Implementing continuations:
ANF and CPS conversion

CS 245 — Spring 2019

Logistics

- I am gone after this afternoon (until weekend)
- No labs tomorrow
- Exam topics:
    - https://docs.google.com/document/d/1yxTsObP87Ssp_LQgBjfaZgK8zf69JGdgSrIuE0Gj99w/
- Project 5 is last project in the class: distributed ~Thursday-Friday, due last day of class
- Tentatively: nothing in finals period.

```
(let ([cc (call/cc (lambda (k) k))])
   ...)
```

A common idiom for `call/cc` is to let-bind the current continuation.

```
(let ([cc (call/cc (lambda (k) k))])
  ...)
```

Note that applying call/cc on the identity function is exactly the same as applying it on the u-combinator!

```
(let ([cc (call/cc (lambda (k) (k k)))])
  ...)
```

Why is this the case?

`call/cc` makes a tail call to `(lambda (k) …)`, so the body of the function is the same return point as the captured continuation k!

```
(let ([cc (call/cc (lambda (k) k))])
  ...)
```

This return point …is the same as this one…

```
(let ([cc (call/cc (lambda (k) (k k)))])
  ...)
```

…and calling k on itself, returns k to itself!

Returning value v is the same as *calling* that saved return point *on* v.

```
(let ([cc (call/cc (lambda (k) k))])

   ;; loop body goes here

   (if (jump-to-top?)
       (cc cc)
       return-value))
```

Continuations can be used to jump back to a previous point.

Just as we could have invoked `call/cc` on the u-combinator, to jump back to the let-binding of `cc`, returning `cc`, we call `(cc cc)`.

```
(define (fun x)

  (let ([y (if (p? x)
               …
               …)])

    (g x y)))
```

A simple use of continuations is to implement a ***preemptive return***.

What if we wanted to return from `fun` within the right-hand-side of the `let` form?

Binds the return-point of the current call to `fun` to a continuation `return`.

```
(define (fun x)
  (call/cc (lambda (return)

    (let ([y (if (p? x)
                 …
                 (return x))])

      (g x y)))))
```

Uses the continuation `return` to jump back to the return point of `fun` and yield value `x` instead of binding `y` and calling `g`.

Try an example. What do each of these 3 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 3)))))))
```

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 (k1 3))))))))
```

```
(call/cc (lambda (k0)
           (+ 1
              (call/cc (lambda (k1)
                         (+ 1 (k1 3))))
              (k0 1))))
```

Stack-passing (CEK) semantics

(implementing first-class continuations)

C — Control-expression

Term-rewriting / textual reduction

Context and redex for deterministic eval

CE — Control & Env machine

Big-step, explicit closure creation

CES — Store-passing machine

Passes addr->value map in evaluation order

CEK — Stack-passing machine

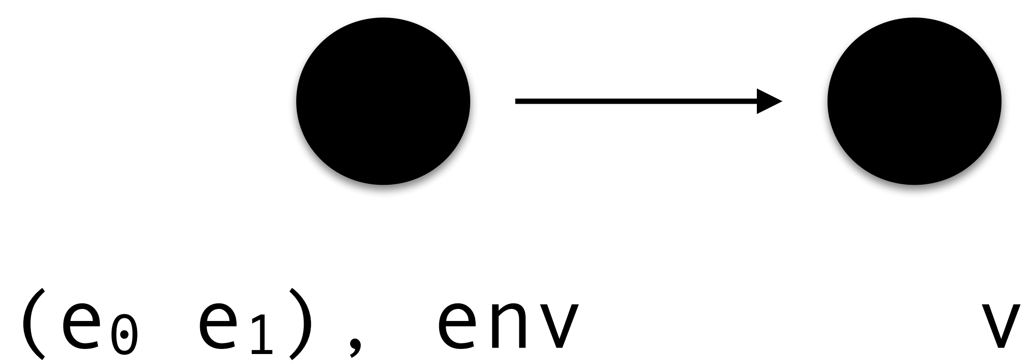
Passes a list of stack frames, small-step

$$\frac{(\texttt{e}_0, \mathrm{env}) \Downarrow (\texttt{(}\lambda\ \texttt{(x)}\ \texttt{e}_2\texttt{)}, \mathrm{env}') \qquad (\texttt{e}_1, \mathrm{env}) \Downarrow v_1 \qquad (\texttt{e}_2, \mathrm{env}'[x \mapsto v_1]) \Downarrow v_2}{(\texttt{(e}_0\ \texttt{e}_1\texttt{)}, \mathrm{env}) \Downarrow v_2}$$

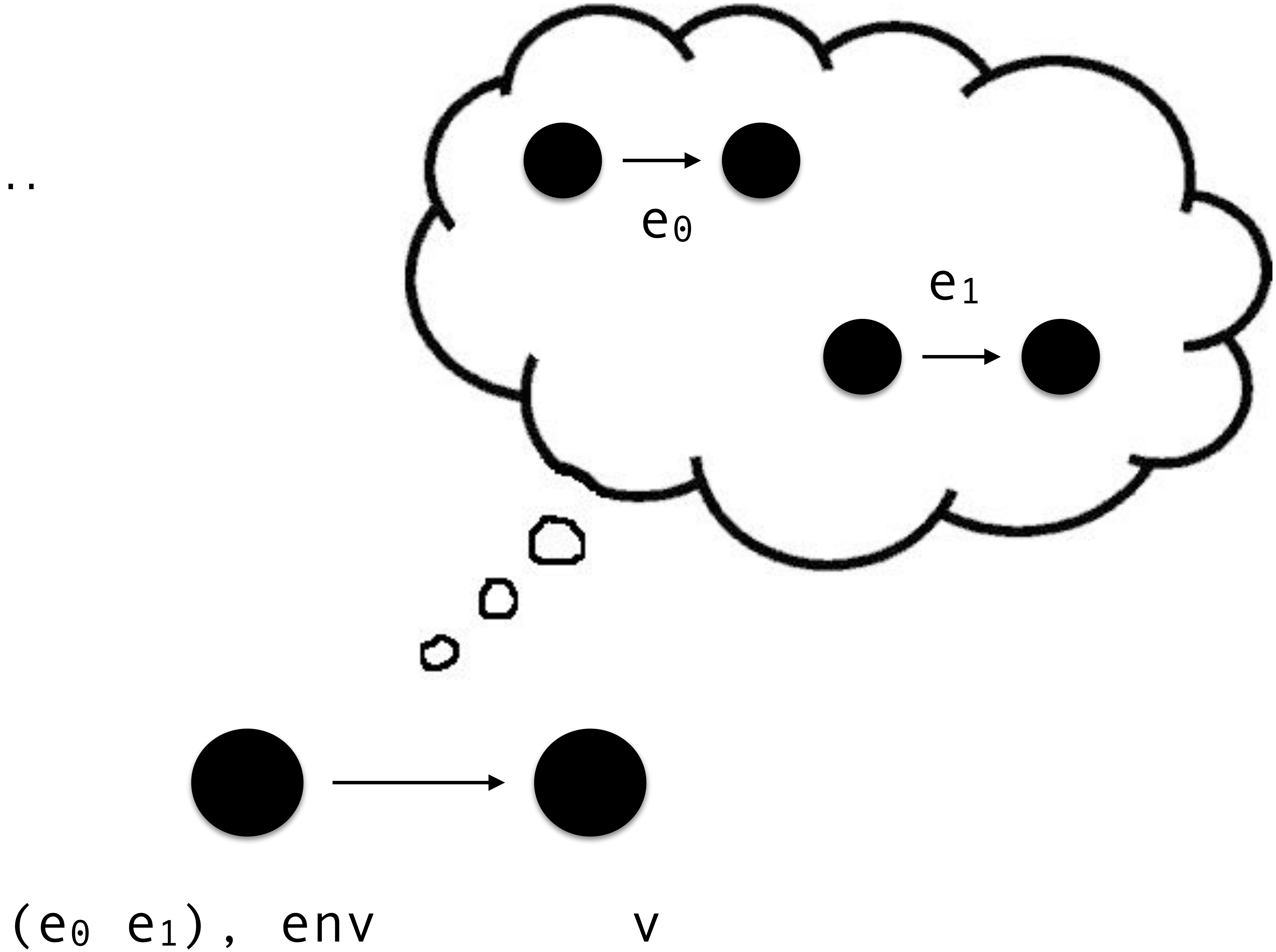
$$\frac{}{(\texttt{(}\lambda\ \texttt{(x)}\ \texttt{e)}, \mathrm{env}) \Downarrow (\texttt{(}\lambda\ \texttt{(x)}\ \texttt{e)}, \mathrm{env})}$$

$$\frac{}{(x, \mathrm{env}) \Downarrow \mathrm{env}(x)}$$

Previously…



Previously…



```
(define (interp e env)
  (match e
        [(? symbol? x)
         (hash-ref env x)]

        [`(λ (,x) ,e0)
         `(clo (λ (,x) ,e0) ,env)]

        [`(,e0 ,e1)
         (define v0 (interp e0 env))
         (define v1 (interp e1 env))
         (match v0
           [`(clo (λ (,x) ,e2) ,env)
            (interp e2 (hash-set env x v1))])]))
```

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= **halt** | **ar**(e, env, k)
| **fn**(v, k)

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= **halt** | **ar**(e, env, k)
| **fn**(v, k)

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

$$\mathcal{E} ::= (\mathcal{E}\ e) \mid (v\ \mathcal{E}) \mid \square$$

$$((\texttt{e}_0\ \texttt{e}_1), \mathrm{env}, k) \rightarrow (\texttt{e}_0, \mathrm{env}, \mathbf{ar}(\texttt{e}_1, \mathrm{env}, k))$$

$$(\texttt{x}, \mathrm{env}, \mathbf{ar}(\texttt{e}_1, \mathrm{env}_1, k_1)) \rightarrow (\texttt{e}_1, \mathrm{env}_1, \mathbf{fn}(\mathrm{env}(\texttt{x}), k_1))$$

$$((\lambda\ (\texttt{x})\ \texttt{e}), \mathrm{env}, \mathbf{ar}(\texttt{e}_1, \mathrm{env}_1, k_1)) \rightarrow (\texttt{e}_1, \mathrm{env}_1, \mathbf{fn}(((\lambda\ (\texttt{x})\ \texttt{e}), \mathrm{env}), k_1))$$

$$(\texttt{x}, \mathrm{env}, \mathbf{fn}(((\lambda\ (\texttt{x}_1)\ \texttt{e}_1), \mathrm{env}_1), k_1)) \rightarrow (\texttt{e}_1, \mathrm{env}_1[\texttt{x}_1 \mapsto \mathrm{env}(\texttt{x})], k_1)$$

$$((\lambda\ (\texttt{x})\ \texttt{e}), \mathrm{env}, \mathbf{fn}(((\lambda\ (\texttt{x}_1)\ \texttt{e}_1), \mathrm{env}_1), k_1)) \rightarrow (\texttt{e}_1, \mathrm{env}_1[\texttt{x}_1 \mapsto ((\lambda\ (\texttt{x})\ \texttt{e}), \mathrm{env})], k_1)$$

call/cc semantics

$$((\texttt{call/cc (λ (x) } e_0\texttt{))}, \text{env}, k) \rightarrow (e_0, \text{env}[x \mapsto k], k)$$

$$((\texttt{(λ (x) } e_0\texttt{)}, \text{env}, \mathbf{fn}(k_0, k_1)) \rightarrow ((\texttt{(λ (x) } e_0\texttt{)}, \text{env}, k_0)$$

$$(x, \text{env}, \mathbf{fn}(k_0, k_1)) \rightarrow (x, \text{env}, k_0)$$

$$e ::= \ldots \mid (\texttt{let}\ ([x\ e_0])\ e_1)$$

$$k ::= \ldots \mid \textbf{let}(x, e, env, k)$$

$$(x, env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$((\lambda\ (x)\ e), env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto ((\lambda\ (x)\ e), env)], k_1)$$

e ::= ...

| (let ([x e_0]) e_1)

$(x, env, \textbf{fn}((\texttt{(λ (}x_1\texttt{)}\ e_1\texttt{)}, env_1), k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$

$(\texttt{(λ (x) e)}, env, \textbf{fn}((\texttt{(λ (}x_1\texttt{)}\ e_1\texttt{)}, env_1), k_1))$
$\rightarrow (e_1, env_1[x_1 \mapsto (\texttt{(λ (x) e)}, env)], k_1)$

k ::= ... | $\textbf{let}(x, e, env, k)$

These are nearly identical because a let form is
just an immediate application of a lambda!

$(x, env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$

$(\texttt{(λ (x) e)}, env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto (\texttt{(λ (x) e)}, env)], k_1)$

CEK-machine evaluation

$(e_0, [], ()) \to \ldots$

$\to \ldots$

$\to \ldots$

$\to \ldots$

$\to (x, env, \textbf{halt}) \to env(x)$

consider the following question.

Is it possible to take an arbitrary Racket/Scheme program and transform it systematically so that no function ever returns?

ANF conversion

Conversion to ***administrative normal form (ANF)*** means rewriting the language so that the only continuation is a let-continuation! Subexpressions must be let-bound to a temp. variable.

```
((f x) (g y))   ⟶   (let ([fun (f x)])
                       (let ([arg (g y)])
                         (fun arg)))
```

CPS conversion

Conversion to ***Continuation-passing style (CPS)*** means encoding continuations explicitly as first-class functions (which makes it easy to compile away call/cc!) and passing them forward at each call site (just as our CEK interpreter passed forward K). Every call becomes a tail call, and return points become calls to continuations.

```
(let ([fun (f x)])
  (let ([arg (g y)])
    (fun arg)))
```

⟶

```
(f (lambda (_ fun)
     (g (lambda (_ arg)
          (fun k arg))
        y))
   x)
```

CPS conversion

Assume the current continuation is bound to a variable k.
In this case, a let-form *extends* the current continuation by defining a new lambda that saves k (the tail of the stack) in its environment:
(Note that continuations are passed a continuation, but ignore it.)

```
(let ([fun (f x)])
  (fun y))
```

⟶

```
(f (lambda (_ fun)
          (fun k y))
   x)
```

and a returned value (e.g., a lambda) is instead passed to the current continuation **(Note: when applying a kont the first parameter is ignored)**:

```
(lambda (x) x)
```

⟶

```
(k k (lambda (x) x))
```

Visualizing CPS (example)

IR

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

```
(fib 4)
```

IR

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

```
(fib 3)
```

```
 fn {+} (fib (- n 2)) [n = 4]
```

IR

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

```
(fib 2)
```

fn {+} (fib (- n 2)) [n = 3]
fn {+} (fib (- n 2)) [n = 4]

IR

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

```
(fib 1)
```

`fn {+} (fib (- n 2)) [n = 2]`
`fn {+} (fib (- n 2)) [n = 3]`
`fn {+} (fib (- n 2)) [n = 4]`

IR

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

```
{1}
```

fn {+} (fib (- n 2)) [n = 2]
fn {+} (fib (- n 2)) [n = 3]
fn {+} (fib (- n 2)) [n = 4]

IR

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

```
(fib 0)
```

fn {+} {1}
fn {+} (fib (- n 2)) [n = 3]
fn {+} (fib (- n 2)) [n = 4]

IR

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

0

fn {+} {1}
fn {+} (fib (- n 2)) [n = 3]
fn {+} (fib (- n 2)) [n = 4]

IR

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

1

fn {+} (fib (- n 2)) [n = 3]
fn {+} (fib (- n 2)) [n = 4]

IR

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

```
(fib 1)
```

`fn {+} {1}`
`fn {+} (fib (- n 2)) [n = 4]`

IR

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

```
1
```

`fn {+} {1}`
`fn {+} (fib (- n 2)) [n = 4]`

IR

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

```
2
 fn {+} (fib (- n 2)) [n = 4]
```

IR

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

```
(fib 2)
```

```
fn {+} {2}
```

IR

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

```
(fib 1)
```

```
fn {+} (fib (- n 2)) [n = 2]
```

```
fn {+} {2}
```

IR

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

```
1
```

```
fn {+} (fib (- n 2)) [n = 2]
```

```
fn {+} {2}
```

IR

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

```
(fib 0)
```

```
fn {+} {1}
```

```
fn {+} {2}
```

IR

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

```
0
fn {+} {1}
fn {+} {2}
3
```

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                4(let ([s (+ v0 v1)])
                  5s)))))))))
```

```
(fib 4)
```

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                4(let ([s (+ v0 v1)])
                  5s))))))))
```

```
(fib 3)
```

```
letk v0 e2 [n=4,n-1=3,…]
```

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                4(let ([s (+ v0 v1)])
                  5s))))))))
```

```
(fib 2)
```

letk v0 e_2 [n=3,n-1=2,…]
letk v0 e_2 [n=4,n-1=3,…]

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                4(let ([s (+ v0 v1)])
                  5s))))))))
```

```
(fib 1) -> 1
```

letk v0 e_2 [n=2,n-1=1,…]
letk v0 e_2 [n=3,n-1=2,…]
letk v0 e_2 [n=4,n-1=3,…]

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                4(let ([s (+ v0 v1)])
                  5s)))))))
```

```
(fib 0) -> 0
```

letk v1 e4 [v0=1,n=2,n-1=1,…]
letk v0 e2 [n=3,n-1=2,…]
letk v0 e2 [n=4,n-1=3,…]

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                4(let ([s (+ v0 v1)])
                  5s))))))))
```

```
s [s=1,v0=1,v1=0,…]
```

```
letk v0 e2 [n=3,n-1=2,…]
```

```
letk v0 e2 [n=4,n-1=3,…]
```

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                4(let ([s (+ v0 v1)])
                  5s))))))))
```

(fib 1) -> 1

letk v1 e_4 [v0=1,n=3,n-1=2,…]
letk v0 e_2 [n=4,n-1=3,…]

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
               3(let ([v1 (fib n-2)])
                  4(let ([s (+ v0 v1)])
                     5s))))))))
```

```
s [s=2,v0=1,v1=1,…]
```

```
letk v0 e2 [n=4,n-1=3,…]
```

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                 4(let ([s (+ v0 v1)])
                   5s))))))))
```

```
(fib 2)
```

```
letk v1 e4 [v0=2,n=4,n-1=3,…]
```

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                4(let ([s (+ v0 v1)])
                  5s))))))))
```

(fib 1) -> 1

letk v0 e_2 [n=2,n-1=3,…]

letk v1 e_4 [v0=2,n=4,n-1=3,…]

ANF

```
(define (fib n)
  (let ([c (<= n 1)])
    (if c
        n
       0(let ([n-1 (- n 1)])
         1(let ([v0 (fib n-1)])
            2(let ([n-2 (- n 2)])
              3(let ([v1 (fib n-2)])
                4(let ([s (+ v0 v1)])
                  5s)))))))))
```

```
(fib 0) -> 0
```

```
letk v1 e4 [v0=1,n=2,n-1=3,…]
```

```
letk v1 e4 [v0=2,n=4,n-1=3,…]
```

CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```

For simplicity, in *this example*, the added parameter ‘k’ comes *last*, by convention, and is *not added* to continuations (which is only needed for first-class konts so they may be treated as functions).

```
(fib 4 print)
```

CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```

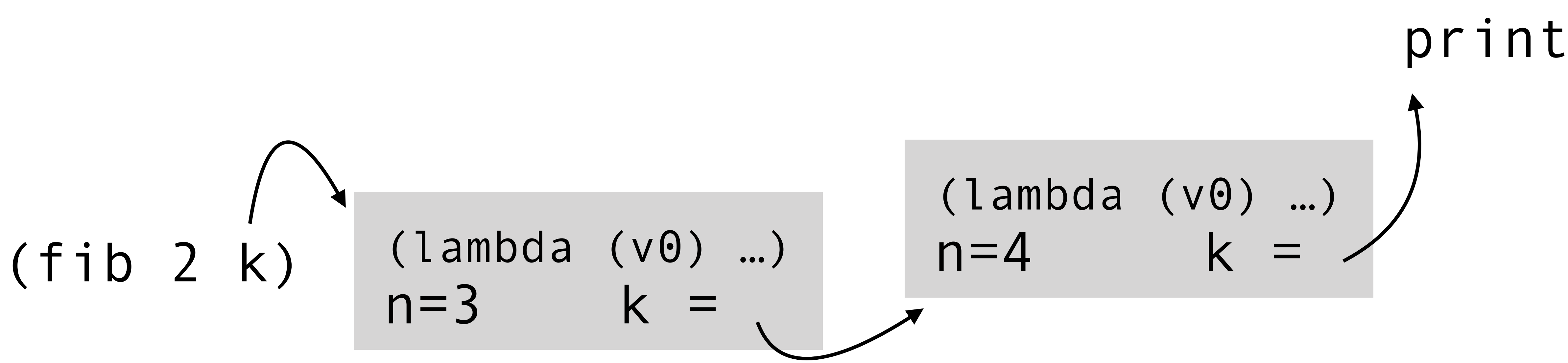
(fib 3 k)

(lambda (v0) …)
n=4 k =

print

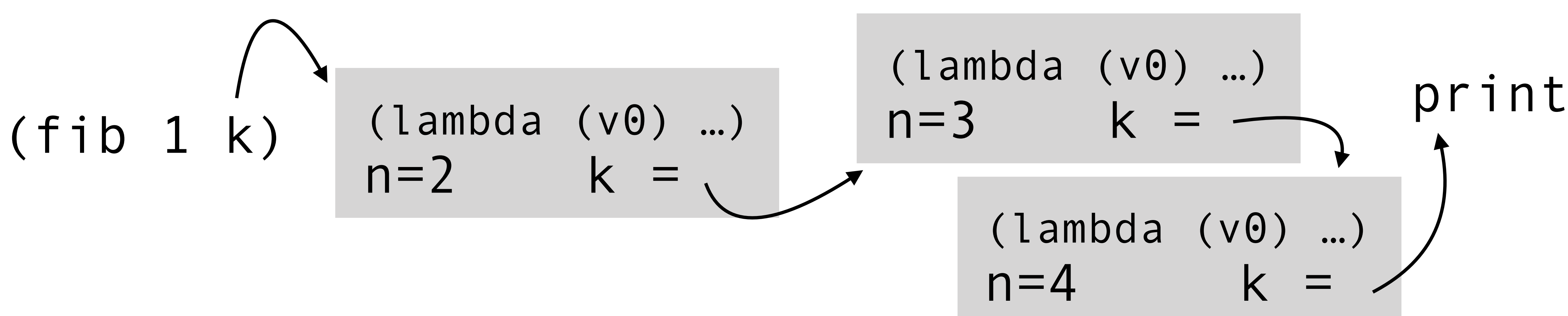
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



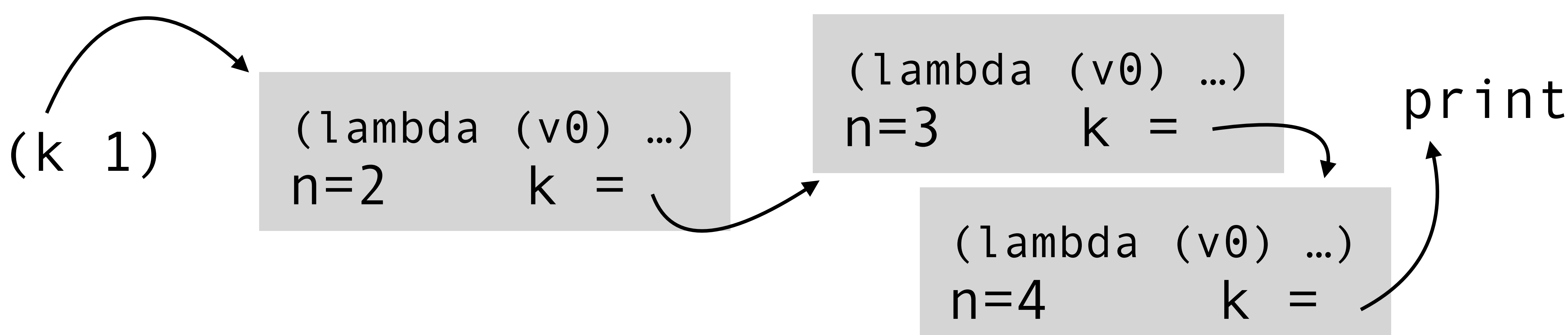
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s)))))))))))
```



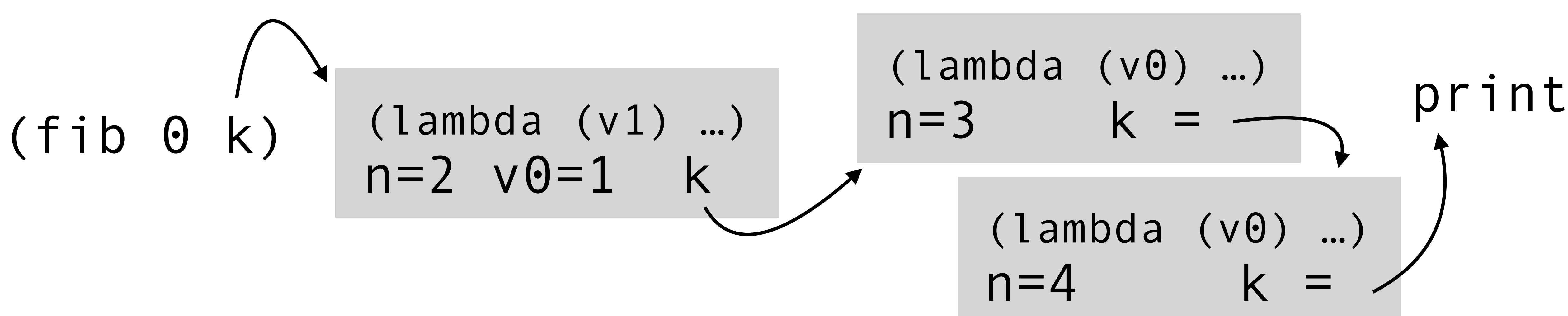
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



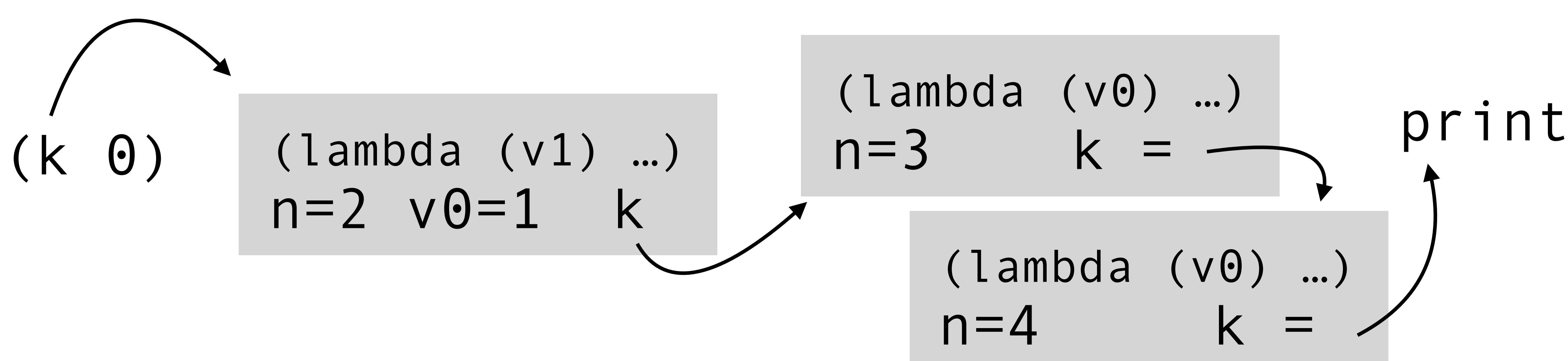
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s)))))))))))
```



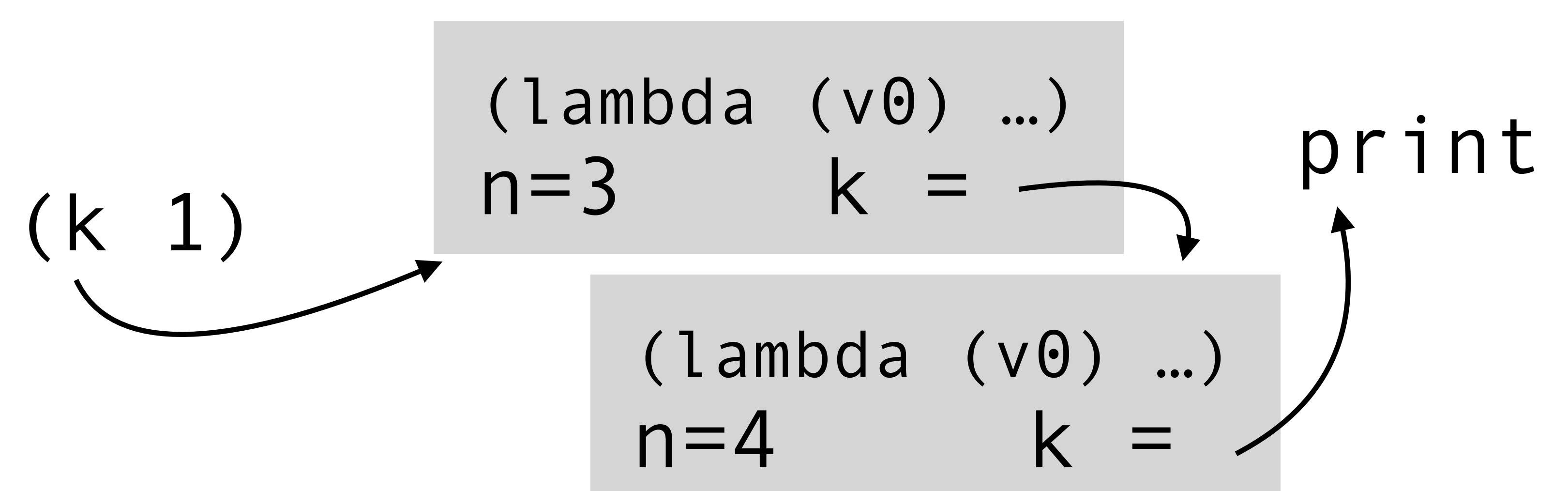
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



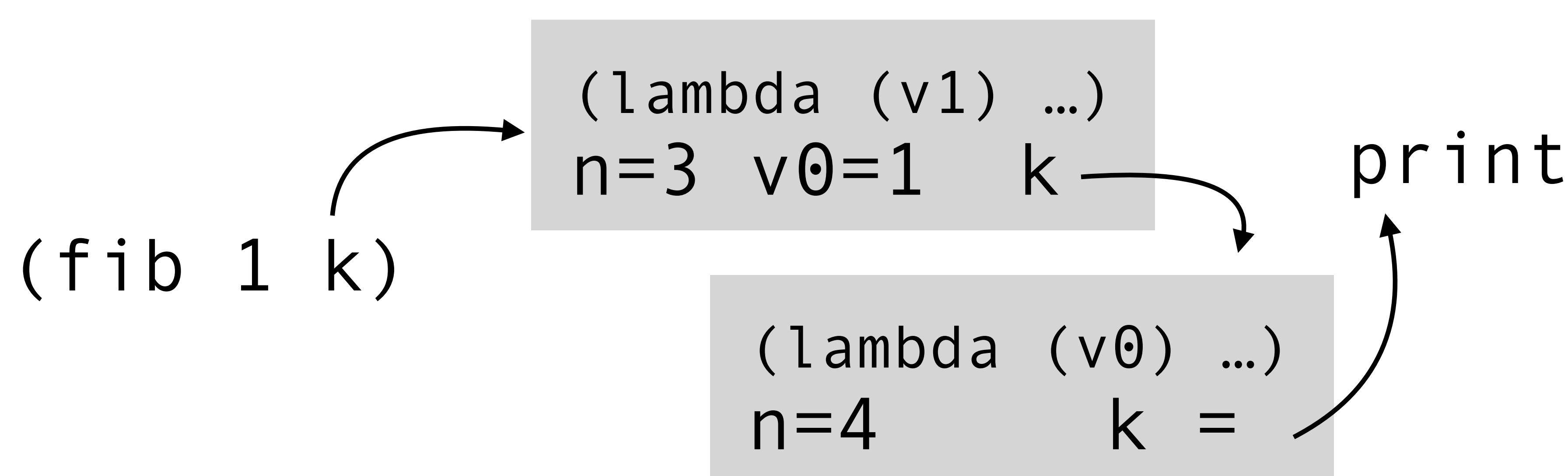
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s)))))))))))
```



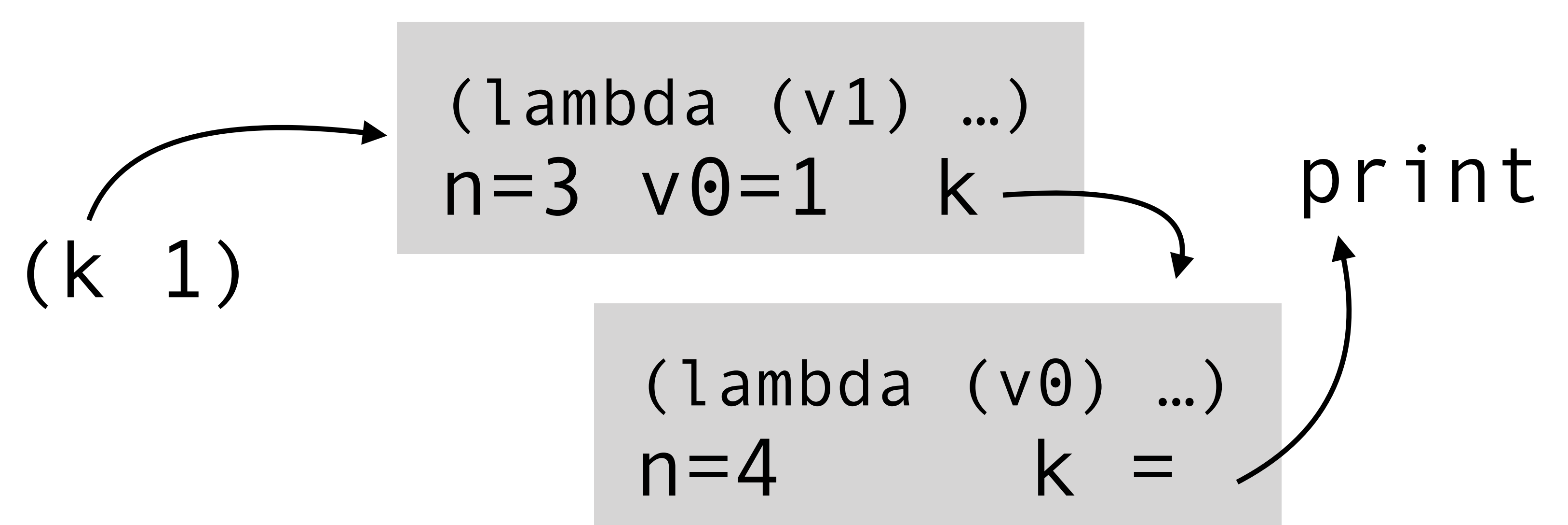
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```

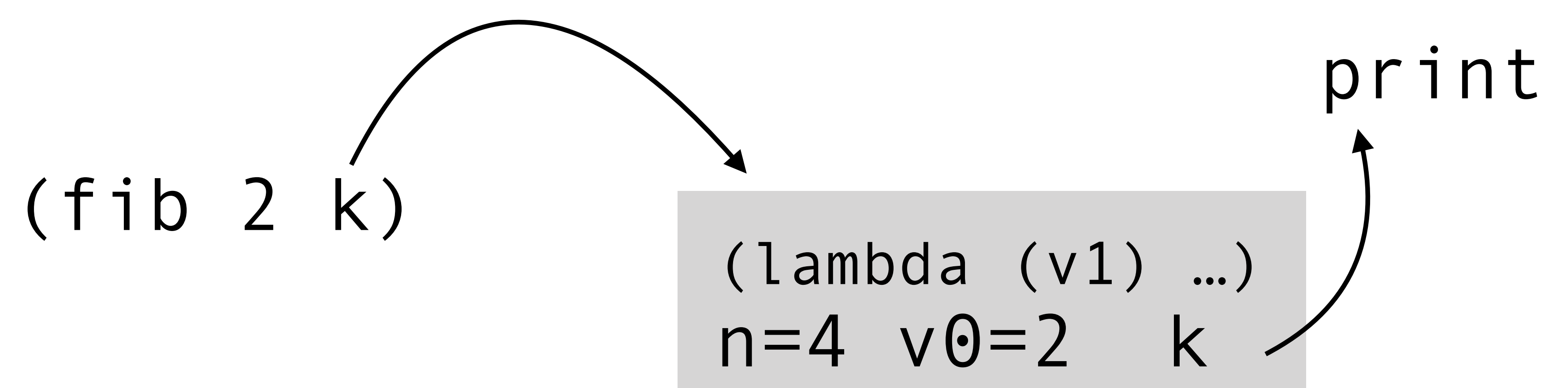
(k 2)

(lambda (v0) …)
n=4 k =

print

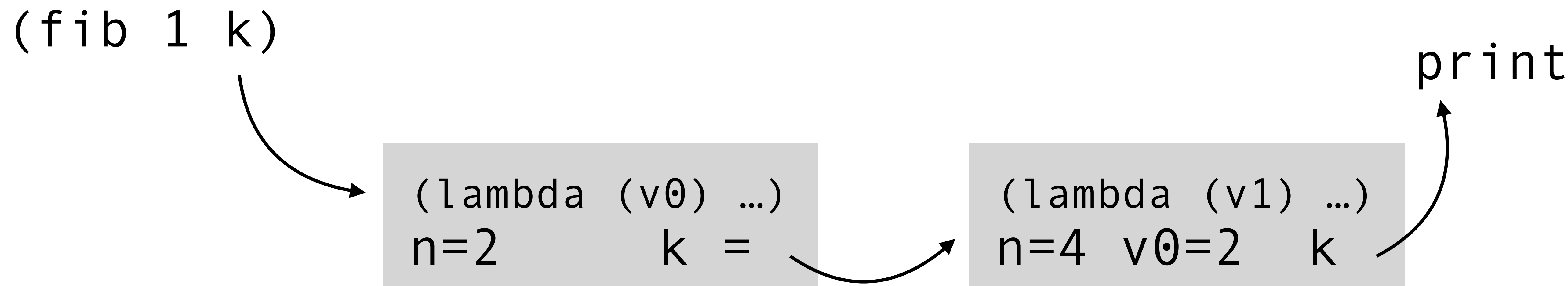
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



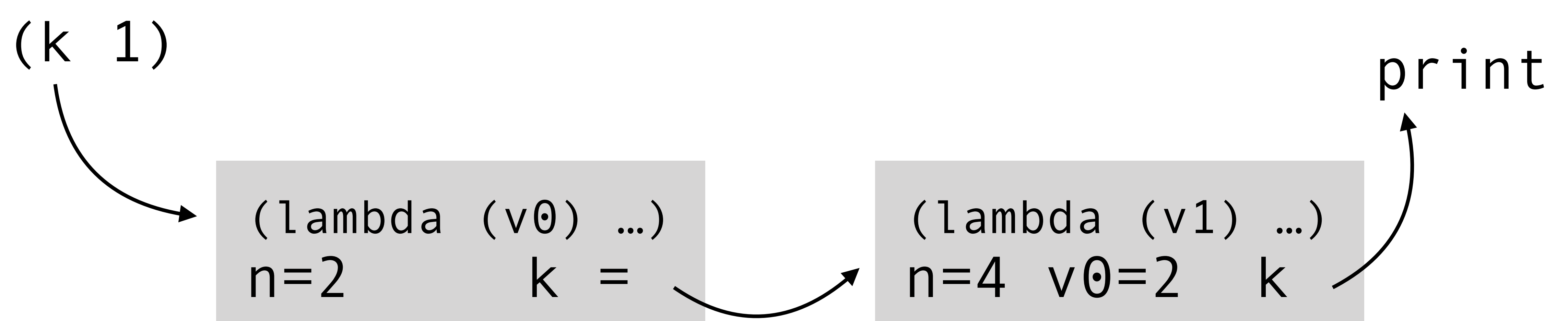
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



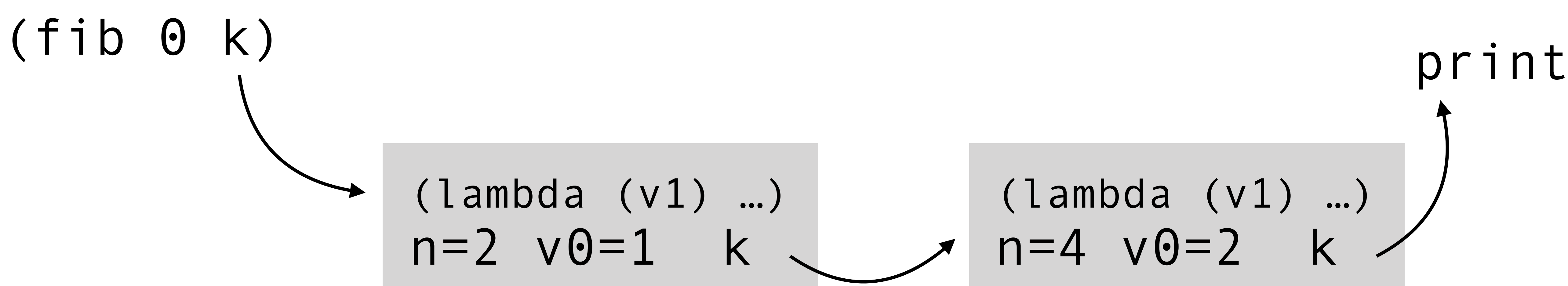
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



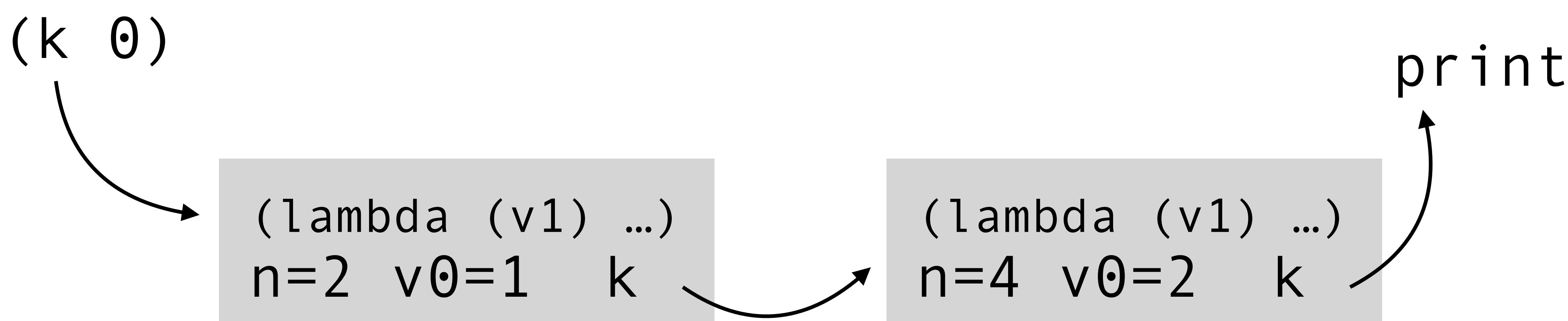
CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```



CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s)))))))))))))
```

```
print

(k 1)

(lambda (v1) …)
n=4 v0=2  k
```

CPS

```
(define (fib n k)
  (let ([c (<= n 1)])
    (if c
        (k n)
        (let ([n-1 (- n 1)])
          (fib n-1
            (lambda (v0)
              (let ([n-2 (- n 2)])
                (fib n-2
                  (lambda (v1)
                    (let ([s (+ v0 v1)])
                      (k s))))))))))))
```

```
(k 3)

                print
```