Pattern matching

Pattern matching (překlad podle smyslu by byl něco jako vybírání podle vzoru) je velmi důležitá konstrukce.
Je to jakýsi switch. Vámi daná hodnota se porovná s danými vzory ("patterns") a první,
který na ní pasuje ("matching"), se použije.

Syntaxe je

```
match hodnota with
  [|] vzor_1 [when podm_1] -> vyraz_1
   |  vzor_2 [when podm_2] -> vyraz_2
  ...
   |  vzor_n [when podm_n] -> vyraz_n
```

Části v hranatých závorkách jsou nepovinné.

Začněme příklady:

```
match 1 with 0 -> "nula"
           | 1 -> "jedna"
```

Ekvivalentní zápis (všimněte si prvního znaku |):

```
match 1 with
  | 0 -> "nula"
  | 1 -> "jedna"
```

Tento nesmyslný kód vezme hodnotu 1 a začne ji porovnávat s danými vzory. Porovnání s nulou se nepovede,
porovnání s 1 ano, takže se vrátí řetězec "jedna".

Jak přesně funguje porovnávání s vzory: Řekněme, že jsme do `match` dali číselnou hodnotu.
♣ Pokud je vzor číslo, porovnávají se hodnoty.
♣ Pokud je vzor podtržítko, porovnávání se vždy povede, tj. `match cokoliv with _ -> "a"` vrátí vždy "a".
♣ Pokud je vzor jméno proměné, porovnávání se také vždy povede, a proměnná potom obsahuje onu číselnou hodnotu.
Následuje další příklad:

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

Protože je časté definovat funkce pomocí pattern matchingu, tj. používat `fun i -> match i with ...`,
existuje v F# zkratka –– výraz `function` je přesně to samé jako `fun prom -> match prom with`,
přičemž `prom` je nějaké vám nedostupné a v tom místě nepoužité jméno proměnné.
Řečeno slovně, `function` definuje novou funkci pomocí pattern matchingu.

Pro ilustraci následující dvě ekvivalentní definice:

```
let rec factorial n = match n with          let rec factorial = function
  | 1 -> 1                                    | 1 -> 1
  | n -> n * factorial (n - 1)                | n -> n * factorial (n - 1)
```

Výrazy v pattern matchingu mohou tedy být hodnoty, podtržítka či jména proměnných.
Dále to mohou být intervaly hodnot:

```
let is_lower = function | 'a' .. 'z' -> true
                        | _ -> false
```

Výrazy v pattern matchingu mohou být kombinovány pomocí | (nebo) nebo & (a současně):

```
let rec fib = function                        let dvojice = function
  | 0 | 1 -> 1                                  | (0, _) && (_, 0) -> "samé nuly"
  | n -> fib (n - 1) + fib (n - 2)              | _ -> "nějaká nenulová hodnota"
```

Použitý výraz `(0, _) && (_, 0)` je zde nadbytečný, `(0, 0)` by bylo stejné.

Každý vzor v pattern matchingu může mít ještě přiřazenou podmínku pomocí `when podmínka_typu_bool`.
Pokud vzor vyhovuje dané hodnotě, dojde ještě k vyhodnocení této podmínky a k namatchování dojde jenom,
když podmínka vrátí `true`.

Pro příklad:

```
let rec factorial = function
  | n when n < 0 -> invalidArg "n" "n cannot be negative"
  | 0 | 1 -> 1
  | n -> n * factorial (n - 1)
```

Pokud pattern matching neuspěje (žádný vzor neodpovídá), program vyvolá výjimku `MatchFailureException`.

Pattern matching a struktury a algebraické datové typy

Od minule umíme vytvářet vlastní datové typy jako struktury s pojmenovanými položkami či algebraické datové typy.
V celém následujícím textu předpokládejme následující definice:

```
type Point<'a> = { X : 'a; Y : 'a }
type Tree<'a> = Nil | V of Tree<'a> * 'a * Tree<'a>
```

Připomeňme, že seznamy prvků vypadají skoro jako

```
type list<'a> = ([]) | (::) of 'a * list<'a>
```

Je jednoduché konstruovat nyní hodnoty typu `Point` či `Tree`:

```
{ X = 4; Y = 6 }
V (V (Nil, 1, Nil), 2, V (Nil, 3, Nil))
```

Seznamy můžeme konstruovat stejně, nebo můžeme použít zabudovaného zkráceného zápisu:

```
1 :: []                  nebo  [1]
1 :: 2 :: 3 :: []        nebo  [1; 2; 3]
```

Zatím ale neumíme s hodnotami typu `Point`, *`list`* či `Tree` pracovat, protože je neumíme "rozebrat".
K tomu také slouží pattern matching.

Doteď jsme pattern matching používali pouze s číselnými hodnotami, můžeme ho ale použít s libovolným typem.

Pro struktury můžeme dostat hodnoty jednotlivých položek:

```
let get_x = function
  | { X = hodnota_x } -> hodnota_x
```

Pattern matching se provádí na jednotlivých položkách dané struktury:

```
let is_x_zero = function
  | { X = 0 } -> "yes"
  | { X = _ } -> "no"
```

v tomhle konkrétním případě by fungovalo i `| _ -> "no"`

Jméno proměnné a jméno položky struktury se mohou shodovat, čili toto funguje:

```
let get_x = function
  | { X = X } -> X
```

Pattern matching lze použít i pro algebraické datové typy jako *`list`* nebo `Tree`:

```
let je_to_nil = function
  | Nil -> true
  | _ -> false

let hodnota_vrcholu = function
  | V (_, x, _) -> x
```

Při použití poslední funkce překladač řekne

```
(*   warning FS0025: Incomplete pattern matches on this expression. For example,
     the value 'Nil' may indicate a case not covered by the pattern(s).*)
```

a program spadne na výjimku `MatchFailureException`, pokud spustíme funkci `hodnota_vrcholu` s `Nil`.

Následují funkce najde a vrátí nejlevější prvek ve stromě, pokud tam existuje:

```
let rec leftmost = function
  | Nil -> None
  | V (Nil, x, _) -> Some x
  | V (l, _, _) -> (* l is not Nil *) leftmost l
```

Následující funkce zvýší hodnotu v každém vrcholu o jedna

```
let rec inc = function
  | Nil -> Nil
  | V (l, x, r) -> V (inc l, x + 1, inc r)
```

Se seznamy to funguje stejně, pro ukázku funkce pro zjištění počtu prvku v seznamu:

```
let rec len = function
  | [] -> 0
  | _ :: xs -> 1 + len xs
```

Zachování pouze těch hodnot seznamu, které projdou daným filtrem (funkce vracející *`bool`* pro hodnotu ze seznamu):

```
let rec filter should_keep = function
  | [] -> []
  | x :: xs when should_keep x -> x :: filter should_keep xs
  | _ :: xs (* otherwise *)    -> filter should_keep xs
```

Nebo trochu jinak:

```
let rec filter should_keep = function
  | [] -> []
  | x :: xs -> let xs' = filter should_keep xs
               if should_keep x then x :: xs' else xs'
```

Tail rekurze

Již zmíněná funkce pro spočítání délky seznamu má jednu nevýhodu:

```
let rec len = function
  | [] -> 0
  | _ :: xs -> 1 + len xs
```

Problém je v tom, že než se provede výraz `1 + len xs`, musí se vyhodnotit `len xs`.
Tedy dojde k rekurzivnímu volání, po jehož vyhodnocení se k výsledku přičte jednička.

Takový výpočet tedy potřebuje tolik vnořených rekurzivních volání, jaká je délka seznamu.
To je nemilé, protože zásobník je na většině systémů ve výchozím nastavení omezen (1MB Windows, 8MB Linux).

Uvažme následující definici funkce `len`:

```
let len xs =
  let rec len' len_so_far = function | [] -> len_so_far
                                     | _ :: xs -> len' (len_so_far + 1) xs
  len' 0 xs
```

Použili jsme pomocnou funkci, která má dva parametry –– délka již zpracované části seznamu
a dosud nezpracovaná část seznamu. Délce zpracované části seznamu se také říká "akumutátor".

Tato definice nepotřebuje žádné rekurzivní volání díky "tail rekurzi".

Většina funkcionálních jazyků (včetně jazyka F#) poskytuje tail rekurzi. To znamená, že pokud je výsledek
funkce roven výsledku jiné funkce (například `f x = g (2 * x)`), nepoužije se k vyhodnocení rekurze.
Ještě před rekurzivním voláním vnitřní funkce dojde k odstranění vnější funkce ze zásobníku.

V případě rekurzivní funkce jako již zmíněné `len'`:

```
let rec len' len_so_far = function | [] -> len_so_far
                                   | _ :: xs -> len' (len_so_far + 1) xs
```

není tedy k vyhodnocení třeba žádný prostor na zásobníku, místo toho se použije cyklus typu **`while`** k projití seznamu.

Sekvence typu seq<'a>

Již známe seznamy typu *list*<'a>. To je jednosměrný spojový seznam hodnot.
Kromě něj existuje ještě druhý důležitý typ uchovávající posloupnost prvků, a to "sekvence", *seq*<'a>.
Pro znalé .NETu, *seq*<'a> je jenom typový alias pro `IEnumerable<'a>`,
tj. *seq*<'a> je definováno jako **`type`** *seq*<'a> = `IEnumerable<'a>`.

Stejně jako *list*<'a> reprezentuje *seq*<'a> sekvenci prvků, ke které lze přistupovat jako k jednosměrnému
spojovému seznamu. Na rozdíl od seznamu ale není tato sekvence reprezentovaná jako spojový seznam v paměti.
Sekvence *seq*<'a> je objekt, který umí vrátit následující prvek v sekvenci. Také si ji můžete představit
jako iterátor. Prvky jako takové nemusí být vůbec nikde v paměti uloženy, jenom existuje kód, který je generuje.

Shodné vlastnosti *list*<'a> a *seq*<'a>:
♣ Postupný přístup k prvkům v pořadí od prvního k poslednímu, jako jednosměrný spojový seznam.

Rozdílné vlastnosti *list*<'a> a *seq*<'a>:
♣ Prvky *list*<'a> jsou vždy v paměti jako jednosměrný spojový seznam, projít je trvá lineární čas
♣ Prvky *seq*<'a> nemusí být reprezentované explicitně v paměti, projití může trvat libovolně dlouho
(kód generující prvky je může generovat libovolně dlouho nebo se dokonce zacyklit)
♣ *list*<'a> reprezentuje konečný seznam, *seq*<'a> je často nekonečná (například sekvence generující samé 1).
♣ Při práci s *list*<'a> se často používá pattern matching, při práci s *seq*<'a> nikoliv.

Pro práci se sekvencemi slouží funkce v modulu `Seq`. Zde vyjmenuji pouze ty základní:
♣ Konstrukce:

`Seq.empty : seq<'T>`	prázdná sekvence
`Seq.singleton : 'T -> seq<'T>`	sekvence obsahující jeden prvek
`Seq.ofList : list<'T> -> seq<'T>`	sekvence vracející prvky daného seznamu
`Seq.init : int -> (int -> 'T) -> seq<'T>`	sekvence vytvořená pomocí délky a funkce, která dostane index a vrátí prvek na tomto indexu
`Seq.initInfinite : (int -> 'T) -> seq<'T>`	jako `init`, jenom je sekvence nekonečná

`Seq.unfold : ('State -> 'T * 'State option) -> 'State -> seq<'T>`
Generuje prvky sekvence postupně. Začne v daném stavu.
Poté vždy zavolá danou funkci v aktuálním stavu. Tato funkce buď nevrátí nic (konec sekvence) nebo vrátí jednak další prvek sekvence a jednak nový stav.
Příklad: funkce generující přirozená čísla je `Seq.unfold (fun i -> Some (i, i+1)) 1`

♣ Základní vlastnosti:

`Seq.head : seq<'T> -> 'T`	první prvek sekvence
`Seq.isEmpty : seq<'T> -> bool`	je sekvence prázdná?
`Seq.length : seq<'T> -> int`	délka sekvence
`Seq.nth : int -> seq<'T> -> 'T`	i–tý prvek sekvence počítáno od 0

♣ Další metody konstrukce:

`Seq.append : seq<'T> -> seq<'T> -> seq<'T>`	spojí dvě sekvence
`Seq.concat : seq<#seq<'T>n> -> seq<'T>`	spojí sekvenci sekvencí do jedné placaté sekvence
`Seq.take : int -> seq<'T> -> seq<'T>`	vrátí prvních n prvků, výjimka, když je jich málo
`Seq.truncate : int -> seq<'T> -> seq<'T>`	jako `take`, ale prvků může být méně než n
`Seq.skip : int -> seq<'T> -> seq<'T>`	vrátí sekvenci bez prvních n prvků
`Seq.takeWhile : ('T -> bool) -> seq<'T> -> seq<'T>`	vrátí první prvky splňující podmínku
`Seq.skipWhile : ('T -> bool) -> seq<'T> -> seq<'T>`	přeskočí první prvky splňující podmínku

♣ Ostatní:

`Seq.cache : seq<'T> -> seq<'T>`
Vytvoří sekvenci, která si pamatuje již jednou spočtené hodnoty. Každá nová hodnotu, kterou sekvence vygeneruje, je uložena do spojového seznamu a pokud je potřeba znovu, nepočítá se znova, ale vrátí se ta uložená.
Každá hodnota v sekvenci je tedy vyhodnocena jednou nebo vůbec, ať už je potřeba kolikrátkoli.

`Seq.filter : ('T -> bool) -> seq<'T> -> seq<'T>`	vrátí jen ty prvky vyhovující dané podmínce
`Seq.iter : ('T -> unit) -> seq<'T> -> unit`	spustí danou funkci na každý prvek sekvence
`Seq.map : ('T -> 'U) -> seq<'T> -> seq<'U>`	prožene každý prvek danou funkcí
`Seq.toList : seq<'T> -> list<'T>`	vyhodnotí sekvenci a vrátí její prvky v seznamu

`Seq.{fold, reduce, scan, mapi, iteri, zip, zip3, (max, min, sort, sum)[_by]}`

Konstruovat sekvence tedy můžeme pomocí `empty`, `singleton`, `ofList`, `init`, `initInfinite`, `unfold`
Kromě toho lze použít konstrukci podobnou vytváření seznamu, přesně:
`seq { 1 .. 10 }` nebo jenom `{1..10}`
`seq { 10 .. -1 .. 1 }` nebo jenom `{10..-1..1}`
`seq { for i in {1..10} -> i * i }`
`seq { for i in {1..10} -> if isprime i then yield i }`

Pro práci s hodnotami v sekvenci můžeme použít `iter`, `head` a `skip 1`, `fold`
nebo můžeme sekvenci převést pomocí `toList` na seznam a ten zpracovat. Jenom pozor na to, že sekvence může být nekonečná :–)

```
let ones = Seq.initInfinite (fun _ -> 1)
let from n = Seq.initInfinite ((+) n)
```

Kód pro rozpoznání prvočíselnosti využívající sekvence:

```
let is_prime n =
  {2..n} |> Seq.takeWhile (fun i -> i * i <= n)
         |> Seq.filter (fun i -> n % i = 0)
         |> Seq.isEmpty
```

Sekvence prvočísel:

```
let primes =
  let rec next_prime = function
    | n when is_prime n -> Some (n, n+1)
    | n (* otherwise *) -> next_prime (n + 1)
  Seq.unfold next_prime 2
```

Seznam prvních 1000 prvočísel: `primes |> Seq.take 1000 |> Seq.toList`

Funkce main

Funkci main můžete vytvořit dvěma různými způsoby. Ani v jednom nezáleží na jejím jméně.
Pokud nepotřebujete argumenty ani vracet návratovou hodnotu jinou než 0, jako poslední funkci v modulu napište:

```
let main = (* zde kód vracející libovolný typ, například*) printf "Hello world!"
```

Pokud chcete argumenty nebo vracet návratový kód jiný než 0, zadefinujte

```
[<EntryPoint>]
let main args = (* zde kód vracející návratovou hodnotu typu @@int@@*)
```

Argumenty jsou typu pole stringů, tj. *string* `[]`, přičemž obsahuje jenom skutečné argumenty, ne jméno spouštěného programu jako například program v C.