# An Introduction to Functional Programming

Intermediate Computer Science Pre-College Program
23-27 July 2018
– Module 3 –

# Topics for Today

- Higher-order functions on lists
  - Map
  - Filter
  - Iter
  - Fold
- Exercises on Lists

Short but revealing class! Pay careful attention!
Note: Most of the contents of this class will be developed on the board

# Motivating Examples

- Let us implement the following functions
  - `succl : int list -> int list`
  - `to_upperl : char list -> char list`
  - `all_zero : int list -> bool list`
- What do you notice in common among all these implementations?

# Map

```
let rec map f l =
  match l with
  | [] -> []
  | (x::xs) -> (f x)::(map f xs)
```

- What does `map` do?
- What is its type?
- How can we use it to define `succl`, `to_upperl` and `all_zero`?

```
let succl’ = map (fun x -> x+1)
let to_upperl’ = map Char.uppercase_ascii
let all_zero = map (fun x -> x=0)
```

# Filter

- Lets implement the following functions:
  - `greater_than_zero : int list -> int list`
  - `uppercase : char list -> char list`
  - `non_empty : ’a list list -> ’a list list`
- What do you notice that they have in common?

# Filter

```
let rec filter p l =
 match l with
 | [] -> []
 | (x::xs) -> if (p x)
                then x::(filter p xs)
                else filter p xs
```

- What does `filter` do and what is its type?
- How can we use filter to implement `greater_than_zero`, `uppercase` and `non_empty`?

```
let greater_than_zero = filter (fun x -> x>0)
let uppercase = filter (fun x -> x=Char.uppercase_ascii x)
let non_empty = filter (fun x -> x!=[])
```

# Iterate

- Suppose we want to print out all the strings in a list of strings
- Here is one possible implementation of `print_list_of_strings`

```
let rec print_list_of_strings l =
match l with
| [] -> ()
| (x::xs) -> print_string x;
             print_list_of_strings xs
```

# Iterate

- OCaml provides `List.Iter`

```
List.iter print_string
```

# Fold

Consider the implementation of the following functions

- `sum_list : int list -> int`, that adds all the elements in a list of integers
- `and_list : bool list -> bool`, that indicates whether all the booleans in the list are true
- `concat : 'a list list -> 'a list`, that concatenates all the lists in a list

What do you notice in common among their implementations?

# Fold

```
let rec fold_right f l a =
match l with
| [] -> a
| (x::xs) -> f x (fold_right f xs a)
```

- Here is a description of the result of `fold_right f [x1; ...; xn] a`:

  ```
  f x1 (f x2 (... (f xn a) ...))
  ```

- What is its type?
- How can we define `all_fives`, `all` and `concat` in terms of `fold_right`?

# Function Schemes

- map, filter, iter and fold are known as function schemes
- They abstract common patterns of behaviour
- Also, they allow for code reuse
- Finally, they help better understand the problem

# Higher-Order Function Schemes

```
take

append
```

- Function schemes over function types