==========

Higher-order functions (aka "functional forms")

Take a function as argument, or return a function as a result.

We saw some examples in previous lectures. In the DFA simulation program, for example, the `find` function took a predicate as its first argument.

```
find (fun x -> x*x > 100) [7; 9; 11; 13]  => 11
```

If you ask OCaml to print the type of `find` it will say

```
('a -> bool) -> 'a list -> 'a
```

That is, `find`'s first argument is a function from `'a` to `bool` (i.e., a predicate), its second argument is a list of `'a` objects, and its return value is the first object in the list for which the predicate returns true.

Other examples:

```
map (fun x -> x*x) [2; 3; 5; 7]    => [4; 9; 25; 49]
```

`compose` (not pre-defined in some implementations)

```
let compose f g = fun x -> f (g x);;
(* or just *)
let compose f g x = f (g x);;
```

```
(compose hd tl) [1; 2; 3]    => 2
```

Folding (reduction)

```
fold_left ( * ) 1 [2; 3; 5; 7]     => 210
   (* note the spaces around * -- so it's not a comment *)
   (* as is typical in uses of this function, 1 is the identity element
      for multiplication; in nested calls the corresponding argument
      will be a subtotal *)
```

Folding is predefined, but we could have declared it as

```
let rec fold_left f i l =
  match l with
  | [] -> i
  | h :: t -> fold_left f (f i h) t;;
```

There's also a `fold_right`, but it isn't as used as much, because it isn't tail recursive.

Higher-order functions are great for building other functions:

```
let total l = fold_left ( + ) 0 l;;
(* or just *)
let total = fold_left ( + ) 0;;
```

```
total [1; 2; 3; 4; 5]           => 15

let total_all ll = map total ll;;
(* or just *)
let total_all = map total;;

total_all  [[1; 2; 3; 4; 5];
            [2; 4; 6; 8; 10];
            [3; 6; 9; 12; 15]]     => [15; 30; 45]
```

Currying

Applying a function to only some of its arguments in order to produce a function that expects the rest of the arguments.

Named after the logician Haskell Curry (the same guy Haskell is named after).

Automatic in ML family languages (takes some effort in Lisp)

```
let total = fold (+) 0;;

let plus3 = ( + ) 3;;

plus3 4      => 7

let plusn n = fun k -> n + k;;
let inc = plusn 1;;

let plus3 = plusn 3;;
inc 5              => 6

let comb a b = fun x y -> a * x + b * y;;
let comb23 = comb 2 3;;
comb23 5 6               => 28
```

NB: `plusn` and `comb` require ***unlimited extent*** (covered in chapter 3)

So what is going on here?

```
let ave a b = (a +. b) /. 2.0;;
```

is shorthand for

```
let ave = fun a -> fun b -> (a +. b) /. 2.0;;
```

This explains why OCaml says that the type of ave (even with the first definition) is "`float -> float -> float`".

```
val ave : float -> float -> float = <fun>
```

Juxtaposition helps makes things really clean.
When I say

```
ave a b
```

do I mean (in mathematical notation)

```
ave (a b)
```

or

```
(ave (a)) (b)    ?
```

It doesn't really matter!
I do need to be aware what's going on, though, because if I give a function two few arguments the error message will usually be "type clash", rather than "too few arguments".

As an assignment using functional programming, I often provide students with a scanner and a parser generator in OCaml, and ask them to extend it to build a simple interpreter or compiler. As an example of the use of functional forms, here's an except from the parser generator.

Grammars are represented as a list of pairs, in which the first element of each pair is a nonterminal and the second element is a list of right-hand sides for productions with that nonterminal on the left-hand side. Each right-hand side is itself a list of symbols.

Here's the calculator grammar:

```
# calc_gram;;
- : (String.t * String.t list list) list =
[("P", [["SL"; "$$"]]);
("SL", [["S"; "SL"]; []]);
("S", [["id"; ":="; "E"]; ["read"; "id"]; ["write"; "E"]]);
("E", [["T"; "TT"]]);
("T", [["F"; "FT"]]);
("TT", [["ao"; "T"; "TT"]; []]);
("FT", [["mo"; "F"; "FT"]; []]);
("ao", [["+"]; ["-"]]);
("mo", [["*"]; ["/"]]);
("F", [["id"]; ["num"]; ["("; "E"; ")"]])]
```

Parse tables (for a table-driven top-down parser) are also a list of pairs, and again the first element of each pair is a nonterminal. The second element of each pair is more complicated: it's a nested list of pairs, in which the first element of each pair is a predict set (a list of terminals) and the second element is the right-hand side to predict when an element of the predict set is seen on the input.

Here's the table for the calculator grammar:

```
# get_parse_table calc_gram;;
- : (String.t * (String.t list * String.t list) list) list =
[("P", [(["$$"; "id"; "read"; "write"], ["SL"; "$$"])]);
("SL", [(["id"; "read"; "write"], ["S"; "SL"]); (["$$"], [])]);
("S",
[(["id"], ["id"; ":="; "E"]);
 (["read"], ["read"; "id"]);
 (["write"], ["write"; "E"])
]);
("E", [(["("; "id"; "num"], ["T"; "TT"])]);
("T", [(["("; "id"; "num"], ["F"; "FT"])]);
("TT",
[(["+"; "-"], ["ao"; "T"; "TT"]);
 (["$$"; ")"; "id"; "read"; "write"], [])
]);
("FT",
[(["*"; "/"], ["mo"; "F"; "FT"]);
 (["$$"; ")"; "+"; "-"; "id"; "read"; "write"], [])
]);
("ao", [(["+"], ["+"]); (["-"], ["-"])]);
("mo", [(["*"], ["*"]); (["/"], ["/"])]);
("F", [(["id"], ["id"]); (["num"], ["num"]); (["("], ["("; "E"; ")"])])
]
```

Now suppose we have a parse table and we'd like to extract the original grammar. Here's a function that does so:

```
let grammar_of parse_tab =
  map (fun p -> (fst p,
                 (fold_left (@)
                            []
                            (map (fun (a, b) -> [b]) (snd p)))))
      parse_tab;;
```

If you understand how it does that, you're probably in good shape for an OCaml project. If you don't understand it, you should study it carefully, review Sec. 10.5 in the text, bring it up in workshop, or talk to the TA or instructor.