Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion form of Structural Recursion
- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
- Wait until whole structure has been traversed to start building answer
Forward Recursion: Examples

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

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

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

Base Case      Operator      Recursive Call

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

Base Case      Operator      Recursive Call
Encoding Forward Recursion with Fold

``` Ocaml
# 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]
```
Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list = List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```
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
An expression is in **continuation passing style (CPS)** if every procedure call in it that is not directly a call to a continuation takes a continuation to which to give (pass) the result, and it returns no result (except the unknown ultimate result of the final continuation).
Recursive Functions

Recall:

```ocaml
# 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 =
  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) (* Second computation *)
   (fun s -> factorialk s (* Third computation *)
    (fun r -> timesk (n, r) k)))
val factorialk : int -> (int -> 'a) -> 'a = <fun>
# factorialk 5 report;;
120
- : unit = ()
Recursive Functions

To make recursive call, must build intermediate continuation to

- take recursive value: $r$
- build it to final result: $n \times r$
- And pass it to final continuation:
  - $\text{times } (n, r) \; k = k \; (n \times r)$
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?
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?

```
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) -> let r1 = length bs in 1 + r1
```

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

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

What is the CSP version of this?
# 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 = ()
```

9/14/21
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?
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 = 
```
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 [] -> 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
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
```
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) ->
```
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) -> pk x
```
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) -> pk x
    (fun b -> if b then 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 
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 
  | (x :: xs) -> pk x 
    (fun b -> if b then allk (pk, xs) k else k false) 
val allk : ('a -> (bool -> 'b) -> 'b) * 'a list -> 
  (bool -> 'b) -> 'b = <fun>
```
CPS for sum

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

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

# let rec sum list = match list with [] -> 0
    | x :: xs -> let r1 = sum xs in x + r1;;
```
# let rec sum list = match list with [ ] -> 0
    | x :: xs -> x + sum xs ;;
val sum : int list -> int = <fun>

# let rec sum list = match list with [ ] -> 0
    | x :: xs -> let r1 = sum xs in x + r1;;
val sum : int list -> int = <fun>

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

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

val sum : int list -> int = <fun>

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

val sum : int list -> int = <fun>

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

val sumk : int list -> (int -> 'a) -> 'a = <fun>

# sumk [2;4;6;8] report

20

- : unit = ()
```
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 \((g \,(x + x))\)

Not available
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.
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) ⇒ f arg (fun r -> k(op r))
- op represents a primitive operation

- return f(g arg) ⇒ g arg (fun r-> f r k)
Example

**Before:**

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

**After:**

```ocaml
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 *)
;;
```
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
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

```ocaml
# 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

- 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

# 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 = ()
Variants - Syntax (slightly simplified)

- type \( \text{name} = C_1 \text{[of} \ ty_1]\ | \ldots | \ C_n \text{[of} \ ty_n] \)
- Introduce a type called \( \text{name} \)
- \( \text{(fun} \ x \Rightarrow C_i x) : ty_1 \rightarrow \text{name} \)
- \( C_i \) is called a \textit{constructor}; if the optional type argument is omitted, it is called a \textit{constant}
- Constructors are the basis of almost all pattern matching
Enumeration Types as Variants

An enumeration type is a collection of distinct values

In C and Ocaml they have an order structure; order by order of input
Enumeration Types as Variants

```haskell
# type weekday = Monday | Tuesday | Wednesday
   | Thursday | Friday  | Saturday | Sunday;

type weekday =
    Monday
   | Tuesday
   | Wednesday
   | Thursday
   | Friday
   | Saturday
   | Sunday
```
Functions over Enumerations

# let day_after day = match day with
    Monday -> Tuesday
| Tuesday -> Wednesday
| Wednesday -> Thursday
| Thursday -> Friday
| Friday -> Saturday
| Saturday -> Sunday
| Sunday -> Monday;;

val day_after : weekday -> weekday = <fun>
Functions over Enumerations

# let rec days_later n day =
match n with 0 -> day
| _ -> if n > 0
  then day_after (days_later (n - 1) day)
else days_later (n + 7) day;;

val days_later : int -> weekday -> weekday
= <fun>
Functions over Enumerations

# days_later 2 Tuesday;;
- : weekday = Thursday

# days_later (-1) Wednesday;;
- : weekday = Tuesday

# days_later (-4) Monday;;
- : weekday = Thursday
Disjoint Union Types

- Disjoint union of types, with some possibly occurring more than once

- We can also add in some new singleton elements
Disjoint Union Types

# type id = DriversLicense of int | SocialSecurity of int | Name of string

let check_id id = match id with
  | DriversLicense num -> not (List.mem num [13570; 99999])
  | SocialSecurity num -> num < 900000000
  | Name str -> not (str = "John Doe")

val check_id : id -> bool = <fun>
Polymorphism in Variants

- The type 'a option is gives us something to represent non-existence or failure

```
# type 'a option = Some of 'a | None;;
type 'a option = Some of 'a | None
```

- Used to encode partial functions
- Often can replace the raising of an exception
Functions producing option

# let rec first p list =
    match list with [ ] -> None
    | (x::xs) -> if p x then Some x else first p xs;;
val first : ('a -> bool) -> 'a list -> 'a option = <fun>

# first (fun x -> x > 3) [1;3;4;2;5];;
- : int option = Some 4

# first (fun x -> x > 5) [1;3;4;2;5];;
- : int option = None
Functions over option

```ocaml
# let result_ok r =
  match r with None -> false
  | Some _ -> true;;
val result_ok : 'a option -> bool = <fun>
# result_ok (first (fun x -> x > 3) [1;3;4;2;5]);;
- : bool = true
# result_ok (first (fun x -> x > 5) [1;3;4;2;5]);;
- : bool = false
```
Folding over Variants

# let optionFold someFun noneVal opt =
  match opt with None -> noneVal
  | Some x -> someFun x;;
val optionFold : ('a -> 'b) -> 'b -> 'a option -> 'b = <fun>

# let optionMap f opt =
  optionFold (fun x -> Some (f x)) None opt;;
val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>
Recursive Types

- The type being defined may be a component of itself
Mapping over Variants

# let optionMap f opt =
    match opt with None -> None
    | Some x -> Some (f x);;
val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>

# optionMap
    (fun x -> x - 2)
    (first (fun x -> x > 3) [1;3;4;2;5]);;
- : int option = Some 2
Recursive Data Types

# type int_Bin_Tree =
Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree);

type int_Bin_Tree = Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
Recursive Data Type Values

```plaintext
# let bin_tree = Node(Node(Leaf 3, Leaf 6), Leaf (-7));;

val bin_tree : int_Bin_Tree = Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```
Recursive Data Type Values

bin_tree = Node
    Node
    Leaf (-7)
    Leaf 3
    Leaf 6
Recursive Functions

# let rec first_leaf_value tree =
  match tree with (Leaf n) -> n
  | Node (left_tree, right_tree) ->
    first_leaf_value left_tree;;

val first_leaf_value : int_Bin_Tree -> int = <fun>

# let left = first_leaf_value bin_tree;;

val left : int = 3