



## Structured data and composite types

The kinds of data that we’ve introduced so far have been unstructured. The values are separate atoms,<sup>1</sup> discrete undecomposable units. Each integer is separate and atomic, each floating point number, each truth value. But the power of data comes from the ability to build new data from old by putting together data *structures*.

In this chapter, we’ll introduce three quite general ways built into OCaml to structure data: tuples, lists, and records. For each such way, we describe how to *construct* structures from their parts using *value constructors*; what the associated *type* of the structures is and how to construct a type expression for them using *type constructors*; and how to *decompose* the structures into their component parts using *pattern-matching*. (We turn to methods for generating your own composite data structures in Chapter 11.) We start with tuples.


<sup>1</sup> The term “atom” is used here in its sense from Democritus and other classical philosophers, the indivisible units making up the physical universe. Now, of course, we know that though chemical elements are made of atoms, those atoms themselves have substructure and are not indivisible. Unlike the physical world, the world of discrete data can be well thought of as being built from indivisible atoms.

### 7.1 Tuples

The first structured data type is the **TUPLE**, a fixed length sequence of elements. The smallest tuples are pairs, containing two elements, then triples, quadruples, *quintuples*, *sextuples*, *septuples*, and so forth. (The etymology of the term “tuple” derives from this semi-productive suffix.)

In OCaml, a tuple value is formed using the **VALUE CONSTRUCTOR** for tuples, an infix comma. A pair containing the integer 3 and the truth value `true`, for instance, is given by `3, true`. The order is crucial; the pair `true, 3` is a different pair entirely. (Indeed, as we will see, these two pairs are not even of the same type.)


The type of a pair is determined by the types of its parts. We name the type by forming a type expression giving the types of the parts combined using the infix **TYPE CONSTRUCTOR** `*`, read “cross” (for “**cross product**”). For instance, the pair `3, true` is of type `int * bool` (read, “int cross bool”).

**Exercise 32**  What are the types for the following pair expressions?

1. `false, 5`
2. `false, true`
3. `3, 5`
4. `3.0, 5`
5. `5.0, 3`
6. `5, 3`
7. `succ, pred`


□

Triples are formed similarly. A triple of the elements 1, 2, and "three" would be `1, 2, "three"`; its type is `int * int * string`. This triple should not be confused with the pair consisting of the integer 1 and the `int * string` pair `2, "three"`. Such a pair containing a pair is also constructable, as `1, (2, "three")`, and is of type `int * (int * string)`. The parentheses in both the value expression and the type expression make clear that this datum is structured as a pair, not a triple.

**Exercise 33**  Construct a value for each of the following types.

1. `bool * bool`
2. `bool * int * float`
3. `(bool * int) * float`
4. `(int * int) * int * int`
5. `(int -> int) * int * int`
6. `(int -> int) * int -> int`

□

**Exercise 34**  Integer division leaves a remainder. It is sometimes useful to calculate both the result of the quotient and the remainder. Define a function `div_mod : int -> int -> (int * int)` that takes two integers and returns a pair of their division and the remainder. For instance,

```
# div_mod 40 20 ;;
- : int * int = (2, 0)
# div_mod 40 13 ;;
- : int * int = (3, 1)
# div_mod 0 12 ;;
- : int * int = (0, 0)
```

Using this technique of returning a pair, we can get the effect of a function that returns multiple values. □

**Exercise 35** In Exercise 28, you are asked to implement the computus to calculate the month and day of Easter for a given year by defining two functions, one for the day and one for the year. A more natural approach is to define a single function that returns both the month and the day. Use the technique from Exercise 34 to implement a single function for computus. □

## 7.2 Pattern matching for decomposing data structures

The value constructor is used to construct composite values from parts. How can we do the inverse, extracting the parts from the composite structure? Perhaps surprisingly, we make use of the value constructor for this purpose as well, by matching a template pattern containing the constructor with the structure being decomposed. The match construction is used to perform this matching and decomposition. The general form of a match is<sup>2</sup>

```
match <expr> with
| <pattern1> -> <expr1>
| <pattern2> -> <expr2>
...
```

The structured value given by the  $\langle expr \rangle$  is pattern-matched against each of the patterns  $\langle pattern_1 \rangle$ ,  $\langle pattern_2 \rangle$ , and so on, in that order. These patterns may contain variables. Whichever pattern matches first, the variables therein name the corresponding parts of the  $\langle expr \rangle$  being matched. The corresponding  $\langle expr_i \rangle$ , which may use the variables just bound by the pattern, is evaluated to provide the value of the match construction as a whole. The variables in patterns are newly introduced names, just like those in `let` and `fun` expressions, and like those variables, they also have a scope, namely, the corresponding  $\langle expr_i \rangle$ .

For example, suppose we want to add the integers in an integer pair. We need to extract the integers in order to operate on them. Here is a function that extracts the two parts of the pair and returns their sum.

<sup>2</sup> The first vertical bar is, strictly speaking, optional. We uniformly use it for consistency of demarcating the patterns appearing on consecutive lines, as discussed in Section B.1.6.

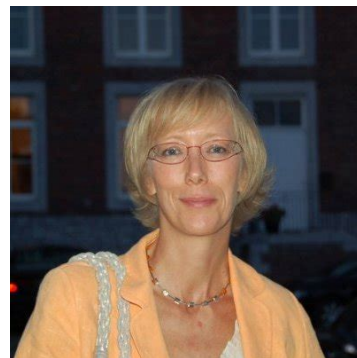


Figure 7.1: Computer scientist Marianne Baudinet's (1985) work with David MacQueen on compiling ML-style pattern matching constructs to efficient matching code proved to be the breakthrough that made the extensive use of pattern matching in ML-style languages practical.

```
# let addpair (pair : int * int) : int =
#   match pair with
#   | x, y -> x + y ;;
val addpair : int * int -> int = <fun>
# addpair (3, 4) ;;
- : int = 7
```

In the pattern `x, y`, the variables `x` and `y` are names that can be used for the two components of the pair, as they have been in the expression `x + y`. There is nothing special about the names `x` and `y`; any variables could be used.

The `match` used here is especially simple in having just a single pattern/result pair. Only one is needed because there is only one value constructor for pairs. We'll shortly see examples where more than one pattern is used.

Notice how the `match` construction allows us to deconstruct a structured datum into its component parts simply by matching against a template that uses the *very same* value constructor that is used to build such data in the first place. This method for decomposition is extremely general. It allows extracting the component parts from arbitrarily structured data.

You might think, for instance, that it would be useful to have a function that directly extracts the first or second element of a pair. But these can be written in terms of the `match` construct.<sup>3</sup>


```
# let fst (pair : int * int) : int =
#   match pair with
#   | x, y -> x ;;
Characters 56-57:
  | x, y -> x ;;
      ^
Warning 27: unused variable y.
val fst : int * int -> int = <fun>
# fst (3, 4) ;;
- : int = 3
```

The warning message arises because the variable `y` appears in the pattern, but is never used in the corresponding action. Often this is a sign that something is awry in one's code: Why would you establish a variable only to ignore its value? For that reason, this warning message can be quite useful in catching subtle bugs. But in cases like this, where the value of the variable is really irrelevant, the warning is misleading. To eliminate it, an **ANONYMOUS VARIABLE** – a variable starting with the underscore character – can be used instead. This codifies the programmer's intention that the variable not be used,

<sup>3</sup> The functions `fst` and `snd` are available as part of the `StdLib` module, but it's useful to see how they can be written in terms of the core of the OCaml language.


and disables the warning message. This is a good example of the edict of intention: by clearly and uniformly expressing our intention not to use a variable, the language interpreter can help find latent bugs where we intended to use a variable but did not (as when a variable name is misspelled).

```
# let fst (pair : int * int) : int =
#   match pair with
#   | x, _y -> x ;;
val fst : int * int -> int = <fun>
# fst (3, 4) ;;
- : int = 3
```

**Exercise 36**  Define an analogous function `snd : int * int -> int` that extracts the second element of an integer pair. For instance,

```
# snd (3, 4) ;;
- : int = 4
```

□

As another example, consider the problem of calculating the distance between two points , where the points are given as pairs of floats. First, we need to extract the coordinates in each dimension by pattern matching:

```
let distance p1 p2 =
  match p1 with
  | x1, y1 ->
    match p2 with
    | x2, y2 -> ...calculate the distance... ;;
```

Rather than use two separate pattern matches, one for each argument, we can perform both matches at once using a pattern that matches against the pair of points `p1, p2`.

```
let distance p1 p2 =
  match p1, p2 with
  | (x1, y1), (x2, y2) -> ...calculate the distance... ;;
```

Once the separate components of the points are in hand, the distance can be calculated:

```
# let distance p1 p2 =
#   match p1, p2 with
#   | (x1, y1), (x2, y2) ->
#       sqrt ((x2 -. x1) ** 2. +. (y2 -. y1) ** 2.) ;;
val distance : float * float -> float * float -> float = <fun>
```

The ability to pattern match to extract and name data components is so useful that OCaml provides syntactic sugar to integrate it into other binding constructs, such as the `let` and `fun` constructs. In cases where there is only a single pattern to be matched (as in the examples above), the matching can be performed directly in the `let`. That is, an expression of the form

```
let <var> = <expr> in
match <var> with
| <pattern1> -> <expr1>
```

can be “sugared” to<sup>4</sup>

```
let <pattern1> = <expr> in
<expr1>
```

Using this sugared form further simplifies the distance function.

```
let distance p1 p2 =
  let (x1, y1), (x2, y2) = p1, p2 in
  sqrt ((x2 -. x1) ** 2. +. (y2 -. y1) ** 2.) ;;
```

Finally, pattern matching can even be used in global `let` constructs, to further simplify.


```
# let distance (x1, y1) (x2, y2) =
#   sqrt ((x2 -. x1) ** 2. +. (y2 -. y1) ** 2.) ;;
val distance : float * float -> float * float -> float = <fun>

# distance (1., 1.) (2., 2.) ;;
- : float = 1.41421356237309515
```

As usual, it is useful to add typings in the global definition to make clear the intended types of the arguments and the result.<sup>5</sup>

```
# let distance (x1, y1 : float * float)
#           (x2, y2 : float * float)
#           : float =
#   sqrt ((x2 -. x1) ** 2. +. (y2 -. y1) ** 2.) ;;
val distance : float * float -> float * float -> float = <fun>

# distance (1., 1.) (2., 2.) ;;
- : float = 1.41421356237309515
```


**Exercise 37**  Simplify the definitions of `addpair` and `fst` above by taking advantage of this syntactic sugar. □

Using this shorthand can make code much more readable, and is thus recommended. See the style guide (Section B.4.2) for further discussion.

<sup>4</sup> Anonymous functions can benefit from this syntactic sugar as well, for instance, as in

```
# (fun (x, y) -> x + y) (3, 4) ;;
- : int = 7
```

<sup>5</sup> This example provides a good opportunity to mention that for readability code lines should be kept short. We use a convention described in the style guide (Section B.3.4) for breaking up long function definition introductions.

**Exercise 38** Define a function `slope : float * float -> float`  
`* float -> float` that returns the slope  between two points. □

### 7.2.1 Advanced pattern matching

Not only composite types can be the object of pattern matching. Patterns can match particular values of atomic types as well, such as `int` or `bool`. One could, for instance, write

```
# let int_of_bool (cond : bool) : int =
#   match cond with
#   | true -> 1
#   | false -> 0 ;;
val int_of_bool : bool -> int = <fun>

# int_of_bool true ;;
- : int = 1
# int_of_bool false ;;
- : int = 0
```

For booleans, however, the use of a conditional is considered a better approach:

```
# let int_of_bool (cond : bool) : int =
#   if cond then 1 else 0 ;;
val int_of_bool : bool -> int = <fun>
```

Using `cond = true` as the test part of the conditional is redundant and stylistically poor. See Section [B.5.2](#).

Integers can also be matched against:

```
# let is_small_int (x : int) : bool =
#   match abs x with
#   | 0 -> true
#   | 1 -> true
#   | 2 -> true
#   | _ -> false ;;
val is_small_int : int -> bool = <fun>

# is_small_int ~-1 ;;
- : bool = true
# is_small_int 2 ;;
- : bool = true
# is_small_int 7 ;;
- : bool = false
```

Notice here the use of an anonymous variable `_` as a **WILD-CARD** pattern that matches any value.



In the `is_small_int` function, the same result is appropriate for multiple patterns. Rather, than repeat the result expression in each case, multiple patterns can be associated with a single result, by listing the patterns interspersed with vertical bars (`|`).

```
# let is_small_int (x : int) : bool =
#   match abs x with
#   | 0 | 1 | 2 -> true
#   | _ -> false ;;
val is_small_int : int -> bool = <fun>
```

### 7.3 Lists

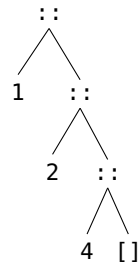
Tuples are used for packaging together *fixed-length* sequences of elements of perhaps *differing* type. **LISTS**, conversely, are used for packaging together *varied-length* sequences of elements all of the *same* type. The type constructor `list` for lists thus operates on a single type, the type of the list elements, and is written in postfix position. For instance, the type corresponding to a list of integers is given by the type expression `int list`, a list of booleans as `bool list`, a list of points (pairs of floats) as `(float * float) list`.

There are two value constructors for lists. The first value constructor, written `[]` and conventionally read as “**NIL**”, specifies the empty list, that is, the list containing no elements at all. The second value constructor, written with an infix `::` and conventionally read as “**CONS**”,<sup>6</sup> takes two arguments – a first element and a further list of elements – and specifies the list whose first element is its first argument and whose remaining elements are the second. (The two parts of a non-empty list, the first element and the remaining elements, are called the **HEAD** and the **TAIL** of the list, respectively.)

Suppose we want a list of integers containing just the integer 4. Such a list can be constructed by starting with the empty list `[]`, and “consing” 4 to it as `4 :: []`. The list containing, in sequence, 2 and 4 is constructed by consing 2 onto the list containing 4, that is, `2 :: (4 :: [])`. The list of the integers 1, 2, and 4 is analogously `1 :: (2 :: (4 :: []))`.

As usual, some notational cleanup is in order. First, we can take advantage of the fact that the `::` operator is right associative, so that the parentheses in the lists above are not needed. We can simply write `1 :: 2 :: 4 :: []`. Second, OCaml provides a more familiar alternative notation – more sugar – for lists, writing the elements of the list in order within brackets and separated by semicolons, as `[1; 2; 4]`. We can think of all of these as alternative concrete syntaxes for the same underlying abstract structure, given by


<sup>6</sup> The term “cons” for this constructor derives from the `cons` function in one of the earliest and most influential functional programming languages, Lisp. It derives from the idea of *constructing* a list by adding a new element.



You can verify the equivalence of these notations by entering them into OCaml:

```
# 1 :: (2 :: (4 :: [])) ;;
- : int list = [1; 2; 4]
# 1 :: 2 :: 4 :: [] ;;
- : int list = [1; 2; 4]
# [1; 2; 4] ;;
- : int list = [1; 2; 4]
```

Notice that in all three cases, OCaml provides the inferred type `int list` and reports the value using the sugared list notation.<sup>7</sup>

**Exercise 39**  Which of the following expressions are well-formed, and for those that are, what are their types and how would their values be written using the sugared notation?

1. `3 :: []`
2. `true :: false`
3. `true :: [false]`
4. `[true] :: [false]`
5. `[1; 2; 3.1416]`
6. `[4; 2; -1; 1_000_000]`
7. `([true], false)`

□

<sup>7</sup> The list containing elements, say, 1 and 2 – written `[1; 2]` – should not be confused with the pair of those same elements – written `(1, 2)`. The concrete syntactic differences may be subtle (semicolon versus comma; brackets versus parentheses) but their respective types make the distinction quite clear.

Using the `::` and bracketing notations, we can construct lists from their elements. How can we extract those elements from lists? As always in OCaml, decomposing structured data is done with pattern-matching; no new constructs are needed. We'll see examples shortly.

### 7.3.1 Some useful list functions

To provide some intuition with list processing, we'll construct a few useful functions, starting with a function to determine if an integer list is empty or not. We start with considering the type of the function. Its argument should be an integer list (of type `int list`) and its result a truth value (of type `bool`), so the type of the function itself is `int list -> bool`. This type information is just what we need in order to write the first line of the function definition, naming the function's argument and incorporating the typing information:

```
let is_empty (lst : int list) : bool = ...
```

Now we need to determine whether `lst` is empty or not, that is, what value constructor was used to construct it. We can do so by pattern matching `lst` against a series of patterns. Since lists have only two value constructors, two patterns will be sufficient.

```
let is_empty (lst : int list) : bool =
  match lst with
  | [] -> ...
  | head :: tail -> ...
```

What should we do in these two cases? In the first case, we can conclude that `lst` is empty, and hence, the value of the function should be `true`. In the second case, `lst` must have at least one element (now named `head` by the pattern match), and is thus non-empty; the value of the function should be `false`.<sup>8</sup>

<sup>8</sup> We've used alignment of the arrows in the pattern match to emphasize the parallelism between these two cases. See the discussion in the style guide (Section B.1.6) for differing views on this practice.

```
# let is_empty (lst : int list) : bool =
#   match lst with
#   | [] -> true
#   | head :: tail -> false ;;
Characters 79-83:
  | head :: tail -> false ;;
  ^^^

Warning 27: unused variable head.
Characters 87-91:
  | head :: tail -> false ;;
  ^^^

Warning 27: unused variable tail.
val is_empty : int list -> bool = <fun>
```

Since neither `head` nor `tail` are used in the second pattern match, they should be made anonymous variables to codify that intention (and avoid a warning message).<sup>9</sup>

<sup>9</sup> The “wild card” anonymous variable `_` is special in not serving as a name that can be later referred to, and is thus allowed to be used more than once in a pattern.

```
# let is_empty (lst : int list) : bool =
#   match lst with
#   | []      -> true
#   | _ :: _ -> false ;;
val is_empty : int list -> bool = <fun>

# is_empty [] ;;
- : bool = true
# is_empty [1; 2; 3] ;;
- : bool = false
# is_empty (4 :: []) ;;
- : bool = false
```

Sure enough, the function works well on the test cases.

Let's try another example: calculating the **LENGTH** of a list, the count of its elements. We use the same approach, starting with the type of the function. The argument is an `int list` as before, but the result type is an `int` providing the count of the elements; overall, the function is of type `int list -> int`. The type of the function in hand, the first line writes itself.

```
let length (lst : int list) : int = ...
```

And again, a pattern match on the sole argument is a natural first step to decide how to proceed in the calculation.

```
let length (lst : int list) : int =
  match lst with
  | [] -> ...
  | hd :: tl -> ...
```

In the first match case, the list is empty; hence its length is 0.

```
let length (lst : int list) : int =
  match lst with
  | [] -> 0
  | hd :: tl -> ...
```

The second case is more subtle, however. The length must be at least 1 (since the list at least has the single element `hd`). But the length of the list overall depends on `tl`, and in particular, the length of `tl`. If only we had a method for calculating the length of `tl`.

But we do; the `length` function itself can be used for this purpose! Indeed, the whole point of `length` is to calculate lengths of `int` lists like `tl`. We can call `length` recursively on `tl`, and add 1 to the result to calculate the length of the full list `lst`.<sup>10</sup>

<sup>10</sup> As with the definition of the recursive factorial function in Section 6.4, the well-founded basis of this recursive definition depends on the recursive calls heading in the direction of the base case. In this case, the recursive application of the function is to a *smaller* data structure, the tail of the original argument, and all further applications will similarly be to smaller and smaller data structures. This process can't continue indefinitely. Inevitably, it will bottom out in application to the empty list, at which point the computation is non-recursive and terminates. Recursive computation may seem a bit magical when you first confront it, but over time it becomes a powerful tool.

```
# let rec length (lst : int list) : int =
#   match lst with
#   | [] -> 0
#   | _hd :: tl -> 1 + length tl ;;
val length : int list -> int = <fun>
```

(We've made `_hd` an anonymous variable for the same reasons as above, and also inserted the `rec` keyword to allow the recursive reference to `length` within the definition.)


We can test the function on a few examples to demonstrate it.

```
# length [1; 2; 4] ;;
- : int = 3
# length [] ;;
- : int = 0
# length [[1; 2; 4]] ;;
```

Characters 8-17:

```
length [[1; 2; 4]] ;;
^^^^^^^^
```

```
Error: This expression has type 'a list
      but an expression was expected of type int
```

**Exercise 40**  Why does this last example cause an error, given that its input is a list of length one? Chapter 9 addresses this problem more thoroughly. □

As a final example, we'll implement a function that, given a list of pairs of integers, returns the list of products of the pairs. For example, the following behaviors should hold.

```
# prods [2,3; 4,5; 0,10] ;;
- : int list = [6; 20; 0]
# prods [] ;;
- : int list = []
```

By now the process should be familiar. Start with the type of the function: `(int * int) list -> int list`. Use the type to write the function introduction:

```
let rec prods (lst : (int * int) list) : int list = ...
```

Use pattern-matching to decompose the argument:

```
let rec prods (lst : (int * int) list) : int list =
  match lst with
  | [] -> ...
  | hd :: tl -> ...
```

In the first pattern match, the list is empty; we should thus return the empty list of products.

```
let rec prods (lst : (int * int) list) : int list =
  match lst with
  | [] -> []
  | hd :: tl -> ...
```

Finally, we get to the tricky bit. If the list is nonempty, the head will be a pair of integers, which we'll want access to. We could pattern match against `hd` to extract the parts:

```
let rec prods (lst : (int * int) list) : int list =
  match lst with
  | [] -> []
  | hd :: tl ->
    match hd with
    | (x, y) -> ...
```

but it's simpler to fold that pattern match into the list pattern match itself:

```
let rec prods (lst : (int * int) list) : int list =
  match lst with
  | [] -> []
  | (x, y) :: tl -> ...
```


Now, the result in the second pattern match should be a list of integers, the first of which is  $x * y$  and the remaining elements of which are the products of the pairs in `tl`. The latter can be computed recursively as `prods tl`. (It's a good thing we thought ahead to use the `rec` keyword.) Finally, the list whose first element is  $x * y$  and whose remaining elements are `prods tl` can be *constructed* as  $x * y :: \text{prods } tl$ .

```
# let rec prods (lst : (int * int) list) : int list =
#   match lst with
#   | [] -> []
#   | (x, y) :: tl -> x * y :: prods tl ;;
val prods : (int * int) list -> int list = <fun>
# prods [2,3; 4,5; 0,10] ;;
- : int list = [6; 20; 0]
# prods [] ;;
- : int list = []
```

You'll have noticed a common pattern to writing these functions, one that is widely applicable.


1. Write down some examples of the function's use.
2. Write down the type of the function.
3. Write down the first line of the function definition, based on the type of the function, which provides the argument and result types.
4. Using information about the argument types, decompose one or more of the arguments.
5. Solve each of the subcases, paying attention to the types, to construct the output value.
6. Test the examples from Step 1.

Using this **STRUCTURE-DRIVEN PROGRAMMING** pattern can make it so that simple functions of this sort almost write themselves. Notice the importance of types in the process. The types constrain so many aspects of the function that they provide a guide to writing the function itself.

**Exercise 41**  Define a function `sum : int list -> int` that computes the sum of the integers in its list argument.


```
# sum [1; 2; 4; 8] ;;
- : int = 15
```

What should this function return when applied to the empty list? □

**Exercise 42**  Define a function `prod : int list -> int` that computes the product of the integers in its list argument.


```
# prod [1; 2; 4; 8] ;;
- : int = 64
```

What should this function return when applied to the empty list? □

**Exercise 43**  Define a function `sums : (int * int) list -> int list`, analogous to `prods` above, which computes the list each of whose elements is the sum of the elements of the corresponding pair of integers in the argument list. For example,


```
# sums [2,3; 4,5; 0,10] ;;
- : int list = [5; 9; 10]
# sums [] ;;
- : int list = []
```

□

**Exercise 44**  Define a function `inc_all : int list -> int list`, which increments all of the elements in a list.


```
# inc_all [1; 2; 4; 8] ;;
- : int list = [2; 3; 5; 9]
```

□

**Exercise 45**  Define a function `square_all : int list -> int list`, which squares all of the elements in a list.


```
# square_all [1; 2; 4; 8] ;;
- : int list = [1; 4; 16; 64]
```

□

**Exercise 46**  Define a function `append : int list -> int list -> int list` to append two integer lists. Some examples:


```
# append [1; 2; 3] [4; 5; 6] ;;
- : int list = [1; 2; 3; 4; 5; 6]
# append [] [4; 5; 6] ;;
- : int list = [4; 5; 6]
# append [1; 2; 3] [] ;;
- : int list = [1; 2; 3]
```

□

**Exercise 47**  Define a function `concat : string -> string list -> string`, which takes a string `sep` and a string list `lst`, and returns one string with all the elements of `lst` concatenated together but separated by the string `sep`.<sup>11</sup> Some examples:

```
# concat ", " ["first"; "second"; "third"] ;;
- : string = "first, second, third"
# concat "... " ["Moo"; "Baa"; "Lalala"] ;;
- : string = "Moo...Baa...Lalala"
# concat ", " [] ;;
- : string = ""
# concat ", " ["Moo"] ;;
- : string = "Moo"
```

□

**Exercise 48**  Define a function `permutations : int list -> int list list`, which takes a list of integers and returns a list containing every permutation of the list. For example,

```
# permutations [1; 2; 3] ;;
- : int list list =
[[1; 2; 3]; [2; 1; 3]; [2; 3; 1]; [1; 3; 2]; [3; 1; 2]; [3; 2; 1]]
```

<sup>11</sup> The OCaml library module `String` provides just this function already under the same name.



*It doesn't matter what order the permutations appear in the returned list. Note that if the input list is of length  $n$ , then the answer should be of length  $n!$  (that is, the factorial of  $n$ ). Hint: One way to do this is to write an auxiliary function, `interleave : int -> int list -> int list list`, that yields all interleavings of its first argument into its second. For example:*

```
# interleave 1 [2; 3] ;;
- : int list list = [[1; 2; 3]; [2; 1; 3]; [2; 3; 1]]
```

You may find the `List` module functions `List.map` and `List.concat` helpful in this exercise. □

We've gone through the valuable exercise of writing a bunch of useful list functions. But list processing is so ubiquitous that OCaml provides a library module for just such functions. We'll discuss the `List` module further in Section 9.4.

## 7.4 Records

Tuples and lists use the *order* within a sequence to individuate their elements. An alternative, **RECORDS**, name the elements, providing each with a unique *label*. The type constructor specifies the labels and the type of element associated with each. For instance, suppose we'd like to store information about people: first and last name and year of birth. An appropriate record type would be

```
{lastname : string; firstname : string; birthyear : int}
```

Each of the elements in a record is referred to as a **FIELD**. Since the fields are individuated by their label, the order in which they occur is immaterial; the same type could have been specified reordering the fields as

```
{firstname : string; birthyear : int; lastname : string}
```

with no difference (except perhaps to add a bit of confusion to a reader expecting a more systematic ordering).

Unlike lists and tuples, which are built-in types in OCaml, particular record types are user-defined. OCaml needs to know about the type – its fields, their labels and types – in order to make use of them. Records are the first of the user-defined types we'll explore in detail in Chapter 11. To define a new type, we use the type construction to give the type a name:

$$\text{type } \langle \text{typename} \rangle = \langle \text{typeexpr} \rangle$$

We might name the type above `person`:

```
# type person = {lastname : string;
#               firstname : string;
#               birthyear : int} ;;
type person = { lastname : string; firstname : string; birthyear :
               int; }
```

Now that the type is defined and OCaml is aware of its fields' labels and types, we can start constructing values of that type. To construct a record value, we use the strikingly similar notation of placing the fields, separated by semicolons, within braces. In record value expressions, the label of a field is separated from its value by an `=`.<sup>12</sup> We define a value of the record type above:

```
# let ac =
#   {firstname = "Alonzo";
#     lastname = "Church";
#     birthyear = 1903} ;;
val ac : person =
  {lastname = "Church"; firstname = "Alonzo"; birthyear = 1903}
```

Notice that the type inferred for `ac` is `person`, the defined name for the record type.

As usual, we use pattern matching to decompose a record into its constituent parts. A simple example decomposes the `ac` value just created to extract the birth year.

```
# match ac with
# | {lastname = lname;
#   firstname = fname;
#   birthyear = byear} -> byear ;;
Characters 29-34:
  | {lastname = lname;
      ^^^^^
Warning 27: unused variable lname.
Characters 48-53:
  firstname = fname;
      ^^^^^
Warning 27: unused variable fname.
- : int = 1903
```

The warnings remind us that there are unused variables that should be explicitly marked as such:

```
# match ac with
# | {lastname = _lname;
#   firstname = _fname;
```

<sup>12</sup> A common confusion when first using record types concerns when to use `:` and when to use `=` within fields. Here's a way to think about the usages: The use of `:` in record *type* expressions evokes the use of `:` in a typing. In a sense, the type constructor provides a typing for each of the fields. The use of `=` in record *value* expressions evokes the use of `=` in naming constructs.

```
#    birthyear = byear} -> byear ;;
- : int = 1903
```

By way of example, we can define a function that takes a value of type `person` and returns the person's full name by extracting and concatenating the first and last names.

```
# let fullname (p : person) : string =
#   match p with
#   | {firstname = fname;
#      lastname = lname;
#      birthyear = _byear} ->
#     fname ^ " " ^ lname ;;
val fullname : person -> string = <fun>
```

This function can be used to generate the full name:

```
# fullname ac ;;
- : string = "Alonzo Church"
```

It's a bit cumbersome to have to mention every field in a record pattern match when we are interested in only a subset of the fields. Fortunately, patterns need only specify a subset of the fields, using the notation `_` to stand for any remaining fields.<sup>13</sup>

```
let fullname (p : person) : string =
  match p with
  | {firstname = fname; lastname = lname; _} ->
    fname ^ " " ^ lname ;;
```

Another simplification in record patterns, called **FIELD PUNNING**, is allowed for fields in which the label and the variable name are identical. In that case, the label alone is all that is required. We can use field punning to simplify `fullname`:

```
let fullname (p : person) : string =
  match p with
  | {firstname; lastname; _} ->
    firstname ^ " " ^ lastname ;;
```

As a final simplification, the syntactic sugar allowing single-pattern matches in `let` constructs allows us to eliminate the explicit match entirely:

```
# let fullname ({firstname; lastname; _} : person) : string =
#   firstname ^ " " ^ lastname ;;
val fullname : person -> string = <fun>

# fullname ac ;;
- : string = "Alonzo Church"
```

<sup>13</sup> In fact, the `_` notation isn't necessary, but it performs the useful role of capturing the programmer's intention that the set of fields is not complete. In fact, OCaml will provide a warning (when properly set up) if an incomplete record match isn't marked with this notation.

### 7.4.1 *Field selection*

Pattern matching permits extracting the values of all of the fields of a record (or any subset). When only one field value is needed, however, a more succinct technique suffices. The familiar dot notation from many programming languages allows selection of a single field.

```
# ac.firstname ;;
- : string = "Alonzo"
# ac.birthyear ;;
- : int = 1903
```

Thus, the `fullname` function could have been written as

```
# let fullname (p : person) : string =
#   p.firstname ^ " " ^ p.lastname ;;
val fullname : person -> string = <fun>
# fullname ac ;;
- : string = "Alonzo Church"
```

Which notation to use is again a design matter, which will depend on the individual case.

## 7.5 *Comparative summary*

These three data structuring mechanisms provide three different approaches to the same idea – agglomerating a collection of elements into a single unit. The differences arise in how the elements are individuated. In tuples and lists, an element is individuated by its *index* in an ordered collection. In records, an element is individuated by its *label* in a labeled but unordered collection.

Tuples and records collect a fixed number of elements. Because the number of elements is fixed, they can be of differing type. The type of the tuple or record indicates what type each element has. Lists, on the other hand, collect an arbitrary number of elements. In order to be able to operate on any arbitrary element, the types of all the elements must be indicated in the type of the list itself. This constraint is facilitated by having all elements have the same type, so that they can be operated on uniformly.

Table 7.1 provides a summary of the differing structuring mechanisms.

	Tuples			Records	Lists
element types	differing			differing	uniform
selected by	order			label	order
type constructors	$\langle \rangle * \langle \rangle$	$\langle \rangle * \langle \rangle * \langle \rangle$	$\dots$	$\{a : \langle \rangle ; b : \langle \rangle ; c : \langle \rangle ; \dots\}$	$\langle \rangle \text{ list}$
value constructors	$\langle \rangle , \langle \rangle$	$\langle \rangle , \langle \rangle , \langle \rangle$	$\dots$	$\{a = \langle \rangle ; b = \langle \rangle ; c = \langle \rangle ; \dots\}$	$[] \quad \langle \rangle :: \langle \rangle$

7.6 Problem set: The prisoners’ dilemma

Table 7.1: Comparison of three structuring mechanisms: tuples, records, and lists.

7.6.1 Background

I’m an apple farmer who hates apples but loves broccoli. You’re a broccoli farmer who hates broccoli but loves apples. The obvious solution to this sad state of affairs is for us to trade – I ship you a box of my apples and you ship me a box of your broccoli. Win-win.

But I might try to get clever by shipping an empty box. Instead of cooperating, I “defect”. I still get my broccoli from you (assuming you don’t defect) and get to keep my apples. You, thinking through this scenario, realize that you’re better off defecting as well; at least you’ll get to keep your broccoli. But then, nobody gets what we want; we’re both worse off. The best thing to do in this DONATION GAME seems to be to defect.

It’s a bit of a mystery, then, why people cooperate at all. The answer may lie in the fact that we engage in many rounds of the game. If you get a reputation for cooperating, others may be willing to cooperate as well, leading to overall better outcomes for all involved.

The donation game is an instance of a classic game-theory thought experiment called the PRISONER’S DILEMMA. A prisoner’s dilemma is a type of game involving two players in which each player is individually incentivized to choose a particular action, even though it may not result in the best global outcome for both players. The outcomes are commonly specified through a payoff matrix, such as the one in Table 7.2.

To read the matrix, Player 1’s actions are outlined at the left and Player 2’s actions at the top. The entry in each box corresponds to a payoff to each player, depending on their respective actions. For instance, the top-right box indicates the payoff when Player 1 cooperates and Player 2 defects. Player 1 receives a payoff of  $-2$  and Player 2 receives a payoff of  $5$  in that case.

To see why a dilemma arises, consider the possible actions taken by Player 1. If Player 2 cooperates, then Player 1 should defect rather than cooperating, since the payoff from defecting is higher ( $5 > 3$ ). If Player 2 defects, then Player 1 should again defect since the payoff from defecting is higher ( $5 > -2$ ). The same analysis applies to Player

		Player 2	
		Cooperate	Defect
Player 1	Cooperate	(3, 3)	( $-2$ , 5)
	Defect	(5, $-2$ )	(0, 0)

Table 7.2: Example payoff matrix for a prisoner’s dilemma. This particular payoff matrix corresponds to a donation game in which providing the donation (of apples or broccoli, say) costs 2 unit and receiving the donation provides a benefit of 5 units.

2. Therefore, both players are incentivized to defect. However, the payoff from both players defecting (each getting 0) is objectively worse for both players than the payoff from both players cooperating (each getting 3).

An **ITERATED PRISONER'S DILEMMA** is a multi-round prisoner's dilemma, where the number of rounds is not known.<sup>14</sup> A **STRATEGY** specifies what action to take based on a history of past rounds of a game. We can (and will) represent a history as a list of pairs of actions (cooperate or defect) taken in the past, and a strategy as a function from histories to actions.

For example, a simple strategy is to ignore the histories and always defect. We call that the “nasty” strategy. More optimistically is the “patsy” strategy, which always cooperates.

Whereas the above analysis showed both players are incentivized to defect in a single-round prisoner's dilemma (leading to the nasty strategy), that is no longer necessarily the case if there are multiple rounds. Instead, more complicated strategies can emerge as players can take into account the history of their opponent's plays and their own. A particularly effective strategy – effective because it leads to cooperation, with its larger payoffs – is **TIT-FOR-TAT**. In the tit-for-tat strategy, the player starts off by cooperating in the first round, and then in later rounds chooses the action that the other player just played, rewarding the other player's cooperation by cooperating and punishing the other player's defection by defecting.

In this problem set, you'll complete a simulation of the iterated prisoner's dilemma that allows for testing different payoff matrices and strategies.

<sup>14</sup> If the number of rounds is known by the players ahead of time, players are again incentivized to defect for all rounds. We will not delve into the reasoning here, as that is outside the scope of this course, but it is an interesting result!

### 7.6.2 Some practice functions

To get started, you'll write a series of functions that perform simple manipulations over lists, strings, numbers, and booleans. (Some of these may be useful later in implementing the iterated prisoner's dilemma.) See the comments in `ps1.ml` for the specifications. Give the functions the names listed in the comments, as they must be named as specified in order to compile against our automated unit tests.

The best way to learn about the core language is to work directly with the core language features. Consequently, you should not use any library functions in implementing your solutions to this problem set.

### 7.6.3 Unit testing

Thorough testing is important in all your work. Testing will help you find bugs, avoid mistakes, and teach you the value of short, clear functions. In the file `ps1_tests.ml`, we've put some prewritten tests for one of the practice functions using the testing method of Section 6.5. Spend some time understanding how the testing function works and why these tests are comprehensive. Then, for each function in Problem 1, write tests that thoroughly test the functionality of each of the remaining sections of Problem 1, covering all code paths and corner cases.

To run your tests, run the shell command

```
% make ps1_tests.byte
```

in the directory where your `ps1.ml` is located and then run the compiled program by executing `ps1_tests.byte`:

```
% ./ps1_tests.byte
```

The program should then generate a full report with all of the unit tests for all of the functions in the problem set.

### 7.6.4 The prisoner's dilemma

Having implemented some practice functions, you will apply functional programming concepts to complete a model of the **iterated prisoner's dilemma**, including the implementation of various strategies, including one of your own devising.

Follow the comments in `ps1.ml` for the specifications. Feel free to use any of the functions that you implemented in the earlier parts of the problem set.

All of the programming concepts needed to do the problem set have already been introduced. There's no need to use later constructs, and you should refrain from doing so. In particular, you'll want to avoid imperative programming, and you should not use any library functions.

### 7.6.5 Tournament

To allow you to see how your custom strategy fares, we will run a course-wide round-robin tournament in which your strategy will be run against every other student's strategy for a random number of rounds. To keep things interesting, we will add a small amount of noise to each strategy: 5% of the time, we will use the opposite action of the one a strategy specifies.

The tournament is pseudonymized and optional. If you want to participate, you'll provide a pseudonym for your entrant. If you

specify an empty pseudonym, we won't enter you in the tournament, though we encourage you to come up with clever strategies to compete against your fellow classmates! We'll post and regularly update a leaderboard, displaying each student's pseudonym and their strategy's average payoff. The top score will win a rare and wonderful prize!

#### 7.6.6 *Testing*

You should again provide unit tests in `ps1_tests.ml` for each function that you implement in Problem 2.

In addition, you can choose to uncomment the `main` function in `ps1.ml` and then recompile the file by running `make ps1.byte` in your shell. Then, run the command `./ps1.byte` and you should see via printed output that Nasty earns a payoff of 500 and Patsy earns a payoff of -200. You can change the strategies played to see how different strategies perform against each other.