Funkcionální programování
Haskell

Jan Hric, KTIML MFF UK,1997-2018d

http://ktiml.ms.mff.cuni.cz/~hric/

Jan.Hric@mff.cuni.cz

Funkcionální programování

Programování s pomocí funkcí

- program = definice funkcí
- výpočet = aplikace funkce na argumenty
  - matematické funkce, bez vedlejších efektů
  - datové struktury: lambda-termy

β-pravidlo: $(\lambda x.M)N \rightarrow M[x:=N]$

- výsledek: vyhodnocený tvar, normální forma (pokud existuje)

Dnešní jazyky integrují části FP (nějak)

Funkcionální programování

Historicky:

- **lambda-kalkul, cca. 1930, spíš teorie vyčíslitelnosti**
- **LISP, 1959-1960**
- **Scheme 1975**
- **ML 1985 typovaný jazyk**
- **Haskell 1989**
- **.. A další: Erlang (pro realtime aplikace), Scala (kompiluje do JVM), Clojure, OCaML**

Funkcionální programování

Zdroj: http://www.haskell.org -> impl. GHC Glasgow Haskell Compiler (ghci); Hugs (WinHugs)

Literatura:

- **Richard Bird, Philip Wadler, Introduction to Functional Programming, Prentice Hall, New York, 1988**

  jazyk Miranda

- **Simon Thompson: Haskell: The Craft of Functional Programming, Addison-Wesley, 1997**
- **Graham Hutton: Programming in Haskell, Cambridge University Press, 2007**

Funkcionální jazyky obecně

- **Pracují s lambda-termy (jen se jinak zapisují)**
  - **Funkce a složené hodnoty jsou „občany první kategorie“**
  - **Výsledek je případně složený**
    - **(seznam, (synt.) strom, graf, funkce …)**
  - **Funkce můžu tvořit za běhu a vracet (anonymní lambda-funkce)**
  - **Funkce se předávají jako parametry do jiných funkcí**
    - **Obecná parametrizace, ! (ta nejobecnější)**
  - **Konstrukce funkcí je laciná -> jiný styl programování**
    - **Skládání funkcí, kompoziční programování, rekurze**
    - **Mnoho malých funkcí**
  - **Matematické proměnné: označují hodnotu, ne místo v paměti**
    - **Immutable struktury, neměnné (pro novou hodn. zavedu nové jméno)**
  - **Na rozdíl od Prologu: argumenty funkcí jsou „vstupní“, vyhodnocovaný term nemá volné proměnné**

Jazyk Haskell

- **Funkcionální jazyk**
  - **Silně typovaný: každý výraz má typ**
    - **Typy chrání proti (některým) chybám**
  - **Staticky typovaný: při překladu**
    - **polymorfizmus, parametrický (vs. podtypový v C/C++)**
    - **implicitní typování (systém si typy odvodí) – uživ. je (většinou) nepíše**
    - **(při běhu už typy nejsou potřeba)**
  - **Líné vyhodnocování**
    - **Umožňuje nekonečné datové struktury**
    - **Argumenty se vyhodnocují, až/pokud jsou potřebné (call-by-need)**
    - **Trade-off: space leak**
  - **Čistý jazyk (bez sideefektů)-> referenční transparentnost**
    - **V/V se řeší pomocí monád (ty určují pořadí operací)**
  - **Jednoduchá syntax, jednoduchá sémantika -> jednoduché manipulace s programy**

Jazyk Haskell

... další rysy

porovnávání se vzorem - pattern matching

Stručnější program, má méně selektorů

funkce vyšších řádů (higher-order), anonymní lambda funkce

uživatelsky definované datové typy, datové konstruktory

typové třídy (uživatelsky definované), přetížené funkce (overloading)

lexikální rozsahy platnosti

„dvourozměrná“ syntax (indentace) – 2D layout

operátory (binární)

moduly (N/I)

- **jiný styl programování než OOP**
  - **Zpracování celých struktur najednou**
  - **Naprogramujete obecné vzory rekurze a ty pak parametrizujete podle potřeby**

Základy

interpretační cyklus:

čti - vyhodnoť - vytlač

(read - eval – print loop, REPL)

vyhodnocení výrazu

každý výraz má hodnotu (a typ)

zdrojový text v souboru

-- komentáře do konce řádku

{- vnořené komentáře -}

prelude.hs --soubor načítaný při spuštění, předdefinované funkce ...

prompt >

> :quit -- :q

> :? help, seznam příkazů

> :load "myfile.hs" načtení souboru (a kompilace)

> :type map -- vypíše typ výrazu

```
map :: (a -> b) -> [a] -> [b]
```

> :set +t nastavení přepínačů, např. výpis typu

Skripty a interakce

```
> ”Hello, world!”
”Hello, world!” :: String
> 3*4
   12
delka = 12
kvadrat x = x*x
plocha = kvadrat delka
> kvadrat (kvadrat 3)
   81
```

Kontext, prostředí - obsahuje dvojice: proměnná a hodnota

Zpracování zdrojáku je víceprůchodové: hodnota nebo funkce nemusí být definována před použitím

Výrazy a hodnoty

redukce

výrazu podle pravidla-definice fce

```
kvadrat(3+4) =>(3+4)*(3+4) -- kvadrat
             => 7*(3+4)    -- +
             => 7*7        -- +
             => 49         -- *
```

líné vyhodnocování: volání funkce se nahrazuje tělem, za formální argumenty se dosadí (nevyhodnocené) aktuální arg., arg. se vyhodnocuje pouze pokud je potřeba (pro výstup, pattern matching, if, case, primitivní fce (např. aritmetické), …)

pouze část, která je potřeba (př: aspoň dvouprvkový seznam)

při uvažování o programech spec. hodnota $\perp$ [bottom]: nedefinováno – chyba, nekonečný výpočet: `undefined`

Hodnoty a typy

`5 :: Int` – příklady zápisu hodnot a jejich typů

`10000000000 :: Integer` – dlouhá čísla

`3.0 :: Float`

`'a' :: Char '\t','\n','\\','\'','\"'`

`True :: Bool ; False :: Bool` – v prelude

`[1,2,3] :: [Int], 1:2:3:[]::[Int]`

`"abc" :: [Char]` – řetězce, `String`

`(2, 'b') :: (Int, Char)` – dvojice, n-tice

`succ :: Int -> Int` – typ funkce

`succ n = n+1` – definice funkce

`*špatně [2,'b']` – hodnoty v seznamu musí mít stejný typ -> kompilátor odmítne

Typy 1

Základní: Int, Integer (neomezená přesnost), Bool, Char, Float, Double

Konkrétní typy (i uživ. definované): začínají velkým písmenem

Složené:

n-tice (1,'a'):: (Int,Char), (x,y,z) ... speciálně ( ) :: ( )

idea použití () - dummy: date :: () -> String; write :: a -> () --nepřesné

seznamy: `[a] = [] | a : [a]` -- *pseudokód*

typy funkcí: `Int -> Int`

(uživatelsky definované typy a typové konstruktory: Tree a, Assoc a b, …)

(Typové) proměnné: a, b, ... v popisu typu, zač. malým písmenem

Implicitní typování:

! uvádět typ není nutné, ale odvozuje se (a kontroluje se)

Nekonzistentní typy => typová chyba, při překladu (tj. při :load)

Motivace zavedení typů

- **Ochrana proti některým druhům chyb**
- **Ale: typy (v Hs) neodchytí zvláštní sémantiku hodnot:** `1/0, head []`

Typy 2

- **Suma – součet seznamu, s prvky typu `Int`**

```
suma::[Int]->Int --  nepovinné určení typu
suma [] = 0      --  pattern matching
suma (x:xs) = x + suma xs  --dtto, x je hlava, xs tělo
```

- **Délka seznamu (s prvky libovolného typu)**

```
length::[a]->Int  --  polymorfní typ, a je typová prom.
length [] = 0
length (x:xs) = 1 + length xs
```

  - **Typová proměnná `a`: struktura parametru x není využívána anebo je předán jako celek dál (např. do funkcionálních argumentů)**

(později: typové třídy – def. přetížených fcí, stejné jméno fce. pro různé typy)

```
(==) :: Eq a => a -> a -> Bool
```

Převody typů

Nutné explicitní volání převodních funkcí

Hodnoty (např. čísla) se nepřevádí mezi typy automaticky

Převodní funkce je také hint kompilátoru, protože má konkrétní typ

```
toEnum :: Int -> Char   -- předdefinováno, zjednoduš. (přetížení)
fromEnum :: Char -> Int -- lze def. pro výčt. typy

ord = fromEnum      --
chr = toEnum
offset = ord 'A' - ord 'a'
capitalize :: Char -> Char
capitalize ch = chr (ord ch + offset)

isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')
```

Funkce

Definice v souboru - skriptu

funkce se *aplikuje* na *argumenty* a vrací *výsledek*

curried forma (podle Haskell B. Curry)

- **“argumenty chodí po jednom”**
  - **funkce dostane jeden argument a vrací funkci v dalších argumentech**
- **f :: Int -> Int -> Int**
  - **-> v typu je asociativní doprava: f :: Int -> (Int -> Int)**
  - **místo: f :: (Int, Int) -> Int**
  - **f (3,4) = ... -- platný kód, nepreferovaný tvar**
    - **Vlastně funkce jednoho argumentu, a to dvojice**
- **f 3 4, (f 3) 4 => ...**
  - **Volání funkce je asociativní doleva: (f 3) 4**
- **typy: pokud `f :: a -> b, x :: a`, potom `f x :: b`**

Anonymní funkce (stručně)

Anonymní funkce (lambda funkce)

Zpětné lomítko \ před argumenty, -> mezi arg. a tělem

```
succ = \ x -> x+1

mkDatum = \ d m r -> (d,m,r)
```

Definice jsou pouze ukázka, ve skriptu nepoužíváme

Typické použití lambda funkce: jako funkcionální parametr při volání

```
> map (\x->x+2) [1,2,3]   -- nová fce
> map succ [1,2,3]        -- ale známá fce
> opBodoveMat (\x y->x+y) m1 m2
```

Definice funkcí 1

-- definice funkcí začínají v 1. sloupci

-- jednořádková

```
prumer x y = (x+y)/2.0
```

-- if then else: jako výraz, 3 klíčová slova

```
max1 :: Ord a => a -> a -> a
max1 x y = if x>=y then x else y
```

-- vždy s „else“ – i když podmínka neplatí, potřebujete vracet nějakou hodnotu (vs. stejný „stav“ v OOP)

Definice funkcí 2, pattern matching

-- porovnávání se vzorem, používaný styl

```
fact 0 = 1
fact n = n*fact (n-1)
```

-- klauzule/rovnosti procházíme shora, deterministicky první přijatá klauzule vrací pravou stranu jako výsl.

-- ve vzoru: proměnné a konstanty/konstruktory

-- argument se vyhodnotí do potřebné podoby, aby šlo rozhodnout, zda vzor přijmout nebo odmítnout

```
sum1 [] = 0
sum1 (x:xs) = x + sum1 xs
```

-- proměnné ve vzoru (x a xs) jsou definiční výskyty, nesmí se opakovat

Definice funkcí 3

```
-- styl bez porovnávání se vzorem, pomocí selektorů
sum2 xs = if null xs then 0 else
                head xs + sum2 (tail xs)
null :: [a] -> Bool
null [] = True
null _  = False -- anonymní vzor
tail :: [a] -> [a]
tail (_x:xs) = xs
-- tail []=[] --nepokrytý vstup,opravitelné
head :: [a] -> a
head (x:xs) = x
-- head []=?? --nepokrytý vstup, nemáme x::a
```

Definice funkcí 4, stráže

-- Stráže/guards: podmínka typu Bool

-- speciální syntax pro nejvyšší úroveň def. funkce

-- svislítka jsou zarovnaná (bude: 2D syntax)

-- stráže se vyhodnocují shora a první pravdivá (`True`) vrací svou pravou stranu

-- nemusí pokrývat všechny případy: nekontroluje se

-- poslední stráž je záchytná podmínka: `True` , `otherwise`

```
signum1 x
   | x > 0  = 1
   | x == 0 = 0
   | x < 0  = -1
```

Int a vestavěné funkce

:: Int -- typ celých čísel

běžné aritmetické operátory

+,*,^,-,div,mod :: Int -> Int -> Int –nepřesné, je přetížené

abs, negate:: Int -> Int

:: Integer – celá čísla s libovolnou přesností

Bool a vestavěné funkce

:: Bool , výsledky podmínek, stráží

```
== /= > >= <= < :: a->a->Bool
```
–nepřesné (1)

```
(&&), (||) :: Bool -> Bool -> Bool
```

Binární spojky and a or

```
not :: Bool->Bool
```

– (1) některé typy nelze porovnávat, např. funkce a typy funkce obsahující

! Pokud je výsledek funkce typu `Bool`, tak nepoužíváme podmínky, ale log. spojky

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs
```

☹ `and (x:xs) = if x then and xs else False`

Příklady

```
const :: a -> b -> a
const c x = c
```

Volání `const c :: b->a` je pro `c::a` konstantní funkce

```
> const 7 (1.0/0.0)  -- bez chyby kvůli línému vyhodnocení
  7
> const 7 undefined -- dtto, použití undefined
  7
```

```
length xs = sum(map(const 1)xs)
```

-- `const 1` je částečná aplikace, mění hodnotu `1` na fci `\x->1`

-- typický způsob FP: definice `length` neobsahuje explicitní rekurzi, ale skládáme předpřipravené fce

-- pozn: určení a/nebo zjednoznačnění typu ve výrazu: `const (1::Int)`

(Příklad)

```
fact3 :: Int -> Int
fact3 n = if n==0 then 1
          else n*fact3 (n-1)
> fact3 4
24

iSort :: [Int] -> [Int]
iSort [] = []
iSort (a:x) = ins a (iSort x)
ins :: Int -> [Int] -> [Int]
ins a [] = [a]              -- ins cmp a [] = [a]
ins a (b:x)                 -- ins cmp a (b:x)
   | a<=b       = a:b:x     -- zobecnění | cmp a b = …
   | otherwise = b : ins a x
```

s porovnávací fcí cmp: (i obecnější typ) - parametrizace kódem

ins :: (a->a->Bool)->a->[a]->[a] , iSort :: :: (a->a->Bool)->[a]->[a]

Lexikální konvence

identifikátory

- posl. písmen, číslice, ‘(apostrof), _(podtržítko)
- funkce a proměnné - začínají malým písm.
  - velké začáteční písmeno - konstruktory
  - vyhrazeno (klíčová slova):
    - case of where let in if then else data type infix infixl infixr primitive class instance module default ...

operátory

- jeden nebo víc znaků
  - `: ! # $ * + - . / \ < = > ? @ ^ |`
  - spec. : ~
  - symbolové konstruktory začínají znakem :
- identifikátor jako operátor v `(zpětný apostrof): 5 `mod` 2
- závorky pro neoperátorové (tj. prefixní) užití operátorů: (+), (+) 3 4

Skripty, Literate Haskell

Dva druhy skriptů:

1. **S příponou .hs - obvyklý**

   Definice funkcí v skriptu začínají na prvním řádku

   Používáme v této přednášce

2. **Literate Haskell**

   Přípona .lhs

   Za platný kód jsou považovány pouze řádky začínající znakem >

   Okolo řádků kódu musí být prázdné řádky

   Použití: soubor jiného určení (blog, TeX) je také platný Literate Haskell (typicky po změně přípony na .lhs)

Syntax - 2D layout

2D layout, offside pravidlo: definice, stráže (|), case, let, where ...

- **umožňuje vynechání separátorů a závorek**
  - **Protože máme operátory (které můžou být za výrazem), je těžké určit konec výrazu => potřebujeme závorky nebo konvenci**
- **zapamatuje se sloupec zač. definice fce**
  - **cokoli začíná víc napravo, patří do stejné definice (včetně prázdných řádků)**
  - **co začíná ve stejném sloupci, je nová definice ve stejném bloku**
  - **pokud něco začíná v levějším sloupci, ukončí to příslušný blok definic**

```
nsd n m
  | m==0       =
      n
  | n>=m       = nsd m (n `mod` m)
  | otherwise = nsd m n -- převedení na minulý případ
```

(!! Pozor na zacyklení při podm. `n>m`)

explicitní skupina v {}, jako separátor znak ‘;’

- **víc definic na jednom řádku**

```
let { x = 1; y = 2 } in …
```

Porovnávání se vzorem 2

Speciální vzory (neodmítnutelné):

- **vzor `_` podtržítko, na hodnotě nezáleží**

```
and1 :: (Bool,Bool) -> Bool  -- necurryované
and1 (True,True) = True  -- and není líné
and1 (_   ,_   ) = False
and2 (False, _ ) = False -- líná verze
and2 (_     , x ) = x  -- x se nevyhodnotí
```

- **vzor `@` přístup na celek (`xs`) i části (`x2:_`)**
  - **Matching `@` neselže, ale matching podčásti může selhat (a tím i vzoru)**

```
ordered (x1:xs@(x2:_)) =
     x1 <= x2 && ordered xs
ordered _                   = True
     -- default: pouze [x] a []
```

  - **Impl: použití vzoru @ ušetří stavbu nové struktury v těle funkce**
- **(vzor `n+k` *zastaralé*)**

Porovnávání se vzorem 3 – časté chyby

! Složené argumenty do závorek:

*** length x:xs = … -- pochopí se jako: (length x):xs =…**

! Proměnné ve vzorech se nesmí opakovat (na rozdíl od Prologu) – nutno explicitně porovnat pomocí (==)

- **Protože to jsou definiční výskyty proměnných**

- **Chybně:**

```
allEq (x:x:xs) = allEq (x:xs)
allEq  _       = True
```

- **OK:**

```
allEq (x1:x2:xs)= x1==x2 && allEq (x2:xs)
allEq  _        = True
```

Seznamy

```
data List a = []          --
            | a : (List a) -- ~cons ze Scheme
```

- ve skutečnosti vestavěný typ [a]
- `filter p xs` : vybere z `xs` všechny prvky splňující podm. `p`

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if p x then x:filter p xs
                         else   filter p xs
map :: (a->b)->[a]->[b]
map f [] = []
map f (x:xs) = f x : map f xs
> map (+1) [2,3,4]        -- [3,4,5] :: [Int]
> map (+) [2,3,4]         -- [(2+),(3+),(4+)] :: [Int->Int]
> map (const 1) "abc"     -- [1,1,1] :: [Int]
DC: zip :: [a]->[b]->[(a,b)]
```

Konstruktory seznamu

- Datové konstruktory jsou automaticky vytvořeny z definice typu

```
[]   :: [a]          -- polymorfní konstanta
(:)  :: a -> [a] -> [a]
-- data [a] = [] | a : [a]
```

  - Pseudokód (<-speciální syntax), lexikálně nesprávný
  - symbolové konstruktory musí začínat ':', jinak to jsou funkce
    - Ekvivalentní velkému písmenu u alfanumerických konstruktorů

konvence: `[1,2,3]` je výraz `1:2:3:[]`

Tj. `1:(2:(3:[])), 1:[2,3]`

nelze: `[1,2:xs]` – nutno použít zápis: `1:2:xs`

- `[1,2]:xs` má jiný význam, je typu `[[Int]]`

Standardní funkce (některé) 1

Soubor prelude.hs – se načte při spouštění hask. prostředí a obsahuje standardní funkce (a …)

```
(++) :: [a] -> [a] -> [a]
```
-- append

```
head :: [a] -> a
```
- pokud je arg. [], skončí chybou

```
tail :: [a] -> [a]
null :: [a] -> Bool
id   :: a -> a
fst  :: (a,b) -> a
```
-- pouze pro dvojice

```
snd  :: (a,b) -> b
(,)  :: a -> b -> (a,b)
```
-- spec. syntax, konst. dvojice, atd.

```
take :: Int -> [a] -> [a]
```
- `take n xs` **vrátí prvních *n* prvků *xs* nebo všechny, pokud je *xs* kratší**

```
drop :: Int -> [a] -> [a]
```
- **zahodí prvních *n* prvků**

```
elem :: Eq a => a -> [a] -> Bool
```
- **test výskytu prvku v seznamu (př: převod relací na funkce typu `Bool`)**

```
(!!) :: [a] -> Int -> a  -- v prelude
```
- **výběr n-tého, číslováno od 0**

Standardní funkce 2, higher-order

takeWhile:: (a-> Bool) -> [a] -> [a]

- **vrací úvodní prvky, pro které je splněna podmínka**

dropWhile:: (a-> Bool) -> [a] -> [a]

- **zahazuje úvodní prvky, pro které je splněna podmínka**

map :: (a->b) -> [a] -> [b]

- **Transformuje prvky 1:1 podle funkc. parametru**

filter :: (a->Bool) -> [a] -> [a]

- **Vrací všechny prvky, pro které je splněna podmínka**

Standardní funkce 3, zipWith

zip :: [a] -> [b] -> [(a,b)]

- **Páruje prvky na stejné pozici, tj. paralelní procházení**
- **Analogicky zip3, zipWith3 …**

zipWith :: (a->b->c) -> [a] -> [b] -> [c]

- **Paralelní procházení s obecnou funkcí, délka výsledku je délka kratšího**

```
zip xs ys = zipWith (\x y->(x,y)) xs ys
zip xs ys = zipWith (,) xs ys
zip = zipWith (,)
```

- **Jednořádková definice získaná specializací, typické pro FP**

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y :
                          zipWith f xs ys
zipWith f  _      _     = []
```

Souvislosti

- **Porovnejte:**

take :: Int -> [a] -> [a]

takeWhile :: (a-> Bool) -> [a] -> [a]

  - **Pro různé situace se hodí různé funkce, s různými druhy podmínek**
- **Funkce vyššího řádu umožňují psaní nových způsobů zpracování datových struktur (zde: seznamů)**
  - **Př: pole a for-cyklus, seznam a map**
- **Funkce pro zpracování seznamů: `filter, map, foldr, (unfold)`**
  - **Vybrání prvků, transformace prvků, zpracování seznamu na hodnotu (~strukturální rekurze), zpracování hodnoty na seznam (~rekurze podle výstupu)**
  - **(`reduce` v Scheme odpovídá `foldr`)**
- **Pozn: předávání relací jako charakteristických funkcí (v `takeWhile` unární relace), tj. funkcí do `Bool`**

příklad: použití specializace

- **Práce s vektory: sčítání, odečítání, násobení po bodech, násobení konstantou**
  - **Vektory (resp. matice) libovolné délky (resp. velikosti)**

```
plusV :: Num a => [a] -> [a] -> [a]
plusV  xs ys = zipWith (\x y-> x+y) xs ys
minusV xs ys = zipWith (\x y-> x-y) xs ys
kratV  xs ys = zipWith (\x y-> x*y) xs ys
```

  - **Vlastně „pozvednutí“ binární operace na zpracování dvou seznamů**

```
kratCV c ys  = map (\y->(c*y)) ys
```

  - **Násobení konstantou: jiný vzor: používá jen jeden vektor**

- **Maticové operace, … nejen sčítání**

```
plusM m n = zipWith (\u v-> u `plusV` v) m n
```

  - **(Jednořádková definice získaná specializací, typické pro FP)**

- **(Lze definovat přetížení: (+), (-), (*) i pro vektory a matice)**

(Dělení řádku na slova)

```
type Word = String
splitWords :: String -> [Word]
splitWords st = split (dropSpace st)
split :: String -> [Word] -- dostává ř. bez úvodních mezer
split [] = []
split st = (getWord st):split(dropSpace(dropWord st))

dropSpace st = dropWhile (\x->elem x whitespace) st
dropWord st = dropWhile (\x->not(elem x whitespace)) st
dropWhile :: (t -> Bool) -> [t] -> [t] -- obecná výkonná procedura
dropWhile p [] = []       -- zahodí poč. prvky, které nesplňují podm.
dropWhile p (x:xs)        -- v prelude
   | p x       = dropWhile p xs
   | otherwise = (x:xs)
getWord st = takeWhile (\x-> not(elem x whitespace)) st
DC: takeWhile p x = …     -- v prelude

whitespace = [' ', '\t', '\n']
```

Lokální definice: where (a let)

- **Lokální def. k fci uvádí klíč. slovo where – speciální syntax**
  - **Syntakticky povoleno pouze 1x na nejvyšší úrovni (netvoří výraz)**
  - **Vztahuje se na 1 klauzuli, případně na víc stráží v ní**
  - **Definice za `where` jsou vzájemně rekurzivní (vždycky, vs. Scheme)**
- **Pro where i let : používá se 2D layout definic**

```
norma :: Complex->Float
norma cnum = sqrt(re^2+im^2)
    where re = rpart cnum
          im = ipart cnum
```

- **lokální vazby přes několik střežených rovností (nelze pomocí let)**

```
f x y | y >z  = ...
      | y==z  = ...
      | y <z  = ...
              where z = x*x
```

Lokální definice: let

- let: lokální definice ve výrazu
  - Definice (konstant, funkcí) jsou vzájemně rekurzivní
  - Použitelné pro optimalizaci společných podvýrazů
  - (! Let definuje/pojmenovává hodnoty, nevykonává přiřazení)
  - Lokální definice s 2D syntaxí, nebo oddelovač `;` a závorky `{ }`

```
c = (3.0,4.0)  -- testovací data
> let re = rpart c
      im = ipart c
    in sqrt(re^2+im^2) -- tělo výrazu
5.0       -- c = (3.0,4.0)
rpart :: Complex -> Float
rpart (re,_) = re                  -- definice selektorů
ipart :: Complex -> Float
ipart (_,im) = im
```

- Konstrukce let je výraz: může být na místě výrazu (na rozdíl od where)

```
f x y = 4*let norma (x,y) = sqrt (x^2+y^2)
              prumer x y = (x+y)/2.0
          in prumer (norma x) (norma y)
```

Pattern Matching: where, let (a lambda)

- **Na levé straně definic lze použít pattern matching, př:**

```
> let (zprava1,v1) = faze1 x  -- zpráva1
      (zprava2,v2) = faze2 v1
      (zprava3,v ) = faze3 v2
      zprava = zprava1++zprava2++zprava3
   in (zprava,v)


qsort _cmp []     = []
qsort  cmp (x:xs) = qsort cmp l ++
                    (x: qsort cmp u) where
  (l,u) = split cmp x xs
```

  - `split` se vykonává jednou a jednou prochází seznam (pro opt. změřit)
    - V Hs je lacinější (opakovaně) číst než (jednou) vytvářet (i pomocné) d.s. (… g.c.)

Stručné seznamy 1

- **analogické množinám {x | x in Xs & p(x)}**
  - **př. použití: seznam řešení - simulace nedeterminizmu**
    - **(často neefektivní) generuj a testuj**
  - **angl.: list comprehensions**
- **př.:** `[ x | x <- xs, p x ] -- filter p xs`
- **př.:** `[ f x | x <- xs]  -- map f xs`
- **vlevo od '|': výraz – pro každou hodnotu generátorů se přidá výraz**
- **vpravo od '|': generátor (`x<-xs`) nebo podmínka (stráž) nebo let**
- **pořadí zpracování generátorů: zleva doprava**
  - **Generátory a `let` zavádějí proměnné, ty lze v dalších výrazech používat**

```
> [(x,y,z)| x<-[1,2], y<-[3,4], let z = x+y]
  ~~> [(1,3,4),(1,4,5),(2,3,5),(2,4,6)]
delitele n = [d| d<-[1..n], n `mod` d == 0]
prvocislo n = delitele n == [1,n]
```

Q: kolik dělitelů se vygeneruje, aby se zjistilo, že n není prvočíslo?

```
spaces n=[' '| i<-[1..n]] -- i se nepoužije
horniTroj n = [(i,j)| i<-[1..n],j<-[i..n]] -- i se použije
```

Stručné seznamy 2

- **testy (stráže) jsou boolovské výrazy**

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

  - **xs se prochází 2x**

- **podobně: aritmetické posloupnosti**

  `[1..5] ~~> [1,2,3,4,5]`

  `[1,3..8] ~~> [1,3,5,7]` **--rozdíl určuje krok**

  `[1,3..]` **~~> [1,3,5,7,.. --nekonečný seznam**

- **! Pouze <u>aritm.</u> posloupnosti**

  map ((1/).(2^)) [1..] -- {2^(-i)}_i=1^∞

  ☺Doplňte posloupnost: 1,2,2,3,2,4,2,4,3,4,2,6,2…

Seznamy výsledků

Programátorský idiom

- **Nemáme backtracking, pracujeme s celou d.s. najednou**
- **díky línému vyhodnocování se seznam (podle implementace) postupně generuje a zpracovává, tj. není v paměti najednou**

- **Kartézský součin a kombinace**

```
kart :: [a] -> [b] -> [(a,b)]
kart xs ys = [(x,y) | x <- xs, y <- ys]
kombinace 0 ps = [[]]
kombinace _ [] = []
kombinace k (p:ps) =
   [p:k1 | k1<-kombinace (k-1) ps] ++
    kombinace k ps
```

Generuj a testuj

- **Programy „generuj a testuj“ se lehce píšou; včetně NP-úplných (př. barvení grafu, SAT, batoh …) i optimalizačních**
  - **Jinde: constraint programming (p. s omezeními), konečné domény**
- **Trade-off paměť vs. čas, (podobné *memory leak*)**
- **Př. kartézský součin víc seznamů**

```
kartn1, kartn2 :: [[a]] -> [[a]]
kartn1 []  = [[]]
kartn1 (xs:xss) = [x:ks | x<-xs, ks<-kartn1 xss]
kartn2 []  = [[]]
kartn2 (xs:xss) = [x:ks | ks<-kartn2 xss, x<-xs]
```

- **Chování: `kartn1` generuje (stejné) `ks` opakovaně, jako při prohledávání do hloubky => časově náročné**
- **`kartn2` vygeneruje *velkou* d.s. `ks` a zapamatuje si ji => (neúnosně) paměťově náročné (a příp. GHC optimalizace)**

Operátory (~syntaktický cukr)

závorky pro deklaraci operátoru

(&&&)::Int->Int->Int -- maximum

a&&&b = maxi a b

převod : op na fci, bin. fce na op.

ekvivalentní: (+) 3 4 , 3+4

5 `div` 2 , div 5 2

(deklarace operátorů) – další slajd

aplikace fce váže nejvíc:

fact 3 + 4 vs. fact (3+4) -- synt. správně, sém. různé

závorky okolo složených formálních parametrů při porovnávání se vzorem

**** abs -12 ; správně: abs (-12)**

Sekce: (+):: Int->Int->Int ; (1+), (+1)::Int->Int

První nebo druhý argument operátoru je zadán

Lze použít pro všechny operátory, taky `(`div`  2)` je sekce

I pro funkce víc (než dvou) argumentů, pokud je zadán první nebo druhý

Operátory

9-0 ; 9 nejvyšší priorita , 0 nejnižší

funkční volání .. váže těsněji než 9

```
9,doleva asoc.   !!        -- n-tý prvek
9,doprava        .         -- skládání fcí
8,doprava        ^ ^^ **
7,doleva         * / `div` `mod` `rem` `quot`
6,doleva         + -
```

i unární -

```
5,doprava        : ++
5,neasoc         \\          -- delete
4,neasoc         == /= < <= > >= `elem` `notElem`
3,doprava        &&          -- and
2,doprava        ||          -- or
```

Definice operátorů

vyšší priorita váže víc, jako v matematice

pouze binární operátory

```
infixl 6 +        -- sčítání, doleva asoc.
infixr 5 ++       -- append, doprava asoc.
infix  4 ==       -- porovnávání, neasoc.
infix  4 `elem`   -- prvek seznamu, neasoc.
```

I pro alfanumerické operátory lze def. prioritu, aby se správně vázaly v kontextu

Odlišnosti aplikačního programování

- **bezstavové, nemá (klasické) proměnné**

  Proměnné pojmenovávají hodnoty (ne adresy), i složené

  ⇒Nové hodnoty jsou v nové paměti, případně část paměti je sdílená

  ⇒Uvolňování paměti prostřednictvím GC – garbage collector

- **neobsahuje příkazy, např. přiřazení**

  neobsahuje sekvenci příkazů

  FP: výrazy, vnořené funkce h(g(f x))

- **výraz je "definovaný" svým výskytem (jako aktuální parametr)**

  použije se jen 1x, na místě svého výskytu

- **Program pracuje s celými d.s. (seznamy , stromy …)**
  - **analogie: roura/pipe '|' v Unixu a nástroje na zpracování souborů**
  - **Funkce pro vzory zpracování/rekurze d.s., i uživatelských**
    - **Pro seznamy např: filter, map, zip, …**
- ***Pozorování: rekurzivní struktury se dobře zpracovávají rekurzivními programy***

Odlišnosti aplikačního programování 2

- **Program vzniká (obvykle) skládáním z jednodušších funkcí (ne jako posloupnost změn položek d.s.)**
  - **higher-order funkce umožní při skládání bohatou parametrizaci**
    - **kombinátory: funkce určené pro skládání funkcí**
  - **definice jednoúčelových funkcí specializací hi-ord fcí, pro práci v určité oblasti/doméně, viz `zipWith`**
    - **Vhodné pro stručnost zdrojáku, ne efektivitu**
    - **př. (s vnořenou specializací): f (g1 g2 (h1 h2)) x**
- **Režie definice nové funkce je malá, možnost zabstraktňovat je velká**

  ⇒Mnoho krátkých funkcí ; lépe se skládají

  ⇒Celkově kratší kód, pouze jednou impl.

  (DRY - Don't Repeat Yourself)

Syntax

case, if - porovnávání se vzorem

stráže

let, where - vložené definice

\ x -> *f(x)* - definice funkcí (lambda)

dvourozměrná syntax (layout)

stručné seznamy

data - definice nových uživatelských datových typů

type - typové synonyma

```
type Complex = (Float,Float)
```

(rozšiřující část)

Uživatelské typy – nerekurzivní

- Klíčové slovo `data`, vlevo od „=” typ, vpravo datové konstruktory

```
data Bool = False | True  -- předdef. prelude
data Ordering = LT | EQ | GT -- prelude
data Color = Red | Green | Blue
data Point a = Pt a a     -- polymorfní
data Maybe a = Nothing | Just a --např. repr. chyb
data Either a b = Left a | Right b --sjednocení typů
```

Bool, Color, Point,… jsou jména typů, tj. *typových* konstruktorů

False, Red, Pt ... jsou jména (uživatelsky def.) *datových* konstruktorů

- vyhodnocený tvar výrazu obsahuje (pouze) datové konstruktory

`Pt` má dva argumenty, `Just` jeden arg., `False`, `Red`, `Nothing` ... jsou konstanty a mají nula arg., ...

- Datové konstruktory mají svůj konkrétní typ, odvozený z definice typu
  - True :: Bool ; Just :: a-> Maybe a,
  - Pt :: a-> a-> Point a ; Left:: a -> Either a b

- Datové konstruktory jsou vyhodnocený tvar, můžou mít nevyhodnocené argumenty

Uživatelské typy – nerekurzivní

- Datové konstruktory generují výraz určitého typu
  - Protože nejsou přetížené ; typ může být „parametricky“ nedourčený
  - D.k. pomáhají typovému systému při odvozování
    - Konstruktory `Just, Left, Right` určují (úroveň) vnoření hodnoty
  - D.k. slouží na rozlišení variant hodnoty, obdobně jako tagy v Scheme
    - Př: Komplexní čísla v pravoúhlých a polárních souřadnicích
- porovnávat se vzorem lze i uživ. typy

```
jePocatek (Pt 0 0) = True
jePocatek _        = False
```

- konkrétní hodnoty jsou konkrétního typu

```
Pt 1 2 :: Point Int
Pt 1.5 (-0.2)::Point Float;
Pt 'a' 'b' :: Point Char
```

- vestavěné typy nejsou speciální; liší se lexikálně, ne sémanticky

Použití Maybe

Hledání prvku v seznamu (asociativní paměť, reprezentace slovníku s (klíč,hodnota))

```
lookup :: Eq a=> a-> [(a,b)]-> Maybe b
```

Pokud se 1. arg. v seznamu nenajde:

1. **Nevím, co mám vrátit (správného typu b)**
   - např. vstupní seznam je prázdný
2. **Není to chyba, ale situace, kterou chci (detekovat a) ošetřit**

Vracet typ `(Bool,b)` neřeší 1.

Vracet pevnou konstantu (určitého typu: -1,'\0') není polymorfní (tj. dostatečně obecné)

```
lookup _     []              = Nothing
lookup klic ((h,val):tab) =
  if klic == h then Just val
    else lookup klic tab
```

Výhody:

`lookup` je polymorfní v typu klíče i výsledku (-> jeden source)

„escape" hodnota `Nothing` je mimo typ výsledku (nezabírá žádnou hodnotu)

pozn: Nothing je polymorfní konstanta (a datový konstruktor):

```
Nothing ::       Maybe b
Just    :: b -> Maybe b
```

Pojmenované položky (Hs98)

Hs: položky podle polohy

```
data Point = Pt Float Float
pointx :: Point -> Float
pointx (Pt x _) = x          -- definování selektoru
```

Hs 98 navíc: pojmenované položky

```
data Point =
  Pt{pointx,pointy :: Float} -- pojmenované složky
pointx::Point->Float -- implicitně zaváděné selektory
absPoint :: Point -> Float
absPoint p =
  sqrt(pointx p*pointx p + pointy p*pointy p)
absPoint2 (Pt {pointx=x, pointy=y}) =
  sqrt(x*x + y*y)
```

- Pattern matching pro pojmenované položky

Rekurzivní uživ. typy 1

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
data Tree2 a = Void
            |  Node (Tree2 a) a (Tree2 a)
data Tree3 a b = Leaf3 b
            | Branch3 (Tree3 a b) a (Tree3 a b)
```

- **Deklarací typu jsou zavedeny konstruktory jako funkce**
- **Různé stromy pro různé použití**
  - **`Tree a`: Huffmanův strom, struktura mergesortu**
  - **`Tree2 a`: obvyklé vyhledávací stromy, binární halda, struktura quicksortu**
  - **`Tree3 a`: strom s různými vnitřními a vnějšími vrcholy**

Rekurzivní uživ. typy 2

- **přístup na složky: porovnávání se vzorem, tj. case, (uživatelem definované) selektorové funkce leftSub,..**

```
Leaf :: a -> Tree a
Branch :: Tree a -> Tree a -> Tree a
leftSub :: Tree a -> Tree a
leftSub (Branch l _) = l
leftSub _            =
   error "leftSub: unexpected Leaf"
```

- **př.: seznam listů stromu, ++ je spojení seznamů**

```
listy :: Tree a -> [a]
listy (Leaf a) = [a]
listy (Branch l r) = listy l ++ listy r
-- listy (Branch l r) = append (listy l)
  (listy r)
```

Regulární typy

- **n-ární stromy**

```
data NTree a = Tr a [NTree a]
```

  - **nepřímá rekurze při definici dat. Typu**

  => Typické zpracování dat typu `NTree`: 2 vzájemně rek. fce, jedna pro stromy, druhá pro seznam podstromů

  - **Ukončení rekurze: list má *prázdný* seznam podstromů**
  - **Typ konstruktoru: `Tr :: a -> [NTree a] -> Ntree a`**

- **regulární typy v Hs: typové konstruktory (`NTree`) použité na pravé straně definice mají stejné argumenty (typové proměnné, zde: `a`) jako na levé straně definice**
- **Aplikace NTree: syntaktické stromy, Html a XML**
- **!nejde (v std. Hs): datový konstruktor má vlevo jiné argumenty:**

  **** data Twist a b = T a (Twist b a) | Notwist (A)**

  **** data Vtree a = In (Vtree (a,a)) | Vlist a -- repr. vyvážené stromy (B)**

  - **Ad (A): jde rozepsat na vzájemně rekurzivní typy**
  - **(V implementaci GHC je spousta rozšíření, i typových)**

Příklady typů

- **Kolekce: skládá se z prvků a jiných kolekcí**

```
data Coll a = Item a | ItemSet [Coll a]
```

- **Převod kolekce na seznam prvků**

```
c2l :: Coll a -> [a] -- kolekce na seznam
c2l (Item x)     = [x]
c2l (ItemSet xs) =
  concat(    -- ::[[a]] -> [a]
    map c2l xs)  -- map::f-> [Coll a] -> [[a]]
```

- **DC: přidejte ke každé položce hloubku**
  - **Hint: výsledek je typu `Coll (a,Int)`**
- **DC: definujte typ pro regulární výrazy**
- **DC: navrhněte dat. strukturu pro syntaktické stromy vět přirozeného jazyka, kde budeme rozlišovat, zda je podstrom vlevo nebo vpravo od řídícího slova**

Typová synonyma 1

- **Klíčové slovo `type`**
  - **na pravé straně od '=' jsou jména jiných *typů***
  - **nepoužívá datový konstruktor**

```
data Barva = Trefy | Kara | Srdce |
  Piky
data Hodnota = K2|K3|…|K9|K10|J|D|K|A
type Karta = (Barva,Hodnota)
type RGB = (Int, Int, Int)
type Complex = (Double, Double)
type Matice a = [[a]]
```

- **Systém při svých výpisech typová synonyma nepoužívá, vrací rozepsané typy**

Typová synonyma 2

Klíčové slovo `newtype`

- definování nového nekompatibilního typu stejné struktury
  - Typový systém nedovolí typy míchat (např. `Int` s `Euro`),
- datový konstruktor na pravé straně má právě jeden arg. (a to původní typ)

```
newtype KartaM = M Karta
```

datový konstruktor `M` pro rozlišení typů

rozlišení se používá pouze při kompilaci => bez run-time penalizace/overheadu

```
newtype Euro = Euro Int --  odlišme peníze od čísel
```

- Typový (první) a datový (druhý) konstruktor Euro jsou v různých „namespace“

```
plusEu :: Euro -> Euro -> Euro
Euro x `plusEu` Euro y = Euro (x+y)
```

- funkci `(+)` , … lze přetížit i pro typ `Euro`, pomocí typových tříd (později)

```
plusEu = lift2Eu (+)   -- definice pomocí „pozvednutí“ funkce
lift2Eu :: (Int->Int->Int) -> Euro -> Euro -> Euro
lift2Eu op (Euro x) (Euro y) = Euro (x `op` y)
```

Args. `lift2Eu` jsou typu `Euro`, po patt.m. `x` a `y` typu `Int`, `` `op` `` na t. `Int`, dat. konstr. `Euro` převede výsledek na typ `Euro`

Fce `lift2Eu` je analogická funkci `zipWith` pro (nerekurzivní) typ `Euro`

Komplexní aritmetika

- **Pro porovnání:**

```
type Complex = (Double, Double)
data Complex2 = C2 Double Double
newtype Complex3 = C3 (Double,Double)
```

- **komplexní číslo jako dvojice reálných č.**

```
cadd :: Complex -> Complex -> Complex
cadd (rx,ix) (ry,iy) = (rx+ry,ix+iy)
```

-- implicitní typ fce cadd je obecnější

```
...
> cadd (1.0,0.0) (0.1,2.0)
(1.1,2.0)
```

- **varianta**

```
cadd3 (C3(rx,ix)) (C3(ry,iy)) = C3(rx+ry,ix+iy)
```

Case

- **Výraz `case`, obecné porovnávání se vzorem: syntaktické schéma**
  - **Pro použití ve výrazech (v def.), zákl. způsob rozlišování konstruktorů**
  - **Také jeden výraz: `case xs of …`**

```
case (vstupníVýraz, ...) of
    (vzor, ...) -> výraz
    ...
```

`take n xs` vybere ze seznamu `xs` prvních `n` prvků, pokud existují

```
take2 :: Int -> [a] -> [a]
take2 n xs = case (n,xs) of
  (0,_)     -> []
  (_,[])    -> []
  (n,x:xs) -> x : take2 (n-1) xs
```

Př.:Take

- **Funkce `take`, ekvivalentní (a obvyklý) zápis**

`take n xs` vybere ze seznamu `xs` prvních `n` prvků, pokud existují

```
take1 0 _        = []
take1 _ []       = []
take1 n (x:xs) = x : take1 (n-1) xs
```

Podmíněný výraz

if výraz je "syntaktický cukr/pozlátko"

```
   if e1 then e2 else e3
```

ekvivalentní

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

Funkce 1

- **funkce se konstruují lambda-abstrakcí**

```
succ x = x+1 -- obvyklý zápis
succ = \ x -> x+1 -- ekv. lambda výraz
add  = \ x y -> x+y -- víc parametrů
```

  - **alternativně, stejný význam:** `add = \x -> \y -> x+y`
  - **formální parametry mají rozsah platnosti tělo definice**
  - **succ a add se vyhodnotí na (interní) repr. daných fcí**
  - **funkci lze aplikovat na argument (!mezerou)**
  - **typy: když** `x::a`, `e::b`, **pak** `\x->e :: a->b`
- **anonymní funkce - jako parametry fcí vyšších řádů**
- **referenční transparentnost: volání fce na stejných parametrech vrátí stejnou hodnotu**
  - **protože nemáme "globální" proměnné, přiřazení a sideefekty**
  - **Nezáleží na pořadí vyhodnocování => je možné líné vyhodnocování**

Funkce 2

- **na funkcích není definována rovnost**
- **funkci můžu aplikovat na argumenty**
  - **“aplikace je (jediný) selektor”**

  => lze zjistit hodnotu funkce v jednotlivých “bodech”
- **skládání fcí**

```
(.) :: (b->c) -> (a->b) -> (a->c)
f . g = \ x -> f (g x) -- operátor v definici
-- (f . g) x = f (g x) -- ekviv.
-- (.) f g x = f (g x) -- ekviv.
id :: a -> a
id x = x --levý i pravý neutrální prvek pro (.), tj. skládání fcí
```

- **Aplikace, pro pohodlnější zápis**

```
($) :: (a->b) -> a -> b  -- tj. typ aplikace funkce
f $ x = f x  -- asociativita doprava: f3 $ f2 $ f1 x
```

Funkce 2 použití skládání

- definici lze psát bez posledních argumentů na obou stranách

```
odd :: Int -> Bool
odd = not . even
```

--tzv. bezbodový zápis fcí, často jako arg. fcí vyššího řádu

Nemusím vypisovat (poslední) argumenty, protože systém si je domyslí (podle typu)

```
-- odd x = (not.even) x
```

-- ekv. zápis se všemi argumenty

```
-- odd x = not(even x)
```

-- ekv. zápis

```
-- odd x = not $ even x
```

-- ekv. zápis bez závorek

- Varianta fce `filter`, která vypouští prvky, pro které je splněna podmínka

```
negFilter :: (a->Bool) ->[a] -> [a]
negFilter f = filter (not.f)
```

-- bez argumentů

Funkce 3

- **currying / curryfikace: argumenty chodí po jednom**
  - **typ a->b->c místo (a,b)->c**
  - **výhoda: lze částečně aplikovat**
- **sekce: př: (+1), (1+), (+), (`mod`2), (10`div`)**

(x *op*) ekv. \y ->x *op* y, (*op* y) ekv. \x ->x *op* y

(*op*) ekv. \x y-> x *op* y

```
harmonicky_prumer :: [Float] -> Float
harmonicky_prumer xs =
   fromInt(length xs)/    -- transf. na Float
     (sum (map (1/) xs)) -- inverzní hodnota
```

Funkce 4 – transformace funkcí

- Někdy potřebujeme fci upravit pro plynulé použití
- Přehození dvou argumentů, flip, v prelude

```
flip:: (a->b->c) -> b->a->c
flip f x y = f y x
> map (flip elem tab) [1,2,3]
```

- Fce curry, uncurry, v prelude

```
curry :: ((a,b)->c) -> a->b->c
curry f x y = f (x,y)
uncurry (a->b->c) -> (a,b)->c
uncurry f (x,y) = f x y
```

- Př.: data jsou dvojice, ale f. očekává dva arg. => přizpůs. fce

```
paryEq :: [(a,a)] -> Bool
paryEq xs = and $ map (uncurry (==)) xs
> paryEq [(1,1),(5,5)]
```

Aritmetika s testováním chyb (pomocí Maybe)

```
lift2M :: (a->b->c)->Maybe a -> Maybe b -> Maybe c
```

- Pozn.: typy arg. a výsl. jsou obecně různé, ! typ chceme co nejobecnější
- Vlastně taky transformace funkcí, („boilerplate“ kód)

```
lift2M op Nothing _          = Nothing – chyba v 1. arg.
lift2M op _       Nothing    = Nothing – chyba v 2. arg.
lift2M op (Just x) (Just y) = Just (x `op` y) – bez chyb
minusM :: Maybe Float -> Maybe Float -> Maybe Float
minusM x y = lift2M (-) x y
```

- Vyvolání chyby

```
delenoM x y = if y==Just 0 then Nothing --test a vyvolání chyby
              else lift2M (/) x y
> delenoM(Just 3)(minusM(Just 2)(Just 2)) -- 3/(2-2)
Nothing
> delenoM(Just 3)(lift2M(-)(Just 2)(Just 2)) -- dtto
Nothing
```

  - Zápis výrazu je méně přehledný
  - Můžu si napsat interpret, který bude volat/používat rozšířené funkce

Aritmetika s testováním chyb, aritm.výrazy

- **Typ výrazu**

```
data AV a = AV a :-: AV a
          | AV a :/: AV a
          | Con a
          | …
```

  - **Používáme symbolové datové konstruktory (musí začínat dvojtečkou)**

- **Vyhodnocování výrazu, s chybou**

```
eval :: AV Integer  -> Maybe Integer
eval (av1 :/: av2) = delenoM (eval av1) (eval av2)
eval (av1 :-: av2) = lift2M (-) (eval av1) (eval av2)
eval (Con x)       = Just x  -- cena za Maybe: zabalení výsledku
```

- **Odchycení chyby: 2. arg jsou opravná data**
  - **Argument `eval` jsou konkrétní data definovaného typu**

```
> catch (eval (Con 3 :/: (Con 2 :-: Con 2)) ) 1
catch Nothing  oprData = opravnaFce oprData
catch (Just x) _       = x
```

Nekonečné d.s.

- **líné vyhodnocování umožňuje potenciálně nekonečné d.s.**
  - **použijeme jen konečnou část (anebo přeteče paměť)**
  - **! konečná reprezentace v paměti: funkce, která generuje výsl.**

```
numsFrom n = n : numsFrom (n+1)
fib = 1 : 1 : [a+b | (a,b) <- zip fib (tail fib)]
factFrom = map fact (numsFrom 0)
```

- **Autotest: efektivnější faktoriál**
  - **idea: prvky seznamu budou dvojice (n,f(n))**

(Nevýhoda líného vyhodnocování, hackerský koutek:)

memory leak **– nespočítání výrazu a neuvolnění paměti, dokud není hodnota výrazu potřeba**

Typicky to vadí, pokud je výsledek jednoduchá hodnota (anebo ji výsledek obsahuje)

Př: třetí fibonacciho číslo – Hs počítá od 0 :

`fib!!2 ~~>` **interně je v seznamu nevyhodnocený term :** `1+1`

… který nějakou dobu pobývá v paměti místo (očekávané) `2` **-- memory leak**

Pro vyšší fib.č. f(n): podtermy se sdílí, paměť (i čas) jsou O(n)

Nekonečné dat. struktury

- **několik užitečných procedur, z prelude**

```
repeat x = x : repeat x
cycle xs = xs ++ cycle xs
iterate f x = x : iterate f (f x)
```

  - **počítá [$f^0(x)$, $f^1(x)$, $f^2(x)$, $f^3(x)$, …], ale sdílí mezivýsledky $f^i(x)$,**

- **př.: jednotková matice, potenciálně nekonečná**

```
jedn_mat =
```

`iterate (0:)` -- další řádky, `0:` se přidá na začátek

`(1:repeat 0)` -- první řádek `1:0:0:0:`…

```
jedn_mat = [ 1:0:0:0: … ,
             0:1:0:0: … ,
             0:0:1:0: … ,
             …
```

  - **Při sčítání `jedn_mat` s konečnou maticí pomocí `zipWith` se vygeneruje a použije pouze konečná část**

Nekonečné dat. struktury a fce

- **funkce psát kompatibilně s líným vyhodnocováním:**

```
and1 True True = True -- není líná ve 2. arg.
and1 _    _    = False
and2 True  x = x     -- je líná ve 2. arg.
and2 False _ = False
```

- **zpracování nekonečných struktur:**
  - **mít fce, které odeberou pouze potřebnou část d.s. (take, takeWhile, ... getWord)**
  - **mít fce, které zpracují nekon. d.s. na nekon. d.s. (tj. *ne*striktní)**
    - **Např. map, zip, filter, drop, dropWhile ...**
    - **typicky fce zpracují úvodní část vstupu na úvodní část výstupu**
  - **Past ☹: `[x| x<-[0..], x<4 ]  ~~> [0,1,2,3,⊥`**
    - **nedostaneme „očekávaný“ výsledek: `[0,1,2,3]`, tj. `0:1:2:3:[]`**
    - **systém neodvozuje „limitní“ chování, ale počítá podle předpisu**
- **striktní funkce: vyhodnotí vždy svůj argument (úplně)**
  - **Nevhodné pro zpracování nekonečných d.s.**
  - **Pro striktní funkce platí : `f ⊥ ~> ⊥`**

Šifrování klíčovým slovem

```
sifruj :: String -> String -> String
sifruj klicSlovo plainText =
  zipWith sifrujZnak (cycle klicSlovo)
                     plainText
  where
   sifrujZnak klic plain =
     int2Ch((ch2Int klic + ch2Int plain)
            `mod`26)
   int2Ch ch = … ; ch2Int i = …
```

Šifrování znaků samostatně, „klic“ je popis (jméno) tab.

analogicky: šifrování bloků dat - místo znaků

DC: proudové šifrování (stream) – cipherZnak závisí na předchozím plainTextu

- jiný vzor rekurze (s akumulátorem, tj. vnitřním stavem)

Příklad: Pascalův trojúhelník

- idea: z jednoho řádku generujeme další

```
pascalTr = iterate nextR [1]  where
  nextR r = zipWith (+) (0:r) (r++[0])
```

- Stavové programování
- (Návrhový vzor *iterátor*)
  - N.vzory ve FP lze (často) napsat jako kod s funkc. parametry
    - vs. (často) pseudokód v OOP

```
iterator::s->(s->s)->(s->Bool)->s
```

-- vrací celý (interní) stav, iterator nemá výstupní projekci

```
iterator init next done =
  head( dropWhile (not.done)          -- mezivýsl. se průběžně
          ( iterate next init ) )     -- zahazují
```

DC: fixpoint :: `(s->s)->s->s` : vrací v=f^n(x) pro min. n:v=f(v)

Pozn: protože se mezivýsledky průběžně zahazují, nemám v paměti celý seznam stavů

Prohledávání grafu po vrstvách (do šířky)

Pamatujeme si `open` – hraniční vrcholy a `closed` – spracované vrcholy; jeden krok je generování jedné vrstvy

```
type Graf a = …
vrstvy :: Graf a -> a -> [[a]] -- výstup je nekonečný
vrstvy gr start =
  map fst -- chci pouze nové vrstvy, interní stav na výst. hodn.
    (iterate (nextV gr) -- přechod k nové vrstvě
      ([start],[]) )  -- počáteční stav
nextV gr (open, closed) = (open1,closed1) where
  dalsi = concat (map sousede open) --multimnožina
  closed1 = open `union` closed
  open1 = toSet dalsi `setDiff` closed1
  sousede v = … lookup v gr … --
     -- gr je form.par. nextV, schovaný (zlokalizovaný) param.
  toSet bag = …  -- převod z multimnožiny na množinu
```

Polymorfní typy

- **T: Funkce má jeden nejuniverzálnější typ (polymorfní nebo monomorfní)**
  - **(Typové proměnné jsou kvantifikovány univerzálně a v prefixu, tj. na nejvyšší úrovni pro celou funkci)**
  - **(implementace (např. GHC) mají mnoho typových rozšíření, někdy musí uživatel typovému odvozovači napovědět)**
- **odvozování typů: unifikací**
  - **Odvozování typů je těžší úloha než kontrolování typů**

```
f :: (t,Char) -> (t,[Char])
f (x,c) = (x,['a'..c])
g :: (Int,[u])->Int
g (m,l) = m+length l
h = g.f  -- (.):: (b->c)->(a->b)->(a->c)
-- (t,[Char]) unifikuj s (Int,[u])
h::(Int,Char)->Int -- monomorfní (bez typových proměnných)
```

- **při unifikaci se zjistí pouze nekonzistence, ne místo (nebo důvod) chyby**
  - **př: unifikuj `t1=[(a,b)]` s `t2= ([c],d)` : chybí `head` na t1 nebo `fst` na t2 ?**
- **zpracování zdrojáku je víceprůchodové, typy se kontrolují po syntaxi**

Polymorfní typy

- **Polymorfní konstanty, funkce:**
  - **každý výskyt se typuje zvlášť**

`length([]++[1])+length([]++[True])` – OK

:: [Int] -> Int :: [Bool] -> Int --obecně :: [a] -> Int

- **Proměnná: má pouze jeden (polymorfní) typ**
  - **Typ `x` nejde určit:**
- **`length(x++[1])+length(x++[True])` -- NE x::?**

Polymorfizmus a typové třídy

Parametrický p.

Př.: length :: [a] -> Int

Realizace: Typové proměnné

Na typu argumentu nezáleží

Stejná impl. pro všechny typy

- **Kompiluje 1x**
- **Kód funguje i pro budoucí typy**

Podtypový p. (přetížení)

(==) :: Eq a => a -> a -> Bool

(+), (*) :: Num a => a -> a -> a

Typové třídy (typový kontext)

... Záleží

... Různá (instance třídy pro typ)

- **Kompiluje pro každý typ zvlášť**
- **Pro nový typ přidat další kód**

- **Přetížení nevyužívá dedičnost a hierarchii z OOP**
  - **Třída může záviset na jiné třídě/třídách – forma hierarchie**
  - **Instance může záviset na instanci (pro jiný typ nebo jinou třídu)**
- **Hs: funkce mají t. kontext; OOP: objekty mají TVM (tabulku virt. metod)**
  - **Haskell podle (odvozených) typů args. určí správnou implementaci**
- **Pozn: Generické procedury (template, makra): jeden (meta)zdroják, samostatné generování zdrojáku a překlad pro jednotlivé typy**
  - **Rozšíření: Template Haskell, Generic Haskell**

Typové třídy - idey

- **ne všechny operace jsou definovány na všech typech**
  - **typová třída: abstrakce těch typů, které mají definovány dané operace**
  - **`Eq Ord Show Read Enum Num Integral Fractional Ix`**
- **tento mechanizmus odpovídá ad hoc polymorfizmu, tj. přetížení**
- **2 klíčová slova:**

`class` - zavedení typové třídy

`instance` - definování typu jako prvku typové třídy, spolu s definováním operací

- **typový kontext funkce: podmínky na typy, př. eqpair:**
  - **Funkci `eqpair` lze použít pouze na typy, pro které je def. rovnost**
  - **Funkce není přetížená (kompilace 1x), pouze využívá přetížení (==)**
  - **Systém podle konkr. typu arg. použije správnou implementaci (==)**

```
eqpair :: (Eq a,Eq b) => (a,b)->(a,b)->Bool
eqpair (x1,x2) (y1,y2) = x1==y1 && x2==y2
elem :: Eq a => a -> [a] -> Bool  -- prelude
```

Přetížení - detaily

```
(+) :: (Num a) => a -> a-> a
```

- **Přetížené konstanty**

```
> :t 1
1 :: (Num a) => a
```

  - **Implementace: využívá `fromInteger`**
  - **Analogie k `[] :: [a]`**

- **Použití `(+)` , s výpisem typu**

```
> :set +t      ; nastavení přepínače t, aby Hs vypisoval typy
> 1.1 + 2.2
3.3 :: Float
```

- **Chybné použití `(+)` na typ `Char`, který není instancí `Num`**

```
> 'a'+'b'
No instance for (Num Char)
```

- **Pro jeden typ a třídu lze mít pouze jednu instanci**
  - **Ale lze: pomocí `newtype` získat izomorfní typ s jinou instancí**

Typové třídy, Eq

- **Deklarace typové třídy (`class`) obsahuje**
  - **jména a typové signatury funkcí – přetížené (overloaded) fce**
  - **případně defaultní definice některých funkcí**
    - **Použijí se, pokud nejsou předefinovány**
- **definice třídy typů, které lze porovnávat: `Eq`**
  - **Instance: všechny zákl. typy `Bool Int Integer Float Double Char`**
  - **Seznamy a n-tice, pokud položky jsou instance (-> a tranzitivně odvozené)**
  - **Nelze porovnávat: funkce; obecně typy, které nemají instanci `Eq`**

```
class Eq a where
   (==),(/=) :: a -> a -> Bool -- typy funkcí
   x/=y  = not (x==y)     -- defaultní definice
```

  - **! Upozornění: Haskell nekontroluje vlastnosti a „kompatibilitu“ definic funkcí: např., že platí `x/=y || x==y` (možná za pár let ☺, potřebuje silnější typ. systém)**
- **typ rovnosti `(==) :: (Eq a) => a -> a -> Bool`**
  - **Při použití == na typ `a`: pro typ `a` musí být definována instance třídy `Eq`**

```
instance Eq Int where    -- implementace fcí z třídy Eq pro typ: Int
   x == y  = intEq x y   -- vest. fce intEq
```

  - **Nerovnost jsme nedefinovali, tj. použije se defaultní**
- **Všechny fce zavedené v `class` musí být deklarovány, ale lze použít „stub“**

```
x /= y  = undefined
```

Typové třídy - Eq

- **definice pro neparametrický uživatelský typ**

```
instance Eq Bool where --implementace fcí pro typ: Bool
  False == False  = True -- def. rozpisem
  True  == True   = True
  _     == _      = False
```

- **definice pro parametrický uživatelský typ (tj. typový konstruktor)**
  - **Typ prvků je instance `Eq`**

```
instance (Eq a) => Eq (Tree a) where
     -- umíme porovnávat hodnoty v listech
 Leaf a          == Leaf b          = a == b  ; používá Eq a
 (Branch l1 r1) ==(Branch l2 r2)  = (l1==l2) && (r1==r2)
  _              == _               = False ; Branch vs. Leaf
```

- **Víc podmínek v typovém kontextu**

```
instance (Eq a,Eq b) => Eq (a,b) where
  (a1,b1) == (a2,b2) = a1 == a2 && b1 == b2
```

- **^na dvojicích ^na typu a ^na typu b**

Typové třídy - deriving

- **(Některé) typové třídy: `Eq, Ord, Read, Show` … si lze nechat odvodit při definici nového typu**
  - **Klíčové slovo `deriving` při definici typu**
  - **Vytvoří se standardní definice tříd Eq, Ord, Show, Read**
    - **Rovnost konstruktorů a složek**
    - **Uspořádání podle pořadí konstruktorů v definici, složky lexikograficky**

```
data Bool = False | True
  deriving (Eq, Ord, Show, Read)
data Point a = Pt a a deriving Eq
```

třída Ord

- **Hodnoty typu jsou lineárně uspořádané**
  - **Musíme mít (už/také) definovanou rovnost, typ je instance Eq**
  - **Instance: základní typy; seznamy a n-tice, pokud jsou položky instance**

```
class (Eq a) => Ord a where
 (<=),(<),(>=),(>) :: a->a->Bool
 min,max :: a->a->a
 compare :: a->a->Ordering
 x < y = x<=y && x/=y     -- defaultní deklarace
 (>=) = flip (<=)    -- prohodí args
 min x y = if x<=y then x else y
 ...
```

  - **Funkce <= je základní. Defaultní fce lze (např. kvůli efektivitě) předefinovat**

- **Použili jsme:**

```
flip op x y = y `op` x
data Ordering = LE | EQ | GT
   deriving (Eq, Ord, Read, Show, Enum)
```

- **DC: compare, pomocí (<=)**

třída Ord

- **Instance pro typy a typové konstruktory:**

```
instance Ord Bool where
True  <= False = False
_     <= _     = True
instance (Ord a,Ord b)=> Ord (a,b) where
(a1,b1) <= (a2,b2) = a1<a2 || a1==a2 && b1<=b2
instance (Eq [a],Ord a) => Ord [a] where
 []   <= _     = True          -- lexikograficky
 x:xs <= y:ys = x<y || x==y && xs<=ys -- systém určí typy
 _    <= _     = False
```

- **Pouze jedno třídění seznamů pod jménem <= z třídy `Ord`**
  - **jiné třídění lze definovat na novém typu, využít `newtype`**

```
newtype L1 a = L1 [a] -- pro jiné třídění
instance Eq a => Eq (L1 a) -- Haskell 98 OK
  where
   L1 x == L1 y  =  x==y
instance Ord a => Ord (L1 a) where
  L1 x <= L1 y = (length x,x)<=(length y,y)
> L1 "ab" <= L1 "b"
False
```

třída Show, Read , Enum

- **`Show a`: Hodnoty typu `a` lze převést na znakové řetězce**
  - **Instance: základní typy**
  - **Seznamy a n-tice, pokud jsou položky instance**
  - **Pozn: pouze převod na String, vlastní výstup je samostatně**

```
show :: a -> String
```

- **`Read a`: Hodnoty typu `a` lze převést ze znakového řetězce**

```
read :: String -> a
```

  - **často nutná specifikace typu: `read " 1 " :: Int`**

Enum: výčtové typy

```
enumFrom :: a->[a]  -- [n..]
enumFromTo :: a->a->[a] --[k..n]
enumFromThen :: a->a->[a] --[k,l..]
enumFromThenTo :: a->a->a->[a] --[k,l..n]
```

typové třídy – čísla

Num: `(+),(-),(*) :: a->a->a`

`abs, negate, signum :: a->a`

`fromInt :: Int -> a  --` **převod ze standardních čísel**

`fromInteger :: Integer -> a`

- **Typy jsou instance** `Eq` **a** `Show`
- **Hodnoty jsou číselné**
- **Instance:** `Int Integer Float Double`
  - **Lze def. pro: komplexní č., zbytkové třídy, vektory, matice**

Integral: `div,mod,quot,rem :: a->a->a`

`toInteger :: a->Integer  -- dtto. toInt`

- **Typy jsou instance** `Num`
- **Hodnoty jsou celočíselné**
- **Instance:** `Int, Integer`

typové třídy – čísla

Fractional:

```
(/) :: a -> a -> a
recip ::a -> a
fromDouble :: Double->a
```

- Typy jsou instance třídy `Num`
- Hodnoty jsou neceločíselné
- Instance: `Float Double` (a přesné zlomky `Ratio a`)

Floating: `exp,log,sin,cos,sqrt :: a->a` --...

```
(**) :: a->a->a – umocňování
```

- Typy jsou instance `Fractional`
- Instance: `Float Double`

Bounded: `minBound, maxBound :: a`

Ix: `range::(a,a)->[a];`

```
index::(a,a)->a->Int
```

– indexy polí

Vlastní instance pro Num

- **Zbytkové třídy mod 17 (neúplné)**

```
data Z17 = Z17 Int deriving (Eq,Ord,Show)
instance Num Z17 where
   Z17 x + Z17 y = Z17 ((x+y)`mod`17)
   (*) = lift2Z17 (*)   -- pro obvyklou def. lift2Z17
   fromInt x = Z17 (x `mod` 17)
lift2Z17 op (Z17 x)(Z17 y)=Z17((x`op`y)`mod`17)
```

- **pevná desetinná čárka, na 100 míst**

```
data Fixed=Fixed Integer deriving (Eq,Ord,Show)
instance Num Fixed where
   Fixed x + Fixed y = Fixed (x+y)
   Fixed x * Fixed y = Fixed ((x*y)div 10^100)
```

- **DC: pro komplexní čísla**
- **FFT napsaná pomocí + - * (a fromInt) funguje v C i Z17**
- **-> výhoda t.tříd: stejný kód (s ops. z t.tříd) pro různé typy**

(Vlastní třídy)

- **Monoid: binární skládání ^^^ a neutrální prvek**

```
class Monoid a where
   (^^^) :: a -> a -> a
   neutral :: a
```

- **Grupa: monoid a má inverzí prvek**

```
class (Monoid a) => Grupa a where
   inv :: a -> a
```

- **<u>Konstruktorová</u> třída: Functor – pro strukturu lze definovat „map“**

```
class Functor a where
   fmap :: (b->c) -> a b -> a c  -- a je t.konstr., ne typ
instance Functor [] where
   fmap f xs = map f xs
instance Functor Tree2 where
   fmap _f Void         = Void
   fmap f (Node l x p) = Node (fmap f l)(f x)(fmap f p)
```

Funkce vyšších řádů: map

- **map pro matice**

```
map_m :: (a->b)-> [[a]] -> [[b]]
map_m f x = map (map f) x
```

- **map je kompatibilní s líným vyhodnocováním**
  - **Tj. generuje výsledek postupně**
- **Možnost optimalizací**

1. `map f (map g xs)  -- dva průchody`
2. `(map f.map g) xs  -- pořád dva p.`
3. `map (f.g) xs      -- jeden průchod xs`

  - **Ver. 3 je efektivnější, protože negeneruje mezilehlý seznam (pouze jeho položky)**
    - **Některým impl. (GHC) lze *zadat* optimalizační (přepisovací) pravidlo z 1.->3., ale ještě ho neumí odvodit a ani skontrolovat ekvivalenci**

map pro jiné d.s.

- **pro binární stromy**
  - **rozbor podle konstruktoru**

```
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)
```

- **n-ární stromy**

```
data NTree a = Tr a [NTree a]
mapNT :: (a->b) -> NTree a -> NTree b
mapNT f (Tr x trees) = Tr (f x) (map (mapNT f) trees)
```

- **nerekurzivní typy Maybe, Either - s variantou**

```
mapM :: (a->b) -> Maybe a -> Maybe b
mapM f Nothing = Nothing
mapM f (Just x)= Just (f x)
mapE :: (a->c)->(b->d) -> Either a b -> Either c d
mapE f g (Left x)  = Left (f x)
mapE f g (Right y) = Right(g y)
--mapE :: (a->c,b->d) -> Either a b -> Either c d
```

Obecná strukturální rekurze - seznamy

- **Tzv. foldr/svinutí - pro seznamy, doprava rekurzivní**
  - **nahrazování konstruktorů funkcemi, výsledek je lib. typu**
  - **někdy musíme výslednou hodnotu dodatečně zpracovat, viz průměr …(na dalším slajdu)**
  - **! někdy potřebujeme jiný druh rekurze (zleva, končit na 1 prvku - např. maximum, …)**
    - **foldr zpracovává prvky seznamu samostatně, např. nedostává zbylý seznam**
  - **Př: `foldr f z (1:2:3:[])` pro vstup `[1,2,3]` počítá `1`f`(2`f`(3`f` z))**

```
foldr              :: (a->b->b) -> b -> [a] -> b
foldr f z []        = z
foldr f z (x:xs)    = f x (foldr f z xs)
```

- **1. příklad**

```
length xs = foldr (\_ n->n+1) 0 xs
```

Použití `foldr` (13x)

```
length xs = foldr (\_ n->n+1) 0 xs  -- xs lze vynechat
sum xs = foldr (\x s->x+s) 0 xs
product xs = foldr (*) 1 xs
faktorial n = product [1..n]
reverse xs = foldr (\x rs->rs++[x]) [] xs  -- v O(n^2)
concat xss = foldr (++) [] xss
xs ++ ys  = foldr (:) ys xs
map f xs = foldr (\x ys->f x:ys) [] xs   -- ((:).f)
iSort cmp xs = foldr (insert cmp) [] xs -- vs. jiné sort
  where insert = …
and xs = foldr (&&) True xs -- a pod.
```

- Funkce `and` ”zdědí” líné vyhodnocování (zleva) od `&&`

```
or xs  = foldr (||) False xs
any p xs = foldr (\x b->p x||b) False xs  -- duálně: all
all p xs = foldr (\x b->p x&&b) True  xs
prumer xs = s/fromInt n where
   (s,n) = foldr (\x (s1,n1) -> (x+s1,1+n1)) (0,0) xs
```

- Počítá se složená hodnota, obvykle ji potřebujeme postzpracovat

Varianty fold pro seznamy

- **Varianty: na neprázdných seznamech `foldr1`, doleva rekurzivní `foldl`, (a analogicky `foldl1`)**

```
minimum :: Ord a => [a] -> a
minimum xs = foldr1 min xs -- fce. min nemá neutrální prvek, pro []
foldr1 :: (a->a->a) -> [a] -> a
foldr1 _ [x] = x
foldr1 f (x:xs) = x `f` foldr1 xs
```

- **Zpracování prvků zleva, pomocí akumulátoru, u konečných sezn**
  - `(..((e `f` a1) `f` a2) .. `f` an)`

```
foldl :: (a->b->a)->a->[b]->a
foldl f e [] = e
foldl f e (x:xs) = foldl f (e `f` x) xs
reverse = foldl (\xs x-> x:xs) [] -- lineární slož.
```

Striktní vyhodnocení $!

- **Měli jsme: nestriktní apl. `$ : f $ x = f x`**
- **Funkce `$!` je striktní verze aplikace funkce**
  - **`f $! x` vyhodnotí nejvyšší konstruktor `x`, potom volá f**
    - **Pokud je `x` elementárního typu (Int, Bool…), vyhodnotí se *úplně***
    - **Po vyhodnocení víme, že `x` není *bottom*, tj. ⊥**
  - **Motivace a typické použití: předcházení *memory leak***
- **`foldl` nevyhodnocuje svůj akumulátor hned – př.:**

```
length xs = foldl (\d _x -> d+1) 0 xs
length [a,b]
~> foldl .. 0 [a,b]
~> foldl .. (0+1+1) []
```

  - **problémy:**
  1. **memory leak**
  2. **při vyhodnocování hlubokého termu přeteče zásobník (Hugs: d~10000)**
- **Akumulátor chceme hned vyhodnocovat, definujeme `foldl'` pomocí `$!`**

```
foldl' f e []     = e
foldl' f e (x:xs) = (foldl' f $!(e`f`x)) xs
```

Striktní vyhodnocení 2

```
seq :: a ->  b -> b
```

- `seq` první argument vyhodnotí, dokud není znám nejvyšší konstruktor
- Vrací druhý argument

- Možná definice `$!`

```
f $! x = x `seq` f x
```

- Pro porovnání, funkce f dvou argumentů

```
> (f $! x) y
```

- Vyhodnotí striktně první argument

```
> f x $! y
```

- Vyhodnotí striktně druhý argument

```
> (f $! x) $! y
```

- Vyhodnotí striktně oba argumenty, nutné závorky

- Pozor: ve funkci `prumer` pomocí `foldl'` je hlavní konstruktor dvojice , tj. `(,)`
- Obě položky složeného akumulátoru musím vyhodnotit explicitně striktně

```
prumer xs = s/fromInt n where
    (s,n) = foldl' (\(s1,n1)x -> ((,)$!(x+s1))$!(1+n1)) (0,0) xs
```

- Výhoda: pokud zpracovávám stream nebo postupně generovaný seznam, tak režijní paměť je konst.

(Zpracování seznamů: fold a scan)

```
foldr :: (a->b->b) -> b -> [a] -> b
foldl :: (a->b->a) -> a -> [b] -> a
scanr :: (a->b->b) -> b -> [a] -> [b]
scanl :: (a->b->a) -> a -> [b] -> [a]
> foldr (:) [0] [1,2]
[1,2,0]
> foldl (flip (:)) [0] [1,2]
[2,1,0]
> scanr (:) [0] [1,2] -- zpracování přípon
[[1,2,0],[2,0],[0]]
> scanl (flip (:)) [0] [1,2] -- zpracování předpon
[[0],[1,0],[2,1,0]]
```

Obecná strukt. rekurze - stromy

```
foldT :: (a->b)->(b->b->b)->Tree a->b
```

typ konstr. `Leaf` , resp. typ `Branch` s nahrazením `b` za `Tree a`

```
foldT fL fB (Leaf a) = fL a
foldT fL fB (Branch l r) =
    fB (foldT fL fB l) (foldT fL fB r)
hloubka t = foldT(\_->1) (\x y -> 1+max x y) t
sizeT t = foldT (\x->size x) ((+).(1+)) t --\x y->1+x+y
mergeSort :: Ord a => Tree a -> [a]    -- fáze 2 mergesortu
mergeSort t = foldT (\x->[x]) merge t
```

- mergesort používá strukturální rekurzi podle stromu bez vnitřních vrcholů
- jiné třídící algoritmy používají rekurzi podle jiných d.s.

```
mergeSort' cmp t = foldT (\x->[x]) (merge' cmp)  t
```

Funkce `((+).(1+))` počítá `\x y->1+x+y`. Dk:

```
((+).(1+)) x y =              / přidání ()
(((+).(1+)) x) y =            / def (.)
((+)((1+) x)) y =             / aplikace
((+)(1+x)) y =
(+)(1+x) y =
(1+x)+y =
1+x+y ☺ počítaní s programy, v čistém programování lehké
```

Unfold – pro seznamy

- **Obecný vzor pro tvorbu seznamu (i !nekonečného), tj. rekurze podle výstupu**

```
unfold :: (b->Bool)->(b->(a,b))->b->[a]
unfold done step x  -- různé impl. stejné myšlenky
  | done x    = []
  | otherwise = y : unfold done step yr
     where (y,yr) = step x
```

  - **Pokud podmínka `done` platí, vrací `unfold` prázdný seznam, jinak funkce `step` vrací hlavu `y` a `yr` jako „zbylou část“, která se použije pro generování těla**

- **Převod čísla do binární repr. (od nejméně význ. bitů)**

```
int2bin = unfold (0==) (\x->(x`mod`2,x`div`2))
> int2bin 11 ~> 1:(i 5) ~> 1:1:i 2 ~> 1:1:0:i 1 ~> 1:1:0:1:i 0
   ~> 1:1:0:1:[]
selectSort :: Ord a => [a] -> [a]
selectSort = unfold null findMin
  where findMin [x] = (x,[])
        findMin (x:xs) = let (y,ys)=findMin xs in
          if x<y then (x,y:ys)
                 else (y,x:ys)
```

- **DC: `map f` pomocí `unfold`, `iterate f` pomocí `unfold`**

Unfold, pro stromy

- **Vzor lze navrhnout i pro jiné typy: stromy, …**
  - **Obecné schéma: `unfold f x`: fce `f` vrací jednu úroveň výstupní struktury**
    - **Různé konstruktory kóduje pomocí `Either`, případně vnořeného**

Pro seznamy typu `[a]`: `f:: b -> Either () (a,b)`

Pro stromy typu `Tree a`: `f::b-> Either a (b,b)`

Pro stromy typu `Tree2 a`: `f::b-> Either () (b,a,b)`

Na typ `b` volám `unfold f` rekurzivně

```
unfoldT2 f x = case f x of
  Left  _           -> Void
  Right (xl,v,xp) -> Node (unfoldT2 f xl) v
                          (unfoldT2 f xp)
```

Pro stromy typu `NTree a`: `f :: b -> (a,[b])`

```
unfoldNT f x = Tr v (map (unfold f) xt) where
  (v,xt) = f x
```

Počítání s programy

Typicky: dokazování vlastností programů

(Částečná) správnost vzhledem k specifikaci

Transformace programů pro optimalizaci

př.: asociativita (++) – append

[] ++ ys = ys

(x:xs) ++ ys = x: (xs++ys)

Tvrzení: x++(y++z) = (x++y)++z pro konečné x, y, z.

x=[] : LS = []++(y++z) = y++z / def. ++

PS = ([]++y)++z = y++z / def. ++

x=a:v : LS = (a:v)++(y++z) = / def. ++

a:(v++(y++z)) = / ind. předp. ++

a:((v++y)++z)

PS = ((a:v)++y)++z = / def. ++

((a:(v++y)) ++ z) = / def. ++

a:((v++y)++z) QED ☺

Testování programů, ověřování vlastností

- **Pro testování: využití vlastností programů**
  - **testování vlastností výsledku je lehčí/jednodušší než spočítání výsl.**
    - **Např. setřídění vs. test uspořádání**
  - **pokud program/funkce vlastnost nesplňuje (a má splňovat), je chybný**
  - **lze testovat uvnitř funkcí (typicky při vývoji SW)**
  - **(typy jsou taky vlastnosti; a mnohé chyby ohlídají)**
- **Příklady:**

```
reverse (reverse xs) == xs
length xs == length (map f xs)
```

  - **Asociativita, komutativita, …**
- **V Hs byly vytvořeny systémy pro testování vlastností:**
  - **a přeneseny do jiných jazyků (Java, Python, …)**
  - **Quickcheck – testování na (generovaných) náhodných datech,**
  - **Smallcheck – testování na malých datech**
    - **Využívá se referenční transparentnost, typové třídy pro generování dat**

(Strategie vyhodnocování)

redukční krok: nahrazení některého volání fce její definicí, přičemž se formální parametry nahradí aktuálními (v dosud vyhodnoceném stavu)

- **nahrazovaný výraz: tzv. redex**

redukční strategie: vybírá první redukční krok

strategie: eager (ML, Scheme) ~"volání hodnotou"

normální (λ-kalkul) ~"volání jménem"

líná (Haskell) ~optimalizovaná normální;… a další

Věta: všechny redukční strategie, pokud skončí, vydají stejný výsledek

Věta: pokud nějaká strategie skončí, pak skončí i normální (a líná)

sq x = x*x

př: sq(sq 3)

(Strategie vyhodnocování 2)

eager (pilná,dychtivá), striktní red. strategie, volání hodnotou (redexu vybírám zleva zevnitř):

sq(sq 3) ~> sq (3*3) ~> sq 9 ~> 9*9 ~> 81

- **každý argument se vyhodnocuje právě jednou**
- **všechny funkce považuje za: *striktní* f. potřebuje své args.**

normální red. strategie, volání jménem (zleva zvnějšku):

sq(sq 3)~> sq 3*sq 3 ~> (3*3)*sq 3~>9*sq 3 ~> (A)
9*(3*3)~>9*9~>81

- **nepoužité argumenty se nevyhodnocují**
- **nevýhoda: opakované vyhodnocování spol. podvýrazů (A), které vznikly z opakovaného použití formálních parametrů**

Podobná terminologie pro implementace datových struktur: pilná vs. líná

(Líná redukční strategie)

líná: jako normální, ale neopakuje vyhodnocování argumentů (z kterých vzniknou společné podvýrazy)

sq(sq 3) ~> (x:=)sq 3*x(=sq 3)~> (3*3)*x(=3*3)->9*x(=9)~>81

- **vyhodnocuje potřebné výrazy (argumenty) právě jednou**
- **vyhodnocuje lib. argument nejvýš jednou**

```
> my_if True (0/7) (7/0{-⊥ BUM-}) --líně: bezchybně
my_if p x y = if p then x else y -- dychtivě: ⊥
```

- ***grafová* redukce (cca 1986) vs. termová redukce**
- **využívá *referenční transparentnost* – nezáleží, kdy se vyhodnocuje**
- **umožňuje *nekonečné* datové struktury**
- **vyhodnocování v `case` a *pattern matching*: vyhodnotí tolik z výsledku na konstruktory, aby se dala určit jednoznačně správná větev pro další výpočet**
- **zefektivnění se týká pouze argumentů, ne lib. podvýrazů => použít `let`**
- **(má méně efektivní implementaci než striktní strategie: nejprve zapisuje neredukovaný tvar, pak redukovaný; při přístupu nutno testovat, zda je term už redukovaný=>když se nezmění sém., komp. použije eager s.)**
  - **(je nutný přepis termu na místě, protože nevím, kdo na term ukazuje)**

(Hackerský koutek, pojmy)

- **lambda funkce**
- ***closure*/uzávěr: funkce s částečně zadanými (počátečními) argumenty**
  - **Př: `zmen k xs = map(\y->(k+)y) xs`**
  - **Částečně aplikovaná funkce si musí zapamatovat svoje dané argumenty, protože může být použita v jiném kontextu (bloku), kde arg. nejsou dostupné.**
  - **To je také důvod, proč první arg. jsou metadata a parametry.**
- ***thunk*: procedura na místě očekávané hodnoty, která hodnotu vygeneruje (podobné proxy)**
  - **Líné vyhodnocování nejdřív zapíše thunk, až později hodnotu**
  - **Eager vyh. spočítá a zapíše přímo hodnotu -> je efektivnější**
  - **Thunk může být velký: memory leak**

Monády - idea

- **Monáda `M` s hodnotou typu `a`:**

```
data M a = ...
```

- **Hodnota/-y typu `a` je zabalena v "kontejneru" `M`, s přidanou informací**
  - **`M a` je typ výpočtu (např. nedet., s chybou, se stavem, IO …)**
  - **Hodnotu `a` nemůžeme vybrat a pracovat s ní přímo (hodnota s kaňkou)**
    - **Monadická hodnota po zpracování zůstane monadickou hodnotou**
  - **Monadický kód určuje pořadí vyhodnocování (důležité pro IO)**
- **Místo fce `zdvih :: (a-> M b) -> (M a -> M b)`, která se podobá `map :: (a-> M b) -> (M a -> M(M b))`, se používá `(>>=):: M a ->(a-> M b) -> M b` (bind), která přirozeně popisuje pořadí vyhodnocování (jednovláknovost) a obslouží strukturu monády `M a`**
- **Druhá funkce je `return :: a -> M a` ; vyrobí monadickou hodnotu**
- `join :: M(M a) -> M a`
- `join mma = mma >>= id`
- `ma >>= f = join $ map f ma`
- **return, join a map je alternativní definice monády**
  - **Monády v teorii mají vlastnosti (např. asociativitu), které překladač nekontroluje**

Monády

- **Monády jsou def. pomocí typových tříd**
  - **`return` a `bind` (`>>=`) jsou přetížené**
  - **Jednotlivé monády mají specifické výkonné procedury**
- **Typ Maybe je monáda s chybou**
  - **Specifická fce. je „vyvolání chyby“ - vracení `Nothing`**

```
return x = Just x
Nothing >>= f = Nothing --”chyba” se propaguje
Just x  >>= f = f a
```

- **Typ seznam je monáda pro “nedeterminizmus”**
  - **Specifická akce je vracení dvou výsledků**

```
return x      = [x]
[]      >>= f = []
(x:xs) >>= f = f x++(xs >>= f)
join          = concat -- již známe
```

- **Funkce `f::a->M b` a `g::b->M c` máme problém složit (kvůli typům), ale bind pomůže:**

```
compM f g = \x -> f x >>= g
```

  - **Potřebovali bychom `g′::M b->M(M c)`**

Monadický vstup a výstup

Vestavěný typový konstruktor `IO`: pro typ *a* je `IO` *a* typ (vstupně-výstupních) akcí, které mají vnitřní výsledek typu *a*.

`getChar :: IO Char` -- **načte znak (do vnitřního stavu)**

`putChar :: Char -> IO ()` -- **vypíše znak, výsledek je nezajímavý**

`putStr :: String -> IO ()` -- **výpis řetězce na standardní výstup**

- **Na vnitřní stav se nedostaneme přímo – porušila by se referenční transparentnost**
  - **Pokud existuje `upIO :: IO a -> a`, potom `upIO getChar` vrací různé výsledky**
- **Funkce pro převod mezi znakovou a strukturovanou reprezentací:**
  - **show :: Show a => a -> String**
  - **read :: Read a => String -> a**
    - **Funkce, které nepracují s IO, jsou čisté**
- **(funkce pro práci se soubory (otevření, výpis do, uzavření): knihovny)**
  - **IOExtensions: readBinaryFile *jmeno* :: IO a, readFile, writeFile *jmeno vystup***

IO 2

Operátor `(>>=)` `:: IO a -> (a -> IO b) -> IO b` aplikuje druhý argument (funkci) na vnitřní stav z prvního argumentu (čte se: bind)

- jednovláknový výpočet (sekvencializace), není porušena referenční transparentnost

Převod na velké písmeno:

`getChar >>= putChar . toUpper` , typ je `IO Char`

Obecně:

get*Data* >>= \ data -> putStr (show (*zpracuj* data)) , typ výrazu je `IO Char`

Operátor `(>>)` ignoruje výsledek první akce (např. výpis `putChar::IO()`)

```
(>>) :: IO a -> IO b -> IO b
o1 >> o2 = o1 >>= \ _ -> o2
```

`> putStr "Hello, " >> putStr "world"` – vypis "Hello, world"

Vytvoření prázdné akce s daným vnitřním stavem: `return :: a -> IO a`

(Příklad:)

```
sequence_ :: [IO ()] -> IO ()
sequence_ = foldr (>>) (return ())
> sequence_ (map putStr ["Hello"," ","world"])
```

-- jednoznačný výstup i při lazy vyhodnocování

(do-notace)

Program s IO

Převod 1 řádku na velká písmena

```
import IO
main :: IO ()  -- main pro kompilované programy
main =
  putStr ”Zadej vstupní řádek:\n”   >>
  getLine                           >>=
  \vstup -> putStr ”Původní řádek:\n” >>
  putStr vstup                        >>
  putStr ”\nPřevedeno na velká písmena:\n” >>
  putStr (map toUpper vstup)
```

Proveditelný program se jménem `main` a typem `IO ()` lze skompilovat a spustit. Při běhu se projeví vedlejší účinky akcí.

Monády obecně

Monády: konstruktorová třída s `return` a >>= (a >>)

Monadická hodnota obsahuje (případně) data a další informace, ale není na ně přímý přístup

Výpočet „v monádě“ způsobí sekvencializaci výpočtu, tj. pořadí výpočtů je určeno

Monády jsou:

- **`Maybe a` (např. výpočet s chybou),**
- **`[a]` (nedeterminizmus),**
- **`IO a` (I/O)**
- **`(Either b) a` (s fixovaným b v parametru a)**

```
class Monad M where  -- před GHC 8.x (ještě zobecňovali)
  return :: a -> M a
  (>>=)  :: M a -> (a -> M b) -> M b  -- bind
instance Monad Maybe where
  return x       = Just x
  Nothing >>= f = Nothing
  Just x  >>= f = f x
```

Př.: in1 se zpracuje (např. načte) před in2; pokud vznikne chyba, vrací se Nothing

```
f :: Int -> Int -> Maybe (Int, Int)
f in1 in2 = zpracuj in1 >>= \v1 ->
            zpracuj in2 >>= \v2 ->
            return (v1,v2)
```

Použití Maybe

- **Definujme `safehead`, analogicky `safetail`**

```
safehead :: [a] -> Maybe a
safehead []     = Nothing
safehead (x:_) = Just x
```

- **Bezpečný součet prvních dvou čísel seznamu, v monádě `Maybe`**

```
safeplus xs = safehead xs  >>= \x1  ->
              safetail xs  >>= \xs1 ->
              safehead xs1 >>= \x2  ->
              return (x1+x2)
```

1. **Monáda ošetřuje informace navíc (zde Nothing a Maybe)**
2. **Funkce `safehead` a `safetail` se nemění**
   1. **dostávají nechybové vstupy (původního typu)**
   2. **Kontrolují a případně chybu vyvolávají**

– **Pro porovnání: `safehead(safetail xs)` – typová chyba**

- **Možná oprava, ale…: `safehead2 :: Maybe [a] -> Maybe a`**

Další monády

- **Monáda identity, pro sekvencializaci**
  - **Ve zdrojáku je určeno pořadí zpracování**

```
newtype Id a = I a
instance Monad Id where
  return x = I x
  m >>= f  = f m
```

- **Seznamy jako monáda, pro repr. nedeterminismu**

```
instance Monad [a] where
  return x = [x]
  l >>= f  = concat (map f l)
```

  - **Samostatně jsou funkce pro vytvoření nedeterministického výsledku**
  - **Př: k danému binárnímu stromu generujeme stromy, přičemž dovolujeme prohodit větve ve vrcholu**
  - **Použitelné pro generuj a testuj, pro prohledávání stav. prostoru (f popisuje 1 krok expanze)**

Monády

- **Další použití monád: předávání kontextu, výstupu, stavu; pouze sekvencializace výpočtu (typ `Id a`), …**
  - **Většina funkcí je čistých, v monádě pouze to, co ji potřebuje**
- **Každá monáda potřebuje speciální funkce pro vyvolání efektu v monádě, např. změna kontextu, změna a update stavu, …**
  - **Každá monáda potřebuje jiné vlastní výkonné funkce, proto nejsou součástí typové třídy**
- **Speciální syntax pro monády: do-notace: idea podobná se stručnými seznamy**
  - **do-notace umožňuje psát kód podobný procedurálním jazykům (proto je důležitá a oblíbená)**
  - **Monadické zpracování běží na pozadí**
- **I pro návrh DSL/DSEL: Domain Specific Embedded Language: uživatel vidí (pouze) doménové operace**

do-notace

- **Notace je použitelná pro lib. monádu (tj. je přetížená)**

```
do v1 <- e1
   v2 <- e2
   e3          -- když návr. hodnotu nepotřebujeme
   …
   vn <- en
   return (f v1 v2 … vn)
```

- **Syntaktický cukr pro**

```
e1 >>= \v1 ->
e2 >>= \v2 ->
e3 >>= \_  ->  --neboli e3 >>
…
en >>= \vn ->
return (f v1 v2 … vn)
```

  - **vi může být anonymní proměnná, pokud ji nepotřebujeme pro výsledek**

Př.

Příklad monadického kódu

- **Hodnota `ku` se vyhodnocuje před `kv`**
- **Kód nemá specifické operace monád, funguje pro víc monád**

```
sectiM tab ku kv =
  do u <- lookup tab ku
     v <- lookup tab kv
     return (u+v)
sectiM' tan ku kv =
  lookup tab ku >>= \u ->
  lookup tab kv >>= \v ->
  return (u+v)
```

Další nástroje pro monády

Další funkce, nástroje

```
import Control.Monad  -- knihovna
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f [] = return []
mapM f (a:as) = do
    b <- f a    -- prochází zleva
    bs <- mapM f as
    return (b:bs)
```

- **mapM zřetězí monadické efekty (zleva)**

```
-- foldM …
```

(Monáda State)

Monáda pro předávání stavu

```
data State s a = S (s -> (s,a))
data State' s a = S {runS::s->(s,a)}

instance Functor (State s) where
  fmap f x = S $ \s -> (s, f x)
instance Monad (State s) where
  return x = S (\s -> (s,x))
  S fa >>= f = S (\s -> let (s1,a) = fa s
                            S  fb  = f  a
                        in fb s1)
```

Moduly

Modul, definice začíná: (Hs hledá soubor se stejným jménem)

```
module Z17 where …  -- v souboru Z17.hs
```

Použití, každý import na samostatném řádku:

```
import Z17          -- na začátku skriptu
import Z17 hiding (lift2Z17)  -- skrývání
```

Moduly, hierarchické knihovny, příklady:

```
import GHC.Base  -- místo prelude v GHC
import IO
import Data.List
import Control.Monad
```

Scheme, v kostce

- Používali jsme impl.: http://racket-lang.org/
- I dnes jsou formáty založené na syntaxi Lispu/Scheme
  - Planning Domain Definition Language, PDDL
- Máte zdarma syntaktické stromy (a jejich zpracování)

```
(define (mkTree l x r)  ; def. konstruktor stromu
  (cons 'node (cons l (cons x r))) )
(define (getVal t)
  (if (eq (car t) 'node)  ; node jako tag
      (car (cdr (cdr t)))  ; výběr složky/selektor
      '() ))  ; 'val pro (quote val),nevyhodnotit
>(map (lambda (x) (+ 1 (* x 2))) '(1 2 3))
```

Redukce - Obrázky ☹

```
> ([1,2]++[3,4])++[5,6]
> take (1+0) (″a″++ tail ″″) -- obsahuje ⊥
[]     ++ ys = ys             -- (A)
(x:xs)++ ys = x : xs++ys      -- (B)
take 0 _   = []               -- (C)
take _ [] = []                -- (D)
take n (x:xs) = x:(take (n-1) xs) -- (E)
```

Postup redukce při porovnávání se vzorem:

`t(1+0)(('a':[])++⊥) ~>`	**vyh. + pro vzor C**
`t 1(('a':[])++⊥) ~>`	**vyh. B pro vzor D**
`t 1('a':([]++⊥)) ~>`	**pravidlo E**
`'a': t (1-1)([]++⊥) ~>`	**vyh. –**
`'a': t 0([]++⊥) ~>`	**pravidlo C**
`'a':[]`	**″a″ - pouze konstruktory**

Funkce jako výsledky

typ aritm. výrazu a jeho překlad do funkce

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

```
> (Con 2 :*: Var "x") :+: Con 1
```
– *vstup skoro zdarma*

```
cc :: (Tvyraz a) -> a -> a
cc (Con a)   = \x -> a
cc (Var _)   = \x -> x
cc (a :+: b) = \x -> (cc a x) + (cc b x)
cc (a :*: b) = \x -> (cc a x) * (cc b x)
```

ekv. `cc (a:+:b) x = cc a x + cc b x`

Na rozdíl od Scheme nerozlišujeme které „úrovni“ funkce se předává argument – díky curryfikaci

v Hs. jsou ekv. `b->c->d` a `b->(c->d)`

(implementačně: optimalizace vyhodnocování podvýrazů pomocí *full laziness*)

Autotest

Hornerovo schema

je daný seznam koeficientů, “vyrobte” k němu funkci, která pro daný vstup x spočte hodnotu v bodě x

Array

- Pole, např. pro dynamické programování
- Hodnoty v poli se nemění
- Třída pro indexy pole: Ix

```
class (Ord a) => Ix a where
 range :: (a,a) -> [a]
```

  - Generuje všechny indexy

```
 index :: (a,a) a -> Int
```

  - Vrátí pořadové číslo indexu v poli

```
 inRange :: (a,a) -> a -> Bool
```

  - Testuje, zda je index v platném rozsahu, po komponentách

- První arg. jsou (dolni,horni) meze, i složené: (0,9) , ((1,1),(10,10))
- V modulu Array, resp. Data.Array:

```
> :m +Array
```

Array

- **Typ pole:**

```
array :: (Ix a) => (a,a) -> [(a,b)]->Array a b
```

- **Příklad vytvoření pole**
  - **Druhý par. je asociační seznam (index, hodnota). Indexy v něm se nesmí opakovat**

```
squares = array (1,100) [(i,i*i) | i<-[1..100]]
```

- **Indexace, pomocí !**

```
> squares!7
49
```

- **Meze**

```
> bounds squares
(1,100)
```

Použití Array

- **Příklad použití**

```
fibs :: Int -> Array Int Int
fibs n = a where
    a = array (0,n) ([(0, 1), (1, 1)] ++
      [(i, a!(i-2) + a!(i-1)) | i<-[2..n]]
```

- **Líné vyhodnocování se postará o správné pořadí vyhodnocování**

Akumulační pole

- **V ak.poli se indexy můžou opakovat, proto určíme *akumulační* funkci**

```
accumArray :: (Ix a) =>
   (b->c->b)->b->(a,a)->[(a,c)]->Array a b
```

  - **1.par : akumulační fce**
  - **2.par : počáteční hodnota, pro všechny položky stejná**

- **Pro histogram je akumulační funkce sčítání**

```
hist :: (Ix a, Integral b) =>
    (a,a) -> [a] -> Array a b
hist bnds is =
   accumArray (+) 0 bnds
     [(i, 1) | i <- is, inRange bnds i]
```

Incremental update

- **Popíšeme změny v poli: seznam (index, nová_hodnota)**

```
(//) :: (Ix a) => Array a b->[(a,b)]->Array a b
```

- **Přehození řádků (méně naivně)**

```
swapRows i i' a =
  a // [assoc | j <- [jLo..jHi],
                assoc <- [((i ,j), a!(i',j)),
                          ((i',j), a!(i, j))] ]
  where ((iLo,jLo),(iHi,jHi)) = bounds a
```

  - **Progr. trick: v `assoc` jsou jednotlivé změny, vyhnu se concat**

- **DC: násobení matic (genMatMult)**
  - **Klasicky, ale operace sčítání a násobení jsou jako parametry:**
    - **genMatMult sum (*) , gMM minimum (+) , gMM maximum (-)**

(Dodatky)

I/O, show, read, do-syntax

Neodmítnutelný vzor ~

(moduly)

(Návrhové vzory)

Fft: za typové třídy

příklady: beamsearch

Typování: map_m

Explicitní typy pro výraz

$$map\ (map\ f)\ x$$

Aktuální typy jsou v horních indexech

$$map^{([a]\to[b])\to[[a]]\to[[b]]}$$

$$\underbrace{(map^{(a\to b)\to[a]\to[b]}\ f^{a\to b})}_{[a]\to[b]}\ x^{[[a]]}$$

Výraz je typu `[[b]]`

Podmínky: if, stráže

```
fact3 :: Int -> Int
-- neošetřuje záporné vstupy
fact3 n = if n==0 then 1
          else n*fact3 (n-1)
```

! if má vždy větev else (, ve které musíte vrátit nějaký výsledek)

```
fact2 n | n==0 = 1
        | n>0  = n*fact2 (n-1)
        | otherwise = error "in: fact2"
```

První pravdivá stráž aktivuje klauzuli a vrací hodnotu za '=', bez backtrackingu.

Stráže jsou speciální (ne-výrazová) syntax, používaná při definice funkce.

Porovnávání se vzorem 1

- **pattern matching (není to unifikace, pouze "jednostranná")**
  - ***implicitní podmínka* na místě *formálního* parametru na tvar dat. struktury, tj. na datové konstruktory**
    - **Aktuální parametry se musí dostatečně vyhodnotit, aby šla podmínka rozhodnout**
      - **tj. pattern matching vynucuje vyhodnocení potřebné části struktury**
    - **Ve FP (na rozdíl od Prologu) jsou aktuální parametry konkrétní struktury (ale ne nutně vyhodnocené)**
  - **Lze použít i pro uživatelsky definované typy**
  - **Pojmenuje složky aktuálních argumentů -> nepotřebuju selektory (obvykle)**
    - **Jméno proměnné ve vzoru je definiční výskyt**

```
length1 :: [a] -> Int
length1  []     = 0
length1 (x:xs) = 1 + length1 xs --závorky nutné
> length1 ((ys++[])++((1+2):zs))
```

- ys=[] ~> `length1([]++(…))` ~> `length1 ((1+2):zs)`
- ys=u:us ~> `length1(u:((us++[])++(…))`

Bloková struktura - let

Let - (vnořená) množina vazeb pro výraz

- **vazby jsou vzájemně rekurzivní**

konstrukce `let` je výraz

```
f x y = let norma (x,y) = sqrt (x^2+y^2)
            prumer x y = (x+y)/2.0
        in prumer (norma x) (norma y)
```

- **let je použitelné pro optimalizaci společných podvýrazů**

Bloková struktura - where

lokální vazby přes několik střežených rovností

```
f x y | y >z  = ...
      | y==z  = ...
      | y <z  = ...
            where z = x*x
```

Rozložení, layout

Dvourozměrná syntax - předpoklad:

definice jsou zarovnány do sloupců

za klíčovými slovy: where, let, of

první znak určuje sloupec (indentaci)

jakýkoli znak nalevo končí vnoření

explicitní skupiny definic

```
let {a=1 ; b=2} in a+b
```

Příklad: slévání

- **příklad použití vzoru @**

```
merge                   :: Ord a => [a]->[a]->[a]
merge []      ys        = ys
merge xs      []        = xs
merge u@(x:xs) v@(y:ys)
        | x <= y        = x : merge xs v
        | otherwise     = y : merge u  ys
```

(Stavba dopředného stromu AC)

Dávkové zpracování, pomocí rozkladu na třídy, které začínají stejným písmenem.

```
-- ac :: (Char,Maybe b)->[([Char],b)]->NTree (Char,Maybe b)
-- na vstupu jsou neprázdná slova
ac (c,i) ss = Tr(c,i)(map(\(ekv,dv)->ac dv ekv)(rozklad1 ss)) where
  rozklad1 xs = map pp (rozkladEkv pp2 xs) -- rozklad a úprava tříd
  pEmpty (x,_) = x==[]
  stripChar (x,y) = (tail x,y)
  pp xs = let xs1 = map stripChar xs -- ubrání prvního znaku slov
              ff1 = filter pEmpty xs1 -- slovo, které skončí
              ff2 = filter (not.pEmpty) xs1 -- slova, která pokračují
          in (ff2,
              ( if xs==[] then ' ' else head(fst(head xs)), -- první znak slov, spol. třídě
                if ff1==[] then Nothing else Just (snd(head ff1)) )) -- info o slově
  pp2 (x,ix)(y,iy) = head x==head y -- popis ekvivalence, tj. shoda prvního písmene

rozkladEkv::(a->a->Bool)->[a]->[[a]]
rozkladEkv f [] = []
rozkladEkv f (x:xs) = (x:filter (f x) xs) : -- x patří do své třídy
  rozkladEkv f (filter(\y->not(f x y))xs) -- (not.f x), rekurze na zbylé prvky
filter p [] = [] -- prelude: vybere prvky, které splňují podmínku p
filter p (x:xs) = if p x then x:filter p xs
                         else filter p xs
? ac ('x',Nothing) (map(\x->(x,x))["he","she","her","his"]
Tr ('x',Nothing) [Tr ('h',Nothing) [Tr ('e',Just "he") [Tr ('r',Just "her") []],
                                    Tr ('i',Nothing)   [Tr ('s',Just "his") []] ],
                  Tr ('s',Nothing) [Tr ('h',Nothing)   [Tr ('e',Just "she") []]] ]
```

Funkce vyšších řádů: map

map f xs aplikuje funkci f na každý prvek seznamu xs a vrací seznam odpovídajících výsledků

```
map :: (a->b) -> [a] ->[b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
> map succ [1,2,3]  ~~> [2,3,4]
> map (add 2) [3,4]  ~~> [5,6]
> map (+) [7,8]      ~~> [(7+),(8+)]
```

- **Použitelné i pro "duální" přístup: jedny data, mnoho funkcí**

```
fitness :: [a->b] -> a -> [b]
fitness kritFce data = map (\f-> f data) kritFce
```

Použití map na seznam kriteriálních funkcí

filter

(ze souboru standard.pre)

filter p xs returns the sublist of xs

-- containing those elements which satisfy the predicate p.

```
filter              :: (a -> Bool) -> [a] -> [a]
filter _ []          = []
filter p (x:xs)
    | p x            = x : xs'
    | otherwise      = xs'
                       where xs' = filter p xs
```

Příklad: zip

-- zip z1 z2 -- ze dvou seznamů prvků vyrobí seznam dvojic

-- seznamy mohou být různě dlouhé,

-- zip skončí, když skončí jeden z argumentů

-> kompatibilní s líným vyhodnocováním

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _      _      = []
```

-- zipWith: zobecnění zip – skládací procedura daná parametrem

```
zipWith :: (a->b->c)-> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith xs ys
zipWith f _      _      = []
```

map - varianty

synchronní přechod přes dva seznamy

```
zipWith :: (a->b->c) -> [a]->[b]->[c] -- prelude
zipWith f x y = map (\(x,y)-> f x y) (zip x y)
```

př: plus_vektor x y=zipWith (+) x y "lift" operací na vekt.

operace s konstantou

```
map_c :: (a->b->c)-> a->[b]->[c]
map_c f c x = map (\x->f c x) x
```

lokální x v definici

```
map_c f c x = zipWith f (map(\ _->c)x) x -- (const c)
```

generujeme pomocný vektor ze samých konstant

pro matice

```
map_m :: (a->b)-> [[a]] -> [[b]]
map_m f x = map (map f) x
```

Rekurze a iterace: Factorial

```
fact 0      = 1
fact (n+1) = (n+1)*fact n
? fact 4
    fact 4
~~> (3+1)* fact 3
~~> 4* fact 3
~~> 4*(3*fact 2)
~~> 4*(3*(2*fact 1))
~~> 4*(3*(2*(1* fact 0)))
~~> 4*(3*(2*(1* 1 )))
~~> 24
```

pracuje v lineární paměti

Faktorial - iterativne

```
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)
? fact5 4
    fact5 4
~~> fact_iter 4 0 1
~~> fact_iter 4 1 1
~~> fact_iter 4 2 2
~~> fact_iter 4 3 6
~~> fact_iter 4 4 24
~~> 24
```

pracuje v konstantní paměti

DC: Fibonacciho čísla, stav: seznam nebo “dvojice”

Morfologie

- Morfologická analýza slova

```
type TSlovnik = (String, String, Foo) --(lema, kmen, vzor)
type TVzor = (Foo, [String])
morf :: TSlovnik -> TVzor -> String -> [(String, String)]
  -- slovo -> lema, koncovka
morf sl vz slovo = […
```

Příklady funkc. idiomů

- **Pracujeme s dvojicemi, na první a druhou složku používáme různé funkce**

```
(***) f g (x1,x2) (y1,y2) = (x1`f`y1, x2`g`y2)
sectiASpoj = (+) *** (++)
> sectiASpoj (1,"a") (2,"b")
  (3,"ab")
```

- **Lexikografické složení dvou uspořádání**

```
data Ordering = LT | EQ |GT
cmpLex :: Ordering -> Ordering -> Ordering
cmpLex EQ c2 = c2
cmpLex c1 _  = c1
```

Kompoziční programování

Zpracováváme celé datové struktury (! za vzniku nových d.s.)

```
-- vyhodnocení polynomu v bodě (v O(n))
evalPoly ks x =
   sum $  zipWith (*) (reverse ks) $  iterate (x*) 1
      -- nepouzili jsme „k“
… = foldr (+) 0
          (zipWith (\k xn -> k*xn) (reverse ks)
              (iterate (\xn -> x*xn) 1) )
```

izotopické latinské čtverce – simulace backtrackingu

… or [and […

-- funkce jako výsledky

př: doplňování „adres“, doslovný překlad

Generuj a testuj: batoh, a pod.

Unfold – idea:varianta 2

- **Obecný vzor pro tvorbu seznamu (i nekonečného)**

```
unfold :: (b->Bool)->(b->a)->(b->b)->b->[a]
unfold p h t x  -- různé impl. stejné myšlenky
  | p x         = []
  | otherwise = h x : unfold p h t (t x)
```

  - **Pokud podmínka `p` platí, vrací `unfold` prázdný seznam, jinak funkce `h` vrací hlavu a `t` „zbylou část“, která se použije pro generování těla**
- **Převod čísla do binární repr. (od nejméně význ. bitů)**

```
int2bin = unfold (==0) (`mod`2)(`div`2)
```

- **DC: map f pomocí unfold, iterate f pomocí unfold**
  - **selectSort, permutace v přirozeném uspořádání**
- **Vzor lze navrhnout i pro jiné typy: stromy, …**

  Obecné schéma: `unfold f x`: fce `f` vrací jednu úroveň výstupní struktury

  Pro seznamy typu `[a]`: `f:: b -> Either () (a,b)`

  Pro stromy typu `Tree a`: `f :: b-> Either a (b,b)`

  Pro stromy typu `Tree2 a`: `f :: b-> Either () (b,a,b)`

  Na typ `b` volám `unfold f` rekurzivně

Doplnit

- **doplnit :**

```
data Ordering = LT | EQ | GT
```

- **doplnit :**
- **doplnit : …**
- **Closure**
- **Memory leak: bere víc paměti než je „*zamýšlená sém.*“**
  1. **Drží dlouho velkou strukturu (->generuj víckrát (funkcí))**
  2. **Nevyhodnocuje argument (např. foldl)**
  3. **Drží pointr v nevyhodnocené struktuře (`x=fst(a,b)`)**

Příklady typů 2

- **Výrazy**

```
data Expr a = Con a  -- konstanty
            | Var String  -- proměnné
            | Plus (Expr a) (Expr a) -- výrazy
            | Fce String [Expr a]
```

- **Analogicky reprezentace pro logické nebo regulární výrazy**