Recall

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

What is its running time?

Comparison

- poor_rev [1,2,3] =
- (poor_rev [2,3]) @ [1] =
- ((poor_rev [ ]) @ [3]) @ [2]) @ [1] =
- (([ ] @ [3]) @ [2]) @ [1]) =
- ([3] @ [2]) @ [1] =
- (3:: ([ ] @ [2])) @ [1] =
- [3,2] @ [1] =
- 3 :: ([ ] @ [1])) = [3,2,1]

Tail Recursion - Example

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

```ocaml
# let rec rev list = rev_aux list [];
val rev : 'a list -> 'a list = <fun>
```

What is its running time?

Your turn now

Write a function

```ocaml
map_tail : ('a -> 'b) -> 'a list -> 'b list
```

that takes a function and a list of inputs and gives the result of applying the function on each argument, but in tail recursive form.

```ocaml
let map_tail f lst =
```
Folding - Tail Recursion

# let rec rev_aux list revlist =
match list with
  [ ] -> revlist
| x :: xs -> rev_aux xs (x::revlist);;
# let rev list = rev_aux list [ ];;

Folding

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

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>

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

Simplest CPS Example

Identity function:

- let ident x = x

Identity function in CPS:

- let identk x ret = ret x
Example

- 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
- : unit = ()
```

- Simple reporting continuation:

```ocaml
# let report x = (print_int x; print_newline(); exit 0);
val report : int -> unit = <fun>
```

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 easily identified
- Strict forward recursion converted to tail recursion
  - At the expense of building large closures in heap

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(); exit 0);
val report : int -> unit = <fun>
```

- 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
- : unit = ()
```
Simple Functions Taking Continuations

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

```ocaml
# 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_three : 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 -> ('a -> 'a) = <fun>

Terms

- A function is in **Direct Style** when it returns its result back to the caller.
- A **Tail Call** occurs when a function returns the result of another function call without any more computations (e.g., tail recursion).
- 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.

Terminology

- **Tail Position:** A subexpression \( s \) of expressions \( e \), such that if evaluated, will be taken 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 \( x \) else \( x + g x \)

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
Recursive Functions

```ocaml
# let rec factorial n = 
  if n = 0 then 1 else n * factorial (n - 1);;

# let rec factorial n = 
  let b = (n = 0) in 
  if b then 1 (* Returned value *) 
  else let s = n - 1 in (* 2nd computation *) 
  let r = factorial s in (* 3rd computation *) 
  n * r (* Returned value *);;

val factorial : int -> int = <fun>

# factorial 5;;
- : int = 120
```

Example: CPS for length

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

What is the let-expanded version of this?

let rec length list = match list with 
  [] -> 0 
  | (a :: bs) -> let r1 = length bs in 
    1 + r1
```

Example: CPS for length

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

What is the CPS version of this?
```
Example: CPS for length

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

What is the CPS version of this?

```ocaml
# let rec lengthk list k = match list with
  | [] -> k 0
  | x :: xs -> lengthk xs (fun r -> addk (r,1) k)
# lengthk [2;4;6;8] report
4
```

Order of Evaluation Matters!

- Your turn (MP2): Write a function `quaddk` that takes three integer arguments, `a`, `b`, and `c`, and "returns" the result of the expression \((2*a*b + 5*b + c)\)
- # let quaddk (a, b, c) k = ... ;;

- Is the CPS form the same as for \(2*a*b + 5*b + c\) ?
  - Refresher: Eval/App slides & Madhu’s notes posted on Piazza
  - MP2 solutions implement the order of evaluation of arithmetic operators from right to left

CPS for Higher Order Functions

- In CPS, every procedure / function takes a continuation to receive its result
- Procedures passed as arguments take continuations
- Procedures returned as results take continuations
- CPS version of higher-order functions must expect input procedures to take continuations

Example: all

```ocaml
# let rec all (p, l) = match l with
  | [] -> true
  | (x :: xs) -> let b = p x in
    if b then all (p, xs) else false
val all : ('a -> bool) -> 'a list -> bool = <fun>
```

What is the CPS version of this?

```ocaml
# let rec allk (pk, l) k = match l with
  | [] -> k true
  | x :: xs -> allk (pk, xs) (fun r -> addk (r,1) k)
```

Example: all

```ocaml
# let rec all (p, l) = match l with
  | [] -> true
  | (x :: xs) -> let b = p x in
    if b then all (p, xs) else false
val all : ('a -> bool) -> 'a list -> bool = <fun>
```

- What is the CPS version of this?
- # let rec allk (pk, l) k = match l with
  | [] -> k true
```
Example: all

```ocaml
#let rec all (p, l) = match l with [] -> true
  | (x :: xs) -> let b = p x in
    if b then all (p, xs) else false
```

What is the CPS version of this?

```ocaml
#let rec allk (pk, l) k = match l with
  [] -> k true
  | (x :: xs) ->
    pk x (fun b -> if b then allk (pk, xs) k else )
```

Example: all

```ocaml
#let rec all (p, l) = match l with [] -> true
  | (x :: xs) -> let b = p x in
    if b then all (p, xs) else false
```

What is the CPS version of this?

```ocaml
#let rec allk (pk, l) k = match l with
  [] -> k true
  | (x :: xs) ->
    pk x (fun b -> if b then allk (pk, xs) k else )
```

Example: all

```ocaml
#let rec all (p, l) = match l with [] -> true
  | (x :: xs) -> let b = p x in
    if b then all (p, xs) else false
```

What is the CPS version of this?

```ocaml
#let rec allk (pk, l) k = match l with
  [] -> k true
  | (x :: xs) ->
    pk x (fun b -> if b then allk (pk, xs) k else )
```

Example: all

```ocaml
#let rec all (p, l) = match l with [] -> true
  | (x :: xs) -> let b = p x in
    if b then all (p, xs) else false
```

What is the CPS version of this?

```ocaml
#let rec allk (pk, l) k = match l with
  [] -> k true
  | (x :: xs) ->
    pk x (fun b -> if b then allk (pk, xs) k else )
```
Terms

- A function is in **Direct Style** when it returns its result back to the caller.
- A **Tail Call** occurs when a function returns the result of another function call without any more computations (eg tail recursion)
- 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.

Terminology

- **Tail Position**: A subexpression \( s \) of expressions \( e \), such that if evaluated, will be taken 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 \((x + g \ x)\)

Terminology

- **Available**: A function call that can be executed by the current expression
- The fastest way to be unavailable is to be guarded by an abstraction (anonymous function, lambda lifted).
  - if \((h \ x)\) then \(f \ x\) else \((x + g \ x)\)
  - if \((h \ x)\) then \((\text{fun} \ x \rightarrow f \ x)\) else \((g \ (x + x))\)

CPS Transformation

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

CPS Transformation

- **Step 3**: Pass the current continuation to every function call in tail position
  - return \(f \ arg\) \Rightarrow \(f \ arg\) \(k\)
  - The function "isn’t going to return," so we need to tell it where to put the result.

CPS Transformation

- **Step 4**: Each function call not in tail position needs to be converted to take a new continuation (containing the old continuation as appropriate)
  - return \(op \ (f \ arg)\) \Rightarrow \(f \ arg\) \((\text{fun} \ r \rightarrow k(\text{op} \ r))\)
  - \(op\) represents a primitive operation
  - return \(f(g \ arg)\) \Rightarrow \(g \ arg\) \((\text{fun} \ r \rightarrow f \ r \ k)\)
Example

Step 1: Add continuation argument to any function definition
Step 2: A simple expression in tail position should be passed to a continuation instead of returned
Step 3: Pass the current continuation to every function call in tail position
Step 4: Each function call not in tail position needs to be converted to take a new continuation (containing the old continuation as appropriate)

Before:

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

After:

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

Step 1: Add continuation argument to any function definition
Step 2: A simple expression in tail position should be passed to a continuation instead of returned
Step 3: Pass the current continuation to every function call in tail position
Step 4: Each function call not in tail position needs to be converted to take a new continuation (containing the old continuation as appropriate)

CPS for sum

```ocaml
# let rec sum list = match list with
  | [] -> 0
  | x :: xs -> x + sum xs ;;
val sum : int list -> int = <fun>
```

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

```ocaml
# let rec sum list = match list with
  | [] -> 0
  | x :: xs -> let r1 = sum xs \in x + r1;;
```

```ocaml
# let rec sumk list k = match list with
  | [] -> k 0
  | x :: xs -> sumk xs (fun r1 -> addk x r1 k);
```

Other Uses for Continuations

- CPS designed to preserve evaluation order
- Continuations used to express order of evaluation
- Can also be used to change order of evaluation
- Implements:
  - Exceptions and exception handling
  - Co-routines
  - (pseudo, aka green) threads
**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 values are thrown away

**Implementing Exceptions**

```ocaml
# let rec list_multk_list k = list_multk_list k
  ```