Functions are Important

Function values

Example

```
> let add x y = x + y;;
val add : x:int -> y:int -> int
> let addxy = add;;
val addxy : (int -> int -> int)
> addxy 4 3;;
val it : int = 7
> let add4y = add 4;;
val add4y : (int -> int)
> add4y 3;;
val it : int 7
```

- **let** notation for defining new named functions
- New names for old functions behaves just like assigning other values

Evaluate parameters

Example

```
> let printResult x = printfn "%O" x;;
val printResult : x:'a -> unit
> printResult (4*3);;
12
> printResult (let x = 3 in x);;
3
> printResult (
    let car (L : 'a list) = L.Head
    in
    car [1;2;3]);;
1
```

- If a function is called, the parameter is evaluated before the function is called

Eager Evaluation

Example

```
> let f x = printfn "%d" (1 / x);;
> f 0;;
System.DivideByZeroException: Attempted to
  divide by zero.
```

- **Eager** evaluates before call, whether used or not.
- Error because `1/0` always evaluated.
- F# uses eager evaluation by default.

Lazy Evaluation

Example

```
> let f eval x =
    let y = lazy (1 / x)
    if eval then
      printfn "%d" y.Value;;
> f true 0;;
System.DivideByZeroException: Attempted to divide
  by zero.
> f false 0;;
```

- **Lazy** evaluates only when needed.
- No error in last call because `1/0` not needed for an executed branch.

Function Values

Example (C# Declaration)

```
delegate int func(int x);
```

- Functions in F# do *not* have names
- Variables are *bound* to function values, as with other kinds of values
- The **let** syntax does two separate things:
  - Creates a new function value
  - Binds a function name to that value
- Names are syntactic sugar to define functions more easily

Function Names

Example (Named function)

```
> let f (x:int) = x + 2;;
val f : x:int -> int
> f 1;;
val it : int = 3
```

Example (Anonymous function)

```
> let f = (fun x -> x + 2);;
val f : x:int -> int
> f 1;;
val it : int = 3
> (fun x -> x + 2) 1;;
val it : int = 3
```

Anonymous Functions – Foldback

Example

```
let rec foldback f L a =
  match L with
    | [] -> a
    | h::t -> f h (foldback f t a);;
```

Example (Named function)

```
> let add a b = a + b;;
> foldback add [1;2;3;4] 0;; //  returns 10
```

Example (Anonymous function)

```
> foldback (fun a b -> a+b) [1;2;3;4] 0;;
val it : int = 10
> foldback (fun a b -> a*b) [1;2;3;4] 1;;
val it : int = 24
```

Anonymous Functions – Order

Example (let)

```
> let intBefore a b = a < b;;
> quicksort [1;4;3;2;5] intBefore;;
val it : int list = [1;2;3;4;5]
```

Example (anonymous)

```
> quicksort [1;4;3;2;5] (fun a b -> a < b)
val it : int list = [1;2;3;4;5]
```

Partially Applied Functions

Example

```
> let add x y = x + y;;
> add 3 2;;
val it : int = 5

> let z = add 3;;
val z : (int -> int)

> z 2;;
val it : int = 5
```

- Functions can return functions as a result.
- `add 3` returns the partially applied function with `x=3`
- The function returned can then be called: `z 2`

Fixing Values

Example

```
> let g a b = a + b;;
val g : a:int -> b:int -> int
> let h = g 2;;
val h : (int -> int)
> h 4;;
val it : int = 6
> let h = (fun b -> 2 + b);;
val h : b:int -> int
```

- Partially applied functions fix one or more of multiple arguments
- `g 2` returns a *partial* function with one unbound parameter

Exercise

Results?

```
let f1 x y = if x>y then x else y;;
f1 3 8;;

let z = f1 3;;
z 8;;

let m = f1;;
m 3 8;;
```

Returning Functions

Example

```
> let inc x = x + 1;;
> let dubal x = x * 2;;

> let pick a =
    match a with
      | 1 -> inc
      | 2 -> dubal;;
val pick : a:int -> (int -> int)

> let p = pick 1;;

> p 4;;
val it : int = 5
```

Returning Functions

Example

```
> let pick a =
    match a with
      | 1 -> (fun x -> x + 1)
      | 2 -> (fun x -> x*2);;
val pick : a:int -> (int -> int)
> let z = pick 2;;
val z : (int -> int)
> pick 2 4;;
val it : int = 8
> z 4;;
val it : int = 8
```

- `pick` returns one of two functions
- Both functions have one `int` parameter and return `int`.
- The function returned can then be called or assigned

Exercises

Results?

```
let f1 x = if x>0 then (fun y -> y-1)
           else (fun y -> y+1);;
f1 (3);;
f1 (3) (8);;
f1;;

let f2 x = (fun y -> x + y);;
f2 (3);;
f2 (3) (4);;
f2;;
```

- A function can be executed by another function
- Functions can be passed as parameters

Higher Order Functions

Function values

- Every function has an order:
  - A function that has no functions as a parameter, and does not return a function value, has order 1
  - A function with a function as a parameter or returns a function value has order $n + 1$, where n is the order of its highest-order parameter or returned value

Currying

- *Currying* transforms a function of multiple arguments into chain of functions each taking one argument

Example

```
> let g a b c = a::b::c::[];;
val g : a:'a -> b:'a -> c:'a -> 'a list

> g;;
val it : ('a -> 'a -> 'a -> 'a list)
> g 2;;
val it : (int -> int -> int list)
> g 2 3;;
val it : (int -> int list)
> g 2 3 4;;
val it : int list = [2; 3; 4]
```

Partially Applied vs. Curried

- Partially applied functions fix parameter(s) in *any* position(s)
- Curried functions fix the parameter of the *first* function

Example (Partially Applied)

```
> let g1 a b c = a::b::c::[];;
val g1 : a:'a -> b:'a -> c:'a -> 'a list
> let h1 a c = g1 a 2 c;;
val h1 : a:int -> c:int -> int list
```

Example (Curried)

```
> let g2 a b c = a::b::c::[];;
val g2 : a:'a -> b:'a -> c:'a -> 'a list
> let h2 = g2 2;;
val h2 : (int -> int -> int list)
> let h3 = g2 2 3;;
val h3 : (int -> int list)
```

Currying Advantage

Example

```
> map (fun x -> x+1) [1;2;3];; //return [2;3;4]
> let sum a b = a+b;;
> map (sum 1) [1;2;3];; // return [2;3;4]

> map (fun x->[1;x]) [1;2;3];;
val it : int list list = [[1; 1]; [1; 2]; [1; 3]]

> let list a b = [a;b];;
> let h = list 1;;  // curried
> map h [1;2;3];; // returns [[1;1]; [1;2]; [1;3]]
```

- Convert binary function to unary with fixed first parameter.
- Useful with `map`, `foldback`, `reduce`.

Another Advantage

Example

```
> let srtorder order x y =
    match order with
      | "ascend" -> x <= y
      | "descend" -> x >= y;;
> srtorder "ascend" 3 4;;
val it : bool = true

> let ordup : int->int->bool = srtorder "ascend";;
val ordup : (int -> int -> bool)
> ordup 3 4;;
val it : bool = true
```

- Compute function at runtime
- Signatures of computed functions must be identical

Another Advantage

Example

```
> let sortorder order x y =
    match order with
      | "ascend" -> x <= y
      | "descend" -> x >= y;;

> let ge:int->int->bool = sortorder "ascend";;

> let L = map (ge 3) [1;2;3;4];;
val L : bool list = [false; false; true; true]
```

- Convert *n-ary* function to lower order with fixed parameters with `map`, `foldback`

Currying Example

Example

```
> let f a b c = a + b * c;;
val f : a:int -> b:int -> c:int -> int
> let g a = fun b -> fun c -> a + b * c;;
val g : a:int -> b:int -> c:int -> int
> let h = f;;
val h : (int -> int -> int -> int)

> f 2 3 4;;
val it : int = 14
> g 2 3 4;;
val it : int = 14
> h 2 3 4;;
val it : int = 14
```

Exercises

Given

```
let f a = fun b -> a / b;;
let g a b = a / b;;
let h = g;;
```

Results?

```
f;;
g;;
f 13;;
h 13;;
f 13 4;;
map (g 12) [2;3;4];;
let x = map (g 12);;
x [2;3;4];;
```

Computing Anonymous Functions

Example

```
> let inc x = x+1;;
> let mapINC = map inc;;  // f bound to inc
val mapINC : (int list -> int list)
> mapINC [1;2;3];; // returns [2; 3; 4]
```

- `map` has two parameters, a unary function f and list
- Returns curried function where `map` function has parameter `f` bound to `inc` and one unbound list parameter.
- `mapINC` is computed to be an unary function that increments every element of an integer list.

Computing Functions

Example

```
> let sub x y = x - y;;
> let bu f a b = f a b;;
> bu sub 1 3;;
val it : int = -2
> let SUB1 = bu sub 1;;
val SUB1 : (int -> int)
> map SUB1 [1;2;3];;    // return [0; -1; -2]
> map (bu sub 1) [1;2;3];;  // return [0; -1; -2]
```

- bu has two parameters, a binary function `f` and `f`'s first parameter
- Returns unary function where `f` and first parameter are bound and second parameter is unbound.
- bu stands for *binary* to *unary*, since it takes a binary function and returns a unary function.

Automating Computing Functions

Example

```
> let bu f a b = f a b;;
> let rev f a b = f b a;;
> sub 1 3;;
val it : int = -2
> rev sub 1 3;;
val it : int = 2
> bu (rev sub) 1 3;;
val it : int = 2
> map (bu (rev sub) 1) [0;1;2];;
val it : int list = [-1; 0; 1]
```

- `rev` has one parameter, a binary function `f`
- Returns binary function where `f` is bound and the first and second parameters are reversed.
- `rev` stands for reverse the binary parameters.

Staging Functions

Example

```
> let foldback f L a =
    let rec fldb L =
      match L with
        | [] -> a
        | h::t -> f h (fldb t)
    in
    fldb L;;
> let r x = foldback sub x 0;;
> r [1;2;3];;  // returns 2
```

- Old `foldback` has parameters `f` and `a` fixed, but passed as parameters for each recursive call
- *Staged* `foldback` returns anonymous foldback function of one parameter
- Returns `fldb` function of one unbound parameter.