IB015 Neimperativní programování

Ukázky funkcionálně řešených problémů

Jiří Barnat
Libor Škarvada

Užitečné konstrukce Haskellu

Aplikační operátor $

Pozorování

- Vnořené aplikace funkcí mohou být díky závorkování nečitelné.
- Některým uzávorkováním se lze vyhnout použitím aplikačního operátoru `$` , který má při vyhodnocování nejnižší prioritu.

Aplikační operátor $

- ```
  ($) :: (a -> b) -> a -> b
  ($) f x = f x
  ```
- `f(g x)` ≡ `($) f (g x)` ≡ `f $ (g x)` ≡ `f $ g x`
- `f(g(h x))` ≡ `f $ g $ h x`
  ```

Aplikační operátor $ – příklady

Ekvivalentní zápisy pomocí operátoru $

- ```
  not ( not ( not ( not ( not True ))))
  not $ not $ not $ not $ not True
  ```

- ```
  (+1) ((+2) ((+3) ((+4) 0)))
  (+1) $ (+2) $ (+3) $ (+4) 0
  ```

- ```
  (even ((+1) 0)) || (odd ((+2) 0))
  (even $ (+1) 0) || (odd $ (+2) 0)
  ```

Příklady

- `(++) [1] $ map (+1) [1]` $\rightsquigarrow^*$ `[1,2]`
- `[1] ++ $ map (+1) [1]` $\rightsquigarrow^*$ **ERROR**
- Rozsah platnost `$` je podřízen syntaktickému pravidlu o zarovnání, tj. v `do` notaci "končí s koncem řádku".
  ```

Pojmenovaný vzor – symbol @

Použití symbolu @

- Pokud `vzor` je korektně vytvořený datový vzor, pak zápisem `jmeno@vzor` získáme proměnnou `jmeno`, která bude po úspěšném použití vzoru odkazovat na celý mapovaný obsah.
- Nejčastěji používané ve spojení se seznamy, ale funguje všeobecně pro jakékoliv hodnoty konstruované s využitím datových konstruktorů.

Příklady

- Při mapování seznamu `[1,2,3]` na vzor `a@(x:t)`, bude

  `a = [1,2,3],  x = 1,  t = [2,3]` .

- `f (a@(x:y)) = x:a++y`

  `f [1,2,3]` $\rightsquigarrow^*$ `[1,1,2,3,2,3]`

  `f []` $\rightsquigarrow^*$ **ERROR**

Syntaktická konstrukce case

Pozorování

- Víceřádkové definice funkcí realizují větvení kódu.
- Více řádkovou definici lze ekvivalentně přepsat s využitím klíčového slova `case` .

Syntaktická konstrukce `case`

- ```
  case expression of
      pattern_1 -> expression_1
      ...
      pattern_n -> expression_n
  ```
- Textové zarovnání vzorů je nutné.
- Všechny výrazy na pravých stranách musí být stejného typu.
  ```

Syntaktická konstrukce case – příklady

Úkol 1

- Definujte funkci `take` s využitím konstrukce `case`.
- Řešení:

```
take m ys = case (m,ys) of
   (0,_) -> []
   (_,[]) -> []
   (n,x:xs) -> x : take (n-1) xs
```

Úkol 2

- Zapište pomocí `case` výraz `if e1 then e2 else e3`.
- Řešení:

```
case (e1) of True -> e2
             False -> e3
```

Strážené definice funkcí

Stráž

- Stráž je výraz typu Bool přidružený k definičnímu přiřazení.
- Při výpočtu bude tato definice funkce realizována, pouze pokud bude asociovaný výraz vyhodnocen na `True` .
- ```
  function args
        | guard_1 = expression_1
  ...
        | guard_n = expression_n
        | otherwise = default expression
  ```

Pozorování

- Konstrukce nerozšiřuje výrazové možnosti jazyka, ale je pohodlná (syntaktický cukr).
  ```

Strážené definice funkcí – příklady

Příklad 1

- ```
  f x | (x>3) = "vetsi nez 3"
      | (x>2) = "vetsi nez 2"
      | (x>1) = "vetsi nez 1"
  ```
- `f 2` $\rightsquigarrow^*$ `"vetsi nez 1"`

  `f 1` $\rightsquigarrow^*$ **ERROR**

Příklad 2

- ```
  g (a:x)
    | x==[]         = "Almost empty."
    | x/=[]         = "At least 2 members."
    | otherwise     = "Unreachable code."
  g _               = "Nothingness."
  ```
- `g []` $\rightsquigarrow^*$ `"Nothingness."`

  `g "Ahoj"` $\rightsquigarrow^*$ `"At least 2 members."`

Použití akumulátoru

Akumulátor


TAB
traction flooded
Tubular
Motion
12V 95T, 130Ah (C120), 115Ah (C20), 104Ah (C10), 95Ah (C5)

Akumulátor a rekurze

Obecný princip

- Vícenásobné nebo nevhodné použití rekurzivního volání v těle rekurzivně definované funkce, může v obecné rovině vést na časově neoptimální algoritmus.
- Opakovanému rekurzivnímu volání pro tutéž hodnotu lze zabránit uchováváním mezivýsledků rekurzivního volání.
- Uchování výsledků se provádí přidáním parametru rekurzivní funkce, tzv. **akumulátoru**.

Pozorování

- Přímé použití rekurzivní funkce má tendenci být čitelnější.

Akumulátor a rekurze - připomenutí

Pozorování

- Neefektivita opakovaným voláním rekurzivní funkce

Připoměňte si

- ```
  fib' 0 = 0
  fib' 1 = 1
  fib' x = fib' (x-2) + fib' (x-1)
  ```
- ```
  fib x = f 0 1 x
    where f a _ 0 = a
          f a b k = f b (a+b) (k-1)
  ```

Akumulátor a rekurze - připomenutí

Pozorování

- Neefektivita přímým použitím rekurzivní funkce.

Připomeňte si

- ```
  reverse' :: [a] -> [a]
  reverse' [] = []
  reverse' (x:s) = reverse' s ++ [x]
  ```
- ```
  reverse :: [a] -> [a]
  reverse z = rev [] z
              where rev s [] = s
                    rev s (x:t) = rev (x:s) t
  ```

Prohledávání a prořezávání

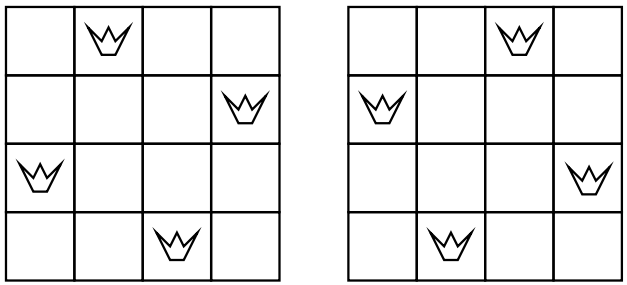
Problém n dam

Problém n dam

Problém n dam

- Rozestavit n šachových dam na pole čtvercové šachovnice $n \times n$ tak, aby se žádné dvě dámy navzájem neohrožovaly.

Všechna řešení pro $n = 4$

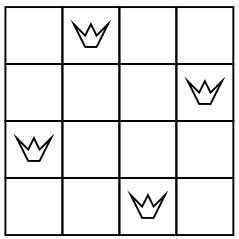
- 

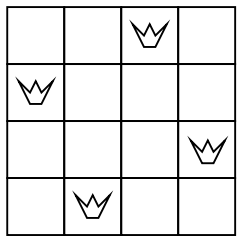
Pozorování

- V každém řádku/sloupku šachovnice může být nejvýše jedna dáma (jinak se ohrožují), a zároveň musí být alespoň jedna dáma, jinak bychom jich neumístili n.

Problém n dam – kodování pozice

Kódování pozice

- Očíslujme 1 až n řádky šachovnici směrem seshora dolů a sloupce šachovnice směrem zleva doprava.
- Řešení budeme kódovat seznamy čísel a to tak, že čísla v seznamu určují pozice dam v odpovídajících sloupcích.
-  `[3,1,4,2]`

   `[2,4,1,3]`

Úkol

- Naprogramujte funkci `damy :: Int -> [[Int]]` , která pro zadané n vrátí seznam všech možných řešení.

Problém n dam – myšlenka řešení

Postupné rozšiřování šachovnice

- Zavedeme funkci `da m n :: Int -> Int -> [[Int]]` , která bude počítat všechna řešení pro obdélníkovou matici s m sloupky a n řádky ($m < n$).
- Nová funkce bude řešení počítat rekurzivně, a to s využitím řešení pro šachovnici s $(m - 1)$ sloupky a n řádky.
- Šachovnice s nula sloupky, má jedno triviální řešení: `[]` , tj. `da 0 n = [[]]` .

Cesta k celkovému řešení

- ```
  damy :: Int -> [[Int]]
  damy n = da n n
  ```
  ```

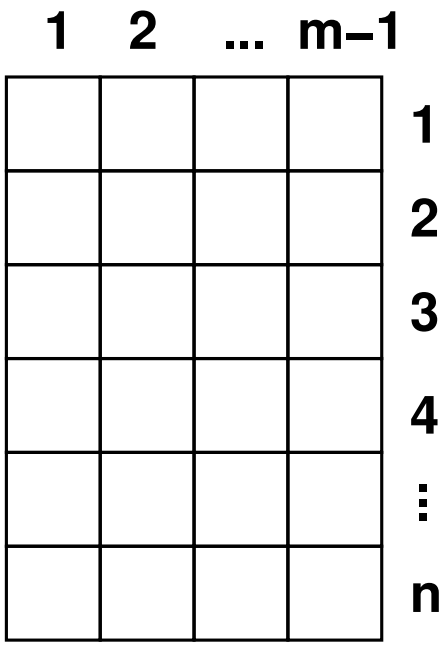
Problém *n* dam – pomocná funkce

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- ```
  hrozba :: [Int] -> [Int]
  hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]
  ```

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
-


  ```

Problém *n* dam – pomocná funkce

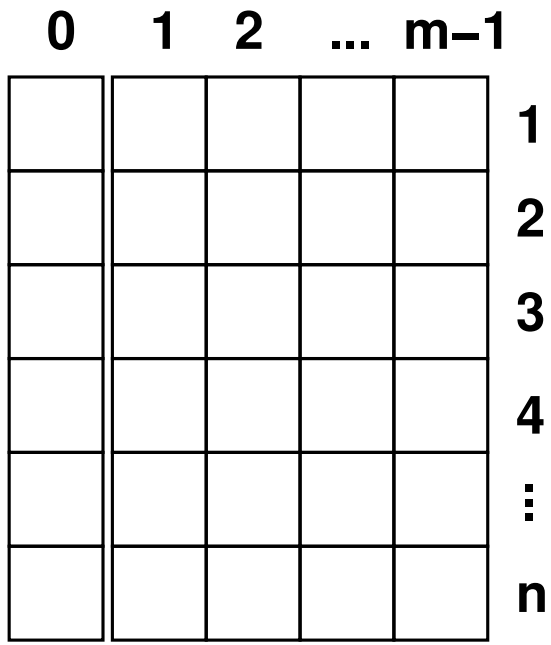
Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- `hrozba :: [Int] -> [Int]`

  `hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]`

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- `hrozba` vrátí seznam ohrožených pozic.



Problém *n* dam – pomocná funkce

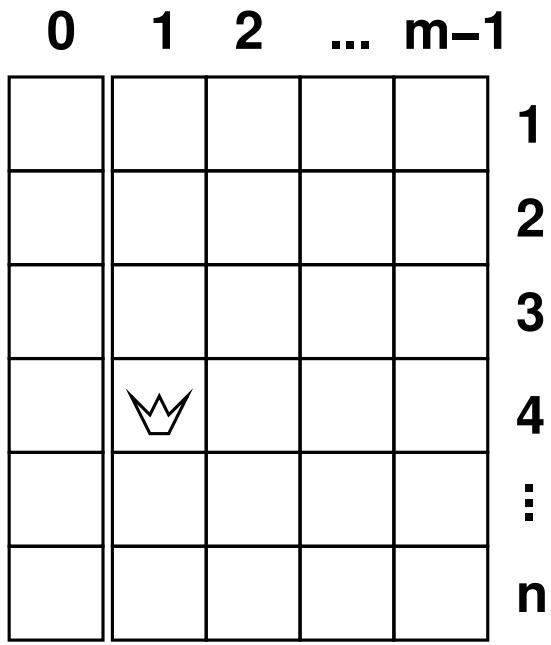
Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- `hrozba :: [Int] -> [Int]`

```
hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]
```

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- `[4, , , ]` `[4, , , ]` `[4, , , ]`



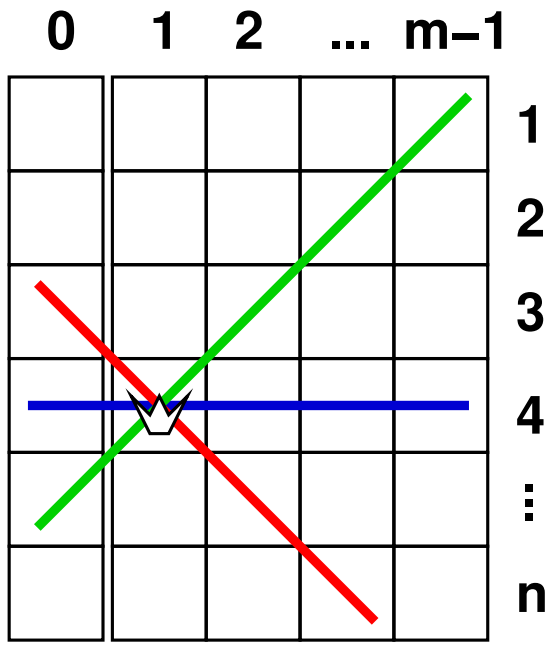
Problém *n* dam – pomocná funkce

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- ```
  hrozba :: [Int] -> [Int]
  hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]
  ```

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- ```
  [4, , , ]   [4, , , ]   [4, , , ]
                +[1,2,3,4] -[1,2,3,4]
  ---------------
  [4, , , ]   [5, , , ]   [3, , , ]
  ```



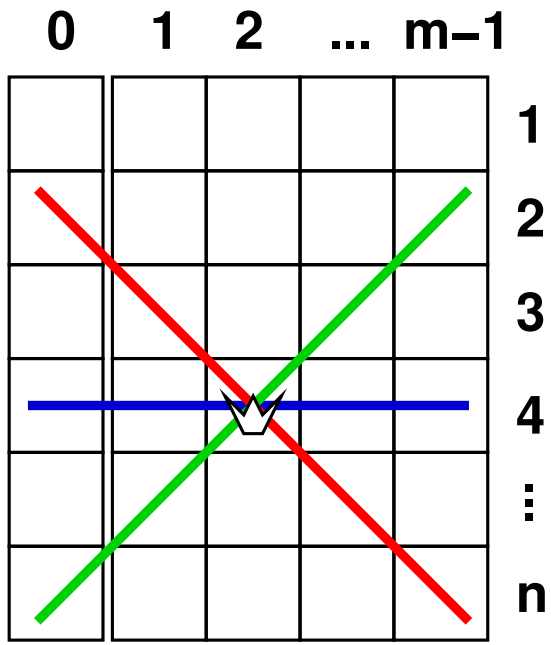
Problém *n* dam – pomocná funkce

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- ```
  hrozba :: [Int] -> [Int]
  hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]
  ```

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- ```
  [ ,4, , ]   [ ,4, , ]   [ ,4, , ]
                +[1,2,3,4] -[1,2,3,4]
  ---------------
  [ ,4, , ]   [ ,6, , ]   [ ,2, , ]
  ```



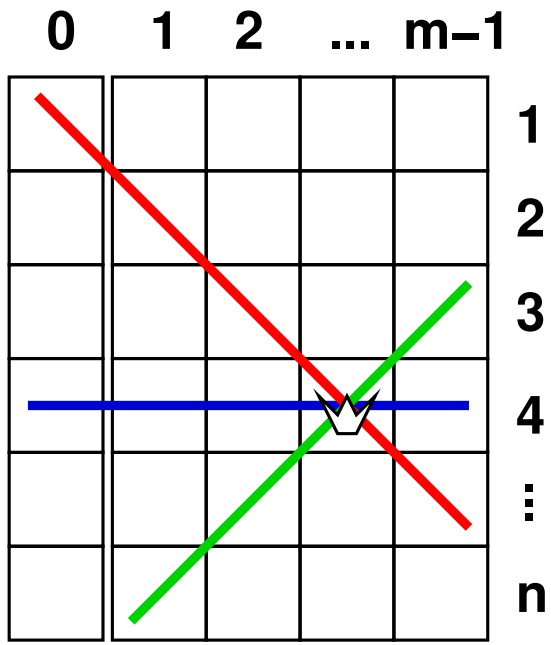
Problém *n* dam – pomocná funkce

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- ```
  hrozba :: [Int] -> [Int]
  hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]
  ```

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- ```
  [ , ,4, ]  [ , ,4, ]  [ , ,4, ]
             +[1,2,3,4] -[1,2,3,4]
  ---------------
  [ , ,4, ]  [ , ,7, ]  [ , ,1, ]
  ```



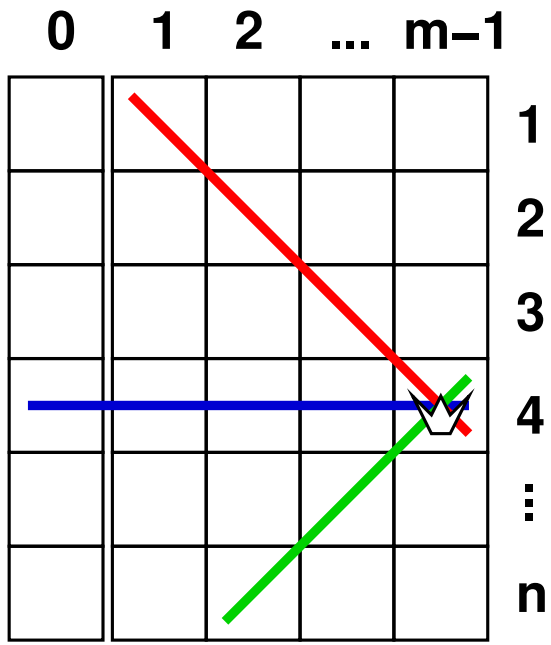
Problém *n* dam – pomocná funkce

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- ```
  hrozba :: [Int] -> [Int]
  hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]
  ```

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- ```
  [ , , ,4]   [ , , ,4]   [ , , ,4]
              +[1,2,3,4] -[1,2,3,4]
  ---------------
  [ , , ,4]   [ , , ,8]   [ , , ,0]
  ```



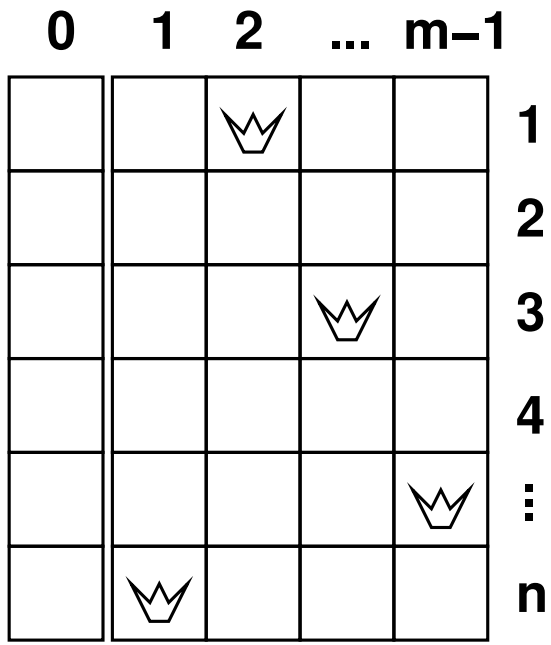
Problém *n* dam – pomocná funkce

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- ```
  hrozba :: [Int] -> [Int]
  hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]
  ```

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- `[6,1,3,5]` `[6,1,3,5]` `[6,1,3,5]`


  ```

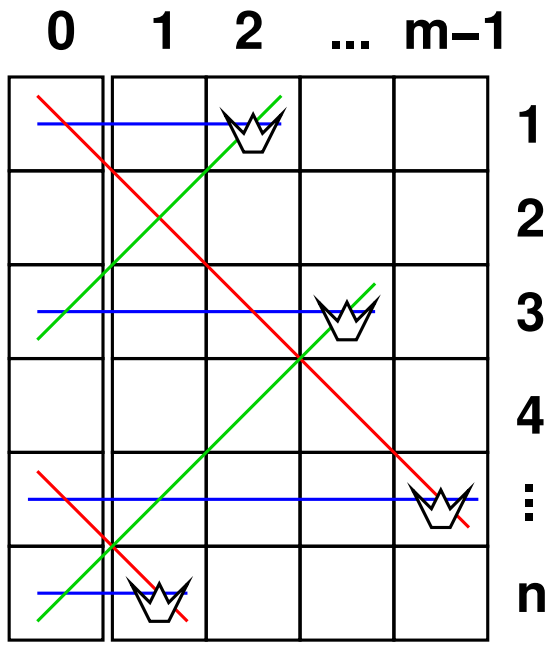
Problém *n* dam – pomocná funkce

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- ```
  hrozba :: [Int] -> [Int]
  hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]
  ```

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- ```
  [6,1,3,5]   [6,1,3,5]   [6,1,3,5]
            +[1,2,3,4] -[1,2,3,4]
  ---------------
  [6,1,3,5]   [-,3,6,-]   [5,-,-,1]
  ```



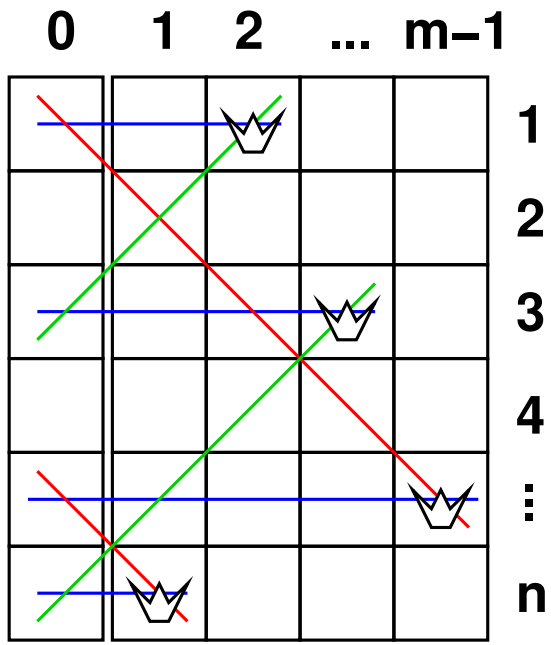
Problém *n* dam – pomocná funkce

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- ```
  hrozba :: [Int] -> [Int]
  hrozba = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]
  ```

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- ```
  [6,1,3,5]   [6,1,3,5]   [6,1,3,5]
            +[1,2,3,4] -[1,2,3,4]
  ---------------
  [6,1,3,5]   [-,3,6,-]   [5,-,-,1]
  ```
- Platné volné pozice: 2 a 4.



Problém *n* dam – řešení

```
type Reseni = [Int]

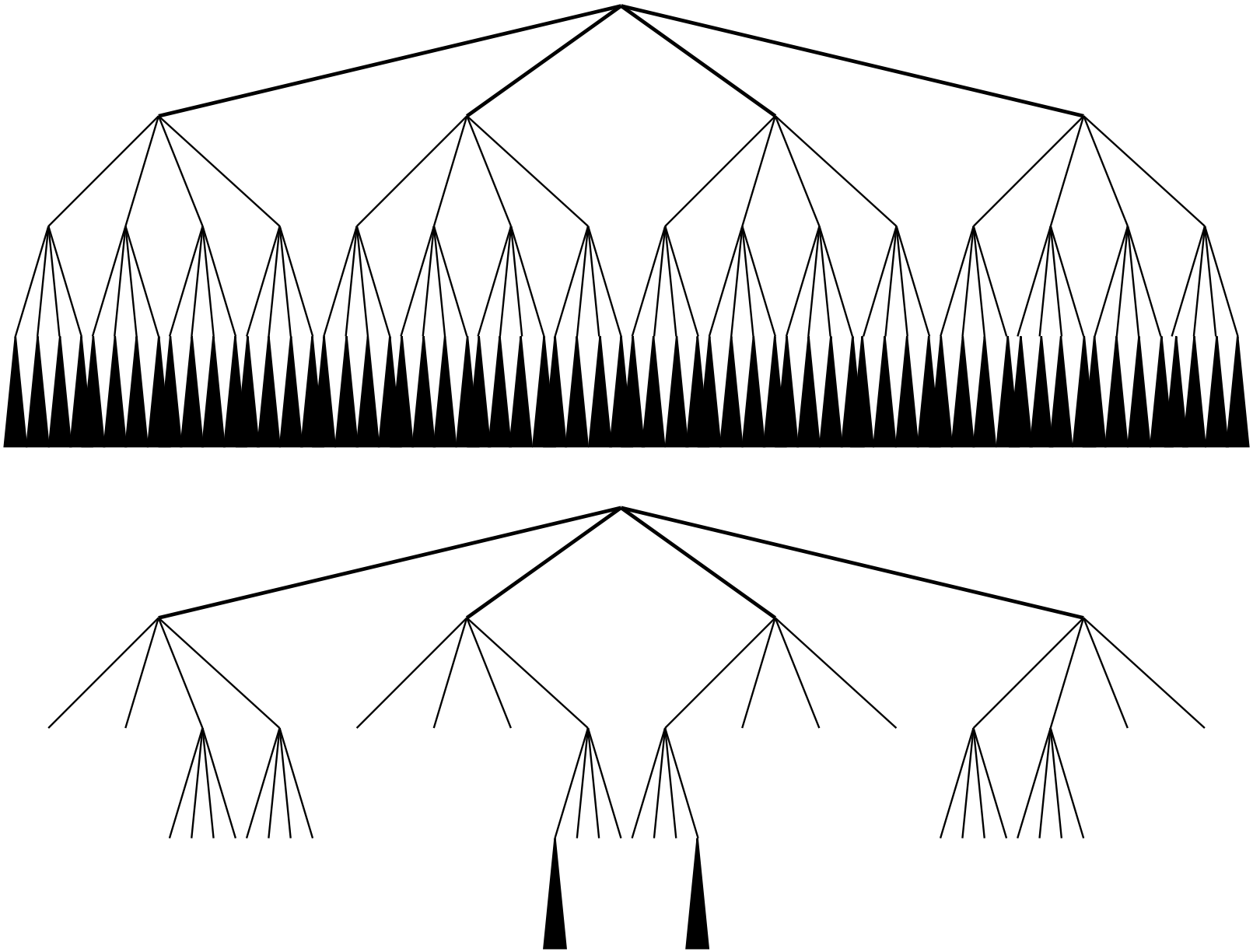
damy :: Int -> [Reseni]
damy n = da n n

da :: Int -> Int -> [Reseni]
da 0 _ = [[]]
da m n = [ k:p | p <- da (m-1) n, k <- [1..n], nh k p]
         where nh k p = k `notElem` ( p
                                      ++ zipWith (+) p [1..]
                                      ++ zipWith (-) p [1..] )
```

Backtracking, Prožezávání

- Backtracking – rekurzivní generování všech možných řešení.
- Prořezávání – časná eliminace neplatných řešení (zde realizováno funkcí `nh k p`).

Problém n dam – efekt prořezávání (n=4)



Nejmenší nepoužité přirozené číslo

(Richard Bird: Pearls of Functional Algorithm Design)

Nejmenší nepoužité přirozené číslo

Zadání

- Pro konečný seznam přirozených čísel zjistěte nejmenší přirozené číslo, které se v seznamu nevyskytuje.

Řešení 1.

- Ze seznamu přirozených čísel "odečteme"zadaný seznam. Nejmenší číslo výsledného seznamu je hledané číslo.

```
minfree :: [Int] -> Int
minfree s = head ([0..] `listminus` s)
```

- Pro odečtení jednoho seznamu od druhého si definujeme pomocnou funkci:

```
listminus :: Eq a => [a] -> [a] -> [a]
listminus u v = filter (`notElem` v) u
```

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]`
  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]`
  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` 1

  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` 2

  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` 3
  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` length v
  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` length v+1
  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` length v+2
  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` length v+3
  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]`
  `v = [9,8,7,6,5,4,3,2,1,0]`

length v + length v-1

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` ...
  `v = [9,8,7,6,5,4,3,2,1,0]`

`minfree` – kvadratické

Funkce `listminus`

- `listminus u v = filter (‘notElem‘ v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]`
  `v = [9,8,7,6,5,4,3,2,1,0]`

Vlastnosti řešení

- Délka výpočtu: length v + length v-1 + length v-2 + length v-3 + ...
- V nejhorším případě kvadratická časová složitost.

Otázka

- Je možné problém řešit s lineární časovou složitostí?

`minfree` – lépe než kvadraticky

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \mathit{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je `(length v + 1)` hodnot `True`.
- Pro každé číslo v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False`, poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[True ,True ,True ,True ,True ]`
  `[0,3,1,4]`

`minfree` – lépe než kvadraticky

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \mathit{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je `(length v + 1)` hodnot `True`.
- Pro každé číslo v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False`, poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[`**`False`**`,True ,True ,True ,True ]`
  `[`**`0`**`,3,1,4]`

`minfree` – lépe než kvadraticky

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \mathit{length}(v)\rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je `(length v + 1)` hodnot `True`.
- Pro každé číslo v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False`, poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[False,True ,True ,False,True ]`
  `[0,3,1,4]`

`minfree` – lépe než kvadraticky

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \mathit{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je `(length v + 1)` hodnot `True`.
- Pro každé číslo v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False`, poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[False,False,True ,False,True ]`
  `[0,3,1,4]`

`minfree` – lépe než kvadraticky

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \mathit{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je `(length v + 1)` hodnot `True`.
- Pro každé číslo v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False`, poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[False,False,True ,False,False]`
  `[0,3,1,4]`

`minfree` – lépe než kvadraticky

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \mathit{length}(v)\rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je `(length v + 1)` hodnot `True`.
- Pro každé číslo v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False`, poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[False,False,True ,False,False]`
  `[0,3,1,4]`

`minfree` – lépe než kvadraticky

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \mathit{length}(v)\rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je `(length v + 1)` hodnot `True`.
- Pro každé číslo v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False`, poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[False,False,True ,False,False]`
  `[0,3,1,4]`

Předpoklady

- Konstantní operace pro přístup k hodnotám omezeně dlouhého seznamu.

Haskell efektivně, aneb od seznamů k polím

Pole v Haskellu

Pole

- Seznam/tabulka adresovatelných míst pro uložení dat.
- Klíčovou vlastností je časově konstantní operace pro přístup k hodnotě na zadané adrese.
- Důležitá datová struktura v imperativním světě.

Poznámka:

- Přístup k *n*-tému prvku seznamu je možné realizovat funkcí

```
(!!) :: [a] -> Int -> a
(!!) (x:s) 0 = x
(!!) (_:s) n = (!!) s (n-1)
```

- Časová složitost operace `(!!)` ale není konstatní.

Pole v Haskellu

Použití

- Pole jsou definována v modulu `Data.Array` .
- `import Data.Array`

Typový konstruktor `Array`

- Binární typový konstruktor: `Array :: *->*->*`
- `Array I A` je typ všech polí, která jsou indexovaná (adresovaná) hodnotami typu `I` a obsahují prvky typu `A`.
- Typ použitý pro indexaci musí být instancí typové třídy `Ix` .

Příklad hodnoty a typu

- ```
  array (1,4) [(1,'a'),(2,'b'),(3,'a'),(4,'b')]
  :: (Num i, Ix i) => Array i Char
  ```
  ```

Pole v Haskellu – indexace

Typová třída `Ix`

- Instance typové třídy `Ix` umožňují efektivní implementaci pole.
- ```
  class (Ord a) => Ix a where
      range :: (a,a) -> [a]
      index :: (a,a) -> a -> Int
      inRange :: (a,a) -> a -> Bool
  ```

Meze pole

- Uspořádaná dvojice indexů tvoří meze pole.
- Všechny hodnoty uvnitř mezí pole jsou platné pro indexaci.
- `range` : Seznam platných hodnot daného rozsahu.
- `inRange` : Test zda je daný index v uvedených mezích.
- `index` : Pozice daného indexu v rozsahu uvedených mezí.
  ```

Pole v Haskellu – meze (příklady)

Instance třídy Ix

- Předdefinovanými instancemi třídy `Ix` jsou typy `Int`, `Integer`, `Char`, `Bool` a jejich kartézké součiny (se sebou samým).

Příklad 1

- `array ('D','F') [('D',2014),('E',1977),('F',2001)]`
  `:: Num e => Array Char e`

Příklad 2

- `range (5,9)` $\rightsquigarrow^*$ `[5,6,7,8,9]`
- `range ('a','d')` $\rightsquigarrow^*$ `"abcd"`
- `range ((1,1),(3,3)))` $\rightsquigarrow^*$
  `[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]`

Pole v Haskellu – přístup k prvkům

Přístupy k prvkům pole

- Uvažme pole `arr :: Array I A` a index do pole `i :: I`, pak `arr!i` je prvek uložený v poli `arr` pod adresou `i`.
- `(!)  :: Ix i => Array i e -> i -> e`
- `(//) :: Ix i => Array i e -> [(i, e)] -> Array i e`

Příklady přístupů k prvkům

- `array (5,6) [(5,"ANO!"),(6,"NE!")] ! 5`
  $\rightsquigarrow^*$ `"ANO!"`
- `array ('D','E') [('D',2014),('E',1977)] ! 'D'`
  $\rightsquigarrow^*$ `2014`
- `array (1,2) [(1,1),(2,2)] // [(1,3),(2,3)]`
  $\rightsquigarrow^*$ `array (1,2) [(1,3),(2,3)]`

Pole v Haskellu – Opearace s poli

Meze pole

- `bounds :: Ix i => Array i e -> (i,i)`

Seznam indexů pole

- `indices :: Ix i => Array i e -> [i]`

Převody mezi poli a seznamy

- `elems :: Ix i => Array i e -> [e]`
- `listArray :: Ix i => (i,i) -> [e] -> Array i e`

Převody mezi poli a seznamy dvojic

- `assocs :: Ix i => Array i e -> [(i,e)]`
- `array :: Ix i => (i,i) -> [(i,e)] -> Array i e`

Pole v Haskellu – accumArray

accumArray

- Knihovní funkce haskellu.
- Vytváří pole z asociativního seznamu a navíc akumuluje (funkcí foldl) prvky seznamu se stejným indexem.
- `accumArray :: Ix i => (e -> a -> e) -> e -> (i, i) -> [(i, a)] -> Array i e`
- Je-li akumulační funkce konstantní, pracuje v lineárním čase.

Příklady

- `accumArray (+) 0 (1,2) [(1,1),(1,1),(1,1)]`
  $\rightsquigarrow^*$ `array (1,2) [(1,3),(2,0)]`
- `accumArray (flip(:)) [] (1,2) [(2,'e'),(1,'l'),(1,'B'),(2,'b')]`
  $\rightsquigarrow^*$ `array (1,2) [(1,"Bl"),(2,"be")]`

Nejmenší nepoužité přirozené číslo II

(Richard Bird: Pearls of Functional Algorithm Design)

minfree – lineárně

Postup

- `minfree s` $\in \langle 0, length(s) \rangle$

  `s = [1,2,6,2,0]`

- Nebudeme uvažovat čísla mimo očekávaný rozsah.

  `(filter (<=n) s) where n = length s` $\rightsquigarrow^*$ `[1,2,2,0]`

- Vytvoříme asociační seznam (příprava na konverzi do pole).

```
t s = (zip (filter (<=n) s) (repeat True))
      where n = length s
```

  `t s` $\rightsquigarrow^*$ `[(1,True),(2,True),(2,True),(0,True)]`

- Vytvoříme pole a odstraníme duplicity tak, aby nepoužité indexy ukazovaly na `False` .

```
checklist :: [Int] -> Array Int Bool
checklist s = accumArray (||) False (0,length s) (t s)
```

Mezivýsledek

- `checklist s` $\rightsquigarrow^*$

  `array (0,5) [(0,True),(1,True),(2,True),(3,False),(4,False),(5,False)]`

minfree – lineárně (pokr.)

Postup pokračování

- V seznamu typu `Array Int Bool` najdeme nejmenší index odkazující na `False`.

  ```
  search :: Array Int Bool -> Int
  ```

- Nejprve převedeme pole na seznam hodnot typu `Bool`

  ```
  elems (checklist s)
  ```

  $\rightsquigarrow^*$ `[True,True,True,False,False,False]`

- Ze seznamu ponecháme pouze jeho prefix s hodnotami `True`

  ```
  (takeWhile id . elems) (checklist s)
  ```

  $\rightsquigarrow^*$ `[True,True,True]`

- Index prvního `False` určíme jako délku tohoto prefixu.

  ```
  search = length . takeWhile id . elems
  search (checklist [1,2,6,2,0]) ⇝* 3
  ```

minfree – lineárně (řešení)

Řešení

- `minfree = search . checklist`

```
search :: Array Int Bool -> Int
search = length.takeWhile id . elems

checklist :: [Int] -> Array Int Bool
checklist s = accumArray (||) False (0,n)
                  (zip (filter (<=n) s) (repeat True))
                      where n = length s
```

Složitost algoritmu

- `checklist` , `search` – lineární
- `minfree` – lineární

Lineární řazení některých seznamů

(Richard Bird: Pearls of Functional Algorithm Design)

Lineární řazení čísel z omezeného rozsahu

Pozorování

- Jsou-li čísla v seznamu z omezeného rozsahu, je možné tento seznam setřídit v **lineárním čase**.

Algoritmus

- ```
  max = 1000

  countlist :: [Int] -> Array Int Int
  countlist s = accumArray (+) 0 (0,max) (zip s (repeat 1))

  sortBoundedList s =
      concat [ replicate k x | (x,k) <- assocs (countlist s) ]
  ```

Příklad

- `sortBoundedList [3,3,4,1,0,3]` $\rightsquigarrow^*$ `[0,1,3,3,3,4]`
  ```

USS Haskell

IB016 – Placená reklama

Zaujal vás Haskell?

- Chcete programovat reálné úlohy v Haskellu?
- Chcete nahlédnout pod roušku magie odpovědníků a zjistit, jak vygenerovat a zobrazit náhodnou funkci?
- Chcete vědět, jak se řeší chybové stavy, parsování, či vysokoúrovňový paralelismus ve funkcionálním paradigmatu?

IB016 Seminář z funkcionálního programování

- Registrace běží, registrujte se i přes limit.
- Vedou Vladimír Štill a Martin Ukrop.