

CIS 500  
Software Foundations  
Fall 2006

September 13

## Administrivia

### BCP gone next week

---

- ▶ I will be out of town for all of next week.
- ▶ My office hours will be cancelled for the week.
- ▶ Lectures and recitations will continue as usual. Brian will give the lectures.

## Polymorphism

### Polymorphism

---

We encountered the concept of polymorphism very briefly last time. Let's look at it now in a bit more detail.

```
# let rec last l =  
  match l with  
  | [] -> raise Bad  
  | [x] -> x  
  | _::y -> last y
```

What type should we give to the parameter `l`?

### Polymorphism

---

```
# let rec last l =  
  match l with  
  | [] -> raise Bad  
  | [x] -> x  
  | _::y -> last y
```

It doesn't matter what type of objects are stored in the list: we could make it `int list` or `bool list`, and OCaml would not complain. However, if we chose one of these types, would not be able to apply `last` to the other.

Instead, we can give `l` the type `'a list` (pronounced "alpha"), standing for an arbitrary type. When we use the function, OCaml will figure out what type we need.

## 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

```
int list -> int
string list -> string
int list list -> int list
etc.
```

In other words,

```
last : '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`.”

## A polymorphic rev

```
# 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 rev (l: 'a list) = revaux l [];;
val rev : 'a list -> 'a list = <fun>
```

```
# rev ["cat"; "in"; "the"; "hat"];;
- : string list = ["hat"; "the"; "in"; "cat"]
```

```
# rev [false; true];;
- : bool list = [true; false]
```

## A polymorphic append

```
# 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]
```

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

## Polymorphic repeat

```
# (* A list of n copies of k *)
let rec repeat (k:'a) (n:int) =
  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; 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.

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

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

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

## Digression: 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.

	Weak	Strong
Dynamic		Lisp, Scheme
Static	C, C++	ML, Java*, C#*

\*Strictly speaking, Java and C# 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`

- ▶ `let f x y = x :: []`
- ▶ `let f x = x @ x`
- ▶ `let f x = x :: x`
- ▶ `let f x y z = if x>3 then y else z`
- ▶ `let f x y z = if x>3 then y else [z]`

And one more:

```
let rec f x =  
  if (tl x) = [] then x  
  else f (tl x)
```

# Programming With Functions

## Functions as Data

Functions in OCaml are *first class* — they have the same rights and privileges as values of any other types. E.g., they can be

- ▶ passed as arguments to other functions
- ▶ returned as results from other functions
- ▶ stored in data structures such as tuples and lists
- ▶ etc.

## 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.

```
# 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.

## filter

Another useful higher-order function is `List.filter`. When applied to a list `l` and a boolean function `p`, it builds a list of the elements from `l` for which `p` returns true.

```
# 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]; []]
```

## Defining map

`List.map` comes predefined in the OCaml system, but there is nothing magic about it—we can easily define our own `map` function with the same behavior.

```
let rec map (f: 'a->'b) (l: 'a list) =
  if l = [] then []
  else f (List.hd l) :: map f (List.tl l)
val map : ('a -> 'b) -> 'a list -> 'b 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:

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

## Defining filter

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

```
# 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>
```

## Multi-parameter functions

We have seen two ways of writing functions with multiple parameters:

```
# let foo x y = x + y;;
val foo : int -> int -> int = <fun>

# let bar (x,y) = x + y;;
val bar : int * int -> int = <fun>
```

The first takes its two arguments separately; the second takes a tuple and uses a pattern to extract its first and second components.

The syntax for applying these two forms of function to their arguments differs correspondingly:

```
# foo 2 3;;
- : int = 5

# bar (4,5);;
- : int = 9

# foo (2,3);;
This expression has type int * int
but is here used with type int

# bar 4 5;;
This function is applied to too many arguments
```

## Partial Application

One advantage of the first form of multiple-argument function is that such functions may be *partially applied*.

```
# let foo2 = foo 2;;
val foo2 : int -> int = <fun>

# foo2 3;;
- : int = 5

# foo2 5;;
- : int = 7

# List.map foo2 [3;6;10;100];;
- : int list = [5; 8; 12; 102]
```

## Currying

Obviously, these two forms are closely related — given one, we can easily define the other.

```
# let foo' x y = bar (x,y);;
val foo' : int -> int -> int = <fun>

# let bar' (x,y) = foo x y;;
val bar' : int * int -> int = <fun>
```

## Currying

Indeed, these transformations can themselves be expressed as (higher-order) functions:

```
# let curry f x y = f (x,y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
          = <fun>

# let foo'' = curry bar;;
val foo'' : int -> int -> int = <fun>

# let uncurry f (x,y) = f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
            = <fun>

# let bar'' = uncurry foo;;
val bar'' : int * int -> int = <fun>
```

## A Closer Look

The type `int -> int -> int` can equivalently be written `int -> (int -> int)`.

That is, a function of type `int -> int -> int` is actually a function that, when applied to an integer, yields a *function* that, when applied to an integer, yields an integer.

Similarly, an application like `foo 2 3` is actually shorthand for `(foo 2) 3`.

Formally: `->` is right-associative and application is left-associative.

## Anonymous Functions

It is fairly common in OCaml that we need to define a function and use it just once.

```
# let timesthreepustwo x = x*3 + 2;;
val timesthreepustwo : int -> int = <fun>

# List.map timesthreepustwo [4;3;77;12];;
- : int list = [14; 11; 233; 38]
```

To save making up names for such functions, OCaml offers a mechanism for writing them in-line:

```
# List.map (fun x -> x*3 + 2) [4;3;77;12];;
- : int list = [14; 11; 233; 38]
```

## Anonymous Functions

Anonymous functions may appear, syntactically, in the same places as values of any other types.

For example, the following let-bindings are completely equivalent:

```
# let double x = x*2;;
val double : int -> int = <fun>

# let double' = (fun x -> x*2);;
val double' : int -> int = <fun>

# double 5;;
- : int = 10

# double' 5;;
- : int = 10
```

## Anonymous Functions

We can even write:

```
# (fun x -> x*2) 5;;
- : int = 10
```

Or (slightly more usefully):

```
# (if 5*5 > 20
   then (fun x -> x*2)
   else (fun x -> x+3))
5;;
- : int = 10
```

The conditional yields a function on the basis of some boolean test, and its result is then applied to 5.

## Quick Check

What is the type of l?

```
# let l = [ (fun x -> x + 2);  
           (fun x -> x * 3);  
           (fun x -> if x > 4 then 0 else 1) ];;
```

## Applying a list of functions

```
# let l = [ (fun x -> x + 2);  
           (fun x -> x * 3);  
           (fun x -> if x > 4 then 0 else 1) ];;  
val l : (int -> int) list = [<fun>; <fun>; <fun>]  
  
# let applyto x f = f x;;  
val applyto : 'a -> ('a -> 'b) -> 'b = <fun>  
  
# List.map (applyto 10) l;;  
- : int list = [12; 30; 0]  
  
# List.map (applyto 2) l;;  
- : int list = [4; 6; 1]
```

## Another useful higher-order function: fold

```
# let rec fold f l acc =  
  match l with  
  | [] -> acc  
  | a::l -> f a (fold f l acc);;  
val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

For example:

```
# fold (fun a b -> a + b) [1; 3; 5; 100] 0;;  
- : int = 109
```

In general:

$$f \ [a_1; \dots; a_n] \ b$$

is

$$f \ a_1 \ (f \ a_2 \ (\dots \ (f \ a_n \ b) \ \dots)).$$

## Using fold

Most of the list-processing functions we have seen can be defined compactly in terms of fold:

```
# let listSum l =  
  fold (fun a b -> a + b) l 0;;  
val listSum : int list -> int = <fun>  
  
# let length l =  
  fold (fun a b -> b + 1) l 0;;  
val length : 'a list -> int = <fun>  
  
# let filter p l =  
  fold  
    (fun a b -> if p a then (a::b) else b)  
    l [];;
```

## Using fold

And even:

```
# (* List of numbers from m to n, as before *)  
let rec fromTo m n =  
  if n < m then []  
  else m :: fromTo (m+1) n;;  
val fromTo : int -> int -> int list = <fun>  
  
# let fact n =  
  fold (fun a b -> a * b) (fromTo 1 n) 1;;  
val fact : int -> int = <fun>
```

## Quick Check

What is the type of this function?

```
# let foo l =  
  fold (fun a b -> List.append b [a]) l [];;
```

What does it do?

## Forms of fold

The OCaml List module actually provides two folding functions:

```
List.fold_left
  : ('a -> 'b -> 'a) -> 'a    -> 'b list -> 'a

List.fold_right
  : ('a -> 'b -> 'b) -> 'a list -> 'b    -> 'b
```

The one we're calling fold (here and in the homework assignment) is `List.fold_right`.

`List.fold_left` performs the same basic operation but takes its arguments in a different order.

## The unit type

OCaml provides another built-in type called `unit`, with just one inhabitant, written `()`.

```
# let x = ();;
val x : unit = ()

# let f () = 23 + 34;;
val f : unit -> int = <fun>

# f ();;
- : int = 57
```

Why is this useful?

## Uses of unit

A function from `unit` to `'a` is a *delayed computation* of type `'a`. When we define the function...

```
# let f () = <long and complex calculation>;;
val f : unit -> int = <fun>
```

... the long and complex calculation is just boxed up in a *closure* that we can save for later (by binding it to a variable, e.g.). When we actually need the result, we apply `f` to `()` and the calculation actually happens:

```
# f ();;
- : int = 57
```

## Thunks

A function accepting a `unit` argument is often called a *thunk*.

Thunks are widely used in functional programming.

A typical example...

Suppose we are writing a function where we need to make sure that some "finalization code" gets executed, even if an exception is raised.

```
# let read file =
  let chan = open_in file in
  try
    let nbytes = in_channel_length chan in
    let string = String.create nbytes in
    really_input chan string 0 nbytes;
    close_in chan;
    string
  with exn ->
    (* finalize channel *)
    close_in chan;
    (* re-raise exception *)
    raise exn;;
```

We can avoid duplicating the finalization code by wrapping it in a thunk:

```
# let read file =
  let chan = open_in file in
  let finalize () = close_in chan in
  try
    let nbytes = in_channel_length chan in
    let string = String.create nbytes in
    really_input chan string 0 nbytes;
    finalize();
    string
  with exn ->
    (* finalize channel *)
    finalize();
    (* re-raise exception *)
    raise exn;;
```

(The `try...with...` form is OCaml's syntax for handling exceptions.)

In fact, we can go further...

```
# let unwind_protect body finalize =
  try
    let res = body() in
    finalize();
  res
with exn ->
  finalize();
  raise exn;;

# let read file =
  let chan = open_in file in
  unwind_protect
    (fun () ->
      let nbytes = in_channel_length chan in
      let string = String.create nbytes in
      really_input chan string 0 nbytes;
      string)
    (fun () -> close_in chan);;
```

## A Larger Example: Streams

### Lazy streams

A thunk is a *lazy computation*: it doesn't do any work until it is explicitly asked for its value.

We can even use thunks to represent *infinite* computations, as long as we only ask for their results a little bit at a time.

For example:

```
# type 'a stream =
  Stream of 'a * (unit -> 'a stream);;
```

That is, an 'a stream is a pair of an 'a value and a thunk that, when evaluated, yields another 'a stream.

```
# type 'a stream =
  Stream of 'a * (unit -> 'a stream);;

# let rec upfrom x =
  Stream (x, fun () -> upfrom (x+1));;
val upfrom : int -> int stream = <fun>

# let rec first n (Stream (x,f)) =
  if n=0 then []
  else x :: (first (n-1) (f()));;
val first : int -> 'a stream -> 'a list = <fun>

# let show s = first 15 s;;
val show : 'a stream -> 'a list = <fun>

# show (upfrom 3);;
- : int list = [3; 4; 5; 6; 7; 8; 9; 10; 11;
               12; 13; 14; 15; 16; 17]
```

### Some convenience functions for streams

```
# let stream_cons x f = Stream (x, f);;
val stream_cons :
  'a -> (unit -> 'a stream) -> 'a stream = <fun>

# let stream_hd (Stream (x,f)) = x;;
val stream_hd : 'a stream -> 'a = <fun>

# let stream_tl (Stream (x,f)) = f ();;
val stream_tl : 'a stream -> 'a stream = <fun>

# let rec first n s =
  if n=0 then []
  else (stream_hd s)
      :: (first (n-1) (stream_tl s));;
val first : int -> 'a stream -> 'a list = <fun>
```

### Transforming streams

```
# let rec map_stream f s =
  stream_cons
    (f (stream_hd s))
    (fun () -> map_stream f (stream_tl s));;
val map_stream :
  ('a -> 'b) -> 'a stream -> 'b stream
  = <fun>

# show (map_stream (fun x -> x mod 4) (upfrom 0));;
- : int list =
  [0; 1; 2; 3; 0; 1; 2; 3; 0; 1; 2; 3; 0; 1; 2]
```



## Transforming streams

```
# let indivisible_by y x = (x mod y <> 0);;
val indivisible_by : int -> int -> bool = <fun>

# show (map_stream (indivisible_by 3) (upfrom 0));;
- : bool list =
  [false; true; true; false; true; true; false;
   true; true; false; true; true; false; true;
   true]
```

## Filtering streams

```
# let rec filter_stream p s =
  if p (stream_hd s)
  then Stream(
    stream_hd s,
    fun() -> filter_stream p (stream_tl s) )
  else filter_stream p (stream_tl s);;
val filter_stream :
  ('a -> bool) -> 'a stream -> 'a stream = <fun>

# show (filter_stream (indivisible_by 3)
  (upfrom 0));;
- : int list = [1; 2; 4; 5; 7; 8; 10; 11; 13;
  14; 16; 17; 19; 20; 22]
```

## A stream of prime numbers

```
# let rec sieve_filter s =
  stream_cons
    (stream_hd s)
    (fun () ->
      sieve_filter
        (filter_stream
          (indivisible_by (stream_hd s))
          (stream_tl s)));;
val sieve_filter : int stream -> int stream = <fun>

# let primes = sieve_filter (upfrom 2);;

# show primes;;
- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23;
  29; 31; 37; 41; 43; 47]
```

## A stream of ...?

```
# let divisible_by y x = (x mod y = 0);;

# let rec funny_filter s =
  stream_cons
    (stream_hd s)
    (fun () ->
      funny_filter
        (filter_stream
          (divisible_by (stream_hd s))
          (stream_tl s)));;

# let funny = funny_filter (upfrom 1);;
```

What familiar sequence is funny?

```
# show funny;;
- : int list = [1; 2; 4; 8; 16; 32; 64; 128;
  256; 512; 1024; 2048; 4096;
  8192; 16384; 32768]
```

## A Taste of Continuations

Consider this pair of functions:

```
# let f x = x + 3;;  
# let g y = 22 * (f y);;
```

Note that, after the call (f y) returns, we still have a multiply left to do.

We can rewrite g to make this remaining work more explicit.

```
# let f x = x + 3;;  
# let g y = (fun r -> 22*r) (f y);;
```

The function (fun r -> 22\*r) is the *continuation* of the expression (f y).

In general, a continuation is a function representing “the work left to be done” when some other computation is finished.

Next, we can pass this continuation as an extra parameter to f, delegating to f the responsibility of calling it:

```
# let f x k = k (x + 3);;  
# let g y = f y (fun r -> 22*r);;
```

In general, a continuation is a function representing “the work left to be done” when some other computation is finished.

The function f is said to be written in *continuation-passing style*.

Is this useful...?

## A simple application of continuations

Consider the following function for multiplying lists of integers:

```
# let rec listProd l =  
  match l with  
  [] -> 1  
  | x::rest -> x * (listProd rest);;  
val listProd : int list -> int = <fun>  
  
# listProd [2;5;23;7;1;7];;  
- : int = 11270  
  
# listProd [2;5;23;7;1;0;7];;  
- : int = 0
```

Observe that, if l contains a 0 element, then the result of listProd will always be 0. Can we avoid doing any multiplies (whatsoever!) in this case?

First, let's rewrite listProd to make the continuation of the recursive call explicit:

```
# let rec listProd l =  
  match l with  
  [] -> 1  
  | x::rest -> (fun y -> x * y) (listProd rest);;
```

As before, this listProd...

```
# let rec listProd l =  
  match l with  
  [] -> 1  
  | x::rest -> (fun y -> x * y) (listProd rest);;
```

... can now be transformed by *passing the continuation* as an extra argument to the recursive call, and *delegating responsibility* for invoking the continuation at the appropriate moment:

```
# let listProd l =  
  let rec listProdAux l k =  
    match l with  
    [] ->  
    k 1  
    | x::rest ->  
    listProdAux rest (fun y -> k (x*y))  
  in listProdAux l (fun x -> x);;
```

Finally, we can add a clause to `listProdAux` that handles the case where a 0 is found in the list by immediately returning 0 *without calling the continuation!*

```
# let listProd l =
  let rec listProdAux l k =
    match l with
    | [] -> k 1
    | 0::rest ->
      0
    | x::rest ->
      listProdAux rest (fun y -> k (x*y))
  in listProdAux l (fun x -> x);;
```

## Uses of continuations

- ▶ Functions can be written to take *multiple continuations* — e.g., a search algorithm might take both a success continuation and a failure continuation  
Gives a clean and flexible way to implement *backtracking* control structures
- ▶ Other *advanced control structures* such as exceptions, coroutines, and (non-preemptive) concurrency can be programmed up using continuations.
- ▶ *Compilers* often transform whole programs into continuation-passing style internally, to make flow of control explicit in the code
- ▶ Some languages (Scheme, SML/NJ) provide a *primitive* (call-with-current-continuation) that “reifies” the continuation at any point in the program and turns it into a data value
- ▶ *Many refinements and variations* have been studied.

## Parting Thoughts

## The rest of OCaml

We’ve seen only a small part of the OCaml language. Some other highlights:

- ▶ advanced *module system*
- ▶ imperative features (`ref` cells, arrays, etc.); the “mostly functional” programming style
- ▶ objects and classes

## Closing comments on OCaml

Some *common strong points* of OCaml, Java, C#, etc.

- ▶ strong, static typing (*no core dumps!*)
- ▶ garbage collection (*no manual memory management!!*)

Some *advantages* of OCaml compared to Java, etc.

- ▶ excellent implementation (fast, portable, etc.)
- ▶ powerful module system
- ▶ streamlined support for higher-order programming
- ▶ sophisticated pattern matching (no “visitor patterns”)
- ▶ parametric polymorphism (Java and C# are getting this “soon”)

Some *disadvantages*:

- ▶ smaller developer community
- ▶ smaller collection of libraries
- ▶ object system somewhat clunky