[ИУ9] Основы информатики

Лекция 10. Стек вызовов в Scheme. Продолжения

В сегодняшней лекции мы рассмотрим, как интерпретатор Scheme выполняет вызовы функций и какие средства контроля за этим Scheme даёт программисту.

Стек вызовов. Как осуществляются вызовы функций на Scheme

Выполнение программы на языке Scheme состоит из двух этапов: раскрытие всех макроподстановок и синтаксического сахара до базовых примитивов языка и выполнение программы, записанной в терминах этих базовых примитивов. Сегодня мы будем рассматривать второй этап вычислений, поэтому большинство примеров кода будет написано в терминах базовых примитивов.

В число базовых примитивов, помимо `define`, `if`, `lambda`, `quote`, `set!`, мы добавим также `begin` для удобства изложения (хотя он может быть выражен через `let`, а тот, в свою очередь, через `lambda`).

Процесс вычисления выражений мы будем изображать путём редукции: выбираем очередное подвыражение и заменяем его на результат.

Пример. Рассмотрим редукцию выражения

```
((lambda (foo bar)
   (begin
     (set! bar (+ foo bar))
     (* foo bar 2)))
 3 4)
```

Шаги редукции:

```
Выражение                                                  Среда

((lambda (foo bar) (begin (set! bar (+ foo bar)) (* foo bar 2))) 3 4)
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
; вызывается lambda, появляется среда с локальными переменными

(begin (set! bar (+ foo bar)) (* foo bar 2))                foo → 3, bar → 4
```

```
                    ^^^
; подстановка значения вместо переменной foo

(begin (set! bar (+ 3 bar)) (* foo bar 2))            foo → 3, bar → 4
                      ^^^
; подстановка значения вместо переменной bar

(begin (set! bar (+ 3 4)) (* foo bar 2))              foo → 3, bar → 4
                  ^
; вызов встроенной функции +

(begin (set! bar 7) (* foo bar 2))                    foo → 3, bar → 4
        ^^^^
; выполнение оператора присваивания

(begin #<void> (* foo bar 2))                         foo → 3, bar → 7
 ^^^^^
; редукция begin

(* foo bar 2)                                         foo → 3, bar → 7
   ^^^
; подстановка значения вместо переменной foo

(* 3 bar 2)                                           foo → 3, bar → 7
     ^^^
; подстановка значения вместо переменной bar

(* 3 7 2)                                             foo → 3, bar → 7
 ^
; вызов встроенной функции *

42
```

При выполнении вызовов функций мы будем предполагать, что аргументы вычисляются слева направо. Согласно R^{5}RS, порядок вычислений не определён, но среда DrRacket вычисляет их слева направо. Также, вслед за DrRacket, будем считать, что операция `set!` и функции типа `display` возвращают значение типа `#<void>` (согласно R^{5}RS значение также не определено).

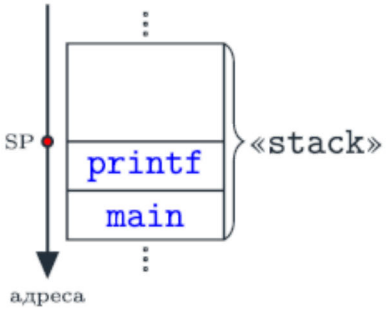
В последующих выкладках некоторые очевидные шаги редукции (вроде подстановки значений вместо переменных) мы будем опускать.

В примере на редукцию выше, мы не касались вопроса о механизме вызовов функций. Сейчас рассмотрим его подробнее.

По курсу «Алгоритмы и структуры данных» мы знакомы с понятием «стек вызовов» и с тем, как в языке Си осуществляются вызовы функций:

Фреймы функций в автоматической памяти

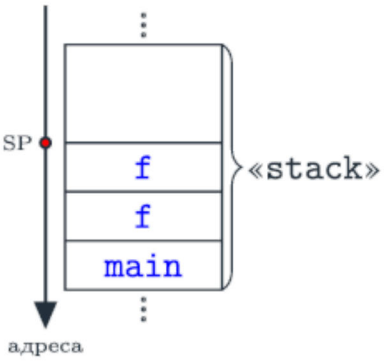
Фрейм функции – это участок автоматической памяти, в котором хранятся локальные переменные (включая формальные параметры) и адрес, на который должно быть передано управление после завершения функции (*адрес возврата*).



При запуске программы в старших адресах области «stack» создаётся фрейм функции `main` и управление передаётся по её адресу.
Если, например, из функции `main` вызывается функция `printf`, то перед фреймом `main` размещается фрейм `printf`. При этом в качестве адреса возврата в этот фрейм помещается адрес инструкции функции `main`, которая следует за инструкцией вызова функции `printf`.
При завершении функции `printf` управление передаётся по записанному в её фрейме адресу возврата, а сам фрейм уничтожается.

Поддержка рекурсии

Хранение локальных переменных функции во фрейме, который создаётся в момент вызова функции, обеспечивает возможность *рекурсии*, т.е. вызова функцией самой себя.



Например, пусть из функции `main` вызывается функция `f`, а из неё ещё раз вызывается функция `f`. В результате в области «stack» появляются два фрейма функции `f`, каждый со своим набором локальных переменных. Естественно, локальные переменные (в которые входят и формальные параметры) в разных фреймах функции `f` никак не пересекаются и могут иметь разные значения. Бывают более сложные виды рекурсии, например, когда функция `f` вызывает `g`, а та, в свою очередь, вызывает `f`. Отметим, что использование рекурсии может привести к исчерпанию свободного места в области «stack».

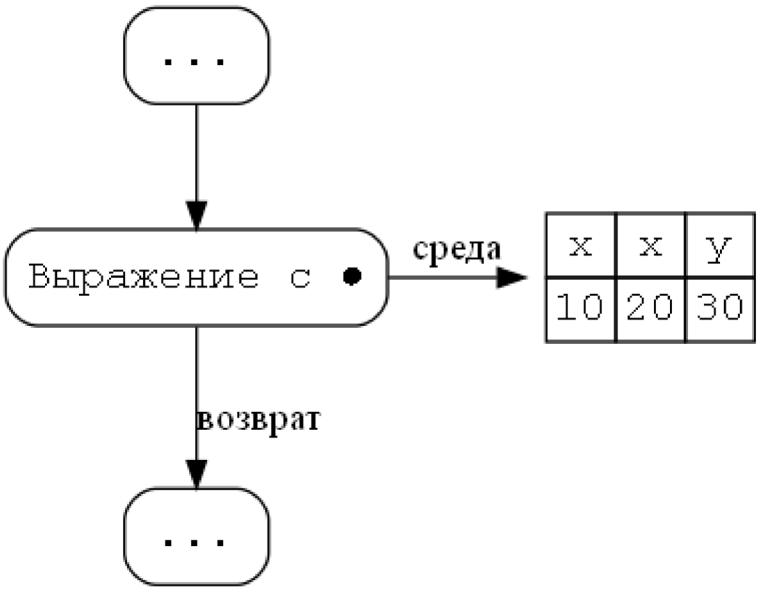
17 / 164

В языке Scheme дело обстоит аналогичным образом, с той лишь разницей, что и фреймы стека, и данные распределяются в динамической памяти. И если ссылка на фрейм где-то сохранена, то объект фрейма останется «жить» даже после возврата функции.

Фреймы стека языка Си содержат адрес возврата и локальные переменные. Параметры функций являются разновидностью локальных переменных.

Как мы помним, конструкции `let`, `let*` и `letrec`, а также `define` внутри `begin`, вводящие локальные переменные, являются синтаксическим сахаром, реализованным поверх `lambda`. Поэтому для языка Scheme локальные переменные во фреймах стека — это *только* параметры функций.

Фрейм стека будем изображать следующим образом:



Фрейм стека содержит две ссылки. Одна из называется «среда» и ссылается на значения локальных переменных — аргументов функции. Вторая — «возврат» ссылается на предыдщий фрейм стека. Если фрейм стека не верхний, то выражение будет содержать символ `•`, означающий точку, куда будет возвращено выполнение вызова другой функции.

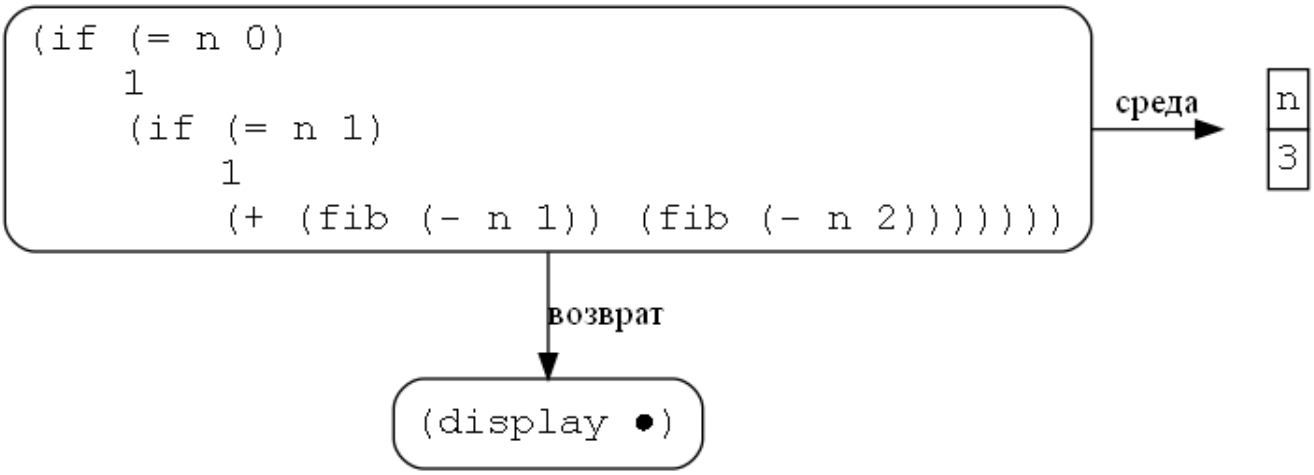
Рассмотрим пример — вычисление числа Фибоначчи по номеру. Пусть нам дана функция

```
(define fib
  (lambda (n)
    (if (= n 0)
        1
        (if (= n 1)
            1
            (+ (fib (- n 1)) (fib (- n 2)))))))
```

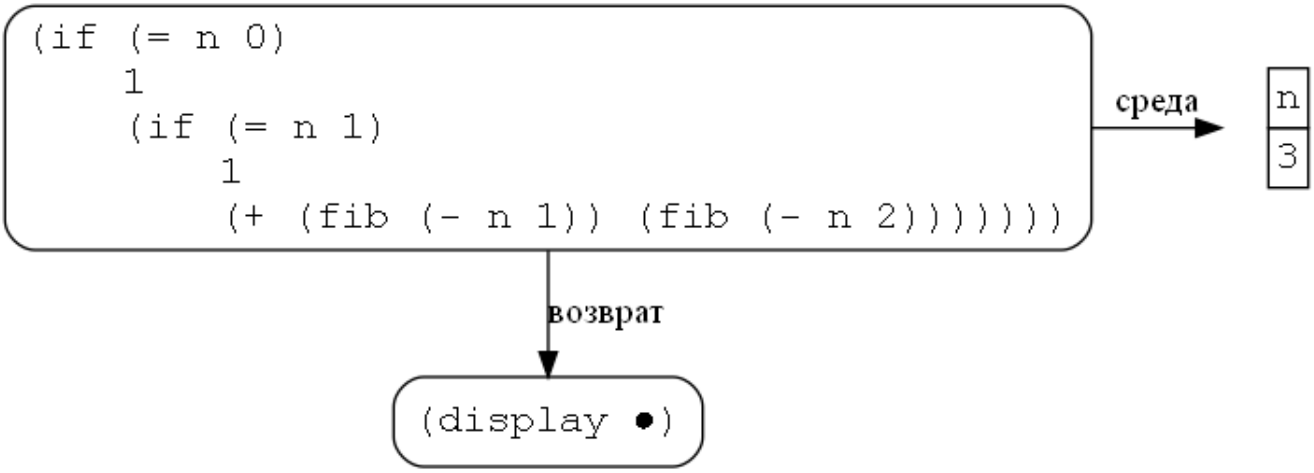
Рассмотрим процесс вычисления выражения

```
(display (fib 3))
```

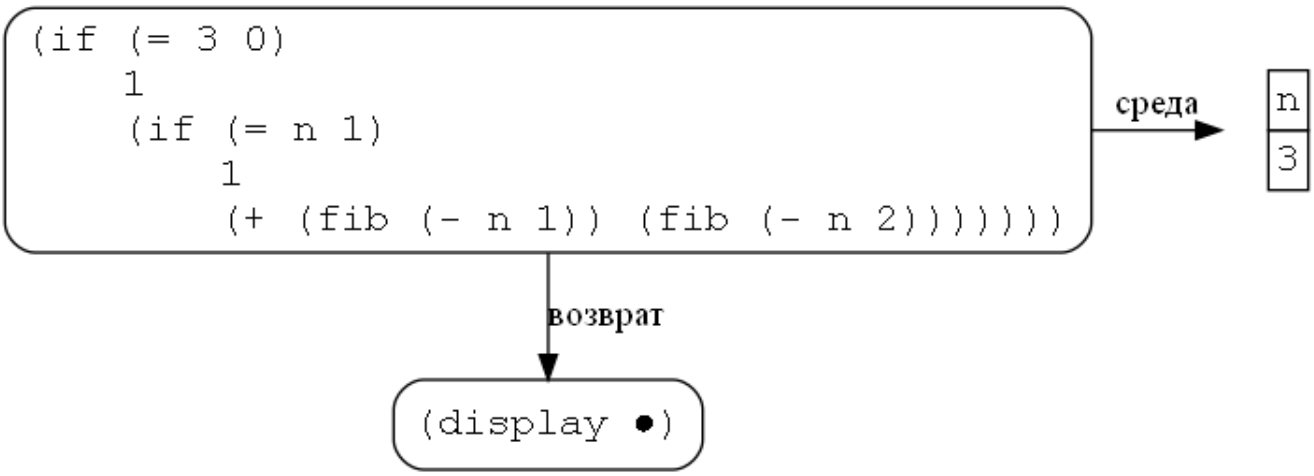
Начальное состояние:



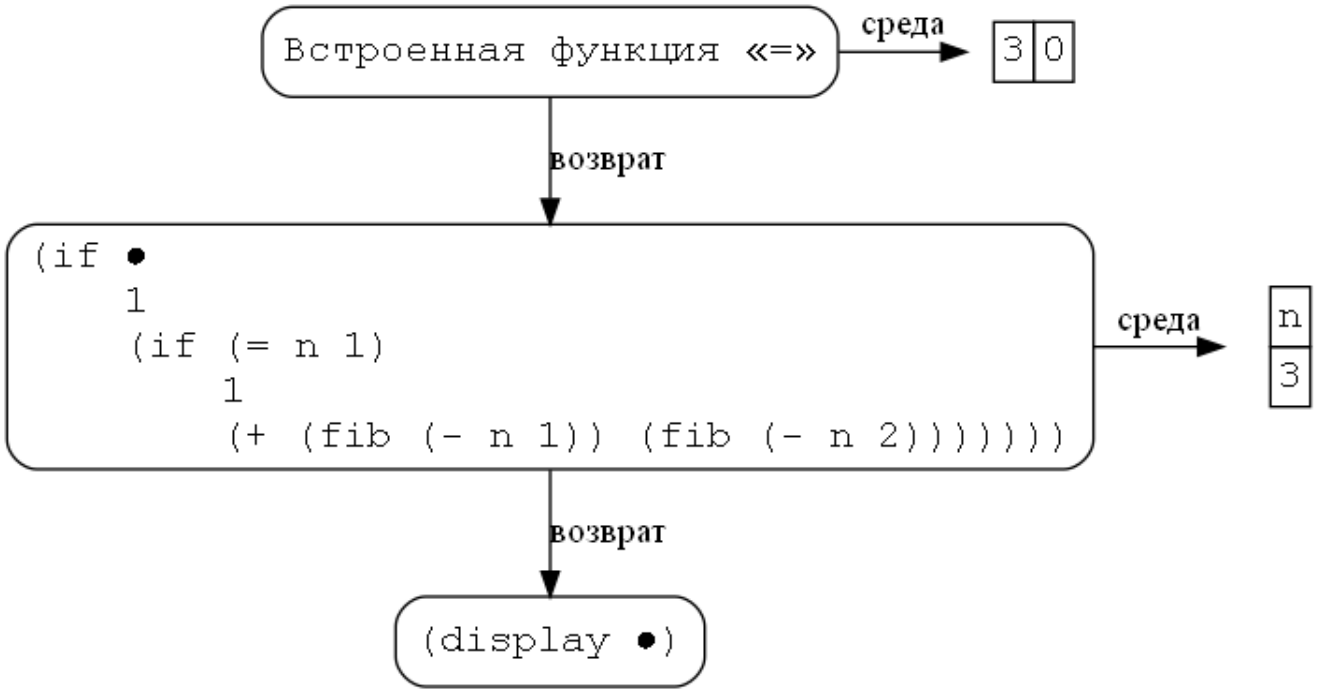
Создаётся фрейм стека для `(fib 3)`:



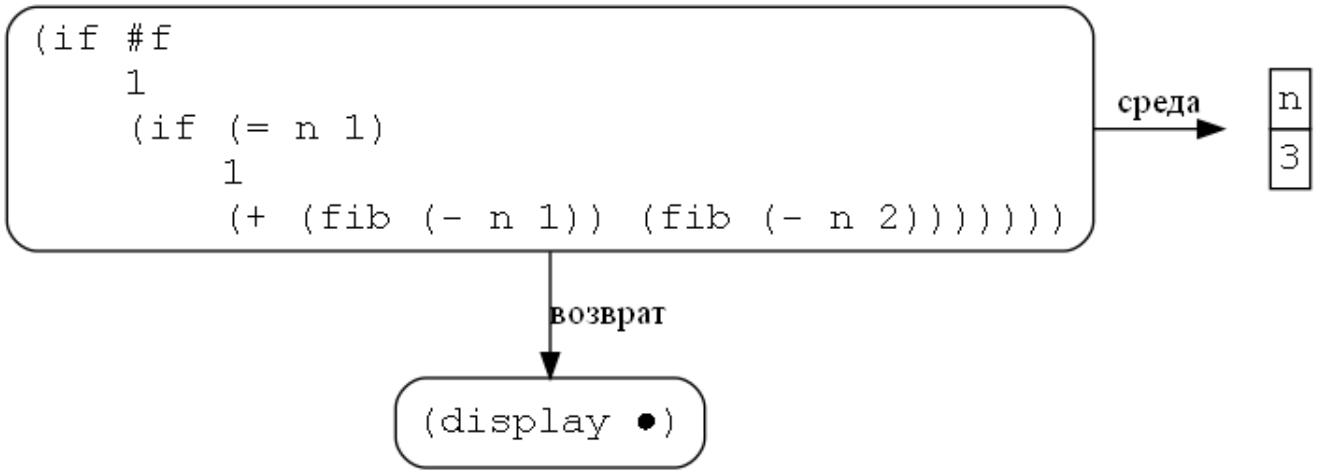
Подстановка значения вместо переменной в `if`:



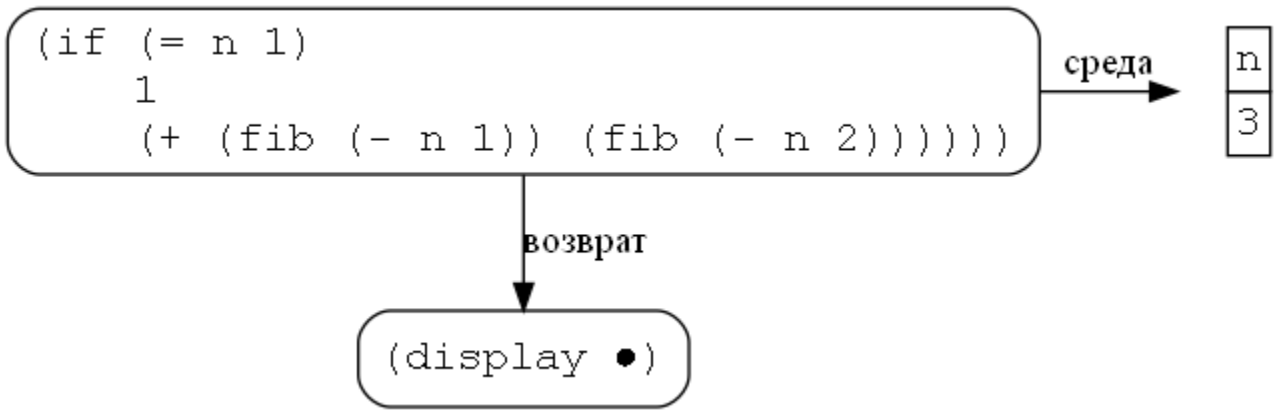
Аргументы для встроенной функции `=` вычислены, вызов встроенной функции. Имена переменных в среде не указаны, т.к. функция встроенная и мы их не знаем.



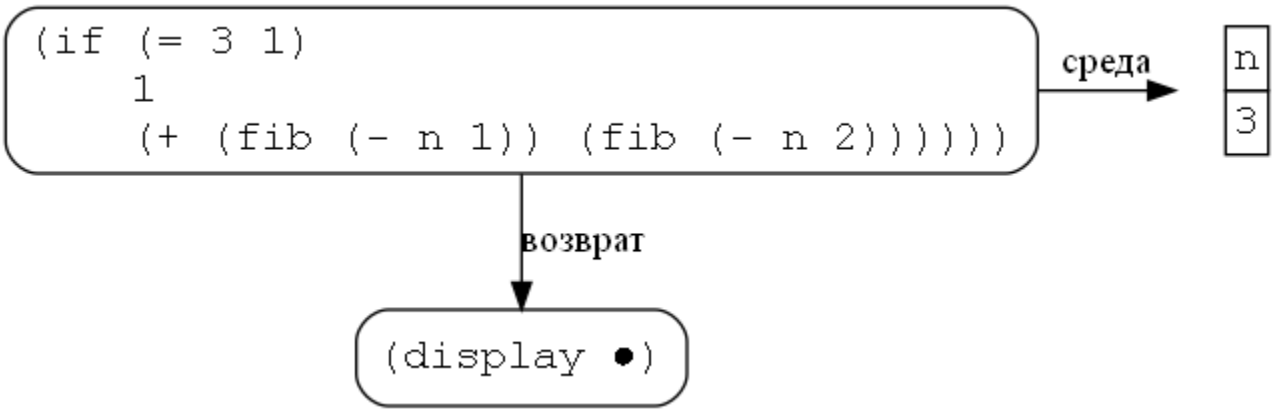
Функция `=` вернула ложь:



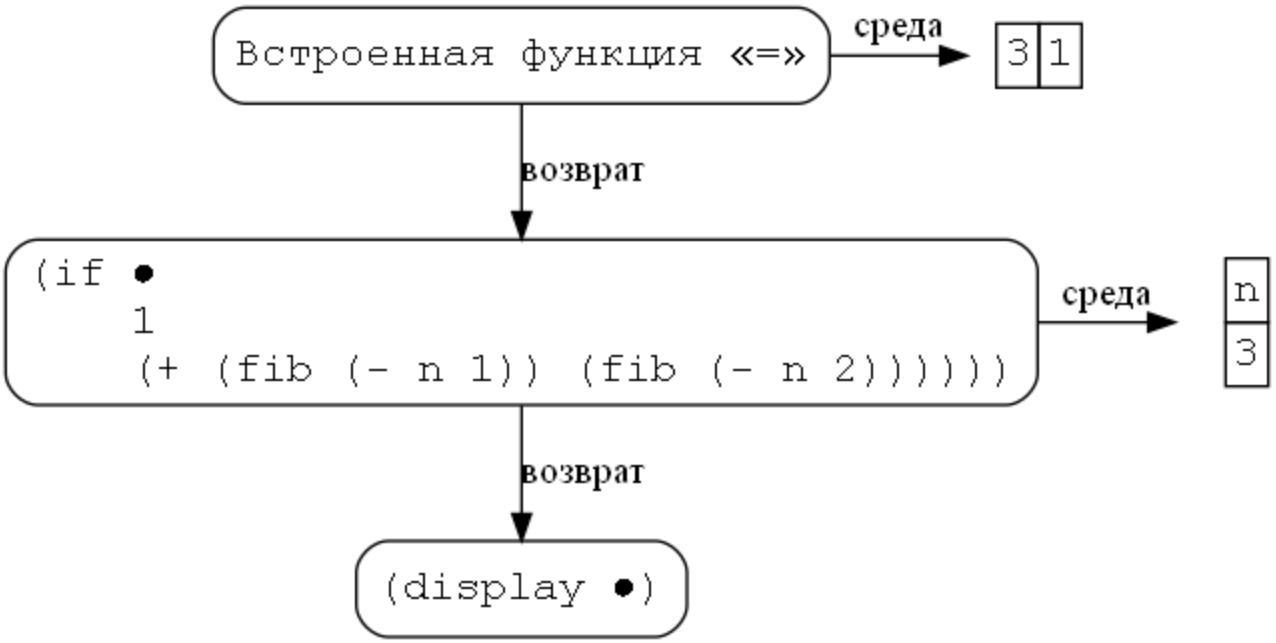
Редукция `if`:



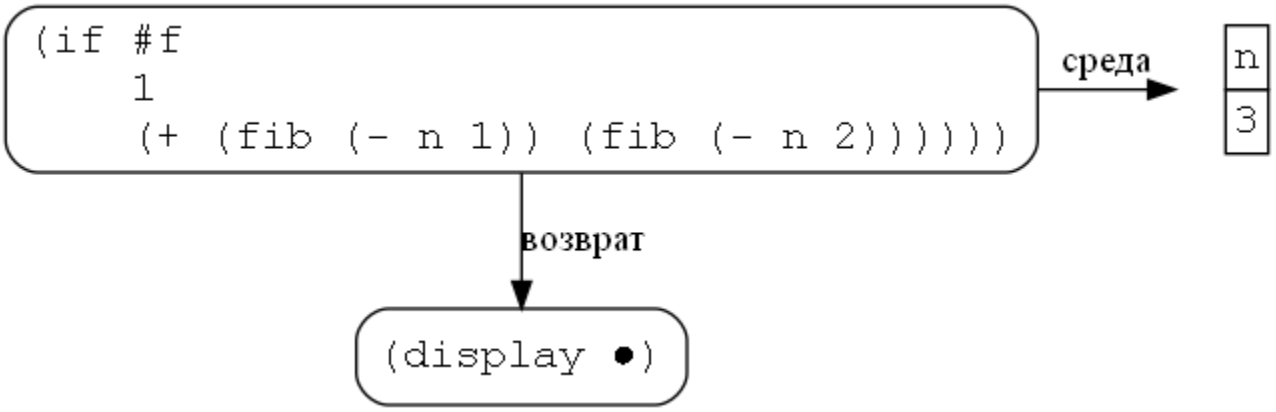
Подстановка значения переменной:



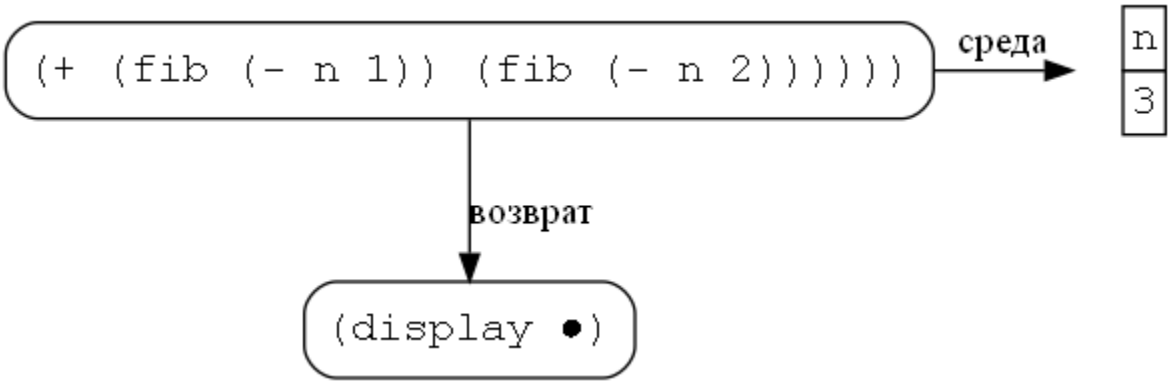
Вызов `=`:



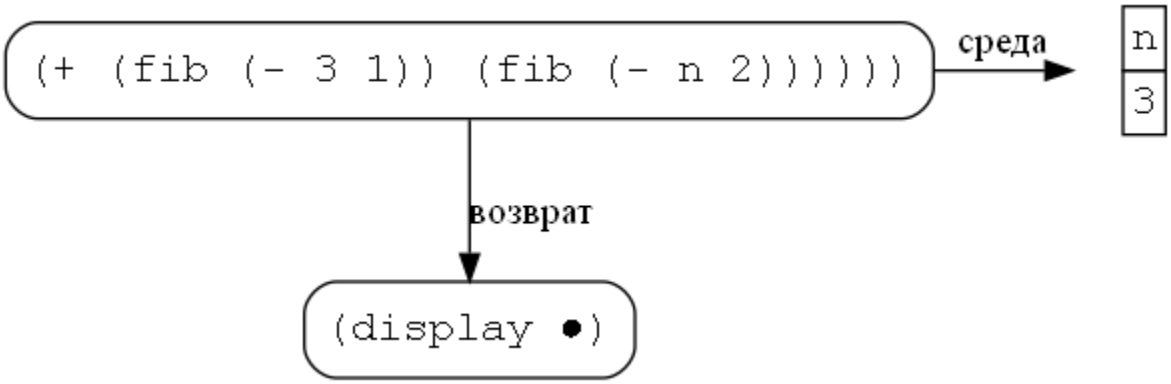
Фрейм стека для `=`:



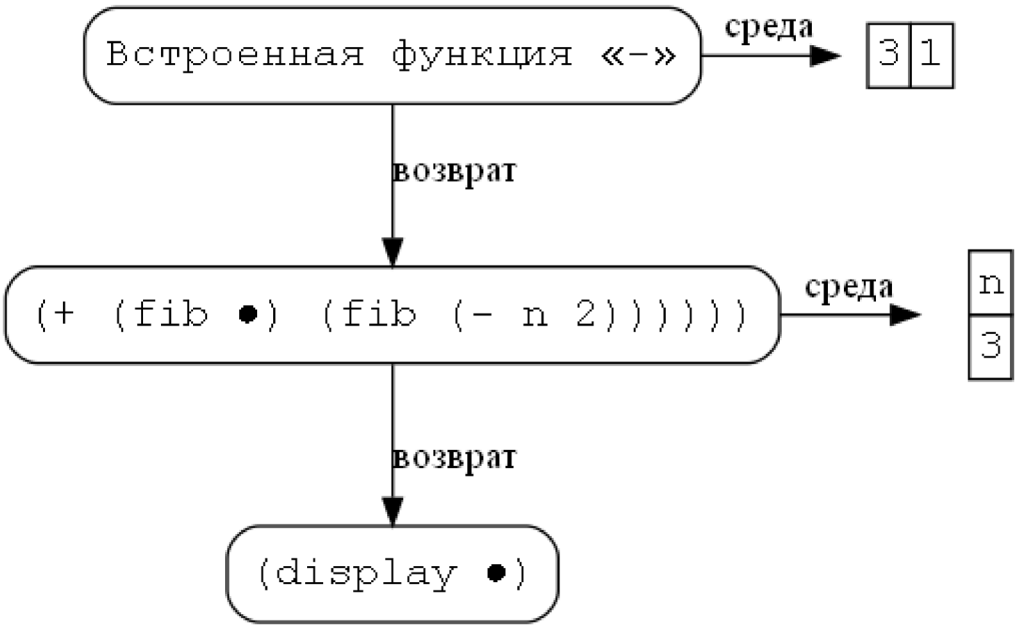
Возврат `#f` из `=` (опущен), редукция `if`:



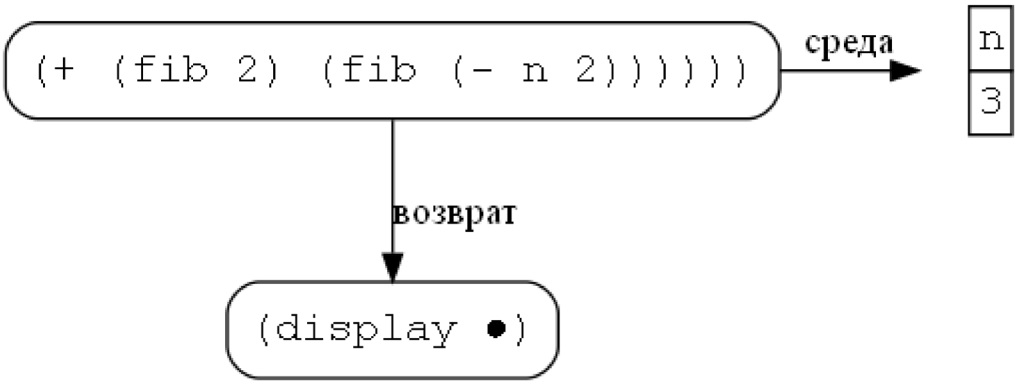
Подстановка значения переменной:



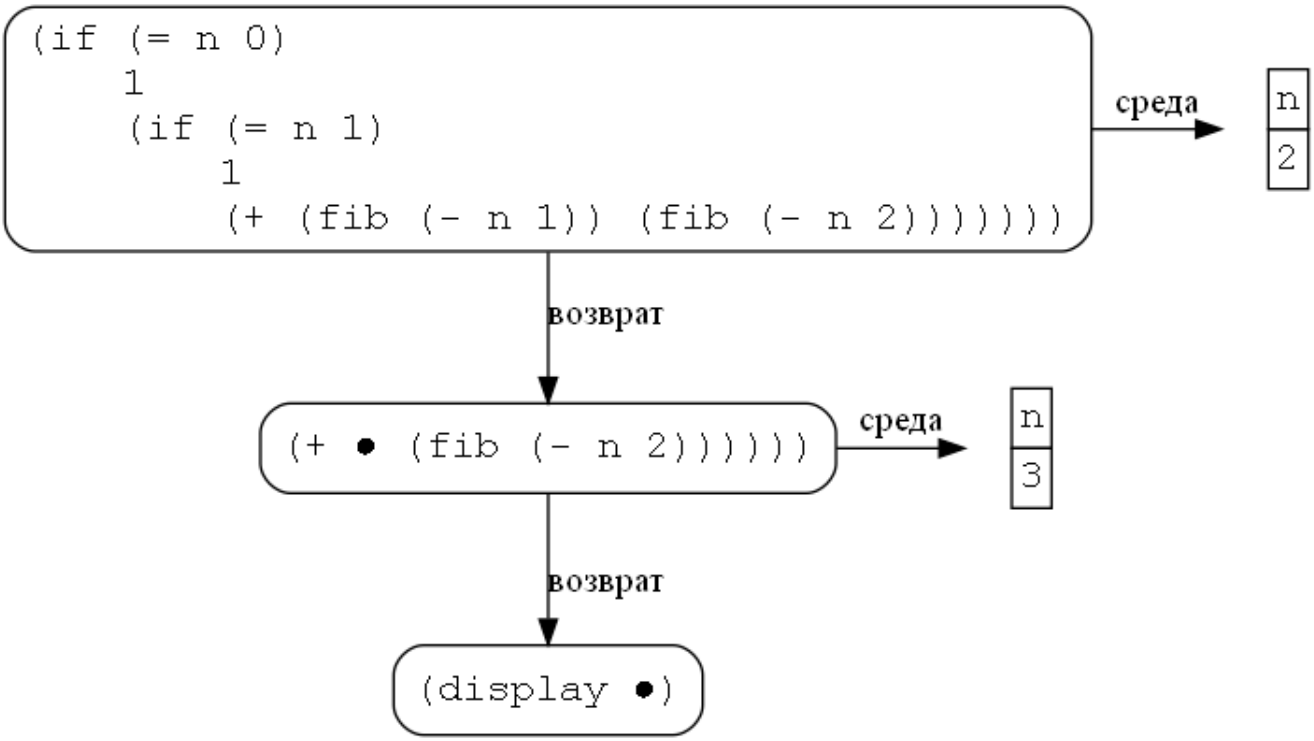
Аргументы у встроенной функции `-` вычислены, её вызов и фрейм стека:



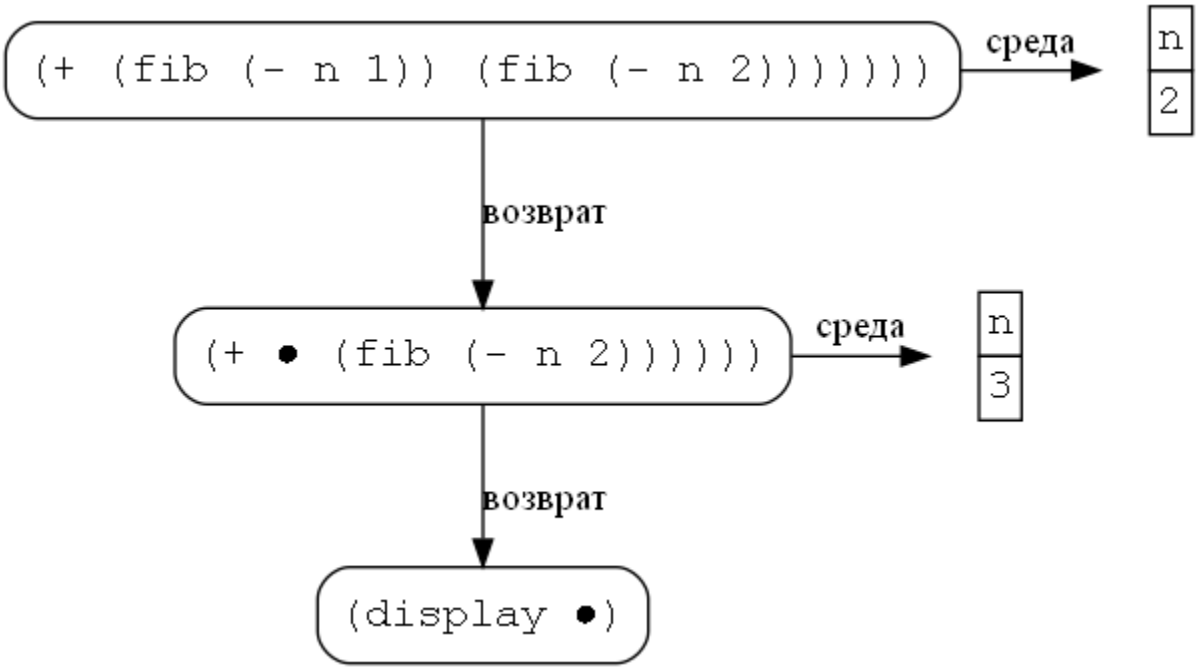
Возврат из функции `-`:



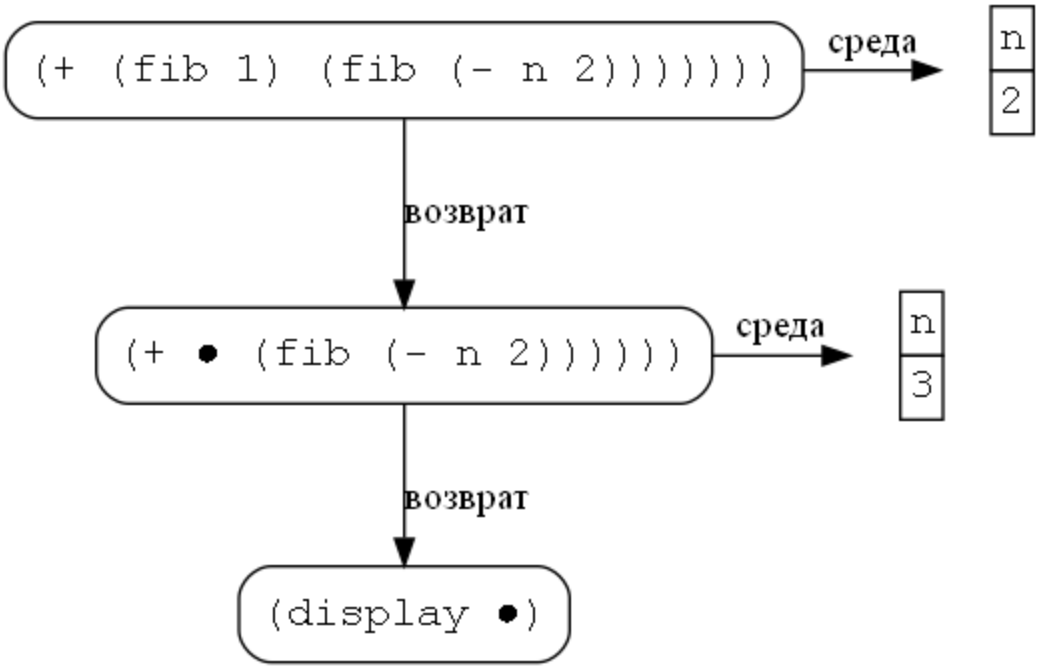
Рекурсивный вызов `fib`:



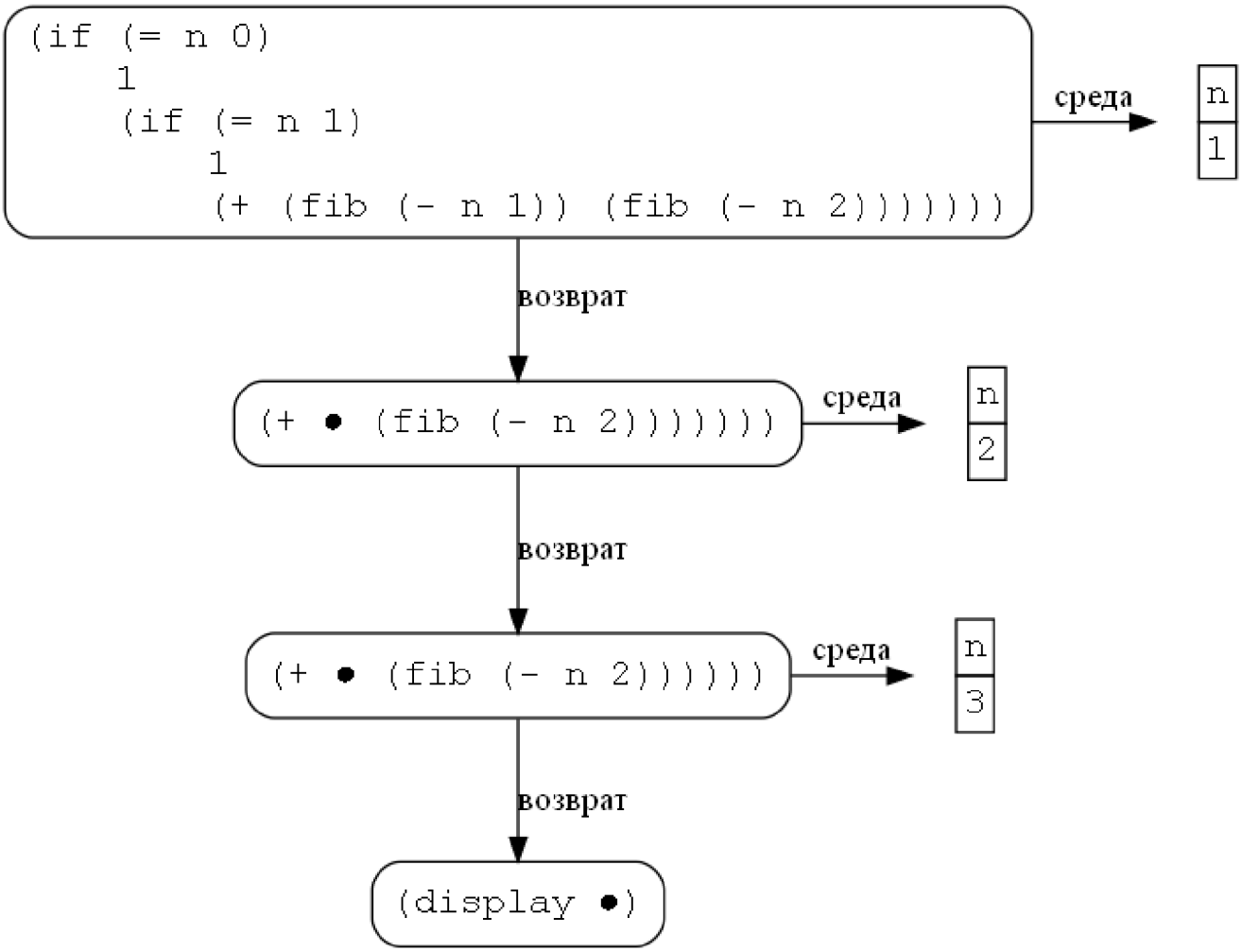
Очевидные вызовы `=` и шаги редукции `if` пропускаем:



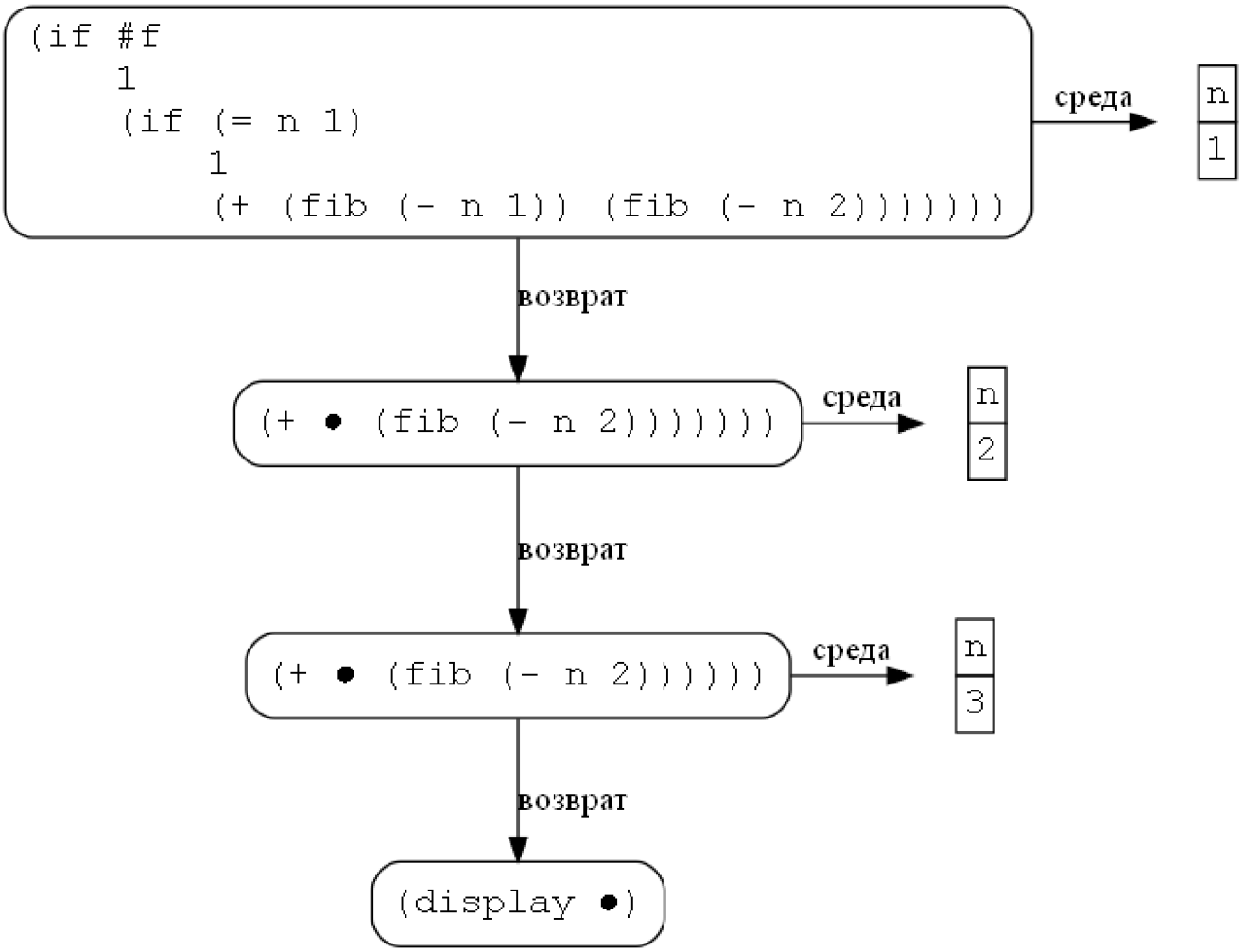
Вычисление `(- n 1)` → `(- 2 1)` → `1`:



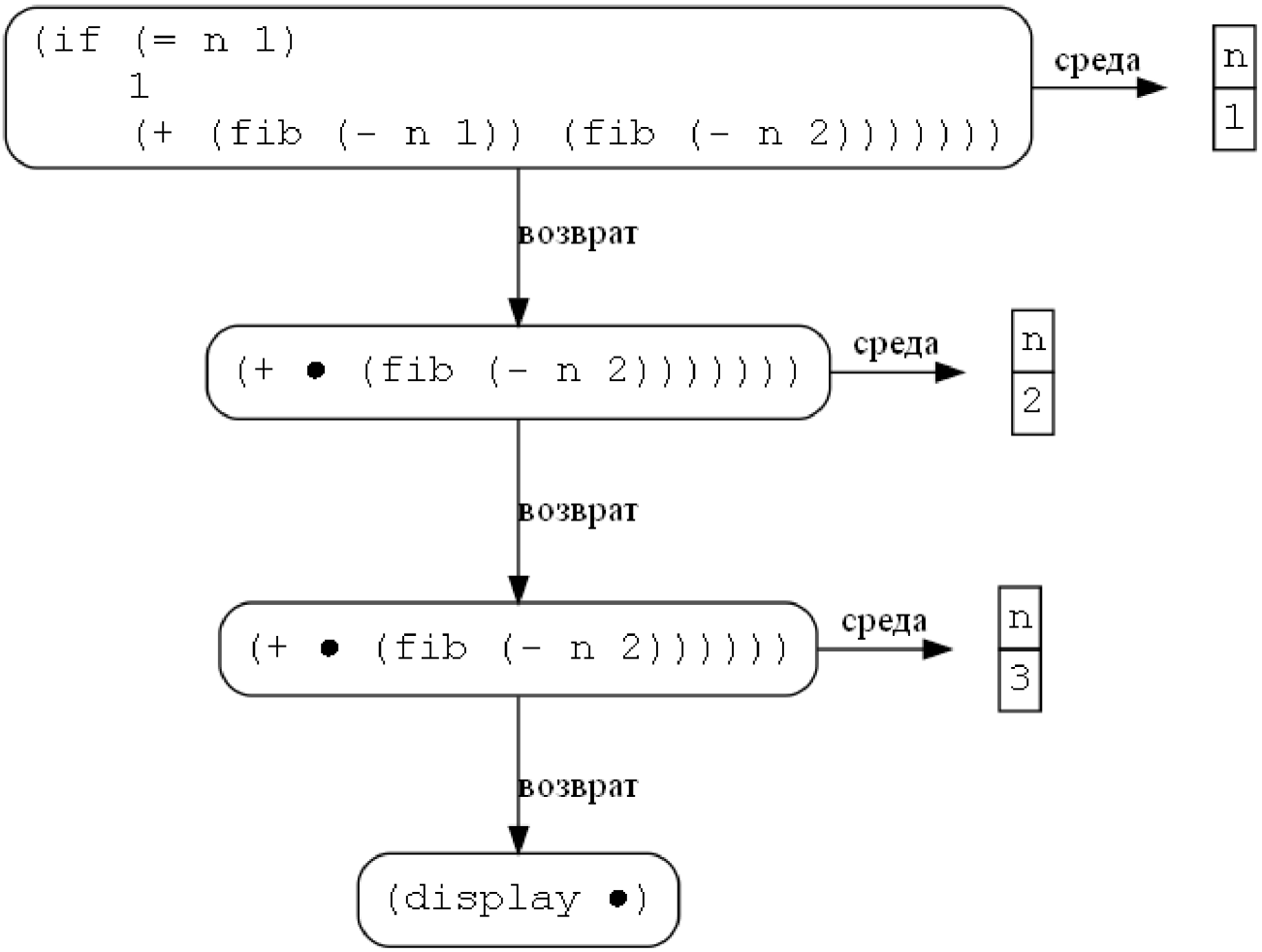
Рекурсивный вызов `fib`:



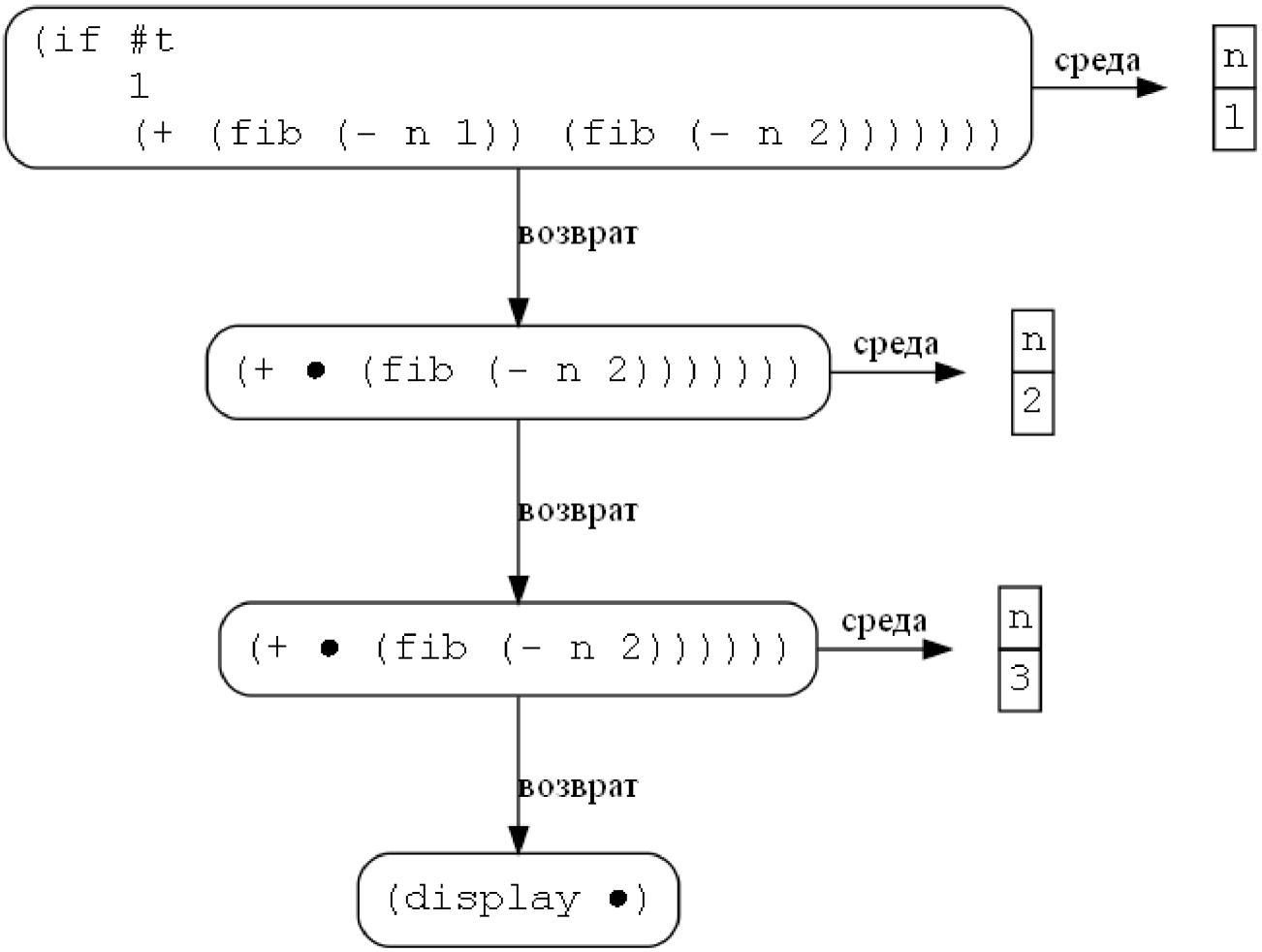
Вычисление `(= n 0)` → `(= 1 0)` → `#f`:


(if #f
1
(if (= n 1)
1
(+ (fib (- n 1)) (fib (- n 2)))))))
среда
n
1
возврат
(+ • (fib (- n 2)))))))
среда
n
2
возврат
(+ • (fib (- n 2))))))
среда
n
3
возврат
(display •)

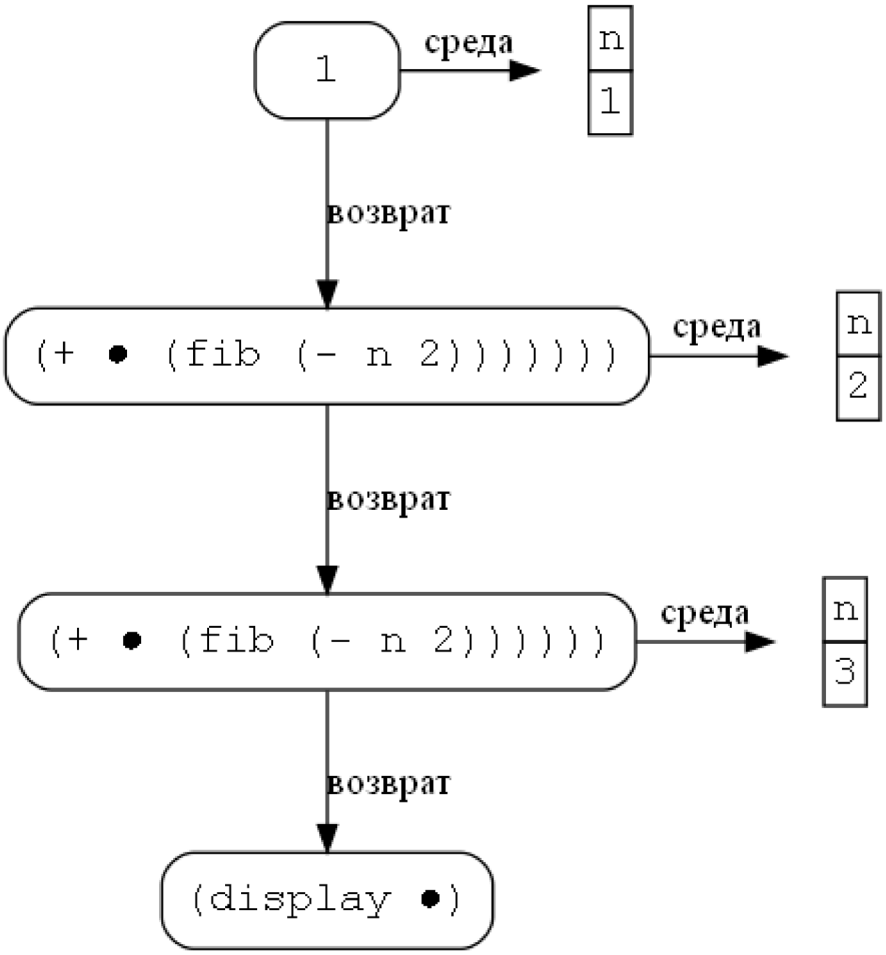
Редукция `if`:



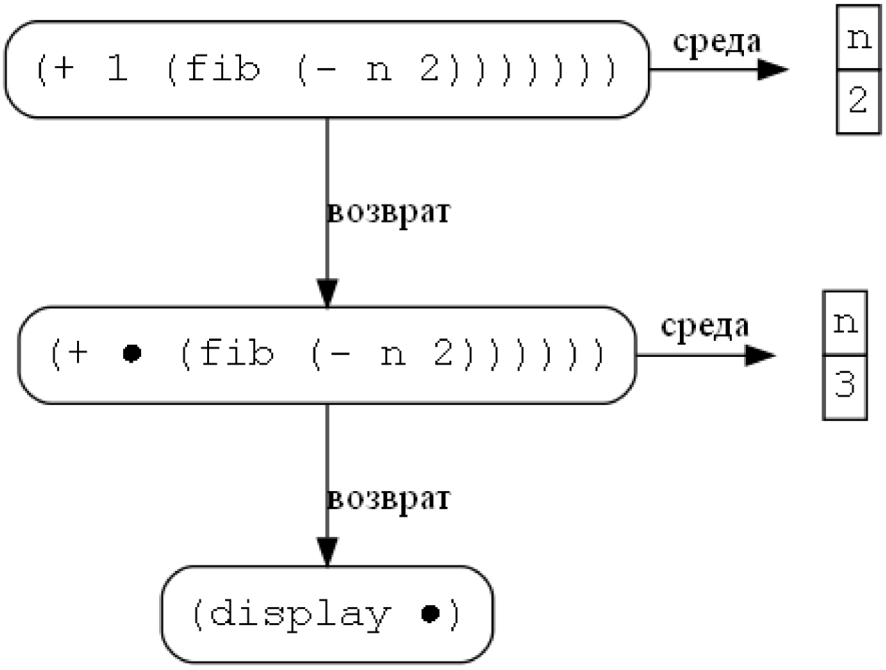
Вычисление `(= n 1)` → `(= 1 1)` → `#t`:


(if #t
1
(+ (fib (- n 1)) (fib (- n 2)))))))
среда
n
1
возврат
(+ • (fib (- n 2)))))))
среда
n
2
возврат
(+ • (fib (- n 2))))))
среда
n
3
возврат
(display •)

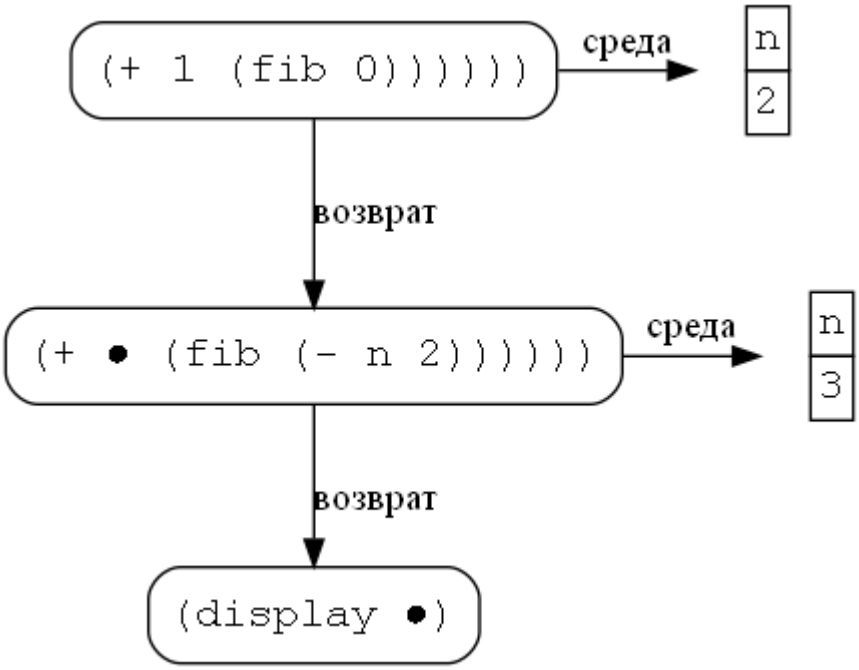
Редукция `if`:



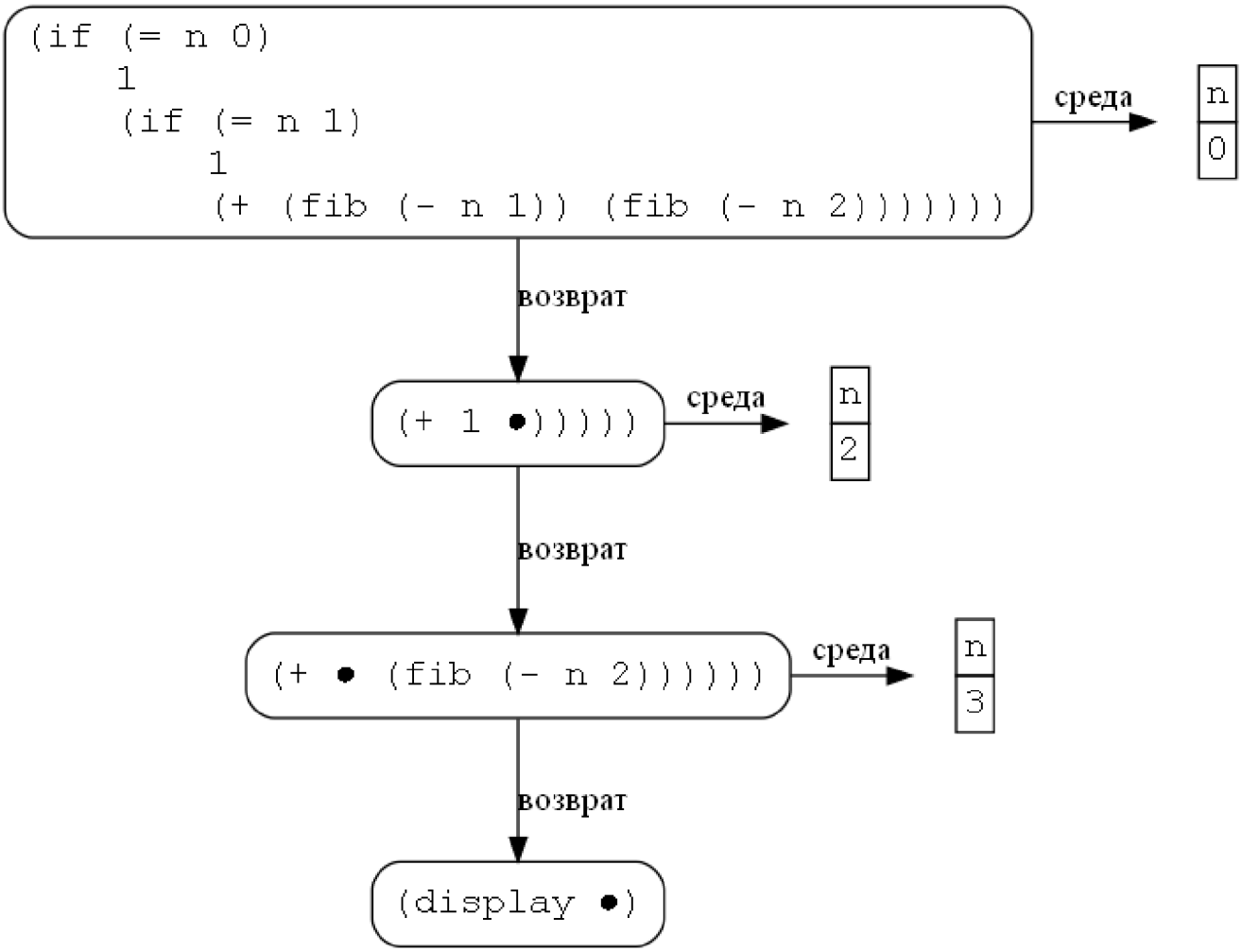
Возврат вычисленного значения:



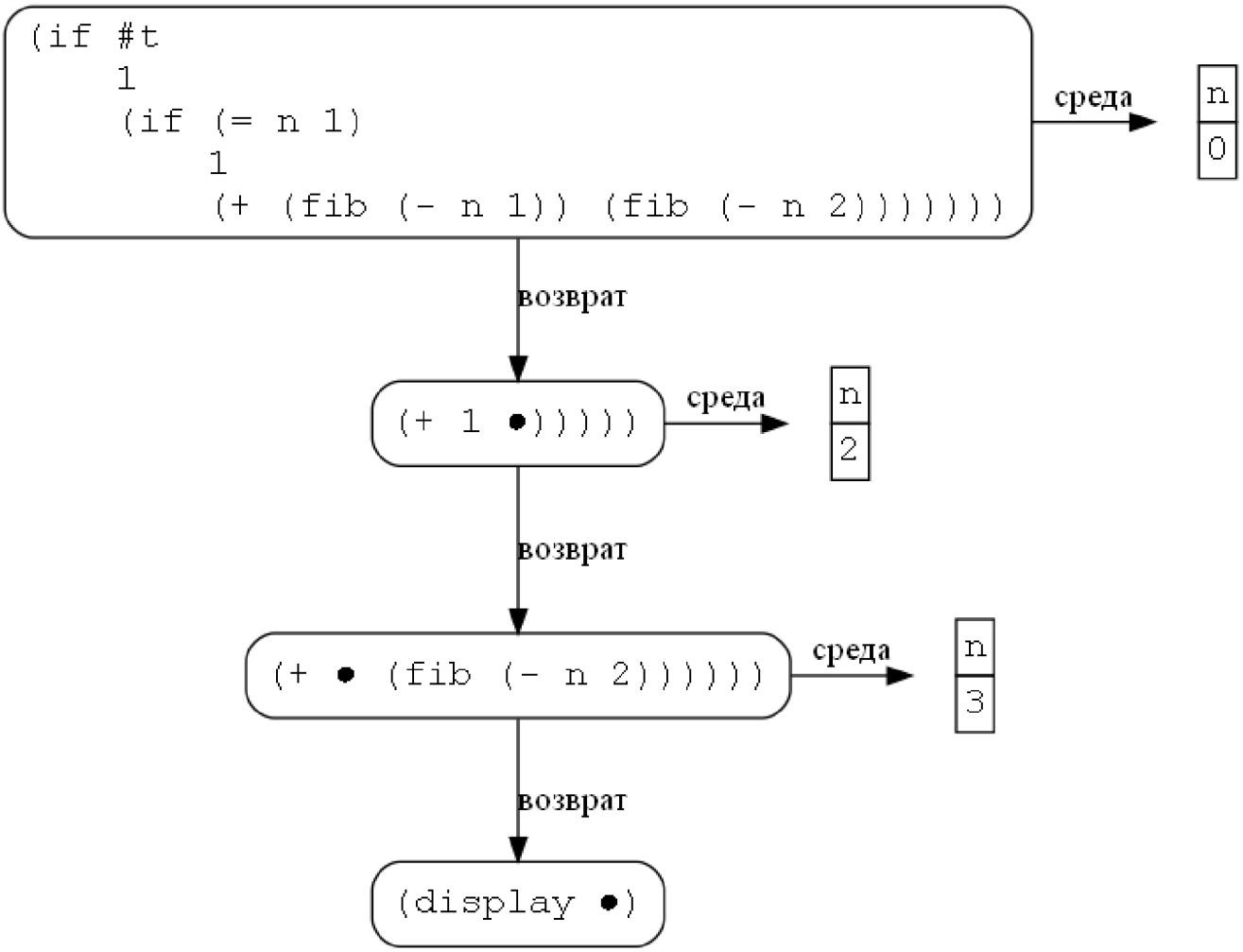
Вычисление `(- n 2)` → `(- 2 2)` → `0`:



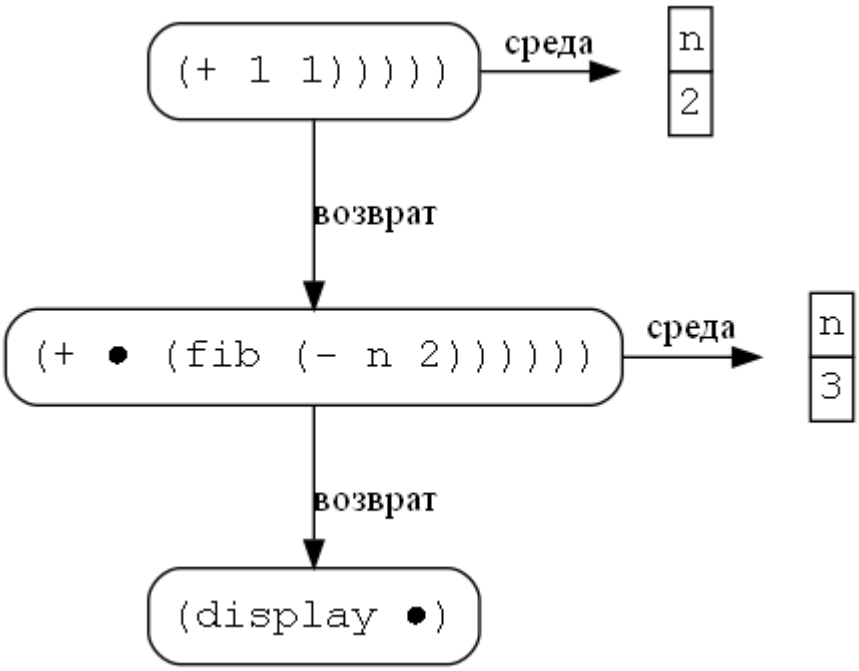
Рекурсивный вызов `fib`:



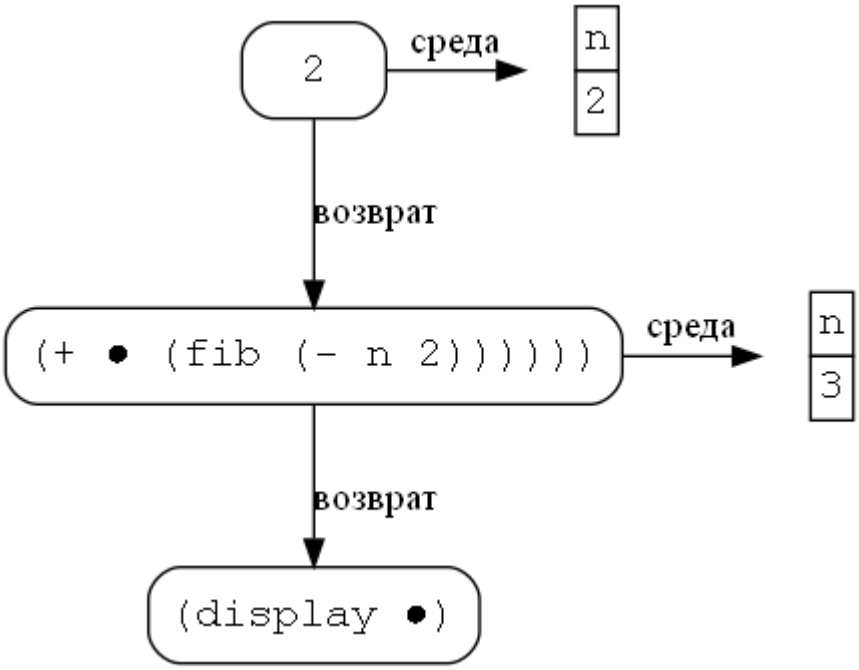
Вычисление `(= n 0)` → `(= 0 0)` → `#t`:



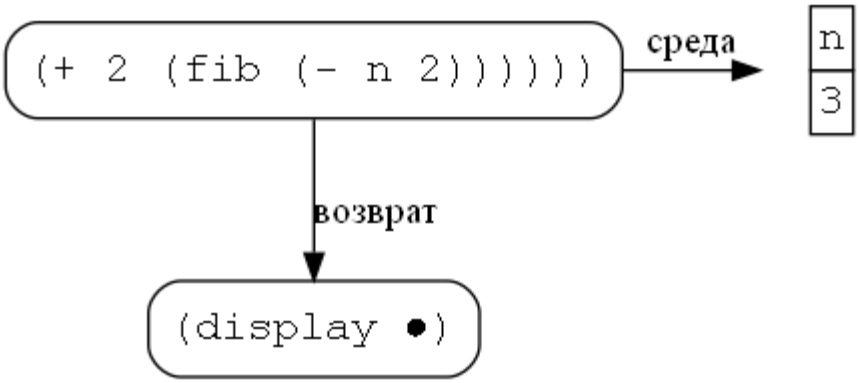
Редукция `#if`, возврат из функции:



Вычисление `(+ 1 1)` → `2`:



Возврат из функции:



Рисунки «на доске»

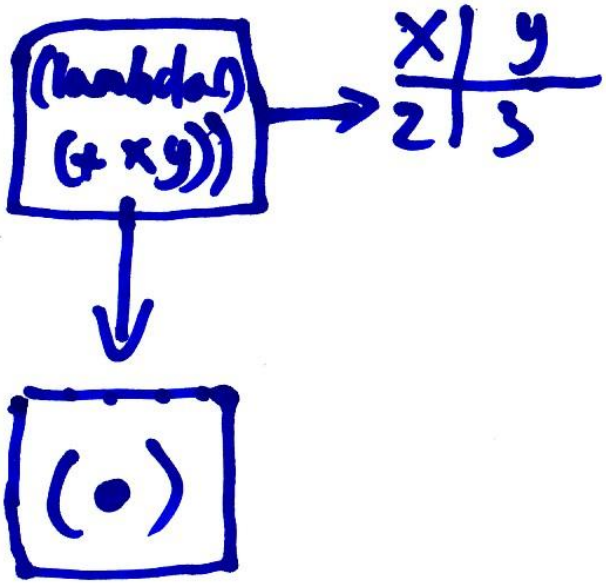
```
(* 2
  (call/cc
    (lambda (c)
      (begin
        (set! r c)
        (display "Hello\n")
```

Редукция begin:

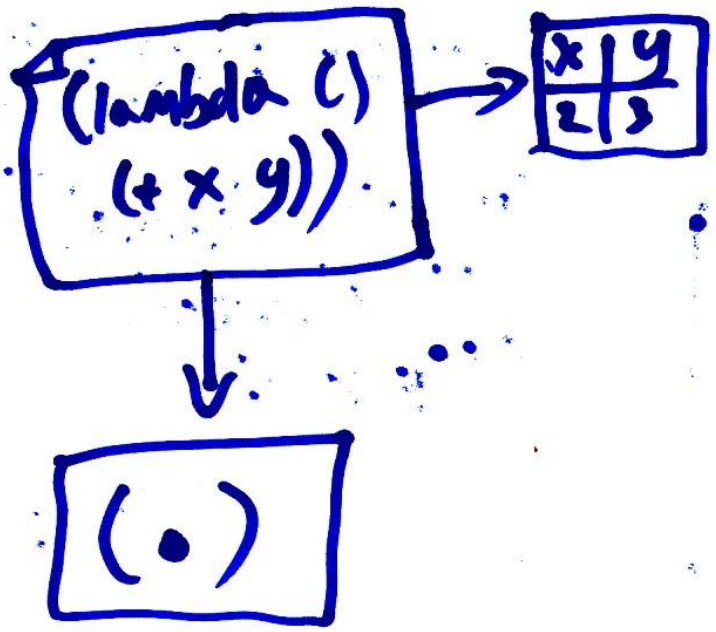
```
(begin a b c)
   ↓
(begin b c)
   ↓
   c
```

Замыкания

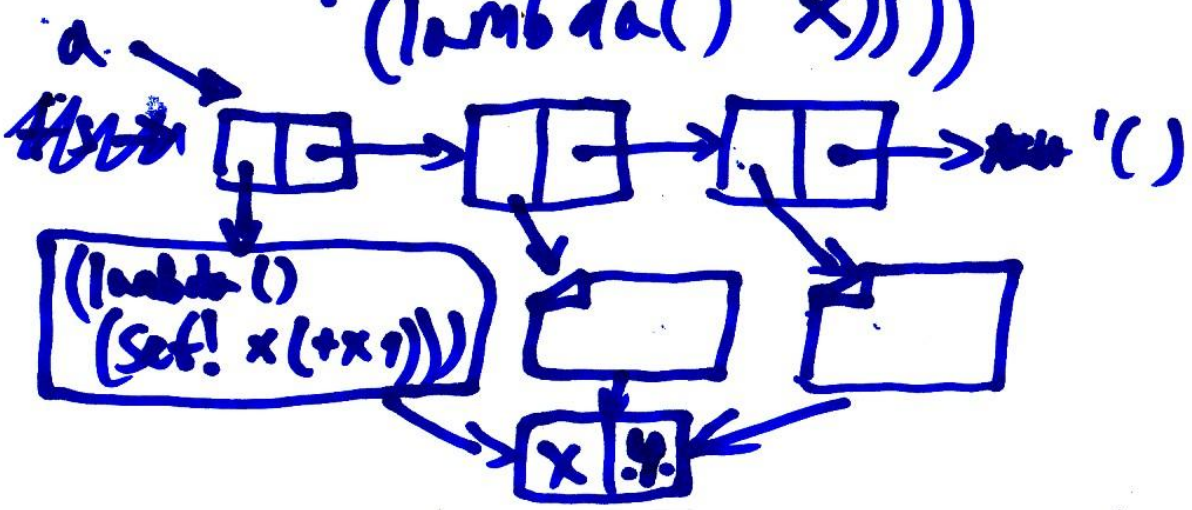
```
(((lambda (x y)
    (lambda ()
      (+ x y)))

 2 3))
```



Замыкания




```
(define fs
  (lambda (x)
    (list (lambda () (set! x (+ x 1)))
          (lambda () (set! x (* x 2)))
          (lambda () x))))
```



(define a (fs 4))

((caddr a)) → 4

((car a))

((caddr a)) → 5

((cadr a))

((caddr a)) → 10

#<void>

(if #f #f)

```
int x;
if(x){
  printf("A");
}
if(!x){
  printf("B");
}
```

Продолжение. ~~call-with-current-continuation~~

call-with-current-continuation

```
(define call/cc
   call-with-current-continuation)
```

Как вызывается:

(call/cc proc)

где proc - процедура.

proc вызывается с одним пара-

cont - продолжение, которое также можно вызвать как процедуру:

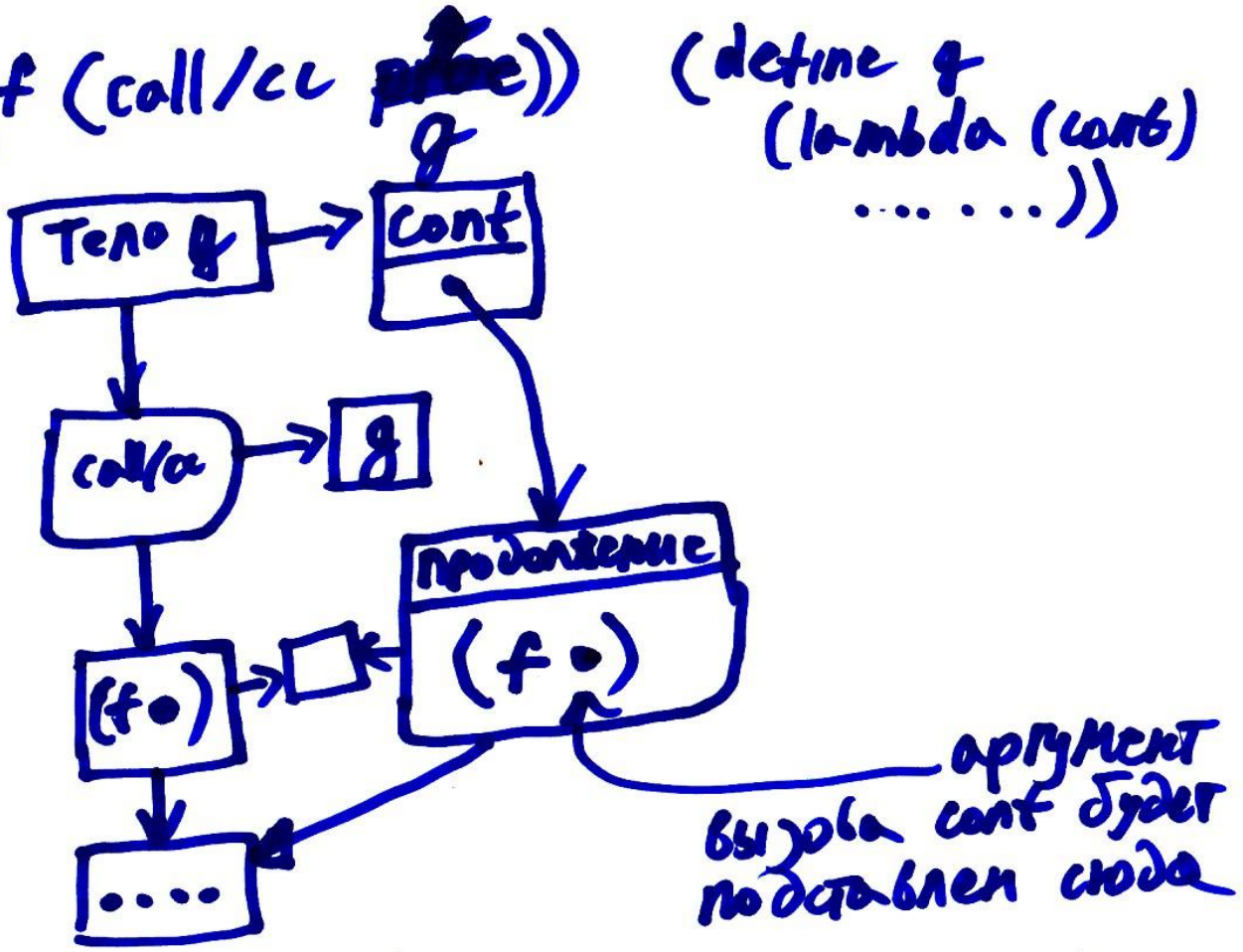
```
(call/cc
  (lambda (cont)
    ........
    .... (cont 5)
    ....
    ....))
```

Если proc завершилась обычным образом - результат (call/cc proc) тот же, что и у (proc cont)

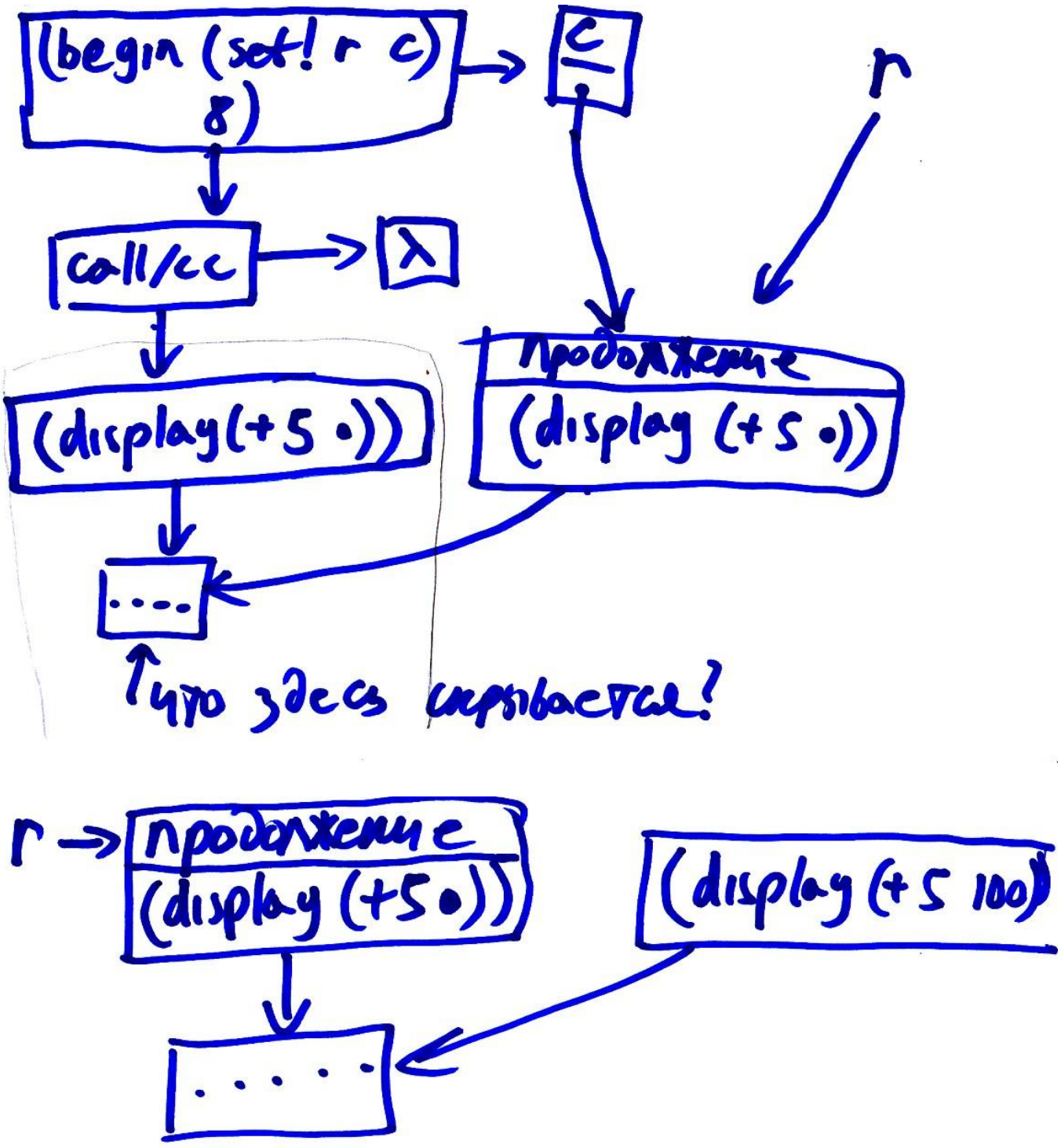
Другой случай - вызов cont (с одним параметром)
В этом случае восстанавливается фрейм стека на момент вызова call/cc.

(f (call/cc g))

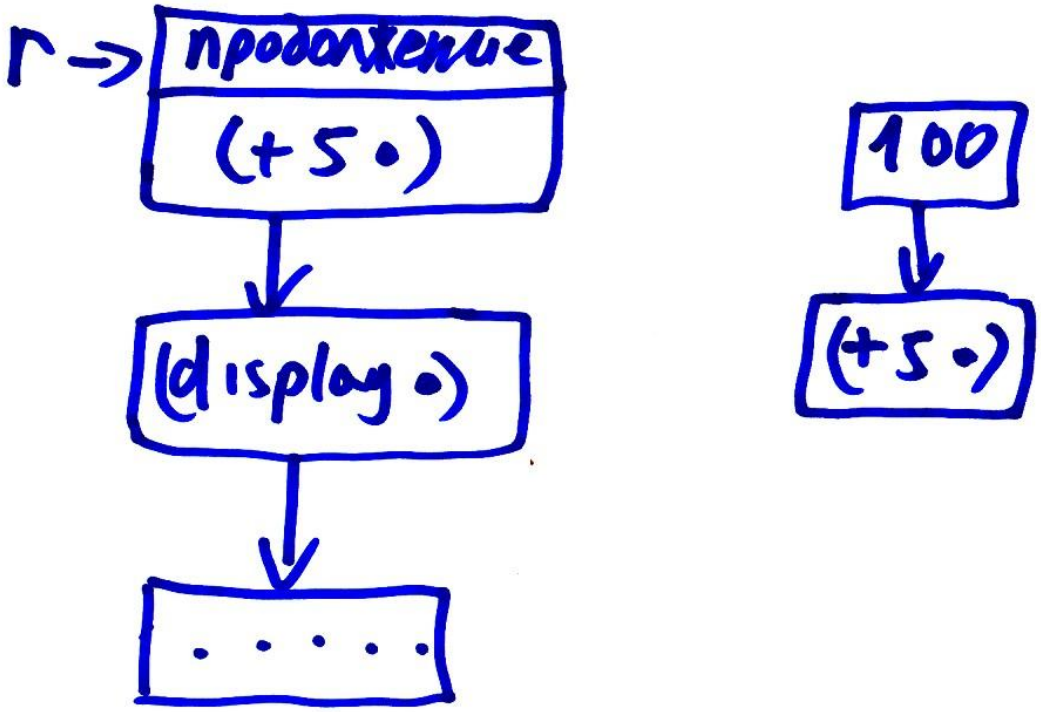
(define g
  (lambda (cont)
    ....))



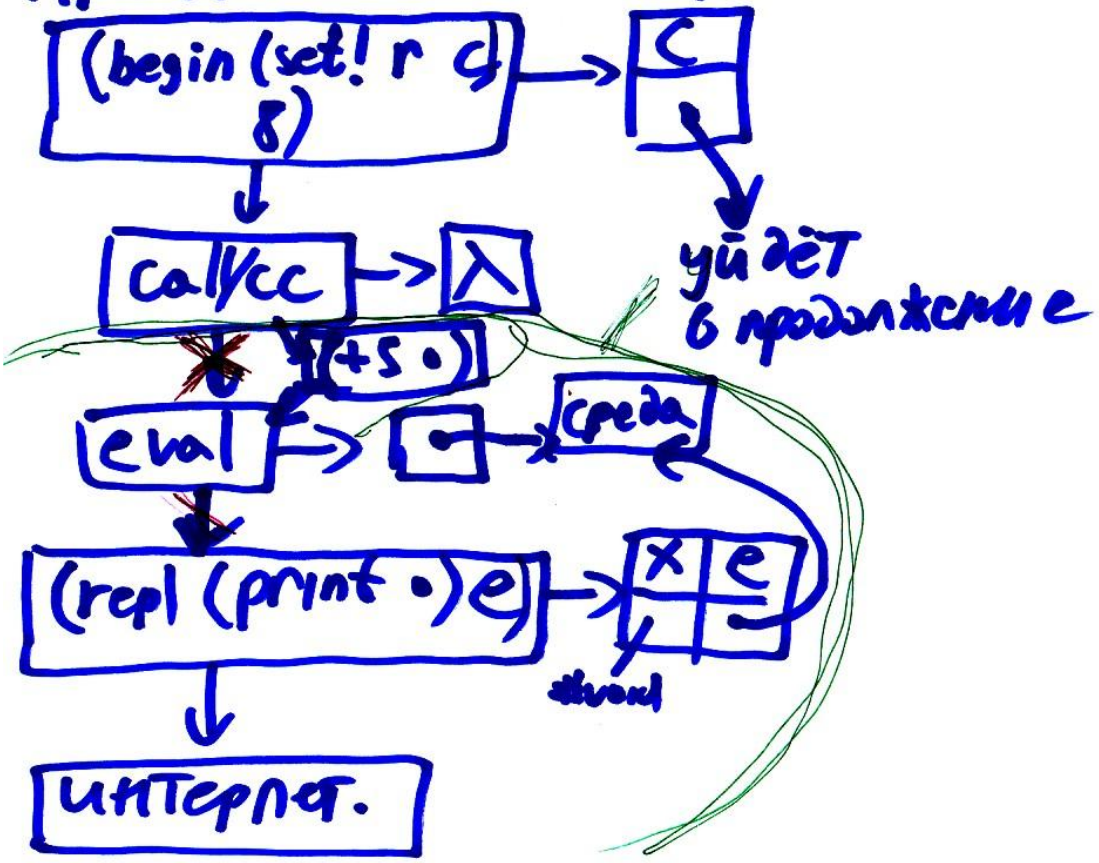

```
(define r #f)
(display (+ 5
            (call/cc
              (lambda (c)
                (set! r c)
                8))))        → 13

(r 100) → 105
```


(begin (set! r c)
8)
c
r
call/cc
λ
(display (+ 5 •))
продолжение
(display (+ 5 •))
....
Что здесь сохраняется?

r
продолжение
(display (+ 5 •))
(display (+ 5 100))
. . . . .



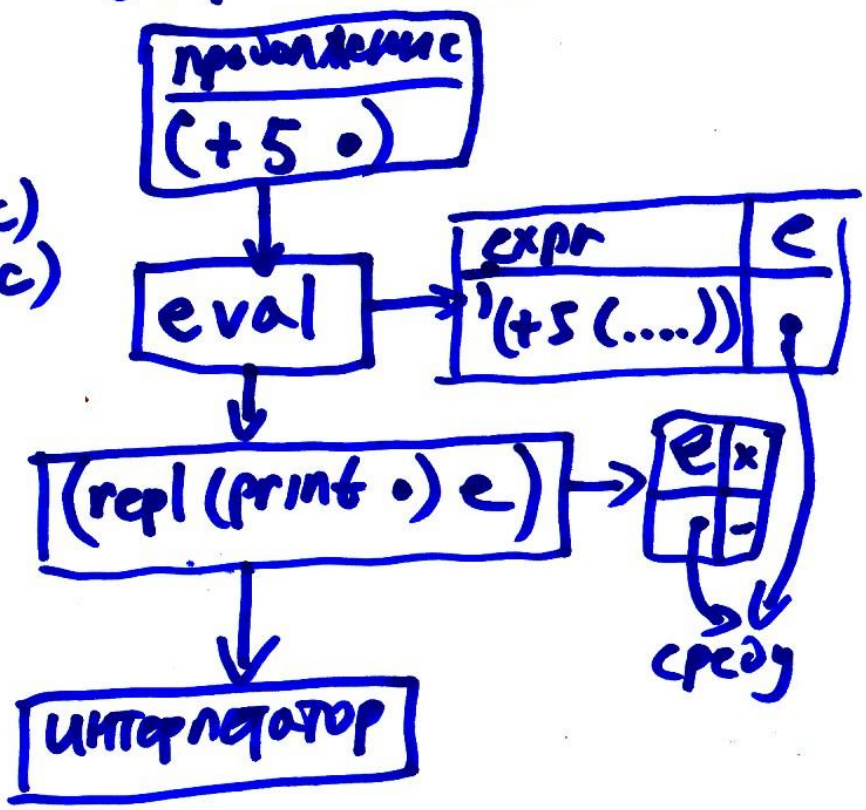
При вычислении второго выражения:



Входной файл

```
(define r #f)
(+ 5 (call/cc
       (lambda (c)
         (set! r c)
         8)))
(display 'Hello)
(newline)
(r 100)
```

В переменной r:




```
(define repl
  (lambda (x e)
    (repl (print (eval (read)
                       e))
          e)))
```

```
(*2 (begin
     (display "Hello\n")
     (call/cc
       (lambda (c)
         (set! r c)))
     (display "Hi\n")
     3))
```

Hello
Hi
→ 6

```
(r #f) →   Hi
           6
```

call/cc запомнил позицию вычислений после того, как Hello напечаталось

```
(*2 (begin
     (display "Hello\n")
     (call/cc ...)
     (display "Hi\n")
     3))
```

↓ напечатается Hello

```
(*2 (begin
     (call/cc ...)
     (display "Hi\n")
     3))
```