

# Programming Languages and Compilers (CS 421)



William Mansky

<http://courses.engr.illinois.edu/cs421/su2013/>

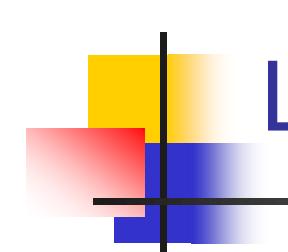
Based in part on slides by Mattox Beckman, as updated  
by Vikram Adve, Gul Agha, and Elsa Gunter



# Lambda Lifting

- Arguments to functions are evaluated immediately; function bodies are not

```
# let add_two = (+) (print_string "test\n"; 2);;
test
val add_two : int -> int = <fun>
# let add2 =      (* lambda lifted *)
    fun x -> (+) (print_string "test\n"; 2) x;;
val add2 : int -> int = <fun>
```



# Lambda Lifting

---

```
# thrice add_two 5;;
```

```
- : int = 11
```

```
# thrice add2 5;;
```

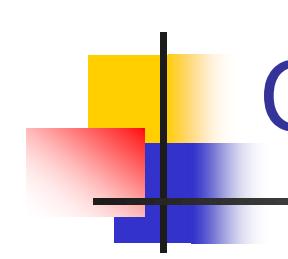
```
test
```

```
test
```

```
test
```

```
- : int = 11
```

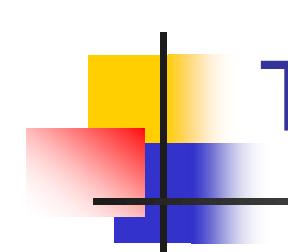
- Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied



# Continuation Passing Style

---

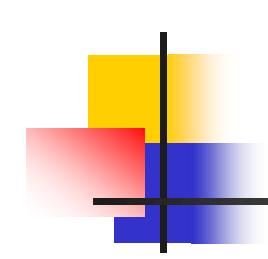
- A programming technique for all forms of “non-local” control flow:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls
- Essentially it's a higher-order version of GOTO



# Tail Calls

---

- Tail Position: A subexpression of an expression e that, if evaluated, will be returned as the value of e
  - if ( $x > 3$ ) then  $x + 2$  else  $x - 4$
  - let  $x = 5$  in  $x + 4$
- Tail Call: A function call that occurs in tail position
  - if ( $h x$ ) then  $f x$  else  $(x + g x)$



# Exercise: Tail Recursion

---

```
# let rec app fl x =
  match fl with [] -> x
  | (f :: rem_fs) -> f (app rem_fs x);;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

# Exercise: Tail Recursion

```
# let rec app fl x =
  match fl with [] -> x
  | (f :: rem_fs) -> f (app rem_fs x);;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
# let app fs x =
  let rec app_aux fl acc =
    match fl with [] -> acc
    | (f :: rem_fs) -> app_aux rem_fs (f acc)
  in app_aux fs x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

Does this work?

# Exercise: Tail Recursion

```
# let rec app fl x =
  match fl with [] -> x
  | (f :: rem_fs) -> f (app rem_fs x);;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
# let app fs x =
  let rec app_aux fl acc =
    match fl with [] -> acc
    | (f :: rem_fs) -> app_aux rem_fs
      (fun z -> acc (f z))
  in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

```
# let app fs x =
  let rec app_aux fl acc =
    match fl with [] -> acc
    | (f :: rem_fs) -> app_aux rem_fs
                           (fun z -> acc (f z))
  in app_aux fs (fun y -> y) x;;
■ app [fun x -> x * x; fun x -> x - 1] 10;;
■ f x y;; (* f(x, y) *)
```

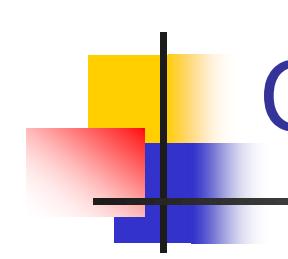
```
# let app fs x =  
  let rec app_aux fl acc =  
    match fl with [] -> acc  
    | (f :: rem_fs) -> app_aux rem_fs  
      (fun z -> acc (f z))  
  in app_aux fs (fun y -> y) x;;
```

- `app [fun x -> x * x; fun x -> x - 1] 10;;`
- `let rec app_aux fl acc = ... in app_aux [fun x -> x * x; fun x -> x - 1] (fun y -> y) 10`

```
# let app fs x =
  let rec app_aux fl acc =
    match fl with [] -> acc
    | (f :: rem_fs) -> app_aux rem_fs
                           (fun z -> acc (f z))
  in app_aux fs (fun y -> y) x;;
■ (app_aux [fun x -> x * x; fun x -> x - 1]
  (fun y -> y)) 10
■ ((app_aux [fun x -> x - 1]) (fun z -> z * z))
  10
```

```
# let app fs x =
  let rec app_aux fl acc =
    match fl with [] -> acc
    | (f :: rem_fs) -> app_aux rem_fs
                           (fun z -> acc (f z))
  in app_aux fs (fun y -> y) x;;
■ ((app_aux [fun x -> x - 1]) (fun z -> z * z))
10
■ app_aux [] (fun a -> (fun z -> z * z) ((fun x
-> x - 1) a)) 10
```

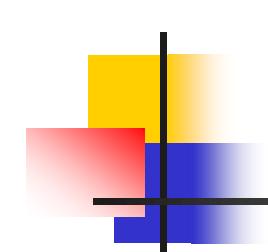
```
# let app fs x =
  let rec app_aux fl acc =
    match fl with [] -> acc
    | (f :: rem_fs) -> app_aux rem_fs
                           (fun z -> acc (f z))
  in app_aux fs (fun y -> y) x;;
■ (fun a -> (fun z -> z * z) ((fun x -> x - 1)
a)) 10
■ (fun z -> z * z) ((fun x -> x - 1) a)
in {a -> 10}
```



# Continuations

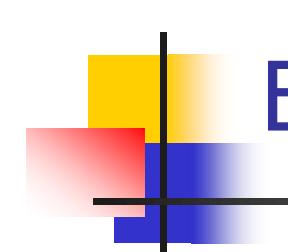
---

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an argument to which to pass its result; outer procedure “returns” no result
- Function receiving the result called a continuation
- Continuation acts as “accumulator” for work still to be done



# Example of Tail Recursion & CPS

```
# let app fs x =
  let rec app_aux fl acc=
    match fl with [] -> acc
    | (f :: rem_fs) -> app_aux rem_fs
                           (fun z -> acc (f z))
  in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
# let rec appk fl x k =
  match fl with [] -> k x
  | (f :: rem_fs) -> appk rem_fs x (fun z -> k (f z));;
val appk : ('a -> 'a) list -> 'a -> ('a -> 'b) -> 'b
```

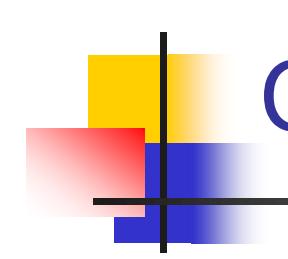


# Example of CPS

---

```
# let rec app fl x =
  match fl with [] -> x
  | (f :: rem_fs) -> f (app rem_fs x);;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

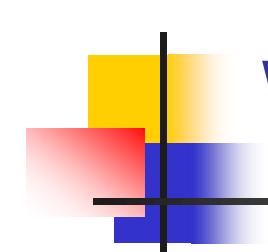
```
# let rec appk fl x k =
  match fl with [] -> k x
  | (f :: rem_fs) -> appk rem_fs x (fun r -> k (f r));;
val appk : ('a -> 'a) list -> 'a -> ('a -> 'b) -> 'b = <fun>
```



# Continuation Passing Style

---

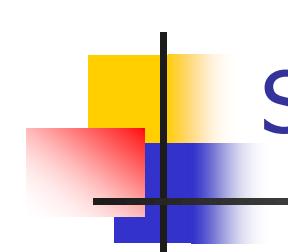
- A function is in CPS if:
- All calls are tail calls
- At tail positions, it passes its return value to either a CPS function (possibly itself) or the continuation



# Why CPS?

---

- Makes order of evaluation explicitly clear
- Allocates variables (to become registers) for each step of computation
- Essentially converts functional programs into imperative ones
  - Major step for compiling to assembly or byte code
- Tail recursion easily identified
- Strict forward recursion converted to tail recursion
- Not all functions should be written in CPS!



# Simple Functions Taking Continuations

- Given a primitive operation, can convert it to pass its result forward to a continuation
- Examples:

```
# let subk x y k = k(x + y);;
```

```
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
```

```
# let eqk x y k = k(x = y);;
```

```
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
```

```
# let timesk x y k = k(x * y);;
```

```
val timesk : int -> int -> (int -> 'a) -> 'a = <fun>
```

# Example

- Simple reporting continuation:

```
# let report x = (print_int x; print_newline () );;
val report : int -> unit = <fun>
```

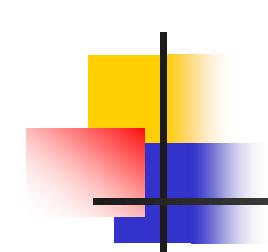
- Simple function using a continuation:

```
# let addk a b k = k (a + b);;
val addk : int -> int -> (int -> 'a) -> 'a = <fun>
```

```
# addk 20 22 report;;
```

42

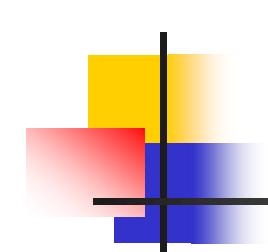
```
- : unit = ()
```



# Nesting Continuations

---

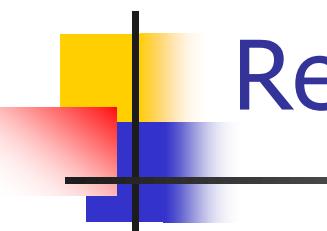
```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int = <fun>
# let add_three x y z = let p = x + y in p + z;;
val add_three : int -> int -> int = <fun>
# let add_three_k x y z k =
  addk x y (fun p -> addk p z k);;
val add_three_k : int -> int -> int -> (int -> 'a)
  -> 'a = <fun>
```



# Nesting CPS

---

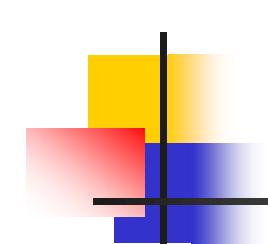
```
# let rec lengthk list k = match list with [ ] -> k 0
| x :: xs -> lengthk xs (fun r -> k (r + 1));;
val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
# let rec lengthk list k = match list with [ ] -> k 0
| x :: xs -> lengthk xs (fun r -> addk r 1 k);;
val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
# lengthk [2;4;6;8] report;;
4
- : unit = ()
```



# Recursive Functions in CPS

## ■ Recall:

```
# let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```



# Recursive Functions in CPS

- Store each intermediate value (pseudo-imperative)

```
# let rec factorial n =
```

```
  let b = (n = 0) in
```

```
    if b then 1 else
```

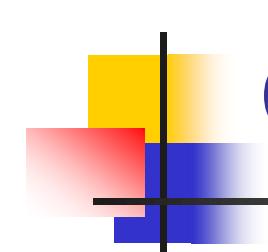
```
      let s = n - 1 in
```

```
        let r = factorial s in n * r;;
```

```
val factorial : int -> int = <fun>
```

```
# factorial 5;;
```

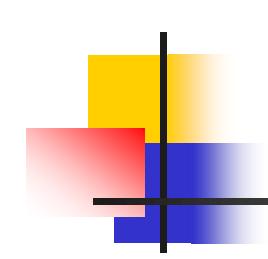
```
- : int = 120
```



# Converting Lets to Functions

---

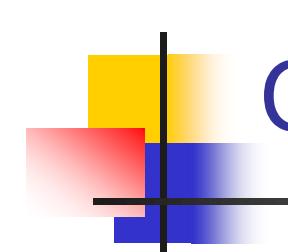
- $\text{let } x = e_1 \text{ in } e_2$
- $\text{let } f x = e_2;; f e_1$
- $\text{let } f = \text{fun } x \rightarrow e_2;; f e_1$
- $(\text{fun } x \rightarrow e_2) e_1$
- $(\text{fun } x \rightarrow e_2)$  is like a continuation, so we can pass it to the CPS version of  $e_1$ :
- $e_1 k (\text{fun } x \rightarrow e_2)$



# Recursive Functions in CPS

```
# let rec factorial n =
  let b = (n = 0) in if b then 1 else
    let s = n - 1 in
      let r = factorial s in n * r;;
val factorial : int -> int = <fun>
```

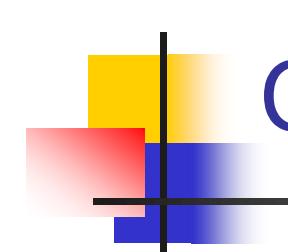
```
# let rec factoriak n k =
  eqk n 0 (fun b -> if b then k 1 else
    subk n 1 (fun s ->
      factoriak s (fun r -> timesk n r k)))
val factoriak : int -> (int -> 'a) -> 'a = <fun>
```



# CPS Transformation

---

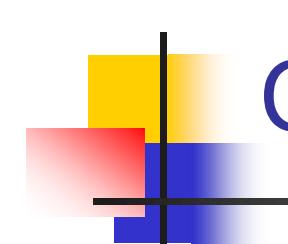
- Step 1: Add continuation argument to any function definition:
  - $\text{let } f \text{ arg} = e \Rightarrow \text{let } f \text{ arg } k = e$
  - Idea: Every function takes an extra parameter saying where the result goes
- Step 2: Name intermediate expressions by let bindings
  - Afterwards functions/match/if-then-else only applied to constants and variables
  - $\text{if } x = 0 \text{ then } e1 \text{ else } e2 \Rightarrow \text{let } b = (x=0) \text{ in }$   
 $\qquad\qquad\qquad \text{if } b \text{ then } e1 \text{ else } e2$



# CPS Transformation

---

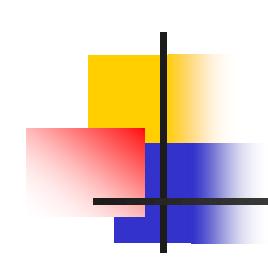
- Step 3: A simple expression in tail position should be passed to a continuation instead of returned:
  - $a \Rightarrow k\ a$
  - $a$  must be a constant or variable
  - “simple” = “no available function calls”
- Step 4: Pass the current continuation to every function call in tail position
  - $f\ arg \Rightarrow f\ arg\ k$
  - The function “isn’t going to return,” so we need to tell it where to put the result
  - Change to CPS version (e.g.,  $\text{add} \Rightarrow \text{addk}$ )



# CPS Transformation

---

- Step 5: Convert let bindings into functions
  - $\text{let } x = e_1 \text{ in } e_2 \Rightarrow (\text{fun } x \rightarrow e_2) \ e_1$
- Step 6: Pass those continuations to the appropriate arguments
  - Again, change functions into CPS versions
  - $(\text{fun } x \rightarrow e) \ (f \ a \ b) \Rightarrow fk \ a \ b \ (\text{fun } x \rightarrow e)$



# Example

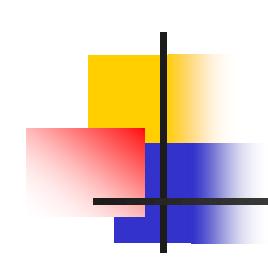
---

## Before:

```
let rec add_list lst =  
  match lst with  
    [] -> 0  
  | 0 :: xs -> add_list xs  
  | x :: xs -> (+) x  
    (add_list xs);;
```

## Step 1:

```
let rec add_listk lst k =  
  match lst with  
    [] -> 0  
  | 0 :: xs -> add_list xs  
  | x :: xs -> (+) x (add_list xs);;
```



# Example

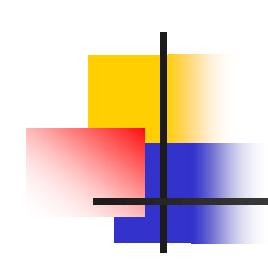
---

## Before:

```
let rec add_list lst =  
  match lst with  
    [] -> 0  
  | 0 :: xs -> add_list xs  
  | x :: xs -> (+) x  
    (add_list xs);;
```

## Step 2:

```
let rec add_listk lst k =  
  match lst with  
    [] -> 0  
  | 0 :: xs -> add_list xs  
  | x :: xs -> let r = add_list xs  
    in (+) x r;;
```



# Example

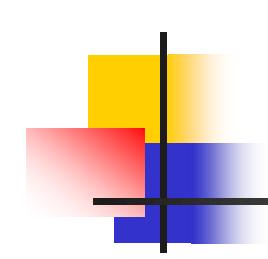
---

## Before:

```
let rec add_list lst =  
  match lst with  
    [] -> 0  
  | 0 :: xs -> add_list xs  
  | x :: xs -> (+) x  
    (add_list xs);;
```

## Step 3:

```
let rec add_listk lst k =  
  match lst with  
    [] -> k 0  
  | 0 :: xs -> add_list xs  
  | x :: xs -> let r = add_list xs  
    in (+) x r;;
```



# Example

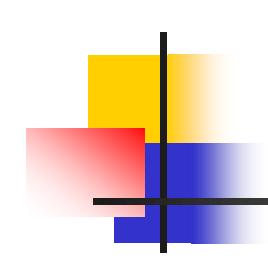
---

## Before:

```
let rec add_list lst =  
  match lst with  
    [] -> 0  
  | 0 :: xs -> add_list xs  
  | x :: xs -> (+) x  
    (add_list xs);;
```

## Step 4:

```
let rec add_listk lst k =  
  match lst with  
    [] -> k 0  
  | 0 :: xs -> add_listk xs k  
  | x :: xs -> let r = add_list xs  
    in addk x r k;;
```



# Example

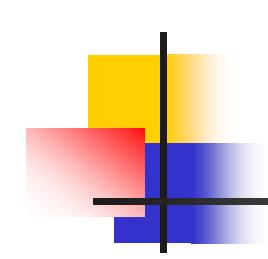
---

## Before:

```
let rec add_list lst =  
  match lst with  
    [] -> 0  
  | 0 :: xs -> add_list xs  
  | x :: xs -> (+) x  
    (add_list xs);;
```

## Step 5:

```
let rec add_listk lst k =  
  match lst with  
    [] -> k 0  
  | 0 :: xs -> add_listk xs k  
  | x :: xs -> (fun r -> addk x r k)  
    (add_listk xs);;
```



# Example

---

## Before:

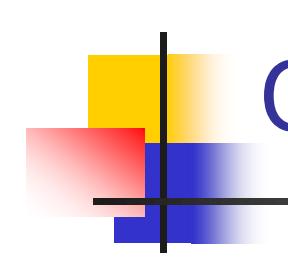
```
let rec add_list lst =  
  match lst with  
    [] -> 0  
  | 0 :: xs -> add_list xs  
  | x :: xs -> (+) x  
    (add_list xs);;
```

## Step 6:

```
let rec add_listk lst k =  
  match lst with  
    [] -> k 0  
  | 0 :: xs -> add_listk xs k  
  | x :: xs -> add_listk xs  
    (fun r -> addk x r k);;
```

# Example Execution

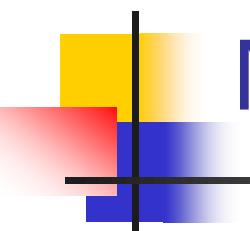
```
add_listk [1,2] k
= add_listk [2] (fun r1 -> addk 1 r1 k)
= add_listk [] (fun r2 -> addk 2 r2 k1)
= (fun r2 -> addk 2 r2 k1) 0 = addk 2 0 k1
= k1 (2+0) = (fun r1 -> addk 1 r1 k) 2
= addk 1 2 k = k (1+2) = k 3
```



# Other Uses for Continuations

---

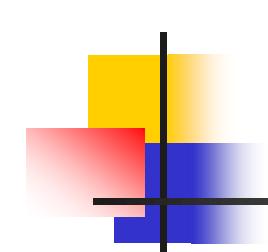
- CPS designed to preserve order of evaluation
- Continuations used to express order of evaluation
- Can be used to change order of evaluation
- Implements:
  - Exceptions and exception handling
  - Co-routines
  - (pseudo) threads



# Multiple Return Types

---

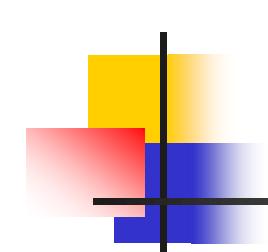
- `let smart_div x y = if y = 0. then "NaN" else x /. y;;`



# Multiple Return Types

---

- `let smart_div x y kf ks = if y = 0. then ks "NaN" else kf x /. y;;`
- `val smart_div : float -> float -> (float -> 'a) -> (string -> 'a) -> 'a`



# Multiple Return Types

---

- smart\_div 4. 2. (fun x -> print\_float x) (fun x -> print\_string x);;

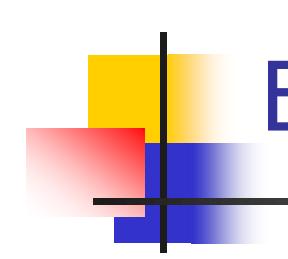
2.

- : unit = ()

- smart\_div 4. 0. (fun x -> print\_float x) (fun x -> print\_string x);;

NaN

- : unit = ()



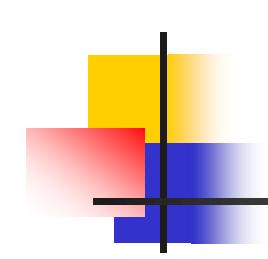
# Exceptions - Example

---

```
# exception Zero;;  
exception Zero  
# let rec list_mult_aux list =  
  match list with [ ] -> 1  
  | x :: xs ->  
    if x = 0 then raise Zero  
    else x * list_mult_aux xs;;  
val list_mult_aux : int list -> int = <fun>
```

# Exceptions - Example

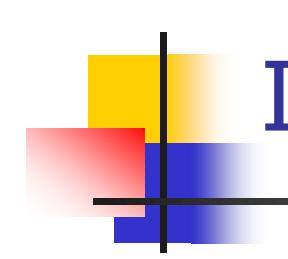
```
# let list_mult list =
  try list_mult_aux list with Zero -> 0;;
val list_mult : int list -> int = <fun>
# list_mult [3;4;2];;
- : int = 24
# list_mult [7;0;4];;
- : int = 0
# list_mult_aux [7;0;4];;
Exception: Zero.
```



# Exceptions

---

- When an exception is raised
  - The current computation is aborted
  - Control is “thrown” back up the call stack until a matching handler is found
  - All the intermediate calls waiting for a return value are thrown away

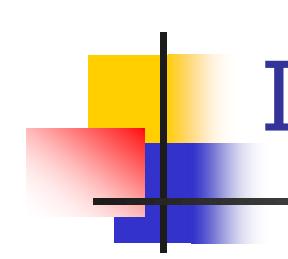


# Implementing Exceptions

---

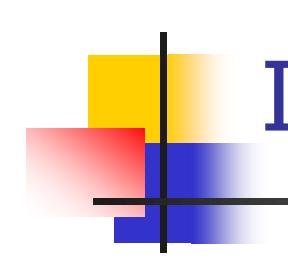
```
# let multkp m n k =
let r = m * n in
  (print_string "product result: ";
   print_int r; print_string "\n";
   k r);;
val multkp : int -> int -> (int -> 'a) -> 'a =
<fun>
```

(instrumented so we can see mult ops)



# Implementing Exceptions

```
# let rec list_multk_aux list k kexcp =
  match list with [] -> k 1
  | x :: xs -> if x = 0 then kexcp 0
    else list_multk_aux xs
      (fun r -> multkp x r k) kexcp;;
val list_multk_aux : int list -> (int -> 'a) -> (int -> 'a)
  -> 'a = <fun>
# let rec list_multk list k = list_multk_aux list k k;;
val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```



# Implementing Exceptions

---

```
# list_multk [3;4;2] report;;
```

```
product result: 2
```

```
product result: 8
```

```
product result: 24
```

```
24
```

```
- : unit = ()
```

```
# list_multk [7;4;0] report;;
```

```
0
```

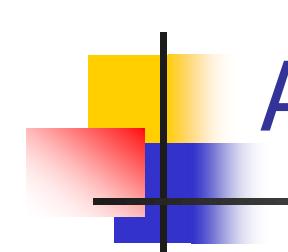
```
- : unit = ()
```

# Another CSP Example

```
let add a b k = print_string "Add "; k (a + b);;
let sub a b k = print_string "Sub "; k (a - b);;
let report n = print_string "Answer is: ";
              print_int n;
              print_newline ();;

let idk n k = k n;;

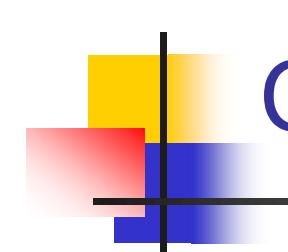
type calc = Add of int | Sub of int
```



# A Small Calculator

```
# let rec eval lst k =
  match lst with
  (Add x) :: xs -> eval xs (fun r -> add r x k)
  | (Sub x) :: xs -> eval xs (fun r -> sub r x k)
  | [] -> k 0;;
# eval [Add 20; Sub 5; Sub 7; Add 3; Sub 5]
  report;;
```

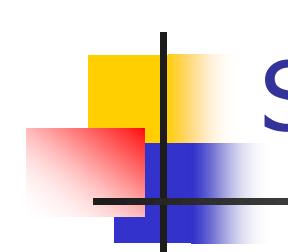
Sub Add Sub Sub Add Answer is: 6



# Composing Continuations

- Problem: Suppose we want to do all additions before any subtractions

```
let ordereval lst k =
  let rec aux lst ka ks =  match lst with
    | (Add x) :: xs -> aux xs (fun r k -> add r x ka k) ks
    | (Sub x) :: xs -> aux xs ka (fun r k -> sub r x ks k)
    | [] -> ka 0 ks k
  in
  aux lst idk idk
```

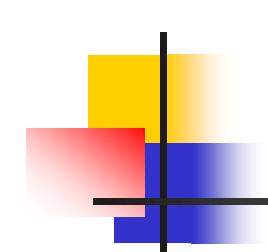


# Sample Run

---

```
# ordereval [Add 20; Sub 5; Sub 7; Add 3;  
Sub 5] report;;
```

```
Add Add Sub Sub Sub Answer is: 6
```



# Execution Trace

---

ordereval [Add 20; Sub 5; Sub 7] report

aux [Add 20; Sub 5; Sub 7] idk idk report

aux [Sub 5; Sub 7]

    (fun r1 k1 -> add 20 r1 idk k1) idk report

aux [Sub 7] (fun r1 k1 -> add r1 20 idk k1)

        (fun r2 k2 -> sub r2 5 idk k2) report

aux [] (fun r1 k1 -> add r1 20 idk k1)

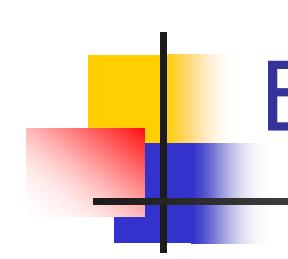
    (fun r3 k3 -> sub r3 7

        (fun r2 k2 -> sub r2 5 idk k2) k3)

report

# Execution Trace

```
aux [] (fun r1 k1 -> add r1 20 idk k1)
        (fun r3 k3 -> sub r3 7
            (fun r2 k2 -> sub r2 5 idk k2) k3)
        report
(* Start calling the continuations *)
(fun r1 k1 -> add r1 20 idk k1)
0
(fun r3 k3 -> sub r3 7
    (fun r2 k2 -> sub r2 5 idk k2) k3)
report
```



# Execution Trace

---

(fun r1 k1 -> add r1 20 idk k1)

0

(fun r3 k3 -> sub r3 7

    (fun r2 k2 -> sub r2 5 idk k2) k3)

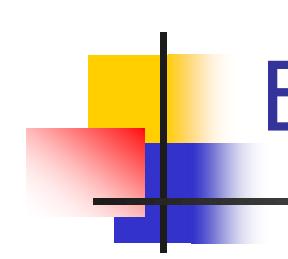
report

add 0 20 idk (\* remember idk n k = k n \*)

(fun r3 k3 -> sub r3 7

    (fun r2 k2 -> sub r2 5 idk k2) k3)

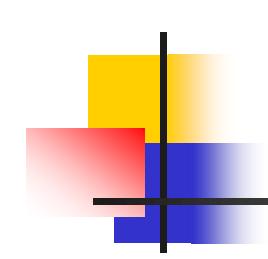
report



# Execution Trace

---

```
add 0 20 idk (* remember idk n k = k n *)
  (fun r3 k3 -> sub r3 7
    (fun r2 k2 -> sub r2 5 idk k2) k3)
report
idk 20
  (fun r3 k3 -> sub r3 7
    (fun r2 k2 -> sub r2 5 idk k2) k3)
report
```



# Execution Trace

idk 20

(fun r3 k3 -> sub r3 7  
                  (fun r2 k2 -> sub r2 5 idk k2) k3)

report

(fun r3 k3 -> sub r3 7 (fun r2 k2 -> sub r2 5 idk k2) k3)

20 report

sub 20 7 (fun r2 k2 -> sub r2 5 idk k2) report

(fun r2 k2 -> sub r2 5 idk k2) 13 report

sub 13 5 idk report

idk 8 report ---> report 8