# ML Crash Course

Zheng Guo

2018/10/03

# Agenda

- OCaml basics
- Questions about PA1
- Preview of tail recursion

# Imperative vs Functional

```java
public class Quicksort {
	public static void swap(int[] array, int i, int j) {
		int tmp = array[i];
		array[i] = array[j];
		array[j] = tmp;
	}

	public static int partition(int arr[], int left, int right) {
		int pivot = arr[(left + right) / 2]; // Pick a pivot point. Can be an element.

		while (left <= right) { // Until we've gone through the whole array
			// Find element on left that should be on right
			while (arr[left] < pivot) {
				left++;
			}

			// Find element on right that should be on left
			while (arr[right] > pivot) {
				right--;
			}

			// Swap elements, and move left and right indices
			if (left <= right) {
				swap(arr, left, right);
				left++;
				right--;
			}
		}
		return left;
	}

	public static void quickSort(int arr[], int left, int right) {
		int index = partition(arr, left, right);
		if (left < index - 1) { // Sort left half
			quickSort(arr, left, index - 1);
		}
		if (index < right) { // Sort right half
			quickSort(arr, index, right);
		}
	}
```

```fsharp
let rec quick l =
  match l with
    [] -> []
  | [x] -> l
  | p :: rl -> (match List.partition (fun x -> x < p) rl with
                  (l1, l2) -> (quick l1) @ [p] @ (quick l2))
```

# Pure Functional Programming Language

- Program is an expression
  - can be evaluated to a value
  - no statements here (no assignments, no pointers, no loops)
- Functions are values
  - can be *passed as arguments* to other functions
  - can be *returned as results* from other functions
  - can be *partially applied* (arguments passed *one at a time*)

# Everything is value

```
# 1;;
- : int = 1
# 1 + 2;;
- : int = 3
# (+) 1 2;;
- : int = 3
# "cat" ^ "dog";;
- : string = "catdog"
```

# Everything is value

```
# (+);;
- : int -> int -> int = <fun>
# if 1 > 0 then "true" else "false";;
- : string = "true"
# let f = (<) 1;;
val f : int -> bool = <fun>
# f 2;;
- : bool = true
```

# Strict static typing

```
# 1 + "cat";;
  “1cat”?
# 1 || false;;
  true?
# 3 +. 4.2;;
  7.2?
```

Type Error!

# Recursion

Do NOT forget the keyword rec

Implement factorial in OCaml

- Base case: n <= 1
- Recursive case: n > 1

```
let rec factorial x =
  if x <= 1
    then 1
    else x * factorial (x-1)
```

# Pattern matching

Match values against pattern (deconstruct) and do variable binding

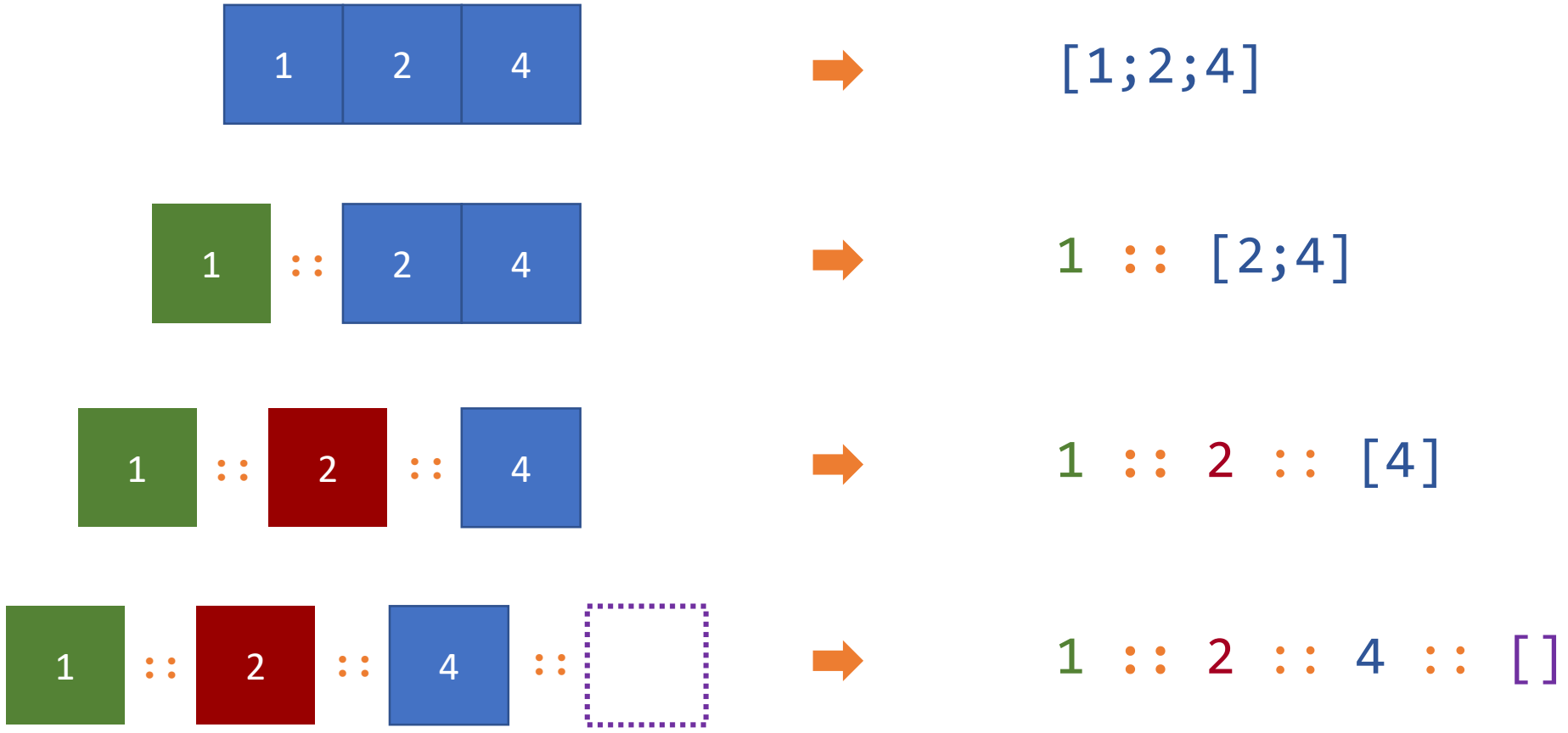
Pattern

- either a variable
- or a constructor applied to other patterns

```
match x with
   | [] -> …
   | hd::tl -> …
let (x,y) = (1,2) in x + y
let (x,h::t) = ("Hello", [1;2;3;4]);;
```

```
let (1+2,y) = (1,2) in y
let (f x,y) = (1,2) in y
```

# Lists



# Pattern matching

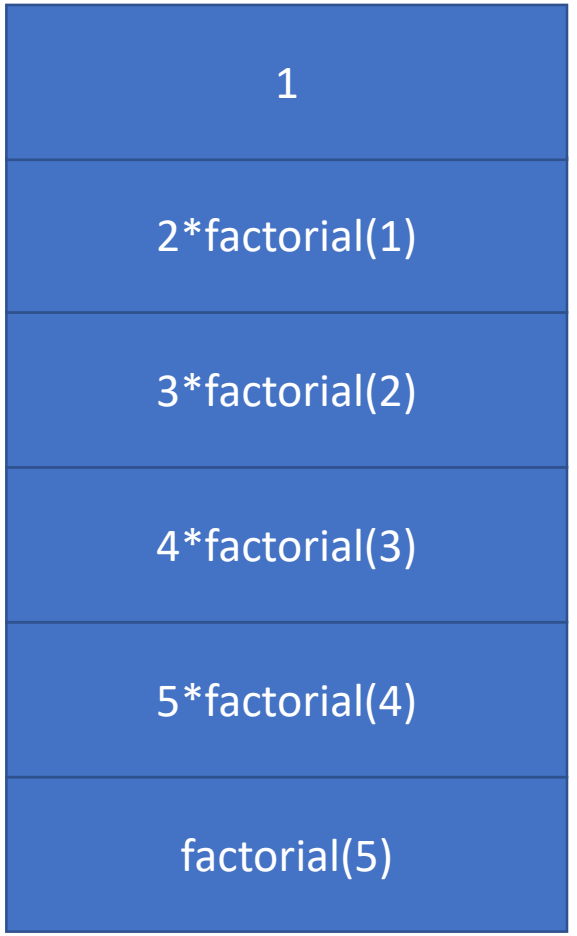
```
(* lastTwo :: ‘a list -> (’a, ’a) *)
let rec lastTwo xs = match xs with
  | [] -> failwith "empty list"
  | [x] -> failwith "only one element"
  | [x;y] ->(x,y)
  | hd::tl -> lastTwo tl

(* duplicate :: ‘a list -> ‘a list -> ‘a list *)
let rec duplicate xs = match xs with
  | [] -> []
  | hd::tl -> hd::hd::(duplicate tl)
```

# PA1

- Any library function is NOT allowed
  - No `@` operator or List.* function
  - mod is allowed
  - Helper functions are allowed
- Functions with rec are not necessarily recursive functions

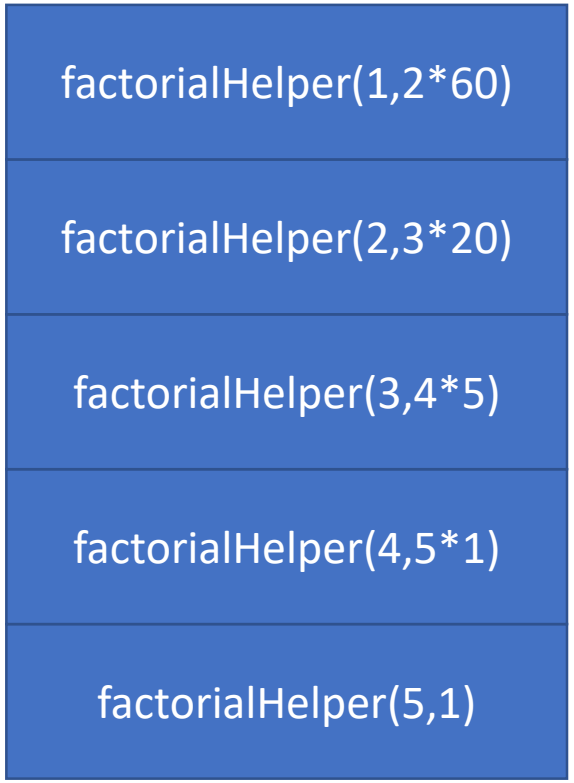
# Recursion

```
let rec factorial x =
  if x <= 1
    then 1
    else x * factorial (x-1)
```



# Tail recursion

```
let rec factorial x =
    let rec factorialHelper x acc =
      if x <= 1
        then acc
        else factorial (x-1, x*acc)
    in factorialHelper x 1
```



# Tail recursion

- Tail recursion: the resulting value is immediately returned (no further computation is performed on it by the recursive caller)

```
let rec factorial x =
  if x <= 1
    then 1
    else x * factorial (x-1)
```

There is computation multiplication here!

```
let rec factorial x =
  let rec factorialHelper x acc =
    if x <= 1
      then acc
      else factorial (x-1, x*acc)
  in factorialHelper x 1
```

# Why tail recursion

- Tail recursion:  the resulting value is immediately returned (no further computation is performed on it by the recursive caller)
- Compiler is SMART!
- Tail recursions are optimized into loops to save memory and time!

# Example: tail recursion

```
sumList : int list -> int

listReverse : ‘a list -> ‘a list

removeOdds : int list -> int list

take : int -> ‘a list -> ‘a list
```