Lists are constructed using brackets or the cons operator. Any given list has members of all the same type.

Use "fun" to define a function. ML responds by telling you the "type" of the new function you've created. This one is a function from two integers to an integer.

Instead of applying a function with syntax "f(x,y)", OCaml uses syntax "(f x y)".

Functions are defined in cases. Each case is pattern-matched in the order given.

When possible, OCaml gives functions a polymorphic type. Polymorphic functions can be applied to arguments of any type.

However, there must be some consistent way to instantiate each type variable. Here we see an example where no such instantiation exists and the compiler therefore rejects the code.

List arguments can also have polymorphic type.

Use "fn" to create anonymous (i.e., unnamed) functions. "fn ... => ..." is the same as if you typed "fun foo ... = ...;;" and then used "foo".

Using anonymous functions, you can build and return functions as values at runtime.
```ml
- val cool = (compose (fn n -> n+1)
  = (fn n -> n*2));;
val cool = fn : int -> int
- cool 1;;
val it = 3 : int

- fun addx x = (fn y => x+y);;
val addx = fn : int -> int -> int
- addx 1 2;;
val it = 3 : int

- val add = (fn x => (fn y => x+y));;
add : int -> int -> int
- add 1 2;;
val it = 3 : int

- val insert cmp x [] = [x]
  = | insert cmp x (h::t) = 
    =   if (cmp x h) then (h::(insert cmp x t))
    = else (h::(insert cmp x t));;
val insert = fn : ('a -> 'a -> bool) -> 'a
  -> 'a list -> 'a list
- fun isort cmp [] = []
  = | isort cmp (h::t) = 
    =   insert cmp h (isort cmp t);
val isort = fn : ('a -> 'a -> bool) -> 'a
  list -> 'a list
- isort (fn x=>fn y=> x<y) [15,2,27,16];;
val it = [2,15,16,27] : int list
- isort (fn x=>fn y=> x>y) [15,2,27,16];;
val it = [27,16,15,2] : int list

- fun count 0 = []
  = | count n = n::(count (n-1));;
val count = fn : int -> int list
- count 5;;
val it = [5,4,3,2,1] : int list

- fun app [] y = y
  = | app (h::t) y = h::(app t y);;
val app = fn:'a list -> 'a list -> 'a list
- fun rev [] = []
  = | rev (h::t) = app (rev t) [h];;
val rev = fn : 'a list -> 'a list

- fun count a 0 = a
  = | count a n = count (n::a) (n-1);;
val count = fn:int list -> int -> int list
- count [] 5;;
val it = [1,2,3,4,5] : int list
```

---

- An anonymous function may refer to variables declared in outer scopes.
- Actually “fun foo x y = ...” is just an abbreviation for “val foo = (fn x => (fn y => ...))”. If you give such a function fewer arguments than it expects, it yields a function from the remaining arguments to the original value. Functions written this way are called “curried functions”. Applying fewer arguments is called “partial evaluation”.
- Let’s use the above to implement a polymorphic insertion sort function.
- Immutable variables do not hinder the power of the language. Here is an example of counting with immutable variables.
- To count in the other direction, we can use list-reverse. To get list-reverse, we first need an implementation of list-append.
- The above implementations are not tail-recursive because they generate O(n)-height call stacks. Here’s a tail-recursive implementation of count whose maximum stack height is O(1). Dr. Gupta will cover tail-recursion next time.