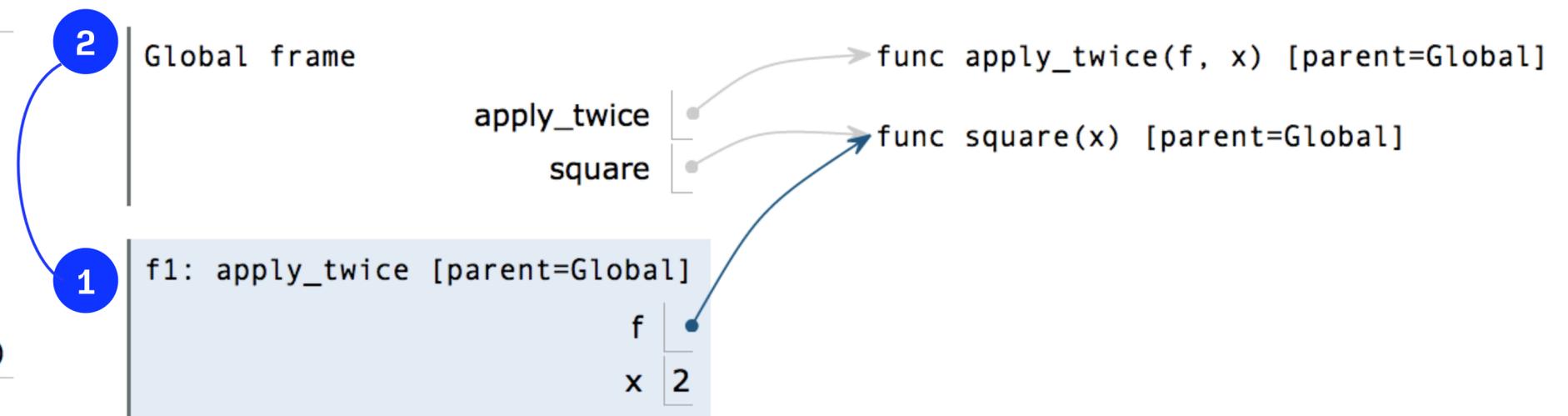
```
1 def apply_twice(f, x):  
2     return f(f(x))  
3  
→ 4 def square(x):  
5     return x * x  
6  
→ 7 result = apply_twice(square, 2)
```



Вызов пользовательской функции:

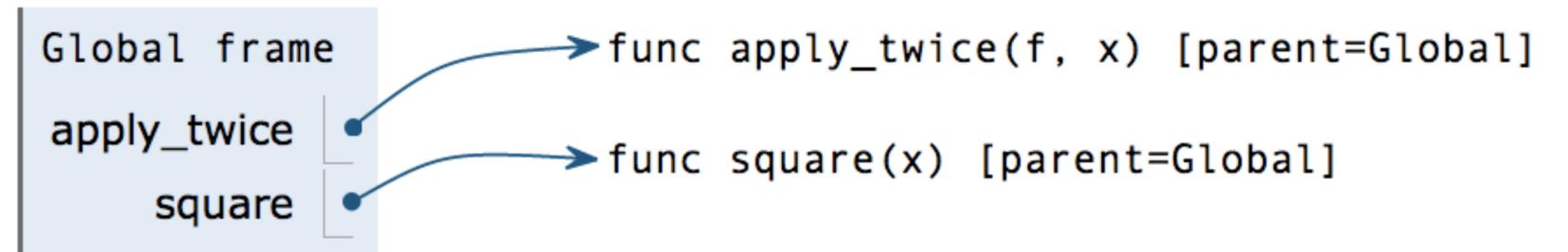
- создаётся новый фрейм;
- формальные параметры (**f** и **x**) связываются с аргументами;
- выполняется тело **return f(f(x))**.

```
→ 1 def apply_twice(f, x):  
→ 2     return f(f(x))  
3  
4 def square(x):  
5     return x * x  
6  
7 result = apply_twice(square, 2)
```



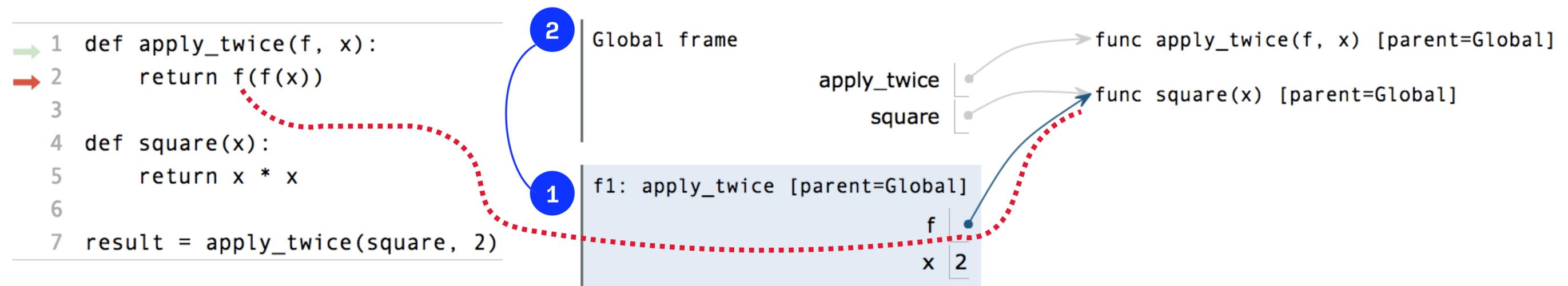
Значения формальных параметров могут быть функциями

```
1 def apply_twice(f, x):  
2     return f(f(x))  
3  
→ 4 def square(x):  
5     return x * x  
6  
→ 7 result = apply_twice(square, 2)
```



Вызов пользовательской функции:

- создаётся новый фрейм;
- формальные параметры (**f** и **x**) связываются с аргументами;
- выполняется тело **return f(f(x))**.

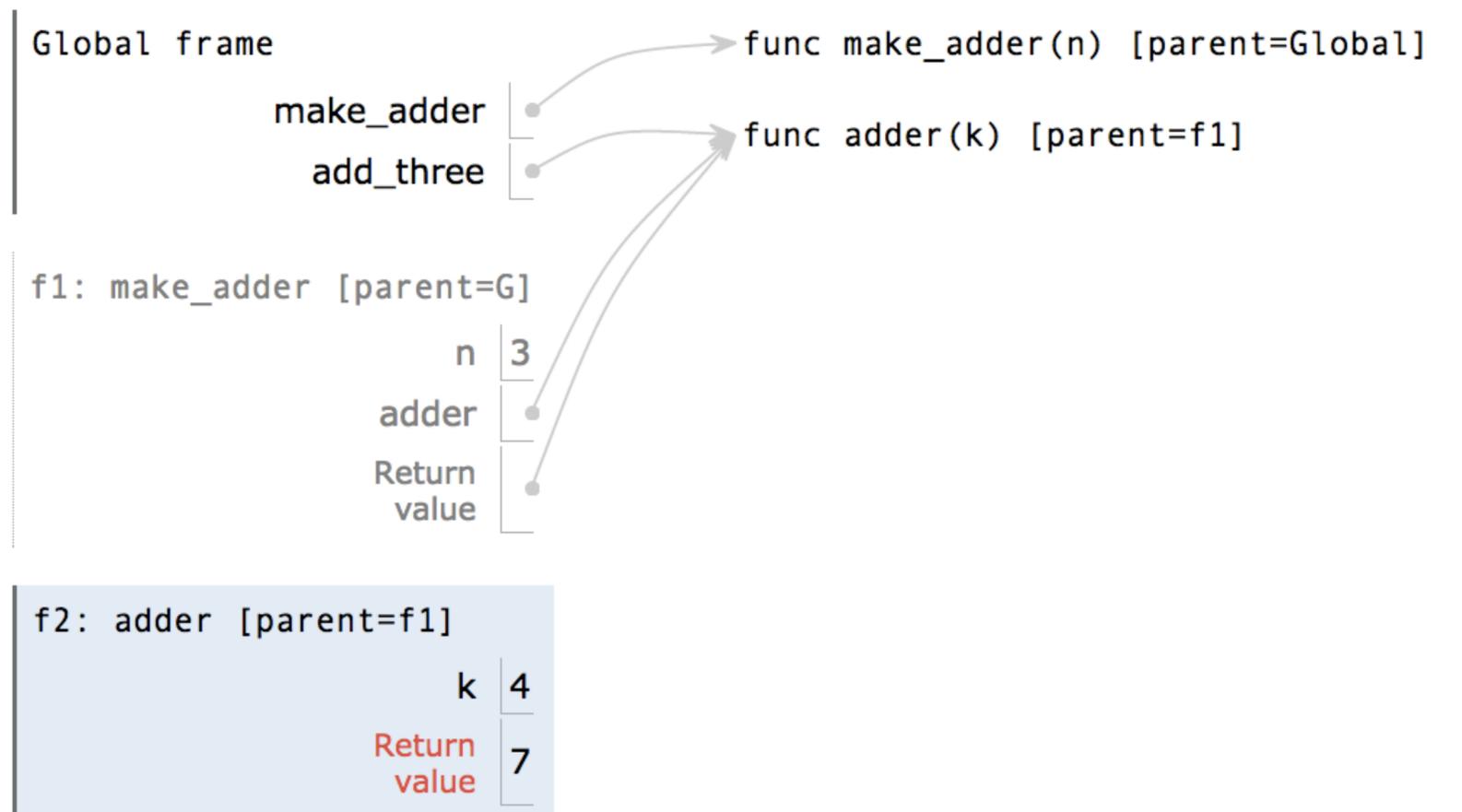


Окружения для вложенных функций

(Пример)

Диаграммы окружения для вложенных инструкций def

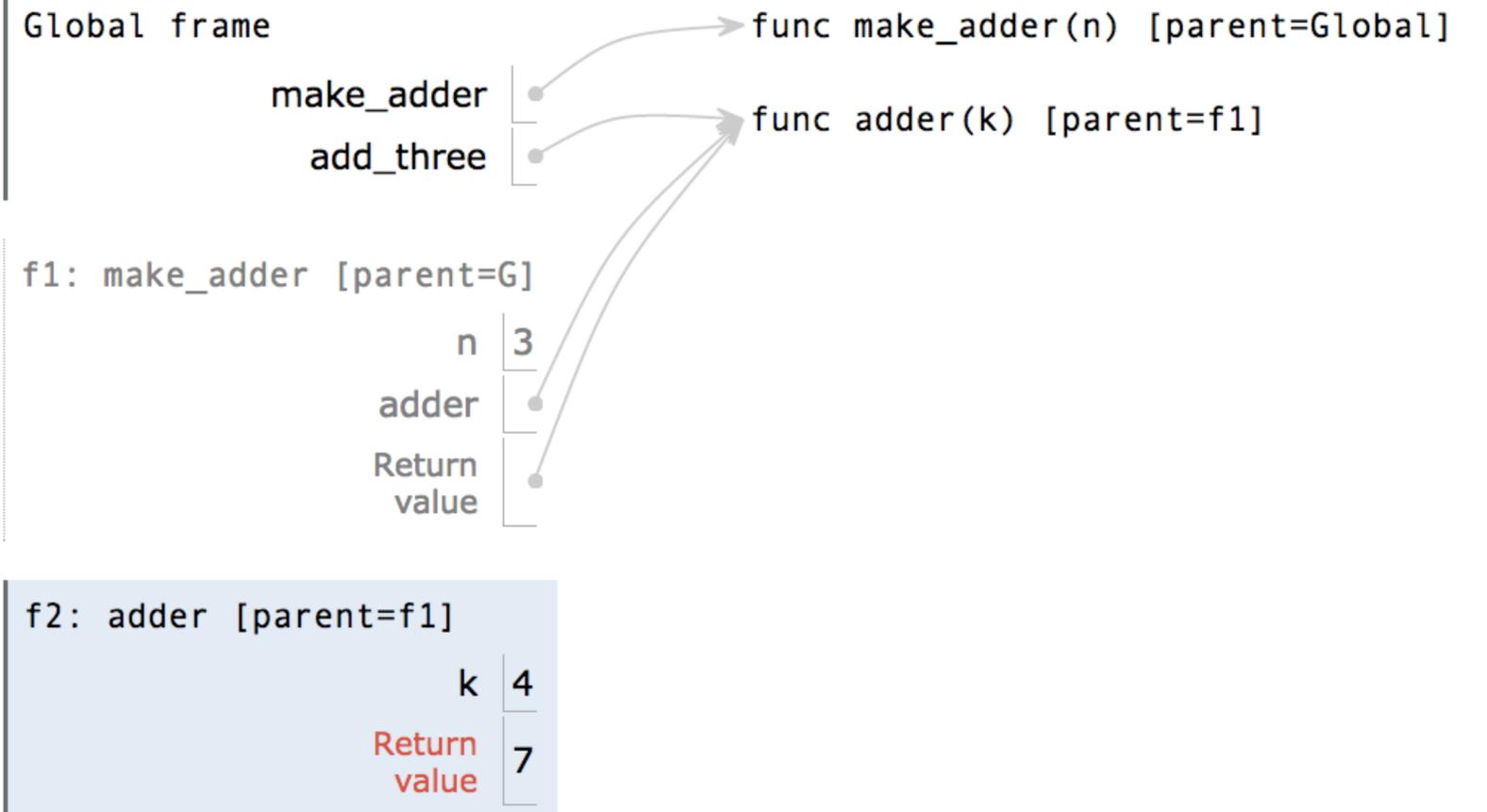
```
1 def make_adder(n):  
2     def adder(k):  
3         return k + n  
4     return adder  
5  
6 add_three = make_adder(3)  
7 add_three(4)
```



Диаграммы окружения для вложенных инструкций def

Вложенный def

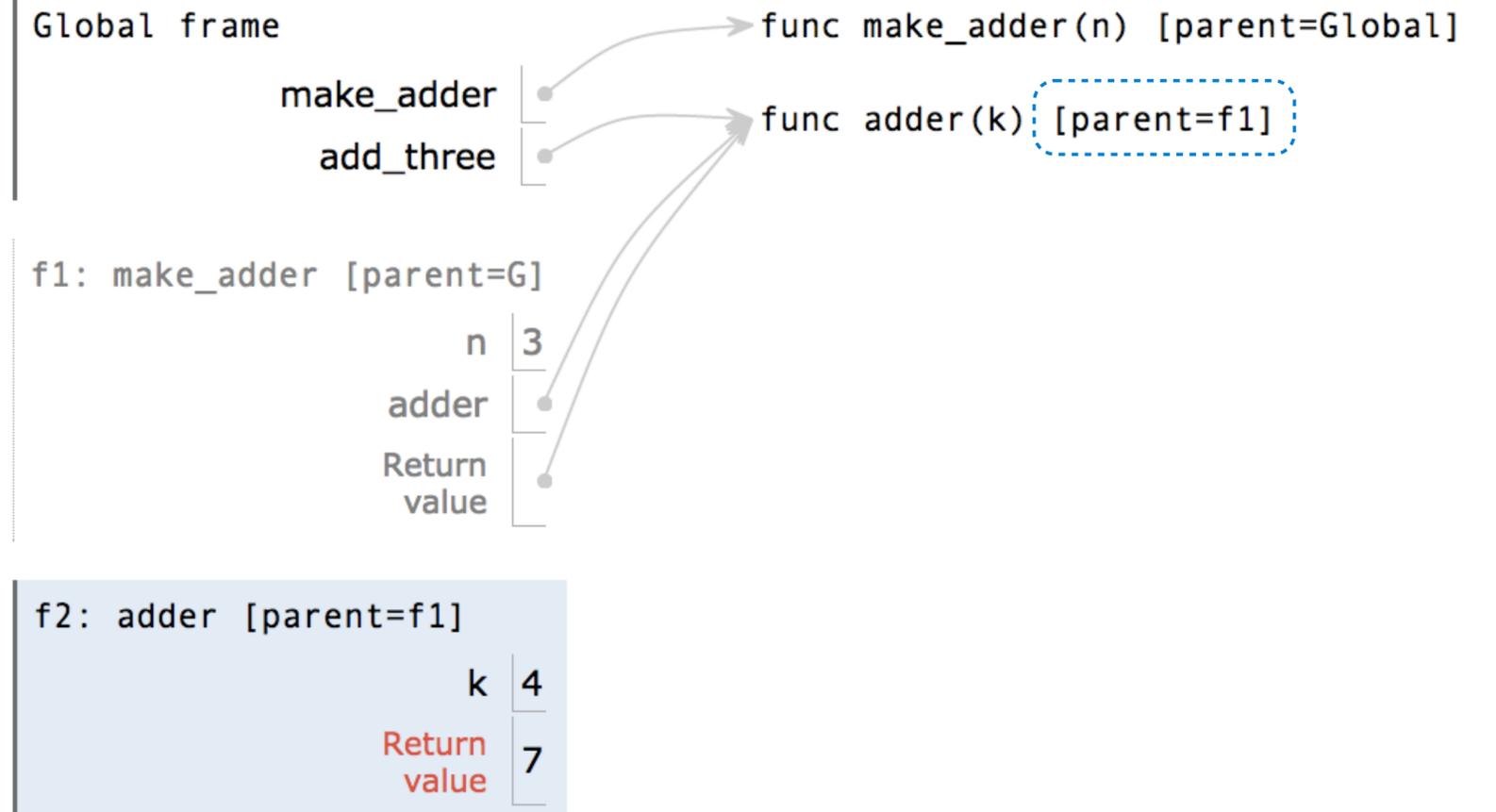
```
1 def make_adder(n):  
2     def adder(k):  
3         return k + n  
4     return adder  
5  
6 add_three = make_adder(3)  
7 add_three(4)
```



Диаграммы окружения для вложенных инструкций def

Вложенный def

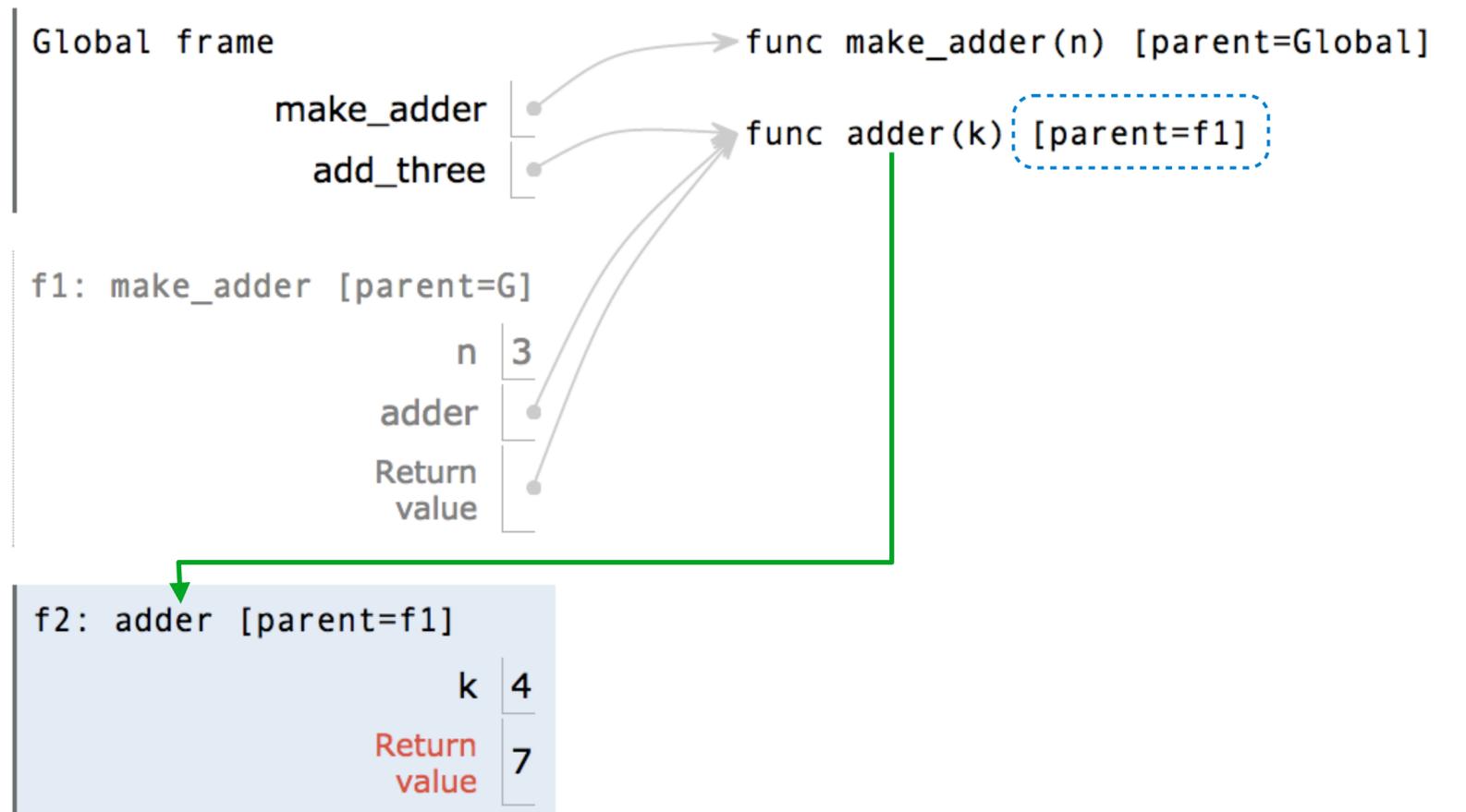
```
1 def make_adder(n):  
2     def adder(k):  
3         return k + n  
4     return adder  
5  
6 add_three = make_adder(3)  
7 add_three(4)
```



Диаграммы окружения для вложенных инструкций def

Вложенный def

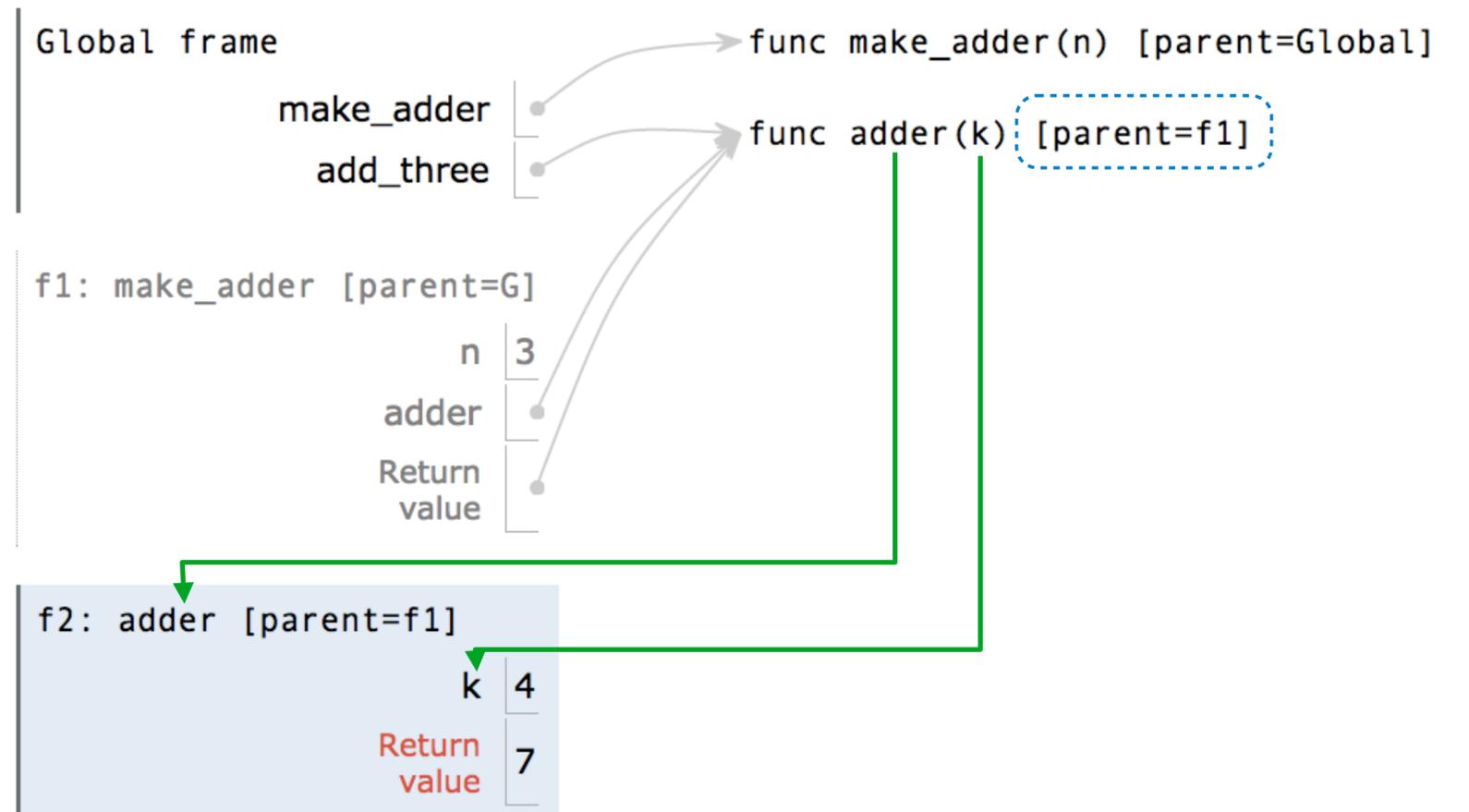
```
1 def make_adder(n):  
2     def adder(k):  
3         return k + n  
4     return adder  
5  
6 add_three = make_adder(3)  
7 add_three(4)
```



Диаграммы окружения для вложенных инструкций def

Вложенный def

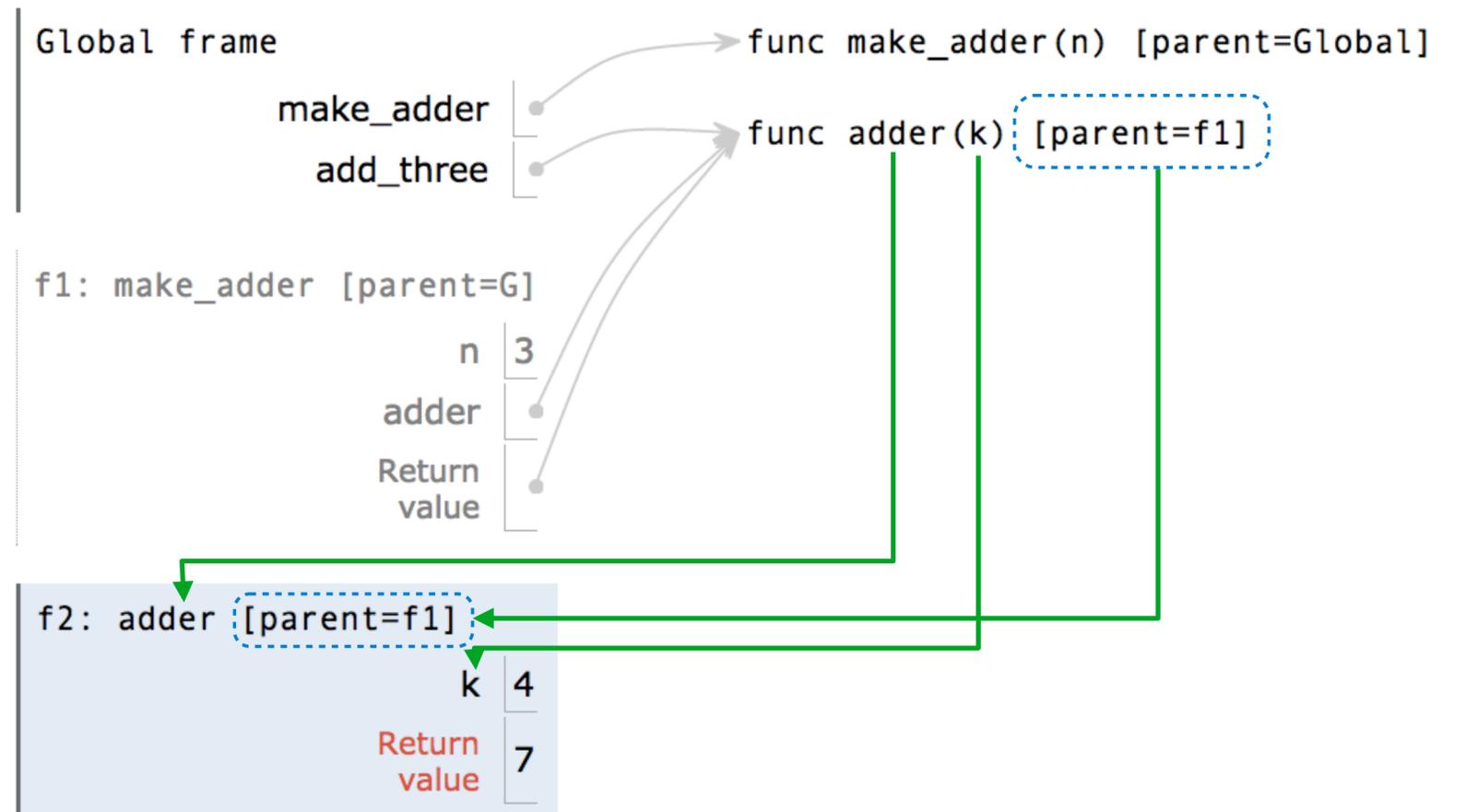
```
1 def make_adder(n):  
2     def adder(k):  
3         return k + n  
4     return adder  
5  
6 add_three = make_adder(3)  
7 add_three(4)
```



Диаграммы окружения для вложенных инструкций def

Вложенный def

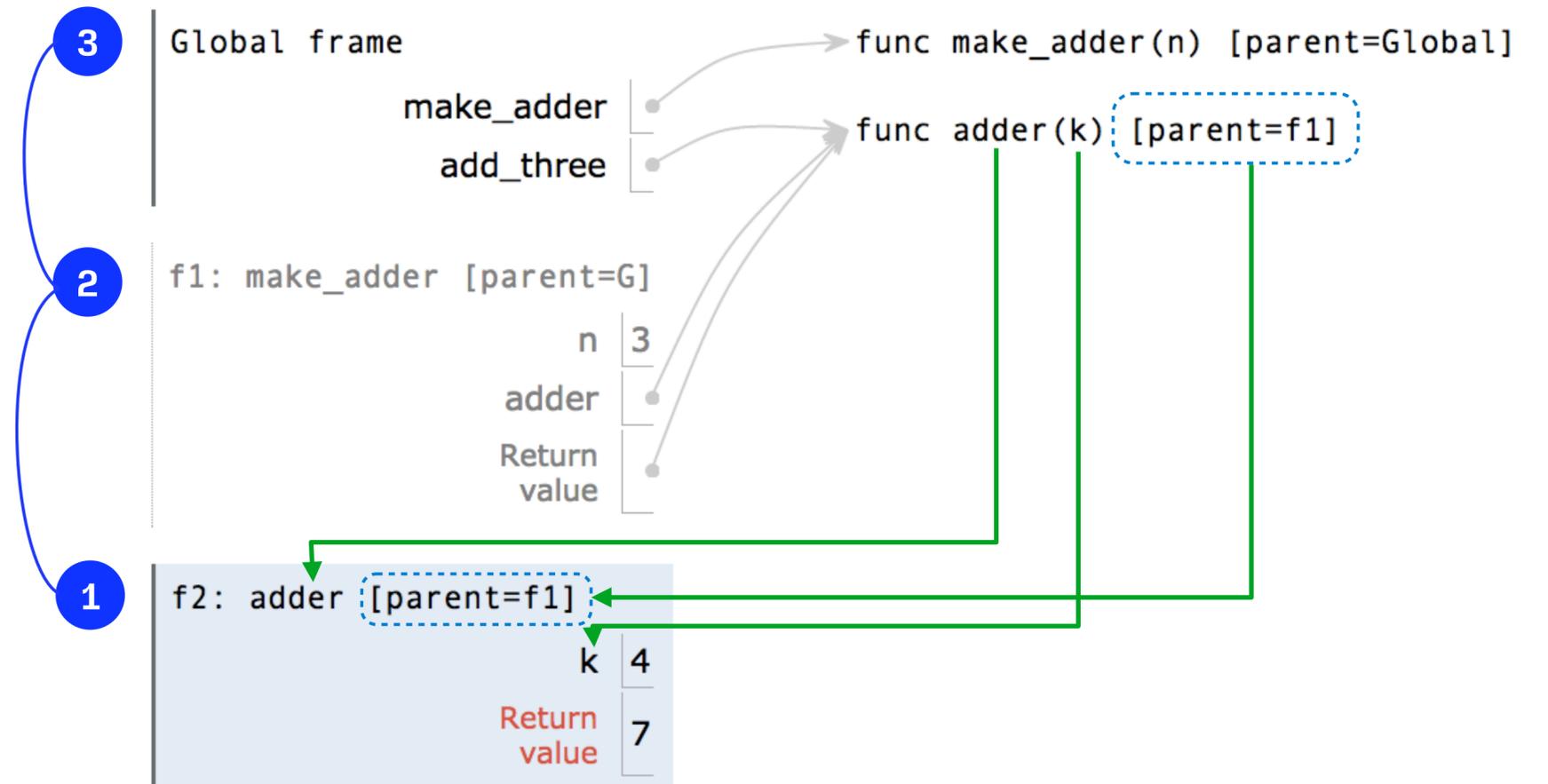
```
1 def make_adder(n):  
2     def adder(k):  
3         return k + n  
4     return adder  
5  
6 add_three = make_adder(3)  
7 add_three(4)
```



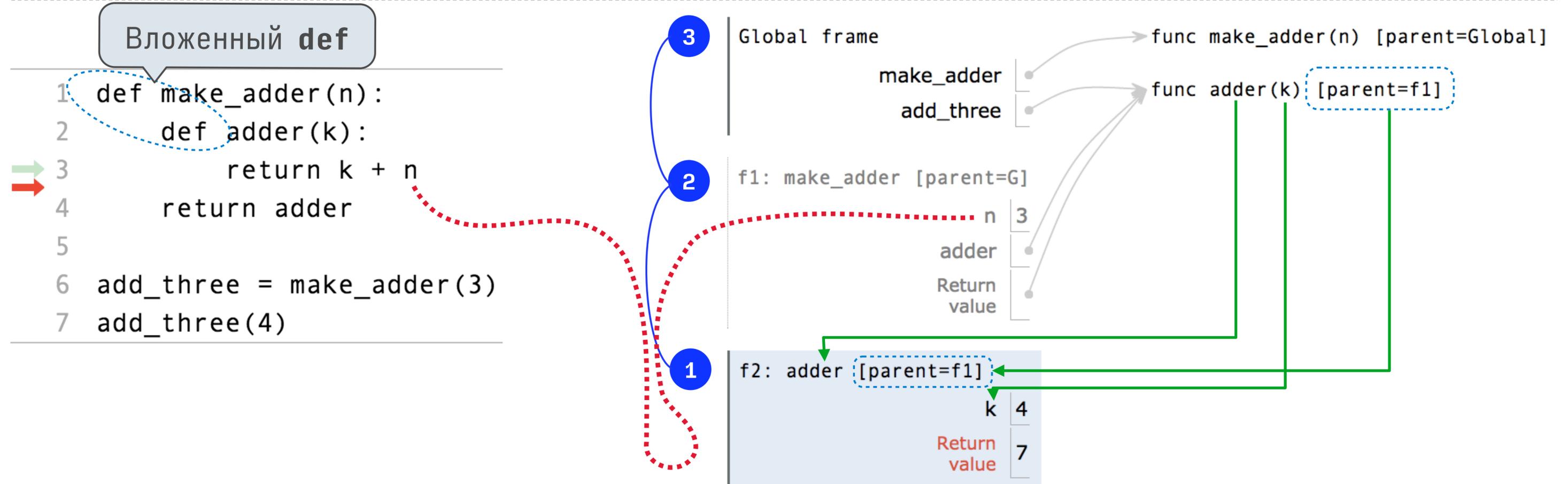
Диаграммы окружения для вложенных инструкций def

Вложенный def

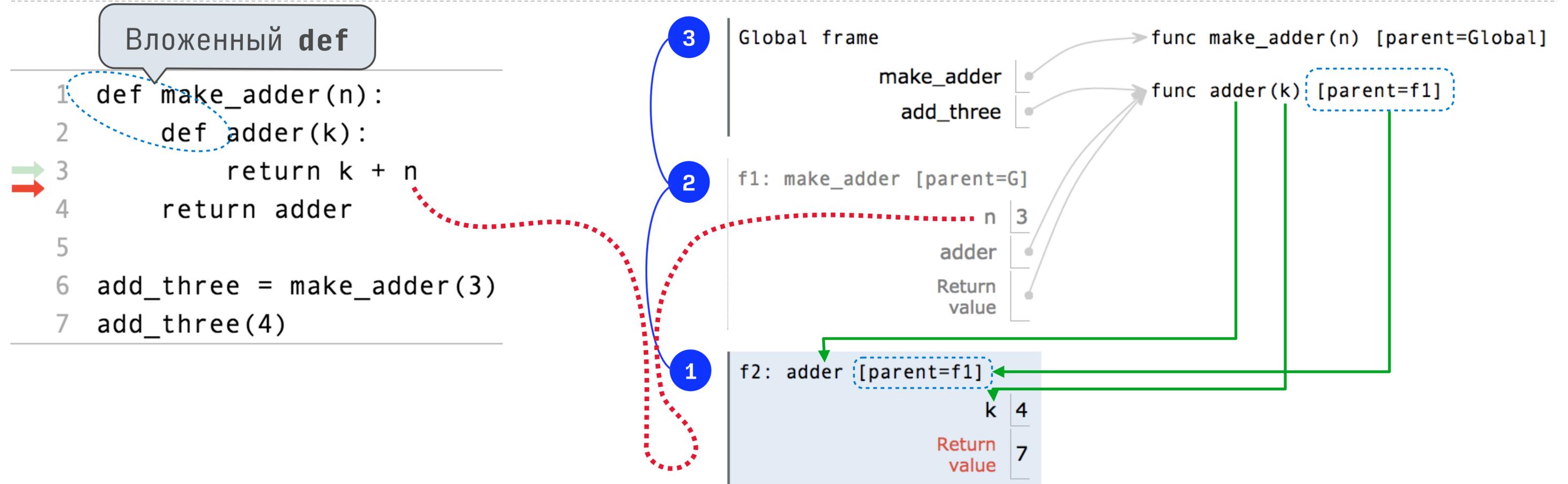
```
1 def make_adder(n):  
2     def adder(k):  
3         return k + n  
4     return adder  
5  
6 add_three = make_adder(3)  
7 add_three(4)
```



Диаграммы окружения для вложенных инструкций def

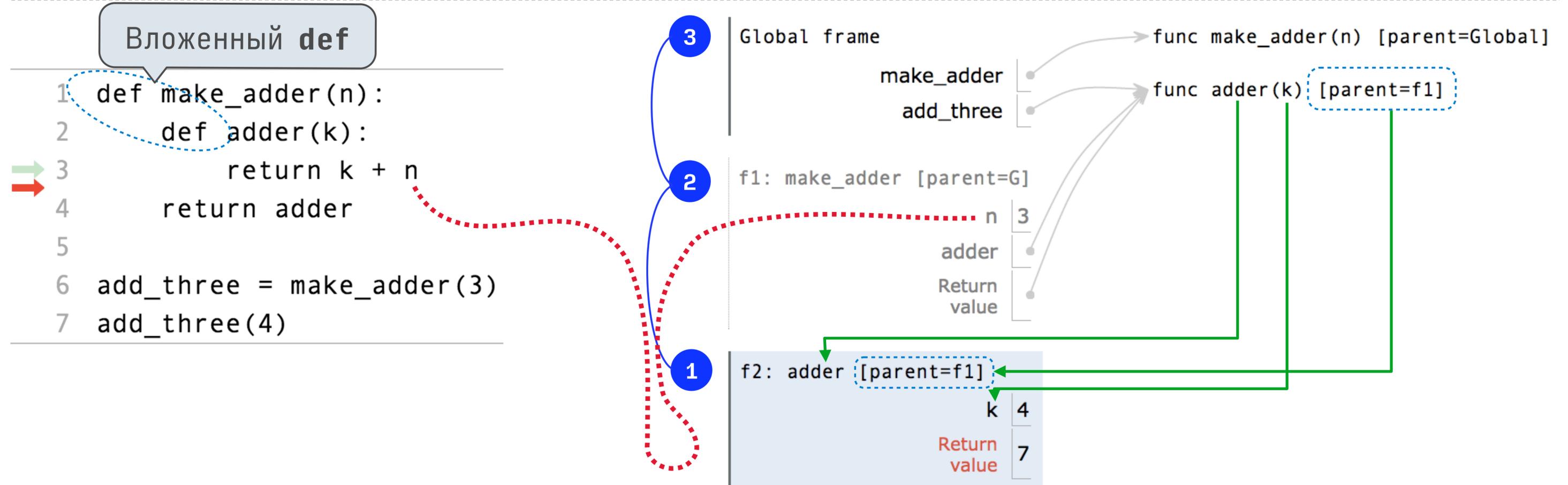


Диаграммы окружения для вложенных инструкций def



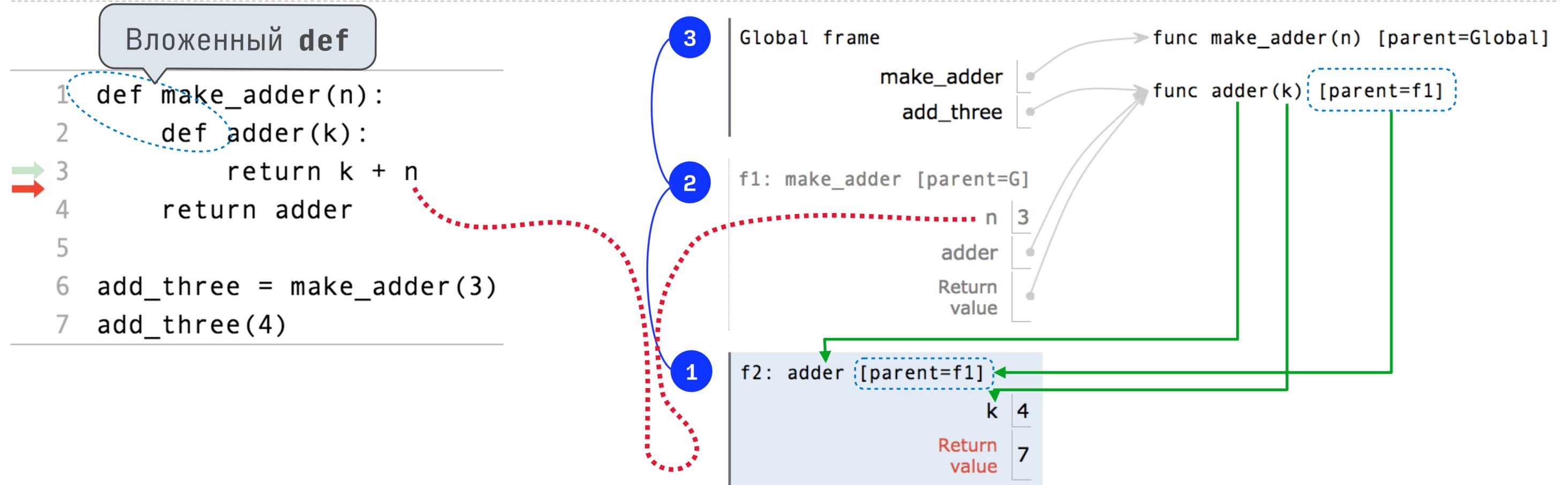
— каждая пользовательская функция имеет родительский фрейм (зачастую глобальный);

Диаграммы окружения для вложенных инструкций def



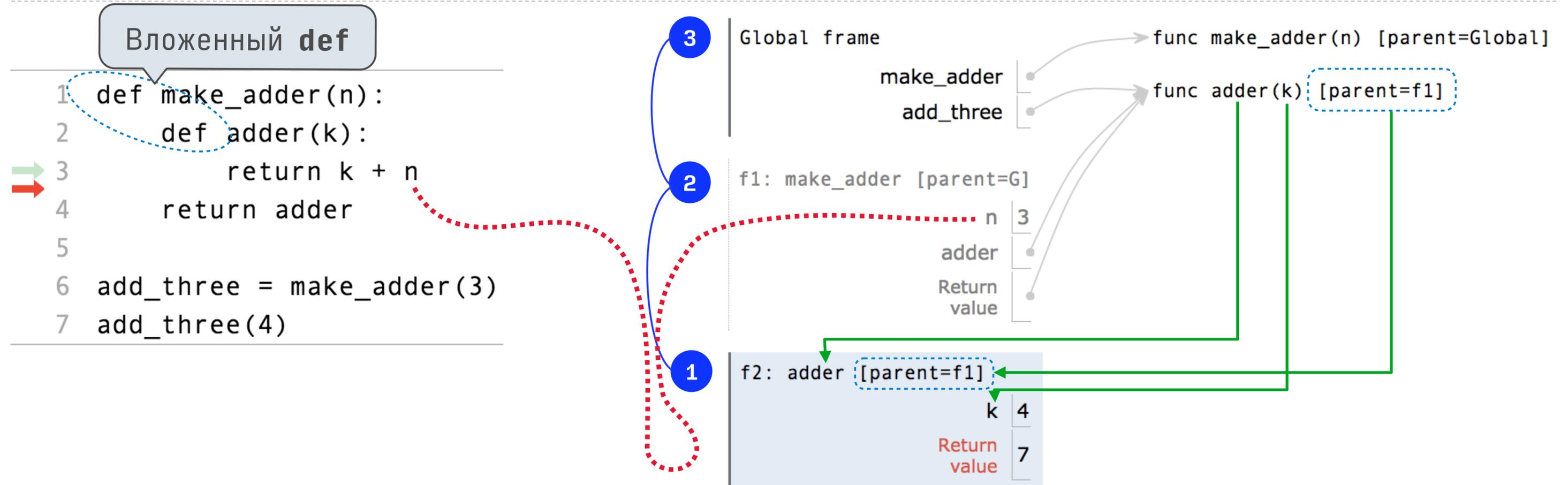
- каждая пользовательская функция имеет родительский фрейм (зачастую глобальный);
- родительский фрейм функции — фрейм, в котором была выполнена инструкция **def**;

Диаграммы окружения для вложенных инструкций def



- каждая пользовательская функция имеет родительский фрейм (зачастую глобальный);
- родительский фрейм функции — фрейм, в котором была выполнена инструкция **def**;
- каждый локальный фрейм имеет родительский фрейм (зачастую глобальный);

Диаграммы окружения для вложенных инструкций def



- каждая пользовательская функция имеет родительский фрейм (зачастую глобальный);
- родительский фрейм функции — фрейм, в котором была выполнена инструкция **def**;
- каждый локальный фрейм имеет родительский фрейм (зачастую глобальный);
- родитель фрейма — это родитель вызванной функции.

Как нарисовать диаграмму окружения

Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Создается функция (значение): `func <имя>(<формальные параметры>) [parent=<метка>]`

Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Создается функция (значение): `func <имя>(<формальные параметры>) [parent=<метка>]`

Её родителем становится текущий фрейм.

Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Создается функция (значение): `func <имя>(<формальные параметры>) [parent=<метка>]`

Её родителем становится текущий фрейм.

f1: make_adder

func adder(k) [parent=f1]



Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Создается функция (значение): `func <имя>(<формальные параметры>) [parent=<метка>]`

Её родителем становится текущий фрейм.



Происходит связывание `<имени>` с функцией в текущем фрейме.

Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Создается функция (значение): `func <имя>(<формальные параметры>) [parent=<метка>]`

Её родителем становится текущий фрейм.

```
f1: make_adder      func adder(k) [parent=f1]
```

Происходит связывание `<имени>` с функцией в текущем фрейме.

При вызове функции:

Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Создается функция (значение): `func <имя>(<формальные параметры>) [parent=<метка>]`

Её родителем становится текущий фрейм.

```
f1: make_adder          func adder(k) [parent=f1]
```

Происходит связывание `<имени>` с функцией в текущем фрейме.

При вызове функции:

1. Добавляется локальный фрейм, озаглавленный `<именем>` вызванной функции.

Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Создается функция (значение): `func <имя>(<формальные параметры>) [parent=<метка>]`

Её родителем становится текущий фрейм.

```
f1: make_adder      func adder(k) [parent=f1]
```

Происходит связывание `<имени>` с функцией в текущем фрейме.

При вызове функции:

1. Добавляется локальный фрейм, озаглавленный `<именем>` вызванной функции.
- ★ 2. Родитель функции копируется в локальный фрейм: `[parent=<метка>]`

Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Создается функция (значение): `func <имя>(<формальные параметры>) [parent=<метка>]`

Её родителем становится текущий фрейм.

```
f1: make_adder      func adder(k) [parent=f1]
```

Происходит связывание `<имени>` с функцией в текущем фрейме.

При вызове функции:

1. Добавляется локальный фрейм, озаглавленный `<именем>` вызванной функции.
- ★ 2. Родитель функции копируется в локальный фрейм: `[parent=<метка>]`
3. `<Формальные параметры>` связываются со значениями аргументами в локальном фрейме.

Как нарисовать диаграмму окружения

Выполнение инструкции `def`:

Создается функция (значение): `func <имя>(<формальные параметры>) [parent=<метка>]`

Её родителем становится текущий фрейм.



Происходит связывание `<имени>` с функцией в текущем фрейме.

При вызове функции:

1. Добавляется локальный фрейм, озаглавленный `<именем>` вызванной функции.
 - ★ 2. Родитель функции копируется в локальный фрейм: `[parent=<метка>]`
 3. `<Формальные параметры>` связываются со значениями аргументами в локальном фрейме.
 4. В окружении, начинающемся с локального фрейма, выполняется тело функции.
-

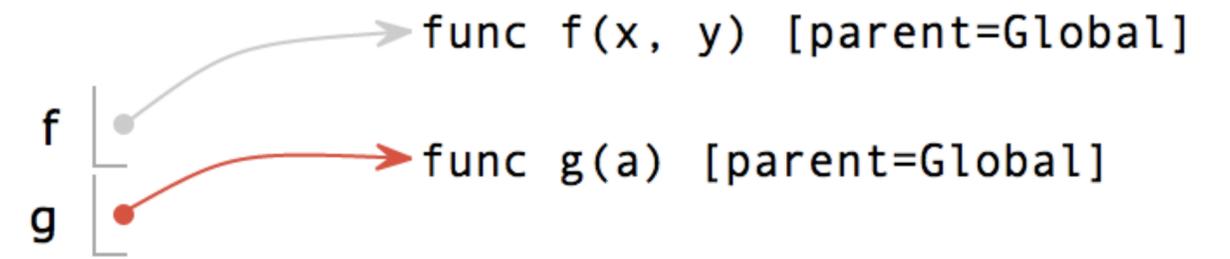
Локальные имена

(Пример)

Локальные имена не видны другим (невложенным) функциям

```
1 def f(x, y):  
2     return g(x)  
3  
4 def g(a):  
→ 5     return a + y  
6  
7 result = f(1, 2)
```

Global frame



f1: f [parent=Global]

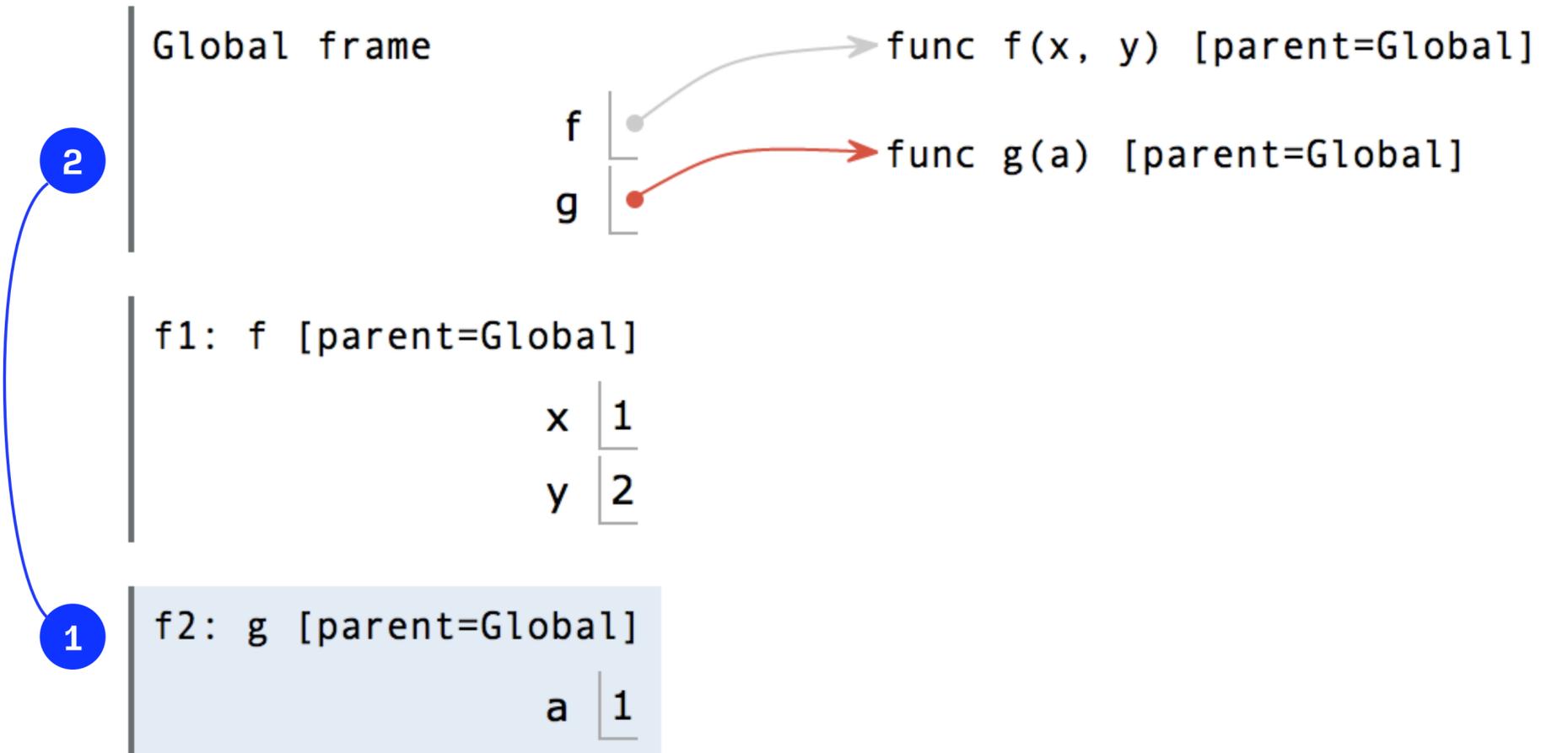
x | 1
y | 2

f2: g [parent=Global]

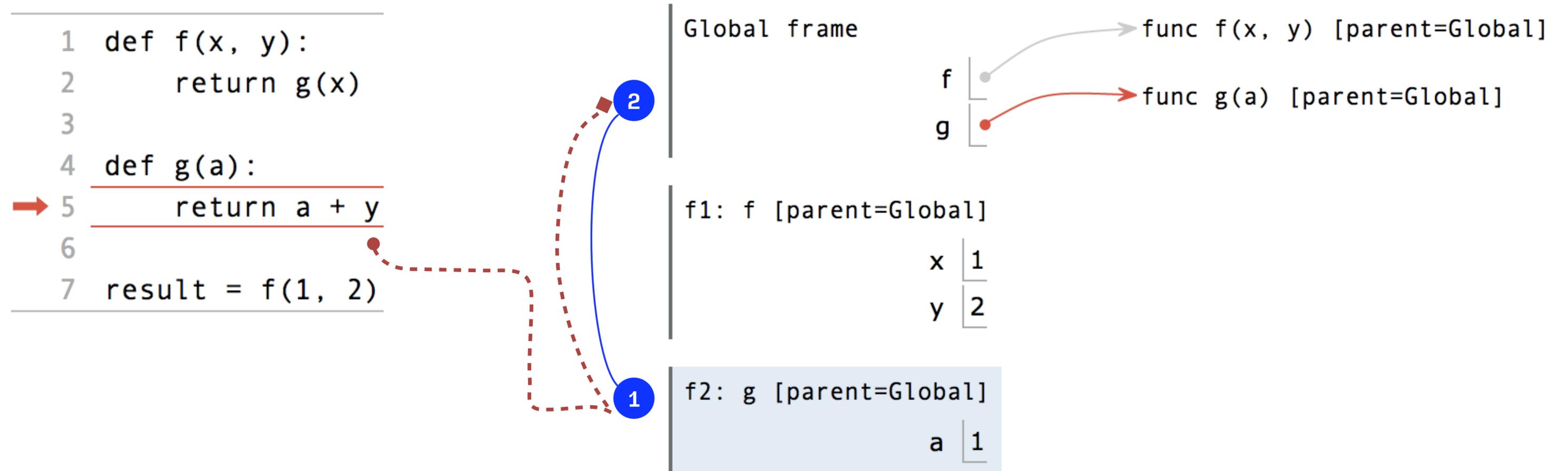
a | 1

Локальные имена не видны другим (невложенным) функциям

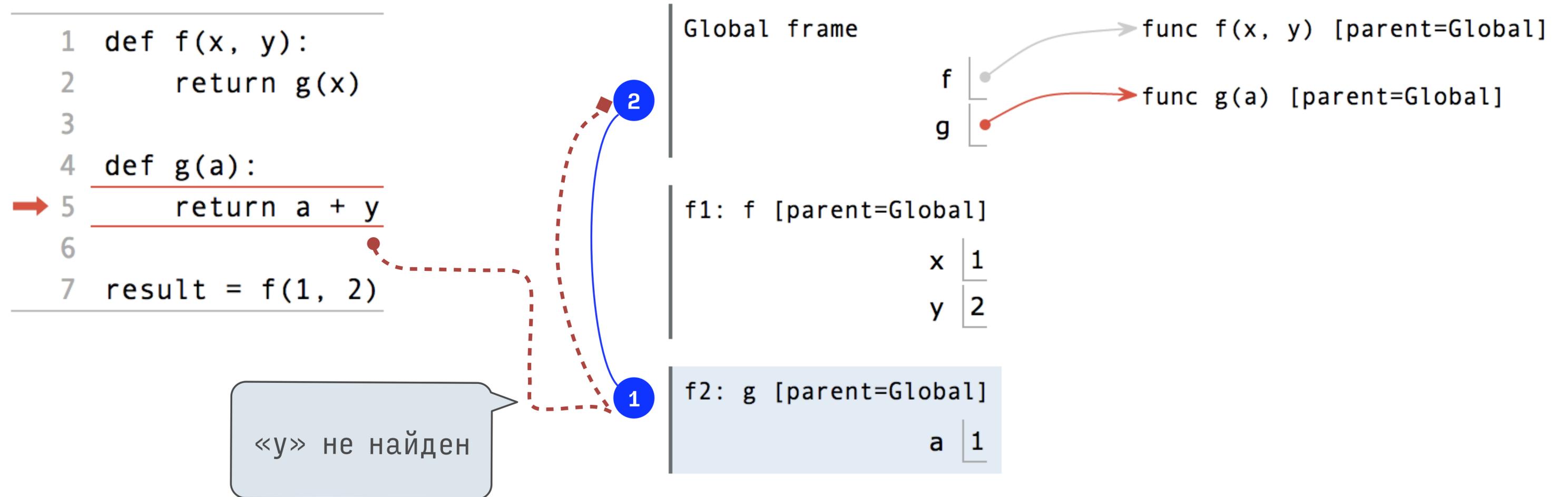
```
1 def f(x, y):  
2     return g(x)  
3  
4 def g(a):  
5     return a + y  
6  
7 result = f(1, 2)
```



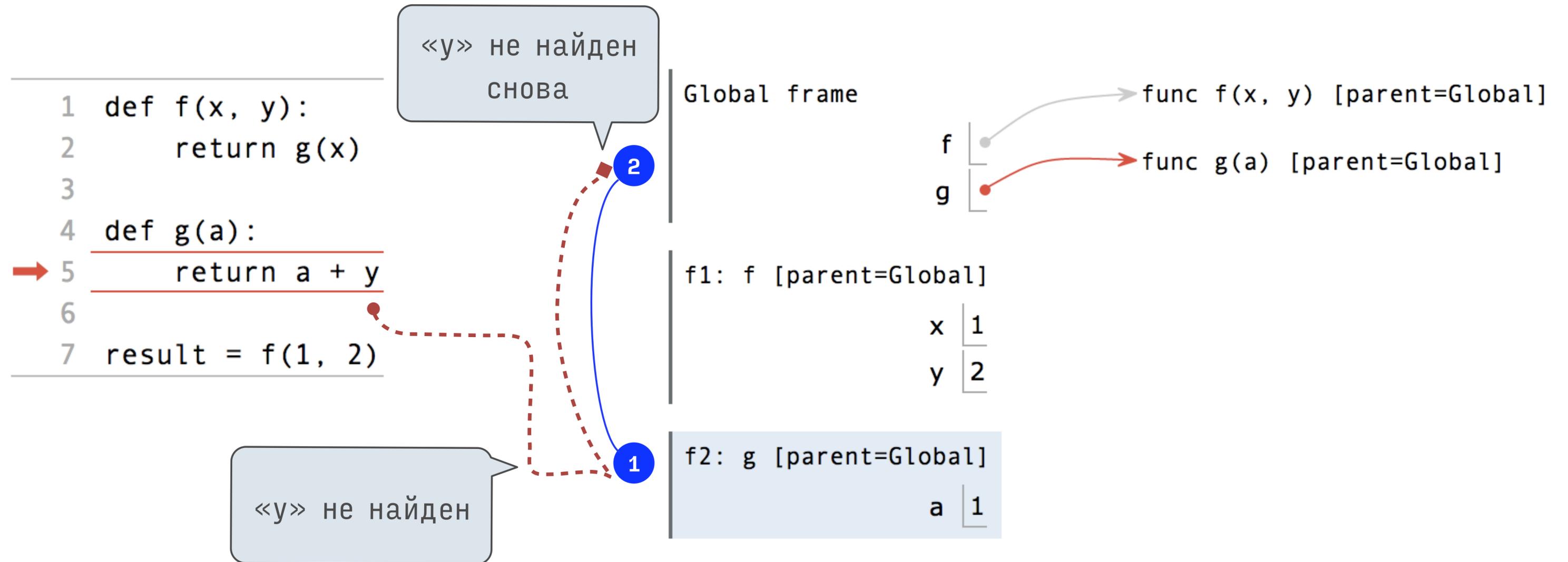
Локальные имена не видны другим (невложенным) функциям



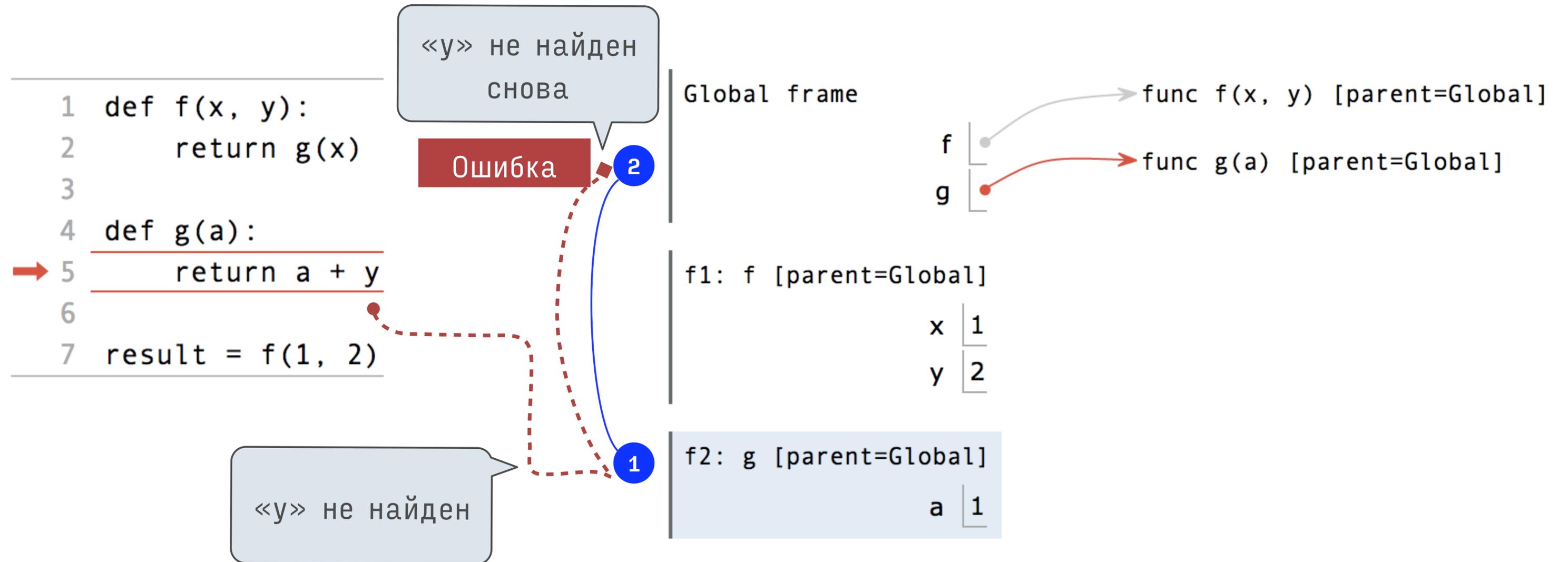
Локальные имена не видны другим (невложенным) функциям



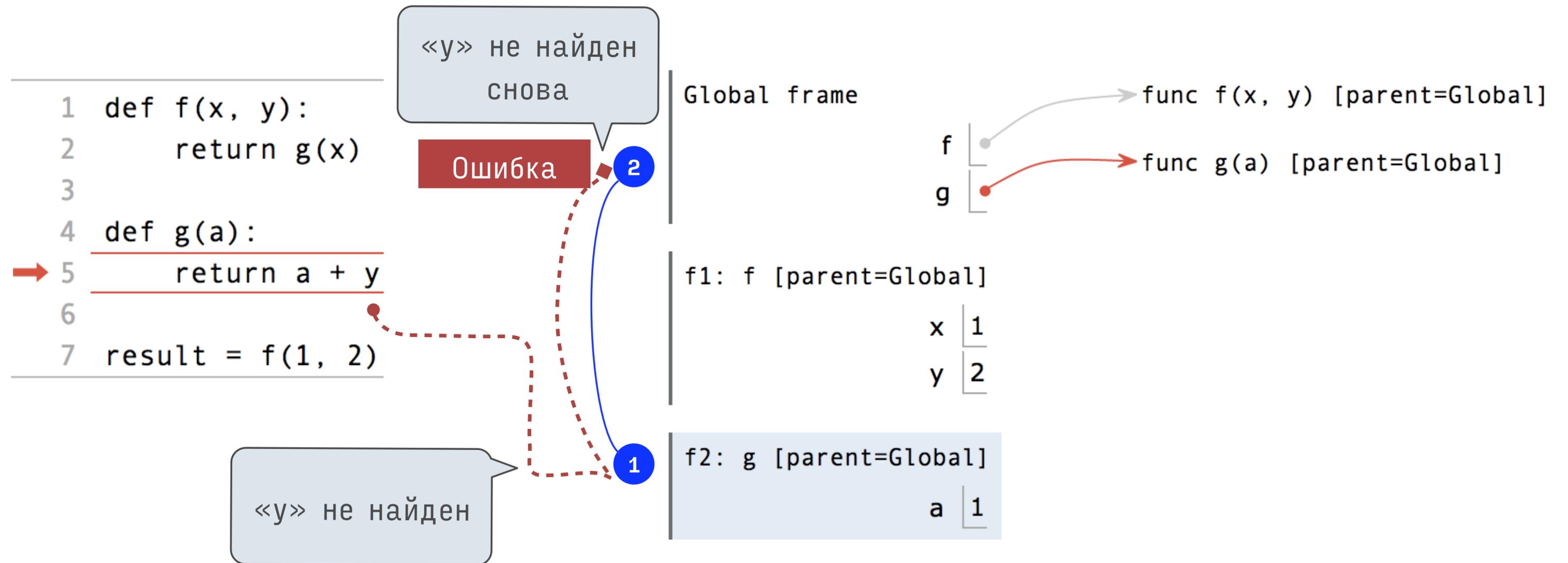
Локальные имена не видны другим (невложенным) функциям



Локальные имена не видны другим (невложенным) функциям

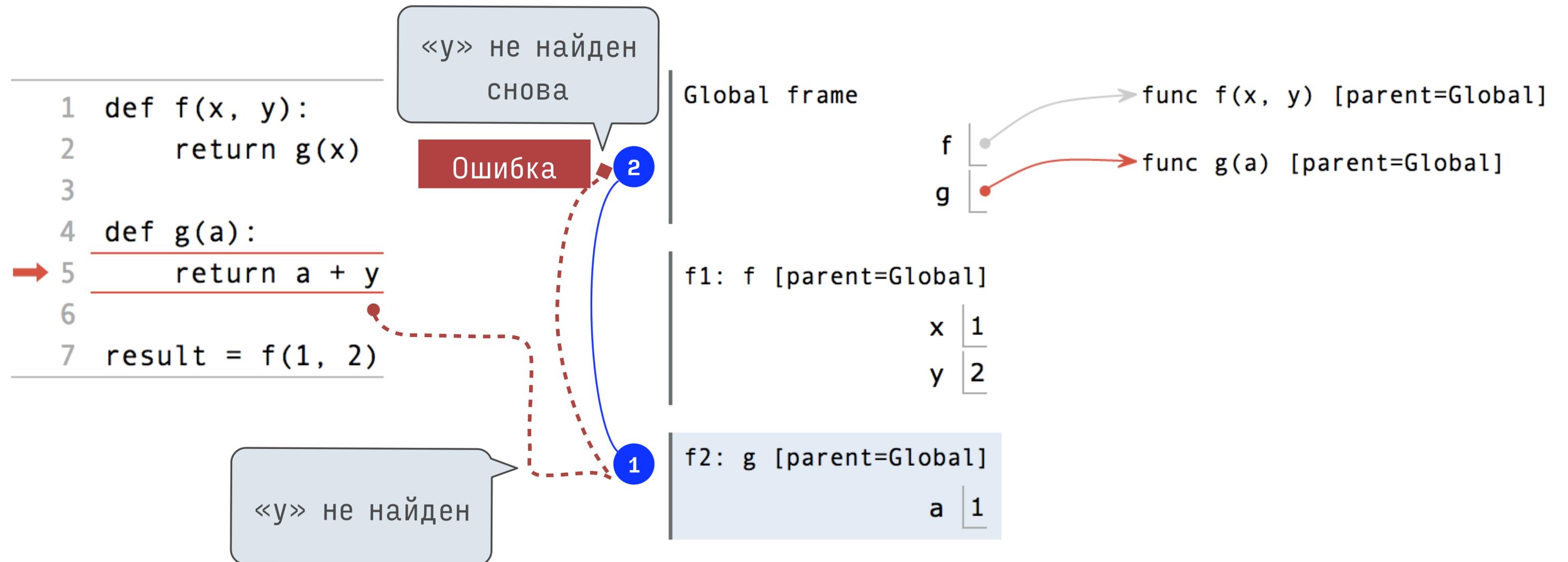


Локальные имена не видны другим (невложенным) функциям



– окружение — последовательность фреймов;

Локальные имена не видны другим (невложенным) функциям



- окружение — последовательность фреймов;
- окружение созданное вызовом функции верхнего уровня (без вложенных def) состоит из локального фрейма, за которым следует глобальный фрейм.

Лямбда – выражения

(Пример)

Лямбда-выражения

Лямбда-выражения

```
>>> x = 10
```

Лямбда-выражения

```
>>> x = 10
```

```
>>> square = x * x
```

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

```
>>> square = lambda x: x * x
```

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Тоже выражение: приводится к функции

```
>>> square = lambda x: x * x
```

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Тоже выражение: приводится к функции

```
>>> square = lambda x: x * x
```

Функция

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Тоже выражение: приводится к функции

```
>>> square = lambda x: x * x
```

Функция

с формальным параметром x

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Тоже выражение: приводится к функции

```
>>> square = lambda x: x * x
```

Функция

с формальным параметром x

которая возвращает значение « $x * x$ »

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Тоже выражение: приводится к функции

```
>>> square = lambda x: x * x
```

Важно: нет инструкции **return**!

Функция

с формальным параметром x

которая возвращает значение « $x * x$ »

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Тоже выражение: приводится к функции

```
>>> square = lambda x: x * x
```

Важно: нет инструкции **return**!

Функция

с формальным параметром x

которая возвращает значение $\langle x * x \rangle$

Должно быть простым выражением

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Тоже выражение: приводится к функции

```
>>> square = lambda x: x * x
```

Важно: нет инструкции **return**!

Функция

с формальным параметром x

которая возвращает значение $\langle x * x \rangle$

```
>>> square(4)
```

```
16
```

Должно быть простым выражением

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Тоже выражение: приводится к функции

```
>>> square = lambda x: x * x
```

Важно: нет инструкции **return**!

Функция

с формальным параметром *x*

которая возвращает значение `<<x * x>>`

```
>>> square(4)
```

```
16
```

Должно быть простым выражением

Лямбда-выражения — не особенность Python, они встречаются в разных языках.

Лямбда-выражения

```
>>> x = 10
```

Выражение: приводится к значению

```
>>> square = x * x
```

Тоже выражение: приводится к функции

```
>>> square = lambda x: x * x
```

Важно: нет инструкции **return**!

Функция

с формальным параметром *x*

которая возвращает значение `<<x * x>>`

```
>>> square(4)
```

```
16
```

Должно быть простым выражением

Лямбда-выражения — не особенность Python, они встречаются в разных языках.

Лямбда-выражения в Python не могут содержать инструкций!

Лямбда-выражение против инструкций «def»



```
square = lambda x: x * x
```

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



Лямбда-выражение против инструкций «def»



```
square = lambda x: x * x
```

VS

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



Лямбда-выражение против инструкций «def»



```
square = lambda x: x * x
```

VS

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



—оба создают функцию с одинаковым поведением, областями определения и значения;

Лямбда-выражение против инструкций «def»



```
square = lambda x: x * x
```

VS

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



- оба создают функцию с одинаковым поведением, областями определения и значения;
- обе функции считают родительским фрейм, в котором они были определены;

Лямбда-выражение против инструкций «def»



```
square = lambda x: x * x
```

VS

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



- оба создают функцию с одинаковым поведением, областями определения и значения;
- обе функции считают родительским фрейм, в котором они были определены;
- обе функции связаны с именем «square».

Лямбда-выражение против инструкций «def»



```
square = lambda x: x * x
```

VS

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



- оба создают функцию с одинаковым поведением, областями определения и значения;
- обе функции считают родительским фрейм, в котором они были определены;
- обе функции связаны с именем «square».
- только инструкция «def» назначает функции внутреннее имя.

Лямбда-выражение против инструкций «def»



```
square = lambda x: x * x
```

VS

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



- оба создают функцию с одинаковым поведением, областями определения и значения;
- обе функции считают родительским фрейм, в котором они были определены;
- обе функции связаны с именем «square».
- только инструкция «def» назначает функции внутреннее имя.

```
Global frame  
square |  
-----  
f1: λ <line 1> [parent=Global]  
      x | 4  
      Return value | 16
```

Лямбда-выражение против инструкций «def»



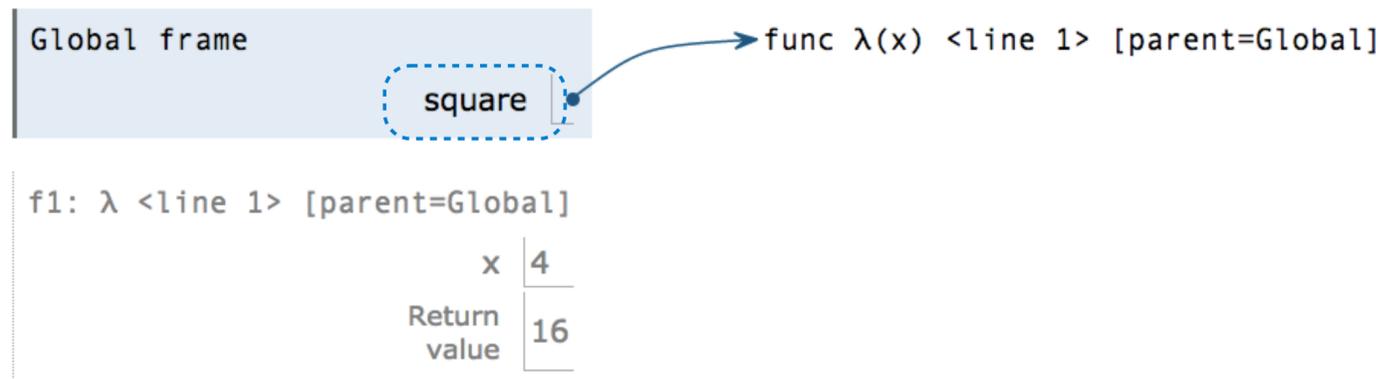
```
square = lambda x: x * x
```

VS

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



- оба создают функцию с одинаковым поведением, областями определения и значения;
- обе функции считают родительским фрейм, в котором они были определены;
- обе функции связаны с именем «square».
- только инструкция «def» назначает функции внутреннее имя.



Лямбда-выражение против инструкций «def»



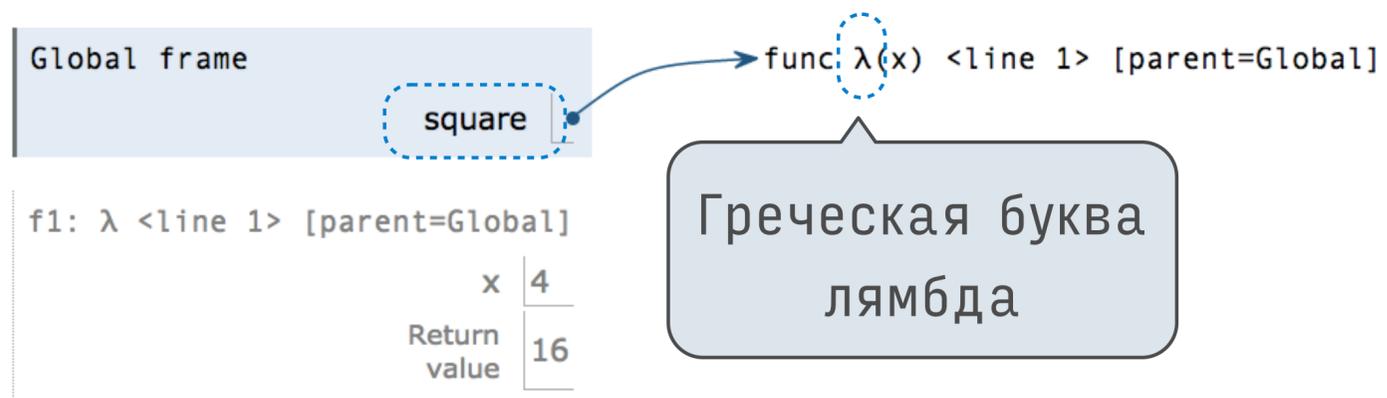
```
square = lambda x: x * x
```

VS

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



- оба создают функцию с одинаковым поведением, областями определения и значения;
- обе функции считают родительским фрейм, в котором они были определены;
- обе функции связаны с именем «square».
- только инструкция «def» назначает функции внутреннее имя.



Лямбда-выражение против инструкций «def»



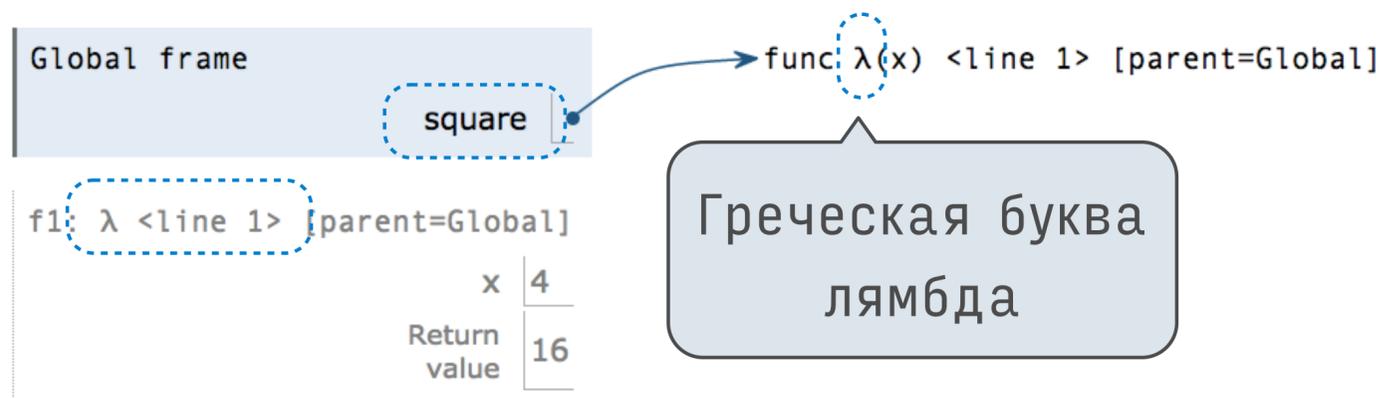
```
square = lambda x: x * x
```

VS

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



- оба создают функцию с одинаковым поведением, областями определения и значения;
- обе функции считают родительским фрейм, в котором они были определены;
- обе функции связаны с именем «square».
- только инструкция «def» назначает функции внутреннее имя.



Лямбда-выражение против инструкций «def»



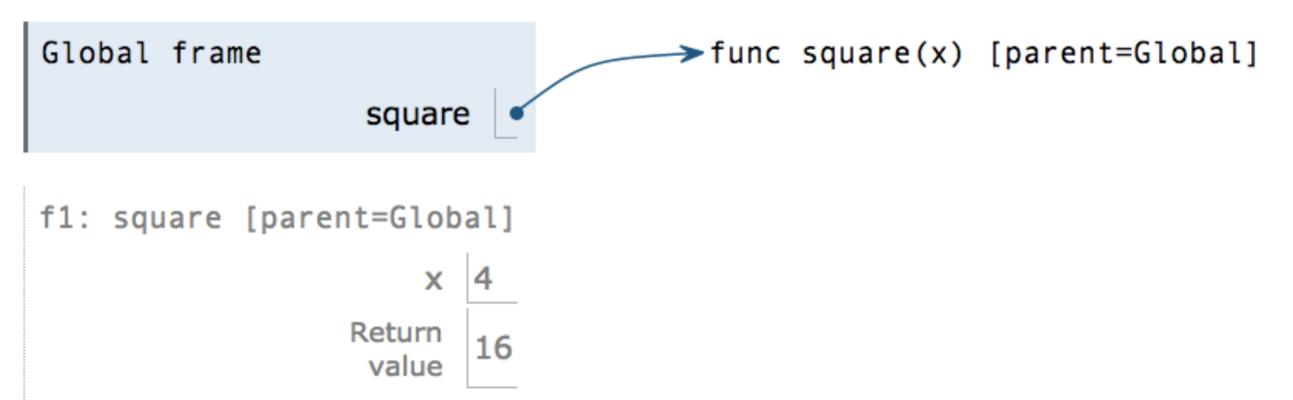
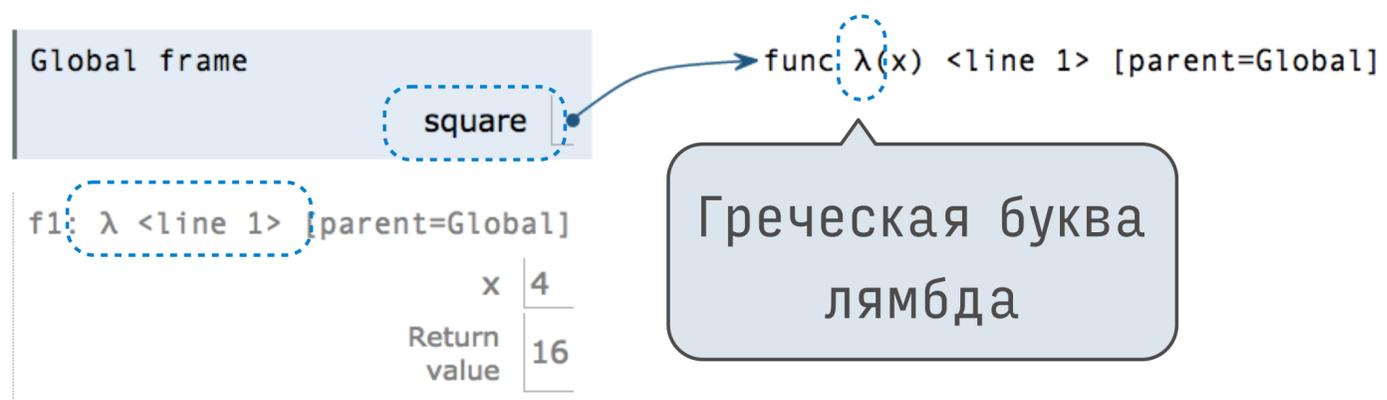
```
square = lambda x: x * x
```

VS

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



- оба создают функцию с одинаковым поведением, областями определения и значения;
- обе функции считают родительским фрейм, в котором они были определены;
- обе функции связаны с именем «square».
- только инструкция «def» назначает функции внутреннее имя.



Каррирование

Каррирование функций

Каррирование функций

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

Каррирование функций

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

```
>>> make_adder(2)(3)
```

```
5
```

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

```
5
```

Каррирование функций

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

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

Между этими функциями есть
глубокая связь

Каррирование функций

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

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

Между этими функциями есть
глубокая связь

(Пример)

Каррирование функций

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

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

Между этими функциями есть глубокая связь

(Пример)

Каррирование: преобразование функции от многих аргументов в функцию высшего порядка, берущую свои аргументы по одному.

Каррирование функций

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

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

Между этими функциями есть
глубокая связь

(Пример)

Каррирование: преобразование функции от многих аргументов в функцию высшего порядка, берущую свои аргументы по одному.

Преобразование было введено Моисеем Шейнфинкелем, но получило свое название позже в честь Хаскелла Карри.

Каррирование функций

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

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

Между этими функциями есть глубокая связь



Каррирование: преобразование функции от многих аргументов в функцию высшего порядка, берущую свои аргументы по одному.

Преобразование было введено Моисеем Шейнфинкелем, но получило свое название позже в честь Хаскелла Карри.

Каррирование функций

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

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

Между этими функциями есть глубокая связь



Каррирование: преобразование функции от многих аргументов в функцию высшего порядка, берущую свои аргументы по одному.

Преобразование было введено Моисеем Шейнфинкелем, но получило свое название позже в честь Хаскелла Карри.

Шейнфинкелирование?

Композиция функций

(Пример)

Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

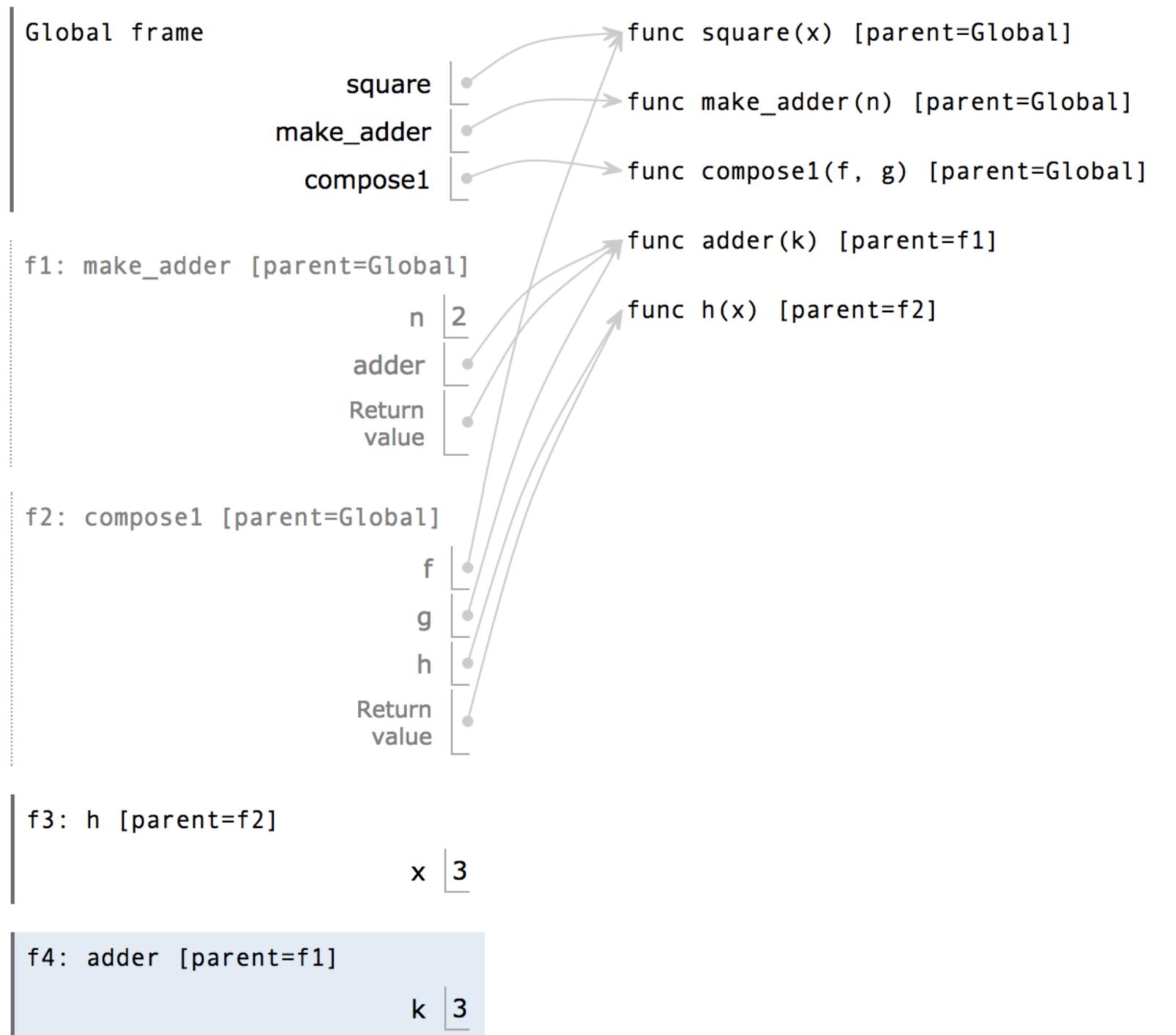


Диаграмма окружения при композиции функций

```
1 def square(x):  
2     return x * x  
3  
4 def make_adder(n):  
5     def adder(k):  
6         return k + n  
7     return adder  
8  
9 def compose1(f, g):  
10    def h(x):  
11        return f(g(x))  
12    return h  
13  
14 compose1(square, make_adder(2))(3)
```

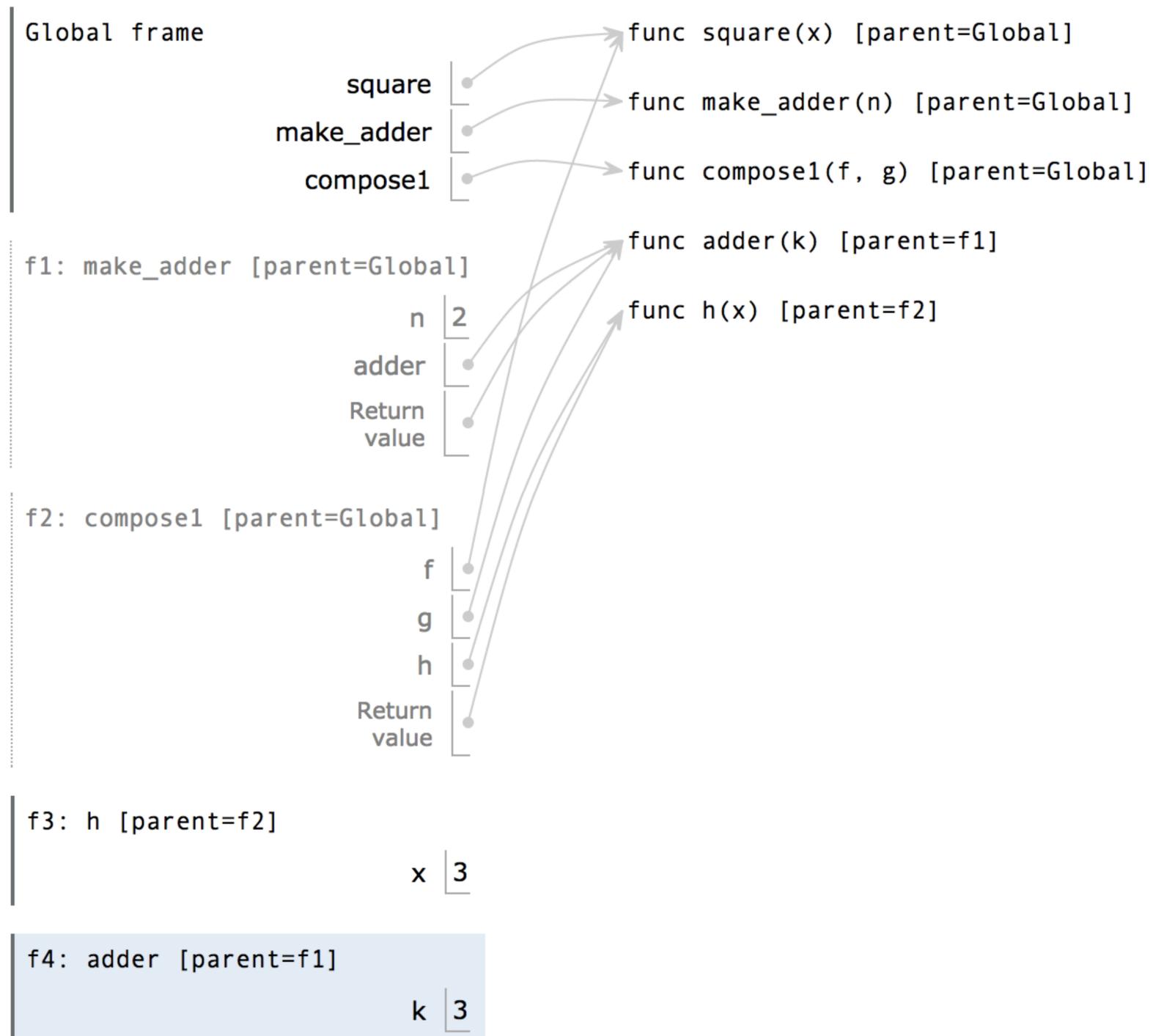


Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

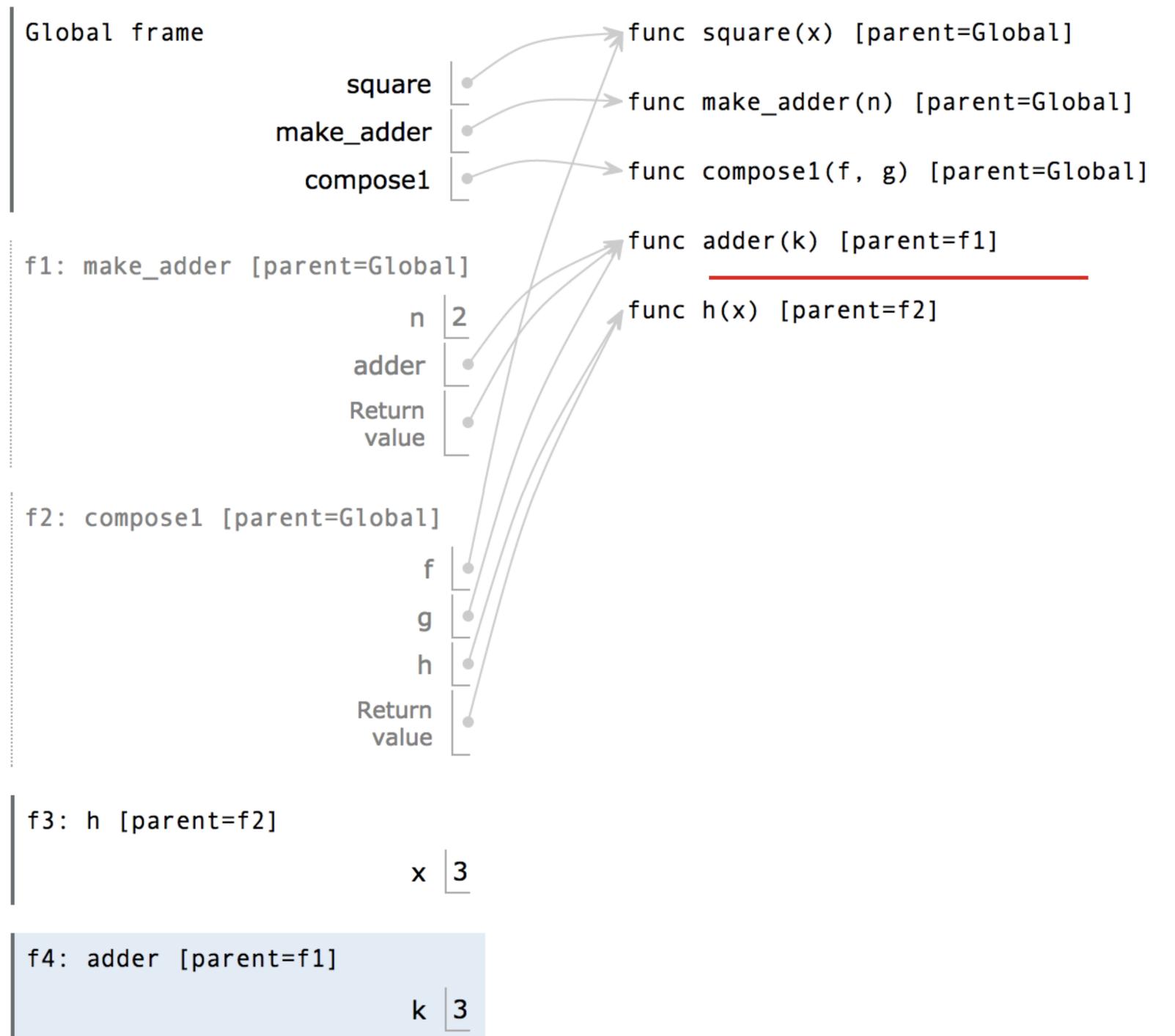


Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

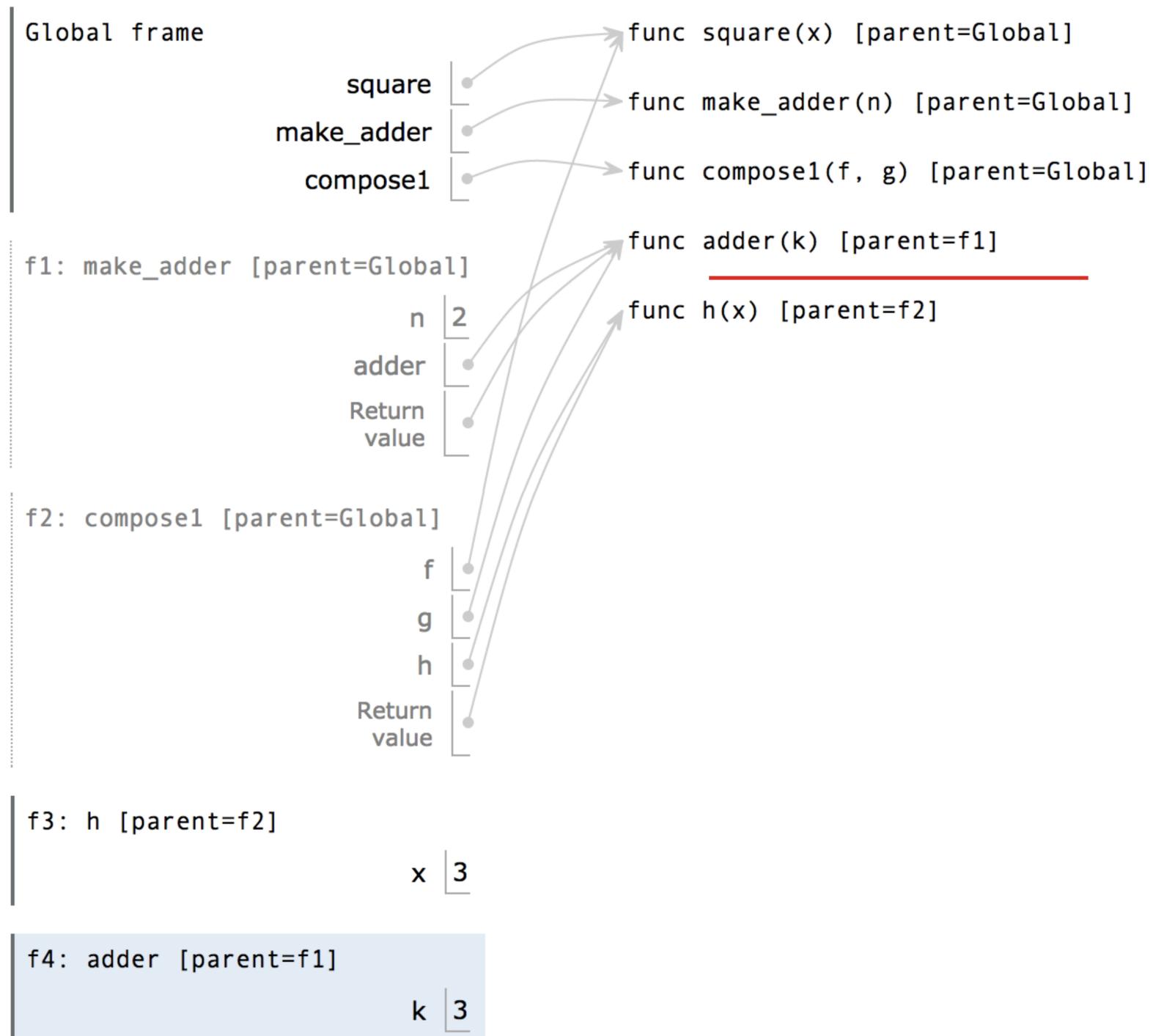


Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

Возвращаемое из `make_adder`
значение — это значение
аргумента для `compose1`



Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

Возвращаемое из `make_adder`
значение — это значение
аргумента для `compose1`

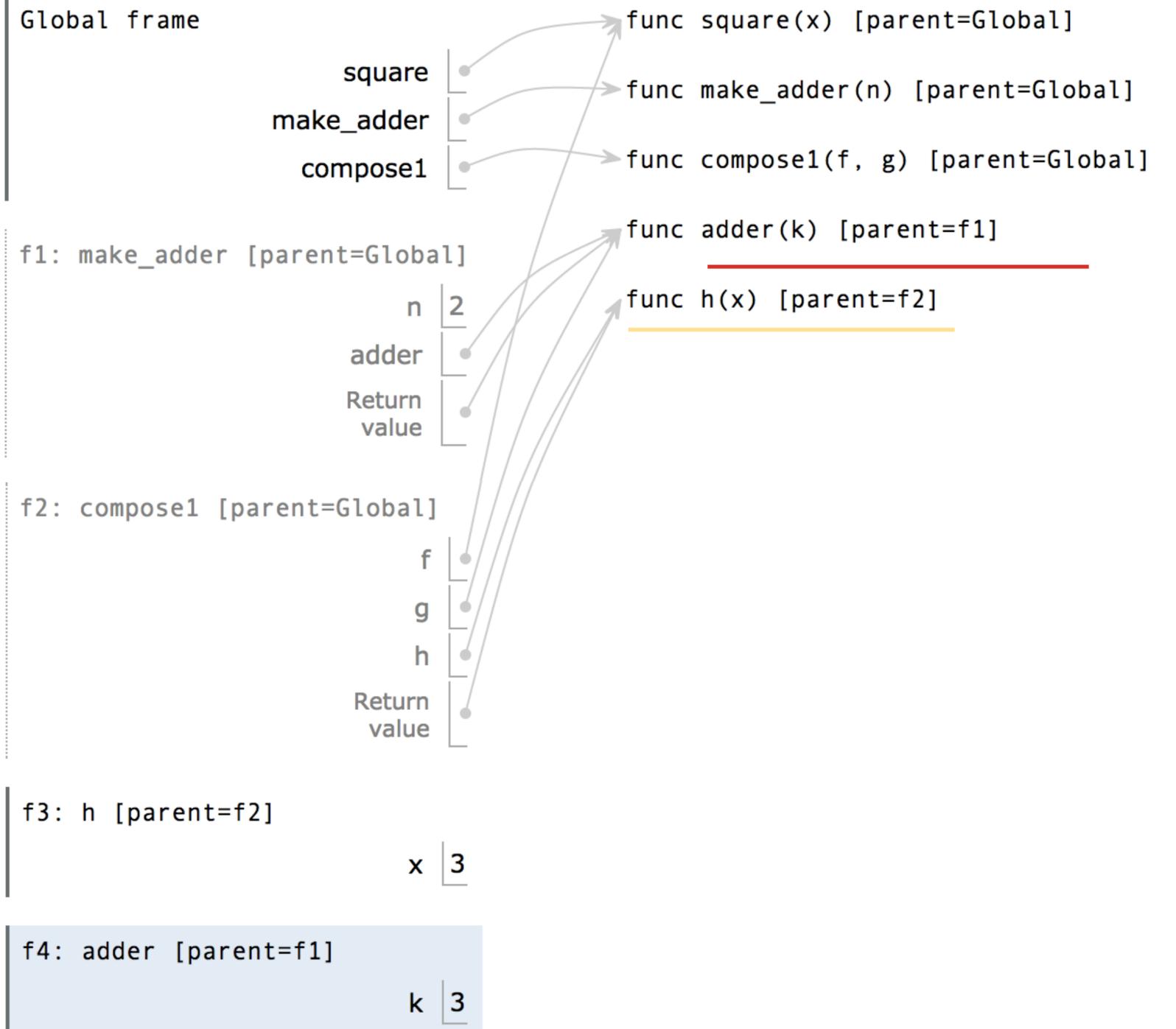


Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

Возвращаемое из `make_adder`
значение — это значение
аргумента для `compose1`

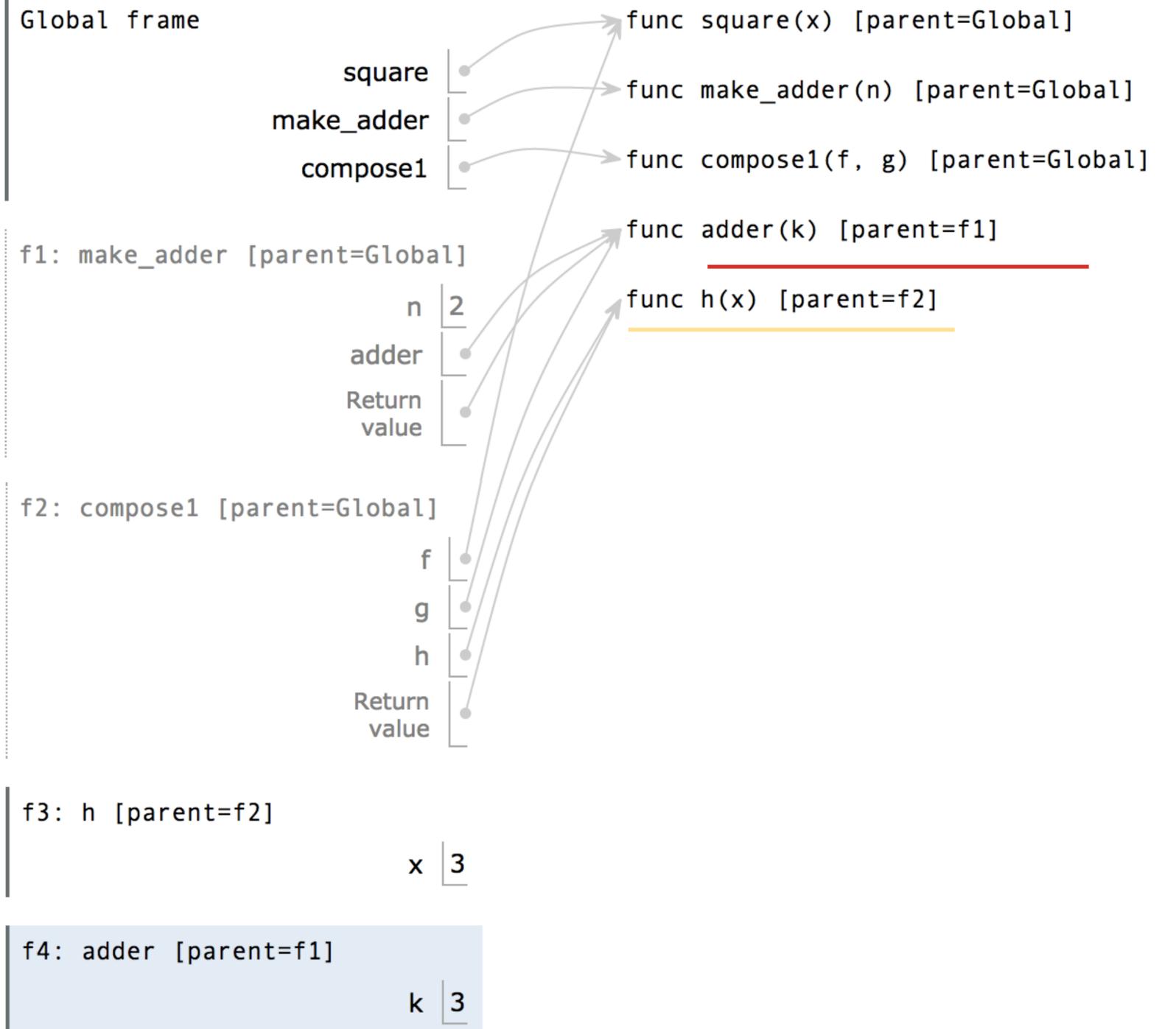


Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

Возвращаемое из `make_adder`
значение — это значение
аргумента для `compose1`

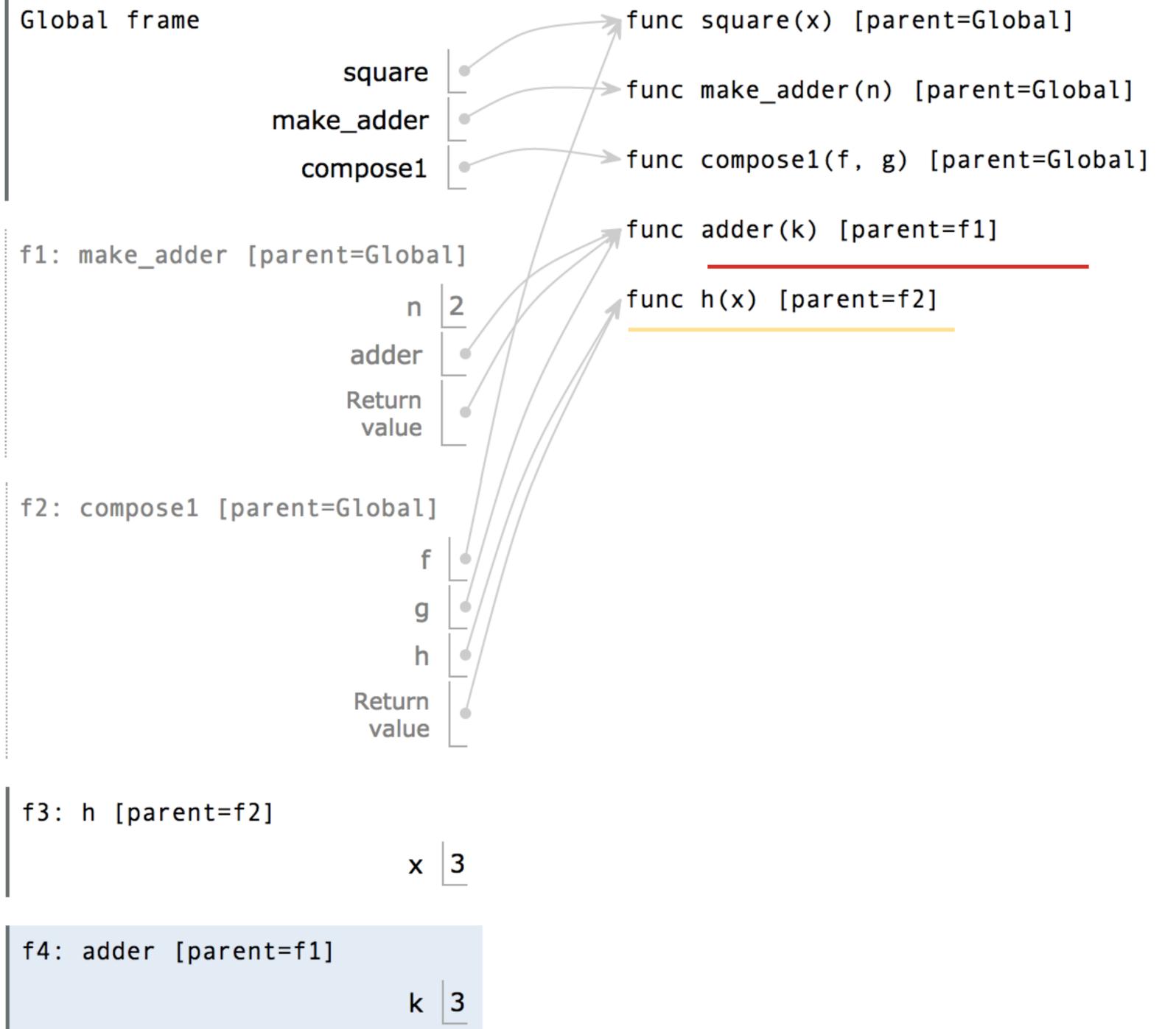


Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

Возвращаемое из `make_adder`
значение — это значение
аргумента для `compose1`

3

2

1

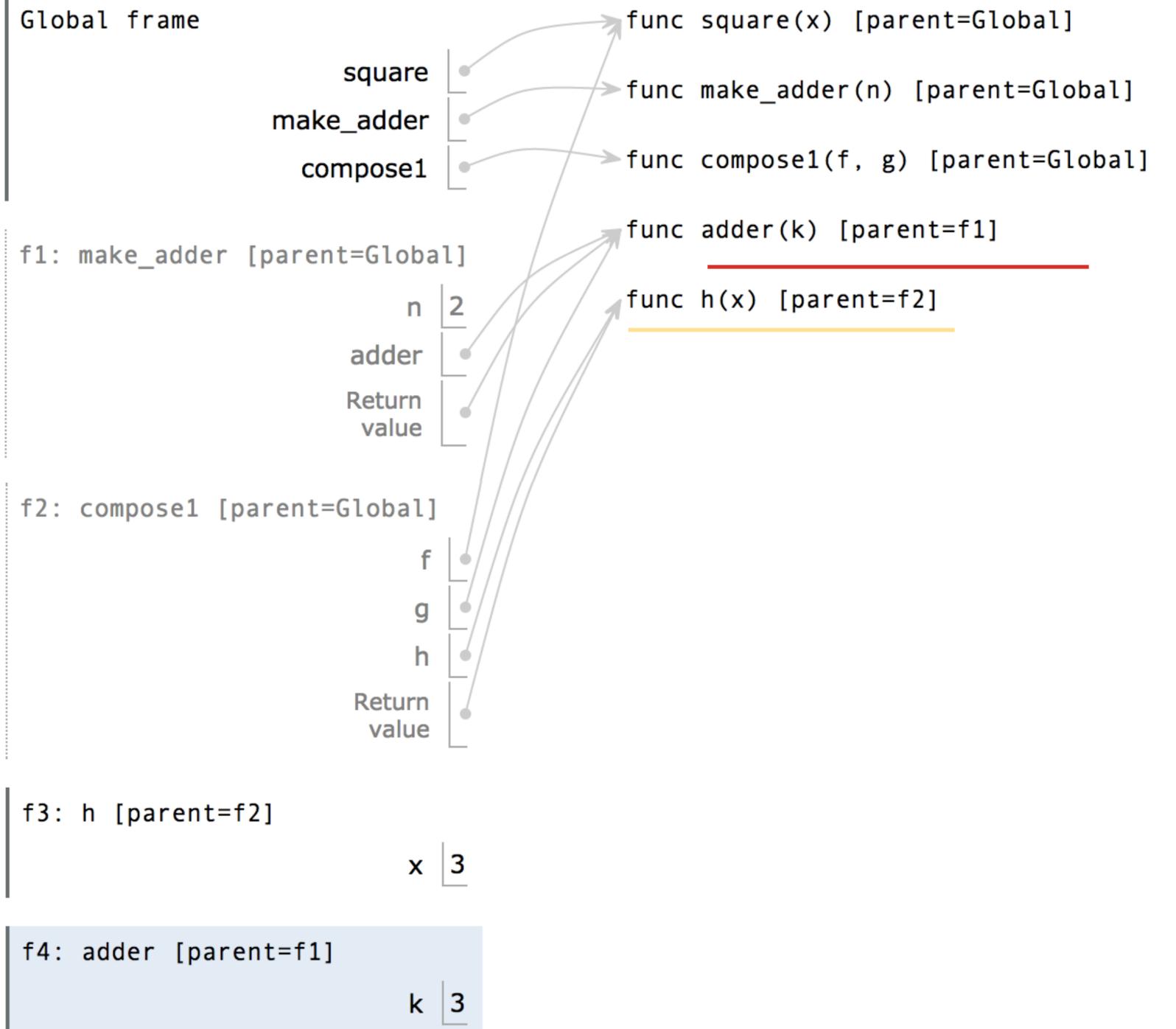


Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

Возвращаемое из `make_adder`
значение — это значение
аргумента для `compose1`

3

2

1

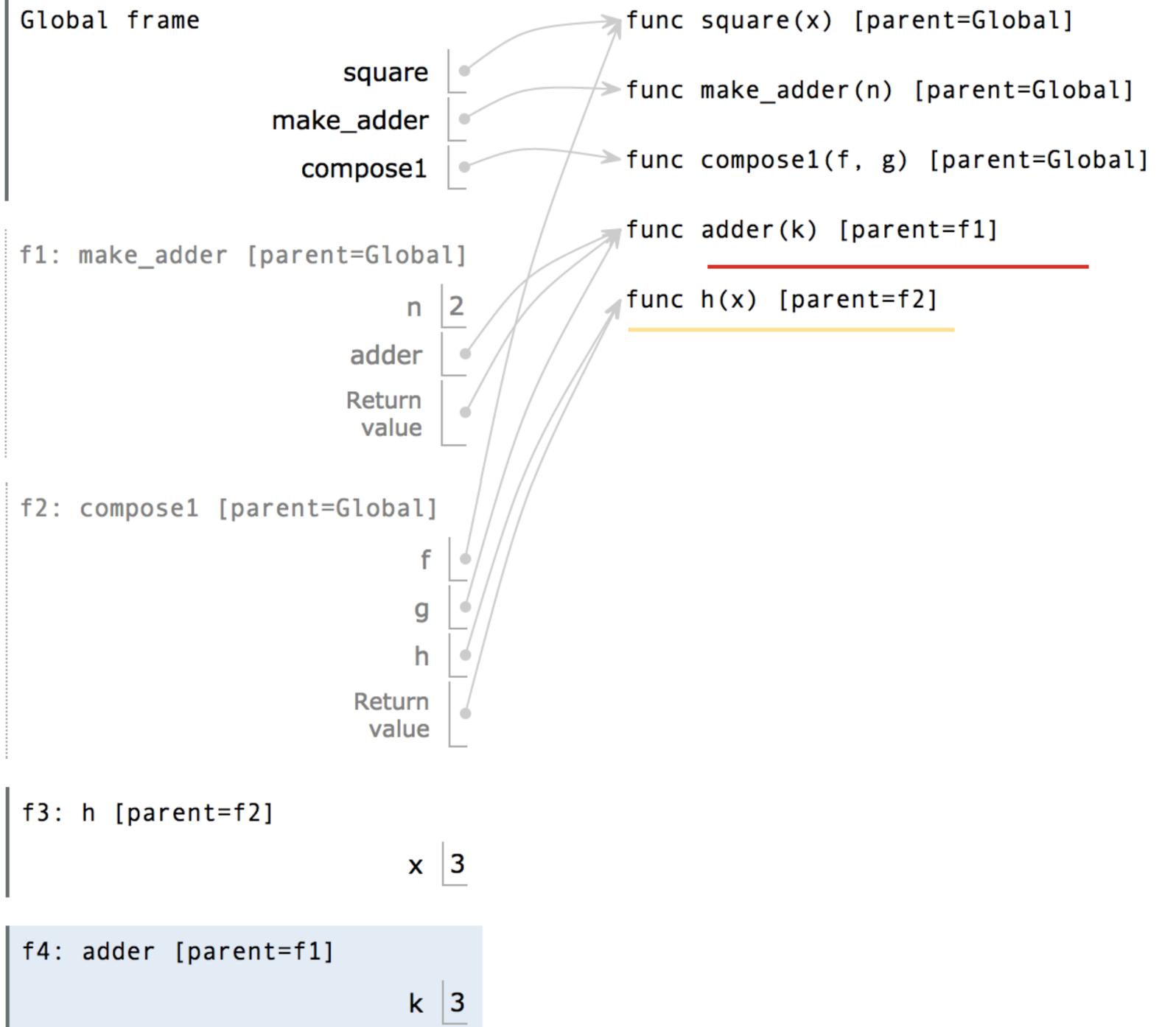


Диаграмма окружения при композиции функций

```
1 def square(x):
2     return x * x
3
4 def make_adder(n):
5     def adder(k):
6         return k + n
7     return adder
8
9 def compose1(f, g):
10    def h(x):
11        return f(g(x))
12    return h
13
14 compose1(square, make_adder(2))(3)
```

Возвращаемое из `make_adder`
значение — это значение
аргумента для `compose1`

