A polymorphic append

```haskell
let rec append (l1: 'a list) (l2: 'a list) =
  if l1 = [] then l2
  else List.hd l1 :: append (List.tl l1) l2;
val append : 'a list -> 'a list -> 'a list = <fun>
```

# append [4; 3; 2] [6; 6; 7];
- : int list = [4; 3; 2; 6; 6; 7]

```haskell
let append ["cat"; "in"] ["the"; "hat"];
- : string list = ["cat"; "in"; "the"; "hat"]
```

A polymorphic `rev`

```haskell
let rec revaux (l: 'a list) (res: 'a list) =
  if l = [] then res
  else revaux (List.tl l) (List.hd l :: res);
val revaux : 'a list -> 'a list -> 'a list = <fun>
```

# let revaux (l: [false; true]);
- : bool list = [true; false]

Polymorphism

This version of `last` is said to be **polymorphic**, because it can be applied to many different types of arguments. ("Poly" = many, "morph" = shape.)

Note that the type of the elements of `l` is `'a` (pronounced "alpha"). This is a **type variable**, which can **instantiated**, each time we apply `last`, by replacing `'a` with any type that we like. The instances of the type `'a list -> 'a` include

<table>
<thead>
<tr>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int list -&gt; int</td>
</tr>
<tr>
<td>string list -&gt; string</td>
</tr>
<tr>
<td>int list list -&gt; int list</td>
</tr>
<tr>
<td>etc.</td>
</tr>
</tbody>
</table>

In other words,

```
lst : 'a list -> 'a
```

can be read, "`last` is a function that takes a list of elements of any type `alpha` and returns an element of `alpha"."
Polymorphic repeat

```ocaml
# let rec repeat (k:'a) (n:int) = (* A list of n copies of k *)
  if n = 0 then []
  else k :: repeat k (n-1);;

# repeat 7 12;;
- : int list = [7; 7; 7; 7; 7; 7; 7; 7; 7; 7]

# repeat true 3;;
- : bool list = [true; true; true]

# repeat [6;7] 4;;
- : int list list = [[6; 7]; [6; 7]; [6; 7]; [6; 7]]
```

What is the type of repeat?

Palindromes

A palindrome is a word, sentence, or other sequence that reads the same forwards and backwards.

```ocaml
# let palindrome (l: 'a list) =
  1 = (rev l);;
val palindrome : 'a list -> bool = <fun>

# palindrome [1; 2; 4; 2; 1];;
- : bool = true

# palindrome [true; true; false];;
- : bool = false

# palindrome ["a";"b";"l";"e"; "w";"a";"e"; "I"; "e";"r";"e"; "I";
  "g";"a";"w"; "e";"l";"b";"a"];
- : bool = true
```

map: “apply-to-each”

OCaml has a predefined function List.map that takes a function f and a list l and produces another list by applying f to each element of l. We’ll soon see how to define List.map, but first let’s look at some examples.

```ocaml
# List.map square [1; 3; 5; 9; 2; 21];;
- : int list = [1; 9; 25; 81; 4; 441]

# List.map not [false; false; true];;
- : bool list = [true; true; false]
```

Note that List.map is polymorphic: it works for lists of integers, strings, booleans, etc.

More on map

An interesting feature of List.map is its first argument is itself a function. For this reason, we call List.map a higher-order function.

Natural uses for higher-order functions arise frequently in programming. One of OCaml’s strengths is that it makes higher-order functions very easy to work with.

In other languages such as Java, higher-order functions can be (and often are) simulated using objects.
Another useful higher-order function is `List.filter`. When applied to a list 1 and a boolean function p, it extracts from 1 the list of those elements for which p returns true.

```ocaml
# let rec even (n:int) =  
  if n=0 then true  
  else if n=1 then false  
  else if n<0 then even (-n)  
  else even (n-2); 
val even : int -> bool = <fun>

# List.filter even [1; 2; 3; 4; 5; 6; 7; 8; 9];; 
- : int list = [2; 4; 6; 8]

# List.filter palindrome [[1]; [1; 2; 3]; [1; 2; 1]; []];; 
- : int list list = [[1]; [1; 2; 1]; []]
```

Note that, like `map`, `List.filter` is polymorphic—it works on lists of any type.

Similarly, we can define our own `filter` that behaves the same as `List.filter`.

```ocaml
let rec filter (p: 'a->bool) (l: 'a list) =  
  if l = [] then []  
  else if p (List.hd l) then List.hd l :: filter p (List.tl l)  
  else filter p (List.tl l)
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

The type of `map` is probably even more polymorphic than you expected! The list that it returns can actually be of a different type from its argument:

```ocaml
# map String.length ["The"; "quick"; "brown"; "fox"];; 
- : int list = [3; 5; 5; 3]
```
Approaches to Typing

- A **strongly typed** language prevents programs from accessing private data, corrupting memory, crashing the machine, etc.
- A **weakly typed** language does not.
- A **statically typed** language performs type-consistency checks at when programs are first entered.
- A **dynamically typed** language delays these checks until programs are executed.

<table>
<thead>
<tr>
<th>Weak</th>
<th>Strong</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>PERL, Lisp, Scheme</td>
</tr>
<tr>
<td>Static</td>
<td>C, C++, ML, ADA, Java*</td>
</tr>
</tbody>
</table>

*Strictly speaking, Java should be called “mostly static”

Practice with Types

What are the types of the following functions?

- `let f (x:int) = x + 1`
- `let f x = x + 1`
- `let f (x:int) = [x]`
- `let f x = [x]`
- `let f x = x`
- `let f x = hd(tl x) :: [1,0]`
- `let f x = hd(tl x) :: []`
- `let f x = 1 :: x`
- `let f x y = x :: y`

Aside: Polymorphism

The polymorphism in ML that arises from type parameters is an example of **generic programming**. (map, filter, etc.) Are good examples of generic functions.

Different languages support generic programming in different ways...

- parametric polymorphism allows functions to work **uniformly** over arguments of different types. E.g., `last : 'a list -> 'a`
- ad hoc polymorphism (or overloading) allows an operation to behave in **different** ways when applied to arguments of different types. There is no such polymorphism in OCaml, but most languages allow some overloading (e.g. 2+3 and 2.4 +3.6). Java and C++ allow one to extend the overloading of a symbol (e.g. "dog" + "house"). This form of overloading is a **syntactic** convenience, but little more.
- subtype polymorphism allows operations to be defined for collections of types sharing some common structure
e.g., a `feed` operation might make sense for values of `animal` and all its “refinements”—`cow`, `tiger`, `moose`, etc.

OCaml supports parametric polymorphism in a very general way, and also supports subtyping (Though we shall not get to see this aspect of OCaml, its support for subtyping is what distinguishes it from other dialects of ML.) It does not allow overloading.

Java provides a subtyping as well as moderately powerful overloading, but no parametric polymorphism. (Various Java extensions with parametric polymorphism are under discussion.)

Confusingly, the bare term “polymorphism” is used to refer to parametric polymorphism in the ML community and for subtype polymorphism in the Java community!