Currying

Currying is a fundamental concept in functional programming, named in honor of the logician Haskell Curry[1], although it was introduced much earlier[2].

Currying has its foundations on the idea that functions can return functions. For instance imagine the following:

```
let add x = fun y -> x + y
```

So add is a function of one argument, that returns a function of one argument. To call add we could do:

```
(add 3) 2                    (* becomes 3 + 2 *)
```

And in fact function application is left-associative, so we can write this also as:

```
add 3 2
```

From a different point of view, we can effectively think of add as a function of two arguments, x and y. Except that we are effectively providing these arguments *one at a time*, first x and then y. We can in fact stop after the "first" argument:

```
add 3
```

and what we get back is a function, that expects the "second" argument, y. We will call these *curried* functions.

OCAML offers us a convenient syntactic sugar for curried functions, allowing us to write our example of add above as:

```
let add x y = x + y
```

Technically the idea of currying is the process of taking a function that takes two or more arguments as a tuple, and producing the function that instead takes them one after the other. So for example going from the first function to the second:

```
let add (x, y) = x + y
let addc x y = x + y
```

In fact we can represent this transformation as a function, if we know the number of arguments involved, and there are many ways to write it:

```
let curry f = fun x y -> f (x, y)
let curry f = fun x -> fun y -> f (x, y)
let curry f = fun y -> f (x, y)
let curry f x y = f (x, y)
```

Practice 1: What is the type of the curry function?

Practice 2: Write an uncurry function after first describing its type.

[1] https://en.wikipedia.org/wiki/Haskell_Curry
[2] https://en.wikipedia.org/wiki/Currying

Functions of multiple arguments in OCAML

Essentially we now know almost everything about functions in OCAML:

1. Every function in OCAML takes exactly one argument. This will make it easy to implement the interpreter in the future, it matches the point of view taken by lambda calculus, and it is wonderfully simple and elegant.
2. Having that argument be of type `unit`, with its unique value `()`, provides a way to have functions that essentially "take no argument" without having to code them in any special way.
3. Having that argument be of tuple type offers us one way to write a "function of multiple arguments", by having as its one argument a tuple `(x, y, z)` and using pattern-matching.
4. Having functions (and closures) as possible return values allows us to have multiple arguments in the curried sense described above, where we provide those arguments one at a time and each time we get as a return value a function. It is the fact that we have closures that makes this possible, as the returned values remember the values of the parameters to the function call that produced them.