

Supervision 2

Michaelmas Term 2026 · Michael Lee

●●●●● difficulty · ★ optional

WORK DUE	A single PDF, 24 hours before the supervision
EXPECTED TIME	6 hours 10 min (+ 15 min for the optional questions)

1 More on lists

PREREQUISITES **Lecture 4** · TIME 1 hour 45 min (+ 15 min)*This section aims to help you get really familiar with lists.*

QUESTION 1

CONCEPTUAL · 15 MIN

●●●●● BASIC

This question, and the next, use a slightly different version of `zip` than in the notes. Specifically, `zip` in the notes is curried

```
val zip: 'a list -> 'b list -> ('a * 'b) list
```

Whereas the `zip` in this question is uncurried

```
val zip: 'a list * 'b list -> ('a * 'b) list
```

The functions `zip` and `unzip` seem like they are each other's opposites. Which of the following statements are true? Justify your answers.

- (a) For all `'a`, `'b`, for all `xs` of type `('a * 'b) list`, `zip (unzip xs) = xs`
- (b) For all `'a`, for all `xs` of type `'a list`, `zip (unzip xs) = xs`
- (c) For all `'a`, `'b`, for all `xsp` of type `'a list * 'b list`, `unzip (zip xsp) = xsp`

QUESTION 2

CONCEPTUAL · 10 MIN

●●●●● BASIC

Mathematically, two functions $f: A \rightarrow B$ and $g: B \rightarrow A$ are inverses of each other if for all arguments $a \in A$, and for all arguments $b \in B$

$$g(f(a)) = a \qquad f(g(b)) = b$$

If only the first equation holds, we call g the *left inverse* of f (g inverts f if on the left of f , gf). If only the second condition holds, we call g the *right inverse* of f (g inverts f if on the right of f , fg).

Let `zip` be f and `unzip` be g .

- (a) What are A and B in this context?

- (b) Is unzip the inverse, right inverse or left inverse of zip?

QUESTION 3

PRACTICAL · 25 MIN

● ● ● ● ● CORE

- (a) Use the member function of [Section 4.2](#) to implement the function `inter` that calculates the *intersection* of two OCaml lists — that is, `inter xs ys` returns the list of elements that occur both in `xs` and `ys`.
- (b) Similarly, code a function to implement set union. It should avoid introducing repetitions, for example the union of the lists `[4;7;1]` and `[6;4;7]` should be `[1;6;4;7]` (though the order should not matter).
- (c) Package your answers above into a module with the following signature. Choose the right implementation of `t`

```

1  module Set: sig
2      type 'a t (* the type of sets *)
3      val empty: 'a t
4      val singleton: 'a -> 'a t
5      val union: 'a t -> 'a t -> 'a t
6      val inter: 'a t -> 'a t -> 'a t
7  end = struct
8      (*Write code here*)
9  end

```

OCaml

Submit only your choice of `t`, not the full implementation.

- (d) OCaml's [implementation of Set](#) requires that two elements of a set can be *compared*: that is, for every two elements x and y , we can establish if $x \leq y$. Why might this stronger requirement allow for more efficient implementations of sets?

HINT For part (d), you might need to wait for lecture 6.

★ QUESTION 4

CONCEPTUAL · 15 MIN

● ● ● ● ● CORE

OCaml's implementation of `Set` is a **functor**, a function that takes a module, and returns a module. In contrast with our earlier definition of Sets, this allows us to accept not only a type `t`, but also a custom comparator on the type.

```

module Make:
  functor (Ord : OrderedType) -> S with type elt = Ord.t

```

For example, we can build an `Int_set` using

```

1  module Int_set = Set.Make(struct
2      type t = int

```

OCaml

OCaml

```

3   let compare a b = (* custom comparator here *)
4   end)

```

Explain why it might make sense to require the user to pass in a compare function, rather than relying on defaults.

QUESTION 5

PRACTICAL · 15 MIN

● ● ● ● ● BASIC

Code a function that takes a list of integers and returns two lists, the first consisting of all nonnegative numbers found in the input and the second consisting of all the negative numbers. How would you adapt this function so it can be used to implement a sorting algorithm?

QUESTION 6

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

How does this version of zip differ from the one in the course?

OCaml

```

1   let rec zip xs ys =
2   match xs, ys with
3   | (x::xs, y::ys) -> (x,y) :: zip xs ys
4   | ([], [])      -> []

```

QUESTION 7

PRACTICAL · 10 MIN

● ● ● ● ● BASIC

Write a function plusl that computes the point-wise addition on two lists. For example,

OCaml

```

1   plusl [1;2;3] [4;5;6] (* [5;7;9] *)

```

QUESTION 8

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

Explain how the simple “making change” function in the lecture notes behaves in the following cases:

- (a) The till is set to [50 ; 10 ; 20 ; 5 ; 2 ; 1]
- (b) The till is set to [50 ; 50 ; 20 ; 10 ; 5 ; 2 ; 1]
- (c) The till is set to [50 ; 20 ; 10 ; 5 ; 2 ; 1 ; 0]
- (d) The till is set to [50 ; 20 ; 10 ; 5 ; 2 ; 0]
- (e) The amt is set to -1

2 Sorting

PREREQUISITES **Lecture 5** · TIME 1 hour 15 min

Algorithms should cover sorting much more thoroughly, so you should treat this as a gentle introduction to algorithmic thinking, rather than a comprehensive set of questions.

QUESTION 9

CONCEPTUAL · 15 MIN

● ● ● ● ● BASIC

You are given a long list of integers. Would you rather:

- (a) Find a single element with linear search or sort the list and use binary search?
- (b) Find a lot of elements with linear search or sort the list and use binary search?
- (c) Find all duplicates by pairwise comparison or sort the list and check for adjacent values?

QUESTION 10

CONCEPTUAL · 15 MIN

● ● ● ● ● BASIC

Another sorting algorithm (*bubble sort*) consists of looking at adjacent pairs of elements, exchanging them if they are out of order and repeating this process until no more exchanges are possible.

- (a) Characterise the lists for which bubble sort will have the worst performance (these lists are sometimes known as *pathological*)
- (b) Analyse the big O time complexity of this approach.
- (c) Suggest a scenario where it might be appropriate to use bubble sort

QUESTION 11

PRACTICAL · 15 MIN

● ● ● ● ● CORE

Implement bubble sort in OCaml.

QUESTION 12

PRACTICAL · 20 MIN

● ● ● ● ● CORE

Another sorting algorithm (selection sort) consists of looking at the elements to be sorted, identifying and removing a minimal element, which is placed at the head of the result. The tail is obtained by recursively sorting the remaining elements.

- (a) State, with justification, the time complexity of this approach.
- (b) Implement selection sort using OCaml.

QUESTION 13

REFLECTION · 10 MIN

● ● ● ● ● BASIC

Many programming languages, like Python, offer a sort function to be used on lists / arrays.

- (a) How could you choose what sorting algorithm to use?
- (b) How would you let the user override the default choice, if necessary?

3 TreesPREREQUISITES **Lecture 6** · TIME 1 hour 50 min

Trees are an important functional data structure, alongside lists. This section aims to both teach you about trees, and uncover their underlying mathematical structure as inductively-defined data.

QUESTION 14

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

Examine the following function declaration. What does `ptree 1 n` accomplish?

OCaml

```
1 let rec ptree k = function
2   | 0 -> Lf
3   | n -> Br (k, ptree (2*k) (n-1), ptree (2*k+1) (n-1))
```

QUESTION 15

PRACTICAL · 15 MIN

● ● ● ● ● BASIC

- (a) Write an function taking a binary tree labelled with integers and returning their sum.
- (b) Write a function taking a binary tree labelled with integers and returning their product.
- (c) Compare your answers to parts (a) and (b). Can you re-write them to share the majority of their implementations?

QUESTION 16

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

A set of elements X is *defined inductively* if it is defined using a set of rules. Each rule must take one of two forms:

$$\frac{}{x \in X} \qquad \frac{x_1 \in X \quad x_2 \in X \quad \cdots \quad x_n \in X}{y \in X}$$

The first states that x must be in X . The second gives a “formula” for forming new elements (y) in the set, from existing elements in the set $(x_1, \dots, x_n)$. (To be pedantic, the first form is subsumed by the second form, where $n = 0$, so there is only really one form of rule.) As an example, the set of natural

numbers can be defined inductively, as follows:

$$\frac{}{0 \in \mathbb{N}} \quad \frac{n \in \mathbb{N}}{s(n) \in \mathbb{N}}$$

The first states that 0 is a natural number. The second states that, if n is a natural number, then its successor ($s(n)$, or $n + 1$) is a natural number. Thus, inductively defined sets are defined as a set of base elements (in the case of natural numbers, 0), and additionally a set of rules that form new elements from existing elements.

- (a) Define the set of binary trees of natural numbers inductively (*you do not have to redefine natural numbers*). Compare your answer to the definition of trees as an OCaml datatype, as given in the notes.
- (b) Let Tree T be the set of trees of type T (this defines a *type-indexed set*). Generalise your answer to binary trees of type T .

QUESTION 17

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

There is a very natural way to prove properties of a set that is defined inductively. This is known as the **induction principle**. For example, to prove a property Φ about the naturals, we can:

- i. Prove that the property holds for 0
- ii. Prove that if the property holds for n , then it holds for $n + 1$.

Since these are the **only** ways to build natural numbers, we have proved it for all natural numbers.

Using your inductive definition of trees, derive the induction principle over trees of naturals.

QUESTION 18

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

In what sense is mathematical induction a special case of structural induction?

QUESTION 19

CONCEPTUAL · 35 MIN

● ● ● ● ● CORE

Prove the inequality involving the depth and size of a binary tree t from [Section 6.10](#).

$$\forall \text{ trees } t. \text{count}(t) \leq 2^{\text{depth}(t)} - 1$$

HINT The easiest way to do this is with *structural induction*. You can also try standard mathematical induction on some numerical property of the tree, but be careful with what you are assuming and what you are proving!

4 Datatypes

PREREQUISITES Lecture 6 · TIME 1 hour 20 min

An introduction to OCaml algebraic datatypes, and their uses. Revisits lists, now cast in the vocabulary of datatypes.

QUESTION 20

CONCEPTUAL · 15 MIN

● ● ● ● ● BASIC

You have seen how binary trees can be defined as a datatype

OCaml

```
1 type 'a tree = Lf | Br of 'a * 'a tree * 'a tree
```

- (a) Define lists similarly. Use `Nil` for the empty list, and `Cons` for `::`.
- (b) Define also *unary* trees
- (c) Establish the relationship between lists and unary trees
- (d) Define lists of type T inductively, as you did for trees.
- (e) Compare your answers in parts a and d.

QUESTION 21

CONCEPTUAL · 30 MIN

● ● ● ● ● BASIC

- (a) Translate the list `[0; 1; 2; 3]` into your inductive definition of lists (Question 20)

HINT It should look like `Cons(...)`

- (b) Consider the alternative inductive definition of lists, which I call snoc lists (For clarity, for this question, I call normal lists cons lists):

$$\frac{}{\text{Nil} \in \text{SnocList } T} \quad \frac{xs \in \text{SnocList } T \quad x \in T}{\text{Snoc}(xs, x) \in \text{SnocList } T}$$

- (i) Translate the list `[0; 1; 2; 3]` into your inductive definition of SnocLists.
- (ii) Define a type alias `'a clist` for `'a my_list`
- (iii) Define an OCaml datatype `'a slist` for snoc lists

QUESTION 22

PRACTICAL · 20 MIN

● ● ● ● ● CORE

- (a) Define `mirror_cs: 'a clist -> 'a slist` such that

```
mirror_cs (Cons (0, Cons (1, Cons (2, Cons (3, Nil)))))
= Snoc (Snoc (Snoc (Snoc (SNil, 3), 2), 1), 0)
```

OCaml

- (b) Define `mirror_sc : 'a slist -> 'a clist` such that

OCaml

```
mirror_sc (Snoc (Snoc (Snoc (Snoc (SNil, 0), 1), 2), 3))
= (Cons (3, Cons (2, Cons(1, Cons(0, Nil)))))
```

(c) Define `reassoc_cs`: `'a clist -> 'a slist` such that

OCaml

```
reassoc_cs (Cons (0, Cons (1, Cons(2, Cons(3, Nil)))))
= Snoc (Snoc (Snoc (Snoc (SNil, 0), 1), 2), 3)
```

(d) Define `reassoc_sc`: `'a slist -> 'a clist` such that

OCaml

```
reassoc_sc (Snoc (Snoc (Snoc (Snoc (SNil, 0), 1), 2), 3))
= Cons (0, Cons (1, Cons(2, Cons(3, Nil)))))
```

HINT For the `reassoc` functions, it might be easier to define a helper function that uses an accumulator

QUESTION 23

CONCEPTUAL · 15 MIN

● ● ● ● ● CORE

For each of the following functions, give the type of the function, and describe it.

- (a) `fun xs -> mirror_sc (mirror_cs xs)`
- (b) `fun xs -> mirror_cs (mirror_sc xs)`
- (c) `fun xs -> reassoc_sc (reassoc_cs xs)`
- (d) `fun xs -> reassoc_sc (mirror_cs xs)`
- (e) `fun xs -> mirror_sc (reassoc_cs xs)`

HINT Acceptable descriptions look like “it is the identity function” and “it reverses the argument”.