Programming Languages and Paradigms – COMP 302
Midterm 1 (Practice)

Instructor: Jacob Thomas Errington

Winter 2026

First name

Last name

Student ID

Question	Level
Use FP to model a scenario	
Higher-order functions	
Types and values	
Tail recursion	

- This exam is *printed on both sides* and is *closed-book*.
- This exam consists of separate question and answer packages. Only the answer booklet is collected. The question package will be destroyed.
- One 8.5x11 inch crib sheet, handwritten only on one side, is permitted.
- Calculators are not allowed.
- Ungraded additional space is provided on pages 5 and 6. If you need us to look there, clearly indicate this in the space dedicated to writing the answer.

Best of luck! – Jake

Answer 1: Use FP to model a scenario

(p.1)

Answer 2: Higher-order functions


```
let reverse l =
```

```
let for_all p l =
```

```
let map f l =
```

```
let rec iterations n f x =
```

```
let rec filter_map tp l =
```

```
let rec last p l =
```

Answer 3: Types and values

(p.2)

Typechecking

1.
2.
3.

Evaluation

1.
2.
3.

Answer 4: Tail recursion

(p.2)

Function	TR?	Operation	Justification for your translation strategy
`filter`			
`len`			
`add`			
`sum`			

Use the spaces below to give translations.

Scratch space

Scratch space

Question 1: Use FP to model a scenario **(p.2)**

In a home are three different smart devices: plugs, lights, and thermostats. Each of these may be on or off. Lights have a color: red, green, or blue. Thermostats also have a temperature value, which is a floating-point value.

Assessment: A level-grade is assigned to this problem based on these criteria, in descending order of importance:

- Completeness: all functions are correctly implemented. (Basic.)
- Readability: code is easy to read and clearly communicates its correctness. (Proficiency.)
- Elegance: use higher-order functions instead of recursion when appropriate to solve problems. (AM or Mastery.)

Define the type `device` to represent a single device in the smart home. You may define additional types as necessary in building your model. Design your model so as to make straightforward the implementations of the later functions in this problem.

Implement a function `set_state` that takes a `device` and an on/off state as input and sets the device's state to that value, by returning a new `device` with the updated state.

Implement the function `temp_changed` that takes a temperature and a list of devices such that `temp_changed t ds` computes an updated list of devices where each thermostat whose temperature is less than `t` are turned off and those whose temperature is greater than `t` are turned on.

Implement the function `average_temperature` to take a list of devices as input and computes the average temperature threshold across all the thermostats. If there are no thermostats, give an an average of zero.

Question 2: Higher-order functions **(p.3)**

Using HOFs:

- Use `fold_left : ('b -> 'b -> 'a) -> 'b -> 'a list -> 'b` to implement `reverse : 'a list -> 'a list` which reverses a list.
- Use `filter : ('a -> bool) -> 'a list -> 'a list` to implement `for_all : ('a -> bool) -> bool` that checks whether all elements of a list satisfy a property
- Use `fold_right : ('b -> 'a -> 'b) -> 'a list -> 'b -> 'b` to implement `map : ('a -> 'b) -> 'a list -> 'b list`

Definitions of `fold_left`, `fold_right`, and `filter` are given on page 3.

Defining HOFs:

- Use recursion to implement `times : int -> ('a -> 'a) -> 'a -> 'a` such that `times n f a` computes `n` applications of `f` to the starting value `a`, e.g. `times 3 f a` should compute `f (f (f a))`.
- Use pattern matching and recursion to implement `filter_map : ('a -> 'b option) -> 'a list -> 'b list`. It is a effectively a combination of `filter` and `map`. Think of the parameter of type `'a -> 'b option` as a transformation from `'a` to `'b` that may fail. Then the output of `filter_map` is all the successfully transformed values from the input list.
- Use pattern matching and recursion to implement `last : ('a -> bool) -> 'a list -> 'a option` which outputs the last element of the input list satisfying the predicate.

Assessment:

- To achieve M: 5/6 correct implementations.
- To achieve AM: at least 2 correct in each category.
- To achieve P: at least 1 correct in each category, 3 correct in total.
- To achieve B: at least 1 correct in each category.

Question 3: Types and values (p.4)

Assessment:

- Basic: at least 1 correct in both categories.
- Proficiency: at least 3 correct, with at least 1 in both categories.
- Approaching Mastery: at least 2 correct in both categories.
- Mastery: 5/6 correct

Typechecking:

Give the most general types of each of the following, or write "error" in case the program is ill-typed.

1. The expression:
   `let g y = fun x -> x *. y in fun x -> g 3`
2. The function `mystery` defined by:

```
let rec mystery a b c = match a with
  | [] -> c
  | Some x :: xs ->
    mystery xs b (b c x)
  | None :: xs -> mystery xs b c
```

3. The function `f` defined by

```
let f b =
  if b then
    let rec whoa x = whoa x in whoa
  else fun x -> x`
```

Evaluation:

For each of the following, give the value according to **OCaml's operational semantics,** or write "error" in case the program would not terminate with a value.

1. The expression:
   `let g y = fun x -> x *. y in fun x -> g 3`
2. The expression: `f true 5` where `f` is defined above in the typechecking portion.
3. The expression:
   `let x = fun () -> 4 in x () * x ()`

Question 4: Tail recursion (p.4)

Assessment: each level requires the preceding level.

- Basic: All but one of the non-TR functions are correctly identified together with the correct part(s) making them non-TR.
- Proficiency: All but one of the non-TR functions are correctly translated to be TR, using any strategy.
- Approaching Mastery: All non-TR functions are identified and correctly translated to be TR using an appropriate strategy.
- Mastery: All justifications are reasonable.

For each of the following recursive functions:

1. Decide whether it is tail-recursive. If it isn't, identify the operation performed in the function that makes it not tail-recursive.
2. For all non-tail-recursive functions, translate it to be tail-recursive using the least complex appropriate strategy. Briefly justify your choice.

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

type 'a tree =
  Empty | Node of 'a tree * 'a * 'a tree
let rec len t acc = match t with
    | Empty -> acc
    | Node (l, _, r) -> len l (len r (acc + 1))
```

```
type nat = Z | S of nat
let rec add n1 n2 = match n1 with
    | Z -> n2
    | S n1' -> S (add n1' n2)

let rec sum t acc ts = match t with
    | Node (l, x, r) -> sum l (x + acc) (r::ts)
    | Empty -> match ts with
        | [] -> acc
        | t::ts -> sum t acc ts
```

Appendix

Some standard higher order functions

```
(* ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b *)
let rec fold_left f b l = match l with
  | [] -> b
  | x::xs -> fold_left f (f b x) xs

(* ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b *)
let rec fold_right f l e = match l with
  | [] -> e
  | x::xs -> f x (fold_right f xs e)

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