VIA University College

Programming Concepts and Languages

Spring 2024

HOF & Lambda

- What do you think is the output of the following?

```
List.map (fun x -> x + 1) [5 .. 5 .. 25];;
```

- What about this?

```
List.map (fun x -> -x) [1 .. 10];;
```

Learning Objectives

- By the end of this session, you will be able to:
  - ✓ explain and implement simple F# programs using
    - ✓ higher-order functions
    - ✓ lambda functions
    - ✓ partial function application and currying
    - ✓ closures

Higher-Order Functions I

- *First-order functions* only take and return non-functions.
  - E.g., types like `int -> int` and `int*int -> int list`
  - Many languages only have first-order functions.
- A *second-order function* can take and return first-order functions.
  - E.g., types like `(int -> int) -> int` and `int -> int*(int -> int)`
- *Higher-order functions* can take and return functions of any order.
  - Functions can also be stored in data structures, etc.
- Higher-order functions are a very powerful feature.
  - They can “glue” functions together to form more complex ones.
- With higher-order functions, functions become just like other types.
  - Functions don’t even need to have names (as for pairs and lists)

Higher-Order Functions II

- F# is a higher-order language
- i.e. functions:
  - are data just as data of any other “ordinary” type can be stored in data structures, passed as arguments, and returned as function values
  - As arguments provides a way to parameterize function definitions, where common computational structure is “factored out”
  - that take functions as arguments are called higher-order functions
- Common computational patterns can be captured as higher-order functions

Higher-Order Functions

- Higher-order functions are functions that take functions as arguments or return functions.
  - It includes functions within data structures.
- Example: full definition of append is as follows:

```
let rec append =
   fun ls1 ->
        (fun ls2 ->
            match ls1 with
               | [] -> ls2
               | hls1::tls1 -> hls1 :: append tls1 ls2
         )
```

- fun constructs a function
  - (function does the same but with many patterns.)
- append is actually a function that returns a function!

Some Higher-Order Functions over Lists

- map: apply a function to all elements in a list
- filter: remove all elements not satisyfing a given condition
- fold (including different versions): combine all elements using a function with two arguments (like binary operators)
- They capture common computation patterns that can be reused.
- also availbale for for other datatypes, such as arrays, sequences, etc.

Some Higher-Order Functions over Lists: map

- The following is an example of a function that takes a function as an argument (Ex.2.2.3b ***pclMap***).

```
let rec pclMap f = function
    | [] -> []
    | hd::tl -> f hd :: pclMap f tl
```

- This function applies another function to every element in a list.

- Example

```
pclMap List.rev [[1;2;3]; [4;5;6]] gives ?

int list list = [[3; 2; 1]; [6; 5; 4]]
```

Some Higher-Order Functions over Lists: map II

- Example: a function that adds one to each element in a list of integers(recall Exercise 2.2.3a ***pmIncList***):

```
// pmIncList : int list -> int list
```

```
let rec pmIncList lst =
    match lst with
    | [] -> []
    | hd::tl -> hd + 1 :: pmIncList tl
```

- define ***incList*** through map:

```
let incList lst = let inc n = n + 1 in pclMap inc lst
```

- Negate a list through map:

```
let negate x = -x
pclMap negate [1 .. 10]
```

Some Higher-Order Functions over Lists: filter

- removes all elements from a list that do not satisfy a given predicate (recall Ex.2.2.4 pclFilter):

```
// filter : ('a -> bool) -> 'a list -> 'a list
```

```
let rec pclFilter predicate lst =
    match lst with
    | [] -> []
    | x::xs -> if predicate x then x :: pclFilter predicate xs
               else pclFilter predicate xs
```

- Example: if pclEven returns true for even numbers, then:

```
pclFilter pclEven [0;1;2;3;4;5] results in [0;2;4]
```

a closer look at fold vs foldBack

- compare `List.foldBack(+)[1;2;3]0` and `List.fold(+) 0[1;2;3]`

```
List.foldBack (+) [1;2;3] 0 gives 1 + List.foldBack (+) [2;3] 0
                            gives 1 + (2 + List.foldBack (+) [3] 0)
                            gives 1 + (2 + (3 + List.foldBack (+) [] 0))
                            gives 1 + (2 + (3 + 0))
                            gives 6
```

```
List.fold (+) 0 [1;2;3] gives List.fold (+) (0 + 1) [2;3]
                        gives List.fold (+) ((0 + 1) + 2) [3]
                        gives List.fold (+) (((0 + 1) + 2) + 3) []
                        gives (((0 + 1) + 2) + 3)
                        gives 6
```

- .

List.fold vs List.foldBack Efficiency

- For operators on atomic types, such as + (int, float, etc.), and && (bool), List.fold is more efficient than List.foldBack
- Reason: since F# is call-by-value, the accumulating argument of List.fold will be evaluated for each new call
- Also, List.fold is tail recursive
- sum, product can better be defined with fold:
- .

```
let sum xs = List.fold (+) 0 xs
let product xs = List.fold (*) 1 xs
```

Lambda functions
Anonymous functions

- Anonymous functions are supported in:
  - JavaScript
  - PHP 4.0.1 – PHP 5.2.x
  - PHP 5.3
  - C#
  - Java
  - Scala
  - Python
  - Etc.

λ – functions

- λ–expressions from λ–calculus (by Alonzo Church, 1930s)
  - Pure λ calculus has neither variables nor loops/recursion
- example λ-expression

```
let f := λp.puv
```

  - anonymous (unnamed) λ-expression of one parameter, `p`
- Example invocation:

```
(f x) = ((λp.puv) x) = xuv
```

- pure λ-expressions have only names for the parameters

Anonymous functions via Lambda-abstraction

- Functions can be nameless
- **fun x -> e** stands for function with formal argument **x** and function body **e**
- Examples:

  `fun x -> x + 1`, a function that increments-by-one

  `List.map (fun x -> x + 1) xs` returns list with all elements incremented by one
- Anonymous functions are often convenient to use with higher-order functions, no need to declare functions that are used only once

Worth knowing

fun x y -> e shorthand for **fun x -> (fun y -> e)**

- Pattern matching as in ordinary definitions:
  - Example: **fun (x,y) -> x + y**
- Currying can be defined through abstraction:

```
add x y = x + y
add2000  =  add 2000
```

- Also note:

let (rec) f x = ... is precisely the same as **let (rec) f = fun x -> (...)**

Examples

- Recall negate with map:

```
let negate x = -x
List.map negate [1 .. 10]
```

- Re-written through lambda:

```
List.map (fun x -> -x) [1 .. 10]
```

- lambda as a second argument:

```
doubleNum x (f:int -> int) = f(f(x))
myVal = doubleNum 4 (fun x -> x * 2) => 16
```

- Also :

```
let f n =
    let doubleNum = (fun x -> x * 2) n
    let tripleNum = (fun x -> x * 3) n
    doubleNum + tripleNum
```

What is the value of `f 5;;` ? gives **25**

More Examples – Map Positive Integers

```
positiveIntegers xs =
    let isPos x = x > 0
    in List.map isPos xs
```

- re-define this using lambda

```
positiveIntegers xs = List.map (fun x -> x > 0) xs
```

- can be done also through “curry-cancelling”

```
positiveIntegers = List.map (fun x -> x > 0)
```

- What do you think?

Currying and Partial Functions

- Functions can have many arguments:
  - f: A -> B -> C -> D

  `let f x y = expr` can be rewritten with lambda:

  `let f = fun x -> fun y -> expr`

- **f** is higher-order function since it returns a function.
- When **f** is applied to one argument, the result is a function that can be applied to another argument.
- This is called a *Curried function*, courtesy: logician Haskell Curry.
  - i.e.: encoding functions with several arguments
  - The ability to transform a function taking n arguments into a chain of n functions, each taking one argument:
- Example: `List.iter (printfn "%d") [1 .. 3]`

Currying II

- We can pass the result of applying one argument to another function
- Example: If we want to add “Software Engr ” in front of each name in a list:

```
map ((+) "Software Engr ") ["Anders"; "Catalin";
"Emil";"Mihai"; "Sachin"]
```

⇩

```
 ["Software Engr Anders"; "Software Engr Catalin";
"Software Engr Emil"; "Software Engr Mihai";
"Software Engr Sachin"]
```

- [Here (+) makes + into a prefix function with type string -> string -> string.]

Closures

- Functions that have some pre-bound arguments (i.e. predefined or “closed” arguments)
- Allows us to make complicated functions from simpler ones.
- We can create functions at any point in an expression
- Example: Multiplying every element in the list by a given number:

```
let listMult i lst = List.map (fun x -> x * i) lst
```

`listMult 10 [1; 3; 5; 7]` ⇨ `[10; 30; 50; 70]`

- Note: **i** is a parameter to **listMult** not to the lambda
- How can **i** be then used inside the lambda?
- **i** is in scope, it is captured by the ***closure*** of the lambda, and therefore accessible.

Closures – another example

- Partially applying a function inside a function

```
closureFun =
   let multi x y = x * y
   let triple = multi 3 // partial application of multiplication
   // triple is a closure that takes one arg
   printfn "%d" (triple 5)
```

- What is the output? `triple 5` gives `15`

Exercises – Higher-order functions

Exercises 2.3.1 – 2.3.3