The basic approach:
Rather than modify existing code, like an imperative language, we glue functions together.
Advantages:

- compile-time checking for types (like C++, Java, & other "Professional" languages) – this avoids exceptions!
- types need not be specified (like scheme, python, & other scripting languages)

```
Thread x = new Thread (); //Java
```

  Note that pure interpreters would fail this, since they can't infer well.

```
let x = Thread ();;
(* OCaml uses type inference *)
```

- memory need not be manually managed (like Java, not C).
- good support for concurrency
- good support for higher-order functions

Our book chooses to use ML rather than OCaml, so why do we use OCaml?

- Intel & other companies use OCaml.
- OCaml is cleaner than ML
- It is good to see multiple forms of the same language

```
(* ML -> OCaml *)
x<y andalso z<w -> x<y \&\& z<w
[1, 3, 5] -> [1; 3; 5]
val x = 5 -> let x = 5
3.0 + 4.0 -> 3.0 +. 4.0
```

Note that OCaml's type inference is not as fancy as ML's.

Properties of Code:

There is no redefinition of variables; only addition of new definitions.

```
# let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 2;;
- : int 3
(* f uses the definition of x within its code block *)
```

OCaml checks all types even if the code is unreachable.

```
# if 1<2 then "a" else "b";;
- : string "a"
#if 1<2 then "a" else 0;;
(* ERROR: OCaml is conservative,
   and assigns types to entire statements;
   We can get errors on non-errors, but if it compiles.
   we are guaranteed that there will be no type errors *)
```

Tuples have set length, lists have set type.

```
# (1, "a");;
- : int * string = (1, "a")
# [1; "a"];;
(* ^^^ ERROR: this expr has type string
but expr was expected of type int *)
# [1; 10; -3; 4*7] = [1; 7];;
- : bool = false
# (1, 10, -3, 4*7) = (1, 10);;
(* ^^^^^^^ ERROR: this expr has type int * int
but expr was expected of type int * ... * int *)
# [1; 5; -3] = [];;
```

```
- : bool = false
# ["a"; "b"; "c"] = [];;
- : bool = false
(* huh? How is [] both int list and char list? *)
# [];;
- : 'a list = []
(* this is called a generic type and 'a is a type variable *)
```

Functions take 1 parameter and return 1 result

```
# let f = fun x -> x+1;;
val f = int -> int = <fun>
(* <fun> is a stand in for the tree
or machine/byte code OCaml uses internally *)
# let tup = (3, 12);;
val tup = int * int = (3, 12)
# let add = fun (x, y) -> x + y;;
vall add : int * int -> int
# add tup;;
-: int = 15
(* a tuple is a way of emulating multiple parameter functions *)
```

Currying is the canonical way to emulate multi-variable functions

```
(* say we are attempting to emulate Lisp's 'cons' keywork *)
(* we do this with currying: using functions that pass/return functions *)
# let ccons = (fun a -> (fun b -> (a::b)));;
val ccons : 'a -> 'a list -> 'a list = <fun>
(* we can use this in the standard two-variable way: *)
# ccons 2 [5; 9; 27];;
-: int list = [3; 5; 9; 27]
(* but we can also take advantage of the function return value *)
# let prepend_3 = (ccons 3);;
val prepend_3 : int list -> int list
(* thus we can create various functions without recompilation! *)
(* this cannot be done with either element of a tuple! *)
(* in practice, OCaml lets us simplify the declaration *)
# let ccons = fun a b -> a::b;;
val ccons : 'a -> 'a list -> 'a list = <fun>
(* because of this, function calls are left-associative:
   f g h = ((f g) h)
   but function types are right-associative:
   int -> float -> int = (int -> (float -> int))
   and the keyword 'list' has the highest priority \& '->' the lowest
   int -> float * float list = int -> (float * (float list)) *)
```

Pattern matching is the canonical way to test expressions

```
# match 1 with
| int n -> true
| _ -> false ;;
(* this returns true for ints and false otherwise *)
(* we can use this in function parameter lists as shorthand *)
# let car (h::_) -> h;;
# let car = fun l -> match l with | (h::_) -> h;;
(* ^^^^^^^^^^ WARNING:
   This pattern matching is not exhaustive
   Here is an example of a case that is not matched:
   [] *)
val car : 'a list -> 'a list = <fun>
(* this compiles, but throws an exception if we violate it *)
# cdr [];;
Exception: Match_failure("//toplevel//", 1, 8).
(* we would thus like to make car safer : *)
# let safer_car = fun l -> match l with
| (h::_) -> h
| _ -> 0;;
```

```
val safer_car : int list -> list = <fun>
(* this only works for integers, we would like it to be more general *)
# let scar = fun d -> fun l = match l with
| h::_ -> h
| [] -> d;;
val scar : 'a -> 'a list -> 'a = <fun>
(* we can apply this specifically to turn it into our earlier safer_car *)
# let safer_car = scar 0;;
val safer_car = int -> int list -> int = <fun>
```

One can define custom types (in particular, unions).

If we were to define a custom type in C

```
union u (long l; char *p;);
union u v;
// both of the following share an address location
v.l = 12;
v.p = "abc";
// so consider a function
int f(union u a) { // is a.l valid or is a.p? We can't know!

(* in OCaml, data type is known
   note that type constructors in OCaml are capitalized *)
# type mytype =
| Foo
| Bar of int
| Baz of int * int;;
(* this leaves us with three constructors: # Foo; # Bar 12; #Baz (3, 5);
   we can use these constructors for pattern matching \& thus determine type *)
# match myvalue with
| Foo -> 0
| Bar x -> x
| Baz (y, z) -> y + z;;
(* these unions can be generic: *)
# type 'a option = | None (* None = type 'a option *)
| Some of 'a;; (* Some + 'a = datatype *)
(* this typing can be used to define an option for any type, string included
   We have thus defined a safer version of the nullptr;
   The nullptr can give runtime errors,
   but the OCaml compiler would give thisto us as a compile-time error *)
(* we can even define types recursively *)
# type 'a list =
| []
| 'a::'a list;;
```

We can use union pattern matching to define “polymorphic” functions:

```
# fun x -> match x with
| None -> 0
| Some y -> y;;
(* this is common enough that we have a shorthand *)
# function | None -> 0 | Some y -> y
(* this is the second major shorthand we have seen*)
# fun x y -> x * x + y = fun x -> fun y -> x * x + y
(* we could combine these notations; the following 3 are equivelant *)
# let car (h:::_) -> h
# let car = fun h_::_ -> h
# let car = fun x -> match x with (h::_) -> h
(* BUT we don't; we use fun for currying \& function for recursion*)
```

OCaml is very conducive to recursion

```
(* let's try our canonical recursive function; reversing a list *)
# let reverse = function
| [] -> []
| h::t -> (reverse t)@h;;
Characters 69-76:
```

```
| h::t (reverse t)@h
Error: unbound value reverse
(* OCaml has an iron-clad rule -- no usage pre-definiton
   thus even x = x + 0 throws an error
   We need this functionality for recursive functions,
   so we use keyword rec *)
# let rec reverse = function
| [] -> []
| h::t -> (reverse t)@h;;
val reverse : 'a list list -> 'a list = <fun>
(* that's not right; We want 'a list -> 'a list! See: *)
# reverse [[3;4]; [-1;-3]; [5]; [6;9;12]];;
int list = [6; 9; 12; 5; -1; -3; 3; 4]
(* the static type checking caught our mistake! *)
# let rec reverse = function
| [] -> []
| h::t -> (reverse t)@h;;
(* this works, but @ is O(n), so reverse is O(n^2); we need better!
   we need an accumulator; solve a specific problem by solving a general
   this is the opposite of C, in which we get more specific to get speed *)
# let revapp a = function
| [] -> a
| h::t -> (* ? *);;
# let reverse l = revapp l [];;
(* we just need to determine what goes in place of the question mark
   |h|t| |a| -> |t'|t|a| -- |t|a| is cheap to compute! Thus our function is:
   PLUS the program doesn't have to save stack since it won't return! *)
# let rec apprev a = function
| [] -> a
| h::t -> apprev (h::a) t;;
```

Function type definition is only as general as the most specific operator/variable.

```
(* we would like to find the minimum value in a list *)
# let rec minlist = function
| h::t -> let m = minlist t in
if h<m then h else m
| [] -> (* ? *);;
(* if we were doing sumlist, we would use the addition identity 0;
   if we had a minlist identity we would use it, we instead pass this to user *)
# let rec minlist id = function
| h::t -> let m = minlist id t in
if h<m then h else m
| [] -> id;;
val minlist : int list -> int = <fun>
(* (<) binds the function to integers; we must pass the comparison to user! *)
# let rec minlist lt id = function
| h::t -> let m = minlist lt id t in
if lt h m then h else m
| [] -> id;;
val minlist ('a -> 'a -> bool) -> 'a -> 'a list -> 'a = <fun>
# minlist (<) 100000000 [3; -5; 7; -20];;
- : int = -20
```