Functions as Parameters

Functions as Parameters

Example (C# function passing)

```
delegate int func(int x);
int comp(func f, int z) {  return f(z); }
int sq(int x) { return x*x; }

static void Main(){ comp(sq, 4); } // return 16
```

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

F# Functions

Example

```
> let comp f z = f z;;
val comp : f:('a -> 'b) -> z:'a -> 'b

> comp sq 4;;
val it : int = 16
```

- Functions can be passed as parameters
- Parameter function executed by another function
- Supports abstraction such as repeated function execution

Mapping sq Function

Example

```
> let sq x = x*x;;
> let rec map_sq L =
    match L with
      | [] -> []
      | h::t -> sq(h)::map_sq t;;

> map_sq [1;2;3;4];;
val it : int list = [1; 4; 9; 16]
```

- Repeated mapping of unary `sq` function to every element of one list: `sq(1)::sq(2)::sq(3)::sq(4)::[]`

Parameter Types

Example

```
> let rec map (f: int -> int) (L: int list) =
    match L with
      | [] -> []
      | h::t -> (f h)::map f t;;

> map sq [1;2;3;4];;
val it : int list = [1; 4; 9; 16]
```

- map parameter `f:int->int` is function with `int` parameter and returns an `int`
- There are no `int`-specific operations in map, but map is not generic, only allows `int->int`

Generic Parameters

Example

```
> let sq (x:int) : int = x * x;;

> let rec map (f: 'a -> 'a) (L : 'a list) =
    match L with
      | [] -> []
      | h::t -> f h::map f t;;

> map sq [1;2;3;4];;
val it : int list = [1; 4; 9; 16]
```

- Use `'a` when parameter can be any type
- `map` operations are generic
- But `sq` uses `int`-specific operation *
- Pass type parameter, creating generic `map`

Mapping *foo*

Example

```
> let foo (x: string) : string = x + "#";;

> let rec map (f: 'a -> 'a) (L : 'a list) =
    match L with
      | [] -> []
      | h::t -> f h::map f t;;

> map foo ["a";"b";"c"];;
val it : string list = ["a#"; "b#"; "c#"]
```

- A *unary* function has a single input parameter
- map of unary `foo` function to every list element

Mismatched parameter/result types

Example

```
> let ItoS (x: int) : string = x.ToString();;
> let rec map (f: 'a -> 'a) (L : 'a list) =
    match L with
      | [] -> []
      | h::t -> f h::map f t;;

> map ItoS [1;2;3];;
error FS0001: Type mismatch. Expecting a int -> int
  but given a int -> string
  The type 'int' does not match the type 'string'
```

- `map` expects `f:'a->'a` – parameter/result must be same type

Fixing mismatched types

Example

```
> let ItoS (x: int) : string = x.ToString();;
> let rec map<'T1, 'T2> (f:'T1->'T2) (L:'T1 list) =
    match L with
      | [] -> []
      | h::t -> f h::map f t;;

> map ItoS [1;2;3];;
val it : string list = ["1"; "2"; "3"]
```

- First way of solving mismatched types
- Note the use of <. . . > after the function name

Fixing mismatched types

Example

```
> let ItoS (x: int) : string = x.ToString();;
> let rec map (f : 'a->'b) (L : 'a list) =
    match L with
      | [] -> []
      | h::t -> f h::map f t;;

> map ItoS [1;2;3];;
val it : string list = ["1"; "2"; "3"]
```

- Second way of solving mismatched types

Exercise

Given

```
let f a = [a];;
let g a = a / 3.0;;
```

Results?

```
map f [1; 2; 3];;
map f [1.0; 2.0; 3.0];;

map g [1; 2; 3];;
map g [1.0; 2.0; 3.0];;
```

Class vs. Type

From **Design Patterns: Elements of Reusable Object-Oriented Software**

> *An object's class defines how the object is implemented. The class defines the object's internal state and the implementation of its operations.*
> *In contrast, an object's type only refers to its interface—the set of requests to which it can respond.*
> *An object can have many types, and objects of different classes can have the same type.*

- Consider a *type* Stack with operations: push, pop, isEmpty
- One *class* for String, another *class* for Int

Binary Mapping

Example

```
> let add x y = x + y;;

> let rec map2 f L1 L2 =
    match (L1,L2) with
      | ([], []) -> []
      | (h1::t1, h2::t2) -> f h1 h2::map2 f t1 t2;;

> map2 add [1;2;3] [4;5;6];;
val it : int list = [5; 7; 9]
```

- Repeated mapping of *binary* add function to every element of two lists with equal number elements

Filtering maps with a guard

Example

```
> let odd x = x%2=1;;

> let rec filter f L =
    match L with
      | [] -> []
      | h::t when f h -> h::filter f t
      | h::t -> filter f t;;

> filter odd [1;2;3;4;5];;
val it : int list = [1; 3; 5]
```

- Filter a list elements by repeated mapping of odd predicate function to every element of one list

FoldBack

Example

```
> let mult a b = a * b;;
> let rec foldb_mult L a =
    match L with
      | [] -> a
      | h::t -> mult h (foldb_mult t a);;
> foldb_mult [2;3;4;5] 1;;
val it : int = 120
```

- Reduction of a list by repeated mapping of binary function from elements of one list to a single combined value
- Equivalent to:
  `(mult 2 (mult 3 (mult 4 (mult 5 1))))`

FoldBack

Example

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

> foldback mult [2;3;4;5] 1;;
val it : int = 120
```

- In this example, function parameters and a and result all have same type.

More Generic FoldBack

Example

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

> foldback cons [1;2;3] [];;
val it : int list = [1; 2; 3]
```

- Requires *second* function parameter and reduction value a and result to have same type.

Closures

Function and its environment – a table storing a reference to each of the non-local variables of that function.
A closure – unlike a plain function pointer – allows a function to access those non-local variables even when invoked outside of its immediate lexical scope.

Local functions

Example

```
let prime n =
  let rec remainder m =
    match m with
      | 0 | 1 -> true
      | m ->
        if n%m=0 then false
        else remainder (m-1)
  in remainder (n-1);;
```

- Observe that `n` parameter remains fixed in `remainder`
- Compute remainder with one parameter

Staged computation

Example

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

> foldback add [4;5;6] 0;;
val it : int = 15
```

- Improves `foldback` by eliminating fixed parameters `f` and `a`
- `red L` returns a reference to the function bound with the enclosing environment of `f`, `L`, and `a`.
- More later when we discuss currying.