Matt Might
University of Utah
matt.might.net

Handling failure

P4 out (last Mon)

```
def f():
  try:
    return 10
finally:
  print("got here!")
return 20
```

```
def f():
  try:
    return 10
  finally:
    return 20
```

```
def f():
  try:
    return 10
finally:
  return 20
return 30
```

```
def f():
    try:
        raise Exception()
    except:
        return 10
finally:
    print("got here!")
return 20
```

```
def f():
  try:
    raise Exception()
  except:
    return 10
finally:
  print("got here!")
  return 20
```

```
while True:
    try:
        continue
    finally:
        print "got here!"
```

return

while

for

try/throw

```
break
continue
finally
```

call/ec

diet-call/cc

call/ec = setjmp + longjmp

return

```
def fun(args):
    if easy case
        return something
    do something
    complicated
```

```
(lambda (args)
  (if easy
      something
      do something complicated))
```

```
def fun(args):
    if easy case:
        return something
    if also easy case:
        return something else
    do something
    complicated
```

```
(lambda (args)
  (if easy
      something
      (if also easy
          something else
          do something complicat
```

```
def contains(e,seq):
    for x in seq:
        if x is e:
            return True
    return False
```

```
(λ (e seq)
  (for ([x seq])
    (if (eq? x e)
        (return #t)))
  #f)
```

```
(λ (e seq)
  (for ([x seq])
    (if (eq? x e)
        ([redacted] #t)))
  #f)
```

What *is* return?

A second-class continuation.

λ/return

```
(λ/return (e seq)
  (for ([x seq])
    (if (eq? x e)
        (return #t)))
  #f)
```

```
(λ/return (params) body)
```

=>

```
(λ (params)
  (call/cc (λ (return)
             body)))
```

`call/cc` too heavyweight!

Heap

Env

Stack

Heap

Env

Stack

Env

Stack

Heap

Env

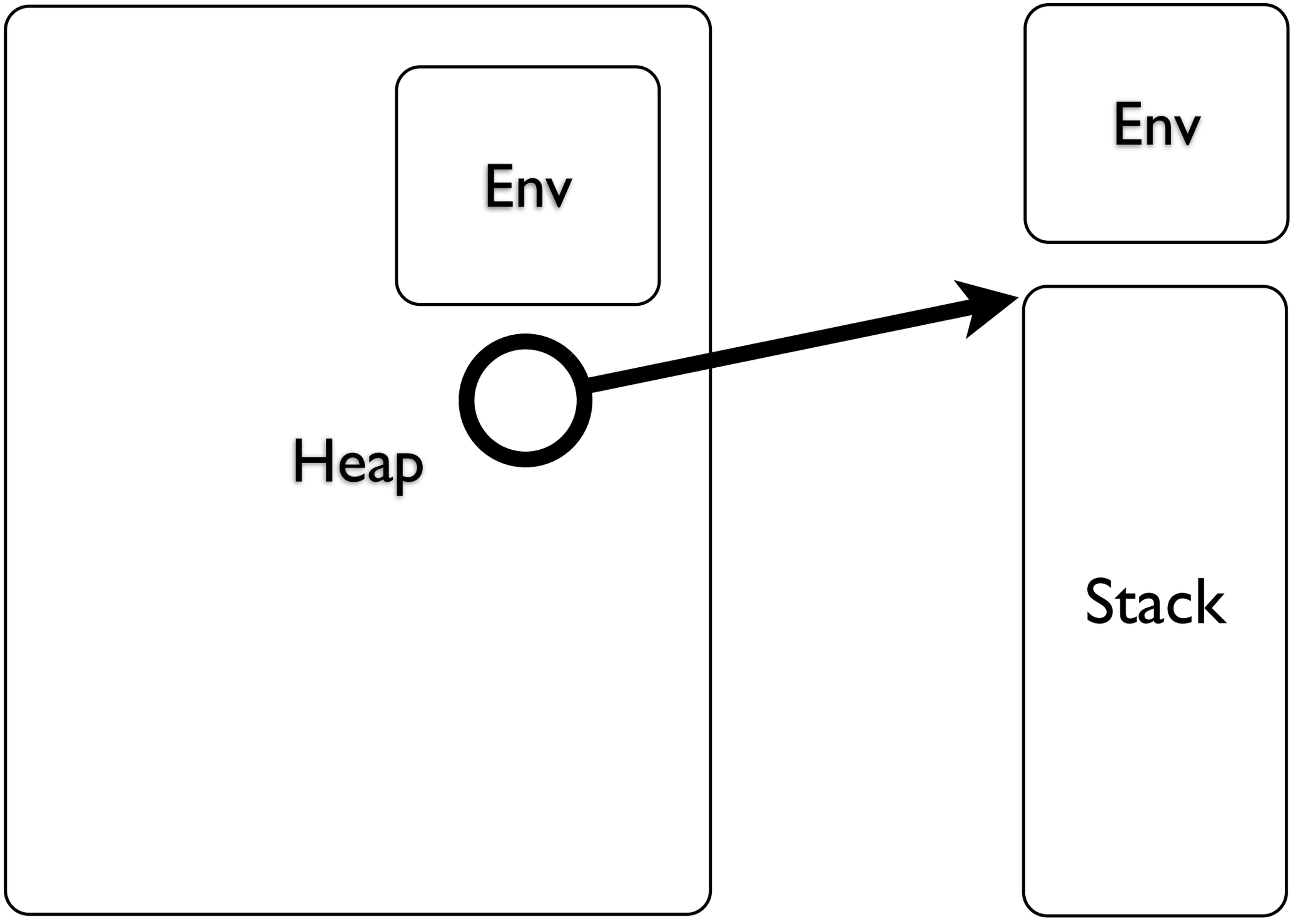
Stack

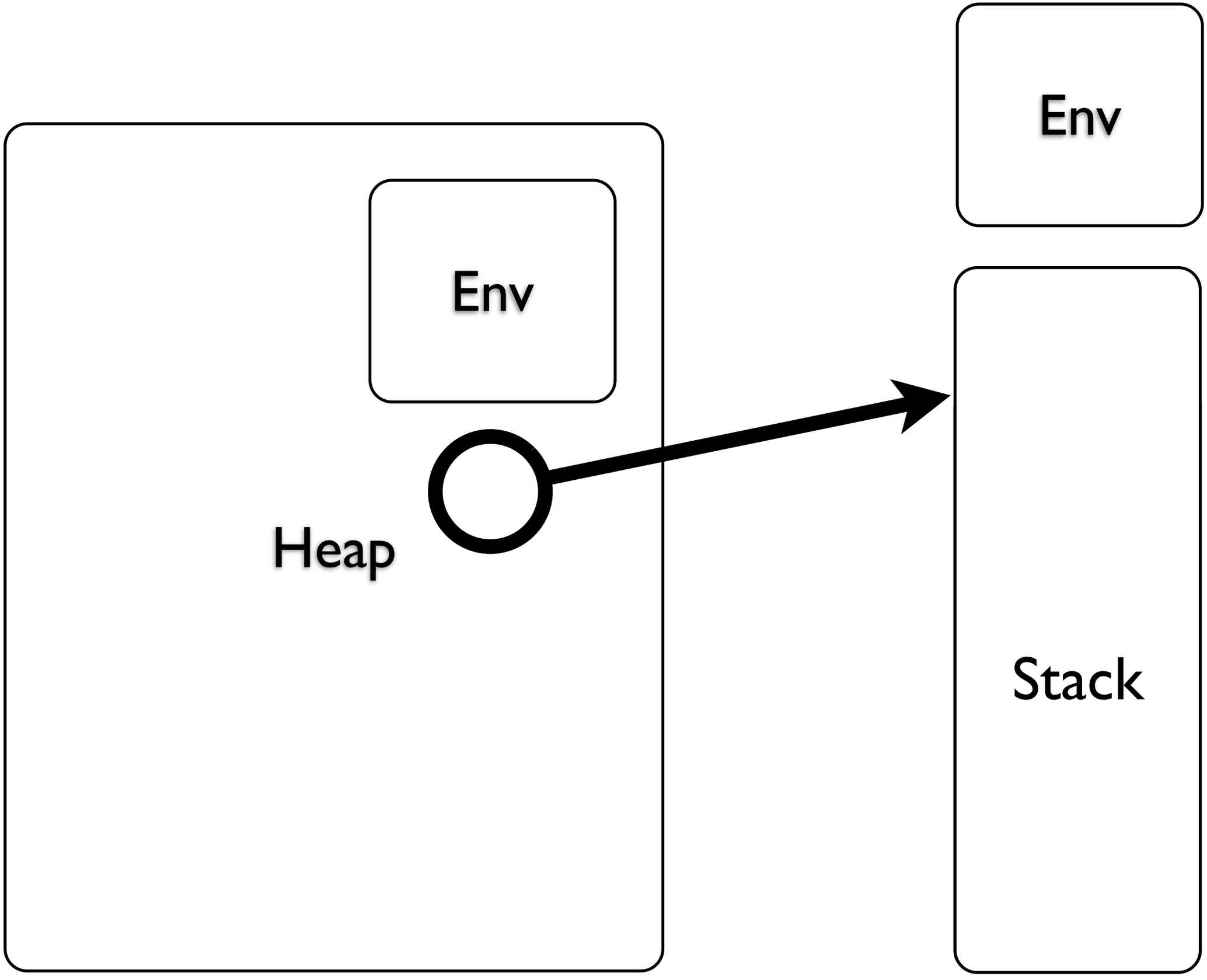
`call/ec` is just right!

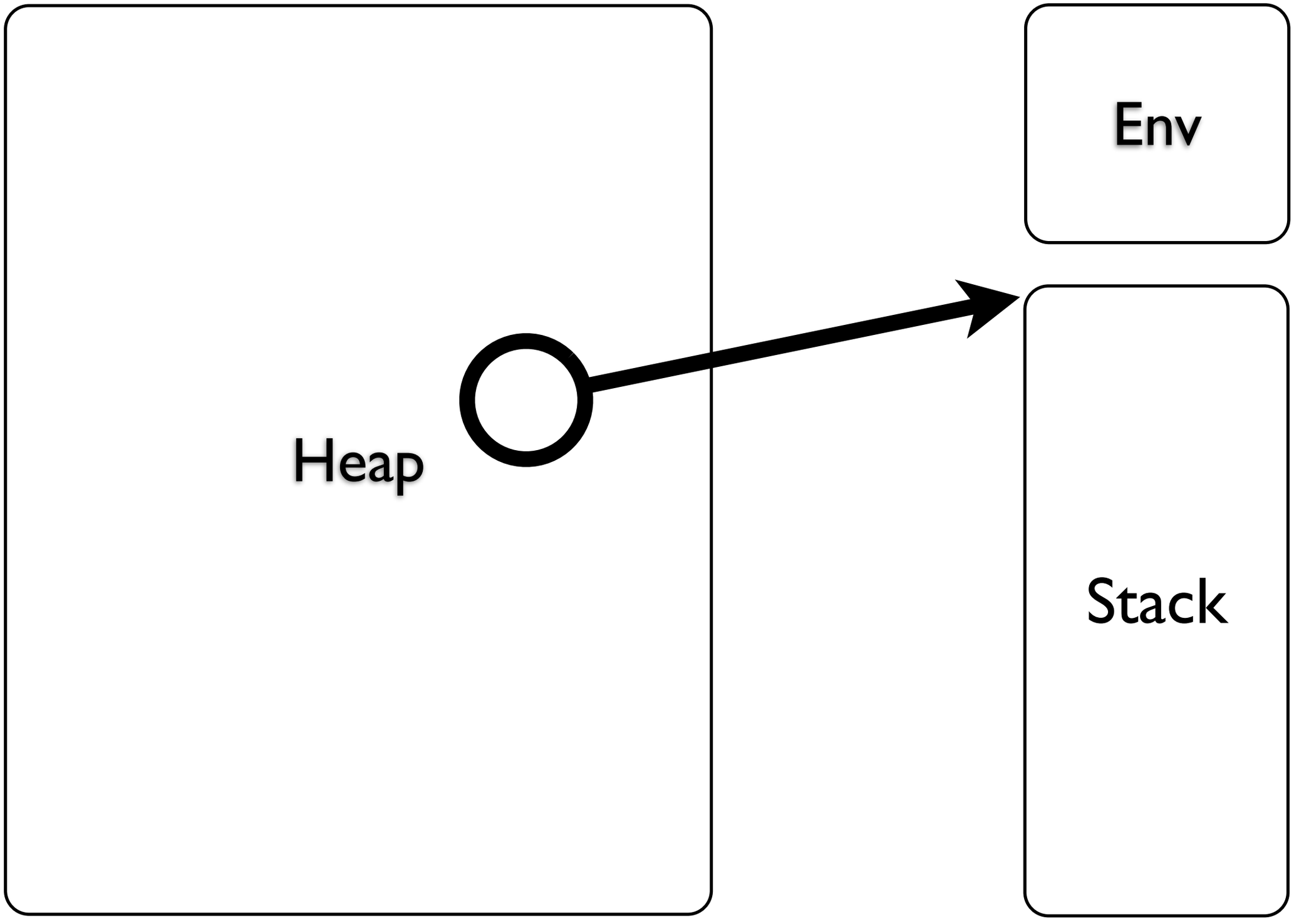
Heap

Env

Stack


Env
Heap
Env
Stack


Env
Env
Heap
Stack


Heap
Env
Stack

```
(λ/return (params) body)
```

=>

```
(λ (params)
  (call/ec (λ (return)
             body)))
```

while

```
while cond:
    do-this
```

```
(letrec ([loop (λ ()
    (when cond
        do-this
        (loop)))])
(loop))
```

```
while cond:
    do-this
    continue
do-that
```

```
(letrec ([loop (λ ()
    (when cond
        do-this
        (continue)
        do-that
        (loop)))])
(loop))
```

```
(letrec ([loop (λ ()
  (when cond
    (call/ec (λ (continue)
      do-this
      (continue)
      do-that))
    (loop))))])
(loop))
```

```
while cond:
    do-this
    break
    do-that
```

```
(letrec ([loop (λ ()
  (when cond
    (call/ec (λ (continue)
      do-this
        (break)
      do-that))
    (loop))))])
(loop))
```

```
(call/ec (λ (break)
 (letrec ([loop (λ ()
   (when cond
    (call/ec (λ (continue)
      do-this
       (break)
      do-that))
   (loop)))
(loop))))
```

```
while cond:
    do-this
else:
    do-that
```

```
(call/ec (λ (break)
 (letrec ([loop (λ ()
   (when cond
    (call/ec (λ (continue)
     do-this))
   (loop)))
(loop)
do-that))
```

for

```
for i in seq:
    do-this
```

```
(for-each seq (λ (i) do-this))
```

```
(call/ec (λ (break)
  (for-each seq (λ (i)
    (call/ec (λ (continue)
      do-this))))))
```

```
for i in seq:
    do-this
else:
    do-that
```

```
(call/ec (λ (break)
  (for-each seq (λ (i)
    (call/ec (λ (continue)
      do-this))))
  do-that)))
```

```
for i in seq:
    do-this
else:
    do-that
print(i)
```

```
(call/ec (λ (break)
  (for-each seq (λ ($v)
    (set! i $v)
    (call/ec (λ (continue)
      do-this)))
  do-that)
```

throw

```
(define (throw ex)
  ($handler ex))
```

try

Two approaches

- Designated global handler
- Second continuation in CPS

```
try:
  do-this
except:
  do-that
```

```
(try
  do-this
  (λ (ex)
    do-that))
```

```
(try
  do-this
  handler)
```

```
(let ([$old $handler])
 (call/ec (lambda (ec)
  (set! $handler
        (λ (ex)
          (set! $handler $old)
          (ec (handler ex))))))
 (let ([rv do-this])
  (set! $handler $old)
  rv))))
```

```
while cond:
  try:
    break
except:
  do-that
```

```
(let* ([$old  $handler]
       [break (λ ()
                (set! $handler $old)
                (break))])
 (call/ec (lambda (ec)
  (set! $handler
        (λ (ex)
          (set! $handler $old)
          (ec (handler ex))))
  (let ([rv do-this])
   (set! $handler $old)
   rv))))
```

finally

```
(try
  do-this
  handler
  final-code)
```

```
(let ([$fin (λ () (void))])
 (call/ec (λ (finally)
  (let* ([$old  $handler]
         [break (λ ()
                  (set! $handler $old)
                  (set! $fin break)
                  (finally))]
    (call/ec (lambda (ec)
     (set! $handler
           (λ (ex)
             (set! $handler $old)
             (ec (handler ex))))
     (let ([rv do-this])
      (set! $handler $old)
      rv))))
final-code
($fin))
```

Questions?