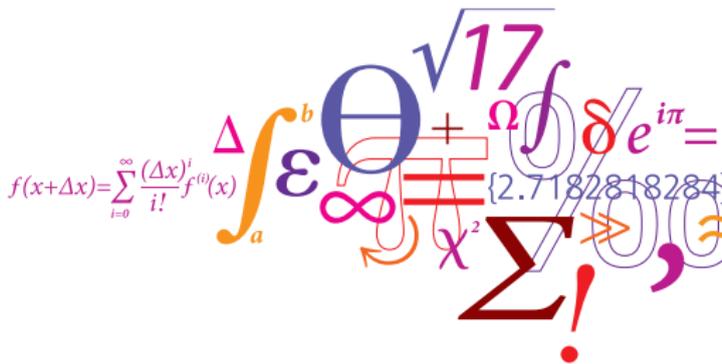


# 02157 Functional Programming

## Lecture 2: Functions, Basic Types and Tuples

Michael R. Hansen



DTU Informatics

Department of Informatics and Mathematical Modelling

- A further look at functions, including higher-order (or curried) functions
- A further look at basic types, including characters, equality and ordering
- A first look at polymorphism
- A further look at tuples and patterns
- A further look at lists and list recursion

Goal: By the end of the day you are acquainted with a major part of the F# language.

Function expressions with general patterns, e.g.

```
function
  | 2          -> 28 // February
  | 4|6|9|11  -> 30 // April, June, September, November
  | _         -> 31 // All other months
;;
```

Simple function expressions, e.g.

```
fun r -> System.Math.PI * r * r ;;
val it : float -> float = <fun:clo@10-1>

it 2.0 ;;
val it : float = 12.56637061
```

# Anonymous functions

Simple functions expressions with *currying*

$$\text{fun } x \ y \ \dots \ z \ \rightarrow \ e$$

with the same meaning as

$$\text{fun } x \ \rightarrow (\text{fun } y \ \rightarrow (\dots (\text{fun } z \ \rightarrow e) \dots))$$

For example: The function below takes an integer as argument and returns a function of type `int -> int` as value:

```
fun x y -> x + x*y;;
val it : int -> int -> int = <fun:clo@2-1>

let f = it 2;;
val f : (int -> int)

f 3;;
val it : int = 8
```

Functions are **first class citizens**:  
the argument and the value of a function may be functions

## Function declarations

A simple function declaration:

$$\text{let } f\ x = e \quad \text{means} \quad \text{let } f = \text{fun } x \rightarrow e$$

For example: `let circleArea r = System.Math.PI *r*r`

A declaration of a **curried function**

$$\text{let } f\ x\ y \dots z = e$$

has the same meaning as:

$$\text{let } f = \text{fun } x \rightarrow (\text{fun } y \rightarrow (\dots (\text{fun } z \rightarrow e)\dots))$$

For example:

```
let addMult x y = x + x*y;;
val addMult : int -> int -> int
```

```
let f = addMult 2;;
val f : (int -> int)
```

```
f 3;;
val it : int = 8
```

## An example

Suppose that we have a cube with side length  $s$ , containing a liquid with density  $\rho$ . The weight of the liquid is then given by  $\rho \cdot s^3$ :

```
let weight ro s = ro * s ** 3.0;;  
val weight : float -> float -> float
```

We can make *partial evaluations* to define functions for computing the weight of a cube of either water or methanol:

```
let waterWeight = weight 1000.0;;  
val waterWeight : (float -> float)
```

```
waterWeight 2.0;;  
val it : float = 8000.0
```

```
let methanolWeight = weight 786.5 ;;  
val methanolWeight : (float -> float)
```

```
methanolWeight 2.0;;  
val it : float = 6292.0
```

We have in previous examples exploited the pattern matching in function expression:

```
function
|  $pat_1$  →  $e_1$ 
  ⋮
|  $pat_n$  →  $e_n$ 
```

A [match expression](#) has a similar pattern matching feature:

```
match  $e$  with
|  $pat_1$  →  $e_1$ 
  ⋮
|  $pat_n$  →  $e_n$ 
```

The value of  $e$  is computed and the expression  $e_i$  corresponding to the first matching pattern is chosen for further evaluation.

# Example

Alternative declarations of the power function:

```
let rec power = function
  | (_,0) -> 1.0
  | (x,n) -> x * power(x,n-1);;
```

are

```
let rec power a = match a with
  | (_,0) -> 1.0
  | (x,n) -> x * power(x,n-1);;
```

and

```
let rec power(x,n) = match n with
  | 0 -> 1.0
  | n' -> x * power(x,n'-1);;
```

The prefix version ( $\oplus$ ) of an infix operator  $\oplus$  is a curried function.

For example:

```
(+);;  
val it : (int -> int -> int) = <fun:it@1>
```

Arguments can be supplied one by one:

```
let plusThree = (+) 3;;  
val plusThree : (int -> int)  
  
plusThree 5;;  
val it : int = 8
```

Function composition:  $(f \circ g)(x) = f(g(x))$

For example, if  $f(y) = y + 3$  and  $g(x) = x^2$ , then  $(f \circ g)(z) = z^2 + 3$ .

The infix operator `<<` in F# denotes functional composition:

```
let f y = y+3;;           // f(y) = y+3

let g x = x*x;;         // g(x) = x*x

let h = f << g;;        // h = (f o g)
val h : int -> int

h 4;;                   // h(4) = (f o g)(4)
val it : int = 19
```

Using just anonymous functions:

```
((fun y -> y+3) << (fun x -> x*x)) 4;;
val it : int = 19
```

Type of `<<` ?

## Basic Types: equality and ordering

The basic types: integers, floats, booleans, and strings type were covered last week. Characters are considered on the next slide. For these types (and many other) equality and ordering are defined.

In particular, there is a function:

$$\text{compare } x \ y = \begin{cases} > 0 & \text{if } x > y \\ 0 & \text{if } x = y \\ < 0 & \text{if } x < y \end{cases}$$

For example:

```
compare 7.4 2.0;;  
val it : int = 1
```

```
compare "abc" "def";;  
val it : int = -3
```

```
compare 1 4;;  
val it : int = -1
```

It is often useful to have **when guards** in patterns:

```
let ordText x y = match compare x y with
    | t when t > 0 -> "greater"
    | 0             -> "equal"
    | -            -> "less";;

ordText "abc" "Abc";;
val it : bool = true
```

The first clause is only taken when `t > 0` evaluates to true.

# Polymorphism and comparison

The type of `ordText`

```
val ordText : 'a -> 'a -> string when 'a : comparison
```

contains

- a **type variable** `'a`, and
- a **type constraint** `'a : comparison`

The type variable can be instantiated to any type **provided** comparison is defined for that type. It is called a **polymorphic type**.

For example:

```
ordText true false;;  
val it : string = "greater"
```

```
ordText (1,true) (1,false);;  
val it : string = "greater"
```

```
ordText sin cos;;  
... '(('a -> 'a)' does not support the 'comparison' ...
```

**Comparison is not defined for types involving functions.**

# Characters

Type name: `char`

Values `'a'`, `' '`, `'\''` (escape sequence for `'`)

## Examples

```
let isLowerCaseVowel ch =  
    System.Char.IsLower ch &&  
        (ch='a' || ch='e' || ch = 'i' || ch='o' || ch = 'u');;  
val isLowerCaseVowel : char -> bool
```

```
isLowerCaseVowel 'i';;  
val it : bool = true
```

```
isLowerCaseVowel 'I';;  
val it : bool = false
```

The *i*'th character in a string is achieved using the "dot"-notation:

```
"abc".[0];;  
val it : char = 'a'
```

## Overloaded Operators and Type inference

A squaring function on integers:

| Declaration                       | Type                       |         |
|-----------------------------------|----------------------------|---------|
| <code>let square x = x * x</code> | <code>int -&gt; int</code> | Default |

A squaring function on floats: `square: float -> float`

| Declaration                              |                                |
|--|--------------------------------|
| <code>let square(x:float) = x * x</code> | Type the argument              |
| <code>let square x:float = x * x</code>  | Type the result                |
| <code>let square x = x * x: float</code> | Type expression for the result |
| <code>let square x = x:float * x</code>  | Type a variable                |

You can mix these possibilities

An ordered collection of  $n$  values  $(v_1, v_2, \dots, v_n)$  is called an  $n$ -tuple

### Examples

|  |                    |
|--|--------------------|
| <pre>(3, false); val it = (3, false) : int * bool</pre>            | 2-tuples (pairs)   |
| <pre>(1, 2, ("ab", true)); val it = (1, 2, ("ab", true)) : ?</pre> | 3-tuples (triples) |

Equality defined componentwise, ordering lexicographically

```
(1, 2.0, true) = (2-1, 2.0*1.0, 1<2);;
val it = true : bool
```

```
compare (1, 2.0, true) (2-1, 3.0, false);;
val it : int = -1
```

provided = is defined on components

## Extract components of tuples

```
let ((x,_),(_,y,_)) = ((1,true),("a","b",false));;
val x : int = 1
val y : string = "b"
```

## Pattern matching yields bindings

### Restriction

```
let (x,x) = (1,1);;
...
... ERROR ... 'x' is bound twice in this pattern
```

## Examples

```
let g x =  
  let a = 6  
  let f y = y + a  
  x + f x;;  
val g : int -> int  
  
g 1;;  
val it : int = 8
```

Note: **a** and **f** are not visible outside of **g**

## Declaration of types and exceptions

Example: Solve  $ax^2 + bx + c = 0$

```
type Equation = float * float * float
type Solution = float * float
exception Solve; (* declares an exception *)
```

```
let solve(a,b,c) =
  if b*b-4.0*a*c < 0.0 || a = 0.0 then raise Solve
  else ((-b + sqrt(b*b-4.0*a*c))/(2.0*a),
        (-b - sqrt(b*b-4.0*a*c))/(2.0*a));;
val solve : float * float * float -> float * float
```

The type of the function `solve` is (the expansion of)

```
Equation -> Solution
```

`d` is declared once and used 3 times

readability, efficiency

```
let solve(a,b,c) =  
  let d = b*b-4.0*a*c  
  if d < 0.0 || a = 0.0 then raise Solve else  
  ((-b + sqrt d)/(2.0*a),(-b - sqrt d)/(2.0*a));;
```

```
let solve(a,b,c) =  
  let sqrtD =  
    let d = b*b-4.0*a*c  
    if d < 0.0 || a = 0.0 then raise Solve  
    else sqrt d  
  ((-b + sqrtD)/(2.0*a),(-b - sqrtD)/(2.0*a));;
```

Indentation matters

## Example: Rational Numbers

Consider the following **signature**, specifying operations and their types:

| Specification                        | Comment                                   |
|--------------------------------------|---|
| <code>type qnum = int * int</code>   | rational numbers                          |
| <code>exception QDiv</code>          | division by zero                          |
| <code>mkQ: int * int → qnum</code>   | construction of rational numbers          |
| <code>.+.: qnum * qnum → qnum</code> | addition of rational numbers              |
| <code>-.: qnum * qnum → qnum</code>  | subtraction of rational numbers           |
| <code>.*.: qnum * qnum → qnum</code> | multiplication of rational numbers        |
| <code>./.: qnum * qnum → qnum</code> | division of rational numbers              |
| <code>.=.: qnum * qnum → bool</code> | equality of rational numbers              |
| <code>toString: qnum → string</code> | String representation of rational numbers |

## Intended use

```
let q1 = mkQ(2,3);;
```

$$q_1 = \frac{2}{3}$$

```
let q2 = mkQ(12, -27);;
```

$$q_2 = -\frac{12}{27} = -\frac{4}{9}$$

```
let q3 = mkQ(-1, 4) .* q2 ./ q1;;
```

$$q_3 = -\frac{1}{4} \cdot q_2 - q_1 = -\frac{5}{9}$$

```
let q4 = q1 ./ q2 ./ q3;;
```

$$q_4 = q_1 - q_2/q_3 = \frac{2}{3} - \frac{-4}{9} / \frac{-5}{9}$$

```
toString q4;;
```

```
val it : string = "-2/15"
```

$$= -\frac{2}{15}$$

Operators are infix with usual precedences

Note: Without using infix:

```
let q3 = (./.)((.*.) (mkQ(-1,4)) q2) q1;;
```

Representation:  $(a, b)$ ,  $b > 0$  and  $\text{gcd}(a, b) = 1$

Example  $-\frac{12}{27}$  is represented by  $(-4, 9)$

### Greatest common divisor (Euclid's algorithm)

|  |  |
|--|--|
| <pre>let rec gcd = function     (0,n) -&gt; n     (m,n) -&gt; gcd(n % m,m);; val gcd : int * int -&gt; int</pre> | <pre>- gcd(12,27);; val it : int = 3</pre> |
|--|--|

Function to **cancel** common divisors:

```
let canc(p,q) =
  let sign = if p*q < 0 then -1 else 1
  let ap = abs p
  let aq = abs q
  let d = gcd(ap,aq)
  (sign * (ap / d), aq / d);;
```

```
canc(12,-27);;
val it : int * int = (-4, 9)
```

# Program for rational numbers

Declaration of the constructor:

```
exception QDiv;;
let mkQ = function
  | (_,0)  -> raise QDiv
  | pr     -> cancel pr;;
```

Rules of arithmetic:

$$\begin{aligned} \frac{a}{b} + \frac{c}{d} &= \frac{ad+bc}{bd} & \frac{a}{b} - \frac{c}{d} &= \frac{ad-bc}{bd} \\ \frac{a}{b} \cdot \frac{c}{d} &= \frac{ac}{bd} & \frac{a}{b} / \frac{c}{d} &= \frac{a}{b} \cdot \frac{d}{c} \quad \text{when } c \neq 0 \\ \frac{a}{b} = \frac{c}{d} &= ad = bc \end{aligned}$$

Program corresponds directly to these rules

```
let ( +. ) (a,b) (c,d) = cancel(a*d + b*c, b*d);;
let ( -. ) (a,b) (c,d) = cancel(a*d - b*c, b*d);;
let ( *. ) (a,b) (c,d) = cancel(a*c, b*d);;
let ( ./ ) (a,b) (c,d) = (a,b) *. mkQ(d,c);;
let ( =. ) (a,b) (c,d) = (a,b) = (c,d);;
```

Note: Functions must preserve the **invariant** of the representation

# Pattern matching and recursion

Consider `unzip` that maps a list of pairs to a pair of lists:

$$\begin{aligned} \text{unzip}([(x_0, y_0); (x_1, y_1); \dots; (x_{n-1}, y_{n-1})]) \\ = ([x_0; x_1; \dots; x_{n-1}], [y_0; y_1; \dots; y_{n-1}]) \end{aligned}$$

with the declaration:

```
let rec unzip = function
  | []          -> ([], [])
  | (x,y)::rest -> let (xs,ys) = unzip rest
                   (x::xs,y::ys);;

unzip [(1,"a");(2,"b")];;
val it : int list * string list = ([1; 2], ["a"; "b"])
```

Notice

- pattern matching on result of recursive call
- `unzip` is polymorphic. Type?
- `unzip` is available in the `List` library.

You are acquainted with a major part of the F# language.

- Higher-order (or curried) functions
- Basic types, equality and ordering
- Polymorphism
- Tuples
- Patterns
- A look at lists and list recursion