Funkce v modulu List a Seq

Modul `List` a `Seq` nabízí mnoho užitečných funkcí pracujících s posloupnostmi prvků.
Začneme všemi funkcemi z modulu `List`. Pokud vás to nudí, skočte rovnou na nejzajímavější část Fold a přátelé:

♣ Základní funkce

`empty : 'T list`	O(1), prázdný seznam
`head: 'T list -> 'T`	O(1), první prvek seznamu
`isEmpty : 'T list -> bool`	O(1), je seznam prázdný
`length : 'T list -> int`	O(N), délka seznamu
`nth : 'T list -> int -> 'T`	O(i), vrací i–tý prvek seznamu, bráno od 0
`tail : 'T list -> 'T list`	O(1), vrací seznam bez prvního prvku

♣ Vytváření seznamů z jednodušších

`replicate : (N:int) -> (x:'T) -> 'T list`	O(N), vytvoří seznam s N kopiemi prvku x
`init : (N:int) -> (f:int->'T) -> 'T list`	O(N), vytvoří seznam s hodnotami f(0),f(1),...,f(N–1)
`append, (@) : 'T list -> 'T list -> 'T list`	O(délka prvního seznamu), spojí dva seznamy
`concat : seq<'T list> -> 'T list`	O(délka všech seznamů kromě posledního), spojí sekvenci seznamů (i seznam seznamů) do jednoho
`rev : 'T list -> 'T list`	O(N), obrátí prvky seznamu
`filter : (p:'T->bool) -> 'T list -> 'T list`	O(N), vrátí seznam s prvky, pro které je `p` splněno

`map : (f:'T->'U) -> 'T list -> 'U list`
O(N), vytvoří seznam aplikací funkce f na každý prvek, tj. `map f xs = [for x in xs -> f x]`

`map2 : (f:'T1->'T2->'U) -> 'T1 list -> 'T2 list -> 'U list`
O(N), totéž jako `map`, ale aplikuje funkci `f` na dva seznamy paralelně, tedy
`map2 f [x_0; ...; x_N] [y_0; ...; y_N] = [f x_0 y_0; ...; f x_N y_N]`

`map3 : (f:'T1->'T2->'T3->'U) -> 'T1 list -> 'T2 list -> 'T3 list -> 'U list`
O(N), jako `map2`, ale pro 3 seznamy

`mapi : (f:int->'T->'U) -> 'T list -> 'U list`
O(N), jako `map`, ale funkce `f` dostává i index prvku, tedy
`mapi f [x_0; x_1; ...; x_N] = [f 0 x_0; f 1 x_1; ...; f N x_N]`

`mapi2 : (f:int->'T1->'T2->'U) -> 'T1 list -> 'T2 list -> 'U list`
O(N), kombinace `map2` a `mapi`

`collect : (f:'T->'U list) -> 'T list -> 'U list`
O(N), aplikuje `f` na každý prvek a výsledné seznamy spojí. Platí: `collect f = map f >> concat`

`choose : (f'T->'U option) -> 'T list -> 'U list`
O(N), jako `map`, ale zachová jenom hodnoty `Some x`. Platí: `choose f = collect (f>>Option.toList)`

`zip : 'T1 list -> 'T2 list -> ('T1 * 'T2) list`
O(N), vytvoří seznamy odpovídajících dvojic ze dvou seznamů. Platí: `zip = map2 (fun a b -> a, b)`

`zip3 : 'T1 list -> 'T2 list -> 'T3 list -> ('T1 * 'T2 * 'T3) list`
O(N), vytvoří seznamy odpovídajících trojic z tří seznamů. Platí: `zip3 = map3 (fun a b c -> a, b, c)`

`unzip : ('T1 * 'T2) list -> 'T1 list * 'T2 list`
O(N), opak k `zip`

`unzip3 : ('T1 * 'T2 * 'T3) list -> 'T1 list * 'T2 list * 'T3 list`
O(N), opak k `zip3`

`partition : (p:'T->bool) -> 'T list * 'T list`
O(N), rozdělí prvky seznamu podle hodnoty `p x` do dvou –– v prvním vrací prvky **`true`**, v druhém **`false`**

`permute : (f:int->int) -> 'T list -> 'T list`
O(N), vytvoří seznam z prvků pomocí dané permutace `f`

`sort : 'T list -> 'T list`
O(N log N), setřídí prvky seznamu stabilním tříděním

`sortBy : (f:'T->'Key) -> 'T list -> 'T list`
O(N log N), jako `sort`, ale třídí pomocí klíčů získaných funkcí `f`

`sortWith : (cmp:'T->'T->int) -> 'T list -> 'T list`
O(N log N), jako `sort`, ale pro porovnávání používá funkci `cmp` (vrací <0, 0, >0 jako v qsortu)

♣ Hledání prvků

`tryFind : (p:'T->bool) -> 'T list -> 'T option`
O(N), vrátí první prvek, pro který vrátí `p` hodnotu **`true`**

`tryFindIndex : (p:'T->bool) -> 'T list -> int option`
O(N), vrátí index prvního prvku, pro který vrátí `p` hodnotu **`true`**

`tryPick : (f:'T->'U option) -> 'T list -> 'U option`
O(N), vrátí hodnotu `y` pro první prvek, pro který vrátí `f` hodnotu `Some y`

`find : (p:'T->bool) -> 'T list -> 'T`	O(N), jako `tryFind`, prvek musí existovat
`findIndex : (p:'T->bool) -> 'T list -> int`	O(N), jako `tryFindIndex`, prvek musí existovat
`pick : (f:'T->'U option) -> 'T list -> 'U`	O(N), jako `tryPick`, prvek musí existovat

♣ Vytváření jedné hodnoty ze všech prvků seznamu
exists : (p:'T->*bool*) -> (xs:'T *list*) -> *bool*
  O(N), vrací **or** všech hodnot p x, tj. **or** hodnot map p xs
exists2 : (p:'T1->T2->*bool*) -> (xs:'T1 *list*) -> (ys:'T2 *list*) -> *bool*
  O(N), vrací **or** všech hodnot odpovídajících dvojic hodnot p x y, tj. **or** hodnot map2 p xs ys
forall : (p:'T->*bool*) -> (xs:'T *list*) -> *bool*
  O(N), vrací **and** všech hodnot p x, tj. **and** hodnot map p xs
forall2 : (p:'T1->T2->*bool*) -> (xs:'T1 *list*) -> (ys:'T2 *list*) -> *bool*
  O(N), vrací **and** všech hodnot odpovídajících dvojic hodnot p x y, tj. **and** hodnot map2 p xs ys
average : 'T *list* -> ^T   O(N), vrací průměr všech hodnot
averageBy : ('T -> ^U) -> 'T *list* -> ^U   O(N), averageBy f = map f >> average*)
max : 'T *list* -> 'T   O(N), vrací maximum všech hodnot
maxBy : ('T -> 'U) -> 'T *list* -> 'T   O(N), maxBy f = map f >> max*)
min : 'T *list* -> 'T   O(N), vrací minimum všech hodnot
minBy : ('T -> 'U) -> 'T *list* -> 'T   O(N), minBy f = map f >> min*)
sum : ^T *list* -> ^T   O(N), vrací součet všech hodnot
sumBy : ('T -> ^U) -> 'T *list* -> ^U   O(N), sumBy f = map f >> sum*)
iter : (f:'T->*unit*) -> 'T *list* -> *unit*   O(N), provede f postupně na všech hodnotách seznamu
iter2 : (f:'T1->'T2->*unit*) -> 'T1 *list* -> 'T2 *list* -> *unit*
  O(N), provede f postupně na všech odpovídajících dvojicích seznamu, tj. iter2 f = map2 f >> iter
iteri : (f:*int*->'T->*unit*) -> 'T *list* -> *unit*
  O(N), provede f postupně na všech hodnotách seznamu a jejich indexech
iteri2 : (f:*int*->'T1->'T2->*unit*) -> 'T1 *list* -> 'T2 *list* -> *unit*
  O(N), kombinace iter2 a iteri

♣ Konverzní funkce, v lineárním čase převádějí z/na pole/sekvenci
ofArray : 'T [] -> 'T *list*
ofSeq : *seq*<'T> -> 'T *list*
toArray : 'T *list* -> 'T []
toSeq : 'T *list* -> *seq*<'T>

Fold a přátelé

Funkce fold a jí příbuzné jsou velmi užitečné a silné nástroje pro zpracování seznamů.
fold : ('State->'T->'State) -> 'State -> 'T *list* -> 'State
 Funkce fold funguje následovně. Dostane:
  binární operaci • : 'stav -> 'a -> 'stav
  hodnotu z :: 'stav
  seznam [x_1; x_2; ...; x_N] :: *list*<'a>
 a spočte hodnotu fold • z xs = ((..(z • x_1) • x_2) • ... • x_N) : 'stav.
 Jistým způsobem "složí" seznam do jedné hodnoty (proto fold). Tato hodnota se dobře počítá
 pomocí jednoho průchodu seznamu v O(N) (v podstatě pomocí **while** cyklu).

 Například fold ((+)) 0 spočte součet všech čísel v seznamu, a vrací 0 pro prázdný seznam.
 I komplikovanější funkce lze vyjádřit pomocí fold. Například iter f = fold (**fun** () x -> f x) ().

scan : ('State->'T->'State) -> 'State -> 'T *list* -> 'State *list*
 Funguje stejně jako fold, ale nevrací jednu hodnotu, nýbrž všechny dočasné výsledky.
 scan • z xs = [z; z • x_1; (z • x_1) • x_2; ...; fold • z xs].
 Opět je jednoduše spočitatelná v O(N), vyhodnocuje se stejně jako fold, jenom se uchovají mezivýsledky.

 Tedy scan ((+)) 0 vrací částečné součty daného seznamu.

reduce : ('T->'T->'T) -> 'T *list* -> 'T
 Jednodušší varianta funkce fold, která funguje jenom pro neprázdné seznamy.
 Dostane operaci • :: 'a -> 'a -> 'a a neprázdný seznam [x_1; x_2; ...; x_N] : *list*<'a>
 a vrátí reduce • xs = ((...(x_1 • x_2) • x_3) • ... • x_N) : 'a.

 Dala by se vyjádřit pomocí funkce fold například jako reduce • (x :: xs) = fold • x xs.

 Například funkce List.max xs = reduce max xs.

foldBack : ('T->'State->'State) -> 'T *list* -> 'State -> 'State
 Funkce fold zpracovává seznam "zleva doprava". Někdy je výhodnější zpracovat ho "zprava doleva".
 K tomu slouží funkce foldBack. Dostane
  binární operaci • : 'a -> 'stav -> 'stav
  seznam [x_1; x_2; ...; x_N] :: *list*<'a>
  hodnotu z :: 'stav
 a spočte hodnotu foldBack • xs z = x_1 • (x_2 • (x_3 • ... • (x_N • z))...) : 'stav.

Protože je `list<'a>` jednosměrný spojový seznam a `foldBack` zpracovává hodnoty pozpátku, potřebuje tato funkce ke svému výpočtu pomocnou paměť velikosti O(N) [buď se použije na rekurzi, nebo si seznam převedete do pole]. Pokud si můžete vybrat mezi `fold` a `foldBack`, použijte tedy `fold`.

`scanBack : ('T->'State->'State) -> 'T list -> 'State -> 'State list`

Funguje jako `foldBack`, ale vrací všechny vypočtené mezihodnoty.

`scanBack • xs z = [foldBack • xs z; ...; x_(N-1) • (x_N • z); x_N • z; z].`

Pořadí hodnot, které `scanBack` vrací, odpovídá způsobu výpočtu.

`reduceBack : ('T->'T->'T) -> 'T list -> 'T`

Stejně jako `reduce` je `reduceBack` jednoduchá varianta `foldBack`.

Dostane operaci `• :: 'a -> 'a -> 'a` a neprázdný seznam `[x_1; x_2; ...; x_N] : list<'a>`

a vrátí `reduceBack • xs = x_1 • (x_2 • ... • (x_(N-1) • x_N))...)`.

Také lze napsat `List.max xs = reduceBack max xs`, ale stejně jako u `foldBack` je vypočtení `reduceBack` náročnější na paměť než `reduce`.

`fold2 : ('State->'T1->'T2->'State) -> 'State -> 'T1 list -> 'T2 list -> 'State`

`foldBack2 : ('T1->'T2->'State->'State) -> 'T1 list -> 'T2 list -> 'State->'State`

Funkce `fold2` a `foldBack2` jsou varianty `fold` a `foldBack` pro dva seznamy.

Tyto seznamy se procházejí oba najednou a kombinují se vždy hodnoty na stejných indexech.

Alternativní možnost práce se seznamy

Kromě modulu `List` existuje ještě další možnost, jak pracovat se seznamy, a tou je samotný typ `list<'a>`.
Nabízí totiž několik základních statických a instančních metod. Tyto metody jsou všechny přítomny v modulu `List`, někdy ale může být pohodlnější je používat přímo na typu `list`:

static member `list.Cons : 'T * 'T list -> 'T list`	varianta `::`, ale bere dvojici	
static member `list.Empty : 'T list`	přejmenování `[]`	
member `this.Head : 'T`	`List.head`	
member `this.IsEmpty : bool`	`List.isEmpty`	
member `this.Item (int) : 'T`	`List.nth`	použití: seznam.[5]
member `this.Length : int`	`List.length`	O(N)
member `this.Tail : 'T list`	`List.tail`	

Funkce v modulu Seq

Typ `list<'a>` je zaměnitelný s typem `seq<'a>`, všechny funkce v modulu `Seq` lze tedy použít i na seznamy.

Modul `Seq` nabízí skoro všechny funkce jako modul `List`. Přesněji nabízí všechno kromě:

`map3`	Dalo by se jednoduše dodat.
`mapi2`	Dalo by se jednoduše dodat.
`iteri2`	Dalo by se jednoduše dodat.
`replicate`	Dalo by se jednoduše dodat.
`sortWith`	Dalo by se jednoduše dodat.
`unzip`	Dalo by se jednoduše dodat.
`unzip3`	Dalo by se jednoduše dodat.
`tail`	Nelze dodat se správnou složitostí, vysvětlení bude následovat.
`rev`	Nelze dodat se správnou složitostí, vysvětlení bude následovat.
`partition`	Nelze dodat se správnou složitostí, vysvětlení bude následovat.
`permute`	Nelze dodat se správnou složitostí, vysvětlení bude následovat.
`foldBack`	Nelze dodat se správnou složitostí, vysvětlení bude následovat.
`scanBack`	Nelze dodat se správnou složitostí, vysvětlení bude následovat.
`reduceBack`	Nelze dodat se správnou složitostí, vysvětlení bude následovat.

Je důležité si uvědomit, jak přesně `seq<'a>` funguje. Jak už bylo řečeno, je to jenom typové synonymum pro IEnumerable<'a>. Tato třída má jedinou metodu `GetEnumerator : () -> IEnumerator<'a>`.

Objekt typu `IEnumerator<'a>` slouží k projítí prvků sekvence. K tomu má položku `Current` a metodu `NoveNext : () -> bool`. Položka `Current` vrací aktuální prvek sekvence a metoda `MoveNext` přechází na další. Na začátku je iterátor před prvním prvkem a `Current` vrací `null`. Každé zavolání `MoveNext` vrací hodnotu typu `bool`, která indikuje, zda se iterátor nachází stále uvnitř sekvence. Pokud se jednou dostane iterátor za poslední prvek, `MoveNext` vrací **`false`** a `Current` vrací `null`. Pokud k tomu dojde, není možné iterátor vrátit opět na začátek, musí se vytvořit nový objekt typu `IEnumerator<'a>` pomocí `GetEnumerator`.

Pokud máte `s : seq<'a>`, můžete ji ručně projít avypsat prvky následujícím způsobem:

```
let vypis (s : seq<'a>) =
```

```
  let e = s.GetEnumerator()
  while e.MoveNext() do
    printf "%A\n" e.Current
```

Důležité je, že *seq* vrací nový iterátor na požádání. Výpočty iterátorů jsou obvykle na sobě nezávislé, takže pokud procházíte sekvenci dvakrát, vytvoříte dva iterátory a každý z nich provádí výpočet členů sekvence nezávisle na druhém.

Důsledek tohoto ale je, že není možné naimplementovat `tail` rozumně efektivně. Řekněme, že máte sekvenci `s`, tisíckrát použijete `tail` a vznikne sekvence `t`. Nyní `t` musí na požádání vracet iterátory. Každý tento iterátor lze získat jenom tak, že se vytvoří iterátor sekvence `s` a přeskočí se 1000 prvků. To ale musíte udělat pokaždé, když budete `t` procházet. Problém je v tom, že nemůžete udělat kopii iterátoru. Kdyby to šlo, `t` si zapamatuje iterátor `s`, který přeskočil 1000 prvků, a bude vracet jeho kopii. Protože to nejde, musí 1000 prvků přeskakovat při každém vytvoření iterátoru.

Díky této vlastnosti by se i ostatní funkce jako `rev`, `partition` a `scanBack` chovaly nečekaně (vytvoření iterátoru by mělo velkou složitost).

Pokud je pro vás problém, že se prvky sekvence vyhodnocují pokaždé znova, můžete sekvenci převést na seznam a nakonec vrátit tento seznam jako sekvenci. Potom se při iterování prochází již spočtené hodnoty seznamu –– místo čistokrevného výpočtu hodnot se vypočítají hodnoty jenom jednou a pak už se jenom procházejí.
S tím také souvisí funkce `Seq.cache`, viz dále.

Modul `Seq` nabízí navíc nějaké funkce, které v modulu `List` nejsou:
♣ Konverzní funkce pochopitelně fungují z/na seznam a pole.
```
ofArray : 'T array -> seq<'T>
ofList : 'T list -> seq<'T>
toArray : seq<'T> -> 'T []
toList : seq<'T> -> 'T list
```

♣ Funkce, které by do modulu `List` šly dodat:
```
countBy : ('T -> 'Key) -> seq<'T> -> seq<'Key * int>
```
O(N) v průměru pomocí hashování, vrátí unikátní seznam klíčů a počet jejich výskytu v původní sekvenci.
```
compareWith : ('T -> 'T -> int) -> seq<'T> -> seq<'T> -> int
```
Porovnání dvou sekvencí danou porovnávací funkcí.
```
distinct : seq<'T> -> seq<'T>
```
O(N) v průměru pomocí hashování, vrátí unikátní seznam prvků sekvence (porovnává pomocí =).
```
distinctBy : ('T -> 'Key) -> seq<'T> -> seq<'T>
```
O(N) v průměru pomocí hashování, vrátí unikátní seznam prvků sekvence (porovnává klíče pomocí =).
```
groupBy : ('T -> 'Key) -> seq<'T> -> seq<'Key * seq<'T>>
```
O(N) v průměru pomocí hashování, vrátí unikátní seznam klíč, pro každý z nich sekvenci prvků v původní sekvenci.
```
singleton : 'T -> seq<'T>
```
O(1), sekvence vracející jeden prvek.
```
take : (n:int) -> seq<'T> -> seq<'T>
```
O(1), vytvoří sekvenci, která vrací n prvních prvků dané sekvence. Pokud jich tolik není, vyhodí výjimku.
```
takeWhile : (p:'T->bool) -> seq<'T> -> seq<'T>
```
O(1), vytvoří sekvenci, která vrací prvky dané sekvence, dokud splňují predikát `p`
```
truncate : int -> seq<'T> -> seq<'T>
```
O(1), jako `take`, ale pokud není prvků dost, vrátí jich méně než požadovaný počet.
```
skip : (n:int) -> seq<'T> -> seq<'T>
```
O(1) na vytvoření výsledku, O(n) na vytvoření iterátoru výsledné sekvence.
Vytvoří sekvenci, která vrací prvky původní sekvence až na l prvních.
```
skipWhile : ('T -> bool) -> seq<'T> -> seq<'T>
```
O(1) na vytvoření výsledku, O(n) na vytvoření iterátoru výsledné sekvence.
Vytvoří sekvenci, která vrací prvky původní sekvence až na úsek prvních prvků splňujících predikát `p`.
```
pairwise : seq<'T> -> seq<'T * 'T>
```
O(1), ze sekvence prvků x_1, x_2, ... vytvoří sekvenci dvojic (x_1, x_2), (x_2, x_3), ...
```
windowed : (n:int) -> seq<'T> -> seq<'T []>
```
O(1), O(n) na vytvoření každého prvku sekvence.
Z dané sekvence vytvoří sekvenci polí délky n. První prvek sekvence obsahuje prvních n prvků dané sekvence, druhý prvek obsahuje 2.–(n+1). prvek původní sekvence a tak dál.
Funkce `pairwise` je skoro `windowed 2`, jenom vrací dvojici místo pole.

♣ Funkce pro nekonečné sekvence, už z minulé lekce
```
initInfinite : (int -> 'T) -> seq<'T>
unfold : ('State -> 'T * 'State option) -> 'State -> seq<'T>
```

♣ Funkce specifické pro *seq*:
`cache : seq<'T> -> seq<'T>`
Vytvoří novou sekvenci, která vrací stejné prvky jako původní. Rozdíl je v tom, že nová sekvence si vypočtené prvky pamatuje. Tedy každý prvek původní sekvence se vyhodnotí nanejvýš jednou. Funguje to podobně jako `|> List.ofSeq |> List.toSeq`, ale s tím rozdílem, že se prvky vyhodnotí až když jsou poprvé potřeba, takže `Seq.cache` funguje i na nekonečné posloupnosti (řešení s `List.ofSeq` by se zacyklilo.

`delay : (unit -> seq<'T>) -> seq<'T>`
Vytvoří sekvenci, která se vyhodnotí až při prvním zavolání `GetEnumerator()`.
Volání `(Seq.ofList [1..10])` a `Seq.delay (Seq.ofList [1..10])` se liší v tom, že první vyhodnotí seznam už při vytváření *seq*, druhé až při prvním procházení vytvořené sekvence.

`readonly : seq<'T> -> seq<'T>`
Vrátí sekvenci, která prochází první a vrací přesně to, co ona. Rozdíl je v tom, že vytvořený objekt je čistokrevná sekvence. Třeba pole nebo seznam jde přetypovat na `seq<'a>` a někdo by si toho mohl všimnout a přetypovat to zpátky. Pokud vytvoříte sekvenci pomocí `readonly`, tak vznikne nový objekt, který nejde přetypovat na nic a nejde z něj poznat, z čeho sekvence vznika.

`append : seq<'T> -> seq<'T> -> seq<'T>`
Tato funkce je jako `List.append`, jenom má jinou časovou složitost. Jedno volání `append` je stejné jako `List.append`, čili má složitost O(délka_první_sekvence) –– při iterování výsledku je třeba kontrolovat, zda první sekvence nedošla, čili O(1) na každý prvek první sekvence.

Nicméně pokud dojde k volání mnoha `appendů`, jsou vždy přezávorkovány doprava –– tedy z append (append (append (... [1]) [2]) ... [N], které by se vyhodnocovalo O(N^2), samo vznikne append [1] (append [2] (... append [N–1] [N])...), které se vyhodnotí v O(N).

Rozšíření list–comprehention syntaxe pro sekvence

Sekvence se dají definovat pomocí `seq { for a in [1..10] do yield a }`. Tato syntaxe umožňuje ještě několik dalších operací:

♣ `yield!`
`seq { yield! [1..10] }`
Pomocí `yield!` je možné vracet celou sekvenci, ne jenom jeden prvek. Je to vlastně varianta @appendu@.
Stejně jako u `appendu` dochází k přezávorkování, takže `yield!` nezvyšuje časovou složitost.

Následný příklad používá líného vyhodnocování sekvencí:
```
let rec fib =
  seq { yield 1I
        yield 1I
        yield! fib |> Seq.pairwise |> Seq.map (fun (x, y) -> x + y)
      }
```
Nicméně něco je v nepořádku: pro vyhodnocení `fib` se použije interní iterátor pro procházení (za `yield!`). Ale tento interní iterátor si také vytvoří interní iterátor a tak dál. Když tedy vyhodnotíte N–tý člen `fib`, bude existovat (N–1) iterátorů, které nesdílejí výsledky –– tedy to bude mít složitost O(N^2).

Správné řešení je způsobit, aby se hodnoty sekvence vyhodnocovaly maximálně jednou a pak se zapamatovaly –– –– k tomu přesně slouží `Seq.cache`:
```
let rec fib =
  seq { yield 1I
        yield 1I
        yield! fib |> Seq.pairwise |> Seq.map (fun (x, y) -> x + y)
      } |> Seq.cache
```

Samozřejmě v tomto konkrétním případě je asi jednodušší použít `unfold`:
```
let fib2 =
  let next (a, b) = Some (b, (b, a+b))
  Seq.unfold next (0I, 1I)
```

♣ **`let`** a **`while`**
Uvnitř definice *seq* lze používat **`let`** i **`while`**, takže můžete dané sekvence procházet ručně:
```
let map f (s:seq<'a>) =
  seq { let e = s.GetEnumerator()
        while e.MoveNext() do
        yield f (e.Current)
      }
```

Moduly Set a Map

Kromě sekvence a seznamu poskytuje F# několik dalších datových struktur. Dvě z nich jsou `Set` a `Map`.

`Set<'a>` slouží k uložení množiny (seznam jedinečných hodnot typu `'a`) a `Map<'key, 'value>` slouží k uložení zobrazení čili asociativního pole (každý přítomný jedinečný klíč typu `'key` má přiřazenu hodnotu typu `'value`).

Obě tyto struktury jsou immutable, neboli neměnitelné. Pokud chcete udělat změnu (přidat prvek), vytvoří se nová kopie struktury, stará obsahuje stále tytéž hodnoty (stejně jako se seznamy). Nicméně jak `Set` tak `Map` jsou reprezentovány pomocí vyvážených stromů, takže mnoho operací má složitost jenom O(log N).

Stejně jako u seznamů lze s množinami a zobrazeními pracovat buď pomocí funkcí v modulu `Set` a `Map`, nebo pomocí funkcí objektů `Set` a `Map`.

Jednotlivé funkce nebudu podrobně komentovat, ze jména a typu lze odvodit, co dělá. Vypisuji pouze složitosti.

♣ Funkce modulu `Set`:

```
add : 'T -> Set<'T> -> Set<'T>                                        O(log N)
contains : 'T -> Set<'T> -> bool                                      O(log N)
count : Set<'T> -> int                                                O(N)
difference, (-) : Set<'T> -> Set<'T> -> Set<'T>                       O(N log N)
empty : Set<'T>                                                       O(1)
exists : ('T -> bool) -> Set<'T> -> bool                              O(N)
filter : ('T -> bool) -> Set<'T> -> Set<'T>                           O(N log N)
fold : ('State -> 'T -> 'State) -> 'State -> Set<'T> -> 'State        O(N)
foldBack : ('T -> 'State -> 'State) -> Set<'T> -> 'State -> 'State    O(N)
forall : ('T -> bool) -> Set<'T> -> bool                              O(N)
intersect : Set<'T> -> Set<'T> -> Set<'T>                             O(N log N)
intersectMany : seq<Set<'T>> -> Set<'T>                           reduce intersect
isEmpty : Set<'T> -> bool                                             O(1)
isProperSubset : Set<'T> -> Set<'T> -> bool                           O(N log N)
isProperSuperset : Set<'T> -> Set<'T> -> bool                         O(N log N)
isSubset : Set<'T> -> Set<'T> -> bool                                 O(N log N)
isSuperset : Set<'T> -> Set<'T> -> bool                               O(N log N)
iter : ('T -> unit) -> Set<'T> -> unit                                O(N)
map : ('T -> 'U) -> Set<'T> -> Set<'U>                                O(N log N)
maxElement : Set<'T> -> 'T                                            O(log N)
minElement : Set<'T> -> 'T                                            O(log N)
ofArray : 'T array -> Set<'T>                                         O (N log N)
ofList : 'T list -> Set<'T>                                           O (N log N)
ofSeq : seq<'T> -> Set<'T>                                            O (N log N)
partition : ('T -> bool) -> Set<'T> -> Set<'T> * Set<'T>              O (N log N)
remove : 'T -> Set<'T> -> Set<'T>                                     O(log N)
singleton : 'T -> Set<'T>                                             O(1)
toArray : Set<'T> -> 'T array                                         O(N)
toList: Set<'T> -> 'T list                                            O(N)
toSeq : Set<'T> -> seq<'T>                                            O(1)
union, (+): Set<'T> -> Set<'T> -> Set<'T>                             O(N log N)
unionMany : seq<Set<'T>> -> Set<'T>                               fold (+) Set.empty
```

♣ Funkce objektu `Set<'a>`:

```
new Set : seq<'T> -> Set<'T>
member this.Add : 'T -> Set<'T>
member this.Contains : 'T -> bool
member this.IsProperSubsetOf : Set<'T> -> bool
member this.IsProperSupersetOf : Set<'T> -> bool
member this.IsSubsetOf : Set<'T> -> bool
member this.IsSupersetOf : Set<'T> -> bool
member this.Remove : 'T -> Set<'T>
member this.Count :  int
member this.IsEmpty :  bool
member this.MaximumElement :  'T
member this.MinimumElement :  'T
static member ( + ) : Set<'T> * Set<'T> -> Set<'T>
static member ( - ) : Set<'T> * Set<'T> -> Set<'T>
```

K vytvoření `Set<'a>` lze použít několik konstrukcí:

- ♣ `Set.ofList`, `Set.ofSeq`
- ♣ funkci `set :: seq<'a> -> Set<'a>`, která je jenom přejmenováním `Set.ofSeq`
- ♣ **`new`** `Set<_>([1..10])`

♣ Funkce modulu `Map`:

`add : 'Key -> 'T -> Map<'Key,'T> -> Map<'Key,'T>`	O(log N)
`containsKey : 'Key -> Map<'Key,'T> -> bool`	O(log N)
`empty : Map<'Key,'T>`	O(1)
`exists : ('Key -> 'T -> bool) -> Map<'Key,'T> -> bool`	O(N)
`filter : ('Key -> 'T -> bool) -> Map<'Key,'T> -> Map<'Key,'T>`	O(N log N)
`find : 'Key -> Map<'Key,'T> -> 'T`	O(log N)
`findKey : ('Key -> 'T -> bool) -> Map<'Key,'T> -> 'Key`	O(N)
`fold : ('State->'Key->'T->'State) -> 'State -> Map<'Key,'T> -> 'State`	O(N)
`foldBack : ('Key->'T->'State->'State) -> Map<'Key,'T> ->'State->'State`	O(N)
`forall : ('Key -> 'T -> bool) -> Map<'Key,'T> -> bool`	O(N)
`isEmpty : Map<'Key,'T> -> bool`	O(1)
`iter : ('Key -> 'T -> unit) -> Map<'Key,'T> -> unit`	O(N)
`map : ('Key -> 'T -> 'U) -> Map<'Key,'T> -> Map<'Key,'U>`	O(N log N)
`ofArray : ('Key * 'T) [] -> Map<'Key,'T>`	O(N log N)
`ofList : 'Key * 'T list -> Map<'Key,'T>`	O(N log N)
`ofSeq : seq<'Key * 'T> -> Map<'Key,'T>`	O(N log N)
`partition : ('Key->'T->bool)->Map<'Key,'T>->Map<'Key,'T>*Map<'Key,'T>`	O(N log N)
`pick : ('Key -> 'T -> 'U option) -> Map<'Key,'T> -> 'U`	O(N log N)
`remove : 'Key -> Map<'Key,'T> -> Map<'Key,'T>`	O(log N)
`toArray : Map<'Key,'T> -> ('Key * 'T) []`	O(N)
`toList : Map<'Key,'T> -> ('Key * 'T) list`	O(N)
`toSeq : Map<'Key,'T> -> seq<'Key * 'T>`	O(1)
`tryFind : 'Key -> Map<'Key,'T> -> 'T option`	O(log N)
`tryFindKey : ('Key -> 'T -> bool) -> Map<'Key,'T> -> 'Key option`	O(N)
`tryPick : ('Key -> 'T -> 'U option) -> Map<'Key,'T> -> 'U option`	O(N)

♣ Funkce objektu `Map<'k, 'v>`:

```
new Map : seq<'Key * 'Value> -> Map< 'Key, 'Value>
member this.Add : 'Key * 'Value -> Map<'Key, 'Value>
member this.ContainsKey : 'Key -> bool
member this.Remove : 'Key -> Map<'Key, 'Value>
member this.TryFind : 'Key -> 'Value option
member this.Count :  int
member this.IsEmpty :  bool
member this.Item ('Key) : 'Value
```
Tedy lze volat `m.["a"]`

K vytvoření `Map` obdobně jako k vytvoření `Set` použít funkci `map`, není definovaná.
Nicméně existuje funkce `dict`, která vytváří neměnitelný objekt typu `IDictionary<'k,'v>`, tj.
`dict : seq<'key*'value> -> System.Collections.Generic.IDictionary<'key, 'value'>`
.
`IDictionary` nabízí především metody:

```
  member this.Count :  int
  member this.IsEmpty :  bool
  member this.ContainsKey : 'key -> bool
  member this.Item ('key) : 'value
  member this.Keys : seq<'key>
  member this.Values : seq<'key>
```
Tedy lze volat `d.["a"]`

Výpisy a operátor středníku

K výpisu je nejběžnější použít funkci `printf` nebo `printfn`. Funkce `printfn` funguje stejně jako `printf`, jenom po výpisu ještě odřádkuje (tj. nakonec vypíše "\n").

Typ těchto funkcí je `printf : skoro_string -> typy_určené_forátovacím_řetězcem -> unit`.

Tedy sám kompilátor rozumí formátovacímu řetězci a otypuje funkci podle něj –– speciálně řetězec musí být znám již při kompilaci, **`let`** `s = "%d"` **`in`** `printf s` neprojde.

Formátovací řetězec je velmi podobný céčkovému. Tedy:

`%b`	vypíše *bool*
`%s`	vypíše *string*
`%d, %i`	vypíše jakýkoliv typ *byte*, *int16*, *int32*, *int64*, *sbyte*, *uint16*, @uint32,uint64,nativeint,unativeint `jako by byl znaménkový*)`
`%u`	vypíše jakýkoliv typ *byte*, *int16*, *int32*, *int64*, *sbyte*, *uint16*, @uint32,uint64,nativeint,unativeint `jako by byl bezznaménkový*)`
`%x`	jako u, ale vypisuje hexadecimálně s ciframi 0..9, a..f
`%X`	jako u, ale vypisuje hexadecimálně s ciframi 0..9, A..F

`%o`	jako u, ale vypisuje v osmičkové soustavě
`%e, %E`	vypíše *`float`*`,` *`float32`* ve formátu d.d···de[+−]ddd
`%f`	vypíše *`float`*`,` *`float32`* jako d···d.d···d
`%g, %G`	vypíše *`float`*`,` *`float32`* v nejkratší variantě
`%M`	vypíše *`decimal`*
`%O`	vypíše hodnotu libovolného typu pomocí funkce `ToString()`
`%A`	vypíše hodnotu libovolného typu jak to dělá fsi (F# hodnoty speciálně, ostatní pomocí `ToString()`)

Kromě `printf` a `printfn` existuje mnoho dalších variant, například:

```
  sprintf : formát -> argumenty -> string
  bprintf : StringBuilder -> formát -> argumenty -> unit
  eprintf : formát -> argumenty -> unit
  failwithf :: formát -> argumenty -> 'a
```

(`eprintf` – vypisující na chybový výstup; `failwithf` – které vytvoří výjimku zformátováním řetězce)

Nyní pokud chceme něco vypisovat a ještě počítat výsledky, potřebujeme k tomu nějakou kombinační funkci. Takovou funkcí je operátor středníku, tj. `;`. Tento operátor má typ `(;) ::` *`unit`* `-> 'a -> 'a`. Dostane tedy nějakou akci, tu provede, a vrátí svůj druhý argument.

Tedy `printfn "%d" a; printfn "%d" b; a, b` vypíše `a`, pak `b` a vrátí dvojici `(a, b)`.

Navíc středníky se automaticky doplňují, pokud následující řádek začíná ve stejném sloupci jako předchozí. Tedy:

```
main = printf "%s" "Hello"
       printf "%O" ' '
       printf "world!"
```

nebo

```
main =
  printf "%s" "Hello"
  printf "%O" ' '
  printf "world!"
```

Automatické testování

Kód je třeba často testovat. Lze se například v každém modulu pomocí podmíněné kompilace vytvořit `main` funkci, která testování povede. Nebo je možné použít nějaký testovací framework.

Jeden ze zajímavých testovacích nástrojů je `QuickCheck`, původně vyvinutý pro Haskell, ale nyní dostupný v mnoha jazycích, včetně F#. Ten se jmenuje FsCheck, poslední verze je 0.7 (starší nepoužívejte).

Pokud chcete, můžete mi odevzdávat úkoly používající `FsCheck`.

Pro použití si stáhněte `FsCheck` assembly. Při kompilaci musíte použít přepínač `-r FsCheck.dll`, abyste ji mohli používat. Pokud dodáte navíc i `--staticlink FsCheck`, zakompiluje se do výsledné binárky a nebude ji třeba při spuštění. Pokud chcete používat svůj program ve `fsi.exe`, musíte mu opět dát přepínač `-r FsCheck.dll` nebo do zdrojového textu programu napište

```
#if INTERACTIVE
#r "FsCheck.dll"
#endif
```

Podmíněný překlad je použit proto, že `#r` není možné použít v souboru překládaném `fsc`.

Všechny funkce jsou v namespace `FsCheck`, takže je buď musíte plně kvalifikovat nebo do souboru uvést **`open`** `FsCheck` –– od této řádky níže se použité funkce hledají i ve jmenném prostoru `FsCheck`. Ten musí samozřejmě v tu dobu již existovat.

Výhoda `FsChecku` je, že si umí generovat vlastní data. Stačí napsat funkci vracející *`bool`* nebo `Property` (viz dále) a `FsCheck` pro ni bude generovat náhodná vstupní data. Ve výchozím nastavení otestuje 100 náhodných vstupů na test.

Vytvořme test, který ověřuje, že dvakrát zavolaný `List.rev` vrátí původní seznam:

```
let rev_test (xs : list<int>) = xs = (xs |> List.rev |> List.rev)
```

Všimněte si, že jsme museli říct, že `xs` je list *`int`*ů, aby `FsCheck` věděl, jaká generovat data.

Nyní můžeme spustit test voláním

```
Check.Quick rev_test
```

Pokud dojde k chybě, `FsCheck` vypíše vstup, na kterém k ní dojde, s tím, že se snaží najít vstup rozumně malý. Test můžete i pojmenovat pomocí `Check.Quick("jméno", metoda)`.

Nyní větší a kompletní příklad:

```
#if TESTING
open FsCheck
type Tests =
  static member sum_as_fold (xs : list<int>) =
    List.sum xs = List.fold ((+)) 0 xs
```

```
  static member ``reverse of reverse is original`` (xs : list<int>) =
    xs = (xs |> List.rev |> List.rev)

Check.QuickAll<Tests>()
#endif
```

Všimněte si, že
♣ používáme podmíněného překladu. Testování se provede jenom, pokud kompilujeme s `-d TESTING` (a nezapomeňte `-r FsCheck.dll`)
♣ spouštíme všechny testy definované v `Tests`
♣ pomocí dvojitých zpětných apostrofů mohou být ve jménách metod i mezery a jiné podivné znaky

Ještě poslední věc. Kromě typu *bool* může test vracet ještě `FsCheck.Property`. To je typ `FsCheck`, který umožňuje více kontrolovat vstupy testu. Nejjednodušší použití je následující:

```
  static member max_as_reduce (xs : list<int>) =
    List.max xs = List.reduce max xs
```

Tento test neprojde, protože nefunguje na prázdném seznamu –– dojde k vyhození výjimky.

Je několik možností, jak to opravit:
♣ Test bude `xs.IsEmpty || List.max xs = List.reduce max xs`. Nevýhoda je, že se prázdné vstupy počítají do 100 vstupů.
♣ Zařídit, aby vstup byl vždy neprázdný pomocí

```
  static member max_as_reduce x (xs : list<int>) =
    List.max (x::xs) = List.reduce max (x::xs)
```

♣ Použití operátoru ==>, který je z prostoru `FsCheck.Prop`. Tento operátor má typ `(==>) :: bool -> 'Testable -> Property`. Je to jakási implikace –– test se provádí jenom tehdy, když je podmínka splněná. Pokud není, generují se nová vstupní data. Celkem se vyzkouší nejvýš 1000 vstupních dat (ve výchozím stavu), takže by to mělo mít velkou šanci uspět.
Tedy:

```
  static member max_as_reduce (xs : list<int>) =
    not xs.IsEmpty ==> lazy (List.max xs = List.reduce max xs)
```

Všimněte si slova **lazy** –– to je velmi důležité. Kdyby tam nebylo, tak se kvůli striktnímu vyhodnocování vyhodnotí pravá strana ==> ještě před vyhodnocením samotného operátoru ==>, tedy výjimku to stále vyhodí. Slovo **lazy** tomu zabrání (časem zjistíme, co přesně se při tom děje).

Pokud chcete generovat nějaké zvláštní hodnoty, můžete použít následující předdefinované:

```
  static member t (PositiveInt pos) (NonZeroInt nenul) (NonNegativeInt nezap)
  static member u (StringNoNulls nenul) (NonEmptyString nepr) (NonEmptySet neps)
  static member v (NonEmptyArray nepr) (Interval mensi vetsi)
```

`FsCheck` umí i generovat náhodné funkce, takže můžete napsat:

```
  static member ``filter as partition`` p (xs : list<int>) =
    List.filter p xs = fst (List.partition p xs)
```

Nicméně pokud se bude funkce vypisovat, vypíše se jako `<fun:Invoke@2927>`, což není příliš užitečné.
Proto umí F# generovat funkce i "s obsahem", takto:

```
  static member ``filter as partition`` (F (_, p)) (xs : list<int>) =
    List.filter p xs = fst (List.partition p xs)
```

Hodnota `F (content, fun) : Function<'a, 'b>` generuje náhodnou funkci typu `'a -> 'b`, která se dá vypsat (a `content` obsahuje mapování, které `p` provádí. Většinou ale na něj není třeba sahat ručně.

Všechno generování funguje rekurzivně, takze není problém vygenerovat si
`m : Map<PositiveInt, Function<int, Set<NonEmptyString>>>` :–)

`FsCheck` umí ještě další zajímavé věci (jako kontrolovat, že testy doběhnou do dané doby, že test vyhodí výjimku, používat uživatelské generátory typů jako `Tree<'a>`), kdyžtak se podívejte do tutoriálu na webu.