Continuing CPS

COS 326
David Walker
Princeton University

Last Time

Last time, we saw we could take code like this:

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

and turn it in to code like this:

```
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    0
;;

sum_to_cont 100 (fun s -> s)
```

Last Time

Last time, we saw we could take code like this:

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

and turn it in to code like this:

continuation-passing style (CPS): each function takes a *continuation* that tells it "what to do next"

```
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    0
;;

sum_to_cont 100 (fun s -> s)
```

one may only call a function if it is the *last* thing one does in the current function

Non-tail recursive

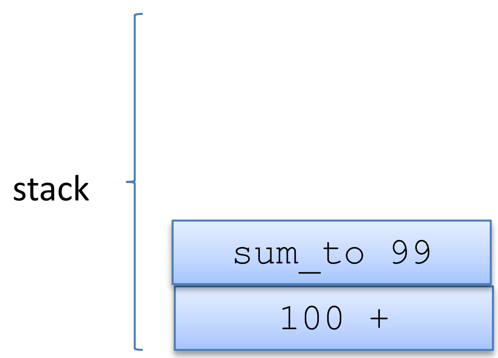
```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

stack

```
sum_to 100
```

Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

stack

sum_to 98
99 +
100 +

Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

sum_to 0

.

.

.

stack

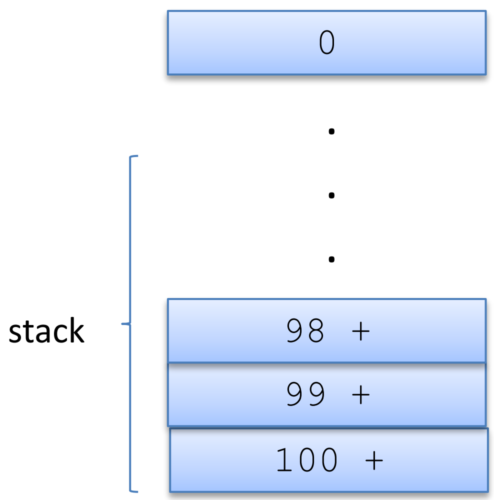
98 +

99 +

100 +

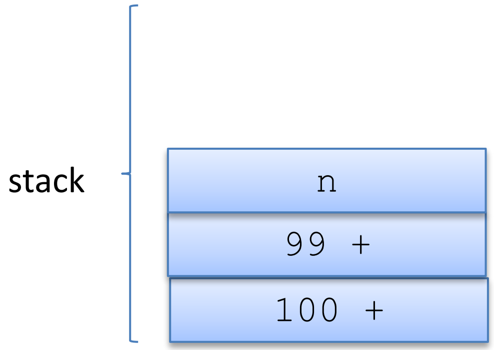
Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



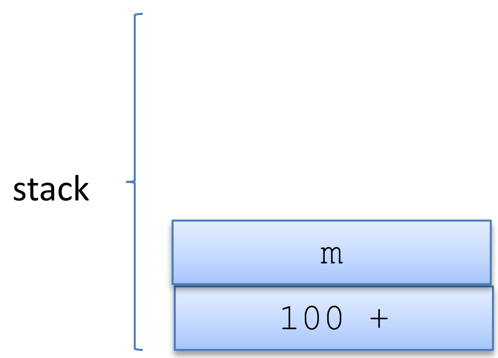
Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



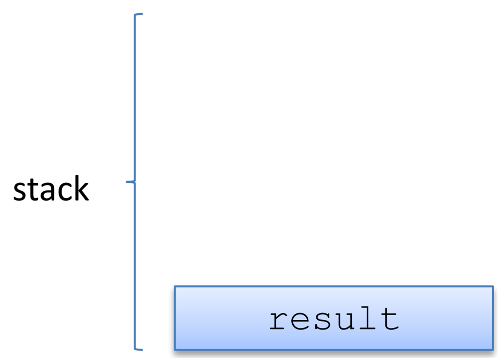
Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



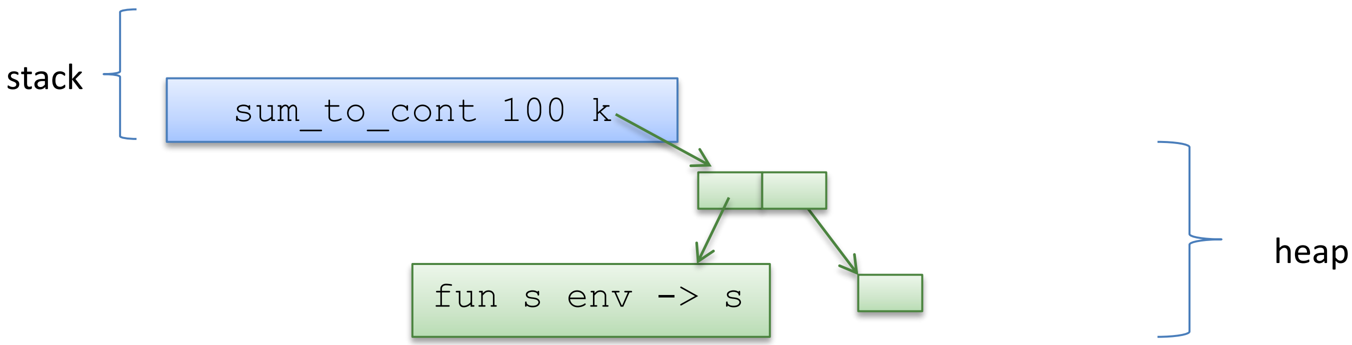
Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



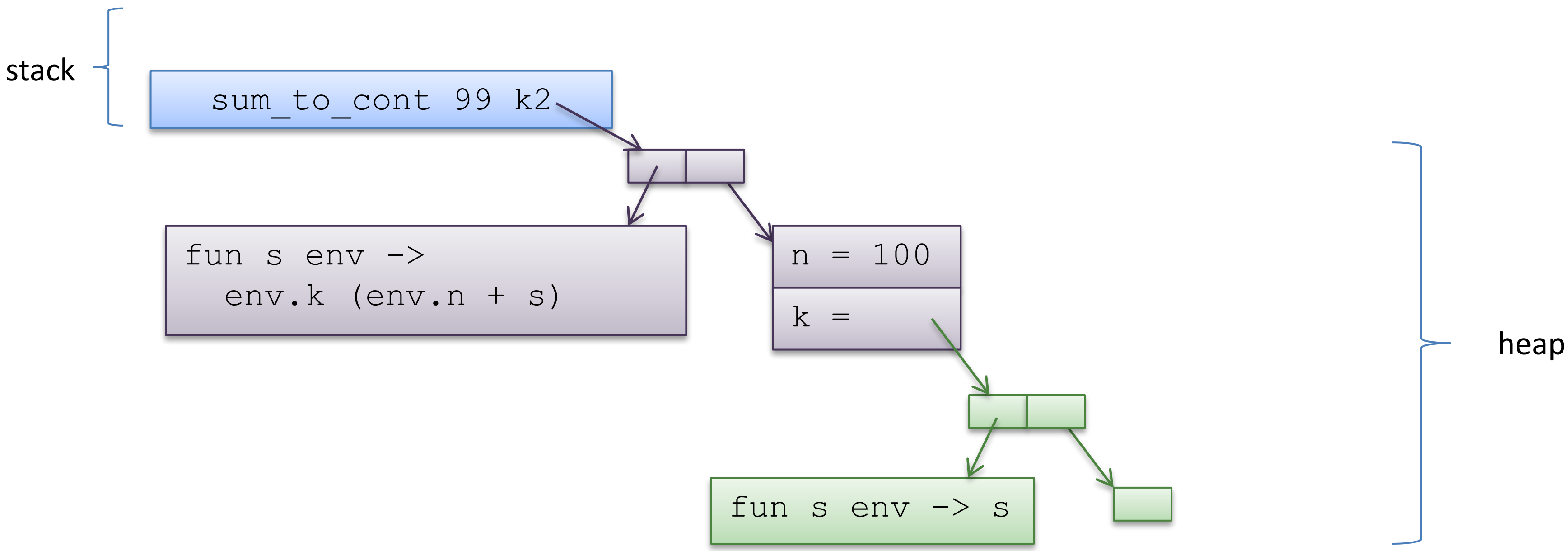
Continuation-passing style

```
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    k 0  ;;

sum_to_cont 100 (fun s -> s)
```



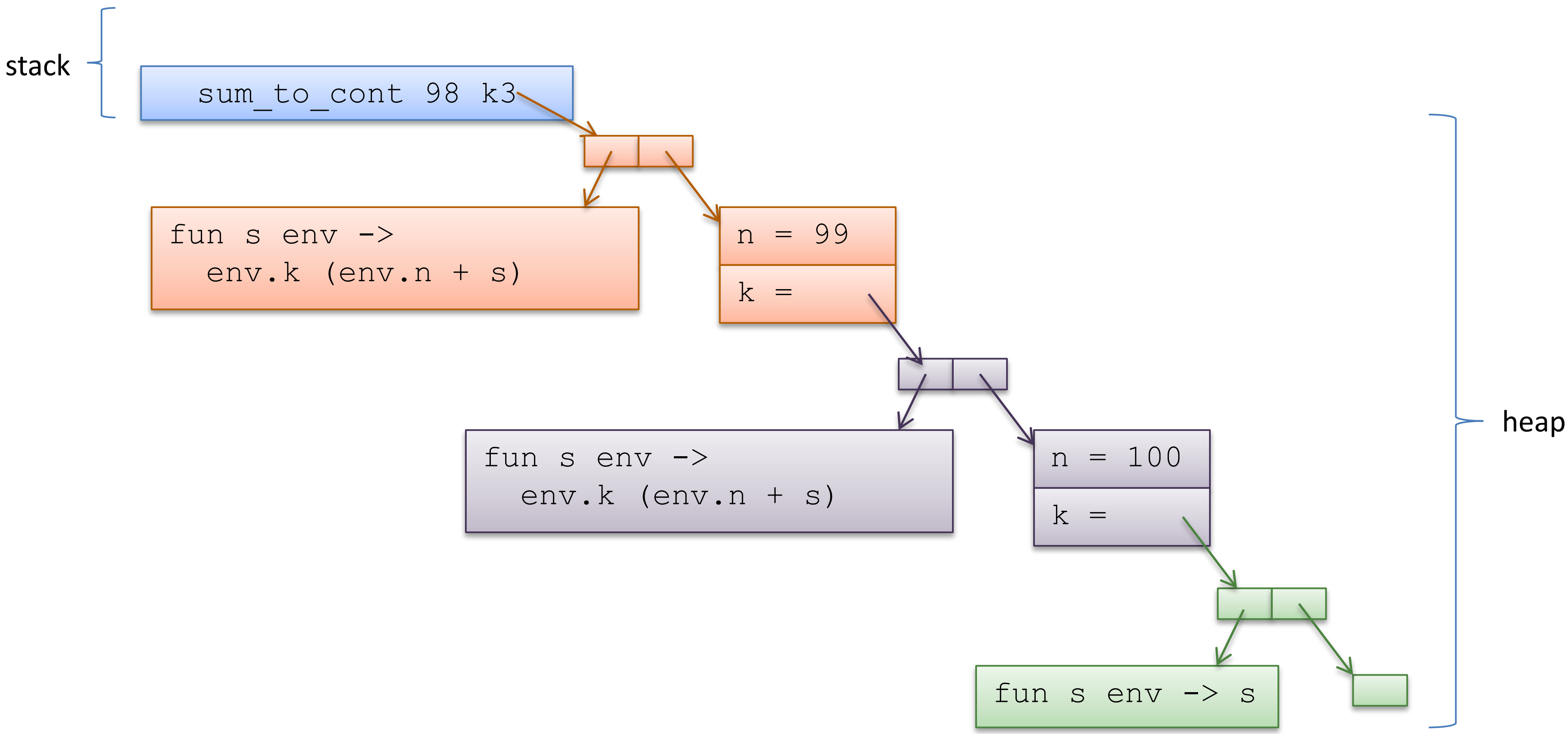
Continuation-passing style

```
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    k 0  ;;

sum_to 100 (fun s -> s)
```



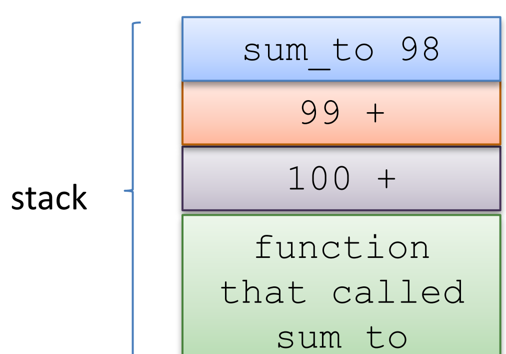
Continuation-passing style

```
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    k 0  ;;

sum_to 100 (fun s -> s)
```



Back to stacks

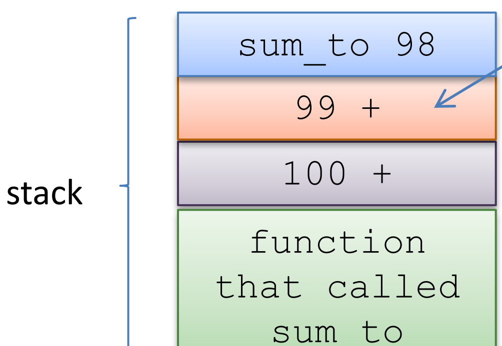
```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



Back to stacks

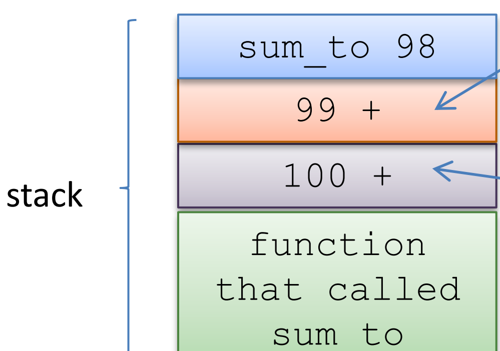
```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

but how do you really implement that?



Back to stacks

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



but how do you really implement that?

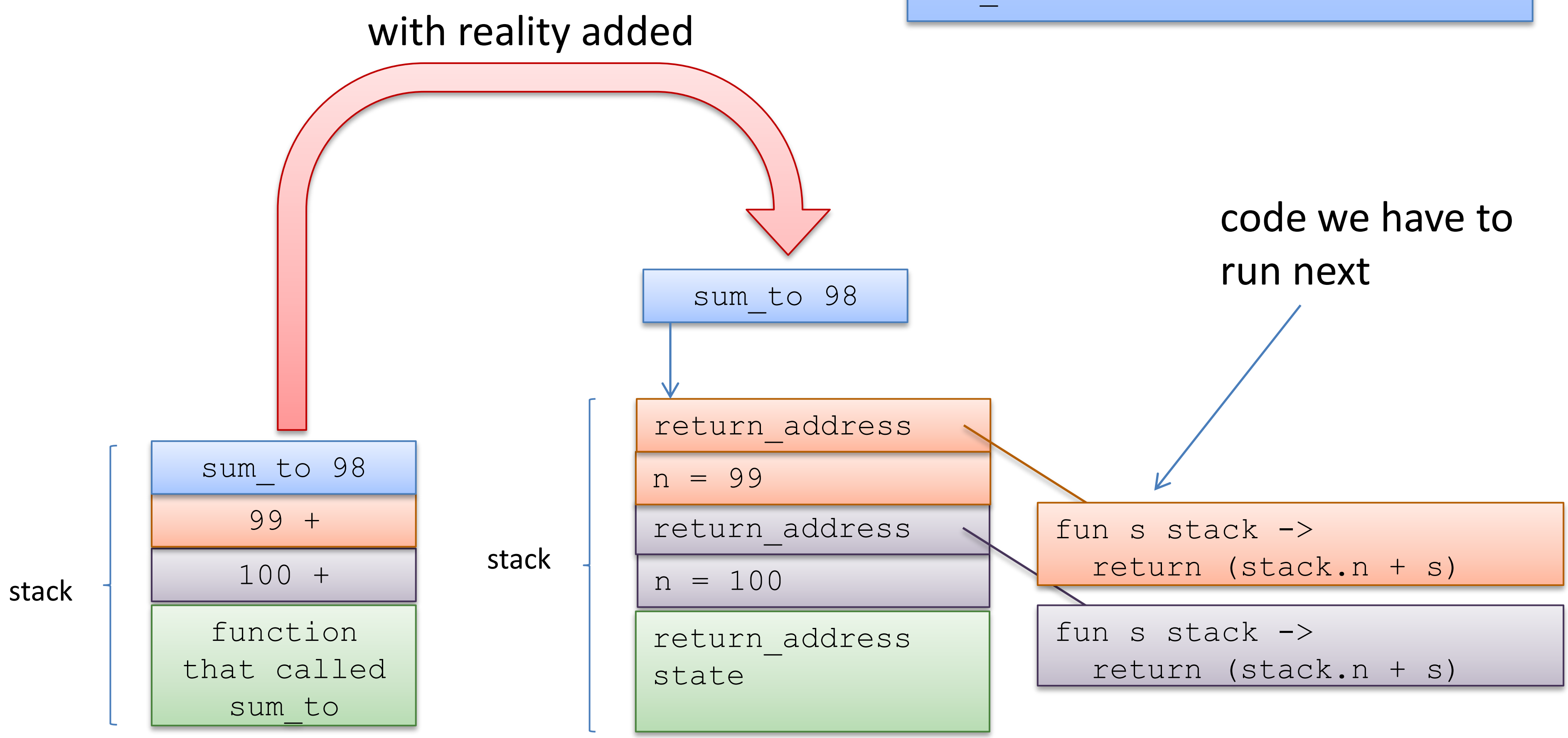
there is two bits of information here:

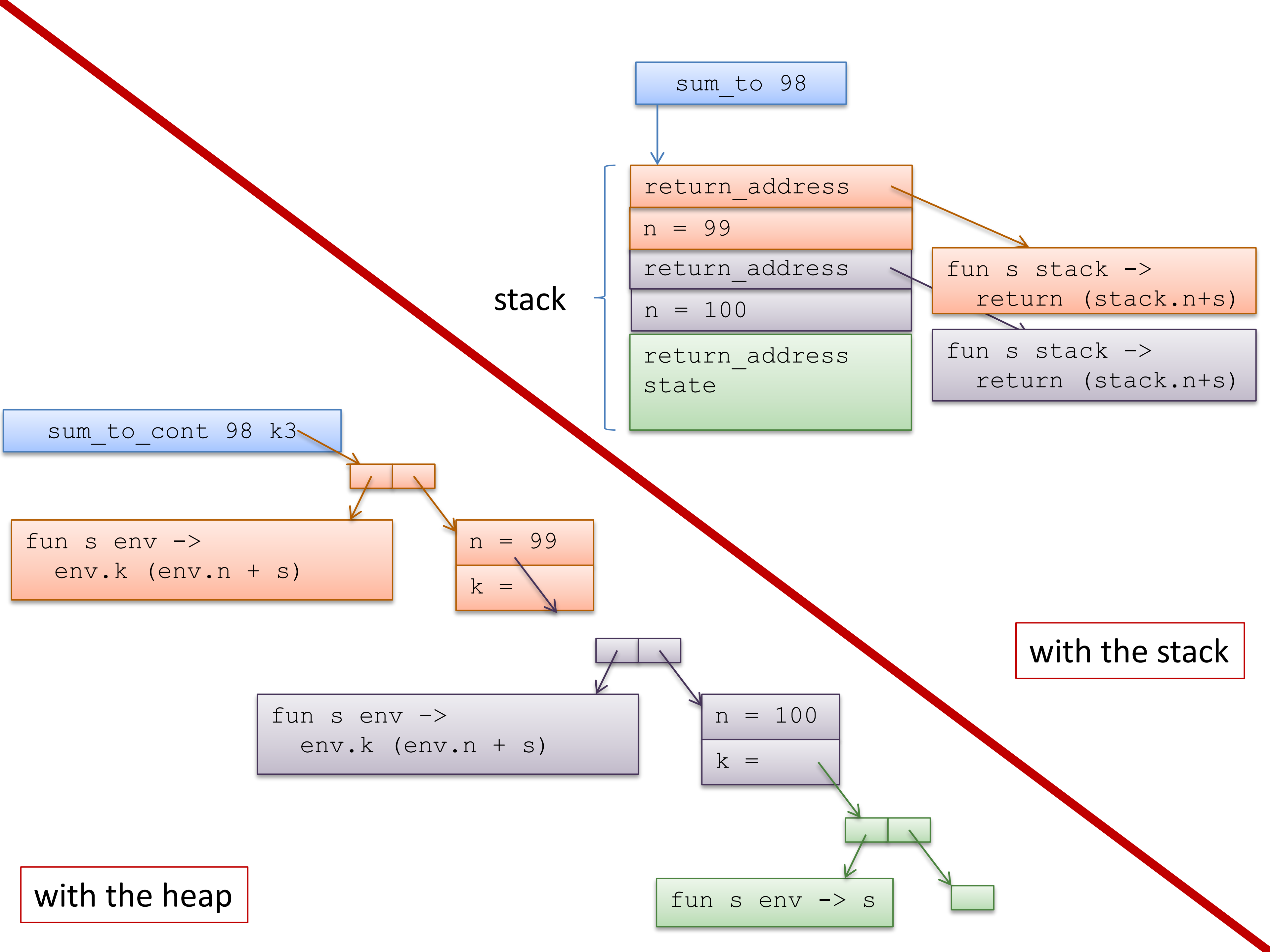
(1) some state (n=100) we had to remember

(2) some code we have to run later

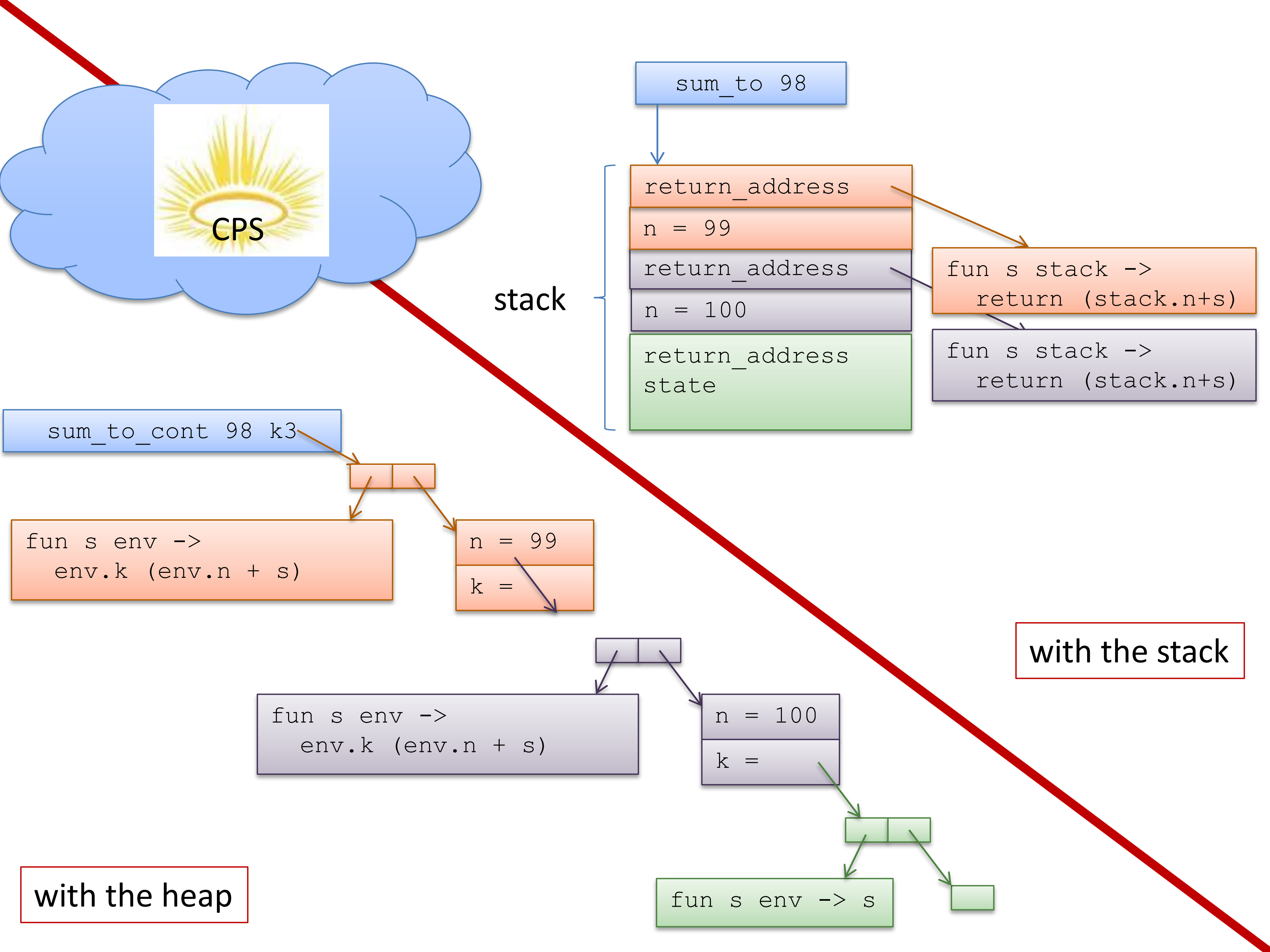
Back to stacks

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```





sum_to 98
stack
return_address
n = 99
return_address
n = 100
return_address
state
fun s stack -> return (stack.n+s)
fun s stack -> return (stack.n+s)
sum_to_cont 98 k3
fun s env -> env.k (env.n + s)
n = 99
k =
fun s env -> env.k (env.n + s)
n = 100
k =
fun s env -> s
with the stack
with the heap



CPS
sum_to 98
stack
return_address
n = 99
return_address
n = 100
return_address
state
fun s stack ->
return (stack.n+s)
fun s stack ->
return (stack.n+s)
sum_to_cont 98 k3
fun s env ->
env.k (env.n + s)
n = 99
k =
fun s env ->
env.k (env.n + s)
n = 100
k =
fun s env -> s
with the stack
with the heap

Why CPS?

Continuation-passing style is *inevitable*.

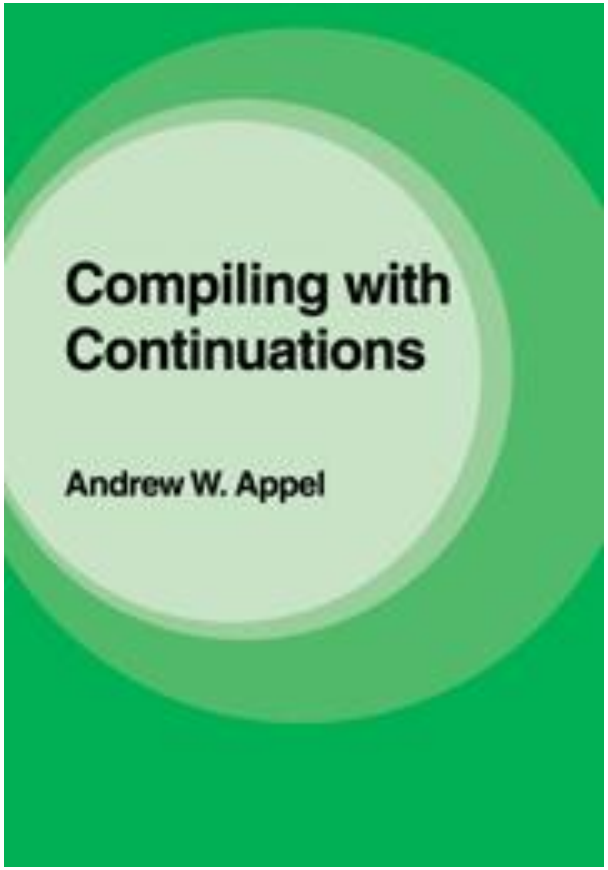
It does not matter whether you program in Java or C or OCaml -- there's code around that tells you "*what to do next*"

- If you explicitly CPS-convert your code, "*what to do next*" is stored on the heap
- If you don't, it's stored on the stack

If you take a conventional compilers class, the continuation will be called a *return address* (but you'll know what it really is!)

The idea of a *continuation* is much more general!

Standard ML of New Jersey



Your compiler can put all the continuations in the heap so you don't have to (and you don't run out of stack space)!

Other pros:

- light-weight concurrent threads

Some cons:

- linked list of closures can be less space-efficient than stack
- hardware architectures optimized to use a stack
- see Empirical and Analytic Study of Stack versus Heap Cost for Languages with Closures. Shao & Appel

Can we get away with zero continuations?

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

```
let rec sum_to_opt (n:int) (acc:int) : int =
  if n > 0 then
    sum_to_opt (n-1) (acc + n)
  else
    acc
;;

sum_to_opt 100
```

not only did we make the function tail-recursive, but we didn't add any stack or linked-list of closures!

Can we get away with zero continuations?

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

```
let rec sum_to_opt (n:int) (acc:int) : int =
  if n > 0 then
    sum_to_opt (n-1) (acc + n)
  else
    acc
;;

sum_to_opt 100
```

not only did we make the function tail-recursive, but we didn't add any stack or linked-list of closures!

```
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    0  ;;

sum_to_cont 100 (fun s -> s)
```

Can we get away with zero continuations?

```
let rec print_stack (n:int) : unit =
  if n > 0 then
    Printf.printf "push %d\n" n;
    print_stack (n-1);
    Printf.printf "pop %d\n" n
  else
    Printf.printf "zero"
;;

print_stack 5 ;;
```

output:

```
push 4
push 3
push 2
push 1
zero
pop 1
pop 2
pop 3
pop 4
```

Can we get away with zero continuations?

```
let rec print_stack (n:int) : unit =
  if n > 0 then
    Printf.printf "push %d\n" n;
    print_stack (n-1);
    Printf.printf "pop %d\n" n
  else
    Printf.printf "zero"
;;

print_stack 5 ;;
```

output:

```
push 4
push 3
push 2
push 1
zero
pop 1
pop 2
pop 3
pop 4
```

```
let rec print_stack_cont (n:int) (k:unit -> unit) : unit =







;;

print_stack_cont 5 (fun () -> ()) ;;
```

Can we get away with zero continuations?

```
let rec print_stack (n:int) : unit =
  if n > 0 then
    Printf.printf "push %d\n" n;
    print_stack (n-1);
    Printf.printf "pop %d\n" n
  else
    Printf.printf "zero"
;;

print_stack 5 ;;
```

output:

```
push 4
push 3
push 2
push 1
zero
pop 1
pop 2
pop 3
pop 4
```

```
let rec print_stack_cont (n:int) (k:unit -> unit) : unit =
  if n > 0 then



  else
    Printf.printf_cont "zero" k
;;

print_stack_cont 5 (fun () -> ()) ;;
```

Can we get away with zero continuations?

```
let rec print_stack (n:int) : unit =
  if n > 0 then
    Printf.printf “push %d\n” n;
    print_stack (n-1);
    Printf.printf “pop %d\n” n
  else
    Printf.printf “zero”
;;

print_stack 5 ;;
```

output:

```
push 4
push 3
push 2
push 1
zero
pop 1
pop 2
pop 3
pop 4
```

```
let rec print_stack_cont (n:int) (k:unit -> unit) : unit =
  if n > 0 then
    Printf.printf_cont “push %d\n” n (fun () ->
          ...
                                         )
  else
    Printf.printf_cont “zero” k
;;

print_stack_cont 5 (fun () -> ()) ;;
```

Can we get away with zero continuations?

```
let rec print_stack (n:int) : unit =
  if n > 0 then
    Printf.printf "push %d\n" n;
    print_stack (n-1);
    Printf.printf "pop %d\n" n
  else
    Printf.printf "zero"
;;

print_stack 5 ;;
```

output:

```
push 4
push 3
push 2
push 1
zero
pop 1
pop 2
pop 3
pop 4
```

```
let rec print_stack_cont (n:int) (k:unit -> unit) : unit =
  if n > 0 then
    Printf.printf_cont "push %d\n" n (fun () ->
      print_stack (n-1) (fun () ->
                                      ...))
  else
    Printf.printf_cont "zero" k
;;

print_stack_cont 5 (fun () -> ()) ;;
```

Can we get away with zero continuations?

```
let rec print_stack (n:int) : unit =
  if n > 0 then
    Printf.printf "push %d\n" n;
    print_stack (n-1);
    Printf.printf "pop %d\n" n
  else
    Printf.printf "zero"
;;

print_stack 5 ;;
```

output:

```
push 4
push 3
push 2
push 1
zero
pop 1
pop 2
pop 3
pop 4
```

```
let rec print_stack_cont (n:int) (k:unit -> unit) : unit =
  if n > 0 then
    Printf.printf_cont "push %d\n" n (fun () ->
      print_stack (n-1) (fun () ->
        Printf.printf_cont "pop %d\n" n k))
  else
    Printf.printf_cont "zero" k
;;

print_stack_cont 5 (fun () -> ()) ;;
```

ANOTHER EXAMPLE

Another Example

```
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) -> Node (i+j, incr left i, incr right i)
;;
```

```
let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) ->
      let left_done = incr left i in
      let right_done = incr right i in
      Node (i+j, left_done, right_done)
;;
```

sometimes called *A-Normal Form*

(don't have function calls as arguments to other function calls)

Another Example

```
let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) ->
      let left_done  = incr left i  in
      let right_done = incr right i in
      Node (i+j, left_done, right_done)
;;
```

```
type cont = tree -> tree ;;

let rec incr_cps (t:tree) (i:int) (k:cont): tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
      let left_done =  incr left i in
      let right_done = incr right i in
      k (Node (i+j, left_done, right_done))
;;
```

Another Example

```
type cont = tree -> tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) ->
      let left_done  = incr left i  in
      let right_done = incr right i in
      Node (i+j, left_done, right_done)
;;
```

```
type cont = tree -> tree ;;

let rec incr_cps (t:tree) (i:int) (k:tree -> tree): tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
                         incr_cps left i  (fun left_done ->
      let right_done = incr right i in
      k (Node (i+j, left_done, right_done))
;;
```

Another Example

```
type cont = tree -> tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) ->
      let left_done  = incr left i in
      let right_done = incr right i in
      Node (i+j, left_done, right_done)
;;
```

```
type cont = tree -> tree ;;

let rec incr_cps (t:tree) (i:int) (k:cont): tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
                      incr_cps left  i (fun left_done ->
                      incr_cps right i (fun right_done ->
      k (Node (i+j, left_done, right_done)))
;;
```

Another Example

```
type cont = tree -> tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) ->
      let left_done  = incr left i in
      let right_done = incr right i in
      Node (i+j, left_done, right_done)
;;
```

```
type cont = tree -> tree ;;

let rec incr_cps (t:tree) (i:int) (k:cont): tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
                      incr_cps left  i (fun left_done ->
                        incr_cps right i (fun right_done ->
                          k (Node (i+j, left_done, right_done)))
;;
```

In general

```
let g input =
  f3 (f2 (f1 input))
;;
```

Direct Style

```
let g input =
  let x1 = f1 input in
  let x2 = f2 x1    in
  f3 x2
;;
```

A-normal Form

```
let g input k =
  f1 input (fun x1 ->
  f2 x1    (fun x2 ->
  f3 x2 k))
;;
```

CPS converted

CPS Fixes Evaluation Order

Without CPS, consider left-to-right evaluation vs. right-to-left evaluation:

```
let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) ->
      print_int j; Node (i+j, incr left i, incr right i)
;;
```

With CPS, you get results regardless of evaluation order:

```
type cont = tree -> tree ;;

let rec incr_cps (t:tree) (i:int) (k:cont): tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
      print_int j (fun () ->
        incr_cps left  i (fun left_done ->
          incr_cps right i (fun right_done ->
            k (Node (i+j, left_done, right_done)))
;;
```

Serial Killer or PL Researcher?

Serial Killer or PL Researcher?

Gordon Plotkin
Programming languages researcher
Invented CPS conversion.

Call-by-Name, Call-by Value
and the Lambda Calculus. TCS, 1975.

Robert Garrow
Serial Killer

Killed a teenager at a campsite
in the Adirondacks in 1974.
Confessed to 3 other killings.

Serial Killer or PL Researcher?

Gordon Plotkin
Programming languages researcher
Invented CPS conversion.

Call-by-Name, Call-by Value
and the Lambda Calculus. TCS, 1975.

Robert Garrow
Serial Killer

Killed a teenager at a campsite
in the Adirondacks in 1974.
Confessed to 3 other killings.

Call-backs: Another use of continuations

Call-backs:

```
request_url : url -> (html -> 'a) -> 'a

request_url "http://www.stuff.com/i.html"
   (fun html -> process html)
```

continuation

Summary

CPS is interesting and important:

- *unavoidable*
    - assembly language is continuation-passing
- *theoretical ramifications*
    - fixes evaluation order
    - call-by-value evaluation == call-by-name evaluation
- *efficiency*
    - generic way to create tail-recursive functions
    - Appel's SML/NJ compiler based on this style
- *continuation-based programming*
    - call-backs
    - programming with "*what to do next*"
- *implementation-technique for concurrency*