==========

Introduction to OCaml

ML dialect developed and maintained by researchers at INRIA,
the French national CS research institute
compiler or interpreter (your choice)
ocamlc ocaml

Interpreter runs a read-eval-print loop (REPL) much like Scheme or Python.
#use "file.ml" load source code
#load "library.cma" load binary library

simple data types
`bool`, `int`, `float`, strings, tuples (pairs &c), lists
`+`, `*`, etc. vs `+.`, `*.`, etc.
`float` constants must contain a decimal point
`^` (string concatenation)
`fst` & `snd`
work ***only*** on two-element tuples (else type error)
`hd`, `tl` (deprecated: prefer pattern matching)
`::` and `@` (cons and append)

Lists are delimited with square brackets; elements are separated by semicolons.
Tuples are delimited with parentheses; elements are separated by commas.
Records (more later) are delimited with braces; elements are separated by semicolons.
Arrays are deliminted with [| and |]; elements are separated by semicolons.

"structural" (same value; aka "deep") vs
"physical" (same instance; aka "shallow") equality
=, <> structural
`2 = 2; "foo" = "foo"; [1;2;3] = [1;1+1;5-2]`
==, != physical
`2 == 2; "foo" != "foo"; [1;2;3] != [1;1+1;5-2]`

ordering (<, >, <=, >=) are defined on all non-function types. They do what you'd expect on arithmetic types, Booleans, characters, strings, and tuples, but may not make much sense on others.

type inference -- more on this in Chapter 7
Type declarations are optional.
Compiler ***infers*** types when declarations are omitted.
Type checking amounts to checking for consistent usage.
Can't treat something as a string in one place and a number or a list somewhere else.

lexical conventions
identifiers made from `a-zA-Z0-9_'`
must start with a letter or underscore
constructors, variant names, modules, and exceptions have to start with an upper case letter
everything else starts with a lower case letter or underscore

`(* (* comments *) nest *)`

Top-level forms terminated by `;;`
This tells the REPL to interpret.

functions

```
let f a1 a2 a3 = ...
let f (a1:t1) (a2:t2) (a3:t3) : rt = ...
let f: t1 -> t2 -> t3 -> rt = fun a1 a2 a3 -> ...
```

Those three versions are equivalent, though the first is implicitly typed.

```
let rec f = ...

let rec g = ...     (* for mutually
and h = ...             recursive functions *)
```

pattern matching

```
match expr with
  var1 -> expr1
| var2 when pred2 -> expr2
| ...
| _ -> exprN
```

Match is sort of like `case` or `switch` on steroids.

also works in other contexts, e.g. `let (s, t, f) = my_tuple;;`

or function definitions: the (bad) Fibonacci example above

```
let rec fib1 n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> fib1 (n-1) + fib1 (n-2);;
```

can be rewritten

```
let rec fib1 = function
  | 0 -> 1
  | 1 -> 1
  | n -> fib1 (n-1) + fib1 (n-2);;
```

arrays

```
let primes5 = [| 2; 3; 5; 7; 11 |];;
```

`.()` subscripting

```
primes5.(2)        => 5
```

elements are mutable (unlike those of lists and tuples)
assignment uses left arrow:

```
primes5.(2) <- 12345;;       => ()
```

strings
like arrays of characters, but with double-quoted literals.
Were mutable in older versions of the language. That's now deprecated.
If you need mutability, use **`bytes`** instead.

records
like tuples, but with fields that are named instead of positional
can declare fields to be mutable (immutable by default)

```
type widget = {name: string; sn: int; mutable price: float};;
let g = {name = "gear"; sn = 12345; price = 23.45};;
   g.name => "gear"
g.price <- 34.56;;    (* inflation *)
```

variants

```
type 'a bin_tree = Empty
                 | Node of 'a * 'a bin_tree * 'a bin_tree;;
...
let rec inorder = function
| Empty -> []
| Node(v, left, right) -> inorder left @ [v] @ inorder right;;
```

side effects
<- (mutable) record field assignment (not allowed in project)
:= and ! refs (like pointers; also not allowed in project)
I/O

```
read_line, read_int, read_float
print_int, print_float,
   print_char, print_string, print_newline,
prerr_int, prerr_float, prerr_string, prerr_newline
```

`Printf` module

```
printf
sprintf
```

`Sys.argv`
exceptions

```
exception Foo of string;;
raise (Foo "ouch")
try expr1 with Foo -> expr2
```

Extended example from the text: simulation of a DFA.

We'll invoke the program by calling a function called '`simulate`',
passing it a DFA description and an input string.
The automaton description is a record with three fields: the start state, the transition function, and a list of the one or more final states. We can trivially make it polymorphic in the type of input symbols:

```
type state = int;;
type 'a dfa = {
  current_state : state;
  transition_function : (state * 'a * state) list;
  final_states : state list;
};;
type decision = Accept | Reject;;
```

We've named the first field "`current_state`" instead of "`start_state`" for reasons that will become apparent in a minute.

The transition function is represented by a list of triples.
The first element and third elements of each triple are the from and to states; the second element is the input symbol that transitions between them.

For example, consider the DFA

```
let a_b_even_dfa : char dfa =   (* input symbols are characters *)
  { current_state = 0;
    transition_function =
      [ (0, 'a', 2); (0, 'b', 1); (1, 'a', 3); (1, 'b', 0);
        (2, 'a', 0); (2, 'b', 3); (3, 'a', 1); (3, 'b', 2) ];
    final_states = [0];
  };;
```

This machine accepts strings containing an even number of a's and an even number of b's.

If we type

```
simulate a_b_even_dfa ['a'; 'b'; 'b'; 'a'; 'b'];;
```

then the OCaml interpreter (read-eval-print loop) will print

```
- : state list * decision = ([0; 2; 3; 2; 0; 1], Reject)
```

If we change the input string to abaaba it will print

```
- : state list * decision = ([0; 2; 3; 1; 3; 2; 0], Accept)
```

Here is the program:

```
open List;;     (* includes rev, find, and mem functions *)

let move (d:'a dfa) (x:'a) : 'a dfa =
  { current_state = (
      let (_, _, q) =
        find (fun (s, c, _) -> s = d.current_state && c = x)
             d.transition_function in
      q);
    transition_function = d.transition_function;
    final_states = d.final_states;
  };;

let simulate (d:'a dfa) (input:'a list)
   : (state list * decision) =
  let rec helper moves d2 remaining_input
     : (state option * state list) =
    match remaining_input with
    | [] -> (Some d2.current_state, moves)
    | hd :: tl ->
        let new_moves = d2.current_state :: moves in
          try helper new_moves (move d2 hd) tl
          with Not_found -> (None, new_moves) in
  match helper [] d input with
  | (None, moves) -> (rev moves, Reject)
  | (Some last_state, moves) ->
     ( rev (last_state :: moves),
      if mem last_state d.final_states
        then Accept else Reject);;
```

The basic idea is this: simulate takes a DFA and an input string as argument. If the input string is empty, it checks to see if the start state of the DFA is a final state. If the input string is not empty, simulate calls itself recursively, passing a one-symbol-shorter input string and a DFA that has been modified to have a different start state, namely the one that the old DFA moved to when given the initial input symbol.