

Supervision 3

Michaelmas Term 2026 · Michael Lee

●●●● difficulty · ★ optional

WORK DUE A single PDF, 24 hours before the supervision
EXPECTED TIME 6 hours 20 min (+ 55 min for the optional questions)

1 Datatypes

PREREQUISITES Lecture 6 · TIME 1 hour

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

QUESTION 1

CONCEPTUAL · 10 MIN

●●●● BASIC

With reference to

```
1 type ty = A of int * bool | B of string | C
```

OCaml

explain why OCaml's datatypes are called algebraic datatypes.

HINT You might find [this resource](#), and the phrase "sums of products" useful.

QUESTION 2

PRACTICAL · 10 MIN

●●●● BASIC

Give the declaration of an OCaml datatype for arithmetic expressions that have the following possible shapes: integers, booleans, or expressions of the form

$$-E \quad \text{or} \quad E \times E \quad \text{or} \quad E \leq E \quad \text{or} \quad \text{if } E \text{ then } E \text{ else } E$$

and write the expression

$$\text{if } 3 \leq 1 \text{ then } 4 \times 5 \text{ else } -7$$

QUESTION 3

PRACTICAL · 15 MIN

● ● ● ● ● BASIC

Continuing the previous exercise, write a function `eval : expr -> expr` that evaluates an expression to return a value. As a sanity check, the example in the previous exercise should evaluate to `Int (-7)`. Some expressions, like `-true`, are nonsense: they have no meaning. Raise an `Ill_typed` exception on encountering these.

HINT Nested match blocks have to be wrapped in either parentheses `()` or `begin ... end`

QUESTION 4

PRACTICAL · 5 MIN

● ● ● ● ● BASIC

Define an `'a option` datatype that describes an optional value. Elements of `'a option` are either `None`, or `Some(v)`, where `v` is an element of `'a`.

QUESTION 5

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

There are two proposals for representing option types in memory (`None` is represented as some null value, e.g. a null pointer):

i. Option 1

`Some v` is represented exactly the same as `v`, so there would be no difference between `32` and `Some(32)`.

ii. Option 2

`Some v` has a different memory representation to `v`. For example, `Some v` is a pointer to a block of memory that tells you it is `Some(v)`, and then points at the value of `v`.

(a) Explain why the first option, while more efficient, cannot be used for option types in OCaml.

HINT Can you think of a value `v` for which it would be really bad if `Some(v) == v`?

(b) In OCaml, the `unboxed` annotation tells the compiler to use Option 1 instead of Option 2. Why might it make sense to add such annotations to the language?

2 Dictionaries and functional arrays

PREREQUISITES [Lecture 7](#) · TIME 2 hours 10 min

Lists and trees are the tools of the functional programmer. Arrays and dictionaries are the tools of the imperative one. This section shows how one can encode the second pair using the first.

QUESTION 6

CONCEPTUAL · 20 MIN

● ● ● ● ● BASIC

Draw the binary search tree that arises from successively inserting the following pairs into the empty tree: (Alice, 6), (Tobias, 2), (Gerald, 8), (Lucy, 9). Then repeat this task using the order (Gerald, 8), (Alice, 6), (Lucy, 9), (Tobias, 2). Why are results different? How could we avoid the issue encountered in the first case?

QUESTION 7

PRACTICAL · 15 MIN

● ● ● ● ● BASIC

Write an insertion function for binary search trees (with keys of `string` type, and values of polymorphic 'a type). It should resemble the existing update function of [Section 7.4](#) except that it should raise the exception `Collision` if the item to be inserted is already present. Now try modifying your function so that `Collision` returns the value previously stored in the dictionary at the given key. What problems do you encounter and why?

QUESTION 8

PRACTICAL · 30 MIN

● ● ● ● ● CORE

Write an algorithm for deleting an entry from a binary search tree. Comment on the suitability of your approach.

HINT There are two reasonable methods — one is simple, the other is efficient but a bit more tricky.

QUESTION 9

PRACTICAL · 30 MIN

● ● ● ● ● ADVANCED

Write a function to remove the first element from a functional array. All the other elements are to have their subscripts reduced by one. The cost of this operation should be linear in the size of the array.

QUESTION 10

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Show that the functions `preorder`, `inorder` and `postorder` of [Section 7.5](#) all require $O(n^2)$ time in the worst case, where n is the size of the tree.

QUESTION 11

CONCEPTUAL · 30 MIN

● ● ● ● ● ADVANCED

Difference lists represent lists as a higher-order function. The list `[0;1;2;3]` is written

OCaml

```
1 fun xs -> 0::1::2::3::xs
```

(a) Write down the type of difference lists

```
type 'a dlist = (* fill this in*)
```

(b) Write a function that turns a difference list into a regular list

(c) Write a function that constructs the empty difference list `[]`

(d) Write a function that constructs the singleton difference list `[x]`

(e) Write a function that appends two difference lists. What is the time complexity of an append?

(f) Rewrite `preorder` to return a difference list, which we then convert back to a list.

(g) Analyse the time complexity of the rewritten `preorder`.

3 Functions as valuesPREREQUISITES **Lecture 8** · TIME 1 hour 50 min (+ 55 min)

A first taste of the power of functional programming, and the strong constraints imposed by polymorphism.

QUESTION 12

PRACTICAL · 10 MIN

● ● ● ● ● CORE

Assume `t` is some known, but fixed, type. You are given a function `compare_l: t -> t -> int` such that the following equations hold

$$\begin{aligned} \text{compare_l } x \ y &= 0 && \text{when } x = y \\ \text{compare_l } x \ y &< 0 && \text{when } x < y \\ \text{compare_l } x \ y &> 0 && \text{when } x > y \end{aligned}$$

Write a function `compare_g` that satisfies the following equations

$$\begin{aligned} \text{compare_g } x \ y &= 0 && \text{when } x = y \\ \text{compare_g } x \ y &> 0 && \text{when } x < y \\ \text{compare_g } x \ y &< 0 && \text{when } x > y \end{aligned}$$
QUESTION 13

CONCEPTUAL · 10 MIN (+ 20 MIN)

● ● ● ● ● ADVANCED

Consider the following polymorphic functions.

```

1  let sw f x y = f y x
2  let co g f x = g (f x)
3  let cr f a b = f (a,b)

```

```

val id  : 'a -> 'a = <fun>
val ap  : ('a -> 'b) -> 'a -> 'b = <fun>
val ucr : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>

```

- (a) Infer the types of `sw`, `co` and `cr` (without asking OCaml)
- (b) Using only their types, give the definitions of `id`, `ap` and `ucr`.
- (c) Give short descriptions for each of these functions.
- (d) Reflect on your solutions to the previous two sheets: could any of your answers be improved by these higher order combinators? Do not submit your answers.
- ★ (e) Write the standard library **composition operators** in terms of the combinators above.

HINT The functions introduced in this exercise are all *higher-order function combinators*: functions that take functions as arguments and return functions as results. Further, they are all polymorphic, so their implementations are uniquely determined by the type.

QUESTION 14

CONCEPTUAL · 10 MIN

● ● ● ● ● ADVANCED

Polymorphic functions are highly constrained. You can often reconstruct a polymorphic function that you know nothing else about.

In this question, you can assume that all code is pure. That is, `f` and `k` do not have side effects. Assume a function `f` whose type is polymorphic in `'r`:

```
val f: (int -> 'r) -> 'r
```

- (a) Give two different functions with this type `(int -> 'r) -> 'r`.
- (b) Since `f` must return a value of some unknown type `'r`, and since we restrict ourselves to the pure fragment of OCaml, convince yourself that `f k = k v` for some fixed integer `v`.
- (c) You cannot see inside `f`. By choosing `'r` cleverly and picking a suitable `k`, write an expression that evaluates to `v`.

QUESTION 15

PRACTICAL · 15 MIN (+ 15 MIN)

● ● ● ● ● CORE

An OCaml type `t` is said to be an **ordered type** if it is equipped with a default comparison operator `< : t -> t -> bool` such that `a < b` returns `true` if `a : t` is “smaller than” `b : t`. Many OCaml types — such as `string` and `int` — can be ordered and compared with the `<` operator in the obvious way.

We often want to compare product types, such as `string * int`, building the comparison operators out of the individual atomic types `string` and `int`.

There are two ways of building compound comparators out of atomic comparators. The first is *pairwise ordering*, which compares elements of the pair individually:

$$(x, y) <_p (x', y') \iff x < x' \wedge y < y'$$

The second is *lexicographic ordering*, which orders by the first elements, and if they are equal, by the second element (for an arbitrary number of elements we get the familiar word ordering used in dictionaries):

$$(x, y) <_\ell (x', y') \iff x < x' \vee (x = x' \wedge y < y')$$

- Write OCaml functions implementing pairwise and lexicographic ordering for the type of pairs `string * int`.
- Your previous answer might have used the default comparison operator `<` for strings and ints. For more flexibility, we might want to pass non-default custom orderings (see next part). Write two higher-order OCaml functions to perform pairwise and lexicographic comparison of values of type `'a * 'b`, where the comparison operators for types `'a` and `'b` are passed as arguments.
- Explain how you could use your functions in the previous part, and the higher-order sorting function `insert`, to sort a list of type `(string * (int * string)) list` according to the following specification:

$$(s_1, (m, s_2)) < (s'_1, (n, s'_2)) \iff s_1 \leq s'_1 \wedge (m > n \vee (m = n \wedge s_2 < s'_2))$$

- ★ (d) With reference to `("a", (1, "x"))` and `("b", (2, "y"))`, explain why you should not pass this function to a sorting algorithm.

QUESTION 16

PRACTICAL · 20 MIN

● ● ● ● ● CORE

Without using `map`, write a function `map2` such that `map2 f` is equivalent to the composition `map (map f)`. The obvious solution requires declaring two recursive functions. Try to get away with one by exploiting nested pattern-matching.

QUESTION 17

PRACTICAL · 5 MIN

● ● ● ● ● BASIC

The built-in type `option`, shown below, can be viewed as a type of lists having at most one element. Declare an analogue of the function `map` for type `option`.

OCaml

```
1 type 'a option = None | Some of 'a
```

FMAP

You might be interested in containers other than `list` and `option`. I'd recommend taking a look at [Haskell's `fmap`](#). Note that the `f` stands for *functor*, but these functors are not the same as OCaml's module functors. Instead, the name derives from category theory.

QUESTION 18

CONCEPTUAL · 15 MIN (+ 5 MIN)

● ● ● ● ● CORE

Consider the following program

```

1  let f () =
2      let x = 1 in
3      let g () = x in
4      g
5
6  let h = f ();;
7
8  let x = 2 in h ()

```

OCaml

- (a) What should the result of this program be?
- ★ (b) Can you name a language where the result would differ from your answer to part (a)?
- (c) One way to implement variables is by looking up their values on the stack. Explain, with reference to the program above, why higher order functions need to be treated specially for this approach to work.

HINT Is `x = 1` still on the stack when `h ()` is evaluated?

QUESTION 19

CONCEPTUAL · 10 MIN (+ 15 MIN)

● ● ● ● ● CORE

Consider the following OCaml program ¹

```

1  let int_and_string1: 'a. ('a -> 'a) -> int * string
2      = fun arg -> (arg 1, arg "hey!")

```

OCaml

Now consider the following OCaml program

```

1  type poly = { id: 'a. 'a -> 'a }
2  let int_and_string2: poly -> int * string
3      = fun arg -> (arg.id 1, arg.id "hey!")

```

OCaml

- (a) Which of the two programs compiles?
- (b) Rewrite the types in second-order logical syntax, something like

$$\forall \alpha \dots$$

¹This exercise uses examples by [Hemant Sai Gouri](#), though adapted, as the examples in the linked blog post do not compile as written.

being careful about the bracketing.

- ★ (c) Can you explain the difference between the two types, and use this difference to explain the difference in behaviour?

HINT You will need to unfold poly.

QUESTION 20

PRACTICAL · 15 MIN

● ● ● ● ● BASIC

Recall the making change function of [Section 4.6](#):

```

1  let rec change till amt = match till, amt with
2    | _, 0  -> [ [] ]
3    | [], _ -> []
4    | c::till, amt ->
5      if amt < c then change till amt else
6        let rec allc = function
7          | [] -> []
8          | (cs::css) -> (c::cs) :: allc css
9        in allc (change (c::till) (amt-c))
10       @ change till amt

```

OCaml

The function allc applies the function “cons a c” to every element of a list. Eliminate it by declaring a curried cons function and applying map.

4 Sequences and laziness

PREREQUISITES [Lecture 9](#) · TIME 1 hour 20 min

The right way to think about lazy data is by the equations it satisfies, not how it is constructed.

QUESTION 21

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

Consider the list function concat, which concatenates a list of lists to form a single list. Can it be generalised to concatenate a sequence of sequences? What can go wrong?

```

1  let rec concat = function
2    | []      -> []
3    | l::ls  -> l @ concat ls

```

OCaml

QUESTION 22

PRACTICAL · 10 MIN

● ● ● ● ● BASIC

Write an analogue of `map` for sequences.

QUESTION 23

PRACTICAL · 25 MIN

● ● ● ● ● CORE

A *lazy binary tree* is either empty or is a branch containing a label and two lazy binary trees, possibly to infinite depth. Present an OCaml datatype to represent lazy binary trees, along with a function that accepts a lazy binary tree and produces a lazy list that contains all of the tree's labels. The order of the elements in the lazy list does not matter, as long as it contains all potential tree nodes.

QUESTION 24

PRACTICAL · 25 MIN

● ● ● ● ● BASIC

Recall `from`, where `from k` generates the infinite sequence $k, k+1, k+2, \dots$

OCaml

```
1 let rec from k = Cons (k, fun () -> from (k+1))
```

- (a) Using `from` and your analogue of `map` for sequences, write the sequence of powers of 5: 1, 5, 25, 125, ...
- (b) Now write a generator `g` such that `g 1` represents the same sequence. Do not use `from` or `map`.
- (c) Let the sequence `g 1` be `seq1` and the sequence `g 5` be `seq2`.
 - (i) Write out the first four elements of `seq1`
 - (ii) Write out the first four elements of `seq2`
 - (iii) Establish a relationship between `seq1` and `seq2` that does not reference how they were generated (does not mention `g`)

- (d) Consider the sequence

$$25, 125, 625, \dots$$

which can be characterised in two ways:

- (i) the powers of 5, from the third onwards;
- (ii) the powers of 5, each multiplied by 25.

How would you modify your answer to part (a) to produce this sequence? How would you modify your answer to part (b)? Which characterisation does each modification correspond to?

- (e) Consider the following infinite sequence. It is a geometric progression, potentially familiar from school.

$$T(a) = a, 0.5a, 0.25a, \dots$$

It is a fact that

$$\sum T(1) = 1 + 0.5 + \dots = 2$$

Here is the proof: multiplying every term in $T(1)$ by 0.5 gives

$$0.5T(1) = 0.5, 0.25, \dots = T(0.5)$$

Notice that $0.5T(1)$ is $T(1)$ with its first term removed. Hence we can establish the recursive equation

$$T(1) = 1, 0.5T(1)$$

and hence

$$\sum T(1) = 1 + \sum 0.5T(1) = 1 + 0.5 \sum T(1)$$

and so $\sum T(1) = 2$.

(i) Write $G(n)$ for the geometric progression

$$n, 5n, 25n, \dots$$

Establish a relationship between $kG(n)$ and $G(kn)$

(ii) Establish a (recursive) equation relating $G(n)$ and $5G(n)$

(iii) Relate the equation in G to your definition of g .

QUESTION 25

PRACTICAL · 10 MIN

● ● ● ● ● CORE

Write the lazy list containing every number of the form $7^a 9^b$, for $a, b \geq 0$: $1, 7, 9, 49, 63, 63, 81, \dots$. The order does not matter, and duplicates are permitted.

HINT **Question 24 is meant to make this one easy!** Define a family of sequences $H(m)$, where $H(1)$ is the sequence you want. As in question 24, describe two sequences $H'(m)$ and $H''(m)$, each obtained from $H(m)$. Write down an equation relating $H(m)$, $H'(m)$ and $H''(m)$, and turn it into a generator.