

Higher Order Functions
What are the types of the following functions?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # compose plus_two plus_two;;
```

```
1 # compose plus_two plus_two;;
```

```
1 # compose plus_two plus_two;;
```

```
1 - : int -> int = <fun>
```

```
1 - : int -> int = <fun>
```

```
1 - : int -> int = <fun>
```

Higher Order Functions
What are the types of the following functions?

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # plus_two;;  
2 - : int -> int = <fun>
```

```
1 # compose plus_two plus_two;;
```

```
1 # compose plus_two plus_two;;
```

```
1 # compose plus_two plus_two;;
```

```
1 - : int -> int = <fun>
```

```
1 - : int -> int = <fun>
```

```
1 - : int -> int = <fun>
```

```
1 # compose plus_two;
```

```
1 # compose plus_two;
```

```
1 # compose plus_two;
```

```
1 - : ('_a -> int) -> '_a -> int = <fun>
```

```
1 - : ('_a -> int) -> '_a -> int = <fun>
```

```
1 - : ('_a -> int) -> '_a -> int = <fun>
```

Higher Order Functions

What do the following functions do?

```
1 # plus_two;;
2 - : int -> int = <fun>
```

```
1 # compose_plus_two_plus_two;;
2 - : int -> int = <fun>
```

Higher Order Functions

What do the following functions do?

```
1 # plus_two;;
2 - : int -> int = <fun>
```

```
1 # compose_plus_two_plus_two;;
2 - : int -> int = <fun>
```

```
1 # compose_plus_two;;
2 - : ('_a -> int) -> '_a -> int = <fun>
```

Thrice

Recall our definition:

```
1 # let thrice f x = f (f (f x));;
2 val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

Thrice

Recall our definition:

```
1 # let thrice f x = f (f (f x));;
2 val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

How would you write thrice with compose?

Thrice

Recall our definition:

```
1 # let thrice f x = f (f (f x));;
2 val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

How would you write thrice with compose?

```
1 # let thrice f = compose f (compose f f);;
2 val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

Is this the only way?

Curried vs Uncurried

```
1 # let add_three x y z = x + y + z;;
2 val add_three : int -> int -> int -> int = <fun>
```

How does this differ from:

```
1 # let add_triple (u,v,w) = u + v + w;;
2 val add_triple : int * int * int -> int = <fun>
```

Curried vs Uncurried

Outline
First Class Values
Higher Order Functions
Method Notations
Sample Problems

Introduction to HOFs
Currying
Partial Application and Lambda Lifting
Currying Again

```
1 # let add_three x y z = x + y + z;;
2 val add_three : int -> int -> int -> int = <fun>
```

How does this differ from:

```
1 # let add_triple (u,v,w) = u + v + w;;
2 val add_triple : int * int * int -> int = <fun>
```

add_three is *curried*, while add_triple is *uncurried*

Partial Application

Outline
First Class Values
Higher Order Functions
Method Notations
Sample Problems

Introduction to HOFs
Currying
Partial Application and Lambda Lifting
Currying Again

```
1 # (+);;
2 - : int -> int -> int -> int = <fun>
3 # (+) 2 3;;
4 - : int = 5
5 # let plus_two = (+) 2;;
6 val plus_two : int -> int = <fun>
7 # plus_two 7;;
8 - : int = 9
```

Partial application with operators is also called *sectioning*.

Lambda Lifting

Outline
First Class Values
Higher Order Functions
Method Notations
Sample Problems

Introduction to HOFs
Currying
Partial Application and Lambda Lifting
Currying Again

Lambda lifting is a transformation to eliminate *free variables* from a function by adding them as *function parameters*

```
1 # let add_two = (+) (print_string "test"; 2);;
2 test
3 val add_two : int -> int = <fun>
4
5 # let add2 = (* lambda lifted *)
6   fun x -> (+) (print_string "test"; 2) x;;
7 val add2 : int -> int = <fun>
```

You must remember the rules for evaluation when you use partial application. Can you see the difference between these two?

Lambda Lifting

Outline
First Class Values
Higher Order Functions
Method Notations
Sample Problems

Introduction to HOFs
Currying
Partial Application and Lambda Lifting
Currying Again

Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied

```
1 # thrice add_two 5;;
2 - : int = 11
3 # thrice add2 5;;
4 test
5 test
6 test
7 - : int = 11
```

Curry and Uncurry

Outline
First Class Values
Higher Order Functions
Method Notations
Sample Problems

Introduction to HOFs
Currying
Partial Application and Lambda Lifting
Currying Again

```
1 # (+);;
2 - : int -> int -> int = <fun>
3
4 # let curry f x y = f (x,y);;
5 val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
6
7 # let uncurry f (x,y) = f x y;;
8 val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Remember these!

Curry and Uncurry

Outline
First Class Values
Higher Order Functions
Method Notations
Sample Problems

Introduction to HOFs
Currying
Partial Application and Lambda Lifting
Currying Again

Curry and Uncurry

```
1 # let plus = uncurry (+);;
2 val plus : int * int -> int = <fun>
3
4 # plus (3,4);;
5 - : int = 7
6
7 # curry plus 3 4;;
8 - : int = 7
```

Related Function: Zip

```

1 # let rec zip list1 list2 =
2   match (list1,list2) with ([ ], _) -> []
3 | (_, [ ]) -> []
4 | (x::xs, y::ys) -> (x,y)::zip xs ys;;
5 val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
6 # zip [1;2;3] [4;5;6];;
7 - : (int * int) list = [(1, 4); (2, 5); (3, 6)]

```

Zipwith

```

1 # let rec zipwith f list1 list2 =
2   match (list1,list2) with ([ ], _) -> []
3 | (_, [ ]) -> []
4 | (x::xs, y::ys) -> f x y ::zipwith f xs ys;;
5 val zipwith : ('a -> 'b -> 'c) ->
6   'a list -> 'b list -> 'c list = <fun>
7 # zipwith (+) [1;2;3] [4;5;6];;
8 - : int list = [5; 7; 9]
9 # zipwith (fun x y -> (x,y)) [1;2;3] [4;5;6];;
10 - : (int * int) list = [(1, 4); (2, 5); (3, 6)]

```

Zip from Zipwith

How do you write zip from zipwith with no explicit recursion?

Zip from Zipwith

How do you write zip from zipwith with no explicit recursion?

```

1 # let zip = zipwith (fun x y -> (x,y));;
2 val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
3 # zip [1;2;3] [4;5;6];;
4 - : (int * int) list = [(1, 4); (2, 5); (3, 6)]

```

Sample problems

- ▶ Write a function flipuc that flips the arguments to an uncurried function, using just curry, flip and uncurry
- ▶ Write a function that has type ('a -> 'b) -> 'a * 'c -> 'b
- ▶ Use fold_right to write a function that takes a list and returns it.
- ▶ Use fold_right to write a function to remove all negative elements from a list

Problem 1

Write a function flipuc that flips the arguments to an uncurried function, using just curry, flip and uncurry

```

1 # let flipuc f = uncurry (flip (curry f));;
2 val flipuc : ('a * 'b -> 'c) -> 'b * 'a -> 'c = <fun>;
3 # let cons (x,xs) = x::xs;;
4 val cons : 'a * 'a list -> 'a list = <fun>
5 # let snoc = flipuc cons;;
6 val snoc : 'a list * 'a -> 'a list = <fun>
7 # snoc(snoc ([1],2),3);;
8 - : int list = [3; 2; 1]

```

Problem 2

Write a function that has type `('a -> 'b) -> 'a * 'c -> 'b`

```
1 # let app_fst f (a,b) = f a;;
2 val app_fst : ('a -> 'b) -> 'a * 'c -> 'b = <fun>
3 # app_fst ((+) 1) (3, 7);;
4 - : int = 4
5 # app_fst ((+) 1) (4, "hi");;
6 - : int = 5
```

Problem 3

Use `fold_right` to write a function that takes a list and returns it.

```
1 # let listId list =
2   fold_right (fun x xs -> x::xs) list [];;
3 val listId : 'a list -> 'a list = <fun>
4 # listId [1;2;3];;
5 - : int list = [1; 2; 3]
```

Problem 4

Use `fold_right` to write a function to remove all negative elements from a list

```
1 # let gezero list =
2   fold_right
3   (fun x xs -> if x >= 0 then x::xs else xs)
4   list [];;
5 val gezero : int list -> int list = <fun>
6 # gezero [1;0;3;-5;7;-2];;
7 - : int list = [1; 0; 3; 7]
```