

Assignment 1

Tern Yuka Ringsted Følбак (nhx906)

24/09/2026

1 Bill of Materials

A. Type

A component is a record with an id, a name and a price. A BOM is a list of `component_and_quantity` records, each pairing a component with the quantity needed.

```
type component = { id : int; name : string; price : float }
type component_and_quantity = { comp : component; quantity : int }
type bom = component_and_quantity list
```

The price is a `float` since there can be cents, and the quantity is an `int` since we only use whole components.

B. Cost

Signature/purpose: `cost : bom -> float` returns the sum of price times quantity over the BOM.

Examples: `cost [] = 0.0`; a car (1 carframe at 5000, 4 wheels at 100, 4 screws at 0.5) costs 5402.0.

Code: a BOM is either empty or an element `caq` (component and quantity) followed by the `rest`, so we get two cases:

```
let rec cost (b : bom) : float =
  match b with
  | [] -> 0.0
  | caq :: rest -> caq.comp.price *. float_of_int caq.quantity +. cost rest
```

Tests: `cost [] = 0.0`, `cost car = 5402.0` and `cost computer = 186.0` all pass.

C. Difference

Signature/purpose: `difference : bom -> bom -> bom`. `difference b1 b2` gives, for each component in `b1` or `b2`, the quantity in `b1` minus the quantity in `b2`. Components with difference 0 are left out, and a negative quantity means that `b2` needs more of that component. **Examples:** `difference b b = []`, `difference b [] = b`, and `difference [] [nail: 2] = [nail: -2]`. **Code:** first a helper `quantity` that finds the quantity of a component in a BOM (0 if it is not there). Then `from_b1` handles the components of `b1`, `only_b2` the components only in `b2`, and `@` joins the two lists:

```
let rec quantity (id : int) (b : bom) : int =
  match b with
  | [] -> 0
  | caq :: rest -> if caq.comp.id = id then caq.quantity else quantity id rest

let difference (b1 : bom) (b2 : bom) : bom =
  let rec from_b1 (b : bom) : bom =
    match b with
    | [] -> []
    | caq :: rest ->
      let diff = caq.quantity - quantity caq.comp.id b2 in
      if diff < 0 then [] else [caq] @ from_b1 rest
  in
  from_b1 b1 @ from_b1 b2
```

```

    if diff = 0 then from_b1 rest else { comp = caq.comp; quantity = diff } :: from_b1 rest
in
let rec only_b2 (b : bom) : bom =
  match b with
  | [] → []
  | caq :: rest →
    if quantity caq.comp.id b1 = 0 then { comp = caq.comp; quantity = - caq.quantity } :: only_b2 rest
    else only_b2 rest
in
from_b1 b1 @ only_b2 b2

```

Tests: besides the examples we check that `cost (difference car computer) = cost car -. cost computer`; all tests pass.

D. Problems

Biggest issue was honestly just getting used to writing ocaml syntax. I have worked as SWE and now MLE, where I code almost on a daily basis (less now with claude code...), with languages such as python, javascript, typescript, C#. Where recursive functions have been written.

So just wrapping my head around the syntax was a bit "annoying." But now I honestly find the recursive function syntax to be cleaner here. Just feels less clanky, if that makes any sense...

So that is also the main issue I experienced in the problem 2. Groups.

2 Groups

```

type study_line = ComputerScience | MachineLearningDS | CSEconomics | MathCS
type student = { id : int; name : string; line : study_line }

```

A. Two representations

```

(* 1: a variant with one constructor per group size *)
type group = One of student | Two of student * student
           | Three of student * student * student
(* 2: a list of students, invariant: length 1 to 3 *)
type group_list = student list

```

Representation 1 makes illegal groups (empty or with 4+ members) which are impossible to construct, and the compiler warns if a match misses a size. Representation 2 is simpler and works with all list functions, but the size invariant is not checked by the type system. We use representation 1 which I find cleaner.

B. Grouping

Signature/purpose:

```

groups : student list → group list * group list * group list * group list

```

returns one list of groups per study line (in the order of the type definition). We split the problem into two helpers: `students_of` selects the students of a study line and `make_groups` splits a list into groups of three, where the last group may have one or two students. **Examples:** `make_groups [a;b;c;d] = [Three (a,b,c); One d]`, `groups [] = ([], [], [], [])`.

```

let rec students_of (l : study_line) (ss : student list) : student list =
  match ss with
  | [] → []
  | s :: rest → if s.line = l then s :: students_of l rest else students_of l rest

let rec make_groups (ss : student list) : group list =

```

```

match ss with
| [] → []
| [ a ] → [ One a ]
| [ a; b ] → [ Two (a, b) ]
| a :: b :: c :: rest → Three (a, b, c) :: make_groups rest

let groups (ss : student list) =
  ( make_groups (students_of ComputerScience ss),
    make_groups (students_of MachineLearningDS ss),
    make_groups (students_of CSEconomics ss),
    make_groups (students_of MathCS ss) )

```

Tests: with 4 CS, 2 ML, 1 CSE and 0 MathCS students, `groups` returns `([Three (s1,s3,s4); One s7], [Two (s2,s5)], [One s6], [])`; all tests pass.

C. Problems

Syntax was the main issue here too. The type declaration was actually quite similar to the Enum data type in C#.

Other than that, then in the `make_groups` recursive function, the `a :: b :: c :: rest ->` took me a little time, as I needed to make sure to accomodate the max 3 student group rule.

3 Concepts and syntax

A. Types

A type is a set of values together with the operations allowed on them, e.g. `int`, `float`, `string` or `int list`. OCaml checks types before running the program, so e.g. `1 + 2.0` is rejected. Yes, we defined new types: record types (`component`, `component_and_quantity`, `student`), which group named fields into one value; type abbreviations (`bom = component_and_quantity list`, `group_list = student list`), which give an existing type a meaningful name; and variant types (`study_line`, `group`), which list the alternatives a value can be.

B. Recursive functions

Recursive functions, are functions which call upon themselves, and they do so until they reach a base case. In our examples that was often checking if the object was empty, if so then return empty object/list. Which then stops the loop, and starts exiting the call stack. I typically use more commonly found loops such as `for`, `while`, loops, mostly due to readability. But recursive functions are useful when the problem can be broken down into smaller sub-problems, or working with data that has a nested structure. Like our BOMs which nests components which contains its own properties.