Functions as Values

When making larger programs, it is necessary to reuse other programs. There are many ways of doing this.

It is possible to pass a function to another function as an argument. For example, the function `map` below applies function f to every element in list l.

```
map : ('a -> 'b) -> 'a list -> 'b list

let rec map f l =
  match l with
    [] -> []
  | h::t -> f h :: map f t
```

It is also possible to use a function within another function without assigning a name in the namespace to the argument function. We call this type of function an *anonymous function*. The syntax is `fun name -> expression`. This can be used when a function is only applied in one place and is short. For example, below we define a function that returns true for an element if it's even.

```
evens : int list -> bool list

let evens l =
  map (fun x -> x mod 2 = 0) l
```

OCaml allows you to convert an operator into a function with the syntax `( operator )`. For example, `( <= ) 3 4` has the value `true` and `( + ) 3 4` has the value `7`.

1 Partial Application

In reality, functions that take multiple arguments are really multiple single argument functions applied to each other. That is, a function f a b has type $\alpha \rightarrow \beta \rightarrow \gamma$, which we can write as $\alpha \rightarrow (\beta \rightarrow \gamma)$. Thus it takes an argument of type α and returns a function of type $\beta \rightarrow \gamma$, which in turn takes an argument of β and returns a value of type γ. This can be written explicitly as `let f = fun a -> fun b -> ...` instead of just `let f = ...`.

As a result, it is possible to apply fewer than the total number of arguments to a function. This is called *partial application*. We now introduce various examples to demonstrate this.

The simplest example is `let add x y = x + y`. If we write `let f = add 6`, f is now a function that adds 6. Notice that add has type int -¿ int -¿ int and f (and by extension add 6) has type int -¿ int.

Recall that it is possible to produce a function from an operator with the syntax `( / )`. This fact can be used in conjunction with partial application to create, for example, a function which returns 2 divided by its input: `let fun = ( / ) 2`. This can be applied to all elements of a list using map.

Consider a function that maps a function over a list of lists:

```
mapl : ('a -> 'b) -> 'a list list -> 'b list list

let rec mapl f l =
  match l with
    [] -> []
  | h::t -> map f h :: mapl f t
```

This can be rewritten using partial application: `let mapl f l = map (map f) l` or even `let mapl f = map (map f)`. In this case, `map (map f)` has type α list list -¿ β list list.

Recall the member function, which checks if some value is in a given list:

```
member : 'a -> 'a list -> bool

let rec member n l =
  match l with
    [] -> false
  | h::t -> n = h || member n t
```

We want to write a similar function that checks if some element is in every list in a list of lists. We can solve this using partial application of member. `member x` will have type α list -¿ bool so `map (member x)` has type α list list -¿ bool list. Thus,

```
member_all : 'a -> 'a list list -> bool

let member_all x ls =
  let booleans = map (member x) ls in
    not (member false booleans)
```

.

If we wanted to make a function which shortens all lists in a list of lists by a given length, we can do the following:

```
new_len : int -> 'a list -> 'a list
truncate : int -> 'a list list -> 'a list list

let new_len n l =
  try take n l with
    Invalid_argument _ -> l

let truncate n l =
  map (new_len n) l
```