```
  5    1    5    2    5    3    5    4    5    5    5    6    5    7    5    8
2001.
xx.11.2001/ 9.cv Scheme (a písomka)

4.12.2001/ 10 cv. Haskell
- jednoducha definicia
  x_mezi_low_high lo hi x = lo <= x && x <= hi
- kolinearne body, body ako dvojice
  - pomocou strazi
- euklidov alg.
- spracovanie zoznamov, zobecnene fib. cisla
- priklady polymorfnych funkcii
  - suda-licha a mergesort

11.12.2001/ 11.cv
- definicia jednoducheho typu (den_v_tydnu), fce. dalsi_den
-? komplexne cisla, Int s okraj hodnotami, ...
- filter: pomocou if, aj strucnych zoznamov
- funkcie vyssich radov: priklady
(- zip, zipWith -bude na predn.)
  ? pouzitie v: dalsi riadok pascalovho trojuholnika
- drop, takeWhile, (dropWhile)
- zobecnenie qsortu (bol na predn.):
a) porovnavacia b) selektorova fcia
- tvorba tried ekvivalencii, podla relacie
- rekurzivne d.š.: binarne vyhl. stromy, fcie hlad., vklad., vyp.
(- n-arne stromy )
  (- tvorba, pomocou tried ekvivalencii)
(- typ s chybou: Maybe a)
(- zjednotenie typov: Either a b )

18.12./12.cv.
cvicenie :
- asi dokoncim operacie na vyhladavacich binarnych stromoch
    (delete)
- nejaku nekonecnu d.s.: eratostenovo sito, nekonecna
    diagonalna matica:
    diag = iterate (0:) (1:repeat 0)
  - nezaviedol som funkcie repeat, iterate, cycle, pripadne ich
    zavedte:
    repeat x = x: repeat x
    iterate f x = x: iterate f (f x)
    cycle xs = xs ++ cycle xs
- nejaka zlozitejsia operacia:
```

```
  - graficka filtracia matice: dana matica pixelov a filter
    - vybranie podmatice danych rozmerov, ak existuje
  - vybudovanie huffmanovho stromu (ak huffmana poznaju)
-? vybudovanie vyhladavacieho stromu

8.1.2002/13.cv.
(typové triedy)
(huffmanovo kodovanie - kodovanie a dekodovanie)
prechod bin. stromu do šírky pomocou fold
aritmeticke vyrazy: vyhodnocovanie, v. s chybou, preklad
  - pomocou fold

-----
kod:
4.12.2001/ 10 cv. Haskell
- jednoducha definicia
      :: Int -> Int -> Int -> Bool  -- zamyslane pouzitie
  x_mezi_low_high lo hi x = lo <= x && x <= hi
  - ciastocna aplikacia: x_mezi.. lo hi :: Int -> Bool
    otestuje, ci je x v danom intervale

- kolinearne body, body ako dvojice
  - pomocna definicia fcie smernica, za where
-- nespravne, problemy s delenim 0, v smernici
kolin :: (Float,Float)->(Float,Float)->(Float,Float)->Bool
kolin a b c = smernica a b == smernica a c
  where smernica (ax,ay) (bx,by) = (by-ay)/(bx-ax)

- vylepšenie pomocou stráží, ak sú body nad sebou: test nad
kolin a b c
  | a == b || a == c = True
  | nad a b          = nad a c
  | nad a c          = False
  | otherwise        = smernica a b == smernica a c
  where smernica .. = ... -- stejne
        nad (ax,_) (bx,_) = ax==bx
  -- nad obsahuje anonymne vzory

- euklidov alg. (Greatest Common Divisor), z prelude
gcd :: Int -> Int -> Int
gcd 0 0 = error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y = gcd' (abs x) (abs y)
          where gcd' x 0 = x
                gcd' x y = y (x `rem` y)
```

```
- spracovanie zoznamov, zobecnene fib. cisla


- priklady polymorfnych funkcii
  - na zoznamoch: length, head, tail, last,
    take (Int->[a]->[a]), drop, ... -- polymorf. v zozname, nie cisle
    (++) tj. append, reverse, map, filter, zip ...
    suda_licha :: [a]->([a],[a]) -- vracia dvojice
    transpozicia matice: [[a]] -> [[a]]
      polymorfna v prvkoch matice, nie v riadkoch
    konstruktorove funkcie zoznamu:
      [] :: [a] -- polymorfna konstanta
      (:) :: a->[a]->[a] -- alias "pridavanie na zasobnik",
                                  tail je doplnkove "uberanie"
  -Q: nasobenie matic: nie je polymorfne, pretože predpokladame
        určite operácie na prvkoch (sčítanie a násobenie)
  -Q: množinove operacie, (multimnožinove operácie - pre "bags")
      predpokladame, že dokážeme prvky porovnavat, takže tiež kladieme
        na typ dodatočné omedzenia

- suda-licha a mergesort
  - priklad na porovnávanie so vzorom: pattern matching
suda_licha :: [a]->([a],[a]) -- vracia dvojice
suda_licha []       = ([],[])
suda_licha [x]      = ([x],[])
suda_licha (x:y:xs) = (x:v1,y:v2)
  where (v1,v2) = suda_licha xs -- porovnanie so vzorom
  - varianta poslednej definicie pomocou selektorov
  suda_licha (x:y:xs) = (x:fst dv,y:snd dv)
    where d = suda_licha xs
  - selektory pre dvojicu su preddefinované
fst (x,_) = x
snd (_,y) = y

- mergesort, bez parametrickeho porovnavania
msort [] = []
msort [x] = [x]
msort xs = -- pre aspon dvojprvkove zoznamy
  merge (msort v1) (msort v2)
  where (v1,v2) = suda_licha xs
        merge [] ys = ys
        merge xs [] = xs
        merge (x:xs) (y:ys) =
```

```
            if x<= y then x:merge xs     (y:ys)
                     else y:merge (x:xs) ys

11.12.2001/ 11.cv
- zobecnenie qsortu (bol na predn.):
a) porovnavacia b) selektorova fcia
- tvorba tried ekvivalencii, podla relacie
- rekurzivne d.š.: binarne vyhl. stromy, fcie hlad., vklad., vyp.
(- n-arne stromy )
  (- tvorba, pomocou tried ekvivalencii)
(- typ s chybou: Maybe a)
(- zjednotenie typov: Either a b )

11.12./11.cv.
nejake jednoduche datove definicie, s funkciami, napr.
-  Den = Po | Ut | St ... Ne
  dalsi_den :: Den -> Den
  dalsi_den Po = Ut
  ...
  weekend :: Den -> Bool
  weekend So = True
  weekend Ne = True
  weekend _  = False

(-? komplexne cisla)

- filter pomocou if aj strucnych zoznamov
  filter p xs = [x | x<-xs, p x] :: (a->Bool)->[a]->[a]
  filter2 p [] = []
  filter2 p (x:xs) = if p x then x:xs' else xs'
        where xs' = filter2 p xs
  - filtracia nevyhovujucich podmienke:
  ...(filter (not.p) xs)...

-? zip:: [a] -> [b] -> [(a,b)]
  (- zipWith :: (a->b->c) ->[a] -> [b] -> [c]
  ? pouzitie v: dalsi riadok pascalovho trojuholnika

- drop, takeWhile, (dropWhile)
-- drop zahodí prvých n prvkov; ak neexistujú, vracia prázdny zoznam
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop (n+1) (x:xs) = drop n xs
drop _     []     = []
```

```
-- takeWhile p xs berie prvky xs, kym je splnena podmienka p
takeWhile :: (a->Bool) -> [a] ->[a]
takeWhile p [] = []
takeWhile p (x:xs) = if p x then x:takeWhile p xs else []

- zobecnenie quicksortu, usporiadanie podla porovnavacej funkcie,
  - podla vyberovej funkcie so std. porovnavanim
-- s porovnavanim, ako parameter
qsort :: (a->a->Bool) -> [a] ->[a]
qsort _ [] = []
qsort lt (x:xs) = [y|y<-xs, lt y x] -- staci 1 porovnavacia fcia.
                ++ [x]
                ++ [y|y<-xs, not(lt y x)]

  - zostupne porovnávanie:
  ... (qsort (flip lt) xs)... where
        flip f x y = f y x -- prehodenie argumentov

- qsort s porovnavanim ppomocou selektora
        nepotrebujem porovnavat cele prvky, ale len podla ich casti
        selektor urcuje, podla ktorej casti sa porovnáva
        napr. podľa prvej zložky dvojíc
qsort2 :: (a->b) ->[a]->[a] -- b porovnateľné
qsort2 sel xs = qsort (\x y -> sel x < sel y) xs
        ... , sel x < sel y] ++ ...
  - ak je výpočet selektora náročný, môžem si ho predpočítať
        do dvojice (sel x,x) a potom porovnávať podľa prvej zložky

- funkcie vyssich radov: priklady
takeWhile, map, zip, zip3, filter, rozdel, ekv_triedy, ...
flip :: (a->b->c) -> b -> a -> c
()curry, uncurry)

- ekv_triedy: vyrobí triedy podľa danej ekvivalencie
ekv_triedy :: (a->a->Bool) -> [a] ->[[a]]
ekv_triedy ekv [] = []
ekv_triedy ekv (x:xs) = (x:filter (ekv x) xs)
                        : ekv_triedy ekv (filter (not.ekv x) xs)
  (- rozobrať línost)
  - prístup so zatriedovaním prvkov do tried je efektívnejší

- rekurzivny d.t. : binarne vyhl. stromy, operacie
  Bt a = Bnil | Bnode (Bt a) a (Bt a)
  - hladanie, vkladanie, vypustanie
```

```
inBT :: a -> Bt a -> Bool
inBt x Bnil = False
inBt x (Bnode l y r) = if x==y then True
                        else if x<y then inBt x l
                                    else inBt x r

insertBt :: a -> Bt a -> Bt a
insertBt x Bnil = Bnode Bnil x Bnil
insertBt x (Bnode l y r) = if x==y then Bnode l y r
                           else if x<y then Bnode (insertBt x l) y r
                                       else Bnode l y (insertBt x r)

delBt :: a -> Bt a -> Bt a
...

- n-arne vyhl stromy: Ntr a = Nt a [Ntr a]
  - tvorba stromu
(- typ s chybou: Maybe a = No | Just a )
(- zjednotenie typov: Either a b = Left a | Right b)

-------------------
18.12./12.cv.
cvicenie :
- asi dokoncim operacie na vyhladavacich binarnych stromoch
    (delete)
- nejaku nekonecnu d.s.: eratostenovo sito, nekonecna
    diagonalna matica:
    diag = iterate (0:) (1:repeat 0)

  - nezaviedol som funkcie repeat, iterate, cycle, pripadne ich
    zavedte:
    repeat x = x: repeat x
    iterate f x = x: iterate f (f x)
    cycle xs = xs ++ cycle xs

- pripony:
-- v.1: spadne na (tail []), prípadne
tails xs = iterate tail xs
> tails [1,2] ~~> [1,2] : [2] : [] : undefined
-- v.2, využíva lazyness
tails xs = takeWhile ([]/=) (iterate tail xs) ++ [[]]

- nejaka zlozitejsia operacia:
  - graficka filtracia matice: dana matica pixelov a filter
```

```
    - vybranie podmatice danych rozmerov, ak existuje
  - vybratie matice mxn
take_mat m n xss = take n (map (take m) xss)

  - násobení filtrem, po složkách
zipWith_m f xss yss = zipWith (zipWith f) xss yss
nasob m filtr = sum (concat (zipWith_m (*) m filtr ))
  -- použití (.), ať není moc vložených funkcí v sobě
nasob m filtr = ( sum . concat . zipWith_m (*) ) m filtr -- s (.)
  -- použití >.> , ať píšu zleva v pořadí vykonávání
nasob m filtr = ( zipWith_m (*) >.> concat >.> sum ) m filtr -- s >.>
  where f >.> g = g.f (:: (a->b) -> (b->c) -> (a->c) )

  - přefiltrování matice, při vhodném dodefinování okrajů matice

  - vybudovanie huffmanovho stromu (ak huffmana poznaju)
huff :: [(a,Int)] -> [(a,[Int])]
  - prva faza: vyrova jednolistovych stromov
f1 xs = map (mpair Leaf id) xs  -- ( \(x,y) -> (Leaf x,y) )
        where mpair ... -- viz dole
  - druha faza: utriedenie podla cetnosti
f2 xs = msort (\x y -> snd x < snd y) xs
  - tretia faza: stavba stromu: iteracne zlievame
f3 :: [(Bt a, Int)] -> Bt a
f3 xs = ( fst           -- vezme len strom, cetnost je nezaujimava
        . head          -- vezme jediny prvok
        . pevny_bod     -- nie je čo zlievat, zoznam je jednoprvkovy
        . iterate zlej) xs -- zlieva vrcholy
        where
          pevny_bod (x:y:xs) = if x==y then x else pevny_bod (y:xs)
          zlej ...
  - potrebujeme proceduru pre iteraciu do pevneho bodu
  - stvrta faza: vyrobime kodovaciu tabulku zo stromu
f4 (Leaf a,cetn) = [(a,"")]
f4 (Node l r,_ ) = map (\(x,y)->(x,'0':y)) (f4 l)
                ++ map (mpair id ('1':))   (f4 r) -- varianty funkc. par.
  - výsledok je lookup tabulka (podla obecneho typu, nielen znaku)

-? vybudovanie n-arneho vyhladavacieho stromu

- techniky
  - map na dvojicu pomocou dvoch funkcii
map_pair f g xs = map (mpair f g) xs where
        mpair f g (x,y) = (f x, g y)
```

```
  (- typ s chybou: Maybe a = No | Just a )
  - zjednotenie typov: Either a b = Left a | Right b)
f_either :: (a->c) -> (b->c) -> Either a b -> c
f_either f g (Left x)  = Left  (f x)
f_either f g (Right x) = Right (g x)
  - f_either s dvomi par. vyrába funkciu typu Either a b -> c, ktorá
spracúva prvok, nech je z ktorejkoľvek zložky
  - potom môžeme používať:
... filter (f_either fleft fright) xs
... map    (f_either fleft fright) xs

8.1.2002/13.cv.
- prechod bin. stromu do šírky pomocou fold:
foldB :: (a->b)->(b->b->b)->Tree a-> b
- prechod do hlbky, zlava
do_hlbky_zlava :: Tree a -> [a]
do_hlbky_zlava s = foldB (\x->[x]) (++) s
  - druhy f. arg. spaja zoznamy vytvorene prechodom podstromov
- anologicky: do_hlbky_sprava
do_hlbky_sprava :: Tree a -> [a]
do_hlbky_sprava s = foldB (\x->[x]) (flip (++)) s
  - druhy argument spaja zoznamy odpovedajuce podstromom v opacnom
    poradi, najprv pravy, potom lavy
- vlastny prechod do sirky:
  pri prechode do sirky musime upravit datovy typ vysledku: z dvoch
zoznamov nedokazeme vytvorit spravny vysledok -> potrebujeme dodatocne
informacie alebo strukturu vysledku.
  mozne dodanie informacii:

1. ku kazdemu pridame cislo pre hlbku. Pri vytvarani vysledku
potrebujeme zliat (merge) usporiadane zoznamy. Hlbku mozeme pocitat
  a) odspodu: kazda uroven potom vyzaduje prechod celym zoznamom a
pricitanie 1 k hlbke, v listoch (Leaf) zaciname hlbkou 1
  b) zvrchu: predavame dodatocny argument, v ktorom si pamatame hlbku.
Pri prechode do podstromov hodnotu zvacsujeme. V listoch ju vyuzivame.
(Pri priamociarej implementacii potrebujeme obecnejsi funkcional nez
foldB, ale je mozne pouzit techniku analogicku k prevodu f. take na
fold. Namiesto elementarneho vysledok sa vracia funkcia 1 argumentu. Jej
argument je aktualne hlbka. Volanie foldB potom musi zadat/inicializovat
aj tento argument.)

2. rozoberanie stromov zvrchu (neni instanciou foldB)
Vytvorime zoznam stromov v hlbke n. V jednom prechode uberieme vsetky
listy a vytvorime zoznam stromov v hlbke n+1. Vysledok sa tvori po
```

urovniach.

3. strukturovanie vystupu do hladin.
Pridanie hlbky ku kazdemu prvku je zbytocne narocne. Staci strukturovat zoznam listov do hladin podla hlbky, tj. vysledok volania bude typu [[a]], tj. zoznam urovni, v ktorych su listy danej hladiny.

```
do_sirky s = foldB (\x->[[x]]) zlej_hladiny s  where
        zlej_hladiny x y = zipAll (++) x y   -- (1)
```
- pre vlastne zlievanie potrebujeme analogiu k zipWith. Funkcia zipAll ale prechadza cele zoznamy, pretoze zoznamy mozu mat roznu dlzku a my chceme , aby zoznam hladin odpovedal dlhsiemu zoznamu. Ak jeden skonci, kopiruje sa do vysledku druhy.

```
zipAll :: (a->a->a) -> [a] -> [a] -> [a]
zipAll f (x:xs) (y:ys) = f x y : zipAll f xs ys
zipAll _ []     ys     = ys
zipAll _ xs     []     = xs
```
- namiesto zavadzania novej funkcie zipAll by sme spajanie hladin mohli osetri pomocou zipWith dvomi sposobmi. Parva moznost je ku kratsiemu zo zoznamov pripojit zoznam prazdnych hladin. F. zipWith potom vrati zoznam hladin odpovedajucich dlhsiemu zoznamu. Druha moznost je skratit jeden zo zoznamov a prebyvajuce prvky pripojit k vysledku samostatne.
(DC: vymysliet prehladavanie !grafu do sirky)

-- aritmeticke vyrazy: vyhodnocovanie, v. s chybou, preklad
  - pomocou fold
- zavadenie datoveho typu:
```
data Exp a =      Con a
                | Var String
                | Plus (Exp a) (Exp a)
                | Div  (Exp a) (Exp a)
```
- definicia foldEx pre prave zavedeny typ
```
foldEx :: (a->b) -> (String->b) -> (b->b->b) -> (b->b->b) ->
                Exp a ->b
foldEx fc fv fp fd exp = f exp where
        f (Con x) = fc x
        f (Var s) = fv s
        f (Plus x y) = fp (f x) (f y)
        f (Div  x y) = fd (f x) (f y)
```
- pomocou foldEx chceme pre vhodne funkcionalne parametry ziskat
1. vyhodnocovanie vyrazu
2. vyhodnocovanie vyrazu s chybami

```
3. preklad do polskej notacie (zasobnikoveho stroja)

1. vyhodnocovanie bez osetrovania chyb pouziva tabulku hodnot premennych
eval :: [(String,Int)] -> Exp Int -> Int
eval tab exp = foldEx id (lookup tab) (+) div exp

2. pre osetrovanie chyb zavedieme typ vysledku Either Int String, kde
typu Int su platne vysledky a typu String su chyby. V tychto vyrazoch
moze dojst k dvom druhom chyby: delenie nulou alebo neznama premenna. Ak
cheme rozlisovat tieto dva druhy chyb, nestaci nam typ Maybe Int, ktory
ma len jednu dodatocnu hodnotu.

evalErr tab exp = foldEx fc fv fp fd exp where ...

jednotlive funkcie budu:
fc = Left --  \x -> Left x
technicky: vnorenie korektnej hodnoty do typu Either

fv = maybe2either . lookup tab
maybe2either Nothing  = Rigth "neznama premenna"
maybe2either (Just x) = Left x
- prevedenie vysledku hladania typu Maybe Int pomocou jednoucelovej
funkcie na pozadovany typ Either
  - ak chceme vydat meno nezname premennej v chybovom retazci, musime ho
predat chybovej premennej:
fv' s = maybe2either' s (lookup tab s)

fp = lift2Ex (+)
lift2Ex op (Left x) (Left y) = Left (x `op` y)
lift2Ex _  (Right x) (Right y) = Right (x ++ "; " ++ y)
lift2Ex _  x1@(Right x) _      = x1
lift2Ex _  _       y1@(Right y) = y1
- pozdvihnutie binarnych operacii, ktore nevyvolavaju chybu, pomocou
funkcie lift2Ex. Analogicky pre ine operacie (krat, minus). Pre zdvih
unarnych operacii by sme zaviedli dalsie funkciu lift1Ex.
  - ak su chyby v oboch argumentoch, vydavame popis obidvoch

fd x y = case y of
        Left 0 -> Right "delenie nulou"
        _      -> lift2Ex div x y
- funkcie, ktore mozu sposobit chybu, osetrujeme tak, ze osetrime
chybove podmienky a potom pouzijeme obvykle zdvinutie operacie div.
  - testovanie chybovych podmienok moze byt spravene obecne, vstupom
bude dvojica (podmienka, chybova hlaska) (alebo zoznam dvojic)
```

```
  - v tejto implementacii ignorujeme pripadnu chybu v prvom parametre,
ak sa deli nulou

3. preklad
hm.

(typové triedy)
  - vymysliet novu rozumnu triedu nie je jednoduche
  -? pr. instancie: maximolexikograficke usporiadanie pre zoznamy
  -? datova struktura s vyctom  prvkov:
        odpoveda vzoru "iterator"
        ?konstruktorova trieda:

(huffmanovo kodovanie - kodovanie a dekodovanie)
kodovanie je priamociare, ak mame tabulku
codehuff :: [(a,[Int])] -> [a] -> [Int]
codehuff tab s = concat(map (lookup tab) s)
decodehuff ::  Tree a -> [Int] -> [a]
decodehuff tr s = ...
        where
        decode1 :: Tree a -> [Int] -> Maybe (a,[Int])
        -- dekodovanie 1 symbolu typu a. Na konci suboru sa vracia
Nothing. Inac sa vracia symbol a nepouzity zvysok vstupu (ako u
parserov).


---------------------
temy:
1. Nekonecne dat. struktury

1a. Budovanie nekonecnych datovych struktur
= fibonaciho cisla, repeat, iter
  == dat. str. ktora ma len jeden konstruktor, sa moze vygenerovat nekonecna
    analogicky k repeat pre zoznam, nap. bin strom - oboch druhov, ntree, ...
= ako sa pracuje s nekon . dat. strukturami: vzdy sa utrhne len konecna cast
= map a ine funkcie napisane tak, aby spracovali aj nekon. dat. strukturu,
  tj. lazy
  == spracovanie akumulatorom takto nefunguje /foldl vs. foldr./
= priklad" pascalov trojuholnik

repeat x = x: repeat x
repeatBt = Bnode repeatBt repeatBt
repeatBt1 x = Node (repeatBt1 x) x (repeatBt1 x)
repeatNTr x = Ntr x (repeat (repeatNTr x))
```

```
-- v.1, neefektivna, vypocet x^n opakovaný
iter1 f x = x: map f (iter1 f x)
-- v.2, OK
iter f x = x: iter f (f x)

-- pascalov trojuholnik
pascal = iter g [1]
  where g xs = zipWith (+) ([0]++xs) (xs++[0])

-- dynamicke programy idu riesit analogicky, ak staci rozvoj v jednej dimenzii
===
Haskell

z cviceni:
13.12'99
- fibonaciho cisla

- lave vs. prave vs. striktne and
andl :: Boolean -> Boolean -> Boolean -- lave
andl True  x = x
andl False _ = False

andr :: Boolean -> Boolean -> Boolean -- prave
andr x True  = x
andr _ False = False

and :: Boolean -> Boolean -> Boolean -- striktne
and True True = True
and _    _    = False

- Binarne stromy s vnutornymi vrcholmi
data Bt a = Bnil
          | Bnode (Bt a) a (Bt a)

  - vyroba zoznamu listov v usporiadani:
    pomocou pattern matching na uzivatelsky definovanom type
listy :: Bt a -> [a]
listy Bnil = []
listy (Bnode l x r) = listy l ++ (x: listy r)

  - vyroba vyvazeneho stromu z usporiadaneho zoznamu:
    - take a drop su uz zname funkcie,
      take::Int->[a]->[a] vezme prvych n prvkov
```

```
      drop::Int->[a]->[a] zahodi prvych n prvkov
mk_bt :: [a]->Bt a
mk_bt xs
  | xs == []  = Bnil
  | otherwise = Bnode (mk_bt (take k xs))
                y
                      (mk_bt ys)
    where
      k = n `div` 2
      y:ys = drop k xs

- map: ...

- zip pre (Bt a):
  kym sa stromy zhoduju, vraciame dvojicu hodnot z oboch stromov.
  Ak sa struktura zmeni, vratime Bnil (pretoze mame nularny
  konstruktor Bnil, nie je problem vyriesit nezhodu struktury)
zip :: Bt a -> Bt b -> Bt (a,b)
zip (Bnode lx x rx) (Bnode ly y ry) =
           Bnode (zip lx ly) (x,y) (zip rx ry)
zip _               _               = Bnil

  - zobecnenie zip na funkciu zipWith: nechcem vracat dvojicu,
ale lubovolnu hodnotu podla danej funkcie
  {- pozn.:
   -- v Haskelli sa aktivuje len prva definicia
   -- v Curry sa splnuje nedeterministicky aj podla druhej klauzule,
   -- vracia viac vysl. (a niektore aj viackrat, pretoze
   -- kombinuje vysl. z rekurzie:
  --   (bnode,bnil),(bnil,bnode),(bnil,bnil) ) -}
zipWith :: (a->b->c) -> Bt a -> Bt b -> Bt c -- OK
zipWith f (Bnode lx x rx) (Bnode ly y ry) =
     Bnode (zipWith f lx ly) (f x y) (zipWith f rx ry)
zipWith f _               _               =
     Bnil

  -- pretoze sa jedna o zobecnenie, da sa zip napisat ako spec.
     pripad zipWith:
zip x y = zipWith (\x y->(x,y)) x y

-- DC: roznasobenie stromov
roznas :: Bt [a] -> [Bt a]
roznas Bnil = [Bnil]
roznas (Bnode l xs r) = map (\(l,(x,r))->Bnode l x r)
```

```
                          (cart (ls, (cart (xs,rs)) )) -- OK
                          -- (ccart ls (ccart xs rs) ) -- xx
  where ls = roznas l
        rs = roznas r
        -- cart je kartezsky sucin
        -- cart :: ([a],[b]) -> [(a,b)]
        ccart x y = cart (x,y) -- neprejde typmi
        -- ccart = curry cart -- nejde otypovat
           -- prehladnejsia varianta: ... (ccart ls (ccart xs rs) )
           -- ale nefunguje: ccart nejde (v Curry) otypovat


  -- kartezsky sucin (ako elementarna operacia roznasobenia)
cart :: ([a],[b]) -> [(a,b)]
cart ([],_) = []
cart (x:xs,ys) = map (\y->(x,y)) ys ++ cart (xs,ys) -- OK
-- def. pomocou foldr:
-- cart (xs,ys) = foldr (++) []
   --                 (map (\x->map (\y->(x,y)) ys) xs) -- OK
-- i jinak
bt1 = Bnode (Bnode Bnil [1,2,3] Bnil)
            [11,12,13]
            (Bnode Bnil [21,22,23] Bnil)
bt2 = Bnode bt1 [31] bt1
-- test: zipWith (++) bt2 bt1

--
- where musi but u kazdej definicie pomocou vzorou (alebo pouzit
  straze)

-- cvicenie 7.1.2000, ladene
{- skladanie fci:
Pozname obvykle skladanie:
(.):: (b->c)->(a->b)->(a->c)
(f.g)x= f(g x)
  - nevyhoda je, ze funkcie sa aplikuju odzadu
  - chceme dopredne skladanie, aby sa funkciue aplikovali zlava doprava
-}
-- infixl 9 >.> -- deklaracia operatora
(>.>) :: (a->b)->(b->c)->(a->c)
f >.> g = g.f
  -- parameter x nemusime pisat; na oboch stranach je funkcia

-- n-nasobne skladanie funkcie:
```

```
{- pre lub. fci f::a->a mozeme definovat opakovane zlozenie. Piseme
   funkciu, ktorej dame parameter n= pocet opakovani a vysledkom bude
   zlozena funkcia. Pre n=0 ma zmysel dodefinovat vysledok ako Identitu
   (preddefinovana fce Id:: a->a)
-}
fsloz 0 f = id
fsloz (n+1) f = f >.> fsloz n f
  -- skladania je asociativne, preto na zatvorkovani nezalezi
  -- druha moznost:
fsloz1 0 f = id
fsloz1 (n+1) f = f . fsloz n f
  {- prva moznost je pamatovo efektivnejsia, pracuje v konst. pamati
     pretoze najprv spocita hodnotu, potom je posiela dalej. V druhej
     moznosti musime rozvinut n slozeni funkcie nez mozeme zacat pocitat
  -}
-- testy
t51 = fsloz 1000 (100+) 51
t52 = fsloz1 1000 (100+) 52

-- skladanie fci delenim na poloviny
{- - namiesto aby sme skladali funkcie po jednej, budeme delit n na poloviny
   ("usetri sa" vyroba funkcie, nie jej vyhodnotenie (to bude typicky
   na inom argumente) )
   - je to mozne robit skladanim odspodu alebo zhora, kvoli lichym cislam
    je zhora (delenim n) pre mna prirodzenejsie
-}
fslozb n f
  | n==0           = id
  | n `mod` 2 == 0 = fslozb (n `div` 2) f >.> fslozb (n `div` 2) f
  | otherwise      = f >.> fslozb (n-1) f   -- liche
t53 = fslozb 1000 (100+) 53 -- test

-- aplikacia
plus1 x y = fsloz1 x (1+) y -- ako opakovane pricitanie jednicky
krat1 x y = fsloz1 x (plus1 y) 0 -- ako opakovane pricitanie y
exp1  x y = fsloz1 y (krat1 x) 1 -- ako opak. nasobenie x
t54 = plus1 1000 54
t55 = krat1 10 55
t56 = exp1 3 4

-- 2-3 stromy ...
-- testovanie usporiadanosti, pridavanie, (vyberanie)

-- testovanie usporiadanosti stromu
```

```
{- pre testovanie usporiadanosti su tieto principialne moznosti:
1. otestuje sa kazdy kluc voci minimu (svojho) praveho a maximu laveho
podstromu.
   - Nevyhoda: opakovane hladanie minim a maxim
2. budeme posielat dolu povolene hranice, kazdy vrchol otestujeme na hranice
a pre podstromy hranice zuzime. Pre okrajove podstromy vyrobime
spec. procedury, ktorym dodame len jednu hranicu (minimum, resp. maximum)
   - Nevyhoda: niekolko podobnych procedur
3. podobne ako 2., ale dolu sa budu posielat namiesto cisel rozsirene
cisla (o spravne nekonecno).
   - Nevyhoda: porovnavanie musime vykonavat na rozsirenych cislach,
     predavat do rekurzie musime rozsirene cisla
Budeme robit 3. moznost. Kvoli moznym opakujucim sa kluciom predpokladame,
ze vsetky stejne prvky su v lavom podstrome (vzhladom ku klucu).
Pre rozsirene cisla zavedieme typ Maybe, bude reprezentovat obidve hranice
-}
data Maybe1 a = No -- Maybe je preddefinovane
              | Yes a

-- porovnavanie: No je dolne nekonecno, le = less equal
-- pouzivame typove triedy.
le :: (Ord a)=> Maybe1 a -> Maybe1 a -> Bool
Yes a `le` Yes b = a<=b
Yes _ `le` No    = False
No    `le` _     = True

-- porovnavanie : No je horne nekonecno, lt = less then
lt :: (Ord a)=> Maybe1 a -> Maybe1 a -> Bool
Yes a `lt` Yes b = a<b
Yes _ `lt` No    = True
No    `lt` _     = False

data Bt a = BV -- Bt void
  | BN (Bt a) a (Bt a) -- BNode

-- sorted :: (Ord a)=>Bt a -> Bool
sorted tree = sorted1 No No tree
  where sorted1 minx maxx BV = True
sorted1 minx maxx (BN l x r) =
            (minx `le` Yes x)  -- porovnanie vrcholu
         && (Yes x `lt` maxx)   -- predavame Yes x !
         && sorted1 minx (Yes x) l  -- kontrola podstromov
 && sorted1 (Yes x) maxx r -- dtto
t57 = t57a (BN (BN BV 1 BV) 3 (BN BV 5 BV))
```

```
t57a x = sorted x

-- datovy typ Union
{- v typovanom programovani nemoze funkcia vracat dva rozne typy.
Ak potrebujeme vracat tva typy, zlucime ich do jednoho typu, ktory bude
rozlisovat skutocny pripad pomocou svojich konstruktorov (alias tagov).
Odpoveda to variantnemu zaznamu z Pascalu. Nie je to union z C, pretoze
NEDOVOLUJE pretypovanie - nema dobry zmysel
-}
-- data Union a b = Left a -- preddefinovany
--                | Right b
{- priklad pouzitia:
vypocet s chybou: namiesto  f:: a->b pocitame f:: a-> Union b String
kde String je popis chyby. Na typ Maybe sa mozeme pozerat podobne: No oznacuje
chybu (bez identifikacie druhu chyby). Taketo vyvolavanie a osetrovanie chyb
musime robit sami: case podla vysledku odpoveda osetreniu chyby (resp.
zisteniu, ci chyba nastala), predavanie (vyslednych) hodnot musi byt vnorene
do noveho typu (pomocou Left alebo Yes), vyvolanie chyby je explicitne (No, Right)
-}
-- priklad : vkladanie do 2-3 stromu: z rekurzie sa vracia bud strom alebo
-- dva stromy a prvok, ktory musime zaradit vyssie

-- cvicenie: 14.1.2000
-- typove triedy
{- rovnost stromov mozeme definovat klasicky ako rovnost datovych struktur
pre ucely cvicenia zoslabime poziadavky a chceme definovat triedu rovnosti
stromov (Bt), ktora bude vyjadrovat ekvivalenciu stromov aj pri prehodeni
podstromov
-}
eqBt BV     BV = True
eqBt (BN l1 x1 r1) (BN l2 x2 r2)=
  x1== x2 &&  -- nerekurzivne porovnavanie prvkov
  (eqBt l1 l2 && eqBt r1 r2) ||
  (eqBt l1 r2 && eqBt l2 r1)
eqBt _    _ = False

instance (Eq a)=> Eq (Bt a) where
  s1 == s2 = s1 `eqBt` s2
{- typove triedy nedovoluju pretazenie v zmysle parametrickeho polymorfizmu.
Na jednom type moze byt len jedna definicia rovnosti
-}

-- testy
tstr58 = (BN (BN (BN BV 21 (BN BV 31 BV)) 11 (BN BV 22 BV)) 1
```

```
    (BN (BN BV 25 BV) 15 (BN BV 26 (BN BV 35 BV)) ))
t58 =  (BN (BN BV 21 (BN BV 31 BV)) 11 (BN BV 22 BV)) ==
    (BN (BN BV 22 BV) 11 (BN BV 21 (BN BV 31 BV)) )
t59 =  (BN (BN BV 21 (BN BV 31 BV)) 11 (BN BV 22 BV)) ==
    (BN (BN BV 22 BV) 11 (BN (BN BV 31 BV) 21 BV))

-- lazy d.s.
-- Erathostenovo sito
sito (p:ps) = p: sito (filter (\x -> x `mod` p /= 0) ps)
sito1 (p:ps) = p: sito1 (filter ((`mod` p)>.>(/= 0)) ps) -- varianta: zlozenim fci
t60 = sito [2..]
t61 = sito [3..]
t62 = sito1 [2..]
t63 = take 20 t60
t64 = take 1000 t60
t65 = take 1000 (zip [1..] t60)
{- ciele t60 az t62 generuju nekonecnu d.s., v cieloch t63 - t65 z nej vyberieme
len konecnu cast danej dlzky.
postup vypoctu: pre jednoduchost oznacim: f p = filter(\x -> x `mod` p /= 0)
  sito (2:[3..]) -> -- vystup ziada hlavny konstruktor, preto:
  2: sito(f 2 [3..]) ->  -- pattern matching v sito ziada konstruktor
  .. sito(3: f 2 [4..]) -> -- sito sa vykona, (3:) sa vypise
  .. 3: sito(f 3 (f 2 [4..])) ->
  .....              [5..]   ->
  .....          (5:f 2 [6..]) ->
  ..... 5: sito(f 5 (f 3 (f 2 [6..]))) -> atd, vnutorna struktura sa
stale zvacsuje
-}
-- poznamka k nekonecnym d.s.
t66 = filter (const False) [1..]
-- dtto: filter (\x -> False) [1..]
{- takyto ciel sa zacykli (v konstantnej pamati), pretoze system nedokaze
zistit z nadhladu (dokazom), ze pre ZIADNY prvok uz nebude platit podmienka.
System HLADA prvy prvok (alebo koniec nekonecneho zoznamu). Pretoze prvky
vstupneho zoznamu, ktore uz spracoval, sa nevyskytuju v jeho aktivnych behovych
strukturach, moze ich gc pozbierat -> konstantna (aktivna) pamat

Mozne spracovanie nekonecneho vysledku: pretoze spracovanie po prvkoch nie
je obecne vhodne (napr. pomocou filter), potrebujeme funkciu, ktora vedome
zahodi cely (nekonecny) zbytok d.s. Takato funkcia takeWhile je definovana
v prelude.hs a berie zo vstupneho zoznamu prvky, kym plati podmienka p.
Ak najde prvok, kde p neplati, vrati prazdny zoznam, bez vyhodnocovania
zbytku zoznamu (a jeho prehladavania)
```

```
takeWhile           :: (a -> Bool) -> [a] -> [a]
takeWhile p []       = []
takeWhile p (x:xs)
 | p x       = x : takeWhile p xs
 | otherwise = []

V nasom pripade pripade, ak chcem prvocisla do 1000 vcetne:
-}
t66a = takeWhile (<=1000) t60

-- priklad: rozvrstvenie binarneho stromu:
{- Mam binarny strom (ako vyssie), chcem ho prehladavat do sirky v tomto zmysle:
vystup bude zoznam vrstiev, n-ta vrstva bude zoznam vrcholov v hlbke n
-}
-- prvy (chybny) pokus:
vrstvy1 BV = []
vrstvy1 (BN l x r) = [x]:zipWith (++) (vrstvy1 l) (vrstvy1 r)
{- rekurzivne volanie vyrobia zoznam vrstiev podstromu, tieto dva zoznamy musime
skombinovat pomocou zipWith, pricom kombinacia vrstiev je spojenie (++) zoznamov.
Na zaciatok pridame vrstvu s jednym prvkom x, co sposobi tiez posunutie vsetkych
ostatnych vrstiev o jednu hlbsie.
pozn.: Ak by sme na zaciatok pridali namiesto [x]: len x: , system ohlasi typovu
chybu (na rozdiel od Prologu)
- Chyba: zipWith je napisany tak, ze vysledok skrati na dlzku kratsieho vstupneho
zoznamu. V nasom kontexte to znamena, ze kym je strom uplne rozvetveny, vystup
je spravny, ale ak sa niekde objavi BN, potom sa vystup  na tejto urovni usekne.
Mozne odstranenia chyby: Ak zmenime definiciu takto:
  vrstvy1 BN = [[]]
nepomozeme si, pretoze, ak strom bude velmi sisaty, aj tak sa bude urezavat.
Mozne riesenie tohoto druhu je zaviest zoznam prazdnych zoznamov (fce nils),
ktory bude generovat potrebne mnozstvo prazdnych zoznamov. Toto je riesenie, ked
zmenime data. (tieto dodatocne data sa daju podstrcit aj na inych miestach
v programe: ktorych?
  V tomto pripade je vysledok nekonecny zoznam, v ktorom len niekolko prvych
vrstiev je neprazdnych. Pomocou takeWhile vyberieme tieto vrstvy, viz t69.
  Iny princip opravy chyby je zmena programu, konkretne funkcie zipWith
na neskracujucu variantu. Kym mame obidva zoznamy neprazdne, tvorime vysledok
klasicky, az jeden dojde, kopirujeme zvysok. Vsimnite si zmeny typu fce zipWithL.
-}
{- prelude.hs :
zipWith                  :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)   = z a b : zipWith z as bs
zipWith _ _      _        = []
-}
```

```
--
vrstvy2 BV = nils
where nils = []:nils
vrstvy2 (BN l x r) = [x]:zipWith (++) (vrstvy2 l) (vrstvy2 r)

zipWithL                  :: (a->a->a) -> [a]->[a]->[a]
zipWithL z (a:as) (b:bs)   = z a b : zipWithL z as bs
zipWithL _ []      bs        = bs
zipWithL _ as      []        = as

vrstvy3 BV = []
vrstvy3 (BN l x r) = [x]:zipWithL (++) (vrstvy3 l) (vrstvy3 r)

t67 = vrstvy2 tstr58 -- nekonecny
t68 = vrstvy3 tstr58
t69 = takeWhile (/=[]) t67
-- n-arne stromy
data NTree a = Fork a [NTree a]
{- ukoncenie datoveho typu je prazdnym zoznamom, takze nepotrebujeme
ziadny nularny konstruktor -}
{- Napiseme funkciu fold alias univerzalny prechod strukturou:
fold pracuje na (NTree a) a konstruuje vysledky typu b, (jediny) funkcionalny
parameter je typu :: (a -> [b] -> b) , pretoze na jednej urovni sa spracuva
prvok vo vrchole :: a, spracovany zoznam podstromov :: [b] (pretoze spracovany
podstrom je :: b) a vysledok je :: b (ako vysledok podstromov)
-}
-- foldNT :: (a->[b] -> b) -> NTree a -> b
-- foldNT f (Fork a subtr) = f a (map (foldNT f) subtr)
                          -- f sa pouzije na spracovane podstromy
                             --   map spracuje zoznam podstromov
     --     na zoznam vysledkov
    -- foldNT f je spracovanie jednoho podstr.

-- ekvivalentne:
foldNT :: (a->[b] -> b) -> NTree a -> b
foldNT f = ff
  where ff (Fork a subtr) = f a (map ff subtr)

-- aplikacie:
size = foldNT (\a bs -> 1+sum bs)
depth = foldNT (\a bs -> 1+max0 bs)
  where max0 [] = 0 -- dolezite dodefinovanie
        max0 (b:bs) = foldl max b bs -- maximum zoznamu
```

```
tstr70 = Fork 1 [Fork 21 [Fork 31 [],Fork 32 [Fork 42 []]],Fork 25 [], Fork 26 []]
t70 = size tstr70
t71 = depth tstr70

-- aplikacia foldNT: vrstvy:
vrstvyN nt = foldNT (\a bs-> [a]:foldl (zipWithL (++)) nils bs) nt
where nils = []:nils
{- vysvetlenie:
foldNT (\a bs-> [a]:foldl (zipWithL (++)) nils bs) nt
foldNT ... nt  -- je volanie na vstupny strom nt
  \a bs ->...  -- je fce, ktore spracovava jedno poschodie NTree
     [a]: ... -- vyrobim nove patro, vrstvy z podstromov spracujem
foldl _ _ -- spracovava zoznam podstromov
  (..) nils -- nils je zoznam prazdnych vrstiev
ku ktoremu sa pripajaju vrstvy dalsich podstromov
(je to neutr. prvok zipWithL (++), az na nekonecno)
  (zipWithL (++)) -- spoji zoznam vrstiev prveho a
(uz spojenych) zbylych podstromov
    (++)  -- spojenie dvoch vrstiev
-}

t72 = vrstvyN tstr70
t73 = takeWhile (/=[]) t72

-- farbenie grafu: neladene (kombinatoricke prehladavanie)
{- Hm. Pomocou list comprehensions
graf je zoznam trojic: cislo vrcholu, domena povolenych farieb, minuly susedia
:: [(Int,Int,Int)]
idea farbenia: chcem vydat vsetky mozne ofarbenia v zozname. Ofarbenie je
zoznam dvojic (vrchol,barva)::(Int,Int). Pre pociatocny usek vrcholov z grafu
budem generovat vsetky ciastocne spravne ofarbenia do zoznamu. Spracovanie
jednoho vrcholu grafu bude znamenat, ze tieto ofarbenia  rozsirim vsetkymi
moznymi sposobmi pridanim farby aktualneho vrcholu.
Ak som ofarbil vsetky vrcholy, tak ciastocne ofarbenia su uplne a vydavam ich
na vystup.
Graf nebyva obvykle zadany s domenami, ale pomocou map ho jednoducho upravime
-}
{-
barveni castecne [] = castecne  -- cely graf ofarbeny, konec
barveni castecne ((v,dom,vs):gr)= barveni nove_castecne gr
  where nove_castecne= concat castecne1
-- castecne1 vydava rozsireni jednoho obarveni v jednom zozname,
-- musime vsetky tieto zoznamy pospajat
castecne1 = [(v,bv):barv|barv<-castecne,bv<-dom,bezpecne barv bv vs]
```

```
-- pre vyber farby som zvolil generuj a testuj, fce bezpecne testuje
-- ci je rozsirenie o (v,bv) mozne
-- predpokladam spravne usporiadanie v zozname susedov
        bezpecne _ _ [] = True
        bezpecne ((v1,b1):bs) bv (v:vs) =
          if v=v1 then  -- v1 je sused
b1/=bv && bezpecne bs bv vs -- farby sa lisia a rekurzia vyjde
   else bezpecne bs bv (v:vs) -- v1 neni sused, susedov nezmensujem
-}
{- ina moznost barveni v castecne1 je vybrat susedov, vybra ich farby, vypustit
ich z domeny v a potom uz len rozgenerovat vsetky ostavajuce moznosti
viz: n kraloven
-}

-- R. Bartak 11.1.2000, ladene
-- STRUCNE SEZNAMY (list comprehensions)

-- [ f x | x <- xs ]
-- stejne jako zapis mnoziny - je potreba specifikovat zdroj, v tomto
-- pripade seznam
-- map f xs = [ f x | x <- xs ]

-- Priklad: [ (x,y) | x<-[1,2], y<-[3,4] ]; vysledkem je [(1,3),(1,4),(2,3),(2,4)]

quicksort   []   =  []
quicksort (x:xs) =  quicksort [ y | y<-xs, y<x]
                 ++ [x]
                 ++ quicksort [ y | y<-xs, y>x]

-- pozor, vsechny prvky seznamu musi byt ruzne
-- DCV: opravit tak, aby fungovalo i pro seznamy s vicenasobnym vyskytem
-- tehoz prvku
--      udelat insertsort, mergesort

insertsort   []   = []
insertsort (x:xs) = insert x (insertsort xs)
  where insert x   []            = [x]
        insert x s@(y:ys) | x>y  = y : insert x ys
                          | x<=y = x : s

mergesort []  = []
mergesort [x] = [x]
mergesort  s  = merge (mergesort ls) (mergesort us)
  where (ls,us) = halve s
```

```
merge    xs         []            = xs
merge    []         ys            = ys
merge sx@(x:xs) sy@(y:ys) | x<y  = x : merge xs sy
                          | x>=y = y : merge sx ys

halve    []      = ([],[])
halve    [x]     = ([],[x])
halve (x1:x2:xs) = (x1:ls,x2:rs)
  where  (ls,rs) = halve xs

-- nekonecne seznamy
ones = 1:ones
numsFrom n = n : numsFrom (n+1)  -- lze zapsat take takto (\n -> [n..])

-- diky linemu (lazy) vyhodnocovani se vyhodnocuji pouze prvky seznamu,
--  ktere jsou potreba
-- numsFrom 3 !! 2 vrati 5 (!! n vraci n-ty prvek, pocitano od nuly)

fiboList = 1 : 1 : [ a+b | (a,b) <- zip fiboList (tail fiboList) ]

-- DCV zipWith

gzipWith f  []      _    = []
gzipWith f   _      []   = []
gzipWith f (x:xs) (y:ys) = f x y : gzipWith f xs ys

-- Pozor! nejde zapsat takto (\f xs ys -> [ f x y | x <- xs, y <-ys])
-- klasicky zip = zipWith (\x y -> (x,y))

factList1 = (0,1) : [ (n+1,(n+1)*fn) | (n,fn) <- factList1 ]
factList2 = 1 : 1 : [ (1+fnn/fn)*fnn | (fn,fnn) <- zip factList2
                                                     (tail factList2) ]

-- mohou byt i nekonecne seznamy funkci
hanoiList = (\from to via -> [(from,to)])
          : [ (\from to via -> (prev from via to) ++ [(from,to)]
              ++ (prev via to from)) | prev <- hanoiList ]

hanoi n = (hanoiList !! (n-1)) 1 2 3

-- DCV udelat obecne s funkci pro presun a funkci spojujici presuny
hanoiListG move join = (\from to via -> move from to)
                     : [ (\from to via ->
```

```
                         join (join (prev from to via) (move from to))
                              (prev via to from))
                       | prev <- hanoiListG move join ]

-- hanoiList = hanoiListG (\x y -> [(x,y)]) (++)

-- N-KRALOVEN aneb jak udelat prohledavani
-- definuji se vsechny moznosti, diky linemu vyhodnocovani se pocita
--  jen co je potreba
-- vystupem je seznam vsech reseni, kde reseni je seznam umisteni
--  kraloven, kazde umisteni ma tvar (sloupec, radek)

check (c1,r1) (c2,r2) = r1/=r2 && c1-r1/=c2-r2 && c1+r1/=c2+r2

safe [] _ = True
safe (q1:qs) q2 = check q1 q2 && safe qs q2

-- rekurzi
queensR n = queens n 1
  where queens n c | c>n  = [[]]
                   | c<=n = [(c,r) : restq
                              | restq <- queens n (c+1),
                                r <- [1..n], safe restq (c,r) ]

-- akumulator
queensA n = queens [[]] n 1
  where queens fin n c | c>n  = fin
                       | c<=n = queens [(c,r) : restq
                                         | restq <-fin, r <- [1..n],
                                           safe restq (c,r) ] n (c+1)


-- pouziti dopredne kontroly (forward checking)
-- pro kazdou kralovnu drzime seznam pozic, ktere jsou bezpecne s jiz
--  umistenymi kralovnami

-- nejprve vytvorime domenu pro jednu kralovnu (domena=seznam moznych
--  pozic ve tvaru (sloupec,radek)
-- bereme, ze kralovna je pevne umistena ve sloupci

q_domain c n = [ (c,i) | i <- [1..n] ]

-- vytvorime domeny pro vsechny kralovny
```

```
qs_domains n = [ q_domain i n | i <- [1..n] ]

-- vyber bezpecnych pozic z domeny kralovny vzhledem k dane (umistene) kralovne

propagate q [] = []
propagate q (qt:qts) = if check q qt then qt:propagate q qts
                                     else propagate q qts

-- vyber bezpecnych pozic sady kraloven vzhledem k dane kralovne + test
--  neprazdnosti vsech domen

fc q [] = (True,[])
fc q (qts:rest) = if domain==[] then (False,[])
                                else (rest_test, domain : rest_domains)
  where domain = propagate q qts
        (rest_test, rest_domains) = fc q rest

queensFC n = queens n (qs_domains n)
  where queens   0   []        = [[]]
        queens (i+1) (qd:doms) = [ q:qs | q <- qd,
    (True,new_doms) <- [fc q doms],
  qs <- queens i new_doms ]

-- DCV umistovat vzdy nejdrive kralovnu s nejmensi domenou

-- z prednasok: 1998,2000

-- Faktorial 4x
-- fact neoÜetouje záporné hodnoty
fact3 :: Int -> Int
-- klasický zápis
fact3 n = if n==0 then 1
          else n*fact3 (n-1)

-- strá?e
fact2 n | n==0 = 1
        | n>0  = n*fact2 (n-1)

fact4 n|n==0=1|n>0=n*fact4(n-1)

-- porovnávání se vzorem (pattern matching)
fact1 0     = 1
fact1 (n+1) = (n+1)*fact1 n
```

```
fact5 n = fact_iter n 0 1
where fact_iter n k f
               | n==k = f
       | True = fact_iter n (k+1) ((k+1)*f)
-- test
fact6 n = fact_iter n 0 [(n,0,1)]
where fact_iter n k f@((_,_,ff):_)
               | n==k = f
       | True = fact_iter n (k+1)
                        ((n,k+1,(k+1)*ff):f)

length1 :: [a] -> Int
length1  []    = 0
length1 (x:xs) = 1 + length1 xs

head1 :: [a]->a
head1 (x:xs) = x

tail1 :: [a] ->[a]
tail1 (x:xs) = xs

and1 :: Bool -> Bool -> Bool
and1 True True = True
and1 _    _    = False

and2 :: (Bool,Bool) -> Bool
and2 (True,True) = True
and2 (_   ,_)    = False

-- Na cvicenia:
-- nty n zoznam -- vyberie n-ty prvok zo zoznamu; ak neex. vrati []
nty 1     (x:xs) = x
nty _     []     = []
nty (n+1) (_:xs) = nty n xs

-- take n zoznam -- vyberie prvych n prvok zo zoznamu;
--                   ak zoznam nie je dost dlhy, vrati ho cely
take1 0     _      = []
take1 _     []     = []
take1 (n+1) (x:xs) = x : take1 n xs

take2 n xs = case (n,xs) of
(0,_)     -> []
(_,[])    -> []
```

```
(n+1,x:xs) -> x : take2 n xs

-- zip1 z1 z2 -- z dvoch zoznamov prvkon vyrobi zoznam dvojic
--              zoznamy musia byt rovnako dlhe
zip1 [] [] = []
zip1 (x:xs) (y:ys) = (x,y) : zip1 xs ys

-- zip2 z1 z2 -- z dvoch zoznamov prvkon vyrobi zoznam dvojic
--              zoznamy mozu byt rozne dlhe, skonci, az jeden
--                zoznam nepokracuje
zip2 (x:xs) (y:ys) = (x,y) : zip2 xs ys
zip2 _      _      = []

-- tiedenia
-- qsort bez opakovania
qsort1 []   = []
qsort1 (x:xs) = qsort1 (mensi x xs) ++ [x] ++ qsort1 (vetsi x xs)
         -- operator ++ je append
mensi _ [] = []
mensi x (y:ys) = if y < x then y : mensi x ys
                          else mensi x ys
vetsi _ [] = []
vetsi x (y:ys) = if y > x then y : vetsi x ys
                          else vetsi x ys

-- s opakujucimi sa prvkami
qsort2 []   = []
qsort2 (x:xs) = let (ys,zs) = split x xs
in  qsort2 ys ++ [x] ++ qsort2 zs
split x [] =            ([],[])
split x (y:ys) | y<x  = (y:z1,  z2)
               | True = (  z1,y:z2)
where (z1,z2)= split x ys  -- este nebolo

-- mergesort asi spravim na prednaske
-- msort1

-- fibonacci
fib2 n |  n == 1 || n == 2  = 1 -- || je spojka or
       |  n>2               = fib2 (n-1) + fib2 (n-2)

fib4 1 = 1
fib4 n = fib_iter n 1 1 -- invariant fib_iter (n-k+2) f_k f_k-1
fib_iter  2    a b = a
```

```
fib_iter (n+1) a b = fib_iter n (a+b) a

------------------
{- testy -}
t1::[(Int,Int)]
t1 = [ (n,fact1 n) | n <- [1..6]]
t2 = (length1 [n|n<-[1..6]], length1 ('b':['a']) )
t3,t4::[Int]
t3 = qsort1 [5,6,3,7,2,8,1,9,4,0,1,1]
t4 = qsort2 [5,6,3,7,2,8,1,9,4,0,1,1]
t5::[(Int,Int,Int)]
t5 = [(i, fib2 i, fib4 i)| i<- [1..10]]
{- -}
------------------

type Complex = (Float,Float)
cadd :: Complex -> Complex -> Complex
-- cadd ma ve skutecnosti obecnejsi typ
cadd (rx,ix) (ry,iy) = (rx+ry,ix+iy)

data Tup2 a = Tup (a,a)
type Complex2 = Tup2 Float
c2add (Tup (rx,ix)) (Tup (ry,iy)) = Tup (rx+ry,ix+iy)

infix 4 `p1`
x `p1` y = x

a1 n = let suda x = 0: licha x
           licha x = 1:suda x
in take n (suda 0)

a2 n = let suda x | x> 4 = suda 0
                  | True  = (0,x): licha (x+1)
           licha x = (1,x):suda (x+1)
in take n (suda 0)
a3 x y =let norma (x,y) = sqrt (x^2+y^2)

    prumer x y = (x+y)/2.0
 in prumer (norma x) (norma y)

t6 :: Float
t6 = a3 (1.0,0.0)(4.0,4.0)

map4 f xs = [f x | x<-xs]
```

```
data Tvyraz a = Con a
  | Var String
  | Tvyraz a :+: Tvyraz a
  | Tvyraz a :*: Tvyraz a

data Tvyraz2 a = Con2 a
  | Var2 String
  | Plus (Tvyraz2 a) (Tvyraz2 a)
  | Krat (Tvyraz2 a) (Tvyraz2 a)
  | Fce String [Tvyraz a]

t7:: Tvyraz Int
t7 = (Con 2 :*: Var "x") :+: Con 1
t8:: Tvyraz2 Int
t8 = Plus (Krat (Con2 2) (Var2 "x")) (Con2 1)

dbl xx@(x:xs) = x:xx
map_m f m = map (map f) m

map2 f x y = map (\(x,y)-> f x y)  (zip x y)
map_c f c x = map (\x-> f c x) x
map_c2 f c x = map2 f (map (\_->c) x) x

data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
mapTree :: (a->b) -> Tree a -> Tree b
mapTree f (Leaf a)     = Leaf (f a)
mapTree f (Branch l r) = Branch (mapTree f l)
                                (mapTree f r)
t9::Tree Int
t9 = mapTree (1+) (Branch (Leaf 3) (Leaf 4))

-- prvocisla
delitele n = [d| d<-[1..n], n `mod` d ==0]
prv1 p = delitele p == [1,p]
t10::[Int]
t10 = [p| p<-[1..10000], prv1 p]

lichacast p = if p`mod`2==0then lichacast (p/2)else p
umocni p 1 = p
umocni p n = if n`mod`2==1then p*umocni (p*p)((n-1)/2)else umocni (p*p)(n/2)

---------------------------------------------
```

```
{- pokusy -}

fact n = foldr (*) 1 [i| i <- [1..n]]
pow2 n = foldr (*) 1 [2| i <- [1..n]]
sum2 n = foldr (+) 0 [i| i <- [1..n]]

--zip2 (x:xs) (y:ys) = (x,y) : zip2 xs ys
--zip2  xs     ys    = []

zip2_ x y | x/=[] && y/=[] = (head x, head y) : zip2_ (tail x) (tail y)
          | True           = []

test :: [(Int,Char)]
test = zip2 (1:2:3:[]) ['a','b']
```