

The F# Language

Tomáš Petříček

F# Language

- Combination of imperative, functional and OOP approach
- Compiled on .NET 2.0 (supports also “Rotor” and .NET 1.1)
- Imperative
 - Mutable variables, loops, arrays
- Functional features
 - Functions as first class values
 - Tuples, lists, pattern matching, immutable data
- Object oriented
 - Classes, interfaces, polymorphism
 - Compiled as .NET classes
- Meta-programming support

F# Language - imperative vs. functional

- Recursive function

- immutable values, if is used as an expression

```
// factorial - recursive function (functional)
let rec fac1(n) =
    if (n = 0) then 1 else n * fac1(n-1);;
```

- Imperative

- uses for loop, mutable variables

```
// factorial - for loop (imperative)
let fac2(n) =
    let ret = ref 1 in
    for i = 1 to n do
        ret := !ret * i;
    done;
    !ret;;
```

F# Type system – why it is interesting?

- Statically typed – type safe
 - Uses type inference and type parameters (implemented using generics in .NET 2.0)

```
// immutable variable - type is inferred to int
let almost_answer = 41;;
```

```
// function adds 41 to parameter -> parameter must be integer
let add10(n) = n + almost_answer;;
```

- Functions as first class values

```
// function declaration using lambda expression
// type of lambda is "int -> int"
let lambda = fun x -> x + 3
```

```
// using lambda expressions for working with lists
let tmp = List.map (fun x -> x * 2) [1;2;3;4;5]
```

F# Type system – details

- **Tuples**

```
let tup1 = (1, 2);;  
let tup2 = (1, "ahoj");;
```

```
// function taking tuple  
let swap (a,b) = (b,a);;  
let tup1r = swap tup1;;
```

- **Arrays**

- compiled as .NET arrays

```
let arr = Array.create 3 ""  
do arr.(0) <- "hello"  
do arr.(1) <- " "  
do arr.(2) <- "world"  
  
let str =  
string.Join(",",arr);;
```

- **Lists**

- Cons operator (::)
- Concatenation operator (@)

```
let list = [1; 2; 3; 4]
```

```
// calculate sum (using pattern matching)  
let rec sum xs = match xs with  
    | []      -> 0  
    | y::ys  -> y + sum(ys);;
```

```
let list_sum = sum list;;
```

```
// reverse list using concatenation  
let rec reverse xs =  
    match xs with  
    | []      -> []  
    | y::ys  -> reverse(ys) @ [y];;
```

```
let list_r = reverse(list);;
```

F# Type system – details

- Functions are first class values in F#
 - function is type as any other
- Can be declared in two ways, but the result is the same
- Parameter binding is supported

```
// Simple declaration
// int -> int
let func x = x + 2;;
```

```
// int -> int -> int
let add x y = x + y;;
```

```
// Parameter binding
// int -> int
let add10 = add 10;;
```

```
// Lambda function
```

```
let lambda = fun x -> x + 3
```

```
// using lambda function
```

```
let tmp = List.map (fun x -> x+4)
[1;2;3]
```

F# Type system – details

- Discriminated unions, type parameters, records

```
// Discriminated union
// can be either Num, Add or Sum
type expr =
    Num of int
    | Add of expr * expr
    | Sub of expr * expr

// pattern matching
let rec eval e =
    match e with
    | Num n -> n
    | Add (x,y) -> eval x + eval y
    | Sub (x,y) -> eval x - eval y;;

// value of 'expr'
let exp =
    Add(Sub(Num 10,Num 5), Num 3)
let res =
    eval exp;;

// Records - "named tuples"
type key_value = {
    key : string;
    value : string;
}

// value of 'key_value'
let sth = { key="a"; value="Ahoj" }

// Records with type variables (generics)
type ('a, 'b) key_value_g = {
    key : 'a;
    value : 'b;
}

// type is identified thanks to type
inference
let sth_g = { key=10; value="deset"; }
```

F# Interoperability

- F# is compiled on .NET 2.0
- You can use .NET libraries in F#
 - dot notation for working with objects
 - “<-” operator for modifying values of properties
 - delegates created using lambda functions
- You can use F# libraries in any .NET language
 - F# libraries fully accessible from C#
 - (sometimes a bit tricky, because of F# functional approach)

F# Interoperability – Windows Forms

```
open System;;
open System.Windows.Forms;;
open System.Drawing;;

let form = new Form() in
form.Width <- 400;
form.Height <- 300;
form.Text <- "Hello from F#";

let btn = new Button() in
btn.Dock <- DockStyle.Fill;
btn.Font <- new Font("Arial", Float32.of_float 20.0);
btn.Text <- "Click me!";
form.Controls.Add btn;

btn.add_Click(new EventHandler (
    fun _ _ -> btn.Text <- "Clicked :-)";
));

Application.Run(form);
```

F# Interoperability – simple F# function

- F# function (in namespace Lib)

```
namespace Lib

let rec Factorial (x) =
    if (x = 0) then 1 else x * Factorial(x - 1);
```

- C# application that uses it

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Call F# method
        Console.WriteLine(Lib.Factorial(10));
    }
}
```

F# Interoperability – F# classes

- Full OOP support
 - abstract members, inheritance, polymorphism
- Simple class written in F#
 - All members must be initialized (in constructor)
 - Value of mutable members can change

```
type SampleObj = class
    val first : int
    val mutable second : int

    new(a,b) = { first=a; second=b }

    member x.First = x.first

    member x.Second
        with get() = x.second
        and set(v) = x.second <- v

    member x.Write() = printf "f=%i, s=%i\n" x.first
                        x.second
end
```

F# Meta-programming

- You can get tree representation of code written in F#
 - Called quotations in F#
 - It can be analyzed, translated to different languages etc...
 - Similar to lambda expressions in C# 3 (but more powerful)
- Quotations, quotation templates

```
// This is a quotation
// Type: int expr (expr with int as type parameter - result of
// expression)
let q = <@ 1 + 2 @>;;

// This is a quotation template
// Function that takes expression and "fills the hole"
// Type: int expr -> int expr
let template = <@ 1 + _ @>;;

// How to use the template
let ret = template <@ 10 @>;;
```

F# Meta-programming

- Quotation queries
 - tests whether template matches with quotation

```
// Sample quotation
let q = <@ sin 1.0 @>;

// Query - takes expression as parameter and returns:
// - Some(p) - where p is parameter passed to sin function
// - None - expressions don't match
let q = <@| sin _ |@>
```

- You can use pattern matching:

```
// Pattern matching using query
begin match <@| sin _ |@> q with
| Some(res) -> res
| None -> failwith "It's not sinus!"
end;;
```

F# Meta-programming

- Raw quotations
 - Allows you to work with underlying structure
 - Useful for quotation compilers and translators
 - Can be queried using `efApp`, `efInt32`, `efVar`, ... values
 - For more info see: <http://tomasp.net/articles/fsquotations.aspx>
- Structure of simple raw quotation:

```
> <@@ 1 + 10/5 @@>
```

```
val it : expr =  
  <@  
    Microsoft.FSharp.MLib.Pervasives.op_Addition (Int32 1)  
      (Microsoft.FSharp.MLib.Pervasives.op_Division (Int32 10) (Int32  
5))  
  @>
```

F# Resources

- F# homepage
 - <http://research.microsoft.com/fsharp>
- Don Syme's weblog
 - <http://blogs.msdn.com/dsyme>
- hubFS – F# community web
 - <http://cs.hubfs.net/>