Your turn: num_neg – tail recursive

# let num_neg list =

let rec num_neg_aux list curr_neg =
    in num_neg_aux  ?  ?

Your turn: num_neg – tail recursive

# let num_neg list =

let rec num_neg_aux list curr_neg =
    match list with [] -> curr_neg
      | (x :: xs) ->
    in num_neg_aux  ?  ?

Your turn: num_neg – tail recursive

# let num_neg list =

let rec num_neg_aux list curr_neg =
    match list with [] -> curr_neg
      | (x :: xs) ->
        num_neg_aux xs  ?
    in num_neg_aux  ?  ?
Your turn: num_neg – tail recursive

```ocaml
# let num_neg list =
  let rec num_neg_aux list curr_neg =
    match list with [] -> curr_neg
     | (x :: xs) ->
       num_neg_aux xs
       (if x < 0 then 1 + curr_neg
        else curr_neg)
    in num_neg_aux ? ?
```

Tail Recursion - length

```
How can we write length with tail recursion?

let length list =
  let rec length_aux list acc_length =
    match list with [] -> acc_length
     | (x :: xs) ->
       length_aux xs (1 + acc_length)
    in length_aux list 0
```

length, fold_left

```
let length list =
  fold_left
  (fun acc -> fun x -> 1 + acc) // comb op
  0 // initial accumulator cell value
  list
```

Your turn: num_neg, fold_left

```
let num_neg list =
  fold_left
  ? // comb op
  ? // initial accumulator cell value
  ?
```
Your turn: num_neg, fold_left

let num_neg list = fold_left
  ? // comb op
  0 // initial accumulator cell value

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 list -> 'a = <fun>
fold_left f a [x_1; x_2;...;x_n] = f(...(f (f a x_1) x_2)...x_n)

# 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 -> 'a) -> 'a list -> 'b = <fun>
fold_right f [x_1; x_2;...;x_n] b = f x_1(f x_2(...(f x_n b)...)b)

Continuations

- 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 function version of GOTO
Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an extra 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

Continuation Passing Style

- Writing procedures such that all procedure calls take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)

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 (and forward recursion) easily identified

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, aka green) threads

Example

- Simple reporting continuation:
  ```ocaml
  # let report x = (print_int x; print_newline( ) );;
  val report : int -> unit = <fun>
  # addk (22, 20) report;;
  42 - : unit = ()
  ```

- Simple function using a continuation:
  ```ocaml
  # let addk (a, b) k = k (a + b);;
  val addk : int * int -> (int -> 'a) -> 'a = <fun>
  # addk (22, 20) report;;
  42
  ```
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>
  `

Nesting Continuations

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

add_three: a different order

- `# let add_triple (x, y, z) = x + (y + z);;
  `
- How do we write `add_triple_k` to use a different order?
- `let add_triple_k (x, y, z) k =`

Recursive Functions

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

Terms

- A function is in **Direct Style** when it returns its result back to the caller.
- A function is in **Continuation Passing Style** when it, and every function call in it, passes its result to another function.
- Instead of returning the result to the caller, we pass it forward to another function giving the computation after the call.
Recursive Functions

# let rec factorial n = let b = (n = 0) in (* First computation *) if b then 1 (* Returned value *) else let s = n - 1 in (* Second computation *) let r = factorial s in (* Third computation *) n * r (* Returned value *);; val factorial : int -> int = <fun>

# factorial 5;; - : int = 120

Recursive Functions

# let rec factorialk n k = eqk (n, 0) (fun b ->  (* First computation *) if b then k 1 (* Passed value *) else subk (n, 1) (fun s -> factorialk s) (fun r -> timesk (n, r) k)) (* Passed value *) val factorialk : int -> (int -> 'a) -> 'a = <fun>

# factorialk 5 report;; - : unit = ()

Example: CPS for length

let rec length list = match list with [] -> 0 | (a :: bs) -> 1 + length bs

What is the let-expanded version of this?