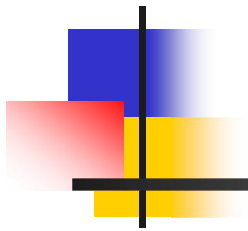


# Programming Languages and Compilers (CS 421)



Reza Zamani

<http://www.cs.uiuc.edu/class/cs421/>

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



# Folding Functions over Lists

---

How are the following functions similar?

```
# let rec sumlist list = match list with  
  [ ] -> 0 | x::xs -> x + sumlist xs;;
```

```
val sumlist : int list -> int = <fun>
```

```
# sumlist [2;3;4];;
```

```
- : int = 9
```

```
# let rec prodlist list = match list with  
  [ ] -> 1 | x::xs -> x * prodlist xs;;
```

```
val prodlist : int list -> int = <fun>
```

```
# prodlist [2;3;4];;
```

```
- : int = 24
```



# Folding

---

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

```
fold_left f a [x1; x2;...;xn] = f(...(f (f a x1) x2)...)xn
```

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

```
fold_right f [x1; x2;...;xn] b = f x1(f x2 (...(f xn b)...))
```



## Folding - Forward Recursion

---

```
# let sumlist list = fold_right (+) list 0;;  
val sumlist : int list -> int = <fun>  
# sumlist [2;3;4];;  
- : int = 9  
# let prodlist list = fold_right ( * ) list 1;;  
val prodlist : int list -> int = <fun>  
# prodlist [2;3;4];;  
- : int = 24
```



# Folding - Tail Recursion

---

```
- # let rev list =  
-     fold_left  
-     (fun l -> fun x -> x :: l)    //comb op  
-     []                            //accumulator cell  
-     list
```



# Folding

---

- n Can replace recursion by `fold_right` in any forward primitive recursive definition
  - n Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- n Can replace recursion by `fold_left` in any tail primitive recursive definition



# Encoding Recursion with Fold

```
# let rec append list1 list2 = match list1 with  
  [ ] -> list2 | x::xs -> x :: append xs list2;;  
val append : 'a list -> 'a list -> 'a list = <fun>
```

Base Case

Operation

Recursive Call

```
# let append list1 list2 =  
  fold_right (fun x y -> x :: y) list1 list2;;  
val append : 'a list -> 'a list -> 'a list = <fun>  
# append [1;2;3] [4;5;6];;  
- : int list = [1; 2; 3; 4; 5; 6]
```



# Mapping

n What do these functions have in common?

```
# let rec inclist list = match list with [ ] -> [ ]  
  | x :: xs -> (1 + x) :: inclist xs;;
```

```
val inclist : int list -> int list = <fun>
```

```
# inclist [2;3;4];;
```

```
- : int list = [3; 4; 5]
```

```
# let rec doublelist list = match list with [ ] -> [ ]  
  | x :: xs -> (2 * x) :: doublelist xs;;
```

```
val doublelist : int list -> int list = <fun>
```

```
# doublelist [2;3;4];;
```

```
- : int list = [4; 6; 8]
```





## Recall Map

---

```
# let rec map f list =  
  match list  
  with [] -> []  
       | (h::t) -> (f h) :: (map f t);;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# map plus_two fib5;;  
- : int list = [10; 7; 5; 4; 3; 3]
```

n Same as List.map



# Mapping

---

```
# let inclist = map ((+) 1);;  
val inclist : int list -> int list = <fun>  
# inclist [2;3;4];;  
- : int list = [3; 4; 5]  
# let doublelist = map (( * ) 2);;  
val doublelist : int list -> int list = <fun>  
# doublelist [2;3;4];;  
- : int list = [4; 6; 8]
```



# Map from Fold

---

```
# let map f list =  
  fold_right (fun x y -> f x :: y) list [ ];;  
val map : ('a -> 'b) -> 'a list -> 'b list =  
  <fun>  
# map ((+)1) [1;2;3];;  
- : int list = [2; 3; 4]
```

n Can you write fold\_right (or fold\_left) with just map? How, or why not?



# Higher Order Functions

n A function is *higher-order* if it takes a function as an argument or returns one as a result

n Example:

```
# let compose f g = fun x -> f (g x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

n The type `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b` is a higher order type because of `('a -> 'b)` and `('c -> 'a)` and `-> 'c -> 'b`



# Thrice

---

**n Recall:**

```
# let thrice f x = f (f (f x));;
```

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

**n How do you write thrice with compose?**



# Thrice

---

**n Recall:**

```
# let thrice f x = f (f (f x));;
```

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

**n How do you write thrice with compose?**

```
# let thrice f = compose f (compose f f);;
```

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



# Partial Application

---

```
# (+);;
```

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

```
# (+) 2 3;;
```

```
- : int = 5
```

```
# let plus_two = (+) 2;;
```

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

```
# plus_two 7;;
```

```
- : int = 9
```

n Partial application also called *sectioning*



# Lambda Lifting

---

n You must remember the rules for evaluation when you use partial application

```
# 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
```

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



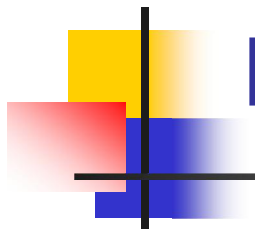
# Lambda Lifting

---

Lambda Lifting is the process of eliminating free variables from a function.

## Algorithm for Lambda Lifting

1. Rename function so that each function has a unique name.
2. Replace free variable with an additional argument.
3. Replace every local function that has no free variables with an identical global function.
4. Repeat steps 2 and 3 until all free variables are eliminated.



# Lambda Lifting: Example

---

Consider the OCaml code.

```
# let rec sum n =  
  if n=1 then 1  
  else  
    let f x = n + x  
    in f (sum (n-1)) in  
  sum 100;;
```

Sum of integers  
from 1 to 100



## Lambda Lifting: Step 1 and 2

---

```
# let rec sum n =  
  if n=1 then 1  
  else  
    let f x =  
      fun w -> w + x  
    in f (sum (n-1)) n in  
sum 100;;
```



## Lambda Lifting: Step 3

---

```
# let rec f x =  
    fun w -> w + x  
and sum n =  
    if n=1 then 1  
    else  
        f (sum (n-1)) n in  
sum 100;;
```



## Partial Application and “Unknown Types”

n Recall `compose plus_two`:

```
# let f1 = compose plus_two;;
```

```
val f1 : ('_a -> int) -> '_a -> int = <fun>
```

n Compare to lambda lifted version:

```
# let f2 = fun g -> compose plus_two g;;
```

```
val f2 : ('a -> int) -> 'a -> int = <fun>
```

n What is the difference?



## Partial Application and “Unknown Types”

`n ' _a can only be instantiated once for an expression`

```
# f1 plus_two;;
```

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

```
# f1 List.length;;
```

Characters 3-14:

```
f1 List.length;;
```

```
^ ^ ^ ^ ^ ^ ^ ^ ^ ^
```

This expression has type 'a list -> int but is here used  
with type int -> int



## Partial Application and “Unknown Types”

---

n 'a can be repeatedly instantiated

```
# f2 plus_two;;
```

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

```
# f2 List.length;;
```

```
- : 'a list -> int = <fun>
```





# Continuations

---

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



# Example of Tail Recursion

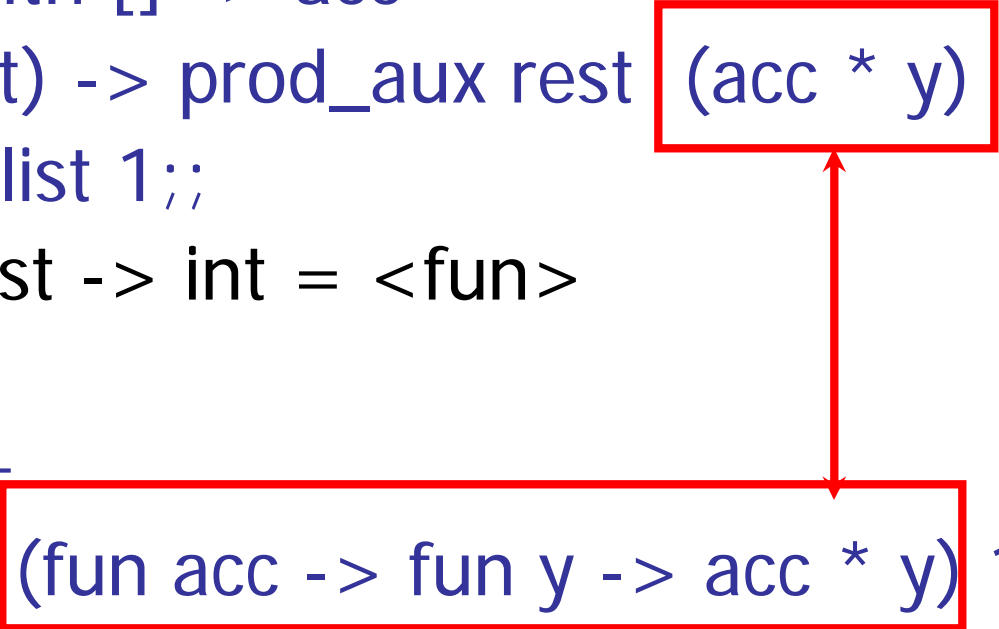
---

```
# let rec prod l =  
  match l with [] -> 1  
  | (x :: rem) -> x * prod rem;;  
val prod : int list -> int = <fun>  
  
# let prod list =  
  let rec prod_aux l acc =  
    match l with [] -> acc  
    | (y :: rest) -> prod_aux rest (acc * y)  
  (* Uses associativity of multiplication *)  
  in prod_aux list 1;;  
val prod : int list -> int = <fun>
```



# Example of Tail Recursion

```
# let prod list =  
  let rec prod_aux l acc =  
    match l with [] -> acc  
    | (y :: rest) -> prod_aux rest (acc * y)  
  in prod_aux list 1;;  
val prod : int list -> int = <fun>
```



```
# let prod list =  
  List.fold_left (fun acc -> fun y -> acc * y) 1 list;;  
val prod : int list -> int = <fun>
```



# Example of 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>
```



# Continuation Passing Style

---

- n Writing procedures so that they take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)



# Example of Tail Recursion & CSP

---

```
# 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 CSP

---

```
# 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 z -> k (f z));;  
val appk : ('a -> 'a) list -> 'a -> ('a -> 'b) -> 'b =  
  <fun>
```



# Continuation Passing Style

---

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





# Continuation Passing Style

---

- n A compilation technique to implement non-local control flow, especially useful in interpreters.
- n A formalization of non-local control flow in denotational semantics



# Terms

---

- n A function is in Direct Style when it returns its result back to the caller.
- n A Tail Call occurs when a function returns the result of another function call without any more computations (eg tail recursion)
- n A function is in Continuation Passing Style when it passes its result to another function.
- n Instead of returning the result to the caller, we pass it forward to another function.



## Example

n Simple reporting continuation:

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

n Simple function using a continuation:

```
# let plusk a b k = k (a + b)  
val plusk : int -> int -> (int -> 'a) -> 'a = <fun>  
# plusk 20 22 report;;  
42  
- : unit = ()
```



# Recursive Functions

---

**n Recall:**

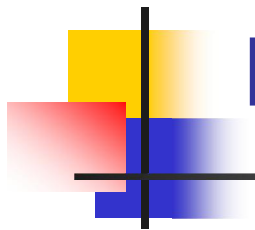
```
# let rec factorial n =
```

```
    if n = 0 then 1 else n * factorial (n - 1);;
```

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

```
# factorial 5;;
```

```
- : int = 120
```



# Recursive Functions

---

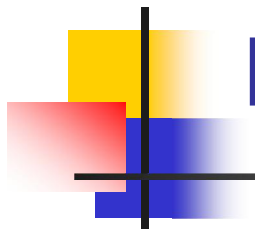
```
# let rec factorial n =  
  if n = 0 then 1 (* Returned value *)  
  else  
    let r = factorial (n - 1) in  
    let a = n * r in  
    a (* Returned value *) ;;  
val factorial : int -> int = <fun>  
# factorial 5;;  
- : int = 120
```



# Recursive Functions

---

```
# let rec factorialk n k =  
  if n = 0 then k 1 (* Passed value *)  
  else  
    let k1 r =  
      let a = n * r in k a (* Passed value  
*) in  
    factorialk (n - 1) k1 ;;  
val factorialk : int -> int = <fun>  
# factorialk 5 report;;  
120  
- : unit = ()
```



# Recursion Functions

---

```
# let rec factorialk n k =  
  if n = 0 then k 1 else factorialk (n - 1)  
  (fun r -> k (n * r));;  
val factorialk : int -> (int -> 'a) -> 'a =  
  <fun>  
# factorialk 5 report;;  
120  
- : unit = ()
```



# Recursive Functions

---

- n Notice: factorialk is now tail recursive
- n To make recursive call, must build intermediate continuation to
  - n take recursive value:  $m$
  - n build it to final result:  $n * m$
  - n And pass it to final continuation:
    - n  $k (n * m)$





# 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 -> plusk r 1 k);;
```

```
val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
```

```
# lengthk [2;4;6;8] report;;
```

```
4
```

```
- : 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;4;0];;  
- : int = 0  
# list_mult_aux [7;4;0];;  
Exception: Zero.
```



# Exceptions

---

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



# Implementing Exceptions

---

```
# let multkp m n k =
```

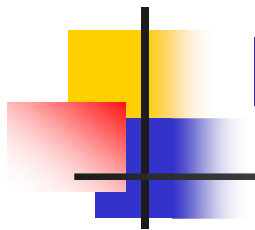
```
  let r = m * n in
```

```
    (print_string "product result: ";
```

```
      print_int r; print_string "\n";
```

```
      k r);;
```

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



# 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 = ()
```



# Terminology

---

n Tail Position: A subexpression  $s$  of expressions  $e$ , if it is evaluated, will be taken as the value of  $e$

n if  $(x > 3)$  then  $x + 2$  else  $x - 4$

n let  $x = 5$  in  $x + 4$

n Tail Call: A function call that occurs in tail position

n if  $(h\ x)$  then  $f\ x$  else  $(x + g\ x)$





# Terminology

---

- n Available: A function call that can be executed by the current expression
- n The fastest way to be unavailable is to be guarded by an abstraction (anonymous function).
  - n `if (h x) then f x else (x + g x)`
  - n `if (h x) then (fun x -> f x) else (g (x + x))`



# CPS Transformation

---

- n Step 1: Add continuation argument to any function definition:
  - n  $\text{let } f \text{ arg} = e \Rightarrow \text{let } f \text{ arg } k = e$
  - n Idea: Every function takes an extra parameter saying where the result goes
- n Step 2: A simple expression in tail position should be passed to a continuation instead of returned:
  - n  $\text{return } a \Rightarrow k \ a$
  - n Assuming  $a$  is a constant or variable.
  - n "Simple" = "No available function calls."



# CPS Transformation

---

- n Step 3: Pass the current continuation to every function call in tail position
  - n  $\text{return } f \text{ arg} \Rightarrow f \text{ arg } k$
  - n The function “isn’t going to return,” so we need to tell it where to put the result.
  
- n Step 4: Each function call not in tail position needs to be built into a new continuation (containing the old continuation as appropriate)
  - n  $\text{return op (f arg)} \Rightarrow f \text{ arg (fun r -> k(op r))}$
  - n op represents a primitive operation



# Example

---

## Before:

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

## After:

```
let rec add_listk lst k =  
  (* rule 1 *)  
  match lst with  
    | [ ] -> k 0 (* rule 2 *)  
    | 0 :: xs -> add_listk xs k  
      (* rule 3 *)  
    | x :: xs -> add_listk xs  
      (fun r -> k ((+) x r));;  
  (* rule 4 *)
```

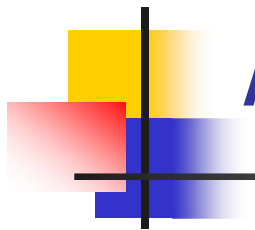


## Continuations Example

---

```
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



## Continuations Can Take Multiple Arguments

---

```
# add 3 5 (fun r -> sub r 2 report);;
```

Add Sub Answer is: 6

```
# add 3 5 (fun r k -> sub r 2 k);;
```

Add - : (int -> 'a) -> 'a = <fun>

```
# add 3 5 ((fun k r -> sub r 2 k) report);;
```

Add Sub Answer is: 6



# Composing Continuations

**n** 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



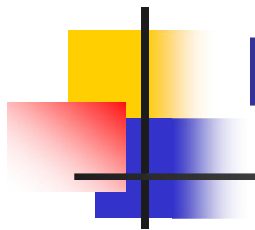


## Sample Run

---

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

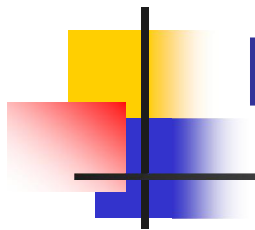
```
Add Add Sub Sub Sub Answer is: 6
```



## 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)
```

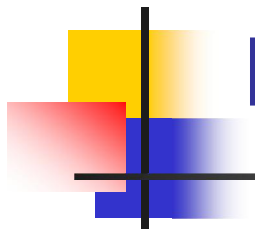
```
report
```

```
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)
```

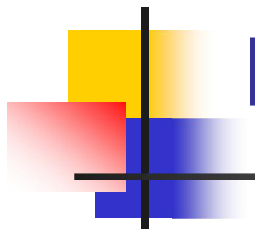
```
report
```



# 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