# Platforma .NET

## 5. seminář – založeno na materiálech Jiřího Valůška

Radek Janoštík

Univerzita Palackého v Olomouci

23. 10. 2025

# Reakce na úkoly

- R/W problém většinou v pořádku
  - Občas nedostatečná synchronizace (čtení při zápisu)
- Rozmanitost v paralelizace
  - Změna pár řádků a funguje paralelně
  - Místy „masivní“ paralelizace
- Opět korektně ošetřovat výjimky

# F# – Úvod

- Autor: Dr. Don Syme z Microsoft Research (2005)
- Functional-first programovací jazyk
- Ale podpora i ostatních paradigmat (OOP, imperativní)
  - ale snaha se tomu vyhýbat (pokud to jde)
- Od .NET Core je multiplatformní
- Plnohodnotný .NET jazyk
  - Překládán do CIL (vše co s tím souvisí)
- Silně typový

# F# – Úvod

- Využití především na analýzu dat
  - Statistická analýza
  - Machine learning
  - Vizualizace dat
- Interaktivní skriptování – REPL
- Může běžet i ve webovém prohlížeči (Fable – kompilován do JS)
- Možné ale vytvářet i klasické aplikace, web, cloud

# F# – idea, vize

- Snaha o stručnost, omezení „kódovacího šumu“ (odvozování typů)
- Pohodlí (zjednodušit psaní), co největší znovupoužitelnost
- Čistota a správnost – typový systém, nemutovatelnost
- Jednoduchá paralelizace
- Interoperabilita s jinými jazyky (vs. čistota)

# F# – online zdroje

- F# Software Foundation `https://fsharp.org`
- Zkouška online: `https://try.fsharp.org`
- F# in y minutes `https://learnxinyminutes.com/docs/fsharp/`
- F# Programming WikiBook
  `https://en.wikibooks.org/wiki/F_Sharp_Programming`
- Oficiální dokumentace
  `https://learn.microsoft.com/en-us/dotnet/fsharp/`

# F# – knihy


- Isaac Abraham. Get Programming with F#. 2018. ISBN:9781617293993
- Robert Pickering, Kit Eason. Beginning F# 4.0. 2016. ISBN:1484213742, 9781484213742
- Antonio Cisternino, Adam Granicz, Don Syme. Expert F# 4.0. 2015. ISBN: 1484207424, 9781484207420
- Mathias Brandewinder. Machine Learning Projects for .NET Developers. 2015. ISBN: 1430267666, 9781430267669

# F# – zprovoznění

- Příkazový řádek: `dotnet fsi source.fs`
  - F# Interactive (REPL): `dotnet fsi`
- VisualStudio – nativní podpora, vytvoření projektu
  - F# Interactive: `View→Other windows→F# Interactive` (CTRL+ALT+F)
  - Každý řádek/blok kódu jde vyhodnotit: pravý klik → `Execute in Interactive` (ALT+Enter)
- JetBrains Rider – také nativní podpora
  - Aktivace pluginu: `Settings→Plugins→F#` – activate
  - F# Interactive: `Tools→F# Interactive→Start new`
  - Každý řádek/blok kódu jde vyhodnotit: `Tools→F# Interactive→Send to Interactive` (CTRL+\)
- VisualStudio Code – VSCode, stiskněte Ctrl+Shift+P a zadejte příkaz:
  `ext install Ionide-fsharp`

# F# – základy

- Komentáře:
  - Jednořádkový klasicky pomocí //
  - Více řádků: začátek (* konec *)

```
// jednoradkovy komentar
(* viceradkovy
komentar *)
```

- Rozlišení kódu pro F# Interactive pomocí direktiv

```
#if INTERACTIVE
printfn "I'am using the F# interactive"
#else
printfn "I was called from the code"
#endif
```

- Příkazy do F# Interactive musí končit dvěmi středníky `;;`

# F# – základy – hodnoty

- Deklarace hodnot („proměnných“)
  - Nejsou mutovatelné, nefunguje příkaz přiřazení

```
let x = 5
let y = 6
let z = x + y
let a = 3.5
let text = "Jazyk F#"
let Cesko = "Ceska republika"
printfn "z: %i" z
printfn "a: %f" a
printfn "text: %s" text
printfn "stat: %s" Cesko
```

# F# – základy – funkce

- Vytvoření anonymní funkce

```
fun x y -> x + y
```

- Deklarace pojmenované funkce

```
let add = fun x y -> x + y
```

- Zkráceně

```
let add x y = x + y
```

- Volání funkcí:

```
> add 20 22;;
```

- Víceřádkové funkce musíme odsazovat

```
let myPrint a b c =
    printfn "1. %s" a
    printfn "2. %s" b
    printfn "3. %s" c
```

# F# – lightweight syntax

- F# je ve výchozím stavu white-space sensitive – používá tzv. lightweight syntax
- Ideou je toho psát co nejméně
- Existuje i verbose syntax: `https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/verbose-syntax`
- Jde vypnout pomocí direktivy na začátku souboru

```
#light "off"
```

# Inference typů

- Každá hodnota má typ, což zahrnuje i hodnoty, které jsou funkce

```
> let x = 1;;
val x : int = 1
> let s = "Ahoj";;
val s : string = "Ahoj"
> let add x y = x + y;;
val add : x:int -> y:int -> int
```

- Je možné typ hodnoty definovat

```
let add (x: float) (y: float) = x + y;;
val add : x:float -> y:float -> float
> let f x = x;;
val f : x:'a -> 'a
```

- Typy začínající ' jsou tzv. proměnné typy (generické typy)

# Větvení

- If je výraz(expression) nikoliv příkaz(statement) – vrací hodnotu

```
if expr then
    expr
elif expr then
    expr
elif expr then
    expr
else
    expr
```

- Operátory: =, <, <=, >, >=, <> (není rovno)
- Logické operátory:&& (AND), || (OR), not (negace)

# Funkce

- V rámci funkcí je možné definovat další funkce – vnořené funkce

```
let myFun a b =
    let add x y = x + y
    let sub x y = x - y
    (add a (sub a b))
```

- Rekurzivní funkce musíme označit klíčovým slovem `rec`

```
let rec fib n =
    if n = 0 then 0
    elif n = 1 then 1
    else fib (n - 1) + fib (n - 2)
```

# Pattern matching

- Moderní přístup jak transformovat vstupní data
- Pattern(vzor) – „dotaz“ na strukturu, tvar, přesné hodnoty v datech
- Usnadní výrazy v podmínkách

```
match expr with
    | pat1 -> result1
    | pat2 -> result2
    | pat3 when expr2 -> result3
    | _ -> defaultResult
```

- Příklad:

```
let rec fib n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | _ -> fib (n - 1) + fib (n - 2)
```

# Pattern matching (alternativní syntax)

```
let getPrice = function
| "banana" -> 0.79
| "watermelon" -> 3.49
| "tofu" -> 1.09
| _ -> nan // Not a Number

> getPrice "tofu";;
val it : float = 1.09

let rec fac = function
    | 0 | 1 -> 1
    | n -> n * fac (n - 1)
```

- Pattern matching se vyhodnocuje shora dolů

# Funkce vyšších řádů

- F# samozřejmě podporuje funkce vyšších řádů:

```
let square x = x * x
let cube x = x * x * x
let passFive f = f 5
printfn "%i" (passFive square)
printfn "%i" (passFive cube)
```

- Také currying

```
let add x y = x + y
let addFive = add 5
> addFive 12;;
val it : int = 17
```

- `let add x y = x + y` je totiž formálně jen zkratka za:

```
let add = (fun x -> (fun y -> x + y))
```

# N-tice (tuples)

- Seskupení nepojmenovaných, ale seřazených hodnot
- Mohou mít odlišné typy
- Jak referenční(výchozí), tak hodntové
- Časté využité v pattern matchingu, přiřazení více proměnným, více návratových hodnot

```
let a = (1, 2)
let b = (1, 3, 5)
let c = (1, "text")

let avg (a, b, c) = (a + b + c) / 3
let getPowTuple a = (a, a*a)

let d = struct (1, 2)
```

# N-tice – použití

- Pattern matching

```
let greeting (name, language) =
    match (name, language) with
    | ("Steve", _) -> "Howdy, Steve"
    | (name, "English") -> "Hello, " + name
    | (name, _) when language.StartsWith("Span") -> "Hola, " + name
    | (_, "French") -> "Bonjour!"
    | _ -> "DOES NOT COMPUTE"
```

- Přiřazení více hodnotám

```
let val1, val2, ... valN = (expr1, expr2, ... exprN)
let x, y = (1,2)
```

- Návratová hodnota

```
> let pows x = (x*x, x*x*x, x*x*x*x);;
val pows: x: int -> int * int * int
> pows 5;;
val it: int * int * int = (25, 125, 625)
```

# Záznamy (records)

- Sdružení pojmenovaných hodnot
- Referenční(výchozí) i hodnotové
- Definice:

```
type recordName = { [ fieldName : dataType ] ... }
```

- Příklad:

```
type website = { Title : string; Url : string }
let homepage = { Title = "Google"; Url = "http://www.google.com" }
```

- Ve výchozím stavu nemutovatelné

```
type coords = { X : float; Y : float }
let setX item newX = { item with X = newX }
let start = { X = 1.0; Y = 2.0 }
let finish = setX start 15.5
```

# Záznamy (records)

- Nad záznamy lze také provádět Pattern matching

```
type coords = { X : float; Y : float }
let getQuadrant = function
    | { X = 0.0; Y = 0.0 } -> "Origin"
    | item when item.X >= 0.0 && item.Y >= 0.0 -> "I"
    | item when item.X <= 0.0 && item.Y >= 0.0 -> "II"
    | item when item.X <= 0.0 && item.Y <= 0.0 -> "III"
    | item when item.X >= 0.0 && item.Y <= 0.0 -> "IV"
```

# Discriminated union

- Datový typ, který může nabývat právě jedné hodnoty z pevně daných
- Jednotlivé případy mohou být různých datových typů
- Mohou být rekurzivní – vhodné pro rekurzivní datové struktury (stromy, seznamy)

```
type unionName =
   | Case1
   | Case2 of datatype
   | ...

type switchstate =
   | On
   | Off

> let x = On;;
val x : switchstate = On
```

# Discriminated union – strom

- Definice a vytvoření stromu

```
type tree =
    | Leaf of int
    | Node of int * tree * tree

let myTree1 = Node(1,Leaf(2),Leaf(3))
let myTree2 = Node(1,
   Node(2,Leaf(3),Leaf(4)),
   Leaf(5))
```

- I discriminated union můžeme používat v Pattern matchingu

```
let rec printTree = function
    | Leaf value -> printf "Hodnota v listu %i;" value
    | Node (value, left, right) ->  printf "Hodnota v uzlu %i;" value;
        printTree left; printTree right;
```

# Discriminated union – option type

- Předdefinovaný datový typ, který může nabývat pouze dvou hodnot
  - Some – má hodnotu
  - None – nemá hodnotu
- Definice ve standardní knihovně:

```
type Option<'a> =
    | Some of 'a
    | None
```

- Příklad použití:

```
let safediv x y =
   match y with
   | 0 -> None
   | _ -> Some(x/y)
```

# Seznamy

- Uspořádaná sekvence hodnot stejného datového typu
- Nemutovatelné
- Prázdný seznam: `[]`

```
let list1 = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

- Vytvoření pomocí cons operátoru `::`

```
let list2 = 1::2::3::[]
```

- Další možnosti vytvoření

```
let list3 = List.init 5 (fun index -> index * 3) // pouziti List.init
let list4 = [1 .. 10] // list comprehensions
let list5 = ['a' .. 'f'] // list comprehensions
let dalsi = 100::list1
let spojene = list1 @ dalsi
```

- Možný přístup přes index (pamatovat na neefektivitu)

```
> list1[3];;
val it: int = 4
```

# Seznamy – Pattern matching

- Seznam můžeme do Pattern matchingu rozložit pomocí cons `::`

```
let sum l =
    let rec sumStep counter = function
        | [] -> counter
        | h::t when t = [] -> sumStep (counter + 5 * h) t
        | h::t -> sumStep (counter + h) t
    sumStep 0 l
```

- Velké množství funkcí v modulu `List`
  - List.rev
  - List.filter
  - List.map
  - List.append nebo infixový operátor @
  - List.choose – filter a map zároveň
  - List.fold a List.foldBack
  - List.find a List.tryFind – find při nenalezení prvku vyvolá vyjímku

# Seznamy

- Na co se vyhodnotí:

```
List.rev (List.map (fun x -> x * x) (List.filter (fun x -> x % 2 = 1)
   (List.init 8 (fun index -> index * 3))))
```

```
val it: int list = [441; 225; 81; 9]
```

- Syntaxe se stává nepřehlednou ⇒ zaveden *Pipe-forward* operátor `|>`

```
let (|>) x f = f x
```

- Vede na značné zpřehlednění

# Seznamy

- Předchozí kód tak může vypadat takto:

```
List.init 8 (fun index -> index * 3)
|> List.filter (fun x -> x % 2 = 1)
|> List.map (fun x -> x * x)
|> List.rev

val it: int list = [441; 225; 81; 9]
```

# Úkol

- Naprogramujte následující funkce (souvisí spolu):
  1. `gcd`, která bude počítat největšího společného dělitele.
  2. `addFrac, subFrac, mulFrac, divFrac` pro sčítání, odčítání, násobení a dělení zlomků (funkce budou vracet zlomky v základním tvaru)
  3. `comb`, která spočítá kombinační číslo
- Pro jednoduchost stačí, že budou všechny funkce počítat pouze s celými čísly
  - Zkuste vhodně navrhnout a zvolit datové typy
- Dále naprogramujte funkce
  1. `sumOfTree`, která spočítá součet hodnot ve výše definovaném stromu
  2. `leavesToList`, která vrátí všechny hodnoty z listů v jednom seznamu