

Currying
Higher Order Functions
The option datatype

Section E • • 7 February 2006

R#4

- Things that are not really your fault:
 - Make sure your PDF prints from Linux
 - Keep text within 80 columns
- Things that you SHOULD be doing:
 - Put name, Andrew ID, TA name at top of everything.
 - **OMG MAKE SURE YOUR CODE RUNS OK??!?!?**
 - Sweat the details... or pay.

- The `unit` type:

- `()`;

- `val it = () : unit`

- `val foo:unit = ()`;

- `val foo = () : unit`

- `print`;

- `val it = fn : string -> unit`

- `print "w00t unit\n"`;

- `w00t unit`

- `val it = () : unit`

Currying and Higher Order Functions

- The map function:

```
(* val map' : ('a -> 'b) * 'a list -> 'b list ...  
- map' (fn x => x+1, [1,2,3,4]);
```

- The map function:

```
(* val map' : ('a -> 'b) * 'a list -> 'b list ...  
- map' (fn x => x+1, [1,2,3,4]);
```

- ↪ what, exactly?

- The map function:

```
(* val map' : ('a -> 'b) * 'a list -> 'b list ...  
- map' (fn x => x+1, [1,2,3,4]);
```

- ↪ what, exactly?

```
val it = [2,3,4,5] : int list
```

- The map function:

```
(* val map' : ('a -> 'b) * 'a list -> 'b list ...  
- map' (fn x => x+1, [1,2,3,4]);
```

- ↪ what, exactly?

```
val it = [2,3,4,5] : int list
```

- Implementation? Two lines...

- The map function:

```
(* val map' : ('a -> 'b) * 'a list -> 'b list ...  
- map' (fn x => x+1, [1,2,3,4]);
```

- ↪ what, exactly?

```
val it = [2,3,4,5] : int list
```

- Implementation? Two lines...

```
fun map' (f, []) = []  
  | map' (f, h::l) = (f h) :: map' (f, l)
```

Now we want a curried version... called map

6

- So that we can do this...

$$\text{map } f \ [x_1, \dots, x_n] \hookrightarrow [f \ x_1, \dots, f \ x_n]$$

- as in...

$$\text{map size } ["tom", "mary", "herman"] \hookrightarrow [3, 9, 4]$$

- Implementation?

- So that we can do this...

$$\text{map } f \ [x_1, \dots, x_n] \hookrightarrow [f \ x_1, \dots, f \ x_n]$$

- as in...

$$\text{map size } ["tom", "mary", "herman"] \hookrightarrow [3, 9, 4]$$

- Implementation?

```
fun map f [] = []  
  | map f (h::l) = (f h) :: (map f l)
```


- Nested maps...

```
fun double (x) = 2*x
```

```
map (map double) [[1], [2,3], [4,5,6]];
```

- Nested maps...

```
fun double (x) = 2*x
```

```
map (map double) [[1], [2,3], [4,5,6]];
```

- Composing functions...

```
map (map (implode o rev o explode))
```

```
[[ "hi", "bye" ], [ "foo", "bar" ]];
```


- The \circ operator composes functions. Definition:

- The \circ operator composes functions. Definition:

- `infix \circ ;`

`infix \circ`

- The `o` operator composes functions. Definition:

```
- infix o;
```

```
infix o
```

```
- fun (f o g) = fn x => f (g x);
```

```
val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

- The o operator composes functions. Definition:

```
- infix o;
```

```
infix o
```

```
- fun (f o g) = fn x => f (g x);
```

```
val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

```
- fun (f o g) x = f (g x);
```

```
val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

```
fun map' (f, []) = []  
  | map' (f, h::l) = (f h) :: map' (f, l)
```

VS.

```
fun map f [] = []  
  | map f (h::l) = (f h) :: map' (f l)
```

Let's implement it...

10

```
(* val curry :  
   ('a * 'b -> c) -> ('a -> ('b -> 'c)) *)
```

```
(* val curry :  
   ('a * 'b -> c) -> ('a -> ('b -> 'c)) *)
```

```
fun curry f x y = f(x, y)
```

```
(* val curry :  
   ('a * 'b -> c) -> ('a -> ('b -> 'c)) *)
```

```
fun curry f x y = f(x, y)
```

```
fun map f l = ((curry map') f) l
```

```
(* val curry :  
   ('a * 'b -> c) -> ('a -> ('b -> 'c)) *)
```

```
fun curry f x y = f(x, y)
```

```
fun map f l = ((curry map') f) l
```

```
map' : ('a -> 'b) * 'a list -> 'b list
```

```
map : ('a -> 'b) -> 'a list -> 'b list
```

```
(* val curry :  
   ('a * 'b -> c) -> ('a -> ('b -> 'c)) *)
```

```
fun curry f x y = f(x, y)
```

```
fun map f l = ((curry map') f) l
```

```
map' : ('a -> 'b) * 'a list -> 'b list
```

```
map : ('a -> 'b) -> 'a list -> 'b list
```

```
(* val curry :  
   ('a * 'b -> c) -> ('a -> ('b -> 'c)) *)
```

```
fun curry f x y = f(x, y)
```

```
fun map f l = ((curry map') f) l
```

```
map' : ('a -> 'b) * 'a list -> 'b list
```

```
map : ('a -> 'b) -> 'a list -> 'b list
```

```
(* val curry :  
   ('a * 'b -> c) -> ('a -> ('b -> 'c)) *)
```

```
fun curry f x y = f(x, y)
```

```
fun map f l = ((curry map') f) l
```

```
map' : ('a -> 'b) * 'a list -> 'b list
```

```
map : ('a -> 'b) -> 'a list -> 'b list
```

Or just: val map = curry map'

`foldl f e [x1, ..., xn]`

- Accumulates function `f`... from start value `e`, over the list, from left to right.

$\Rightarrow f(x_n, f(x_{n-1}, f(x_{n-2}, \dots f(x_1, e) \dots)))$

```
(* val foldl :  
   ('a * 'b -> 'b) -> 'b -> 'a list -> 'b      *)
```

The option datatype

datatype 'a option = NONE | SOME of 'a

datatype 'a option = NONE | SOME of 'a

```
datatype 'a option = NONE | SOME of 'a
```

- Handy for "errors," or when an input lacks a meaningful output. E.g.:

```
- Real.fromString;
```

```
val it = fn : string -> real option
```

```
- Real.fromString "6.0e5";
```

```
val it = SOME 600000.0 : real option
```

```
- Real.fromString "Your Mom";
```

```
val it = NONE : real option
```