Administrivia

Second homework assignment was due at noon.
Third homework assignment is due one week from today.
Must already be reading TAPL chapter 5.

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 stored in data structures such as tuples and lists.

Functions as results from other functions.

Functions as arguments to other functions.

Etc.

Programming with Functions in OCaml

Fall 2004 - Software Foundations
CIS 500
Multi-parameter functions

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

```ocaml
# let foo xy = 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, and the second takes a tuple and uses a pattern to extract its first and second components.

```
# foo 2 3;;
- : int = 5
# bar (4, 5);;
- : int = 9
```

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
```

Currying

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

Correspondingly:

```
This function is applied to too many arguments.

val bar' : (int * int) -> int = <fun>
```

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

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

Currying is the technique of extracting the first and second components.

```
The first takes its two arguments separately; the second takes a tuple and uses

# let par (x, y) = x + y;;
# let foo x y = x + y;;
```

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

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

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

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

ACloserLook

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

Formally, 
-

AnonymousFunctions

Anonymous functions appear syntactically in the same places as values.

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

Anonymous functions may appear syntactically in the same places as values.

```ml
# let timesthreeplustwo x = x * 3 + 2;
val timesthreeplustwo : int -> int = <fun>
# List.map timesthreeplustwo [4;3;7;12];;
- : int list = [14;11;23;38]
```

```ml
# List.map (fun x -> x * 3 + 2) [4;3;7;12];;
- : int list = [14;11;23;38]
```

Formally: 
-

Evaluation

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

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

```
# double 5;;
val double : int -> int = <fun>
- : int = 10
```

```
# double' 5;;
val double': int -> int = <fun>
- : int = 10
```

Indeed, these transformations can themselves be expressed as higher-order (curried)
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 its argument.

Another useful higher-order function: fold

```
#let rec fold flacc =
  match l with
  | [] -> acc
  | a :: l -> fa (fold flacc)
#
```

```
val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

For example:

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

In general:

```
let rec fold = function
  | [ ] -> acc
  | a :: l -> a (fold l acc)

let fold = fun flacc ->
  let rec fold = function
  | [ ] -> acc
  | a :: l -> a (fold l acc)
  in fold
```

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)];;

let vall : (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: map

```
# 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]
```

Applying a list of functions

```
# 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]
```

Anonymous Functions

```
# let rec fold flacc =
  match l with
  | [] -> acc
  | a :: l -> fa (fold flacc)
#
```

```
val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

For example:

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

In general:

```
let rec fold = function
  | [ ] -> acc
  | a :: l -> a (fold l acc)

let fold = fun flacc ->
  let rec fold = function
  | [ ] -> acc
  | a :: l -> a (fold l acc)
  in fold
```

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)];;

let vall : (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 flacc =
  match l with
  | [] -> acc
  | a :: l -> fa (fold flacc);
```

```
val fold: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

For example:

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

In general:

```
let rec fold = function
  | [ ] -> acc
  | a :: l -> a (fold l acc)

let fold = fun flacc ->
  let rec fold = function
  | [ ] -> acc
  | a :: l -> a (fold l acc)
  in fold
```

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)];;

let vall : (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]
```
Using fold

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

```ocaml
# 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>
```

And even:

```ocaml
# (* 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?

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

What does it do?
A typical example...

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

\[ \text{List.fold_right} \]

A function from \texttt{unit} to \texttt{a} is a delayed computation of type \texttt{unit -> a}.

Uses of \texttt{unit}

A function accepting a \texttt{unit} argument is often called a \texttt{thunk}.

\[ \text{Why is this useful?} \]

\[ \text{When we define the function...} \]

\[ \text{...the long and complex calculation is just boxed up in a closure that we can save for later (by binding it to a variable, e.g.)}. \]

\[ \text{When we actually need the result, we apply \texttt{f} to \texttt{()}} \]

\[ \text{A function from \texttt{unit} to \texttt{a} is a delayed computation of type \texttt{unit -> a}}. \]

\[ \text{Why is this useful?} \]

\[ \text{Thunks are widely used in functional programming.} \]

\[ \text{A function accepting a \texttt{unit} argument is often called a \texttt{thunk}.} \]

\[ \text{Why is this useful?} \]

\[ \text{The unit type OCaml provides another built-in type called \texttt{unit}, which just one inhabitant.} \]

\[ \text{The \texttt{unit} type} \]

\[ \text{Forms of \texttt{fold}} \]
Suppose we are writing a function where we need to make sure the finalization code gets executed, even if an exception is raised.

```ocaml
let readfile =
  let chan = open_infile 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:

```ocaml
let finalize () = close_in chan in
let readfile =
  let chan = open_infile 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();
    raise exn;;
```

In fact, we can go further...

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

let readfile =
  let chan = open_infile 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);;
```

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