

IB015 Neimperativní programování

Jiří Barnat

Organizace kurzu

Kurz IB015

- Zakončen zkouškou.
- 6 kreditů ($2/1/1+zk$) = 180 hodin = 22 pracovních dnů

Přednáška

- Fyzicky prezenčně dle rozvrhu.
- Výuková videa v interaktivní osnově.

Cvičení

- Se 14 denní periodou v určené časové sloty.
- Týdně alternují suché a liché skupiny.
- Výjimka – první týden workshopy Naskoč na FI.

Samostatné domácí úlohy

- Zadávány skrze interaktivní osnovu předmětu.

Závěrečná písemná zkouška

- Povinná část – Test Minimálních Dovedností (TMD)
- Nepovinná část, možno získat 10 bodů.

Domácí úlohy

- Odpovědníky v ISu. Úkol = 5 odpovědníků.
- První úkol za 1.5 bodu na *nultou kapitolu sbírky*.
- Poté 5 domácích úloh za 2.5 bodu.
- Celkem bude možné získat 14 bodů.

Forma DÚ

- Teoretické vs. Praktické.
- **Teoretické možno otevřít jednou, na omezenou dobu.**
- Praktické možno odevzdat opakovaně (až 5x).
- *ChatGPT free.*

Neúspěšné ukončení

- Nezískání alespoň 8 bodů z DÚ – hodnocení X.
- Nesplnění TMD – hodnocení F.
- Splnění TMD, nezískání 10 a více bodů – hodnocení F.
- Opravné termíny na TMD dle SZŘ.

Úspěšné ukončení

- Zisk alespoň 8 bodů z DÚ.
- Splnění TMD.
- Znamka dle počtu bodů za DÚ a zkoušku:
E:[10,12) D:[12,14) C:[14,17) B:[17,20) A:[20,24]
- 12+ bodů z DÚ a úspěšné ukončení = odmazání jednoho F.

Cíle kurzu

- Studenti se seznámí s funkcionálním a logickým paradigmatem programování, díky čemuž se odprostí od imperativního způsobu uvažování o problémech a jejich řešení.
- V rámci kurzu se studenti blíže seznámí s funkcionálním programovacím jazykem Haskell a s logickým programovacím systémem Prolog.

Schopnosti absolventa

- Je schopen dekomponovat výpočetní problém na jednotlivé funkce a tuto schopnost používá při vytváření vlastních kódů i v imperativních programovacích jazycích.
- Umí efektivně použít prvky funkcionálního programování v imperativních programovacích jazycích.
- Má základní znalost programovacích jazyků Haskell a Prolog.
- Rozumí způsobu popisu programů ve funkcionálním a logickém výpočetním paradigmatu.
- Umí oddělit CO od JAK.

Předpoklady

- Možné úspěšně absolvovat bez znalosti programování.
- Schopnost abstraktního myšlení.
- Základní počítačová gramotnost (Unix/Linux OS).

Znalost imperativního programování

- Je výhodou pro pochopení rozdílného způsobu myšlení v imperativním a neimperativním světě.
- Může být zpočátku mentální bariérou.

Funkcionální paradigma

- <http://www.haskell.org/>
- <http://haskell.cz/>
- Thompson, Simon. Haskell: the craft of functional programming.

Logické paradigma

- <http://lpn.swi-prolog.org>
- Nerode, Shore: Logic for Applications

Co znamená programovat?

Programování

- Vytvoření a zápis postupu řešení problému s takovou úrovní detailů a přesnosti, aby tento popis mohl být mechanicky vykonáván strojem, zejména počítačem.
- Zápis postupu = zdrojový kód programu.
- Zdrojový kód programu je uložen v textovém souboru.

Programovací jazyk

- Uměle vytvořený jazyk pro přesný a jednoznačný zápis programů člověkem.

Schopnost programovat

- **Mentální schopnost nacházet mechanicky proveditelné postupy za účelem řešení daného problému.**
- Schopnost přesně formulovat postupy v daném programovacím jazyce.

Volba a znalost programovacího jazyka

- Programovacích jazyků je mnoho.
- Volba programovacího jazyka klade omezení na způsob formulace zamýšlených postupů.

Riziko a klam moderní doby

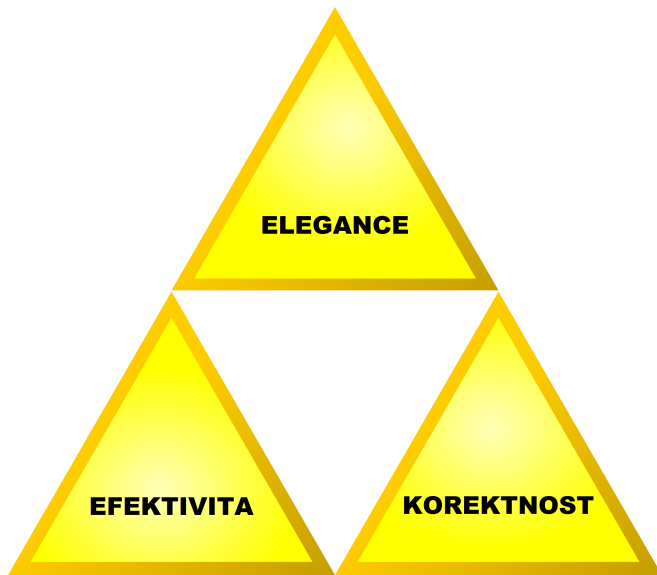
- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné poznání syntaxe a sémantiky programovacího jazyka nedělá dokonalého programátora.

Riziko a klam moderní doby

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné poznání syntaxe a sémantiky programovacího jazyka nedělá dokonalého programátora.

Nedokonalé vs. dokonalé





Klasifikace

- Imperativní — C/C++, Java, Python, ...
- Funkcionální – **Haskell**, OCaml, ...
- Logické – **Prolog**, ...



Jakým jazykem mluví počítač?

- Strojový kód. Program ve strojovém kódu je posloupnost čísel.
- Pro spuštění programu je potřeba provést překlad zdrojového kódu programu do strojového kódu procesoru.
- Překlad se realizuje pomocí **překladače** nebo **interpretu**.
- Pro každý programovací jazyk je potřeba jiný překladač/interpret.

Překladač

- Pro soubor se zdrojovým kódem programu vytvoří soubor obsahující popis programu ve strojovém kódu.
- Výsledný soubor je spustitelný.
- Pracuje se soubory.

Interpret

- Pro daný výraz / příkaz vytvoří odpovídající překlad do strojového kódu a ihned jej provede.
- Nevytváří výsledný spustitelný soubor.
- Často má možnost pracovat interaktivně.
- Pracuje s jednotlivými příkazy/výrazy.

Programovací jazyk Haskell

- Překladač – `ghc`.
- Interaktivní interpret – `ghci`.

Překladače programovacího jazyka C/C++

- GNU C++ Compiler (`g++`, `gcc`)
- Intel C++ Compiler
- Microsoft Visual C++ Compiler

Programujeme pomocí funkcí

Funkce v programování

- Funkce je předpis jak z nějakého vstupu vytvořit výstup.
- Transformace vstupů na výstupy musí být jednoznačná.

Příklady funkcí

- $f(x) = x * (x + 2)$
- `objemKvadr(a, b, c) = a * b * c`
- ...

Typ funkce

- Vymezení objektů, se kterými daná funkce pracuje a které vrací na výstup, je součástí definice funkce. Mluvíme o tzv. **typu funkce**.


Příklady

- Funkce, která otočí obrázek o 90 stupňů směrem vpravo.
`rotuj90vpravo :: Obrázek -> Obrázek`
- Objem kvádru.
`objemKvadrů :: Číslo × Číslo × Číslo -> Číslo`
- Počet hran polygonu.
`hranyPolygonu :: Polygon -> CeléČíslo`



Předpoklady


rotuj90vpravo :: Obrázek -> Obrázek

hranyPolygonu :: Obrázek -> CeléČíslo

 :: Obrázek

Aplikace funkcí

rotuj90vpravo  \rightsquigarrow 

hranyPolygonu  \rightsquigarrow 3

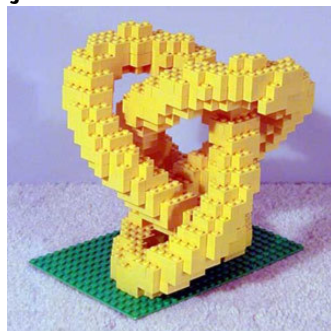
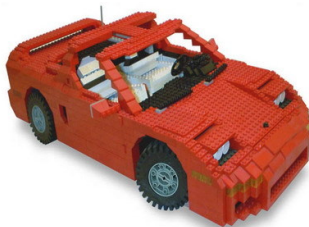
Pozorování

- Složitější úkony lze realizovat pomocí jednodušších operací.
- Složitější funkce lze definovat **složením** jednodušších.

Pozorování

- Složitější úkony lze realizovat pomocí jednodušších operací.
- Složitější funkce lze definovat **složením** jednodušších.

Skládání – cesta ke složitějším objektům a funkcím





Operátor .

- $(f1 . f2) x = f1 (f2 x)$
- Čteme jako „f1 po f2“.

Příklad

- Předpokládejme funkci `zdvoj`, která vezme obrázek a vytvoří nový zkopírováním vloženého obrázku dvakrát vedle sebe.

`zdvoj :: Obrázek -> Obrázek`

`zdvoj`  \rightsquigarrow 

- Novou funkci `rotujAZdvoj` můžeme definovat takto:

`rotujAZdvoj :: Obrázek -> Obrázek`



`rotujAZdvoj x = (zdvoj . rotuj90vpravo) x`


`rotujAZdvoj`  \rightsquigarrow 


(rotujAZdvoj . rotujAZdvoj) \triangle \rightsquigarrow

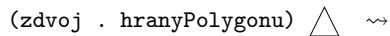
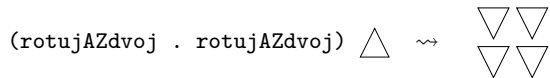
((zdvoj . zdvoj) . zdvoj) \triangle \rightsquigarrow



(zdvoj . hranyPolygonu) \triangle \rightsquigarrow

`(rotujAZdvoj . rotujAZdvoj)`  \rightsquigarrow 


`((zdvoj . zdvoj) . zdvoj)`  \rightsquigarrow


`(zdvoj . hranyPolygonu)`  \rightsquigarrow

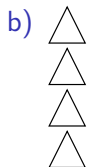
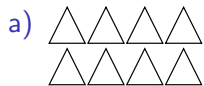



`(rotujAZdvoj . rotujAZdvoj)`  \rightsquigarrow 

`((zdvoj . zdvoj) . zdvoj)`  \rightsquigarrow 

`(zdvoj . hranyPolygonu)`  \rightsquigarrow **ERROR**

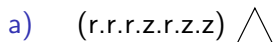
Jak pomocí zdvoj, rotuj90vpravo a  vyrobit následující?



Jak pomocí zdvojení, rotace 90° vpravo a  vyrobit následující?



Řešení



Složené funkce a η -redukce

- Složení funkcí je možné definovat bez uvedení parametru.
- Tj. definici

`rotujAZdvoj x = (zdvoj.rotuj90vpravo) x`

lze zapsat také jako

`rotujAZdvoj = zdvoj.rotuj90vpravo`

POZOR na prioritu vyhodnocování v Haskellu

- Aplikace funkce na parametry má nejvyšší prioritu.

`zdvoj.rotuj90vpravo \triangle = zdvoj.(rotuj90vpravo \triangle) \rightsquigarrow ERROR`

- Závorky kolem výrazu `zdvoj.rotuj90vpravo` jsou při aplikaci na hodnotu \triangle nutné.

Deklarace typu funkce (typová signatura):

rotujAZdvoj :: Obrázek ->Obrázek

Jméno funkce

rotujAZdvoj x = (zdvoj.rotuj90vpravo) x

Tělo funkce

rotujAZdvoj x = (zdvoj.rotuj90vpravo) x

Definice funkce

rotujAZdvoj x = (zdvoj.rotuj90vpravo) x

Formální parametr

rotujAZdvoj x = (zdvoj.rotuj90vpravo) x

Aktuální parametr

rotujAZdvoj △

Výraz

rotujAZdvoj △

Podvýraz

rotujAZdvoj (rotujAZdvoj △)

Funkcionální programování v Haskellu

Funkcionální výpočetní paradigma

- program = výraz + definice funkcí
- výpočet = úprava (zjednodušení) výrazu
- výsledek = hodnota (nezjednodušitelný tvar výrazu)

Příklad programu

- definice funkcí

```
square x = x * x
```

```
pyth a b = square a + square b
```

- výraz

```
pyth 3 4
```

Program

- definice funkcí

```
square x = x * x
```

```
pyth a b = square a + square b
```

- výraz

```
pyth 3 4
```

Výpočet

$$\begin{aligned} \underline{\text{pyth 3 4}} &\rightsquigarrow \underline{\text{square 3}} + \text{square 4} \rightsquigarrow 3 * 3 + \underline{\text{square 4}} \rightsquigarrow \\ &\rightsquigarrow \underline{3 * 3} + 4 * 4 \rightsquigarrow 9 + \underline{4 * 4} \rightsquigarrow \underline{9 + 16} \rightsquigarrow \\ &\rightsquigarrow 25 \end{aligned}$$

Lokální definice

- Definují symboly (funkce, konstanty) pro použití v jednom výrazu, vně tohoto výrazu jsou tyto symboly nedefinované.
- Lokální definice mají vyšší prioritu než globální definice.
- V jazyce Haskell možno dvěma způsoby:
 - **let** lokální definice **in** výraz
 - definice **where** lokální definice

Příklad lokální definice pomocí `let ... in`

- `let fcube x = x * x * x in fcube 12`
`let fcube x = x * x * x in let c = 12 in fcube c`
`let fcube x = x * x * x; c = 12 in fcube c`
- Konstrukce **let ... in** vytváří nový výraz.

Příklad lokální definice pomocí `where`

- Konstrukce `where` musí být součástí definice, nevytváří výraz.
- **Úkol:** Zjisti obvod pravoúhlého trojúhelníku z délek odvěsen.
- `obvodZOdvesen :: Float -> Float -> Float`

```
obvodZOdvesen a b = a + b + c
```

```
  where square x = x * x
```

```
        c = sqrt (square a + square b)
```

Pravidla pro zarovnání

- Dlouhé řádky se zalomí a pokračují s odsazením.
- Definice v jednom rozsahu musí začínat na stejném sloupku.
- Odsazení mohou libovolně zvětšovat, nemohou zmenšovat, chci docílit maximální čitelnost a srozumitelnost kódu.

Čísla

- `Integer` – libovolně velká celá čísla
- `Int` – celá čísla do velikosti slova procesoru
- `Float` – reálná čísla
- `Rational` – racionální čísla

Znaky a řetězce

- `Char` – znak, příklady hodnot: `'a'`, `'2'`, `'>'`
- `String` – řetězec, například: `"Toto je řetězec."`
- `String` je totéž co `[Char]`

Pravdivostní hodnoty

- `Bool`
- Typ `Bool` má pouze 2 hodnoty: `True` a `False`

Příklad

- Definujte funkci `jedna_nebo_dva`, která vrátí `True` pokud dostane na vstupu číslo 1 nebo 2, jinak vrátí `False`.

```
jedna_nebo_dva :: Integer -> Bool
jedna_nebo_dva 1 = True
jedna_nebo_dva 2 = True
jedna_nebo_dva _ = False
```

Definice funkcí s více definičními rovnostmi

- Na místě formálních parametrů se použijí tzv. vzory.
- Použije se první vzor, který vyhovuje, nic jiného.
- Symbol `_` vyhovuje libovolnému parametru.
- Lze použít pro větvení výpočtu.

Podmíněný výraz

- if *podmínka* then *výraz1* else *výraz2*
- *podmínka* – výraz, který se vyhodnotí na hodnotu typu `Bool`
- *výraz1* se vyhodnotí pokud se podmínka vyhodnotí na hodnotu `True`, *výraz2* se vyhodnotí, pokud se podmínka vyhodnotí na hodnotu `False`.
- Výrazy *výraz1* a *výraz2* musejí být stejného typu.

Test na rovnost

- Pro dotaz na rovnost používáme symbol `==`.
- `3 == 4` \rightsquigarrow `False`
- `3 = 4` \rightsquigarrow **Error**

Možnosti zápisu binárních funkcí

- Prefixový zápis binárních funkcí: $(+) 3 4$, $\text{min } 3 5$
- Infixový zápis binárních funkcí: $3+4$, $3 \text{ 'min' } 5$

Volání funkce a parametry

- Jméno funkce a použité parametry jsou odděleny mezerou, pokud je některý z parametrů výraz, který je sám o sobě aplikace funkce na argumenty, je třeba celý tento výraz ozávkovat.
- $(*) 3 4 + 5 \rightsquigarrow 17$
- $(*) 3 + 4 5 \rightsquigarrow \text{Error}$
- $(*) 3 (+) 4 5 \rightsquigarrow \text{Error}$
- $(*) 3 ((+) 4 5) \rightsquigarrow 27$

Co to je?

- Úkol, který je možné vyřešit na základě faktů doposud uvedených na přednáškách.
- Slouží ke kontrole, že zvládám to, co bych už měl(a) umět.

Checkpoint

- V programovacím jazyce Haskell napište funkci, která bude řešit dělitelnost dvou celočíselných čísel, tj. pro své dva celočíselné argumenty dělence a dělitele, rozhodne, zda je zadaný dělenec dělitelný beze zbytku zadaným dělitelem a to tak, že nepoužije operace pro dělení / ani počítání zbytku po dělení `mod`, je povoleno použít operaci celočíselného dělení `div`.
- Funkci otestujte s použitím interpretu jazyka Haskell.

IB015 Neimperativní programování

Programování a data

Jiří Barnat
Libor Škarvada

Pozorování

- Programy pro své fungování potřebují různé informace – **data**.
- Data jsou vstupní hodnoty, výstupní hodnoty, mezivýsledky výpočtů, parametry funkcí, atd.

Programování a data

- Data je třeba uchovávat tak, aby je bylo možné zpracovat mechanicky/strojově.
- Tvorba jednoznačného popisu struktury a **způsobu uložení dat je nedílná součást procesu programování.**

Dekompozice dat

- Veškerá data použitá v programu je třeba vystavět ze základních datových elementů podle definovaných pravidel.
- Existují striktní pravidla pro dekompozici dat, my si však v rámci IB015 vystačíme s intuicí.

Základní datové elementy

- Čísla, Znaky, Pravdivostní hodnoty

Základní způsoby kompozice dat

- **Uspořádané n-tice**
- **Seznamy**
- Stromy

Co to je

- Pevně daný počet nějakých hodnot v pevně daném pořadí.
- Prvek kartézského součinu nosných množin.

Příklady

- Datum: (11, "březen", 1977) .
- Přihlašovací údaje: ("xbarnat", "mam-IQ-tykve")
- Pozice pixelu v rastrovém obrázku: (x, y) ,
všimněme si, že (12,43) \neq (43,12) .

Kdy se má použít

- **Počet prvků v n-tici je znám předem**,
tj. v okamžiku psaní zdrojového kódu.
- Počet prvků v n-tici je malý (hodnota n je malá).

Co to je

- Posloupnost hodnot stejného charakteru (stejného typu).
- Posloupnost může být prázdná, konečná i nekonečná.
- Každý prvek v seznamu je na nějaké (unikátní) pozici.

Příklady

- Seznam čísel: [12,43,-3,15,29]
- Nekonečný seznam přirozených čísel: [1,2..]
- Seznam uspořádaných dvojic:
[("Fero",12), ("Nero",7), ("Pero",5)]
- Prázdný seznam: []

Kdy se má použít

- **Data vznikají nebo se zpracovávají postupně.**
- Počet prvků použitých programem není předem znám.

Aplikace – Diář squashových partnerů.

- Program pro správu kontaktů na různé squashové hráče.
- Hlavní datová struktura je seznam kontaktů.

Datová dekompozice

Aplikace – Diář squashových partnerů.

- Program pro správu kontaktů na různé squashové hráče.
- Hlavní datová struktura je seznam kontaktů.

Datová dekompozice

- Seznam kontaktů – [Kontakt]
[kontakt1, kontakt2, kontakt3, ..., kontakt315]

Aplikace – Diář squashových partnerů.

- Program pro správu kontaktů na různé squashové hráče.
- Hlavní datová struktura je seznam kontaktů.

Datová dekompozice

- Seznam kontaktů – [Kontakt]
[kontakt1, kontakt2, kontakt3, ..., kontakt315]
- Kontakt je uspořádaná trojice
(Prezdivka, Telefon, Adresa)

Aplikace – Diář squashových partnerů.

- Program pro správu kontaktů na různé squashové hráče.
- Hlavní datová struktura je seznam kontaktů.

Datová dekompozice

- Seznam kontaktů – [Kontakt]
[kontakt1, kontakt2, kontakt3, ..., kontakt315]
- Kontakt je uspořádaná trojice
(Prezdivka, Telefon, Adresa)
- Adresa je uspořádaná pětice
(Jmeno, Prijmeni, Ulice, Cislo Popisne, Mesto)

Aplikace – Diář squashových partnerů.

- Program pro správu kontaktů na různé squashové hráče.
- Hlavní datová struktura je seznam kontaktů.

Datová dekompozice

- Seznam kontaktů – [Kontakt]
[kontakt1, kontakt2, kontakt3, ..., kontakt315]
- Kontakt je uspořádaná trojice
(Prezdivka, Telefon, Adresa)
- Adresa je uspořádaná pětice
(Jmeno, Prijmeni, Ulice, Cislo Popisne, Mesto)
- Prezdivka, Jmeno, Prijmeno, Ulice, Mesto jsou seznamy znaků
- Telefon, Cislo Popisne jsou čísla

Hodnoty a Typy



Co je to typ

- Označení množiny všech hodnot dané kvality.
- Komunikační prostředek napomáhající správnému skládání programů z jednotlivých funkcí.

K čemu se používají typy

- Každá hodnota, nebo výraz má svůj typ.
- Definice typové signatury funkcí.
- Kontrola logické konzistence programu v době překladu.
- Popis způsobu kompozice složených datových struktur, protože **typy se komponují podobně jako data.**

Základní datové typy

- `Int`, `Integer`, `Float`, `Char`, `Bool`

Složené typy

- Uspořádané n-tice:

`(Bool, Int)`

- Seznamy:

`[Int]`, `[Char]`, `[[Char]]`

`[Char] ≡ String`

Funkcionální typy

- `Integer -> Bool`, `Float -> Float -> Float`

Konstrukce

- Jsou-li σ a τ nějaké typy, tak $\sigma \rightarrow \tau$ je typ všech funkcí s parametrem typu σ a funkční hodnotou typu τ .

Typ n-árních funkcí

- Jsou-li $\sigma_1, \sigma_2, \sigma_3 \dots \sigma_n$ a τ nějaké typy, tak

$$\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau$$

je typ všech funkcí s prvním parametrem typu σ_1 , druhým parametrem typu σ_2 , ... a funkční hodnotou typu τ .

Terminologie

- **Arita funkce** označuje počet parametrů funkce.
- Konstanty, unární, binární, ternární funkce.
- Nulární funkce ($n=0$) jsou konstanty daného typu.

Pozorování

- Typ výrazu, který je úplná aplikace funkce na parametry, lze odvodit z typu použité funkce **bez nutnosti výpočtu výsledné hodnoty**.

Příklad

```
odd :: Integer -> Bool
27  :: Integer
odd 27 :: Bool
```

Pozorování

- Některé funkce nepotřebují znát konkrétní typy formálních parametrů, pouze jejich strukturu.
- Místo konkrétního typu se použije **typová proměnná**.
- Při aplikaci funkce na konkrétní parametry, se za typovou proměnnou dosadí typ, který odpovídá použitému parametru. (Typová proměnná se specializuje.)
- **POZOR! Typová proměnná zastupuje i složené typy.**

Příklad

```
fst :: (a,b) -> a
(not, "Coze?") :: (Bool -> Bool, [Char])
fst (not , "Coze?") :: Bool -> Bool
```

Pozorování

- Některé funkce nevyžadují konkrétní typ, ale zároveň nedovolují použití libovolného typu, proto je třeba specializaci typové proměnné omezit na vybranou podtřídu typů.

Základní typové třídy

- `Integral` – celočíselné
- `Num` – numerické
- `Ord` – uspořadatelné
- `Eq` – porovnatelné na rovnost

Příklady typů s omezením specializace typové proměnné

```
odd :: Integral a => a -> Bool
```

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

Typ není váš nepřítel

Typ není váš nepřítel



Uspořádané n-tice a seznamy v Haskellu

Zápis uspořádaných n-tic

- Přirozený, pomocí závorek a čárek.
- Příklady zápisu uspořádaných n-tic v Haskellu:

`(12,15)`

`(2,3,'a',5,6)`

`("Fiii","jo", 350, "tisíc", '!')`

`((1,1),(2,2),(3,3))`

Krajní případy

- Jednotice se nepoužívají.
- Nultice: `()`

Hodnotové konstruktory

- $(, , \dots ,)$ – hodnotový konstruktore uspořádané n-tice
- $(,)$ – hodnotový konstruktore uspořádané dvojice

$$(,) :: a \rightarrow b \rightarrow (a, b)$$
$$(,) x y = (x, y)$$

Projekce

- fst , snd – projekce na první a druhou složku

$$\text{fst} :: (a, b) \rightarrow a$$
$$\text{fst } (x, y) = x$$
$$\text{snd} :: (a, b) \rightarrow b$$
$$\text{snd } (x, y) = y$$

Zápis seznamů

- V hranatých závorkách uzavřená posloupnost prvků oddělených čárkou.
- Seznam znaků též jako řetězec (text v uvozovkách).

Příklady

```
[3,3,3,3]
```

```
[ [1], [1,2], [1,2,3] ]
```

```
[]
```

```
"ahoj" = ['a','h','o','j']
```

```
"toto je také seznam"
```

```
[ or, or, or, and ]
```

Hodnotové konstruktory

- Prázdný seznam: `[]`
`[] :: [a]`
- Operátor připojení prvku na začátek seznamu: `(:)`
`(:) :: a -> [a] -> [a]`

Příklady

- Správné použití
`(:) 3 [3,3,3] ~> [3,3,3,3]`
`1:2:3:[] ~> [1,2,3]`
`4:[4,4,4,4] ~> [4,4,4,4,4]`
`'A':"hoj" ~> "Ahoj"`
- Nesprávné použití
`[2] : [3,4,5] ~> ERROR`
`[2,3,4] : 5 ~> ERROR`
`'A' : [1,2,3] ~> ERROR`

Úkol

- Jaké je implicitní ozávkování v následujícím výrazu a proč je takové, jaké je?

1:2:3:4:5: []

Řešení

Úkol

- Jaké je implicitní ozávkování v následujícím výrazu a proč je takové, jaké je?

1:2:3:4:5: []

Řešení

- 1:(2:(3:(4:(5: []))))
- Jiné ozávkování není možné, nevyhovělo by typové signatuře hodnotového konstruktora (:).
- Dvojtečka nejvíce vlevo je **nejvnějšnější**.

Funkce pro spojení seznamů

- Seznamy **stejného typu** lze spojit pomocí funkce `(++)`
`(++) :: [a] -> [a] -> [a]`

Příklady

- Správné použití

`(++) "Ahoj " "světe!" ~> "Ahoj světe!"`

`"Ahoj" ++ " " ++ "světe!" ~> "Ahoj světe!"`

`[1,2,3] ++ [4,5,6] ~> [1,2,3,4,5,6]`

- Nesprávné použití

`2 ++ [3,4,5] ~> ERROR`

`[2,3,4] ++ 5 ~> ERROR`

`[2,3] ++ "text" ~> ERROR`

Hodnotové konstruktory ve víceřádkových definicích

- Fungují jako vzory na levých stranách definice.
- Mapují se vždy na **nejvnějšnější** výskyt.

Příklady

- Funkce `null` aplikovaná na nějaký seznam, vrací `True` pokud je seznam prázdný a `False` pokud je neprázdný.

```
null :: [a] -> Bool
null (_:_) = False
null [] = True
```

- Funkce `snd` aplikovaná na uspořádanou dvojici, vrací druhý prvek dvojice.

```
snd :: (a,b) -> b
snd (_,y) = y
```

Použití symbolu @

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

Příklady

- Při mapování seznamu `[1,2,3]` na vzor `a@(x:t)`, bude
 $a = [1,2,3], \quad x = 1, \quad t = [2,3]$.
- $f(a@(x:y)) = x:a++y$
 $f[1,2,3] \rightsquigarrow^* [1,1,2,3,2,3]$
 $f[] \rightsquigarrow^* \mathbf{ERROR}$

Úkoly

- Napište funkci, která vrátí druhý prvek seznamu, pokud je aplikována na seznam délky alespoň dva.

IB015 Neimperativní programování

Funkce na seznamech a Rekurze

Jiří Barnat
Libor Škarvada

Vím, že programovat znamená především přemýšlet ...

- o správném **postupu**, jak dojít k cíli,
- o **dekompozici** řešeného problému.

Vím, že k obému mi pomáha především ...

- najít vhodný způsob **uložení dat**,
- správné **otypování použitých artefaktů**.

Kudy dál?

- Musím se seznámit se základními stavebními kameny programovacího jazyka a naučit se je správně, efektivně a elegantně používat (Triforce).

Práce se seznamy v Haskellu (poprvé)

První prvek seznamu

- `head :: [a] -> a`

Seznam bez prvního prvku

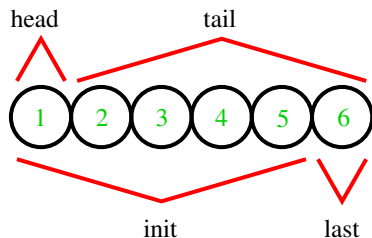
- `tail :: [a] -> [a]`

Poslední prvek seznamu

- `last :: [a] -> a`

Seznam bez posledního prvku

- `init :: [a] -> [a]`



Z minula známe

- (:), (++)

Úkol 1

- Zdvojte první prvek seznamu.
- Zduplikujte seznam a na konec přidejte první prvek.

Úkol 2

- Prohodte první dva prvky seznamu.
- Prohodte první a poslední prvek seznamu.

Zjištění počtu prvků v seznamu

- Vrací počet prvků v seznamu na nejvyšší úrovni, tj. nepočítá prvky v zanořených seznamech!
- `length :: [a] -> Int`

Vyzkoušejte

- `length [5,2,8] ~> 3`
`length [] ~> 0`
`length [[]] ~> 1`
`length [[3],[8,4,5,5,5,4,5]] ~> 2`

Prvních n prvků seznamu

- Vrátí prvních n prvků seznamu jako seznam
- `take :: Int -> [a] -> [a]`

Seznam bez prvních n prvků seznamu

- Vrátí původní seznam bez prvních n prvků
- `drop :: Int -> [a] -> [a]`

Získání n-tého prvku

- Vrátí prvek seznamu na dané pozici, první prvek je na pozici 0.
- `(!!) :: [a] -> Int -> a`

Seznam prvků v opačném pořadí

- `reverse :: [a] -> [a]`

Aritmetické a logické funkce na seznamech

- `minimum :: Ord a => [a] -> a`
- `maximum :: Ord a [a] -> a`
- `sum :: Num a => [a] -> a`
- `product :: Num a => [a] -> a`
- `or :: [Bool] -> Bool`
- `and :: [Bool] -> Bool`

Srovnejte s binárními funkcemi

- `min :: Ord a => a -> a -> a`
- `max :: Ord a => a -> a -> a`
- `(&&) :: Bool -> Bool -> Bool`
- `(||) :: Bool -> Bool -> Bool`

Úkol 3

- Rozhodněte, zda je seznam palindrom.

Úkol 4

- Na neprázdných seznamech typu `Num a => [a]` definujte funkci, která rozhodne, zda je součet prvků seznamu větší, než součin.

Aplikace funkce na prvky v seznamu

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

Vyzkoušejte

- `map not [True,False,False]` \rightsquigarrow^* `[False,True,True]`
- `let f x = x + 1 in map f [4,5,6]` \rightsquigarrow^* `[5,6,7]`
- `map even [3,4,5]` \rightsquigarrow^* `[False,True,False]`

Výběr prvků seznamu podle dané podmínky

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

Vyzkoušejte

- `filter odd [1,2,3]` \rightsquigarrow^* `[1,3]`
`filter (not.odd) [1,2,3]` \rightsquigarrow^* `[2]`

Všimněte si

- Funkce `map` i `filter` berou jako své argumenty jiné funkce. Takovým funkcím říkáme funkce vyšších řádů.

Úkol 5

- Rozhodněte, zda je seznam typu `Integral a => [a]` tvořen pouze sudými čísly.

Úkol 6

- Definujte funkci, která vrátí délku zadaného seznamu, tak, abyste nepoužili funkci `length`.

Úkol 7

- Pro seznam dvojic, zaměňte první složku každé dvojice za druhou a naopak. Funkci definujte s použitím klíčového slova `where`.
- Poznámka: `where` se podobně jako `let ... in` používá pro lokální definice s tím, že klauzule `where` se vyskytuje až za hlavním výrazem.

Ponechání/odstranění podmínkou definovaného prefixu

- První prvek od začátku seznamu nevyhovující zadané podmínce definuje místo, kde končí vracený, nebo zahozený prefix.
- `takeWhile :: (a -> Bool) -> [a] -> [a]`
`dropWhile :: (a -> Bool) -> [a] -> [a]`

Vyzkoušejte

- `takeWhile odd [1,3,5,6,7,8,9]` \rightsquigarrow^* `[1,3,5]`
`dropWhile odd [1,3,5,6,7,8,9]` \rightsquigarrow^* `[6,7,8,9]`

Spojení seznamu seznamů do jednoho seznamu

- `concat :: [[a]] -> [a]`

Spojení dvou seznamů do seznamu dvojic

- `zip :: [a] -> [b] -> [(a,b)]`
- Délka výsledného seznamu je definována kratším ze dvou zadaných seznamů.

Vyzkoušejte

- `concat [[1,2], [2,3], [3,4]]` \rightsquigarrow^* `[1,2,2,3,3,4]`
- `zip [1,2,3,4] "abc"` \rightsquigarrow^* `[(1, 'a'), (2, 'b'), (3, 'c')]`

Spojení seznamu pomocí funkce

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

Vyzkoušejte

- `zipWith (+) [1,2,3] [1,2,3,4,5] ~>* [2,4,6]`
- `zipWith take [1,2,3] ["aaa","bbb","ccc"] ~>* ["a","bb","ccc"]`

Úkol

- Co počítá funkce `fn` a jakého je typu?
- `fn s = and (map f (zip s (head s:s)))`
 `where f (a,b) = a >= b`

Řešení

Úkol

- Co počítá funkce `fn` a jakého je typu?
- ```
fn s = and (map f (zip s (head s:s)))
 where f (a,b) = a >= b
```

## Řešení

- `fn :: Ord a => [a] -> Bool`
- Funkce `fn` pro zadaný seznam rozhodne, zda je seznam neklesající.

## Zadání

- S využitím stávajících znalostí o programovacím jazyce Haskell naprogramujte funkci `len`, která spočítá délku seznamu, a to tak, aniž byste použili funkci `length`, `sum` či jinou podobnou funkci.

## Náznak řešení

- `len [] = 0`  
`len (_:[]) = 1`  
`len (_:_:[]) = 2`  
`len (_:_:_:[]) = 3`  
`len (_:_:_:_:[]) = 4`  
...

# Rekurze

## Co je to rekurze

- Definice funkce, nebo datové struktury, s využitím sebe sama.

## Význam v programování

- Umožňuje konečně dlouhý zápis definice funkce, která je definována pro nekonečně mnoho strukturálně odlišných parametrů.

## Příklad

- Funkci `length`, která při aplikaci na seznam vrátí jeho délku, je nutné definovat rekurzivně.

```
length :: [a] -> Int
length [] = 0
length (_:s) = 1 + length s
```

# Příklad výpočtu rekurzivní definice

```
length :: [a] -> Int
length [] = 0
length (_:s) = 1 + length s
```

```
length [6,7,8,9] ~> 1 + length [7,8,9]
 ~> 1 + (1 + length [8,9])
 ~> 1 + (1 + (1 + length [9]))
 ~> 1 + (1 + (1 + (1 + length [])))

 ~> 1 + (1 + (1 + (1 + 0)))
 ~> 1 + (1 + (1 + 1))
 ~> 1 + (1 + 2)
 ~> 1 + 3
 ~> 4
```



# Rekurze na číselných funkcích

```
factorial :: Integer -> Integer
```

```
factorial 0 = 1
```

```
factorial x = x * factorial (x-1)
```

```
factorial 4 ~> 4 * factorial (4-1)
~> 4 * factorial (3)
~> 4 * (3 * factorial (3-1))
~> 4 * (3 * factorial (2))
~> 4 * (3 * (2 * factorial (2-1)))
~> 4 * (3 * (2 * factorial (1)))
~> 4 * (3 * (2 * (1 * factorial (0))))
~> 4 * (3 * (2 * (1 * 1)))
~> 4 * (3 * (2 * 1))
~> 4 * (3 * 2)
~> 4 * 6
~> 24
```

## Práce se seznamy v Haskellu (podruhé)

# head, tail, init, last

První prvek seznamu

```
head :: [a] -> a
```

```
head (x:_) = x
```

Seznam bez prvního prvku

```
tail :: [a] -> [a]
```

```
tail (_:s) = s
```

Poslední prvek seznamu

```
last :: [a] -> a
```

```
last (x:[]) = x
```

```
last (_:s) = last s
```

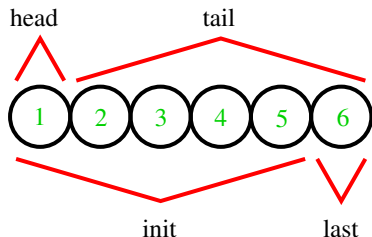
Seznam bez posledního prvku

```
init :: [a] -> [a]
```

```
init (_:[]) = []
```

```
init (x:_:[]) = [x]
```

```
init (x:s) = x:init s
```



## Test na prázdný seznam

```
null :: [a] -> Bool
null (_:_) = False
null [] = True
```

## Délka seznamu

```
length :: [a] -> Int
length [] = 0
length (_:s) = 1 + length s
```

## N-tý prvek seznamu

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

Prvních n prvků seznamu

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

```
take _ [] = []
```

```
take n (x:s) = if (n>0) then x : take (n-1) s
 else []
```

Seznam bez prvních n prvků;

```
drop :: Int -> [a] -> [a]
```

```
drop _ [] = []
```

```
drop n (x:s) = if (n>0) then drop (n-1) s
 else (x:s)
```

## Poznámka

- Při infixovém použití binární funkce klesá její priorita!

**$x : \text{take } (n-1) s = x : (\text{take } (n-1) s)$**

## Spojení seznamů v seznamu

```
concat :: [[a]] -> [a]
concat [] = []
concat (x:s) = x ++ concat s
```

## Vynechání prvků nesplňujících danou podmínku

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:s) = if (f x) then x : filter f s
 else filter f s
```

## Vytvoření seznamu n-násobným kopírováním daného prvku

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate k x = x : replicate (k-1) x
```

Vynechání prvků seznamu od prvního, který nesplňuje podmínku

```
takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:s) = if (p x) then x : takeWhile p s
 else []
```

Vynechání prvků seznamu po první, který nesplňuje podmínku

```
dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p (x:s) = if (p x) then dropWhile p s
 else x:s
```

Aplikace funkce na všechny prvky seznamu

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

Spojení dvou seznamů do seznamu uspořádaných dvojic

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

```
zip [] _ = []
```

```
zip _ [] = []
```

```
zip (x:s) (y:t) = (x,y) : zip s t
```

Rozdělení seznamu dvojic na dvojici seznamů

```
unzip :: [(a,b)] -> ([a],[b])
```

```
unzip [] = ([],[])
```

```
unzip ((x,y):s) = (x : fst t, y : snd t) where t = unzip s
```



Výpočet aplikace binární funkce na seznamy argumentů

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

```
zipWith _ _ [] = []
```

```
zipWith _ [] _ = []
```

```
zipWith f (x:s) (y:t) = f x y : zipWith f s t
```

Pozorování

```
zip = zipWith (,)
```

## **Demonstrace řešení vybraných příkladů (mentálně náročné)**

## Kolikátá mocnina dvojky

- Napište funkci, která zjistí, kolikrát se vyskytuje dvojka v prvočíselném rozkladu zadaného celého čísla.

## Řešení

- Myšlenka: kolikrát mohu beze zbytku podělit dvojkou.
- `kolik2 :: (Integral a, Integral b) => a -> b`  
`kolik2 x = y where (y,_) = f_aux (0,x)`

```
f_aux (cnt, rem) =
 if (rem<2) || (rem `mod` 2 /= 0)
 then (cnt,rem)
 else f_aux (cnt+1, rem `div` 2)
```

## Zatřizení sezařených seznamů

- Napište funkci, která pro dva seřazené seznamy vrátí jeden seřazený seznam, který obsahuje všechny prvky z původních dvou seznamů, tj. zatřídí dva seznamy „do sebe“.

## Řešení

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

Řazení seznamů

- Napište funkci, která pro zadaný seznam vrátí seznam stejných prvků, ovšem seřazený podle operace $<$.

Řešení

- Seznam nejprve rozdělím na dvě části, ty setřídím rekurzivně, a pak zatřídím do sebe.
- $\text{evenodd } (x:y:z) = (x:s,y:t) \text{ where } (s,t) = \text{evenodd } z$
 $\text{evenodd } (x:[]) = ([],[x])$
 $\text{evenodd } _ = ([],[])$
- $\text{msort } [] = []$
 $\text{msort } [x] = [x]$
 $\text{msort } x = \text{merge } (\text{msort } xs) (\text{msort } ys)$
 where $(xs,ys) = \text{evenodd } x$

Pozorování

- Rekurzi je možné najít i na místech, kde byste ji nehledali.
- Vše co se opakuje a směřuje ke konci, lze popsat rekurzí.

Slovní úloha

- Na Jiříkovu narozeninovou párty přišlo 7 kamarádů a přineslo mimo jiné jeden narozeninový dort. Jiřík nebyl lakomý, rozdělil dort rovnoměrně mezi všechny přítomné. Jakou k tomu použil rekurzivní akci? (Jaký je typ příslušné rekurzivní funkce?)



Pozorování

- Rekurzi je možné najít i na místech, kde byste ji nehledali.
- Vše co se opakuje a směřuje ke konci, lze popsat rekurzí.

Slovní úloha

- Na Jiříkovu narozeninovou párty přišlo 7 kamarádů a přineslo mimo jiné jeden narozeninový dort. Jiřík nebyl lakomý, rozdělil dort rovnoměrně mezi všechny přítomné. Jakou k tomu použil rekurzivní akci? (Jaký je typ příslušné rekurzivní funkce?)

Možné řešení

```
type KousekDortu = Float
dort :: [KousekDortu] -> [KousekDortu]
dort s = if (length s >= 8)
         then s
         else dort (map (/2) s ++ map (/2) s)
```



Úkol 1

- Mějme konečný seznam celých čísel. Napište funkci, která vynásobí všechna čísla v seznamu zadaným parametrem, a pak ze seznamu ponechá pouze čísla dělitelná číslem 7.
- Funkci upravte tak, aby výše zmíněné číslo 7 nebylo fixně v kódu funkce, ale byl to její další parametr.

Úkol 2

- Napište funkci, která o seznamu řetězců rozhodne, zda tento seznam obsahuje alespoň 4 řetězce délky minimálně 2.
- Funkci upravte tak, aby výše zmíněné čísla 4 a 2 nebyla fixně v kódu funkce, ale byly to její další parametry.

Challenge 1

- Napište program, který pomocí principu rekurze a s využitím odpřednášených operací na seznamech vypočítá seznam obsahující čísla od 1 do 1024. Snažte se o to, aby hloubka rekurze byla co nejmenší (logaritmická).

Challenge 2

- Jsou-li vám známy cykly s pevným počtem opakování z nějakého imperativního programovacího jazyka, popřemýšlejte o obecném postupu, jak nahradit tyto cykly voláním rekurzivní funkce.

IB015 Neimperativní programování

Funkce vyšších řádů a λ -funkce

Jiří Barnat
Libor Škarvada

Datové struktury

- Uspořádané n-tice, seznamy.

Hodnoty a typy

- Elementární, složené typy, typy funkcí.
- Polymorfní a kvalifikované typy.
- Hodnotové konstruktory.

Rekurze

- Rekurzivní funkce na seznamech.

Funkce vyšších řádů

Definice

- Funkce, je považována za funkci vyššího řádu, pokud alespoň jeden z jejich argumentů, které funkce má, nebo výsledek, který funkce vrací, je opět funkce.
- Funkce vyššího řádu se též označují jako funkcionály.

Příklady

$(.) \quad :: \quad (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$

`map` $:: \quad (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

Pozorování

- Každou funkci, která má alespoň 2 argumenty, lze chápat jako funkci vyššího řádu.

Příklad

- Funkci

$(*) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

lze číst jako

$(*) :: \text{Num } a \Rightarrow a \rightarrow (a \rightarrow a)$

- Funkci, která bere dva číselné argumenty typu a a vrací hodnotu typu a , lze také chápat jako funkci, která bere hodnotu číselného typu a a vrací hodnotu typu $a \rightarrow a$.

Pozorování

- Chápeme-li n -ární ($n \geq 2$) funkce jako funkce vyššího řádu, lze tyto funkce tzv. **částečně aplikovat**, tj. vyhodnotit je i pro neúplný výčet argumentů.

Příklad

- Uvažme funkci násobení
 - $(*) :: \text{Num } a \Rightarrow a \rightarrow (a \rightarrow a)$
 - $(*) \ x \ y = x*y$
- Výsledkem aplikace $(*)$ na hodnotu 3 je funkce.

$(*) \ 3 :: \text{Num } a \Rightarrow a \rightarrow a$

- Tuto funkci je možné označit, a posléze použít.

$f = (*) \ 3$

$f :: \text{Num } a \Rightarrow a \rightarrow a$

$f \ 4 \rightsquigarrow ((*) \ 3) \ 4 \rightsquigarrow 12$

Připomenutí

- Typový konstruktor \rightarrow je binární.
- Typový konstruktor \rightarrow se používá pouze infixově.

Implicitní závorkování

- Typový konstruktor \rightarrow implicitně sdružuje zprava

$$f :: a_1 \rightarrow (a_2 \rightarrow (a_3 \rightarrow \dots \rightarrow (a_{n-1} \rightarrow (a_n \rightarrow a)) \dots))$$

- Aplikace funkce na argumenty implicitně sdružuje zleva

$$(\dots (((f x_1) x_2) x_3) \dots x_{n-1}) x_n$$

Pozorování

- Libovolnou n -ární funkci lze také chápat jako k -ární, kde k nabývá hodnot od 1 do n .

S každou aplikací ubude jeden výskyt \rightarrow v typu výrazu

$(+)$ $::$ Num $a \Rightarrow a \rightarrow a \rightarrow a$

$(+) 2$ $::$ Num $a \Rightarrow$ $a \rightarrow a$

$((+) 2) 3$ $::$ Num $a \Rightarrow$ a

S každou aplikací ubude jeden výskyt \rightarrow v typu výrazu

$(+)$ $::$ $\text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

$(+)$ 2 $::$ $\text{Num } a \Rightarrow$ $a \rightarrow a$

$((+)$ 2) 3 $::$ $\text{Num } a \Rightarrow$ a

Specializací typové proměnné však mohou \rightarrow přibýt.

- Vezměme například funkci identity

$\text{id} :: a \rightarrow a$

$\text{id } x = x$

- Při aplikaci id na $(+)$ ubude \rightarrow z typu id , tj.

$\text{id } (+) :: a$

- Avšak po specializaci typové proměnné a na typ funkce $(+)$

$\text{id } (+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

Částečná aplikace

- Má-li funkce více formálních parametrů částečná aplikace probíhá vždy od parametru nejvíce vlevo.

Problém

- Funkci nelze přímo částečně aplikovat na jiný než první parametr.

Funkce `flip`

- Při aplikaci na binární funkci tuto funkci modifikuje tak, aby své dva argumenty přijímala v obráceném pořadí.

```
flip :: ( a -> b -> c ) -> b -> a -> c
flip f x y = f y x
```

- Funkci `flip` je třeba chápat jako **modifikátor funkce**, ne jako prohazovačku parametrů!

Příklady

```
(-) 3 4 ~> -1
```

```
flip (-) 3 4 ~> 1
```

```
(-) flip 3 4 ~> ERROR
```

Příklad

- Pomocí částečné aplikace funkce `(-)` definujte novou funkci `minus4`, která při aplikaci na číselnou hodnotu vrátí hodnotu o 4 menší.

Řešení

- Definice s využitím částečné aplikace, bez formálních parametrů.

```
minus4 :: Num a => a -> a
minus4 = flip (-) 4
```

- Standardní definice téhož s využitím formálních parametrů.

```
minus4 :: Num a => a -> a
minus4 x = (-) x 4
```

Pozorování

- Funkce vyššího řádu a částečná aplikace souvisí s násobným použitím funkcionálního typového konstrukturu \rightarrow .
- Chceme-li zabránit částečné aplikaci, musíme definovat funkci tak, aby v jejím typu byl pouze jeden výskyt \rightarrow .
- Pokud chceme funkci předat více argumentů najednou, předáme je jako uspořádanou n-tici.

Příklad

- Všimněte si rozdílu v typech a definicích následujících funkcí.

```
krat :: Num a => a -> a -> a
```

```
krat x y = x * y
```

```
krat1 :: Num a => (a,a) -> a
```

```
krat1 (x,y) = x * y
```

Funkce `curry` a `uncurry`

- Předdefinované funkce pro změnu řádu binárních funkcí.

`curry`

- Modifikuje funkci tak, že tato funkce místo uspořádané dvojice hodnot přijímá dva samostatné parametry.

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

`uncurry`

- Modifikuje funkci tak, že tato funkce místo dvou samostatných parametrů přijímá uspořádanou dvojici hodnot.

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

Příklad

- Mějme definovány následující funkce

```
krat :: Num a => a -> a -> a
krat x y = x * y
```

```
krat1 :: Num a => (a,a) -> a
krat1 (x,y) = x * y
```

- Uvedte alternativní definici funkce `krat` pomocí funkce `krat1` a obráceně.

Příklad

- Mějme definovány následující funkce

```
krat :: Num a => a -> a -> a
krat x y = x * y
```

```
krat1 :: Num a => (a,a) -> a
krat1 (x,y) = x * y
```

- Uvedte alternativní definici funkce `krat` pomocí funkce `krat1` a obráceně.

Řešení

```
krat = curry krat1
krat1 = uncurry krat
```

Co je to

- Pro každý **binární operátor** je možné definovat funkci, jež odpovídá částečné aplikaci funkce na první formální parametr a funkci, jež odpovídá částečné aplikaci funkce na druhý formální parametr. Těmto funkcím se říká **operátorové sekce**.

Operátorové sekce

- Předpokládejme binární operátor \oplus a hodnoty p a q
 $\oplus :: a \rightarrow b \rightarrow c$
 $p :: a$
 $q :: b$
- Částečnou aplikaci na první argument zapíšeme jako $(p\oplus)$
 $(p\oplus) = (\oplus) p$
- Částečnou aplikaci na druhý argument zapíšeme jako $(\oplus q)$
 $(\oplus q) = \text{flip } (\oplus) q$

Příklad 1

- Jednoduché použití operátorových sekcí.

`(*2) 34` \rightsquigarrow 68

`(++".") ['V', 'ě', 't', 'a']` \rightsquigarrow "Věta."

Příklad 2

- Odvodte typ následujících funkcí, popište jejich význam:

`(1.0/)`

`('mod' 3)`

`(!!0)`

`(>0)`

Příklad 1

- Jednoduché použití operátorových sekcí.

`(*2) 34 ~> 68`

`(++".") ['V', 'ě', 't', 'a'] ~> "Věta."`

Příklad 2

- Odvoďte typ následujících funkcí, popište jejich význam:

`(1./)` :: `Fractional a => a -> a`

`('mod' 3)` :: `Integral a => a -> a`

`(!!0)` :: `[a] -> a`

`(>0)` :: `(Num a, Ord a) => a -> Bool`

Skládání funkcí

- Základní operátor funkcionálního programování

$$(\cdot) \quad :: \quad (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$
$$(\cdot) \quad f \ g \ x = f \ (g \ x)$$

Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

$$(\cdot) \quad :: \quad (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

- Pro operátor (\cdot) je možné definovat operátorovou sekci.
- Použití operátorové sekce pro operátor (\cdot) na jiné než jednoduché funkce je však velmi matoucí a v praxi se nepoužívá.

$$(\cdot(\cdot)) \quad :: \quad (((a \rightarrow b) \rightarrow a \rightarrow c) \rightarrow d) \rightarrow (b \rightarrow c) \rightarrow d$$

Skládání funkcí

- Základní operátor funkcionálního programování

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

$(.) f g x = f (g x)$

Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

- Pro operátor $(.)$ je možné definovat operátorovou sekci.
- Použití operátorové sekce pro operátor $(.)$ na jiné než jednoduché funkce je však velmi matoucí a v praxi se nepoužívá.



$(.(.)) :: (((a \rightarrow b) \rightarrow a \rightarrow c) \rightarrow d) \rightarrow (b \rightarrow c) \rightarrow d$

Pozorování

- Funkce `flip`, `curry`, `uncurry`, `(.)` a operátorové sekce používáme k vytvoření tzv. **bezparametrové** definice funkce.

Připomenutí bezparametrové definice

- Funkci `f` definovanou s použitím formálního parametru

```
f x = (not.odd) x
```

- Je možné definovat i bez použití formálního parametru

```
f = (not.odd)
```

Příklad

```
f x = (3*x)^7
```

```
f x = flip (^) 7 (3*x)
```

```
f x = flip (^) 7 ((* 3) x)
```

```
f x = (.) (flip (^) 7) ((* 3) x)
```

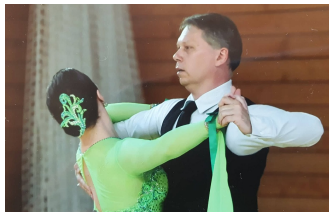
```
f x = (.) (flip (^) 7) (3*) x
```

```
f = (.) (flip (^) 7) (3*)
```

Reklamní blok

Taneční pro dospělé

- pondělí, 20:00
- Horní 729/32, BRNO
- <https://infinitydance.cz/brno>



Speciální akce

- **Stačí dlouhé kalhoty a triko, není třeba kvádro!!!**

Konec reklamního bloku

Nepojmenované funkce

(λ -funkce)

Motivace

- Při standardní definici funkce musíme tuto funkci pojmenovat.
- Mnohé funkce, často jednoduché, použijeme jednorázově.
- Funkce s jednorázovým použitím je zbytečné pojmenovávat.

Příklad

- Globální definice a použití jednoduché funkce

```
f x = x*x + 1
```

```
map f [1,2,3,4,5] ~> [2,5,10,17,26]
```

- Lokální definice a použití jednoduché funkce

```
let f x = x*x + 1 in map f [1,2,3,4,5] ~> [2,5,10,17,26]
```

Co to je

- Definice funkce v místě jejího použití bez uvedení jejího jména.
- Příklad:

`map (\x -> x*x+1) [1,2,3,4,5] ~> [2,5,10,17,26]`

Původ a všeobecné označení

- Myšlenka a teoretický základ pochází z Lambda kalkulu, proto se též nepojmenováním funkcím říká **lambda funkce**.
- Principu vytváření lambda funkcí, se říká lambda abstrakce.

Pozorování

- Koncept lambda funkcí se vyskytuje v mnoha imperativních programovacích jazycích, např. C++, C#, SCALA, PHP, ...

Lambda abstrakce

- Uvažme výraz M , který představuje tělo funkce.

$$M \equiv x * x + 1$$

- Z těla funkce vytvoříme funkci použitím lambda abstrakce.

$$\lambda x.M \equiv \lambda x.(x * x + 1)$$

- Při aplikaci lambda funkce $\lambda x.M$ na výraz N , se všechny volné výskyty formálního parametru x v M nahradí výrazem N . Výskyt proměnné x je označován jako volný, pokud není v rozsahu žádné lambda abstrakce.

$$\lambda x.M N \equiv \lambda x.(x * x + 1) N = N * N + 1$$

Příklady

- $(\lambda x.x * x + 1) 3 = 3 * 3 + 1 = 10$.
- $(\lambda x.x + (\lambda x.x + x) x) 5 = 5 + (\lambda x.x + x) 5 = 5 + (5 + 5) = 15$.
- $(\lambda x.34) 3 = 34$

Definice a použití nepojmenované funkce

- λ se špatně píše na klávesnici.
- V programovacím jazyce *Haskell* je lambda abstrakce zapsána pomocí zpětného lomítka a šipky:

```
\ formální parametry -> tělo funkce
```

Příklady

```
(\ x -> x*x + x*x) 3 ~> ... ~> 18
```

```
(\ x -> x False) not ~> not False ~> True
```

Nepojmenovanou funkci je možné pojmenovat

```
f = (\ x -> x*x + x*x)
```

```
f 3 ~> ... ~> 18
```

```
f 3 ~> (\ x -> x*x + x*x) 3 ~> ... ~> 18
```

Pozorování

- Vnořená lambda abstrakce vytváří funkce vyšších řádů.
- $(\lambda x.(\lambda y.(x - y))) 3 4 = (\lambda y.(3 - y)) 4 = 3 - 4 = -1$

Zápis v Haskellu

- Odvozený přímo z vícenásobné aplikace lambda abstrakce
 $\backslash x \rightarrow (\backslash y \rightarrow x - y)$
 $(\backslash x \rightarrow (\backslash y \rightarrow x - y)) 3 4 \rightsquigarrow \dots \rightsquigarrow -1$
- Zkrácený z důvodu čitelnosti kódu a pohodlí programátorů
 $\backslash x y \rightarrow x - y$
 $(\backslash x y \rightarrow x - y) 3 4 \rightsquigarrow \dots \rightsquigarrow -1$

Nepojmenované funkce a jejich typy

- Obecně platí stejná pravidla jako pro pojmenované funkce.
- Název funkce (ani jeho neexistence) nemá vliv na typ funkce.

Efekt lambda abstrakce na typ

- Každý formální parametr v lambda abstrakci přidá do typu výrazu jeden výskyt typového konstrukturu \rightarrow a novou typovou proměnnou, která se navíc specializuje podle kontextu použití konkrétního formálního parametru.

Pozorování

- Typ funkce si Hugs/GHCi umí odvodit z její definice.

Příklad 1

```
'a'                :: Char
(\ x -> 'a')       :: a -> Char
(\ x y -> 'a')     :: a -> b -> Char
(\ x -> (\ y -> 'a')) :: a -> b -> Char
```

Příklad 2

```
\ x y -> x !! y    :: [a] -> Int -> a
\ x y -> x || y    :: Bool -> Bool -> Bool
```

Příklad 3

```
(\ x y -> x . y)   :: (a -> b) -> (c -> a) -> c -> b
(\ x y -> x y)     :: (a -> b) -> a -> b
```

Pozorování

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

Aplikační operátor (\$)

- $(\$)$:: $(a \rightarrow b) \rightarrow a \rightarrow b$
 $(\$) f x = f x$

Infixové použití

- $f(g x) \equiv (\$) f (g x) \equiv f \$ (g x) \equiv f \$ g x$
- $f(g(h x)) \equiv f \$ g \$ h x$
- $\$ f x \rightsquigarrow^* \text{ERROR}$

Ekvivalentní zápisy pomocí operátoru \$

- `not (not (not (not (not True)))))`
`not $ not $ not $ not $ not True`
- `(+1) ((+2) ((+3) ((+4) 0)))`
`(+1) $ (+2) $ (+3) $ (+4) 0`
- `(even ((+1) 0)) || (odd ((+2) 0))`
`(even $ (+1) 0) || (odd $ (+2) 0)`

Příklady

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

Mentální cvičení

- Definujte operátorové sekce pomocí lambda abstrakce.
- Identifikujte funkce vyššího řádu ve svém každodenním životě.

IB015 Neimperativní programování

Redukční strategie, Nekonečné datové struktury
Zápis a generování seznamů

Jiří Barnat
Libor Škarvada

Redukční strategie

Redukční krok

- Úprava výrazu, v němž se některý jeho podvýraz nahradí zjednodušeným podvýrazem.
- Upravovaný podvýraz (**redex**) má tvar aplikace funkce na argumenty, upravený podvýraz má tvar pravé strany definice této funkce do níž jsou za formální parametry dosazené skutečné argumenty.

Redukční strategie

- Předpis, který určuje jaký podvýraz se bude upravovat v následujícím redukčním kroku.

Striktní redukční strategie

- Při úpravě aplikace $F X$ nejdříve úplně upravíme argument X . Teprve nelze-li už upravovat argument X , upravujeme výraz F . Až nakonec upravíme (podle definice funkce) celý výraz $F X$.
- Při úpravě výrazů tedy postupujeme **zevnitř**.

Normální redukční strategie

- Upravovaným podvýrazem je celý výraz; nelze-li takto upravit aplikaci $F X$, upravíme nejdříve výraz F , pokud to nestačí k tomu, abychom mohli upravit $F X$, upravujeme částečně výraz X , ale pouze do té míry, abychom mohli upravit výraz $F X$.
- Při úpravě výrazů tedy postupujeme **zvnějšku**.

Líná redukční strategie

- Normální redukční strategie, při níž opakované výskyty podvýrazu nevyhodnocujeme více než jednou.
- Využívá referenční transparentnost.
- Nelze aplikovat na výrazy s vedlejším efektem.

Haskell

- Používá **normální redukční strategii**.
- Nicméně mluví se o **líném vyhodnocování**, zjednodušeně řečeno, vyhodnotí se pouze to, co je potřeba k dalšímu výpočtu.

Definice funkce

- $\text{cube } x = x * x * x$

Striktní redukční strategie

- $\text{cube } \underline{(3+5)} \rightsquigarrow \underline{\text{cube } 8} \rightsquigarrow \underline{8 * 8 * 8} \rightsquigarrow \underline{64 * 8} \rightsquigarrow 512$

Normální redukční strategie

- $\underline{\text{cube } (3+5)} \rightsquigarrow \underline{(3+5) * (3+5) * (3+5)} \rightsquigarrow 8 * \underline{(3+5) * (3+5)}$
 $\rightsquigarrow \underline{8 * 8} * (3+5) \rightsquigarrow 64 * \underline{(3+5)} \rightsquigarrow \underline{64 * 8} \rightsquigarrow 512$

Líná redukční strategie

- $\underline{\text{cube } (3+5)} \rightsquigarrow \underline{(3+5) * (3+5) * (3+5)} \rightsquigarrow \underline{8 * 8 * 8} \rightsquigarrow$
 $\underline{64 * 8} \rightsquigarrow 512$

Líná redukční strategie a Haskell?

- V Haskellu lze docílit, pouze pokud je stejný podvýraz ve výrazu zaveden pomocí lokální definice, nebo je výsledkem substituce za formální parametr.
- Haskell používá tzv. Graph Reduction.
- Ne vše lze líně vyhodnotit (výrazy s vedlejším efektem).

Příklad

- $\text{cube } (3+5) * (3+5) \rightsquigarrow (3+5) * (3+5) * (3+5) * (3+5) \rightsquigarrow$
 $8 * 8 * 8 * (3+5) \rightsquigarrow 64 * 8 * (3+5) \rightsquigarrow 512 * (3+5) \rightsquigarrow$
 $512 * 8 \rightsquigarrow 4096$

Více info viz

- https://en.wikipedia.org/wiki/Graph_reduction
- https://wiki.haskell.org/GHC/FAQ#Does_GHC_do_common_subexpression_elimination.3F
- https://wiki.haskell.org/GHC_optimisations#Common_subexpression_elimination

Referenční transparence

- Výsledek vyhodnocení výrazu nezávisí na kontextu, ve kterém se daný výraz vyhodnocuje.
- Může mít vedlejší efekt, ten ale nesmí ovlivnit výsledek.
- Haskell je **referenčně transparentní**,

Vedlejší efekt vyhodnocení výrazu či funkce

- Změna stavu světa, která je pozorovatelná vně volané funkce nad rámec návratové hodnoty.
- Například modifikace globální proměnné, modifikace hodnot v kontextu rodičovské funkce, modifikace externí paměti, atd.

Pozorování

- Použitá strategie může ovlivnit chování programu.

Příklad 1

- Uvažme funkci `const`

```
const :: a -> b -> a
```

```
const x y = x
```

- Při striktním vyhodnocování dojde k dělení nulou

```
const 2 (1/0)  $\rightsquigarrow$  ERROR
```

- Při normálním vyhodnocování k němu nedojde

```
const 2 (1/0)  $\rightsquigarrow$  2
```

Příklad 2

- Uvažme funkci `undf`

```
undf x :: Int -> Int
```

```
undf x = undf x
```

- Striktní vyhodnocování následujícího výrazu vede k zacyklení

```
head (tail [undf 1, 4]) =
```

```
head (tail (undf 1 : 4 : [])) ~>
```

```
head (tail (undf 1 : 4 : [])) ~>
```

```
...
```

- Při normálním vyhodnocování k zacyklení nedojde:

```
head (tail [undf 1, 4]) =
```

```
head (tail (undf 1 : 4 : [])) =
```

```
head (tail (undf 1 : 4 : [])) ~>
```

```
head (4 : []) ~> 4
```

Churchova-Rosserova věta

- Výsledná hodnota ukončeného výpočtu výrazu nezáleží na redukční strategii: pokud výpočet skončí, je jeho výsledek vždy stejný.

Interpretace věty

- Churchova-Rosserova věta **nevylučuje různé chování** výpočtu při různých strategiích. Při některých strategiích může výpočet skončit, při jiných cyklovat. Nebo je výpočet podle jedné strategie delší než podle jiné. Nikdy však **nemůže skončit dvěma různými výsledky**.

O perpetualitě

- Jestliže pro nějaký výraz M existuje redukční strategie, s jejímž použitím se úprava výrazu M zacyklí, pak se tento výpočet zacyklí i s použitím striktní redukční strategie.

Interpretace věty

- Věta o perpetualitě říká, že z hlediska možnosti zacyklení výpočtu je striktní redukční strategie nejméně bezpečná. Když se při jejím použití výpočet nezacyklí, pak se nezacyklí ani při žádné jiné strategii.

O normalizaci

- Jestliže pro nějaký výraz M existuje redukční strategie, s jejímž použitím se úprava výrazu M nezacyklí, pak se tento výpočet nezacyklí ani s použitím normální redukční strategie.

Interpretace věty

- Věta o normalizaci říká, že z hlediska možnosti zacyklení výpočtu je normální redukční strategie nejbezpečnější. To neznamená, že by se s jejím použitím výpočet zacyklit nemohl; z věty však plyne, že když se to stane a výpočet se i při normální redukční strategii zacyklí, pak se zacyklí i při každé jiné strategii.

Jiný pohled

- Při použití normální redukční strategie je výraz vyhodnocen až v okamžiku, kdy je potřeba pro další výpočet.
- Přístup, který jde nad rámec redukční strategie.

Příklady

- Líné čtení řetězce ze vstupu:
`getContents :: IO String`
- Líné vyhodnocování Boolovských operátorů v imperativních programovacích jazycích.
`(True OR (1/0)) = True`
`(open(...) OR die) - "umře" pokud open selže.`

Práce s nekonečnými seznamy

Nekonečné datové struktury

- Vyhodnocení výrazu až v okamžiku, kdy je potřeba pro další výpočet, umožňuje manipulaci s nekonečnými datovými strukturami.
- Příkladem nekonečné datové struktury je **nekonečný seznam**.

Nekonečné opakování jednoho prvku

- `repeat :: a -> [a]`
`repeat x = x : repeat x`

Jak to funguje?

- `take 8 (repeat 1) \rightsquigarrow^* [1,1,1,1,1,1,1,1]`
- `head (repeat 1) \rightsquigarrow head (1 : repeat 1) \rightsquigarrow 1`

Nekonečné opakování seznamu

- `cycle :: [a] -> [a]`
`cycle x = x ++ cycle x`

Opakovaná aplikace funkce

- `iterate :: (a -> a) -> a -> [a]`
`iterate f z = z : iterate f (f z)`

Vyhodnoťte

- `take 4 (iterate not True) ~>*`
- `take 4 (iterate (+1) 0) ~>*`

Nekonečné opakování seznamu

- `cycle :: [a] -> [a]`
`cycle x = x ++ cycle x`

Opakovaná aplikace funkce

- `iterate :: (a -> a) -> a -> [a]`
`iterate f z = z : iterate f (f z)`

Vyhodnoťte

- `take 4 (iterate not True) ~>*` `[True,False,True,False]`
- `take 4 (iterate (+1) 0) ~>*` `[0,1,2,3]`

Alternativní definice

- `jednicky = repeat 1`
- `jednicky = iterate (+0) 1`
- `jednicky = iterate (id) 1`
- `jednicky = cycle [1]`
- `nats = iterate (+1) 0`

Další příklady

- `take 10 (iterate (*2) 1) ~>*`
- `take 5 (iterate ('a':) []) ~>*`
- `take 10 (iterate (*(-1)) 1) ~>*`
- `take 8 (cycle "Ha ") ~>*`

Alternativní definice

- `jednicka = repeat 1`
- `jednicka = iterate (+0) 1`
- `jednicka = iterate (id) 1`
- `jednicka = cycle [1]`
- `nats = iterate (+1) 0`

Další příklady

- `take 10 (iterate (*2) 1) ~>*` `[1,2,4,8,16,32,64,128,256,512]`
- `take 5 (iterate ('a':) []) ~>*` `["","a","aa","aaa","aaaa"]`
- `take 10 (iterate *(-1)) 1) ~>*` `[1,-1,1,-1,1,-1,1,-1,1,-1]`
- `take 8 (cycle "Ha ") ~>*` `"Ha Ha Ha"`

Líně vyhodnocená rekurze

- K definici nekonečných datových struktur jsme použili rekurzi.
- Rekurze se zanoří **pouze tolikrát, kolikrát je třeba**.

Příklad

- Seznam nekonečně mnoha jedniček:

```
jednický = 1 : jednický
```

- Vyhodnocení výrazu `jednický` se zacyklí při každé strategii:

```
jednický ~> 1:jednický ~> 1:1:jednický ~> ...
```

- Ale je-li výraz `jednický` podvýrazem většího výrazu, tak se jeho vyhodnocení při líné strategii nemusí zacyklit.

```
head jednický = head jednický ~> head (1:jednický) ~> 1
```

Nekonečný rostoucí seznam všech přirozených čísel

- `nats = 0 : zipWith (+) nats jednicka`

	0	1	2	3	4	5	...	nats
+	1	1	1	1	1	1	...	jednicka
<hr/>								
	0	1	2	3	4	5	6	...

Fibonacciho posloupnost

- `fibs = 0 : 1 : zipWith (+) fibs (tail fibs)`

		0	1	1	2	3	5	...	fibs
+		1	1	2	3	5	8	...	tail fibs
<hr/>									
	0	1	1	2	3	5	8	13	...

Výhody líného vyhodnocování

- potenciálně rychlejší výpočet
- potenciálně vyžaduje méně paměti
- jednoduchá práce s nekonečnými strukturami
- dobrá abstrakce (volající určuje, kolik je potřeba výsledku)

Nevýhody líného vyhodnocování

- potenciálně pomalejší výpočet
- potenciálně náročnější na paměť
- čas výpočtu se obtížněji předpovídá
- obtížně se kombinuje s vedlejšími efekty (např. vstup/výstup) a výjimkami

Vyhodnocovací strategie v různých jazycích

Prakticky všechny imperativní jazyky používají striktní vyhodnocování:

- Python, C, C++, C#, Java, JavaScript, ...

Prakticky všechny **funkcionální** jazyky používají striktní vyhodnocování:

- Standard ML, OCaml, F#, Scheme, Racket, Scala, Erlang, ...

Haskell je jeden z mála jazyků, který používá téměř čisté líné vyhodnocování.

V některých jazycích jde selektivně udělat části kódu líné:

- `System.Lazy<T>` v C# a `Lazy<T>` v Javě
- `delay/force` v Scheme a `lazy/force` v OCaml
- částečně generátory v Pythonu a JavaScriptu

Zápis a generování seznamů

Prostý výčet

- Zápis pomocí základních hodnotových konstruktorů `(:)` a `[]`

```
1:2:3:4:5:[]
```

- Ekvivalentní zkrácený zápis (syntaktická zkratka pro totéž).

```
[1,2,3,4,5]
```

Hromadný výčet

- Seznamy hodnot, které lze systematicky vyjmenovat (enumerovat) lze zadat tzv. **hromadným výčtem**.

- Seznamy zadané enumerační funkcí `enumFromTo`

```
enumFromTo 1 12 ~>* [1,2,3,4,5,6,7,8,9,10,11,12]
```

```
enumFromTo 'A' 'Z' ~>* "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Nekonečná enumerace

- Enumerovat lze i hodnoty typů s nekonečnou doménou.
- Hromadným výčtem lze definovat nekonečné seznamy.

```
nats = enumFrom 0
```

Enumerace s udaným vzorem

- Udáním druhého prvku lze definovat vzor enumerace:

```
take 10 (enumFromThen 0 2) ~>* [0,2,4,6,8,10,12,14,16,18]
```

```
enumFromThenTo 0 3 15 ~>* [0,3,6,9,12,15]
```

Přehled enumeračních funkcí a syntaktických zkratk

Enumerační funkce	Typ	Zkratka
<code>enumFrom m</code>	<code>Enum a => a->[a]</code>	<code>[m..]</code>
<code>enumFromTo m n</code>	<code>Enum a => a->a->[a]</code>	<code>[m..n]</code>
<code>enumFromThen m m'</code>	<code>Enum a => a->a->[a]</code>	<code>[m,m'..]</code>
<code>enumFromThenTo m m' n</code>	<code>Enum a => a->a->a->[a]</code>	<code>[m,m'..n]</code>

Intenzionální definice seznamu

- Prvky seznamu jsou generovány společným pravidlem, které předepisuje jak prvky z nějaké nosné množiny přepsat na prvky generovaného seznamu.
- Příklad: prvních deset násobků čísla 2
[2*n | n <- [0..9]]

Obecná šablona

- [definiční_výraz | generátory a kvalifikátory]
- Za každý vygenerovaný prvek vyhovující všem kvalifikátorům se do definovaného seznamu přidá jedna hodnota definičního výrazu.
- Definiční výraz může a nemusí použít generované prvky.
- Kvalifikátory a generátory se vyhodnocují **zleva doprava**.

Generátor

- `nová_proměnná <- seznam` nebo `vzor <- seznam`
- Definuje novou proměnou použitelnou buď v definičním výrazu nebo v libovolném kvalifikátoru vyskytujícím se vpravo.
- Nová proměnná postupně nabývá hodnot prvků v seznamu.
- V případě použití vzoru, se vygeneruje tolik instancí, kolik prvků v seznamu odpovídá použitému vzoru.

Kombinace více generátorů

- Při použití více generátorů se generují všechny kombinace.
- Pořadí kombinací je dáno uspořádáním generátorů v definici.
- Generátor vlevo iteruje nejpomaleji, generátor vpravo nejrychleji.

Predikát

- Výraz typu `Bool` .
- Může využít proměnné definované od predikátu vlevo.
- Vygenerované instance, které nevyhovují predikátu, nebudou brány v potaz pro definici výsledného seznamu.

Lokální definice

- `let nová_proměnná = výraz`
- Definuje novou proměnnou použitelnou buď v definičním výrazu nebo v libovolném generátoru nebo kvalifikátoru vyskytujícím se vpravo.
- Výraz může využít proměnné definované vlevo.

- [$n^2 \mid n \leftarrow [0..3]$]
 \rightsquigarrow^* [0,1,4,9]
- [(c,k) | c \leftarrow "abc", k \leftarrow [1,2]]
 \rightsquigarrow^* [('a',1),('a',2),('b',1),('b',2),('c',1),('c',2)]
- [3*n | n \leftarrow [0..6], odd n]
 \rightsquigarrow^* [3,9,15]
- [(m,n) | m \leftarrow [1..3], n \leftarrow [1..3], n \leq m]
 \rightsquigarrow^* [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
- [(m,n) | m \leftarrow [1..3], n \leftarrow [1..m]]
 \rightsquigarrow^* [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
- [(x,y) | z \leftarrow [0..2], x \leftarrow [0..z], let y=z-x]
 \rightsquigarrow^* [(0,0),(0,1),(1,0),(0,2),(1,1),(2,0)]

- `[replicate n c | c<-"xyz", n<-[2,3]]`
 \rightsquigarrow^* `["xx","xxx","yy","yyy","zz","zzz"]`
- `[replicate n c | n<-[2,3], c<-"xyz"]`
 \rightsquigarrow^* `["xx","yy","zz","xxx","yyy","zzz"]`
- `[x^2 | [x]<-[[], [2,3], [4], [1,1..], [], [7], [0..]]]`
 \rightsquigarrow^* `[16,49]`
- `[0 | []<-[[], [2,3], [4], [0..], [], [5]]]`
 \rightsquigarrow^* `[0,0]`
- `[x^3 | x<-[0..10], odd x]`
 \rightsquigarrow^* `[1,27,125,343,729]`
- `[x^3 | x<-[0..10], odd x, x < 1]`
 \rightsquigarrow^* `[]`

Redefinice známých funkcí

- `length :: [a] -> Int`
`length s = sum [1 | _ <- s]`
- `map :: (a->b) -> [a] -> [b]`
`map f s = [f x | x <- s]`
- `filter :: (a->Bool) -> [a] -> [a]`
`filter p s = [x | x <- s, p x]`
- `concat :: [[a]] -> [a]`
`concat s = [x | t <- s, x <- t]`

Nové funkce

- `isOrdered :: Ord a => [a] -> Bool`
`isOrdered s = and [x<=y | (x,y) <- zip s (tail s)]`
- `samohlasky :: String -> String`
`samohlasky s = [v | v <- s, v `elem` "aeiouy"]`

Úkol

- Napište funkci, která při aplikaci na konečný seznam uspořadatelných hodnot vrátí seznam těchto hodnot uspořádaných operátorem `<`.

Řešení

- Funkce `qSort` seřadí seznam hodnot vzestupně.
- `qSort :: Ord a => [a] -> [a]`
`qSort [] = []`
`qSort (p:s) = qSort [x | x<-s, x<p]`
 `++ [p] ++`
 `qSort [x | x<-s, x>=p]`

Prvočísla – Eratosthenovo síto

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	
	3		5		7		9		11		13		15		17		19		21		23		25		27		29	
		5		5		7			11		13				17		19				23		25				29	
			7			7			11		13				17		19				23							29
									11		13				17		19				23							29
											13				17		19				23							29
															17		19				23							29
																	19				23							29
																						23						29
																							23					29
																								23				29
																									23			29
																										23		29
																											23	29

Prvočísla

- Pro každé p , $2 \leq p \in \mathbb{N}$ platí: p je prvočíslo, právě když p není násobkem žádného prvočísla menšího než p .
- `es :: Integral a => [a] -> [a]`
`es (p:t) = p : es [n | n<-t, n`mod`p/=0]`

```
primes = es [2..]
```


Pozorování

- Využití generátorů seznamů ve spojení s nekonečnými seznamy trochu kazí čistotu deklarativního přístupu, ve kterém se v zásadě nezajímáme o způsob vyhodnocení, ale pouze o podstatu vyjádření vztahů.

Porovnejte a vysvětlete rozdíl

- `take 4 [x+y | x <- [1..], y<-[1..], x<2] ~>*`
- `take 4 [x+y | y <- [1..], x<-[1..], x<2] ~>*`

Pozorování

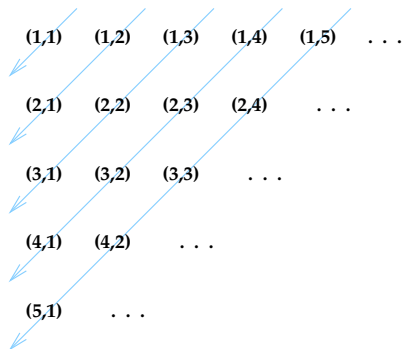
- Využití generátorů seznamů ve spojení s nekonečnými seznamy trochu kazí čistotu deklarativního přístupu, ve kterém se v zásadě nezajímáme o způsob vyhodnocení, ale pouze o podstatu vyjádření vztahů.

Porovnejte a vysvětlete rozdíl

- `take 4 [x+y | x <- [1..], y<-[1..], x<2]` \rightsquigarrow^* [2,3,4,5]
- `take 4 [x+y | y <- [1..], x<-[1..], x<2]` \rightsquigarrow^* [2
(druhý výpočet cyklí)

Definice seznamů

- Definujte seznam všech uspořádaných dvojic přirozených čísel tak, aby dvojice byly v definovaném seznamu uspořádány dle následujícího schématu:



- Nápověda: součet čísel v dvojici je po diagonále shodný a postupně se zvyšuje o jedna.

IB015 Neimperativní programování

A zase ta REKURZE!
(rekurze na typech)

Jiří Barnat
Libor Škarvada

Uživatелеm definované typy

(algebraické typy)

Pozorování

- Počítač veškerá data reprezentuje čísly.
- Programátoři jej dobrovolně, či nedobrovolně napodobují.

Riziko

- Mnohdy číselná reprezentace různých hodnot není přímočará a tedy umožňuje nechtěné zadání neplatných hodnot.
- Neplatné hodnoty mohou vzniknout i neopatrnou aplikací číselných operací.
- **Použití neplatných hodnot může být nebezpečné.**

Příklad

- Chceme reprezentovat den v týdnu a definovat funkce pracující s touto reprezentací.

- Možné číselné kódování, je následující:

pondělí = 1, úterý = 2, ..., neděle = 7

- Funkce `zitra` (s chybou) a funkce `je_pondeli` :

```
zitra :: Int -> Int
```

```
zitra x = x+1
```

- nesprávně i (x+1) 'mod' 7

```
je_pondeli :: Int -> Bool
```

```
je_pondeli x = if (x==1) then True else False
```

Chyba ve výpočtu

```
je_pondeli 8 ~> False
```

```
je_pondeli (zitra 7) ~> ... ~> False
```

Definice typů

- V Haskellu pomocí klíčového slova `data` .
- Obecná šablona:
`data Název_typu = Hodnotové_konstruktory`
- Jednotlivé hodnotové konstruktory se oddělují znakem `|`
- Syntaktické omezení Haskellu: nově definovaný typ i hodnotové konstruktory musí začínat velkým písmenem.

Příklad

- Dny v týdnu lze definovat jako nový typ, který má 7 hodnot.
`data Dny = Po | Ut | St | Ct | Pa | So | Ne`
- Hodnoty jsou definovány výčtem.
- Jsou použity nulární hodnotové konstruktory – konstanty.

Uživatелеm definované

- Obecná definice n-árního hodnotového konstruktoru:

Jméno **Typ**₁ ... **Typ**_n

- Příklad typu s ternárním hodnotovým konstruktorem:

```
data Barva = RGB Int Int Int
```

- Hodnoty typu Barva:

```
RGB 42 42 42
```

```
RGB 12 (-23) 45
```

Částečná aplikace hodnotového konstruktoru

```
RGB          :: Int -> Int -> Int -> Barva
```

```
RGB 23       ::          Int -> Int -> Barva
```

```
RGB 23 23    ::                Int -> Barva
```

```
RGB 23 23 23 ::                    Barva
```

Typové konstanty

- Definicí dle šablony:

```
data Název_typu = Hodnotové_konstruktory  
zavádíme nový typ s označením Název_typu.
```

- `Název_typu` je nulární typový konstruktork, typová konstanta.

N-ární typové konstruktory

- Typové konstruktory jako například `->` nebo `[]` nedefinují typ, pouze předpis jak nový typ vyrobit.

Tvorba typu

- Každá typová konstanta definuje typ.
- Typ získám také úplnou aplikací n-árních typových konstruktorů na již definované typy.

```
(->) Dny Bool = Dny -> Bool
```

```
[] Dny = [Dny]
```

```
(->) (Dny -> Bool) [Dny] = (Dny -> Bool) -> [Dny]
```

Tvorba nových hodnot

- Aplikace hodnotových konstruktorů vytváří nové hodnoty.

Tvorba nových typů

- Aplikace typových konstruktorů vytváří nové typy.

Uspořádané n-tice a seznamy

- Používá se stejné označení pro typové i hodnotové konstruktory!

'a' :: Char

[('a','a'), ('a','a')] :: [(Char,Char)]

[('a','a'), ('a','a')] :: [(Char,Char)]

- **hodnotové**

- **typové**

Polymorfní typové konstruktory

- Seznam prvků typu a , strom hodnot typu a , ...

Definice polymorfních typových konstruktorů

- Definice s využitím typových proměnných:
data **Název_typu** $a_1 \dots a_n = \dots$
- Typové proměnné lze použít pro definici hodnotových konstruktorů.

Kompletní obecná šablona

```
data Tcons  $a_1 \dots a_n =$  Dcons1 typ(1,1) typ(1,2) ... typ(1,arita1)  
      ⋮  
      Dcons $m$  typ( $m$ ,1) typ( $m$ ,2) ... typ( $m$ ,arita $m$ )
```

Maybe

- Předdefinovaný unární polymorfní typový konstruktor.
`data Maybe a = Nothing | Just a`
- Zamýšlené použití pro funkce, jejichž hodnota může být nedefinována.

Příklad

- Chceme ošetřit dělení nulou, definujeme novou funkci `deleni`
`deleni :: Fractional a => a -> a -> Maybe a`
`deleni x y = if (y==0) then Nothing else Just (x/y)`
- Jaký je výsledek aplikace `deleni` na argumenty `32` a `8` ?

Maybe

- Předdefinovaný unární polymorfní typový konstruktor.
`data Maybe a = Nothing | Just a`
- Zamýšlené použití pro funkce, jejichž hodnota může být nedefinována.

Příklad

- Chceme ošetřit dělení nulou, definujeme novou funkci `deleni`
`deleni :: Fractional a => a -> a -> Maybe a`
`deleni x y = if (y==0) then Nothing else Just (x/y)`
- Jaký je výsledek aplikace `deleni` na argumenty `32` a `8` ?
Just 4.0
- Proč je následující definice špatně?
`deleni x y = if (y==0) then Nothing else (x/y)`

type vs. data

- Zatímco s využitím `data` definuji nové typy, pomocí `type` zavádím typové aliasy (synonyma) k již existujícím typům.
- Používá se pro lepší čitelnost kódu.

Příklady

- `type String = [Char]`
- `type Day = Int`
`type Month = Int`
`type Year = Int`
`type Date = (Day,Month,Year)`

Jiný pohled na rekurzivní funkce

Rekurze

- Definice funkce, nebo datové struktury, s využitím sebe sama.

Příklad

- Funkce `length`, která při aplikaci na seznam vrací jeho délku, je definovaná rekurzivně:

```
length :: [a] -> Integer
length [] = 0
length (_:s) = 1 + length s
```

Zacyklení výpočtu

- Ne každé použití definovaného objektu na pravé straně definice je smysluplné.
- Nesprávné použití může vést k nekonečnému vyhodnocování, které nemá žádný efekt – **výpočet cyklů**.

Příklad

- Nesprávné použití rekurze ve funkci `length'` :

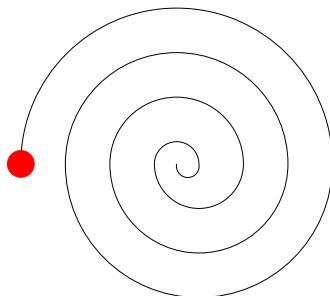
```
length' :: [a] -> Integer
length' [] = 0
length' x = length' x
```
- Při aplikaci `length'` na neprázdný seznam výpočet cyklů.
- Chybu neodhalí typová kontrola, definice je typově správně.

Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
= 1 + length [3,2,1]
= 1 + 1 + length [2,1]
= 1 + 1 + 1 + length [1]
= 1 + 1 + 1 + 1 + length []
= 1 + 1 + 1 + 1 + 0
= 4
```

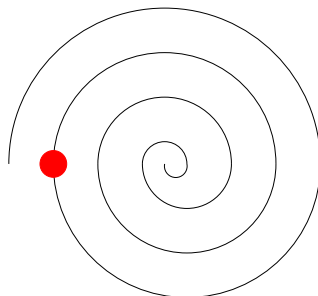


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

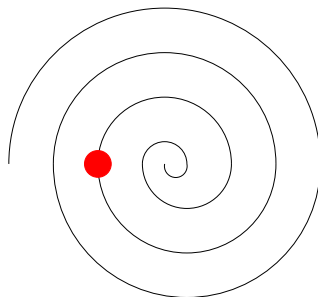


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

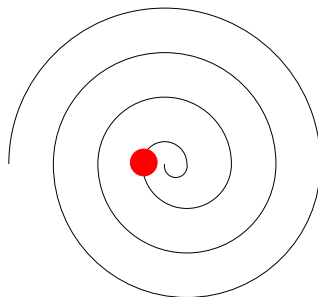


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

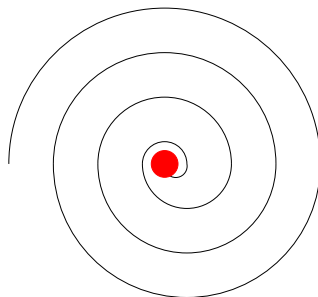


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

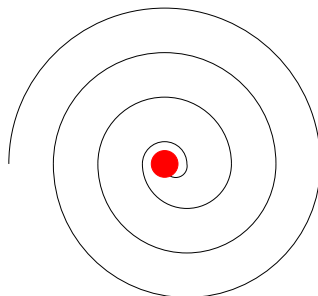


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

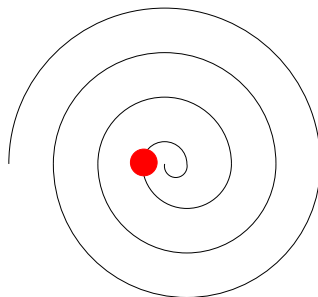


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

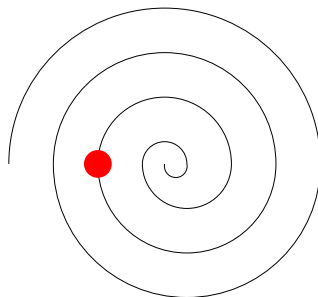


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

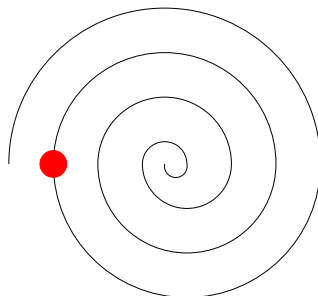


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```

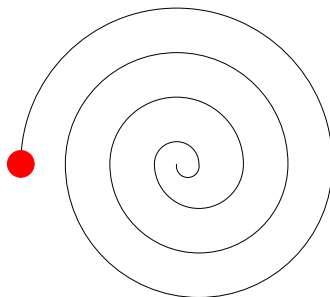


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu**.
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
  = 1 + length [3,2,1]
  = 1 + 1 + length [2,1]
  = 1 + 1 + 1 + length [1]
  = 1 + 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 1 + 0
  = 4
```



Pozorování

- Uvědomění si toho, co udává vzdálenost od středu pomyslné spirály, je klíč k správnému použití rekurze.

Rekurze ve funkci `length`

- Vzdálenost od středu odpovídá délce zbývajících částí seznamu.
- S každým dalším rekurzivním voláním funkce se seznam, který je argumentem funkce, zkracuje.
- Funkce `length` je tedy jednou nevyhnutelně volána pro prázdný seznam, což je volání, které rekurzi zastaví.

2 části definice

- Při definici rekurzivní funkce je nutné si uvědomit, co je **středem spirály**, tj. kde se má výpočet rekurzivní funkce zastavit, a **jak se k tomuto středu bude výpočet blížit**.

Příklad – 2 části definice funkce length

- Ukončení rekurzivního výpočtu (střed spirály)

```
length [] = 0
```

- Jedno rekurzivní volání (přiblížení se o "jednu otáčku")

```
length (x:s) = 1 + length s
```

Příklad – 2 části definice v jednom výrazu

- Obě části v jednom řádku definice

```
f1 :: Integer -> Integer
```

```
f1 x = if (odd x) then x else f1 (x `div` 2)
```

Rekurzivní funkce a větvení

- V případě, že se výpočet funkce větví, vzdálenost od středu pomyslné spirály musí klesat s každou větví.
- Musí existovat větev, která rekurzi ukončuje a je proveditelná, pokud jsme ve středu pomyslné spirály.
- `f2 :: Integer -> Integer`
`f2 x = if (x==0) then 0` - chyba, případ nemusí nastat
`else if (odd x) then f2 (x-2)`
`else f2 (x-1)`

Funkce s nekonečnou rekurzí

- Teoreticky je možné použít rekurzi pro realizaci nekonečného cyklu. V praxi však toto řešení nemusí fungovat vzhledem k omezené velikosti paměti pro uchovávání návratových adres.

Vzdálenost od středu

- To, že pomyslná vzdálenost od středu klesá, nemusí nutně znamenat, že datová struktura, se kterou rekurzivní funkce pracuje, se zmenšuje.

Příklad

- Je-li cílem algoritmu opakovaným dělením celku dosáhnout určitého počtu dílků, počet dílků při každém dělení roste.
- Vzdálenost od středu pomyslné spirály lze v tomto případě identifikovat jako počet dělení, které zbývá k dosažení cílového počtu.
- Všimněme si, že pokud se při každém kroku zdvojnásobí počet dílků, jejich počet roste vzhledem k počtu rekurzivních kroků exponenciálně.

Pozadí rekurze

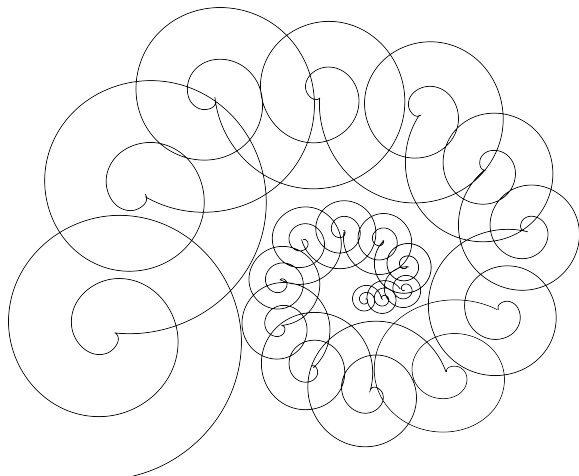
- Struktura, podle níž se řídí rekurze, nemusí být spojena s úplným uspořádáním.
- Musí však být **dobře založená** (well-founded), což znamená, že v ní neexistuje nekonečně dlouhá klesající posloupnost prvků.

Příklad

- Množina všech podmnožin dané množiny je pouze částečně uspořádána vzhledem k inkluzi, avšak postupné odebrání prvků z libovolné podmnožiny nevyhnutelně dospěje k prázdné množině.

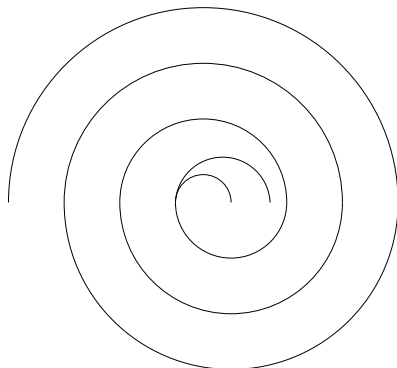
Vnořená rekurze

- Rekurzivně volané funkce se mohou vnořovat.
- "Spirála spirál".



Rozeklaná spirála

- Spirála je v místě dosažení středu rozeklaná, tj. končí ve dvou a více bázových případech.



Rozeklaná spirála

- Spirála je v místě dosažení středu rozeklaná, tj. končí ve dvou a více bázových případech.



Příklad

- Definujte funkci, která pro zadaný seznam vrátí seznam, který vznikne z původního seznamu vynecháním všech prvků na sudých pozicích.
- `oddMembers [1,2,3,4,5,6,7,8] ~>* [1,3,5,7]`
- `oddMembers "Trol ej ej schomoula." ~>* "To je cool."`

Myšlenka a definice

- Rekurzivní volání zkracuje zadaný seznam vždy o dva prvky.
- Krajními případy jsou **prázdný** a **jednoprvkový** seznam.

- `oddMembers :: [a] -> [a]`
`oddMembers [] = []`
`oddMembers (x:[]) = [x]`
`oddMembers (x:y:s) = x : oddMembers s`

Rekurzivní datové struktury

Pozorování

- Pro definici rekurzivních datových struktur (hodnot rekurzivních typů) platí podobná pravidla jako pro definice rekurzivních funkcí.

Opačný směr

- Vytváření hodnot rekurzivního datového typu probíhá od středu pomyslné spirály směrem ven.
- Rekurzivní datová struktura má **základní** (bázovou) **hodnotu**.
- Základní hodnota je rozvíjena **rekurzivním pravidlem**.

Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

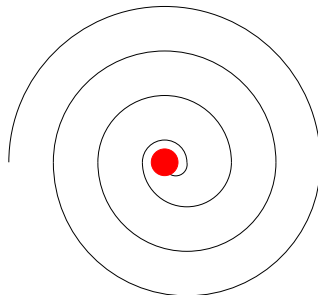
[]

`[1]` = 1 : `[]`

`[2,1]` = 2 : `[1]`

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

`[4,3,2,1]` = 4 : `[3,2,1]`



Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

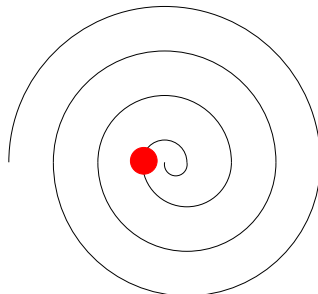
`[]`

`[1] = 1 : []`

`[2,1] = 2 : [1]`

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

`[4,3,2,1] = 4 : [3,2,1]`



Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

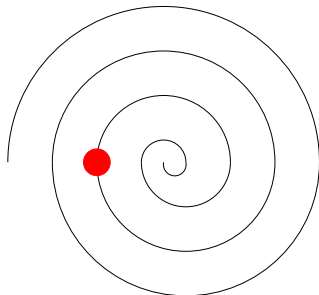
`[]`

`[1] = 1 : []`

`[2,1] = 2 : [1]`

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

`[4,3,2,1] = 4 : [3,2,1]`



Klasický pohled na seznam

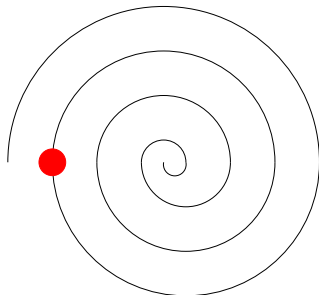
- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

```
[]  
[1] = 1 : []  
[2,1] = 2 : [1]  
[3,2,1] = 3 : [2,1]  
[4,3,2,1] = 4 : [3,2,1]
```



Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `(a : seznam)` je **seznam**.

Demonstrace

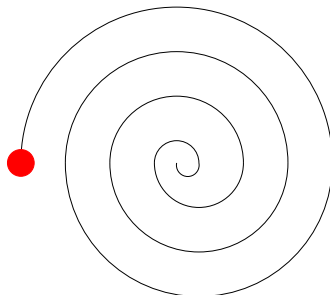
`[]`

`[1] = 1 : []`

`[2,1] = 2 : [1]`

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

`[4,3,2,1] = 4 : [3,2,1]`



Pozorování

- Rekurzivní nahlížení na seznam se může jevit jen jako mentální hříčka.

Stromy jako rekurzivní datové struktury

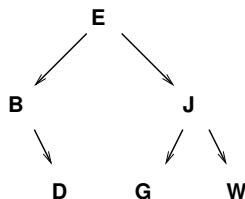
- Mnoho problémů je přirozené řešit s využitím jiné rekurzivně definované datové struktury – **binárního stromu**.
- Nelineární rekurzivní datová struktura.

Rekurzivní definice binárního stromu

- Prázdný strom je **binární strom**
- Hodnota a a k ní asociovaný levý a pravý **binární strom** je **binární strom**

Příklad

- Graficky zadaný binární strom.
- E označujeme jako **kořen** stromu
- E, B a J jsou **vnitřní vrcholy** stromu
- D, G a W označujeme jako **listy**
- Levý a pravý binární strom asociovaný s danou hodnotou označujeme jako levý a pravý **podstrom**.
- Binární stromy asociované k hodnotám D, G, W a levý podstrom asociovaný s hodnotou B jsou prázdné stromy.



Definice datového typu `BinTree a`

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

Příklady hodnot definovaného typu

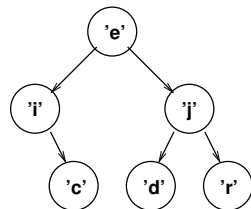
```
tc :: BinTree Char
```

```
tc = Node 'e'
```

```
    (Node 'i' Empty (Node 'c' Empty Empty))
```

```
    (Node 'j' (Node 'd' Empty Empty)
```

```
          (Node 'r' Empty Empty))
```



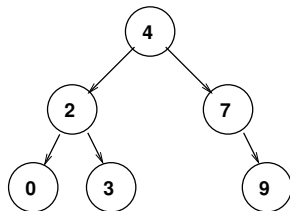
```
tn :: BinTree Int
```

```
tn = Node 4
```

```
    (Node 2 (Node 0 Empty Empty)
```

```
          (Node 3 Empty Empty))
```

```
    (Node 7 Empty (Node 9 Empty Empty))
```



Problém

- Chceme definovat funkci, která při aplikaci na hodnotu typu `BinTree Int` zvýší o jedna všechny hodnoty uložené v uzlech stromu.

Jak takovou funkci definovat?

- Výčtem hodnot nelze – možných hodnot je nekonečně mnoho.

```
treeP1' :: Num a => BinTree a -> BinTree a
treeP1' Empty = Empty
treeP1' (Node x Empty Empty) = Node (x+1) Empty Empty
⋮
```

- **Rekurzivně**, rekurzi vedeme podle struktury stromu

```
treeP1 :: Num a => BinTree a -> BinTree a
treeP1 Empty = Empty
treeP1 (Node x left right)
    = Node (x+1) (treeP1 left) (treeP1 right)
```


Popis funkce `treezipwith`

- Funkce `treezipwith` pomocí binární operace `op` vytvoří ze dvou stromů nový strom, jehož struktura bude průnikem obou stromů a v jehož uzlech budou výsledky aplikace operace `op` na hodnoty uzlů ze stejné pozice v obou stromech.

Pozorování

- V následujícím typu, nemá rekurzivní sestup pevnou aritu.

Příklad

```
data Policie = Hlidka (String,String) | Oddeleni [Policie]
              deriving Show
```

```
h1 = Hlidka ("Pepa", "Emil")
h2 = Hlidka ("Jason", "Drson")
o1 = Oddeleni [h1, h2]
```

```
jmena :: Policie -> [String]
jmena (Hlidka (a,b)) = a:b:[]
jmena (Oddeleni []) = []
jmena (Oddeleni (x:s)) = jmena x ++ jmena (Oddeleni s)
```

Dokazování rekurzivních programů

(pro chytré hlavičky)

Fakta

- Ověřování správnosti navržených algoritmů je součástí práce programátora.
- Testování je nedokonalé.
- Správnost algoritmu můžeme prokázat například tím, že ji formálně (= s matematickou přesností) dokážeme.

Důkaz korektnosti algoritmu

- Dokazujeme, že pokud výpočet algoritmu na platných vstupech skončí, tak algoritmus vrátí korektní výsledek. O algoritmu, který má tuto vlastnost říkáme, že je **částečně správný**.
- Pokud je algoritmus částečně správný a dokážeme, že na platných vstupech svůj výpočet vždy skončí, pak říkáme, že algoritmus je **úplně správný**.

Pozorování

- Pro důkazy částečné správnosti i terminace rekurzivních funkcí se používá **matematická indukce**.

Matematická indukce

- Matematická indukce je metoda dokazování tvrzení, která se používá, pokud chceme ukázat, že dané tvrzení platí pro všechny prvky dobře založené rekurzivně definované nekonečné posloupnosti. (Jako jsou například přirozená čísla.)

Princip matematické indukce

- Ukážeme platnost tvrzení pro bázovou hodnotu.
- Ukážeme, že tvrzení se přenáší při aplikaci rekurzivního kroku.

$$\underline{T(0)} \text{ a } \underline{T(i) \Rightarrow T(i+1)} \implies T(0), T(1), T(2), \dots$$

Pozorování

- Klíčovým problémem při použití matematické indukce je identifikace toho, podle čeho má být indukce vedena.
- Napovědět může místo rekurzivního volání funkce, neboť rekurse a matematická indukce spolu úzce souvisí.

Příklad

- Dokažte, že pro každá dvě přirozená čísla x a y taková, že $x > 0$ platí, že funkce `fpow` aplikovaná na argumenty x a y vrátí hodnotu x^y .
- `fpow :: Integer -> Integer -> Integer`
`fpow x 0 = 1`
`fpow x y = x * fpow x (y-1)`

Pozorování

- Klíčovým problémem při použití matematické indukce je identifikace toho, podle čeho má být indukce vedena.
- Napovědět může místo rekurzivního volání funkce, neboť rekurse a matematická indukce spolu úzce souvisí.

Příklad

- Dokažte, že pro každá dvě přirozená čísla x a y taková, že $x > 0$ platí, že funkce `fpow` aplikovaná na argumenty x a y vrátí hodnotu x^y .
- `fpow :: Integer -> Integer -> Integer`
`fpow x 0 = 1`
`fpow x y = x * fpow x (y-1)`
- **Důkaz povedeme indukcí vzhledem k hodnotě y .**

Bázový krok, T(0)

- Necht $y=0$, a necht x je libovolné.
- x^y se redukuje dle $x^0 = 1$ na hodnotu 1 .
- $x^0 = 1$, pro libovolné x .
- Tudíž pro $y = 0$ tvrzení platí.

Indukční krok, $T(i) \Rightarrow T(i+1)$

- Dokazujeme, že pokud tvrzení platí pro hodnotu i , pak tvrzení platí i pro hodnotu $i + 1$. Platnost tvrzení pro hodnotu i se označuje jako **indukční předpoklad**.
- Platnost tvrzení pro hodnotu i říká, že $\text{fpow } x \ i \rightsquigarrow^* x^i$ pro libovolnou hodnotu x .
- $\text{fpow } x \ (i+1)$
 - $\rightsquigarrow x * \text{fpow } x \ (i+1-1)$
 - $\rightsquigarrow x * \text{fpow } x \ i \stackrel{\text{dleIP}}{=} x * x^i$
 - $\rightsquigarrow x^{i+1}$
- Ukázali jsme, že pokud tvrzení platí pro i , pak platí i pro $i + 1$.
- Z platnosti báze a vlastností matematické indukce plyne, že pro libovolnou hodnotu x tvrzení platí pro všechny hodnoty y .

Věta 1

- Jsou-li s , t dva konečné seznamy stejného typu a délek, pak

$$\text{length } (s ++ t) = (\text{length } s) + (\text{length } t).$$

- Důkaz veden indukci podle délky seznamu s .

Věta 2

- Pro každé tři seznamy s , t , u platí rovnost

$$(s ++ t) ++ u = s ++ (t ++ u).$$

- Důkaz veden indukci podle délky seznamu s .

Věta 3

- Pro každý seznam s a celé číslo $m \geq 0$ platí

$$\text{take } m \ s ++ \text{drop } m \ s = s.$$

- Důkaz veden indukci podle m .

Měřítko „vzdálenosti“ rekurze

- Jaká vlastnost čísla x určuje hloubku rekurze při volání následující funkce?

```
f1 x = if (odd x) then x else f1 (x 'div' 2)
```

Ternární stromy

- Zdefinujte nový datový typ, který odpovídá ternárním stromům a naprogramujte funkci, která pro instanci takového stromu zjistí délku nejlevější větve.

IB015 Neimperativní programování

Akumulační funkce, Typové třídy,
Časová složitost

Jiří Barnat
Libor Škarvada

Akumulační funkce

Pozorování

- Seznam je posloupnost oddělených prvků.
- Motivací akumulčních funkcí je “spojit” jednotlivé prvky seznamu dohromady, tj. akumulovat informaci uloženou v těchto jednotlivých prvcích do jedné hodnoty.
- Počet prvků seznamu je variabilní, proto se tato akumulace realizuje pomocí binárního operátoru postupně.

Spojení hodnot v seznamu pomocí binární operace

$$\text{foldl1 } \oplus [x_1, x_2, \dots, x_n] \rightsquigarrow^* ((\dots (x_1 \oplus x_2) \dots) \oplus x_{n-1}) \oplus x_n$$

$$\text{foldr1 } \oplus [x_1, x_2, \dots, x_n] \rightsquigarrow^* x_1 \oplus (x_2 \oplus (\dots (x_{n-1} \oplus x_n) \dots))$$

Příklady použití

```
foldl1 (*) [1,2,3,4,5] ~> ... ~> 120
```

```
foldl1 (&&) [True, True, True, False, True] ~> ... ~> False
```

```
foldl1 (-) [2,3,2] ~> ... ~> -3
```

```
foldr1 (-) [2,3,2] ~> ... ~> 1
```

```
foldr1 (min) [18,12,23] ~> ... ~> 12
```

Funkce `foldl1`, `foldr1` nejsou definovány pro `[]`

```
foldl1 (*) [] ~> ERROR
```

```
foldr1 (&&) [] ~> ERROR
```

Na jednoprvkových seznamech je to identita s kontrolou typu

```
foldr1 (*) [0] ~> 0
```

```
foldr1 (*) [1] ~> 1
```

```
foldr1 (*) [True] ~> ERROR
```


Princip

- Akumulační funkce, které mají fungovat i na prázdných seznamech, vyžadují navíc **iniciální hodnotu** pro proces akumulace.
- Směr závorkování určuje i místo použití iniciální hodnoty.

Akumulace hodnot s využitím iniciální hodnoty

$$\text{foldl } \oplus \ v \ [x_1, x_2, \dots, x_n] \rightsquigarrow^* \left(\left(\dots \left((v \oplus x_1) \oplus x_2 \right) \dots \right) \oplus x_{n-1} \right) \oplus x_n$$

$$\text{foldr } \oplus \ v \ [x_1, x_2, \dots, x_n] \rightsquigarrow^* x_1 \oplus (x_2 \oplus (\dots (x_{n-1} \oplus (x_n \oplus v)) \dots))$$

Příklady

```
foldl (*) 0 [1,2,3,4,5] ~> ... ~> 0
```

```
foldl (&&) False [True, True, True, True] ~> ... ~> False
```

```
foldl (-) 0 [2,3,2] ~> ... ~> -7
```

```
foldr (-) 0 [2,3,2] ~> ... ~> 1
```

Aplikace na prázdné seznamy

```
foldl max 100 [] ~> ... ~> 100
```

```
foldr (++) "Nic" [] ~> ... ~> "Nic"
```

Výsledek může být opět seznam!

```
foldr (:) [] "Coze?" ~> ... ~> "Coze?"
```

```
foldr (\x y->(x+1):y) [100] [1,2,3] ~> ... ~> [2,3,4,100]
```

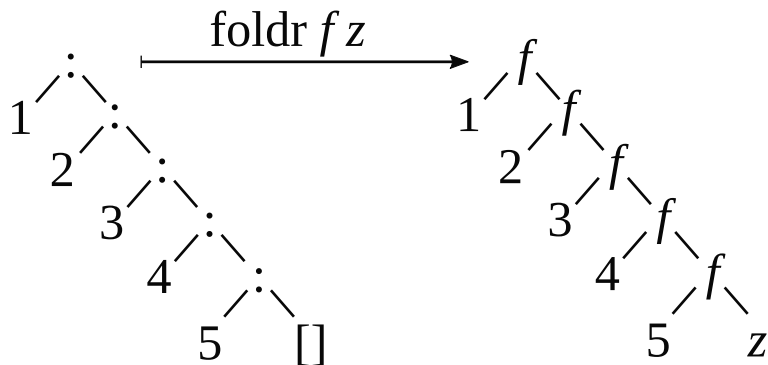
Definice akumuláčnických funkcí

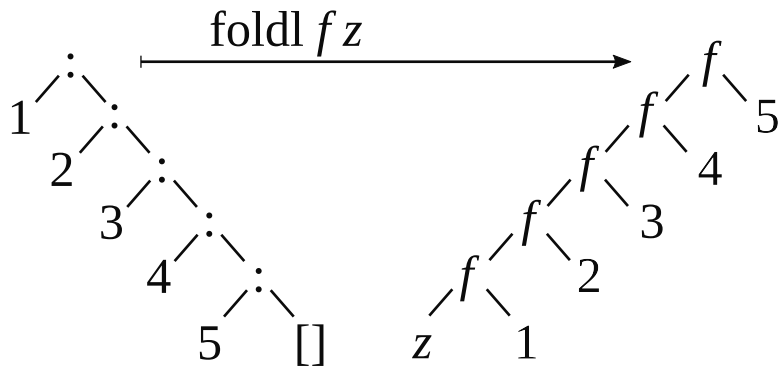
```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ v [] = v
foldl op v (x:s) = foldl op (v 'op' x) s
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v [] = v
foldr op v (x:s) = x 'op' foldr op v s
```

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 op (x:s) = foldl op x s
```

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 _ [x] = x
foldr1 op (x:s) = x 'op' foldr1 op s
```





Katamorfismus

- Výraz vzniklý nahrazením hodnotových konstruktorů v nějaké hodnotě algebraického typu jinými funkcemi vhodné arity.
- Nově vzniklý výraz je následně možné vyhodnotit (pokud lze).

Katamorfismus na seznamech

- Realizován funkcí `foldr`.
- Porovnejte typy hodnotových konstruktorů seznamu

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

`[]` :: `[a]`

s typem funkce `foldr`

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

Peanova čísla

- `data Nat = Succ Nat | Zero`

Katamorfismus na typu Nat

- `natFold :: (a -> a) -> a -> Nat -> a`
`natFold s z (Succ n) = s (natFold s z n)`
`natFold s z Zero = z`

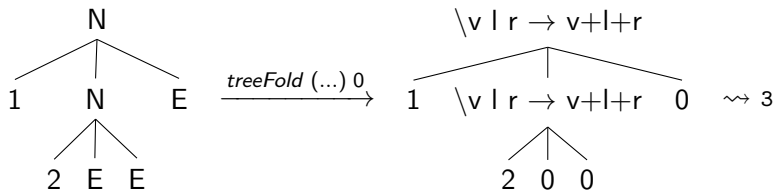
Příklad

- Katamorfismus: `Zero → 0`
`Succ → (1+)`
- `natFold (1+) 0 (Succ (Succ Zero)) \rightsquigarrow^* (1+) ((1+) 0) \rightsquigarrow^* 2`

Fold na binárních stromech

- `data BinTree a = Node a (BinTree a) (BinTree a) | Empty`
- `let tree = Node 1 (Node 2 Empty Empty) Empty in
treeFold (\v l r -> v+l+r) 0 tree
 \rightsquigarrow^* 1+(2+0+0)+0 \rightsquigarrow^* 3`

Graficky:



Typové třídy

Monomorfní typy

- `not :: Bool -> Bool`
`(&&) :: Bool -> Bool -> Bool`

Polymorfní typy

- `length :: [a] -> Int`
`flip :: (a -> b -> c) -> b -> a -> c`

Kvalifikované typy

- `(==), (/=) :: Eq a => a -> a -> Bool`
`sum, product :: Num a => [a] -> a`
`minimum, maximum :: Ord a => [a] -> a`
`print :: Show a => a -> IO ()`

Typové třídy

- Sdružují a identifikují typy se společnými vlastnostmi.
- Typová kvalifikace – `Ord`, `Nat`, `Eq`, `Show`, `Foldable`, ...

Programátorský význam

- Existence typových tříd umožňuje i ve striktně typovaných jazycích sdílet kód funkcí, které dělají totéž, avšak pracují s hodnotami různých typů.
- Sdílení kódu funkcí, které dělají totéž, by měl být **svatý grál** všech programátorů.



Typová třída Eq

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

Přidružení typů k typové třídě (deklarace instance)

- `instance Eq Bool where`
 `False == False = True`
 `True == True = True`
 `_ == _ = False`
- `instance Eq Int where`
 `(==) = primEqInt`
- `instance (Eq a, Eq b) => Eq (a,b) where`
 `(x,y) == (u,v) = x == u && y == v`

Typová třída Ord využívající typovou třídu Eq

- `class (Eq a) => Ord a where`
 `(<=), (>=), (<), (>) :: a -> a -> Bool`
 `max, min :: a -> a -> a`
 `x >= y = y <= x`
 `x < y = x <= y && x /= y`
 `x > y = y < x`
 `max x y = if x >= y then x else y`
 `min x y = if x <= y then x else y`

Deklarace instance

- `instance Ord Bool where`
 `False <= _ = True`
 `_ <= True = True`
 `_ <= _ = False`
- `instance (Ord a, Ord b) => Ord (a,b) where`
 `(x,y) <= (u,v) = x < u || (x == u && y <= v)`

Pozorování

- Instanciací lze přenést vlastnosti typu na složené typy.

Příklad

- Rozšíření uspořadatelnosti hodnot typu na uspořadatelnost seznamů hodnot daného typu.
- ```
instance (Ord a) => Ord [a] where
 [] <= _ = True
 (_:_) <= [] = False
 (x:s) <= (y:t) = x < y || (x == y && s <= t)
```

## Definice typové třídy

- $\text{class } [ (C_1 a, \dots, C_k a) \Rightarrow ] C a$   
     $\left[ \begin{array}{l} \text{where } op_1 :: typ_1 \\ \quad \vdots \\ op_n :: typ_n \\ \quad \left[ \begin{array}{l} default_1 \\ \quad \vdots \\ default_m \end{array} \right] \end{array} \right]$

## Deklarace instance

- $\text{instance } [ (C_1 a_1, \dots, C_k a_k) \Rightarrow ] C typ$   
     $\left[ \begin{array}{l} \text{where } valdef_1 \\ \quad \vdots \\ valdef_n \end{array} \right]$



## Přetížení

- Má-li třída více než jednu instanci, jsou její funkce **přetíženy**.

## Přetížení operací

- Jedna operace je pro několik různých typů operandů definována obecně různým způsobem.
- To, která definice operace se použije při výpočtu, závisí na typu operandů, se kterými operace pracuje.
- Srovnej s parametricky polymorfními operacemi, které jsou definovány jednotně pro všechny typy operandů.

## Typová třída Num

- `class (Eq a, Show a) => Num a where`  
    `(+), (-), (*) :: a -> a -> a`  
    `negate, abs, signum :: a -> a`

## Přetížení operací při deklaraci instancí

- `instance Num Int where`  
    `(+) = primPlusInt`  
    `⋮`
- `instance Num Integer where`  
    `(+) = primPlusInteger`  
    `⋮`
- `instance Num Float where`  
    `(+) = primPlusFloat`  
    `⋮`

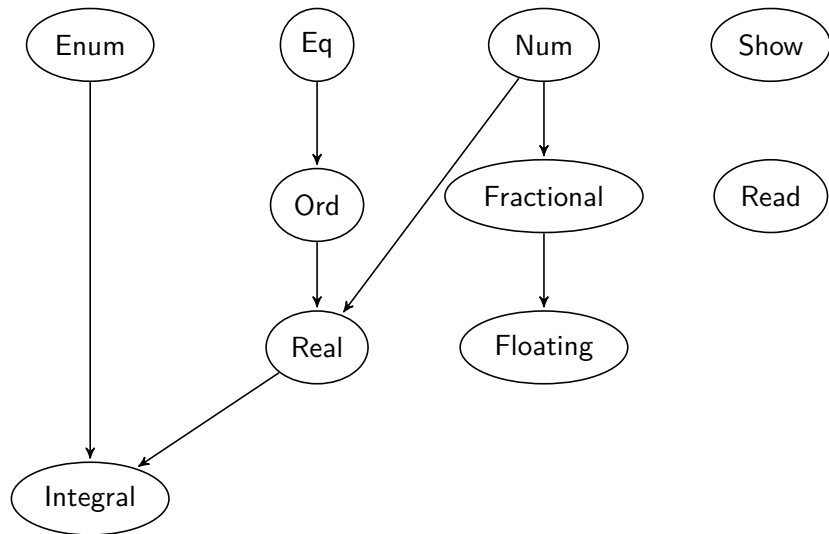
## Implicitní deklarace instance

- V Haskellu lze deklarovat datový typ jako instanci typové třídy (nebo více typových tříd) též implicitně, pomocí klausule `deriving` v definici datového typu.
- Při implicitní deklaraci instance se požadované funkce definují automaticky podle způsobu zápisu hodnot definovaného typu
- Funkce `(==)` se při implicitní deklaraci instance realizuje jako syntaktická rovnost.

## Příklad

- ```
data Nat = Zero | Succ Nat
  deriving (Eq, Show)
```

Hierarchie základních typových tříd



Definice typové třídy

- `class Boolable a where getBool :: a -> Bool`

Instanciace

- `instance Boolable Bool where
 getBool x = x`
- `instance Boolable Int where
 getBool 0 = False
 getBool _ = True`

Použití

- `myIf :: Boolable a => a -> b -> b -> b
 myIf x t f = if getBool x then t else f`
- `myIf (3-3) "Pravda" "Nepravda" ~\~>* "Nepravda"`
- `myIf (3-4) "Pravda" "Nepravda" ~\~>* "Pravda"`

Časová složitost

Podstata

- Časová složitost funkce popisuje **délku výpočtu** v nejhorším případě vzhledem k velikosti vstupních parametrů.

Délka výpočtu v nejhorším případě

- Maximální počet redukčních kroků přes všechny možné výpočty aplikace programu na vstupní parametry stejné velikosti.

Podstata

- Časová složitost funkce popisuje **délku výpočtu** v nejhorším případě vzhledem k velikosti vstupních parametrů.

Délka výpočtu v nejhorším případě

- Maximální počet redukčních kroků přes všechny možné výpočty aplikace programu na vstupní parametry stejné velikosti.

Času je málo!



Reverze seznamu funkce `reverse'`

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

Reverze seznamu funkce `reverse`

- `reverse :: [a] -> [a]`
`reverse = rev []`
 where `rev s [] = s`
 `rev s (x:t) = rev (x:s) t`

Reverse seznamu funkcí reverse'

reverse' :: [a] -> [a]

reverse' [] = []

reverse' (x:s) = reverse' s ++ [x]

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

[] ++ t = t

(x:s) ++ t = x : (s++t)

```
reverse' [1,2,3]
~> reverse' [2,3] ++ [1]
~> (reverse' [3] ++ [2]) ++ [1]
~> ((reverse' [] ++ [3]) ++ [2]) ++ [1]
~> (([] ++ [3]) ++ [2]) ++ [1]
~> ([3] ++ [2]) ++ [1]
~> (3 : ([2] ++ [1])) ++ [1]
~> (3 : [2]) ++ [1]
~> 3 : ([2] ++ [1])
~> 3 : (2 : ([1] ++ []))
~> 3 : (2 : [1]) ≡ [3,2,1]
```

- Délka výpočtu funkce při aplikaci na seznam délky n .
- **$n+1$**

```
reverse' [1,2,3]
~> reverse' [2,3] ++ [1]
~> (reverse' [3] ++ [2]) ++ [1]
~> ((reverse' [] ++ [3]) ++ [2]) ++ [1]
~> (([] ++ [3]) ++ [2]) ++ [1]

~> ([3] ++ [2]) ++ [1]

~> (3 : ([] ++ [2])) ++ [1]
~> (3 : [2]) ++ [1]

~> 3 : ([2] ++ [1])
~> 3 : (2 : ([] ++ [1]))
~> 3 : (2 : [1])
```

- Délka výpočtu funkce při aplikaci na seznam délky n .
- $n+1 + 1$

```
reverse' [1,2,3]
~> reverse' [2,3] ++ [1]
~> (reverse' [3] ++ [2]) ++ [1]
~> ((reverse' [] ++ [3]) ++ [2]) ++ [1]
~> (([] ++ [3]) ++ [2]) ++ [1]
~> ((3) ++ [2]) ++ [1]
~> (3 : ([] ++ [2])) ++ [1]
~> (3 : [2]) ++ [1]
~> 3 : ([2] ++ [1])
~> 3 : (2 : ([] ++ [1]))
~> 3 : (2 : [1])
```

- Délka výpočtu funkce při aplikaci na seznam délky n .
- $n+1 + 1 + 2$

```
reverse' [1,2,3]
~> reverse' [2,3] ++ [1]
~> (reverse' [3] ++ [2]) ++ [1]
~> ((reverse' [] ++ [3]) ++ [2]) ++ [1]
~> (([] ++ [3]) ++ [2]) ++ [1]

~> ([3] ++ [2]) ++ [1]

~> (3 : ([] ++ [2])) ++ [1]
~> (3 : [2]) ++ [1]

~> 3 : ([2] ++ [1])
~> 3 : (2 : ([] ++ [1]))
~> 3 : (2 : [1])
```

- Délka výpočtu funkce při aplikaci na seznam délky n .
- $n+1 + 1 + 2 + 3 + \dots + n$

```
reverse' [1,2,3]
~> reverse' [2,3] ++ [1]
~> (reverse' [3] ++ [2]) ++ [1]
~> ((reverse' [] ++ [3]) ++ [2]) ++ [1]
~> (([] ++ [3]) ++ [2]) ++ [1]

~> ([3] ++ [2]) ++ [1]

~> (3 : ([] ++ [2])) ++ [1]
~> (3 : [2]) ++ [1]

~> 3 : ([2] ++ [1])
~> 3 : (2 : ([] ++ [1]))
~> 3 : (2 : [1])
```

- Délka výpočtu funkce při aplikaci na seznam délky n .
- $n+1 + 1 + 2 + 3 + \dots + n$

```
reverse' [1,2,3]
~> reverse' [2,3] ++ [1]
~> (reverse' [3] ++ [2]) ++ [1]
~> ((reverse' [] ++ [3]) ++ [2]) ++ [1]
~> (([] ++ [3]) ++ [2]) ++ [1]

~> ([3] ++ [2]) ++ [1]

~> (3 : ([] ++ [2])) ++ [1]
~> (3 : [2]) ++ [1]

~> 3 : ([2] ++ [1])
~> 3 : (2 : ([] ++ [1]))
~> 3 : (2 : [1])
```

```
reverse :: [a] -> [a]
```

```
reverse = rev []
```

```
  where rev s [] = s
```

```
        rev s (x:t) = rev (x:s) t
```

```
reverse [1,2,3]
```

```
  ~> rev [] [1,2,3]
```

```
  ~> rev [1] [2,3]
```

```
  ~> rev [2,1] [3]
```

```
  ~> rev [3,2,1] []
```

```
  ~> [3,2,1]
```


- Délka výpočtu funkce při aplikaci na seznam délky n .
- **1**

```
reverse [1,2,3]
~> rev [] [1,2,3]
~> rev [1] [2,3]
~> rev [2,1] [3]
~> rev [3,2,1] []
~> [3,2,1]
```

- Délka výpočtu funkce při aplikaci na seznam délky n .
- $1 + n$

```
reverse [1,2,3]
~> rev [] [1,2,3]
~> rev [1] [2,3]
~> rev [2,1] [3]
~> rev [3,2,1] []
~> [3,2,1]
```

- Délka výpočtu funkce při aplikaci na seznam délky n .
- $1 + n + 1$

```
reverse [1,2,3]
~> rev [] [1,2,3]
~> rev [1] [2,3]
~> rev [2,1] [3]
~> rev [3,2,1] []
~> [3,2,1]
```

- Délka výpočtu funkce při aplikaci na seznam délky n .
- $1 + n + 1$

```
reverse [1,2,3]
~> rev [] [1,2,3]
~> rev [1] [2,3]
~> rev [2,1] [3]
~> rev [3,2,1] []
~> [3,2,1]
```

Pozorování

- Při určování časové složitosti algoritmů je nepraktické a často i obtížné určovat tuto složitost přesně.
- Funkce vyjadřující délku výpočtu vzhledem k velikosti parametru klasifikujeme podle **asymptotického chování**.

Asymptotický růst funkcí

- Při zápisu funkční hodnoty v proměnné n **rozhoduje nejrychleji rostoucí člen**. U něj navíc zanedbáváme kladnou multiplikační konstantu.
- Podle toho hovoříme o funkcích lineárních, kvadratických, exponenciálních apod.

Přehled asymptotických funkcí

t(n)	růst funkce t
1, 20, 729, 2^{64}	konstantní
$2 \log n + 5$, $3 \log_2 n + \log_2(\log_2 n)$	logaritmický
n , $2n + 1$, $n + \sqrt{n}$ $n \log n$, $3n \log n + 6n + 9$ n^2 , $3n^2 + 4n - 1$, $n^2 + 10 \log n$ n^3 , $n^3 + 3n^2$	<i>lineární</i> <i>n log n</i> polynomiální <i>kvadratický</i> <i>kubický</i>
2^n $\left(\frac{1+\sqrt{5}}{2}\right)^n$ 3^n	exponenciální

`reverse'`

- Počet redukčních kroků výrazu `reverse'` $[x_1, \dots, x_n]$ na každém seznamu délky n je

$$n + 1 + 1 + 2 + 3 + \dots + n = \frac{n^2 + 3n + 2}{2}$$

Složitost funkce `reverse'` je **kvadratická** vzhledem k délce obráceného seznamu.

`reverse`

- Počet redukčních kroků výrazu `reverse` $[x_1, \dots, x_n]$ na každém seznamu délky n je

$$1 + n + 1 = n + 2$$

Složitost funkce `reverse` je **lineární** vzhledem k délce obráceného seznamu.

Časová složitost algoritmu

- Posuzuje konkrétní algoritmus.
- Nevypovídá a jiných algoritmech pro řešení téhož problému.

Časová složitost problému

- Daný problém je možné řešit různými algoritmy.
- Složitost problému vypovídá a časové složitosti nejlepšího možného algoritmu pro řešení problému.
- Určovat složitost problému je výrazně obtížnější, než určování složitosti algoritmu.
- Bez znalosti složitosti problému nelze určit, zda daný algoritmus pro řešení problému je optimální.

Pozor

- Časová složitost popisuje délku výpočtu **v nejhorším případě** pro danou velikost argumentu.

Příklad

- Vyšetřujeme časovou složitost funkce `ins` vzhledem k jejímu druhému parametru.
- Funkce `ins` zařazuje prvek do seřazeného seznamu.

```
ins :: Int -> [Int] -> [Int]
```

```
ins x [] = [x]
```

```
ins x (y:t) = if x <= y then x : y : t else y : ins x t
```

Různé délky výpočtu

- Počet kroků při volání `ins x [x1, ..., xn]` je různý.

- Nejkratší výpočet má délku 3:

`ins 1 [2,4,6,8]` \rightsquigarrow^3 `[1,2,4,6,8]`

- Nejdelší výpočet má délku $3n + 1$:

`ins 9 [2,4,6,8]` \rightsquigarrow^{3*4+1} `[2,4,6,8,9]`

Časová složitost

- Časová složitost funkce `ins` je **lineární** vzhledem k velikosti jejího druhého argumentu (tj. vzhledem k délce seznamu).

Pozorování

- Časová složitost závisí nejen na algoritmu (způsobu definování funkce), ale také na redukční strategii.

Příklad

- Uvažme funkci pro uspořádání prvků v seznamu pomocí postupného zařazování.

```
insort :: Ord a => [a] -> [a]
insort = foldr ins []
      where ins x [] = [x]
            ins x (y:t) = if x <= y then x : y : t
                          else y : ins x t
```

- Princip řazení funkcí `insort`

```
insort [x1, x2, ..., xn-1, xn]
  ~> foldr ins [] [x1, x2, ..., xn-1, xn]
  ~>n+1 ins x1 (ins x2 (... (ins xn-1 (ins xn []))...))
```


Striktní vyhodnocování

- Počet redukčních kroků výrazu $\text{minim } [x_1, \dots, x_n]$ je:

$$3 + n + 1 + \sum_{k=0}^{n-1} (3k + 1) + 1 = \frac{3n^2 + n + 10}{2}$$

- Při striktním vyhodnocování má funkce **kvadratickou** časovou složitost.

Líné vyhodnocování

- Počet redukčních kroků výrazu $\text{minim } [x_1, \dots, x_n]$ je:

$$3 + n + 1 + 1 + 3 \cdot (n - 1) + 1 = 4n + 3$$

- Při líném vyhodnocování má funkce **lineární** časovou složitost.

Pozorování

- Není pravda, že časová složitost výpočtu se při líném a striktním vyhodnocování vždy liší.
- Pokud se časová složitost liší, může se lišit víc než o jeden řádek ve zmiňované tabulce asymptotických růstů funkcí.

Příklady

- Konstantní (líně) versus exponenciální (striktně):

$$f\ n = \text{const } n \text{ (fib' } n)$$

- Lineární líně i striktně:

$$\text{length } [a_1, \dots, a_n]$$



Obecný princip

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

Pozorování

- Přímé použití rekurzivní funkce má tendenci být čitelnější.

Definice funkcí

```
fib' :: Integer -> Integer
```

```
fib' 0 = 0
```

```
fib' 1 = 1
```

```
fib' n = fib' (n-2) + fib' (n-1)
```

```
fib :: Integer -> Integer
```

```
fib = f 0 1
```

```
  where f a _ 0 = a
```

```
        f a b k = f b (a+b) (k-1)
```

Složitost vzhledem k argumentu

- Složitost funkce `fib'` je **exponenciální**.
- Složitost funkce `fib` je **lineární**.

Analyzujte časovou složitost

- funkce `map` , vzhledem k délce seznamu.
- funkce `es` (Eratosthenovo síto) vzhledem k délce seznamu.

Lineární vs. kvadratická

- Vytvořte libovolnou funkci `f1 :: a -> [a]` , která má kvadratickou složitost.
- Vytvořte libovolnou funkci `f2 :: a -> [a]` , která má lineární složitost.
- Použijte obě funkce v kontextu funkce `map` na dlouhých seznamech a pozorujte rozdíl.

IB015 Neimperativní programování

Práce se vstupy a výstupy

Jiří Barnat
Libor Škarvada

Úryvek diskuse mezi studenty

- ... *myslel jsem, že se budeme učit programovat ...*
- ... *obskurně zapisujeme funkce ...*
- ... *já si chci napsat svůj vlastní program v Haskellu ...*

Úryvek diskuse mezi studenty

- ... *myslel jsem, že se budeme učit programovat ...*
- ... *obskurně zapisujeme funkce ...*
- ... *já si chci napsat svůj vlastní program v Haskellu ...*

???

Úryvek diskuse mezi studenty

- ... *myslel jsem, že se budeme učit programovat ...*
- ... *obskurně zapisujeme funkce ...*
- ... *já si chci napsat svůj vlastní program v Haskellu ...*

???

Náplň dnešní lekce

- Jak realizovat komunikace mezi programem a jeho okolím.
- Napíšeme, přeložíme a spustíme první program v Haskellu.

Samostaně spustitelný program

- **Posloupnost akcí**, při kterých dochází k interakci programu s okolním světem.

Interakce s okolím programu

- Interakce s uživatelem skrze terminál.
- Manipulace se soubory, čtení a zápis do souboru.
- Interakce skrze grafické rozhraní.
- Komunikace s operačním systémem.
- ...

Práce s interpretem

- Doposud naše programy v Haskellu žádné akce nepřipouštěly.
- Žádný pokyn k výpisu výsledku jsme do programu nekládali.
- GHCi může za to, že po skončení výpočtu se vypíše výsledná hodnota na terminál.

Samostatný program

- Explicitní pokyny ke komunikaci s uživatelem.
- Při akci čtení vstupu z terminálu, se výpočet zastaví a čeká, až uživatel vloží text.
- Dochází k prokládání výpočtu, tak jak jsme jej znali doposud, a realizace vstup-výstupních akcí.

Vstup/výstup

Typování vstup-výstupních akcí

- Unární typový konstruktor **IO a** .
- Typy vytvořené pomocí typového konstruktoru `IO` mají jednu jedinou hodnotu, a to **vstup-výstupní akci**.
- Tato hodnota nemá textovou reprezentaci (nelze ji vypsat).

Výstupní akce

- Výstupní akce mají typ **IO ()** .
`putStrLn "Ahoj!" :: IO ()`

Vstupní akce

- Vstupní akce mají typ **IO a** , kde typová proměnná `a` nabývá hodnoty (typu) podle typu objektu, který vstupuje.
`getLine :: IO String`

Jak si představovat IO?

Analogie

- Vstup/výstupní akci se představme jako krabici.
- Hodnota akce je vždycky stejná ... **krabice**.
- Ovšem obsah krabice je vnitřní výsledek akce.
- Obsah je určen typem, avšak může mít různé hodnoty.



IO

7



Int

IO Int

Základní funkce pro výstup

`putChar :: Char -> IO ()`

- Zapiše znak na standardní výstup.

`putStr :: String -> IO ()`

- Zapiše řetězec na standardní výstup.

`putStrLn :: String -> IO ()`

- Zapiše řetězec na standardní výstup a přidá znak konec řádku.

`print :: Show a => a -> IO ()`

- Vypíše hodnotu jakéhokoliv tisknutelného typu na standardní výstup, a přidá znak konec řádku.
- Tisknutelné typy jsou instancí třídy `Show`.
- Uživatelem definované typy nutno označit přídomkem `deriving Show`.

Základní funkce pro vstup

`getChar :: IO Char`

- Načte znak ze standardního vstupu.

`getLine :: IO String`

- Načte řádek ze standardního vstupu.

`getContents :: IO String`

- Čte veškerý obsah ze standardního vstupu jako jeden řetězec. Obsah je čten líně, tj. až když je potřeba.

`interact :: (String -> String) -> IO ()`

- Argumentem funkce `interact` je funkce, která zpracovává řetězec a vrací řetězec.
- Veškerý obsah ze standardního vstupu je předán této funkci a výsledek vytištěn na standardní výstup.

Referenční transparentnost

- Daný výraz se vždy vyhodnotí na stejnou hodnotu, bez ohledu na okolí (kontext), ve kterém je použit.
- Programovací jazyk Haskell je referenčně transparentní.

Dopady na vstup-výstupní chování

- **Nelze napsat program, který by zpracoval vstup uživatele a vyhodnotil se podle zadaného vstupu na různé hodnoty.**
- Lze napsat program, který zpracuje vstup a podle vstupu vypíše na výstup různé výsledky.
- Hodnoty předávané skrze vstup-výstupní akce nesouvisí s hodnotou výrazu, který tuto vstup-výstupní akci realizuje.

Otázka

- Jestliže `getline` načte řetězec ze vstupu a přitom **má hodnotu vstup-výstupní akce**, což je hodnota typu `IO a`, konkrétně zde `IO String`, tak kde je ten načtený řetězec?

Otázka

- Jestliže `getline` načte řetězec ze vstupu a přitom **má hodnotu vstup-výstupní akce**, což je hodnota typu `IO a`, konkrétně zde `IO String`, tak kde je ten načtený řetězec?

Odpověď

- Načtený řetězec se uchová jako tzv. **vnitřní výsledek** provedení této vstupní akce.
- Skutečné načtení řetězce a zapamatování si vnitřního výsledku je realizováno jako **vedlejší efekt** vyhodnocení výrazu `getline`.

Přístup k hodnotě vnitřního výsledku

- Pomocí binárního operátoru `>>=` .
- Ve výrazu `f >>= g` funguje operátor `>>=` tak, že vezme vnitřní výsledek vstupní akce `f` a na tento aplikuje unární funkci `g` , **jejímž výsledkem je ovšem vstup-výstupní akce.**
- Výraz `f >>= g` tedy znamená, že:

`f :: IO a`

`g :: a -> IO b`

`f >>= g :: IO b`

Operátor >>=

- `(>>=) :: IO a -> (a -> IO b) -> IO b`
- Následující zápis je ekvivalentní:

`getLine >>= putStr`

`getLine >>= (\x -> putStr x)`

Krabicová analogie pro operátor $\gg=$

$(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

$f :: a \rightarrow IO\ b$



with
a-val

$\gg=$

f



with
b-val

Otázka

- Operátor (>>=) nelze použít ke spojení vstup-výstupní akce a funkce, která není vstup-výstupní akce, proč?
- Příklad, **takto nelze**:

```
getLine >>= length
```

```
getLine >>= (\ x -> length x)
```

Otázka

- Operátor (>>=) nelze použít ke spojení vstup-výstupní akce a funkce, která není vstup-výstupní akce, proč?
- Příklad, **takto nelze:**

```
getline >>= length
getline >>= (\ x -> length x)
```

Odpověď

- Hodnota výrazu je závislá na zadaném vstupu!
- Porušuje referenční transparentnost.
- Typově nesprávně. **„Jakmile jste v IO , nelze utéct“**.
- Správné použití:

```
getline >>= print . length
getline >>= (\ x -> print (length x))
```

Funkce `return` jako vstupní akce

- Prázdná akce, jejíž provedení má za cíl pouze naplnit hodnotu vnitřního výsledku.

```
return :: a -> IO a
```

```
return ['A','h','o','j'] :: IO String
```

- Možné použití:

```
return ['A','h','o','j'] >>= putStr
```

Funkce `return` jako výstupní akce

- Funkce `return`, může sloužit i jako výstupní akce, která nemá žádný efekt.

```
return :: a -> IO a
```

```
return () :: IO ()
```

Funkce `pure`

- Alternativa k funkci `return`.
- Použití funkce `return` je vzhledem ke kontextu známého z imperativních programovacích funkcí částečně matoucí.

Řazení akcí, operátor >>

- Binární operátor, který řadí vstup-výstupní akce.
- Zapomíná/ničí hodnotu vnitřního výsledku.
- Výraz má hodnotu poslední (druhé) vstup-výstupní akce.
- `(>>) :: IO a -> IO b -> IO b`

Jak se chovají následující programy?

- `putStr "Jeje" >> putChar '!'`
- `getLine >> putStr "nic"`
- `putStrLn "Napiš mi něco pěkného." >> getLine >>=`
`(\x -> putStr "Napsal jsi:" >> putStrLn x)`

Dávkové spuštění akcí

- Máme-li seznam vstup-výstupních akcí **stejného typu**, lze funkce z tohoto seznamu spustit a provést jako dávku.

- `sequence :: [IO a] -> IO [a]`
`sequence [] = return []`
`sequence (a:s) = do x<-a`
`t <- sequence s`
`return (x:t)`

Příklady použití

- V případě výstupních akcí je výsledkem vyhodnocení výrazu posloupnost výstupů, viz:

```
sequence [ putStr "Ahoj", putStr " ", putStr "světe!" ]
```

- V případě vstupních akcí, je výsledkem vyhodnocení výrazu seznam vstupů, který je uložený jako vnitřní výsledek vstup-výstupní akce, viz:

```
sequence [ getLine, getLine, getLine ] >>= print
```


Mapování IO akce na seznam argumentů

- Aplikuje unární vstup-výstupní akci na seznam hodnot a vzniklý **seznam vstup-výstupních akcí provede**.

```
mapM :: (a -> IO b) -> [a] -> IO [b]
```

```
mapM f = sequence . map f
```

Příklady použití

- `mapM putStr ["Den","Noc"]`
vypíše `DenNoc`
- `mapM (\t -> putStr "Aa") [1,2,3,4,5]`
vypíše `AaAaAaAaAa`
- `mapM (\x -> getLine) "aa" >=> print`
po zadání dvou řádků s obsahem `radek1` a `radek2`
vypíše `["radek1","radek2"]`

```
type FilePath = String
```

- Definuje typový alias.

```
readFile :: FilePath -> IO String
```

- Načte obsah souboru jako řetězec. Soubor je čten líně.

```
writeFile :: FilePath -> String -> IO ()
```

- Zapiše řetězec do daného souboru (existující obsah smaže).
- Hodnoty jiného typu než `String` lze konvertovat funkcí `show`.

```
appendFile :: FilePath -> String -> IO ()
```

- Připíše řetězec do daného souboru.
- Hodnoty jiného typu než `String` lze konvertovat funkcí `show`.

Pozorování

- Syntaktická konstrukce `do` slouží k alternativnímu zápisu výrazu s operátory `>>=` a `>>`.

Následující zápis je ekvivalentní

- ```
putStr "vstup?" >>
 getLine >>= \ x ->
 putStr "výstup?" >>
 getLine >>= \ y ->
 readFile x >>= \ z ->
 writeFile y (map toUpper z)
```

 | 

```
do putStr "vstup?"
 x <- getLine
 putStr "výstup?"
 y <- getLine
 z <- readFile x
 writeFile y (map toUpper z)
```

## Další předdefinované funkce

- Existuje řada dalších funkcí, jejich definice však nejsou automaticky zavedeny při startu programu, nebo interpretu.
- Tyto funkce jsou definovány v tzv. **modulech**.

## Například

- `System.IO`
- `System.Directory`
- `System.Time`
- ...

# Moduly

## Modul v Haskellu

- Kolekce spolu souvisejících funkcí, typů a typových tříd.
- Program v Haskellu je tvořen hlavním modulem `Main`, který následně importuje a používá funkce z jiných modulů.
- `Prelude` je implicitně zavedený modul, který obsahuje definice funkcí, které jsme doposud běžně používali.
- Moduly umožňují **strukturovat kód projektu**, oddělit a případně později znovupoužívat ucelené části kódu.

## Příklady předdefinovaných modulů

- Modul pro práci se seznamy – `Data.List`
- Modul pro práci se znaky – `Data.Char`
- Modul pro práci s komplexními čísly – `Data.Complex`
- ...

## Použití modulu

- Před použitím funkcí a typů z modulu je třeba explicitně požádat o zavedení tohoto modulu, tzv. **importovat modul**.
- Klíčové slova `import` .

## Možnosti importu

- Importovat pouze vybrané, nebo skrýt některé funkce modulu.
- Vynutit při použití funkce povinnou identifikaci modulu, v podobě `jmeno_modulu.jmeno_funkce` (tzv. kvalifikace).
- Přejmenovat kvalifikaci.

## Příklady

- `import Data.Char`
- `import qualified Data.Char`
- `import qualified Data.Char as C`
- `import qualified Data.Char as C (toUpper)`
- `import qualified Data.Char as C hiding (toUpper)`

## Pozorování

- Kvalifikace funkcí z modulů zavede nový význam znaku tečka.

## Příklad

- ```
import qualified Data.Char as C
import System.Directory
main = getDirectoryContents ".." >>=
      print . map (\x -> (C.toUpper.head) x : tail x)
```


Obecná definice

- `[module Jméno [(export1, ..., exportn)] where]`
`[import M1 [spec1]]`
`[⋮]`
`[import Mm [specm]]]`
`[globální_deklarace]`

Automatické doplnění definice Main

- `Není-li uvedena hlavička, doplní se`
`module Main (main) where`
- `Nevyskytuje-li se mezi importovanými moduly M1, ..., Mm`
`modul Prelude , doplní se`
`import Prelude`

Datový typ Fifo

- Datový kontejner (struktura, která uchovává prvky) přístupovaný operacemi **vlož prvek** a **vyber prvek**.
- Prvky jsou z datové struktury odebírány v tom pořadí, ve kterém byly vkládány.
- First-In-First-Out = FIFO
- Operace by měly mít konstantní časovou složitost.

Realizace v Haskellu

- Definice modulu `Fifo`
- Použití modulu:

```
import Fifo
```

Příklad Modulu – Datový typ Fifo

```
module Fifo (FifoTyp, emptyq, headq, enqueue, dequeue) where

data FifoTyp a = Q [a] [a]

emptyq :: FifoTyp a
emptyq = Q [] []

enqueue :: a -> FifoTyp a -> FifoTyp a
enqueue x (Q h t) = Q h (x:t)

headq :: FifoTyp a -> a
headq (Q (x:_) _) = x
headq (Q [] []) = error "headq: prázdná fronta"
headq (Q [] t) = headq (Q (reverse t) [])

dequeue :: FifoTyp a -> FifoTyp a
dequeue (Q (_:h) t) = Q h t
dequeue (Q [] []) = error "dequeue: prázdná fronta"
dequeue (Q [] t) = dequeue (Q (reverse t) [])
```

Samostatně stojící programy

Připomenutí

- Interpretace vs. kompilace kódu.
- Dosud jsme využívali interpret `ghci` .
- Kód lze přeložit do samostatně spustitelného programu.
- Překladač `ghc` .

Výhody přeloženého kódu

- Přeložený kód je rychlejší než interpretovaný.
- Na cílovou platformu není třeba instalovat interpret, ani vývojové prostředí.

Hlavní funkce – `main`

- Spustitelný program musí obsahovat funkci `main :: IO ()` .
- Funkce je automaticky spuštěna po zavedení programu do paměti.

Nazdar světe

- `main = putStrLn "Čaute lidi!"`

Příklad

- Kód umístěn v souboru `hello.hs` .
- Překlad pomocí `ghc hello.hs -o hello`
- Vytvořen spustitelný soubor `hello` .
- Vzniknou pomocné soubory `hello.hi` a `hello.o` , lze smazat.

Změna typu

- Vstup/Výstup je obvykle realizován v objektech typu `String`.
- Převést objekty na řetězec a zpět je možné pomocí funkcí:

```
show :: Show a => a -> String
```

```
read :: Read a => String -> a
```

Asymetrie použití

- Použití `show` a `read` není symetrické, při použití `read` je třeba uvést do jakého typu chceme řetězec transformovat.

```
show 123 ~>* "123"
```

```
read "123" ~>* ERROR
```

```
(read "123") :: Int ~>* 123
```

Parsování vstupu

- Převedou se pouze 100% správně strukturované řetězce.

```
(read "[1,2,3]") :: [Char] ~>* ERROR
```

```
(read "[1,2,3]") :: [Int] ~>* [1,2,3]
```

```
(read "[1,2,3]") :: [Float] ~>* [1.0,2.0,3.0]
```

Program plus1

- Tento program vyzve uživatele, aby zadal celé číslo, pak toto číslo přečte, převede na objekt typu `Int`, přičte jedničku a zvýšené číslo vytiskne.
- ```
main = do putStrLn "Napis cele cislo:"
 x <- getLine
 print (1 + read x::Int)
```



## Užitečné konstrukce Haskellu

## Pozorování

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

## Syntaktická konstrukce case

- ```
case expression of
  pattern1 -> expression1
  ...
  patternn -> expressionn
```
- Textové zarovnání vzorů je nutné.
- Všechny výrazy na pravých stranách musí být stejného typu.

Úkol 1

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

- Řešení:

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

Úkol 2

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

- Řešení:

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

Stráž

- Stráž je výraz typu Bool přidružený k definičnímu přiřazení.
- Při výpočtu bude tato definice funkce realizována, pouze pokud bude asociovaný výraz vyhodnocen na `True`.
- `function args`
 - | `guard1 = expression1`
 - ...
 - | `guardn = expressionn`
 - | `otherwise = default expression`

Pozorování

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

Příklad 1

- `f x | (x>3) = "vetsi nez 3"`
 `| (x>2) = "vetsi nez 2"`
 `| (x>1) = "vetsi nez 1"`
- `f 2 ~>* "vetsi nez 1"`
 `f 1 ~>* ERROR`

Příklad 2

- `g (a:x)`
 `| x==[] = "Almost empty."`
 `| x/=[] = "At least 2 members."`
 `| otherwise = "Unreachable code."`
 `g - = "Nothingness."`
- `g [] ~>* "Nothingness."`
 `g "Ahoj" ~>* "At least 2 members."`

Pozorování

- Při vypisování typů programů pracujících s IO můžete narazit na typovou třídu `Monad`, která je zobezněním typu `IO`.

- Například:

```
(>>) :: Monad m => m a -> m b -> m b
```

- Monadické typy jsou v pokročilém Haskellu běžné, ale jsou za hranicí, která je stanovena pro tuto přednášku.
- Pro účely této přednášky je v pořádku otypovat pomocí `IO`:

```
(>>) :: IO a -> IO b -> IO b
```

Wanna more?

- **IB016 Seminář z funkcionálního programování**

Vstup/výstup

- Napište program, který vyzve uživatele, aby zadal 16 čísel oddělených mezerou, a poté tato čísla vypíše v matici velikosti 4x4.
- Napište program, který ve vhodně formátovaném výstupu (např. rozbalený souborový strom) vypíše obsah aktuálního adresáře včetně všech jeho podúrovní.
- Vstup-výstupní programy vytvářejte jako samostatně spustitelné programy.



IB015 Neimperativní programování

Ukázky funkcionálně řešených problémů

Jiří Barnat
Libor Škarvada

Prohledávání a prořezávání

Problém n dam

Problém n dam

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

Všechna řešení pro $n = 4$

- | | | | |
|---|---|---|---|
| | ♔ | | |
| | | | ♔ |
| ♔ | | | |
| | | ♔ | |

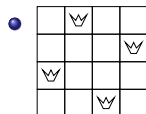
		♔	
♔			
			♔
	♔		

Pozorování

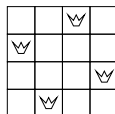
- V každém řádku/sloupku šachovnice může být nejvýše jedna dáma (jinak se ohrožují), a zároveň musí být alespoň jedna dáma, jinak bychom jich neumístili n .

Kódování pozice

- Očíslujeme 1 až n řádky šachovnici směrem shora dolů a sloupce šachovnice směrem zleva doprava.
- Řešení budeme kódovat seznamy čísel a to tak, že čísla v seznamu určují pozice dam v odpovídajících sloupcích.



[3, 1, 4, 2]



[2, 4, 1, 3]

Úkol

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

Postupné rozšiřování šachovnice

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

Cesta k celkovému řešení

- $damy :: Int \rightarrow [[Int]]$
 $damy\ n = da\ n\ n$

Účel

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

Princip

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

1	2	...	m-1	
				1
				2
				3
				4
				⋮
				n

Účel

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

Princip

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

0	1	2	...	m-1	
					1
					2
					3
					4
					⋮
					n

Účel

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

Princip

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

0	1	2	...	m-1	
					1
					2
					3
	♔				4
					⋮
					n

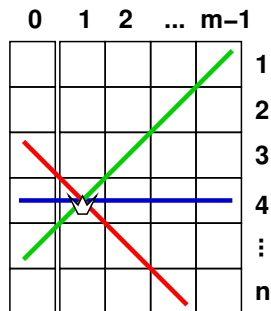
Účel

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

Princip

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

 $[4, \ , \ , \]$ $[5, \ , \ , \]$ $[3, \ , \ , \]$



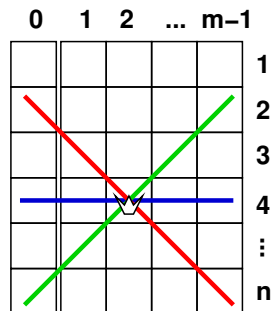
Účel

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

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- $[,4, ,] \quad [,4, ,] \quad [,4, ,]$
 $\quad \quad \quad +[1,2,3,4] \quad -[1,2,3,4]$

 $[,4, ,] \quad [,6, ,] \quad [,2, ,]$



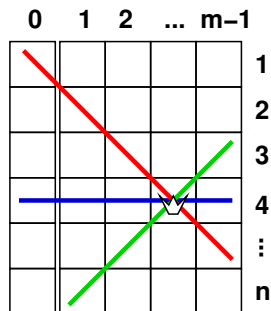
Účel

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

Princip

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

 $[, ,4,]$ $[, ,7,]$ $[, ,1,]$



Problém n dam – pomocná funkce

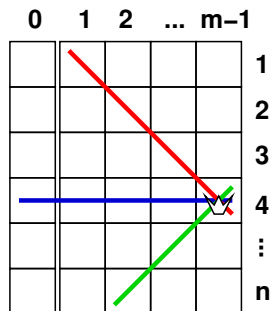
Účel

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

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- $[, , , 4] \quad [, , , 4] \quad [, , , 4]$
 $\quad \quad \quad +[1,2,3,4] \quad -[1,2,3,4]$

 $[, , , 4] \quad [, , , 8] \quad [, , , 0]$



Účel

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

Princip

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

	0	1	2	...	m-1	
			♔			1
						2
				♔		3
						4
					♔	⋮
	♔					n

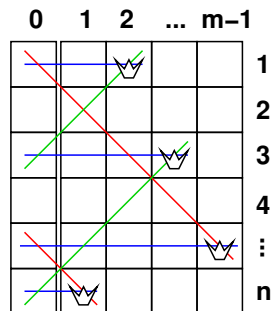
Účel

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

Princip

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

 $[6,1,3,5]$ $[-,3,6,-]$ $[5,-,-,1]$



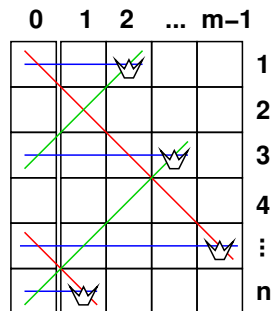
Účel

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

Princip

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

 $[6,1,3,5]$ $[-,3,6,-]$ $[5,-,-,1]$
- Platné volné pozice: 2 a 4.



```
type Reseni = [Int]

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

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

Backtracking, Prořezávání

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

Nejmenší nepoužité přirozené číslo

(Richard Bird: Pearls of Functional Algorithm Design)

Zadání

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

Řešení 1.

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

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

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

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

Funkce listminus

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

Funkce listminus

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

Funkce listminus

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

Funkce listminus

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

Funkce listminus

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

Funkce listminus

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

Funkce listminus

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

Funkce listminus

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

Funkce listminus

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

Funkce listminus

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

Funkce listminus

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

Funkce `listminus`

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

Vlastnosti řešení

- Délka výpočtu: $\text{length } v + \text{length } v-1 + \text{length } v-2 + \text{length } v-3 + \dots$
- V nejhorším případě kvadratická časová složitost.

Otázka

- Je možné problém řešit s lineární časovou složitostí?

Pozorování

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

Idea lepšího řešení

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

Pozorování

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

Idea lepšího řešení

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

Pozorování

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

Idea lepšího řešení

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

Pozorování

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

Idea lepšího řešení

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

Pozorování

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

Idea lepšího řešení

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

Pozorování

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

Idea lepšího řešení

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

Pozorování

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

Idea lepšího řešení

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

Předpoklady

- Konstantní operace pro přístup k hodnotám omezeně dlouhého seznamu.

Haskell efektivně, aneb od seznamů k polím

Pole

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

Poznámka:

- Přístup k n -tému prvku seznamu je možné realizovat funkcí
 - $(!!) :: [a] \rightarrow \text{Int} \rightarrow a$
 - $(!!) (x:s) 0 = x$
 - $(!!) (_ :s) n = (!!)\ s (n-1)$
- Časová složitost operace $(!!)$ ale není konstantní.

Použití

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

Typový konstruktor `Array`

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

Příklad hodnoty a typu

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

Typová třída `Ix`

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

## Meze pole

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

## Instance třídy `Ix`

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

## Příklad 1

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

## Příklad 2

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

## Přístup k prvkům pole

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

```
array (5,6) [(5,"ANO!"),(6,"NE!")] ! 5 ~>* "ANO!"
array ('D','E') [('D',2014),('E',1977)] ! 'D' ~>* 2014
```

## Aktualizace hodnot v poli

- Operace `//` zkopíruje původního pole a při kopírování aktualizuje uvedený seznam dvojic (index, hodnota).
- `(//) :: Ix i => Array i e -> [(i, e)] -> Array i e`
- Například:

```
array (1,2) [(1,1),(2,2)] // [(1,3),(2,3)]
~>* array (1,2) [(1,3),(2,3)]
```

## Meze pole

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

## Seznam indexů pole

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

## Převody mezi poli a seznamy

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

## Převody mezi poli a seznamy dvojic

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

## accumArray

- Knihovná funkce Haskellu.
- Vytváří pole ze seznamu dvojic (index, hodnota) a navíc akumuluje (funkcí foldl) prvky seznamu se stejným indexem.
- $\text{accumArray} :: \text{Ix } i \Rightarrow (\text{e} \rightarrow \text{a} \rightarrow \text{e}) \rightarrow \text{e} \rightarrow (i, i) \rightarrow [(i, \text{a})] \rightarrow \text{Array } i \text{ e}$
- Je-li akumulační funkce konstantní, pracuje v lineárním čase.

## Příklady

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

## Nejmenší nepoužité přirozené číslo II

(Richard Bird: Pearls of Functional Algorithm Design)

## Postup

- `minfree s ∈ ⟨0, length(s)⟩`  
`s = [1,2,6,2,0]`
- Nebudeme uvažovat čísla mimo očekávaný rozsah.  
`(filter (<=n) s) where n = length s ~>* [1,2,2,0]`
- Vytvoříme asociační seznam (příprava na konverzi do pole).  
`t s = (zip (filter (<=n) s) (repeat True))`  
`where n = length s`  
`t s ~>* [(1,True),(2,True),(2,True),(0,True)]`
- Vytvoříme pole a odstraníme duplicity tak, aby nepoužité indexy ukazovaly na `False`.  
`checklist :: [Int] -> Array Int Bool`  
`checklist s = accumArray (||) False (0,length s) (t s)`

## Mezivýsledek

- `checklist s ~>* array (0,5) [(0,True),(1,True),(2,True),(3,False),(4,False),(5,False)]`



## Postup pokračování

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

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

- Nejprve převedeme pole na seznam hodnot typu `Bool`

```
elems (checklist s)
```

```
↪* [True,True,True,False,False,False]
```

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

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

```
↪* [True,True,True]
```

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

```
search = length . takeWhile id . elems
```

```
search (checklist [1,2,6,2,0]) ↪* 3
```

## Řešení

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

## Složitost algoritmu

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

## Lineární řazení některých seznamů

(Richard Bird: Pearls of Functional Algorithm Design)

## Pozorování

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

## Algoritmus

- `max = 1000`

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

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

## Příklad

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



# IB015 Neimperativní programování

## Neimperativní programování v Prologu (relace místo funkcí)

Jiří Barnat

## Pozorování

- Podstata neimperativního programování je v tom, že popisuje (deklaruje) vztahy mezi objekty světa. Počítač se používá k tomu, aby na základě popsáných vztahů mechanicky vypočítal výsledek (dedukoval důsledek vztahů).

## Haskell

- K popisu vztahů mezi objekty se používají **funkce**.
- Funkce má pro dané vstupy nejvýše jeden výstup.

## Zobecnění na relace

- Výpočetní paradigma, kde se pro popis vztahů mezi objekty použijí relace, tj. k daným vstupům existuje více výstupů.

## Logické výpočetní paradigma

- program = databáze faktů a pravidel + cíl.
- výpočet = dokazování cíle metodou SLD rezoluce.
- výsledek = pravda/nepravda a hodnoty volných proměnných, pro které je cíl dokazatelný.
- **Prolog** – jediný programovací jazyk.

## SWI-Prolog

- Relativně úplná implementace Prologu.
- Freeware.
- <http://swi-prolog.org>





## Úvodní intuitivní příklady

## SWI-Prolog

- Interpret spuštěn příkazem `swipl`.

## Textové interaktivní prostředí

- Standardní výzva: `?-`
- Veškeré povely uživatele musí být zakončeny tečkou.

## Základní povely

- Ukončení prostředí: `halt.` (Ctrl+D)
- Návoděda: `help.`
- Načtení souboru `jmeno.pl`: `consult(jmeno).`
- Též alternativně:  
`[jmeno].`  
`consult('jmeno.pl').`

## Struktura jednoduchých příkladů

- Databáze fakt a pravidel uvedena v externím souboru.
- Cíle zadávány skrze interpret.
- Přípona souboru: `.pl`

## Notace fakt

- Fakta začínají malým písmenem a končí tečkou.
- Fakta jsou konstanty (nulární relace) a n-ární relace.
- Počet parametrů udáván u jména za lomítkem: `jmeno/N`.

## Příklady

- `tohleJeFakt.`
- `tohleTaky(parametr1,parametr2,...,parametrN).`
- `fakt /* a /* zanoreny */ komentar */ .`

## Databáze fakt

- `je_teplo.`  
`neprsi.`  
`kamaradi(vincenc,kvido). /* Znají se od mateřské školy. */`  
`kamaradi(vincenc,ferenc). /* Poznali se na pískovišti. */`

## Dotazy na databázi

- `?- je_teplo.`  
`true.`
- `?- prsi.`  
`ERROR: Undefined prsi/0.`
- `?- kamaradi(vincenc,kvido).`  
`true.`
- `?- kamaradi(ferenc,vincenc).`  
`false.`

## Proměnné

- Jména začínají velkým písmenem nebo podtržítkem.
- Je možné je (mimojiné) využít v dotazech.
- Interpreter se pokusí najít vyhovující přiřazení.

## Dotazy s využitím proměnných

- `?- kamaradi(vincenc,X).`  
  `X = kvido ;`  
  `X = ferenc.`
- `?- kamaradi(X,Y).`  
  `X = vincenc,`  
  `Y = kvido ;`  
  `X = vincenc,`  
  `Y = ferenc.`

## Odpověď interpretru zakončená tečkou

- Indikuje, že nejsou další možnosti.

## Odpověď interpretru nezakončená tečkou

- Výzva pro uživatele, zda chce hledání možných řešení ukončit (uživatel vloží tečku), nebo zda si přeje, aby bylo hledáno další řešení (uživatel vloží středník).

## Porovnejte

- `?- kamaradi(vincenc,X).`  
`X = kvido ; /* uživatel vložil středník */`  
`X = ferenc.`
- `?- kamaradi(vincenc,X).`  
`X = kvido . /* uživatel vložil tečku */`

## Pravidla v databázi

- Zápis:  $\text{clovek}(X) :- \text{zena}(X).$
- Význam: Pokud platí  $\text{zena}(X)$ , pak platí  $\text{clovek}(X)$ .

## Disjunkce

- Zápis:  $\text{clovek}(X) :- \text{zena}(X); \text{muz}(X).$
- Alternativní zápis:  
 $\text{clovek}(X) :- \text{zena}(X).$   
 $\text{clovek}(X) :- \text{muz}(X).$
- Význam:  $(\text{zena}(X) \vee \text{muz}(X)) \implies \text{clovek}(X).$

## Konjunkce

- Zápis:  $\text{nevyhranen}(X) :- \text{zena}(X), \text{muz}(X).$
- Význam:  $(\text{zena}(X) \wedge \text{muz}(X)) \implies \text{nevyhranen}(X).$

## Databáze:

- `clovek(X) :- zena(X).`  
`zena(bozena_nemcova).`

## Příklady dotazů

- `?-zena(bozena_nemcova).`  
`true.`
- `?-clovek(bozena_nemcova).`  
`true.`
- `?-zena(jirik).`  
`false.`
- `?- clovek(X).`  
`X = bozena_nemcova.`



## Rozsah platnosti proměnných

- Použití proměnné je lokalizováno na dané pravidlo.

## Příklad

- ```
clovek(X) :- zena(X); muz(X).  
nevyhranen(X) :- zena(X), muz(X).  
zena(bozena_nemcova).  
zena(jara_cimrman).  
muz(jara_cimrman).
```
- | | |
|---|--|
| <pre>?- clovek(X).
X = bozena_nemcova ;
X = jara_cimrman;
X = jara_cimrman.</pre> | <pre>?- nevyhranen(X).
X = jara_cimrman.</pre> |
|---|--|

Termy – Základní stavební kameny

Pozorování

- Fakta, pravidla a dotazy jsou tvořeny z termů.

Termy

- Atomy.
- Čísla.
- Proměnné.
- Strukturované termy.

Proměnné

- Proměnné se zapisují s velkým počátečním písmenem.
- Hodnotou uloženou v proměnné může být libovolný term.
- **Prolog není striktně typovaný.**

Atomy

- Řetězce začínající malým písmenem, obsahující písmena číslice a znak podtržítka.
- Libovolné řetězce uzavřené v jednoduchých uvozovkách.

Příklady:

- Atomy: `pepa`, `'pepa'`, `'Pepa'`, `'2'`.
- Neatomy: `Pepa`, `2`, `ja a on`, `holmes&watson`.

Test na bytí atomem

- `atom/1` – Pravda, pokud parametr je nestrukturovaným atomem.

Čísła

- Celá i desetinná čísla, používá se desetinná tečka.

```
?- A is 2.5 * 1.3.
```

```
A = 3.25.
```

- Porovnání s aritmetickým vyhodnocením pomocí `==`.

```
?- 4 == 3+1.
```

```
true.
```

```
?- 4 == 3+1.
```

```
false.
```

```
?- 4 = 3+1.
```

```
false.
```

- Aritmetické vyhodnocení a přiřazení pomocí `is`.

```
?- A is 2*3.
```

```
A = 6.
```

```
?- A == 2*3.
```

```
false.
```

```
?- A = 2*3.
```

```
A = 2*3.
```

Testy na bytí číslem

- `number/1` – Pravda, pokud je parametr číslo.
- `float/1` – Pravda, pokud je parametr desetinné číslo.
- `=\=/2` – Aritmetická neekvivalence.

Strukturované termy

- Funktor (název relace) následovaný sekvencí argumentů.
- Pro funktor platí stejná syntaktická omezení jako pro atomy.
- Argumenty se uvádějí v závorkách, oddělené čárkou.
- Mezi funktorem a seznamem argumentů nesmí být mezera.
- Argumentem může být libovolný term.
- Rekurze je možná.

Arita

- Počet argumentů strukturovaného termu.
- Identifikace strukturovaného termu: `funktor/N`.
- Stejný funktor s jinou aritou označuje jiný term.
- Je možné současně definovat termy `term/2` i `term/3`.

Unifikace v Prologu

Definice

- Dva termy jsou unifikovatelné, pokud jsou identické, anebo je možné zvolit hodnoty proměnných použitých v unifikovaných termech tak, aby po dosazení těchto hodnot byly termy identické.

Operátor `=/2`

- Realizuje unifikaci v Prologu.
- Lze zapisovat infixově.
- Binární operátor ne-unifikace: `\=`, tj. ve standardní notaci `\=/2`.

Výsledek unifikace

- Ano/Ne
- Unifikační přiřazení do proměnných (substituce).

Bez unifikačního přiřazení

- $?- \text{=(slovo,slovo)}.$
 true.

S unifikačním přiřazením

- $?- \text{=(slovo,X)}.$
 $X = \text{slovo.}$
- $?- \text{a(A,[ble,ble]) = a(b(c(d)),B)}.$
 $A = \text{b(c(d))},$
 $B = [\text{ble, ble}].$

- 1) Pokud jsou `term1` a `term2` konstanty (atomy, čísla), pak se unifikují, jestliže jsou tyto termy shodné.
- 2) Pokud je `term1` proměnná a `term2` je libovolný term, pak se unifikují a proměnná `term1` je instanciována hodnotou `term2`. Podobně, pokud je `term2` proměnná a `term1` je libovolný term, pak se unifikují a proměnná `term2` je instanciována hodnotou `term1`.
- 3) Pokud jsou `term1` a `term2` strukturované termy tak se unifikují, pouze pokud mají stejný funktor a aritu, všechny korespondující páry argumentů se unifikují, a všechny instanciaci proměnných z vnořených unifikací jsou kompatibilní.
- 4) Nic jiného.

Příklady unifikace

- `?- snida(karel,livance) = snida(Kdo,Co).`
`Kdo = karel,`
`Co = livance.`
- `?- snida(karel,Jidlo) = snida(Osoba,marmelada).`
`Jidlo = marmelada,`
`Osoba = karel.`
- `?- cdcko(29,beatles,yellow_submarine) = cdcko(A,B,help).`
`false.`
- `?- fce(X,val) = fce(val,X).`
`X = val.`
- `?- partneri(eva,X) = partneri(X,vasek).`
`false.`
- `?- fce(X,Y) = fce(Z,Z).`
`X = Y, Y = Z.`

Příklad

- Uvažme následující dotaz na Prolog:
 $?- X = \text{otec}(X).$
- Jsou unifikovatelné termy, kde jeden term je proměnná a přitom je vlastním podvýrazem druhého termu?

Nekorektnost algoritmu

- Podle definice ne, neboť neexistuje hodnota této proměnné taková, aby po dosazení nastala identita termů.
- Dle algoritmu na předchozím slajdu, však k unifikaci dojde:
 $?- X = \text{otec}(X).$
 $X = \text{otec}(X).$

Poznámka

- Některé implementace Prologu mohou při této unifikaci cyklit.

Kontrola sebevýskytu

- Algoritmus je možné modifikovat tak, aby dával korektní odpověď. Pokud se samostatná proměnná vyskytuje jako podvýraz v druhém termu, termy se neunifikují.
- V praxi se často jedná o nadbytečný test, unifikace je velice častá operace, z důvodu výkonnosti se tento test vynechává.

`unify_with_occurs_check/2`

- Specifický operátor unifikace s testem na sebevýskyt.
- `?- unify_with_occurs_check(X,otec(X)).`
`false.`

Pozorování

- Unifikace je jeden z fundamentů logického programování.
- Pomocí unifikace můžeme odvozovat i sémantickou informaci.

Příklad

- ```
vertical(line(point(X,Y),point(X,Z))).
horizontal(line(point(X,Y),point(Z,Y))).

?- horizontal(line(point(2,3),point(12,3))).
true.

?- vertical(line(point(1,1),X)).
X = point(1, _G2240).
```

## Jak Prolog počítá

## Teorie

- Výpočet = dokazování.
- Kódování problému pomocí Hornových klauzulí.
- Dokazování Selektivní Lineární Definitní rezolucí.
- Při výpočtu Prologu se konstruuje a prochází SLD strom.

## V rámci IB015

- Princip výpočtu s využitím příkladů a neformální demonstrace postupu bez intelektuální zátěže odpovídajícího teoretického fundamentu.

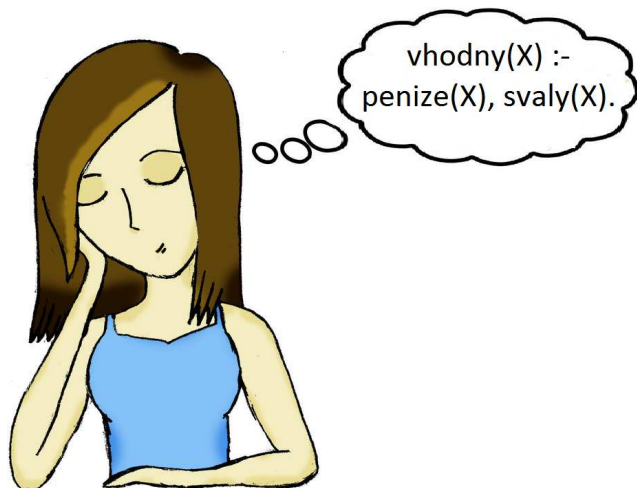


## Pozorování

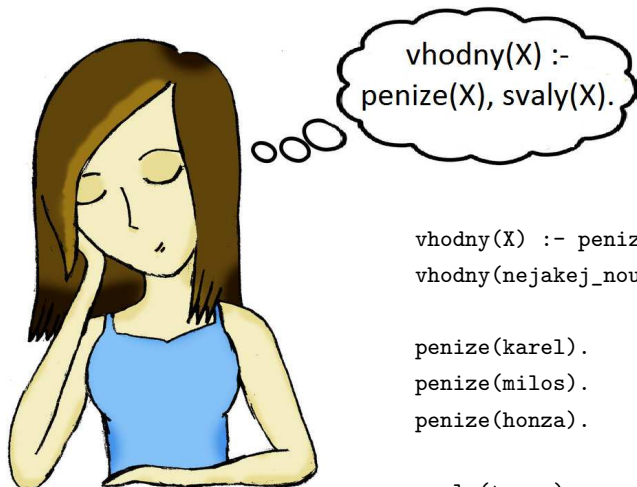
- Při výpočtu Prolog vždy využívá fakta v tom pořadí, v jakém jsou uvedeny v programu.

## Příklad

- `players(flink,diablo_III).`  
`players(flink,world_of_tanks).`  
`players(flink,master_of_orion).`
- `?- players(flink,X).`  
`X = diablo_III; /* uživatel vložil ; */`  
`X = world_of_tanks ; /* uživatel vložil ; */`  
`X = master_of_orion.`
- `?- players(flink,X).`  
`X = diablo_III . /* uživatel vložil . */`



# Příběh slečny Prology, aneb jak s pravidly



```
vhodny(X) :- penize(X), svaly(X).
vhodny(nejakej_nouma).
```

```
penize(karel).
penize(milos).
penize(honza).
```

```
svaly(tomas).
svaly(honza).
svaly(karel).
```

# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

?- v(X)

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

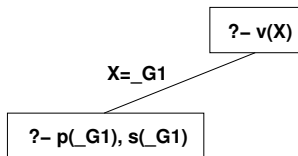
```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

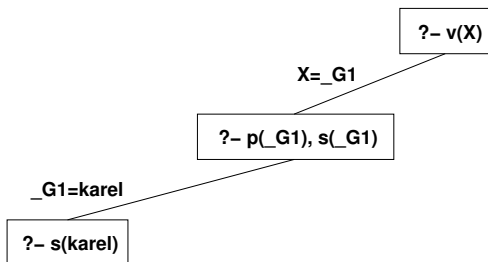
```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

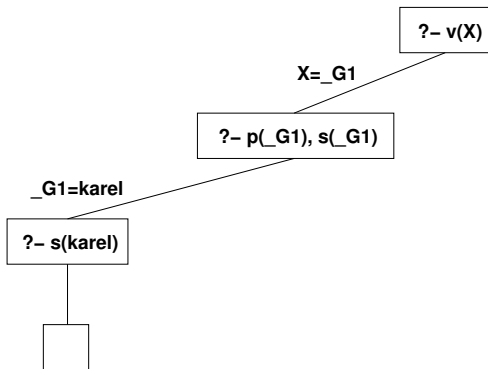
```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

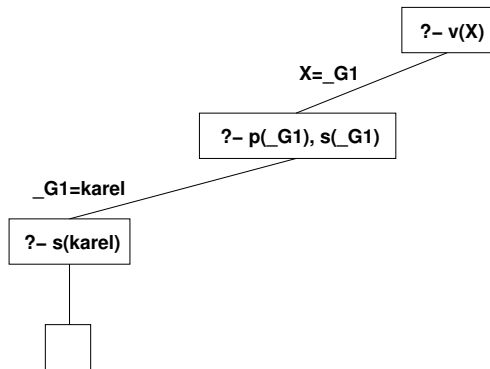
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel
```





# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

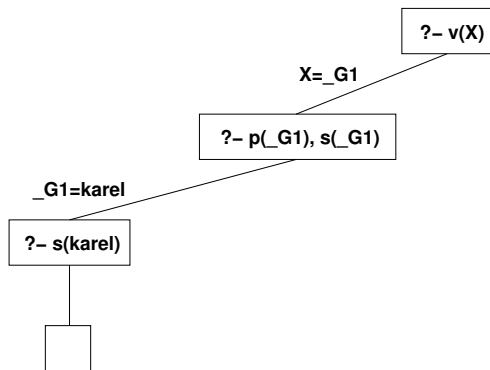
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

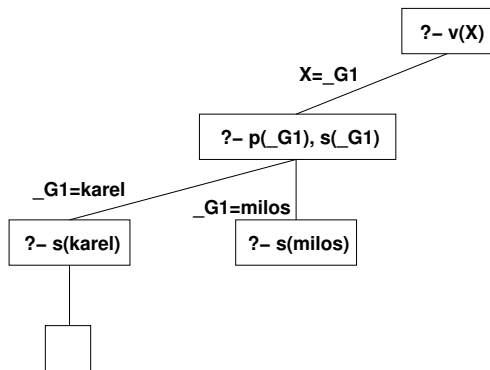
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

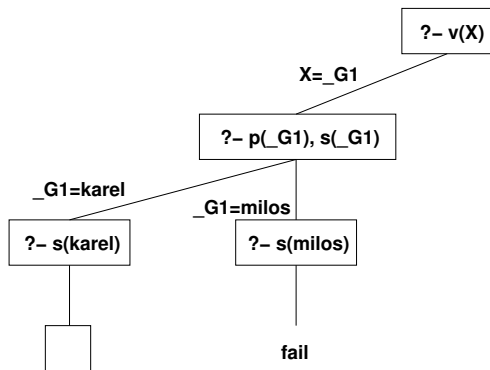
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

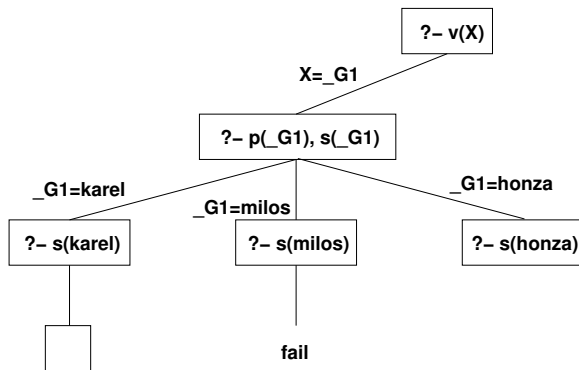
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

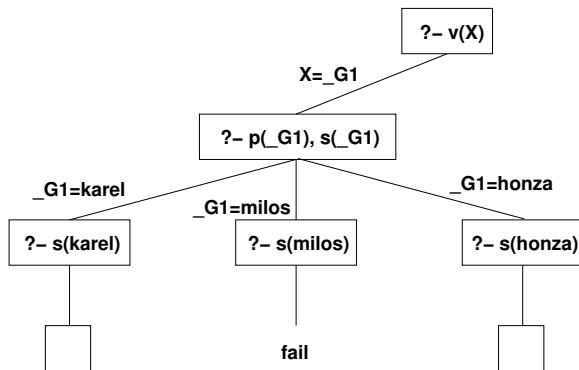
```
s(tomas).
```

```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

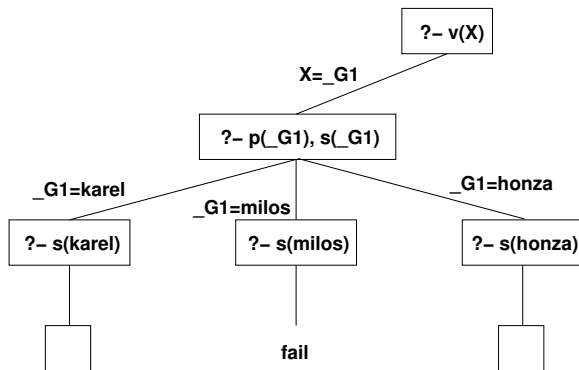
```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```

```
honza
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

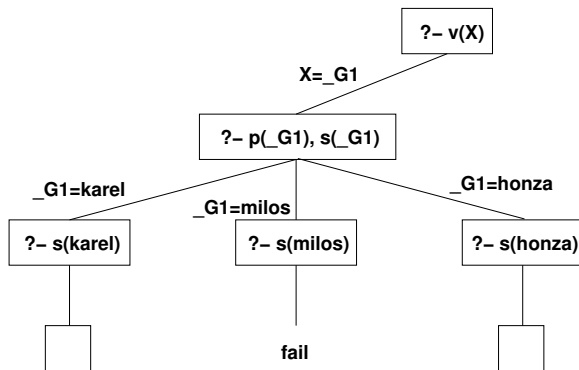
```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```

```
honza ; /* uživatel vložil ; */
```



# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

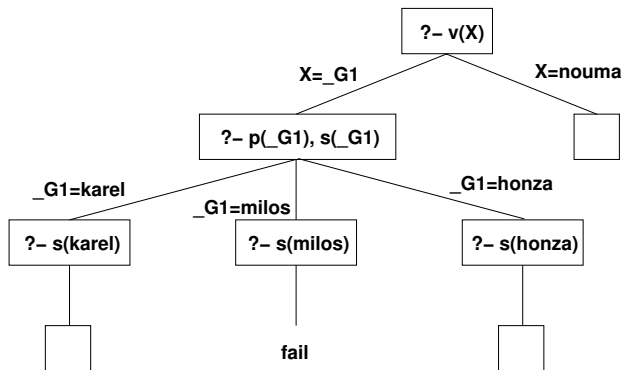
```
s(honza).
```

```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```

```
honza ; /* uživatel vložil ; */
```





# Příběh slečny Prology

```
v(X) :- p(X), s(X).
```

```
v(nouma).
```

```
p(karel).
```

```
p(milos).
```

```
p(honza).
```

```
s(tomas).
```

```
s(honza).
```

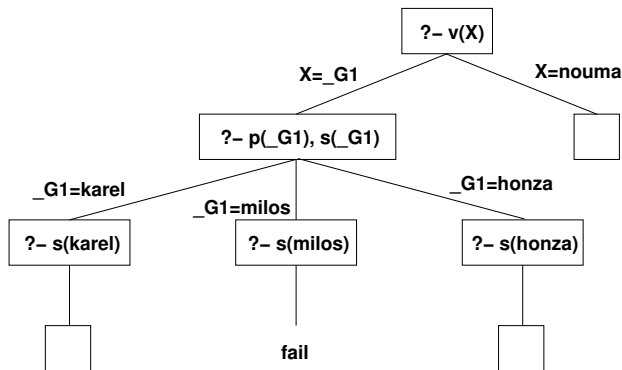
```
s(karel).
```

```
?-v(X).
```

```
karel ; /* uživatel vložil ; */
```

```
honza ; /* uživatel vložil ; */
```

```
nouma.
```



# Konstrukce výpočetního stromu

- Pro první podcíl v daném vrcholu se prohledává databáze faktů a pravidel vždy od začátku. Pro každou nalezenou vyhovující položku je vytvořen nový vrchol.
- Vrchol je vytvořen tak, že první podcíl se unifikuje s hlavou nalezené položky, a v nově vzniklém vrcholu je nahrazen tělem nalezené položky (fakta mají prázdné tělo, cíl je "vynechán").
- Hrana vedoucí do nového vrcholu je anotována unifikačním přiřazením. Proměnné vyskytující se v těle pravidla, které nejsou dotčeny unifikací, jsou označeny čerstvou proměnnou.
- Prázdné vrcholy (listy) značí úspěch, hodnoty hledaných proměnných vyplývají z unifikačních přiřazení na cestě od listu ke kořeni stromu.
- Vrcholy, pro které se nepodařilo nalézt položku v databázi vyhovující prvnímu podcílu jsou označeny podvrcholem **fail**.

?- f(G1)

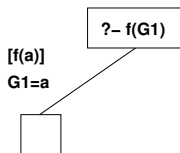
r(a,b).

r(a,c).

f(a).

f(X):-f(Y),r(Y,X).

# Výpočet s rekurzí



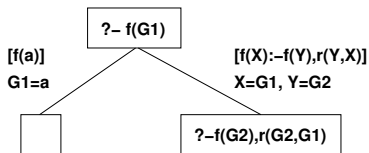
$r(a, b).$

$r(a, c).$

$f(a).$

$f(X) :- f(Y), r(Y, X).$

# Výpočet s rekurzí



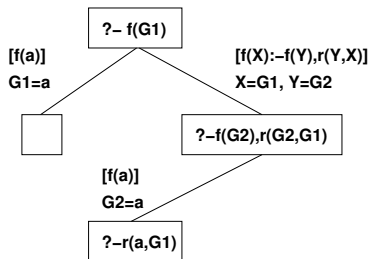
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



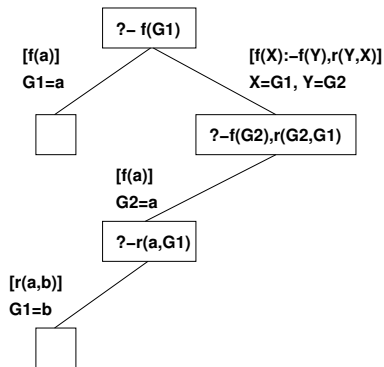
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



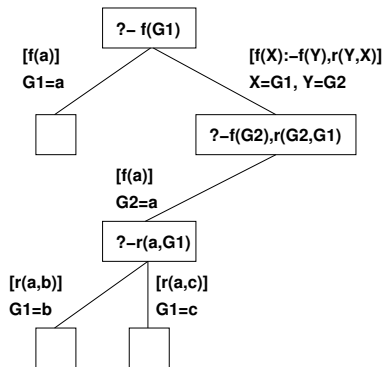
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



$r(a, b) .$

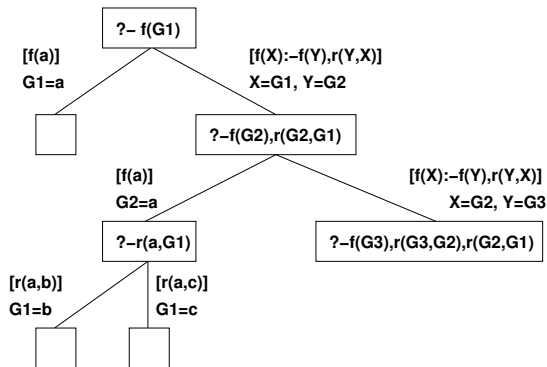
$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$



# Výpočet s rekurzí



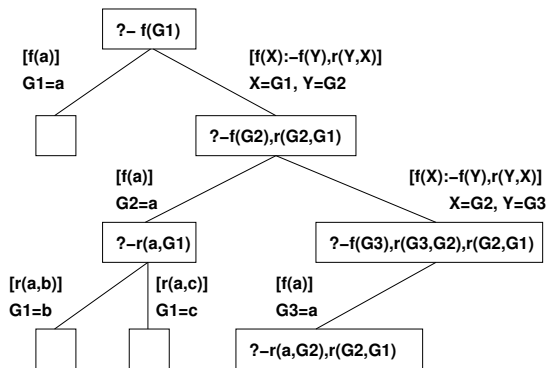
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



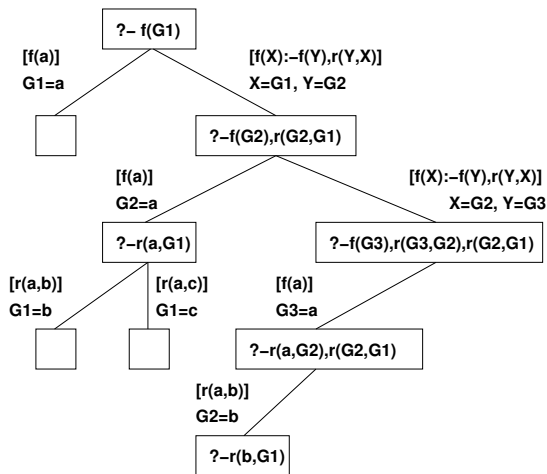
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



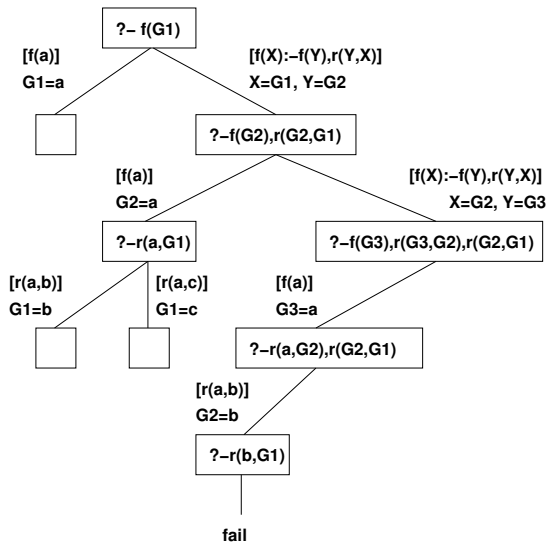
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



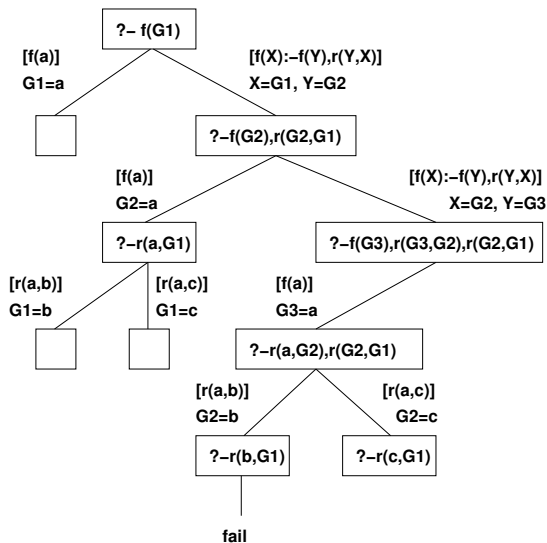
$r(a, b)$ .

$r(a, c)$ .

$f(a)$ .

$f(X) :- f(Y), r(Y, X)$ .

# Výpočet s rekurzí



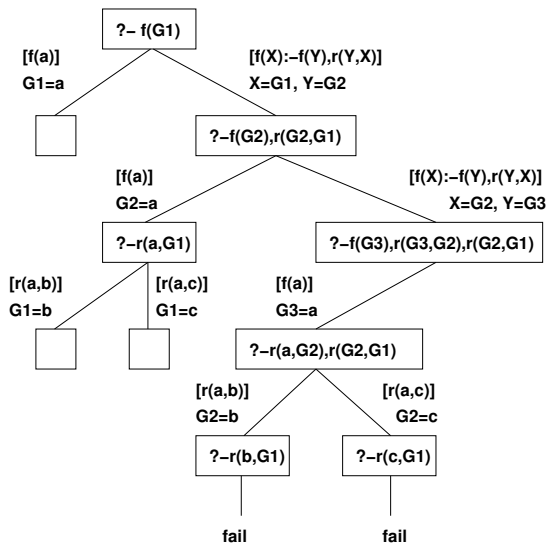
$r(a, b)$ .

$r(a, c)$ .

$f(a)$ .

$f(X) :- f(Y), r(Y, X)$ .

# Výpočet s rekurzí



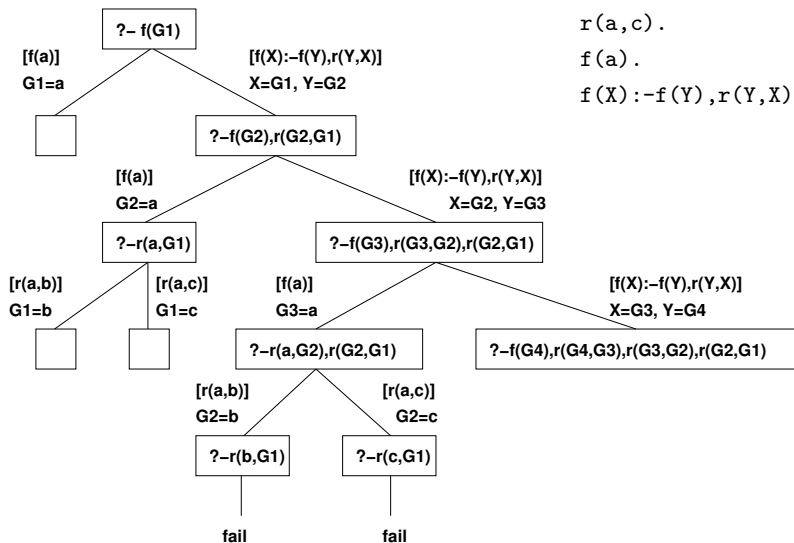
$r(a, b)$ .

$r(a, c)$ .

$f(a)$ .

$f(X) :- f(Y), r(Y, X)$ .

# Výpočet s rekurzí



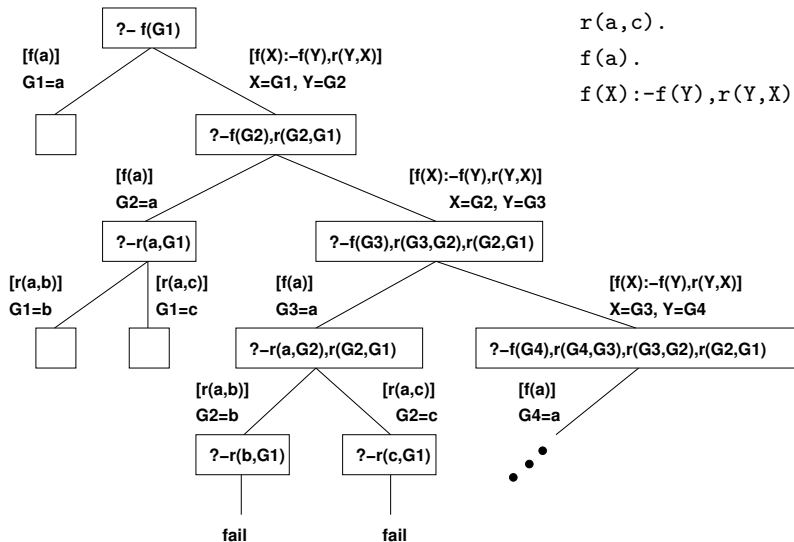
$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$

# Výpočet s rekurzí



$r(a, b) .$

$r(a, c) .$

$f(a) .$

$f(X) :- f(Y), r(Y, X) .$





## Pozorování

- Strom je procházen algoritmem prohledávání do hloubky.
- Výpočet nad nekonečnou větví stromu prakticky končí chybovou hláškou o nedostatečné velikosti paměti zásobníku.
- Použití levorekurzivních pravidel (první podcíl v těla pravidla je rekurzivní použití hlavy pravidla) často vede k nekonečné větvi, a to i v případě, kdy počet vyhovujících přiřazení na původní dotaz je konečný.
- Pořadí faktů a pravidel v databázi neovlivní počet úspěšných listů ve výpočetním stromu, ale ovlivní jejich umístění (tj. pořadí, ve kterém budou nalezeny a prezentovány uživateli.)

## Doporučení

- Používají se pravorekurzivní definice pravidel.
- Fakta se uvádějí před pravidly.

## Příklad

- S využitím znalostí prezentovaných v této přednášce naprogramujte nad databází měst:

```
mesto(prague).
```

```
mesto(viena).
```

```
mesto(warsaw).
```

```
mesto(roma).
```

```
mesto(paris).
```

```
mesto(madrid).
```

predikát `triMesta/3`, který je pravdivý pokud jako argumenty dostane tři různá města uvedené v databázi.

## Nekonečný výpočet

- Narhňte program v jazyce Prolog takový, aby ze zadání bylo zřejmé, že odpověď na určený dotaz je pravdivá, ovšem výpočet Prologu k tomuto výsledků nikdy nedospěje.

# IB015 Neimperativní programování

## Seznamy, Aritmetika a Efektivita výpočtu v Prologu

Jiří Barnat

## Výpočty nad seznamy v Prologu

## Seznam v Prologu

- Sekvence prvků (mohou být typově různorodé).
- Pro dekompozici na hlavu a tělo se používá znak |.
- Hlavu lze zobecnit na neprázdnou konečnou sekvenci prvků.

## Příklady

- $?- [H|T] = [\text{marek}, \text{matous}, \text{lukas}, \text{jan}] .$

H = marek,

T = [matous, lukas, jan].

- $?- [X,Y|Z] = [1,2,3,4] .$

X = 1,

Y = 2,

Z = [3, 4].

- $?- [X|Y|Z] = [1,2,3,4] .$

**ERROR**

## Anonymní proměnná

- Označená znakem podtržítka.
- Nelze se na ni odkázat v jiném místě programu.
- Při použití v unifikačním algoritmu neklade žádné omezení na kompatibilitu přiřazení hodnot jednotlivým proměnným.

## Příklady unifikace s anonymní proměnnou

- $?- f(a,X)=f(X,b).$       $?- f(a,X)=f(\_,b).$       $?- f(a,\_)=f(\_,b).$   
false.                       $X = b.$                       true.
- Unifikací získejte 2. a 4. prvek seznamu Seznam:  
 $[\_,X,\_,Y|\_] = \text{Seznam}.$

## Pozorování

- Při vytváření programu v prologu, který má něco spočítat nebo vytvořit, postupujeme dle obecného pravidla transformace funkce  $f(A) = B$  na predikát  $r(A,B)$ .

## Zadání

- Definujte predikát  $a2b/2$ , který transformuje seznam termů  $a$  na stejně dlouhý seznam termů  $b$ .



## Řešení

- $a2b([], []).$   
 $a2b([a|Ta], [b|Tb]) :- a2b(Ta, Tb).$

## Použití

- $?- a2b([a, a, a, a], X).$                        $?- a2b(X, [b, b, b, b]).$   
 $X = [b, b, b, b].$                                $X = [a, a, a, a].$
- $?- a2b(X, Y).$   
 $X = Y, Y = [] ;$   
 $X = [a], Y = [b] ;$   
 $X = [a, a], Y = [b, b] ;$   
 $X = [a, a, a], Y = [b, b, b].$

## length/2

- `length(L,I)` je pravda pokud délka seznamu `L` má hodnotu `I`.
- Prázdný seznam má délku 0.
- `?- length([a,ab,abc],X).`  
`X = 3.`
- `?- length([[1,2,3]],X).`  
`X = 1.`
- `?- length(X,2).`  
`X = [_G907, _G910].`

## Test na bytí seznamem

- `is_list/1` – Pravda, pokud parametr je seznam.
- `?- is_list( ['aha',[2,'b']], [],2.3 ).`  
`true.`

## member/2

- Zjištění přítomnosti prvku v seznamu.
- `member(X,L)` je pravda, pokud objekt `X` je prvkem seznamu `L`.
- Implementace:

```
member(X, [X|_]).
```

```
member(X, [_|T]) :- member(X,T).
```

## Postup výpočtu:

- `?- member(lukas, [marek, matous, lukas, jan]).`

↪

```
?- member(lukas, [matous, lukas, jan]).
```

↪

```
?- member(lukas, [lukas, jan]).
```

```
true.
```

## Pozorování

- Volnou proměnnou lze použít jak v prvním, tak i v druhém argumentu. Pomocí predikátu je možné enumerovat prvky seznamu, ale také vynutit, že daný prvek je prvkem seznamu.

## Příklady

- `?- member(X, [marek, matous, lukas, jan]).`  
`X = marek ;`  
`X = matous ;`  
`X = lukas ;`  
`X = jan ;`  
`false.`
- `?- member(jan, [marek, matous, lukas, X]).`  
`X = jan ;`  
`false.`

## append/3

- Dotaz `append(L1,L2,L3)` se vyhodnotí na pravda, pokud seznam `L3` je zřetěžením seznamů `L1` a `L2`.

## Definice append

- `append([],L,L)` .  
`append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3)` .

## Použití

- Test na to, zda  $L1 \cdot L2 = L3$ .
- Test na rovnost seznamů.
- Výpočet zřetěžení dvou seznamů.
- Odvození prefixu, nebo sufixu v daném seznamu.
- Generování seznamů, jejichž zřetěžení má daný výsledek.

## Definice append v Prologu

- `append([],L,L).`  
`append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).`

## Definice append v Haskellu

- `append [] l = l`  
`append (h1:t1) l2 = h1:t3 where t3 = (append t1 l2)`

## Pozorování

- 'Ahoj' není totéž co "Ahoj" .
- Řetězce znaků uvedených v uvozovkách jsou chápány jako seznamy čísel, které odpovídají ASCII kódům jednotlivých znaků řetězce.
- "Ahoj" == [65,104,111,106].

## Automatická konverze na seznam čísel

- ?- append("Ale ", "ne!", X).  
X = [65, 108, 101, 32, 110, 101, 33].

## Konverze na řetězce

- Vybrané předdefinované predikáty vynutí prezentaci ve formě "řetězce".

## Syntaktická rovnost

- `?- 'pepa' == "Pepa".`  
`false.`

`?- 'mouka' == 'mouka'.`  
`true.`

`?- 4 == 3+1.`  
`false.`

`?- 3+1 == 3+1.`  
`true.`

## Pozor na syntaktické ekvivalenty

- `?- 'pepa' == pepa.`  
`true.`

`?- [97] == "a".`  
`true.`



# Aritmetika

## Celá čísla - integer

- Nativní typ, využívá knihovnu GMP.
- Velikost čísel omezena pouze velikostí dostupné paměti.

## Desetinná čísla - float

- Nativní typ, odpovídá typu `double` z programovacího jazyka C.

## Racionální čísla - rational

- Reprezentované s využitím složeného termu `rdiv(N,M)`.
- Výsledek vrácený operátorem `is/2` je kanonizován, tj. M je kladné a M a N nemají společného dělitele.

## Konverze a unifikace

- `rdiv(N,1)` se konvertuje na celé číslo N.
- Automatické konverze ve směru od celých čísel k desetinným.
- Riziko vyvolání výjimky přetečení.
- Čísla různých typů se **neunifikují**.

## Relační operátory

|       |                  |
|-------|------------------|
| </2   | menší než        |
| >/2   | větší než        |
| =</2  | menší nebo rovno |
| >=/2  | větší nebo rovno |
| ==/2  | rovno            |
| =\=/2 | nerovno          |

## Bitové operace

|       |                     |
|-------|---------------------|
| <</2  | bitový posun vlevo  |
| >>/2  | bitový posun vpravo |
| \ /2  | bitové OR           |
| &/2   | bitové AND          |
| \ 1   | bitová negace       |
| xor/2 | bitový XOR          |

## Vybrané aritmetické funkce

|      |               |       |                            |
|------|---------------|-------|----------------------------|
| -/1  | unární mínus  | //2   | celočíselné dělení         |
| +/1  | znaménko plus | rem/2 | zbytek po dělení //        |
| +/2  | součet        | div/2 | dělení a zaokrouhlení dolů |
| -/2  | rozdíl        | mod/2 | zbytek po dělení div       |
| */2  | součin        | max/2 | maximum                    |
| //2  | dělení        | min/2 | minimum                    |
| **/2 | mocnina       | is/2  | vyhodnocení a unifikace    |

## Pozorování

- Pro strukturovaný term, který dává do relace dva jiné termy, je možné nechat Prolog dohledat termy, pro které relace platí.
- `rel(a,b).`  
`?- rel(X,b).`  
`X = a.`

## Neplatí pro argumenty aritmetických operací

- Prolog při unifikaci a rezoluci nepočítá inverzní funkce, v okamžiku požadavku na takovouto operaci ohlásí interpret chybu (nedostatečná instanciace).
- Porovnejte:

`?- X is 3*3.`

`X = 9.`

`?- 9 is 3*X.`

**ERROR**

## Vyzkoušejte

- `?- 9 is X + 1.`                      `?- X > 3.`                      `?- X = 2, X > 3.`  
**ERROR**                                      **ERROR**                                      `false.`

## Vysvětlete rozdílné chování

- Korektní definice predikátu pro výpočet délky seznamu.  
`length([],0).`  
`length([_|T],N) :- length(T,X), N is X+1.`
- Nevhodná definice predikátu pro výpočet délky seznamu.  
`length1([],0).`  
`length1([_|T],N) :- N is X+1, length1(T,X).`
- Rozdílné chování při výpočtu.  
`?- length([a,b],X).`                      `?- length1([a,b],X).`  
`X = 2.`                                              **ERROR**

## Pozorování

- Předdefinované predikáty mohou vyžadovat, aby některé parametry byly povinně instanciované, tzn. na jejich místě nelze použít proměnnou.

## Používaná notace v dokumentaci

- +Arg: musí být instanciovaný parametr.
- -Arg: očekává se proměnná.
- ?Arg: instanciovaný parametr nebo proměnná.
- @Arg: parametr nebude vázán unifikací.
- :Arg: parametrem je název predikátu.

## Módy použití

- Je-li binární predikát použit s dvěma instanciovanými parametry, říkáme, že predikát je použit v (+,+) módu.

## `between(+Low, +High, ?Value)`

- Low and High are integers, High  $\geq$  Low. If Value is an integer, Low  $\leq$  Value  $\leq$  High. When Value is a variable it is successively bound to all integers between Low and High. If High is inf or infinite `between/3` is true iff Value  $\geq$  Low, a feature that is particularly interesting for generating integers from a certain value.

## `plus(?Int1, ?Int2, ?Int3)`

- True if Int3 = Int1 + Int2. **At least two of the three arguments must be instantiated to integers.**

## `sort(+List, -Sorted)`

- True if Sorted can be unified with a list holding the elements of List, sorted to the standard order of terms. Duplicates are removed. The implementation is in C, using natural merge sort.

The `sort/2` predicate can sort a cyclic list, returning a non-cyclic version with the same elements.

## **Efektivita – Tail rekurze**



## Pozorování

- Uvažme následující definici predikátu `length`.  
`length([],0).`  
`length([_|T],N) :- length(T,X), N is X+1.`
- Nevýhodou této definice je, že při volání dochází k výpočtu před rekurzivním voláním (při rekurzivním sestupu) i po rekurzivním volání (při vynořování z rekurze).

## Tail rekurze

- Definice, jež nevynucuje výpočet po rekurzivním volání, tj. rekurzivní cíl je jeden a je uveden jako poslední podcíl.
- Výsledek je znám při dosažení dna rekurzivního sestupu.
- Menší režie výpočtu, větší efektivita.
- Platí i ve světě imperativních programovacích jazyků.

## Pozorování

- Tail-rekurzivní definice lze dosáhnout použitím akumulátoru.

## Predikát `length` definován tail-rekurzivně

- Realizace pomocným predikátem s akumulátorem.

`length(Seznam, Delka) :- accLen(Seznam, 0, Delka).`

- Definice pomocného predikátu.

`accLen([], A, A).`

`accLen([_|T], A, L) :- B is A+1, accLen(T, B, L).`

- Mód použití `accLen` je `(?, +, ?)`.

## Efektivita – Řez

## Pozorování

- Základem výpočtu logického programu je **backtracking**.
- Některé větve výpočtu nevedou k požadovanému cíli.
- Jistá kontrola nad způsobem prohledávání SLD stromu, by byla vhodná.

## Dosavadní možnosti ovlivnění výpočtu

- Změna pořadí faktů v databázi.
- Změna pořadí podcílů v definici pravidla.

## Operátor řezu – !/0

- Vždy jako podcíl úspěje.
- Ovlivňuje způsob výpočtu (má vedlejší efekt).
- Eliminuje další volby, které by Prolog udělal při procházení výpočetního stromu, a to od okamžiku unifikace podcíle s levou stranou pravidla, ve kterém se predikát ! vyskytuje, až do místa výskytu !.

## Důsledky vedlejšího efektu

- Prořezává výpočetní strom.
- Rychlejší výpočet.
- Riziko odřezání větví výpočtu, které vedou k dalším (stejným, či jiným) řešením.

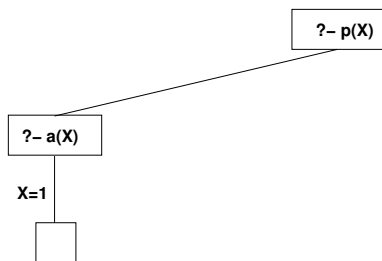
# Příklad fungování řezu – bez řezu

?- p(X)

```
p(X) :- a(X).
p(X) :- b(X), c(X),
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

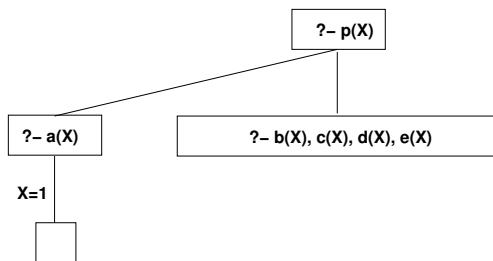
# Příklad fungování řezu – bez řezu



```
p(X) :- a(X).
p(X) :- b(X), c(X),
 d(X), e(X).
p(X) :- f(X).
```

```
a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

# Příklad fungování řezu – bez řezu

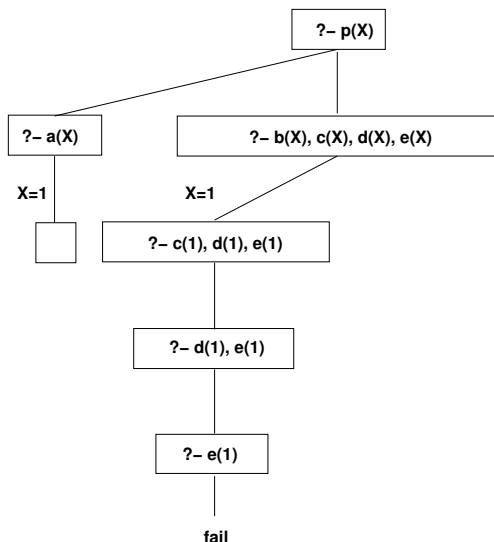


```
p(X) :- a(X).
p(X) :- b(X), c(X),
 d(X), e(X).
p(X) :- f(X).
```

```
a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```



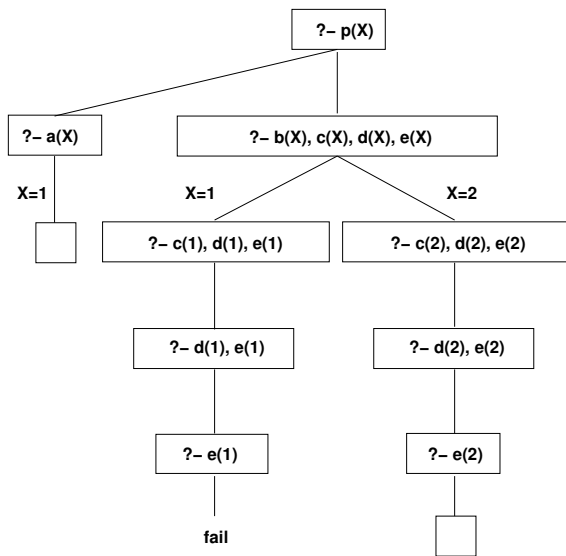
# Příklad fungování řezu – bez řezu



```
p(X) :- a(X).
p(X) :- b(X), c(X),
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

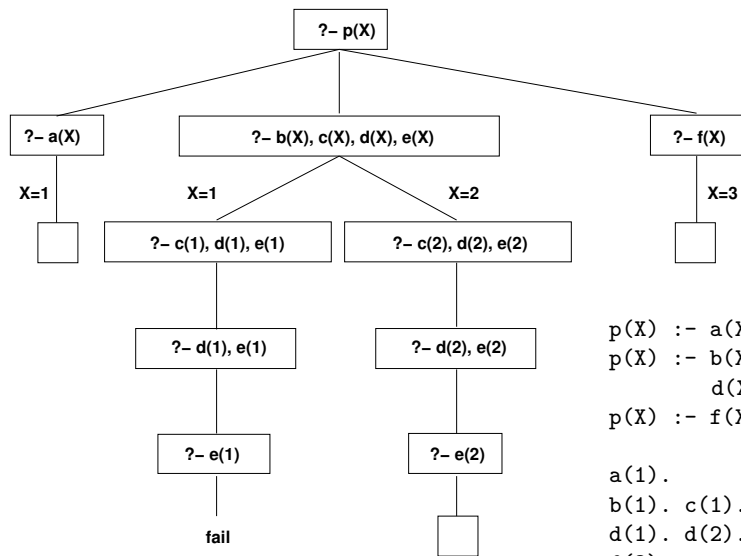
# Příklad fungování řezu – bez řezu



```
p(X) :- a(X).
p(X) :- b(X), c(X),
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

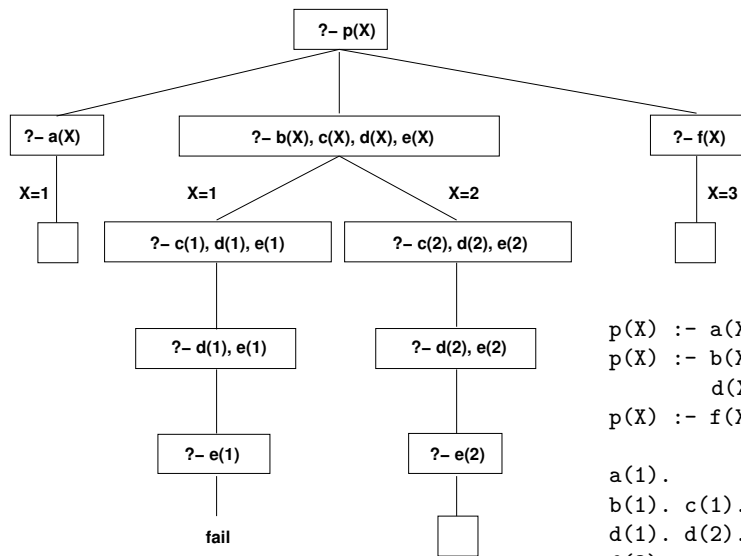
# Příklad fungování řezu – bez řezu



`p(X) :- a(X).`  
`p(X) :- b(X), c(X),`  
`d(X), e(X).`  
`p(X) :- f(X).`

`a(1).`  
`b(1). c(1).`  
`d(1). d(2). e(2).`  
`f(3).`  
`b(2). c(2).`

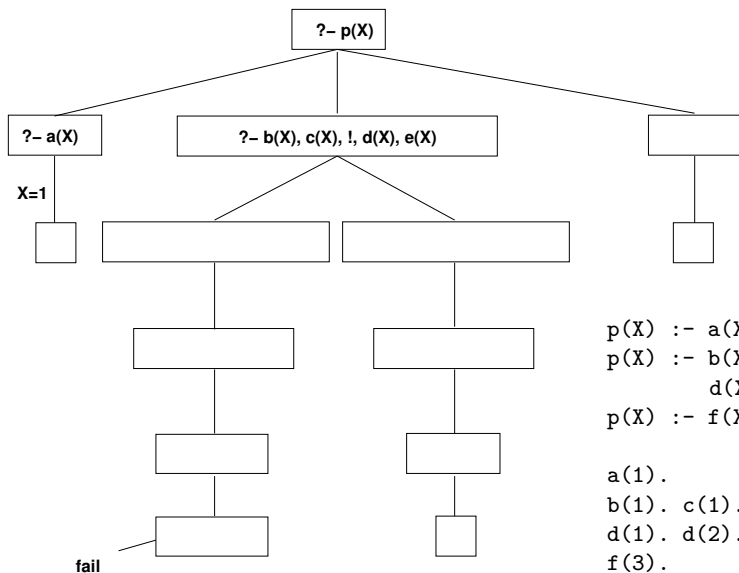
# Příklad fungování řezu – s řezem



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

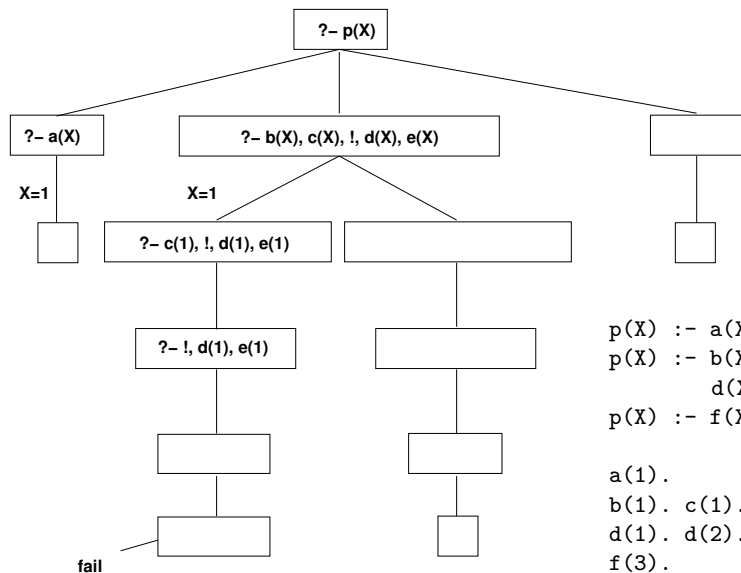
# Příklad fungování řezu – s řezem



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

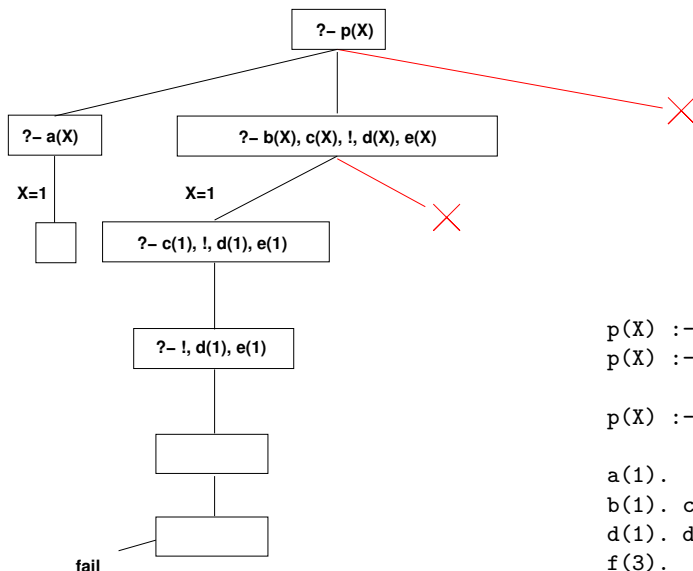
# Příklad fungování řezu – s řezem



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

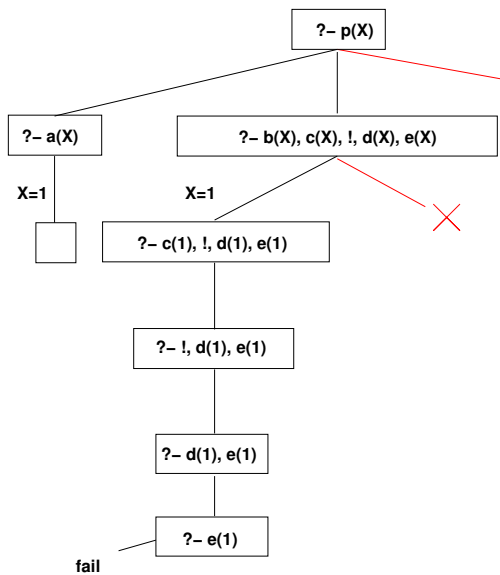
# Příklad fungování řezu – s řezem



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

# Příklad fungování řezu – s řezem



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```



## Popis

- Pokud se při řešení podcíle narazí v těle pravidla na operátor `!`, ostatní fakta a pravidla, se pro právě řešený cíl (ten, který se unifikoval s hlavou pravidla) neberou v potaz.

## Příklad

- Porovnej chování následujících programů.

a) `a(X) :- X = 1.`

`a(X) :- X = 2.`

`?- a(X).`

`X = 1 ;`

`X = 2.`

b) `a(X) :- X = 1, !.`

`a(X) :- X = 2.`

`?- a(X).`

`X = 1.`

## Popis

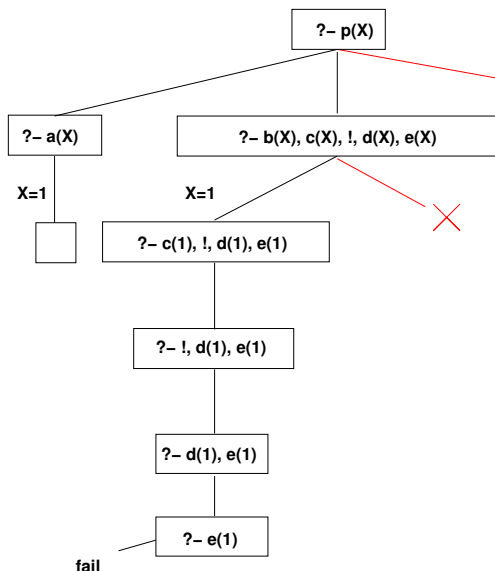
- Pokud se při řešení podcíle narazí v těle pravidla na operátor řezu, všechny unifikace vyplývající z podcílů vyskytujících se v těle pravidla před operátorem ! se fixují (jiné možnosti unifikace těchto podcílů se neuvažují).

## Porovnejte

a)  $a(X) :- X = 0.$   
 $a(X) :- X = 1.$   
 $b(X,Y) :- a(X), a(Y).$   
 $?- b(X,Y).$   
 $X = 0, Y = 0 ;$   
 $X = 0, Y = 1 ;$   
 $X = 1, Y = 0 ;$   
 $X = 1, Y = 1.$

b)  $a(X) :- X = 0.$   
 $a(X) :- X = 1.$   
 $b(X,Y) :- a(X), !, a(Y).$   
 $?- b(X,Y).$   
 $X = 0, Y = 0 ;$   
 $X = 0, Y = 1.$

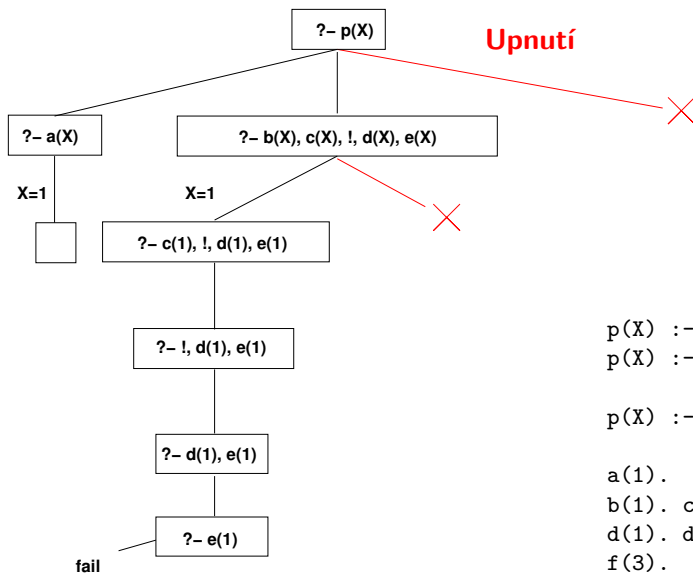
# Příklad fungování řezu – vedlejší efekty



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

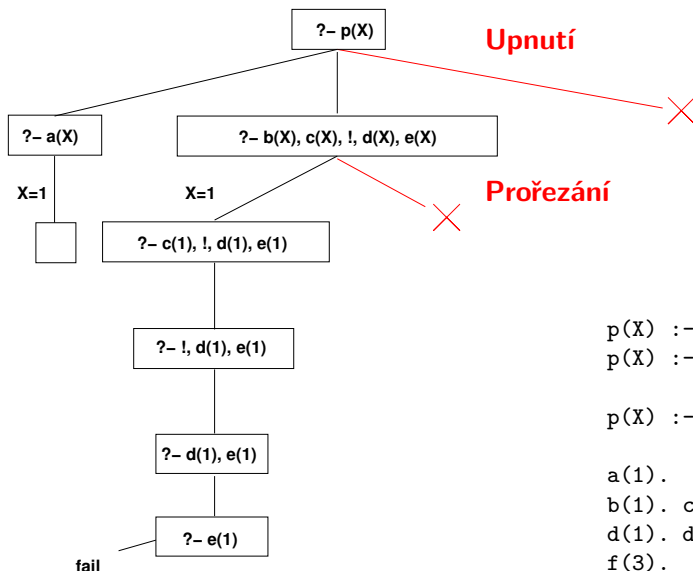
# Příklad fungování řezu – vedlejší efekty



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

# Příklad fungování řezu – vedlejší efekty



```
p(X) :- a(X).
p(X) :- b(X), c(X), !,
 d(X), e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

## Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y)`.

## Zadání 1

- `a(X) :- X = 0.`  
`a(X) :- X = 1, !.`  
`a(X) :- X = 2.`  
`b(X,Y) :- a(X), a(Y).`

## Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y)`.

### Zadání 1

- `a(X) :- X = 0.`  
`a(X) :- X = 1, !.`  
`a(X) :- X = 2.`  
`b(X,Y) :- a(X), a(Y).`

### Řešení 1

- `X = 0, Y = 0 ;`  
`X = 0, Y = 1 ;`  
`X = 1, Y = 0 ;`  
`X = 1, Y = 1.`

## Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y)`.

### Zadání 1

- `a(X) :- X = 0.`  
`a(X) :- X = 1, !.`  
`a(X) :- X = 2.`  
`b(X,Y) :- a(X), a(Y).`

### Řešení 1

- `X = 0, Y = 0 ;`  
`X = 0, Y = 1 ;`  
`X = 1, Y = 0 ;`  
`X = 1, Y = 1.`

### Zadání 2

- `a(X) :- X = 0.`  
`a(X) :- X = 1, !.`  
`a(X) :- X = 2.`  
`b(X,Y) :- a(X), !, a(Y).`



## Úkol

- Určete maximální možný výstup interpretru pro následující kód v Prologu na dotaz `?- b(X,Y)`.

### Zadání 1

- `a(X) :- X = 0.`  
`a(X) :- X = 1, !.`  
`a(X) :- X = 2.`  
`b(X,Y) :- a(X), a(Y).`

### Řešení 1

- `X = 0, Y = 0 ;`  
`X = 0, Y = 1 ;`  
`X = 1, Y = 0 ;`  
`X = 1, Y = 1.`

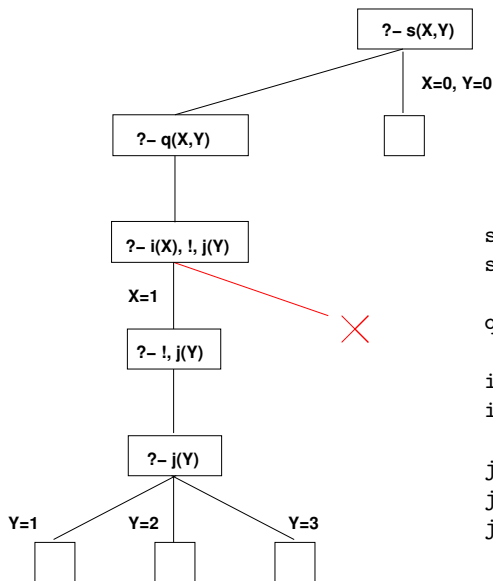
### Zadání 2

- `a(X) :- X = 0.`  
`a(X) :- X = 1, !.`  
`a(X) :- X = 2.`  
`b(X,Y) :- a(X), !, a(Y).`

### Řešení 2

- `X = 0, Y = 0 ;`  
`X = 0, Y = 1.`

# Příklad fungování řezu – imunita nadřazených cílů



`s(X,Y) :- q(X,Y).`  
`s(0,0).`

`q(X,Y) :- i(X), !, j(Y).`

`i(1).`

`i(2).`

`j(1).`

`j(2).`

`j(3).`

## Zelené řезы

- Odstraněním operátoru řezu se nemění sémantika programu (množina řešení po odstranění řezu je shodná).
- Řez je použit pouze z důvodů efektivity.
- Někdy se jako „modré“ označují řезы eliminující duplicity.

## Červené řезы

- Odstraněním operátoru řezu se mění sémantika programu (po odstranění řezu, je možné nalézt další jiná řešení).

## Obecná doporučovaná strategie

- Vyrobit funkční řešení bez řезů.
- Zvýšit efektivitu použitím „zelených“ řезů.
- Využít „červené řезы“ pouze pokud není vyhnutí, dobře okomentovat.

## Zadání

- V Prologu naprogramujte predikáty `encodeRLE/2` a `decodeRLE/2`, které budou realizovat RLE kódování seznamů.
- V RLE kódování je každá  $n$ -tice stejných po sobě jdoucích prvků  $k$  v seznamu nahrazena uspořádanou dvojicí  $(n, k)$ .

## Příklad použití

- `?- encodeRLE([a,a,a,b,b,c,d,d,e],X).`  
`X = [(3,a),(2,b),(1,c),(2,d),(1,e)]`
- `?- decodeRLE([(5,1),(1,2),(3,3)],[1,1,1,1,1,2,3,3,3]).`  
`true.`

# IB015 Neimperativní programování

Programování s omezujícími podmínkami  
a závěrečné zhodnocení

Jiří Barnat

# Programování s omezujícími podmínkami

## Vymezení pojmu

- Obecné neimperativní programovací paradigma.
- V množině možných řešení problému je hledané řešení popsáno pouze omezujícími podmínkami, které musí splňovat.
- Angl. Constraint programming.

## Aplikace

- Problémy vedoucí na těžké kombinatorické řešení.
- Řízení, rozvrhování, plánování.
- DNA sequencing.

## Vymezení pojmu

- Obecné neimperativní programovací paradigma.
- V množině možných řešení problému je hledané řešení popsáno pouze omezujícími podmínkami, které musí splňovat.
- Angl. Constraint programming.

## Aplikace

- Problémy vedoucí na těžké kombinatorické řešení.
- Řízení, rozvrhování, plánování.
- DNA sequencing.
- StarTrek





## Různé instance paradigmatu

- Podle typu proměnných, vystupujících v popisu problému.
- Pravdivostní hodnoty, Celočíselné hodnoty, Konečné množiny, Doména lineárních funkcí, ...

## Postup řešení úloh

- |                                                                          |          |
|--------------------------------------------------------------------------|----------|
| ● Modelování problému v dané doméně.                                     | Myšlenka |
| ● Specifikace proměnných a jejich rozsahů.                               | Program  |
| ● Specifikace omezujících podmínek.                                      | Program  |
| ● Vymezení cíle.                                                         | Program  |
| ● Zjednodušení zadání, propagace omezení.                                | Výpočet  |
| ● Systematické procházení možných valuací a hledání vyhovujícího řešení. | Výpočet  |

## Hostitelské jazyky

- Řešiče uvažovaných úloh jsou obvykle součástí jiného hostitelského programovacího jazyka nebo systému.
- Prvním výrazným hostitelem byly jazyky vycházející z logického programovacího paradigmatu.
- **Constraint Logic Programming** (CLP).
- Mnoho instancí je dostupných jako SAT a SMT řešiče (SMTLib).

## Knihovny ve SWI-Prologu

- **clpfd**: Constraint Logic Programming over Finite Domains  
`?- use_module(library(clpfd)).`
- **clpqr**: Constraint Logic Programming over Rationals and Reals  
`?- use_module(library(clpqr)).`

## Výrazy v celočíselné doméně

- Celé číslo je výrazem v celočíselné doméně.
- Proměnná je výrazem s celočíselné doméně.
- Jsou-li  $E1$  a  $E2$  výrazy v celočíselné doméně, pak
  - $E1$  (unární mínus)
  - $E1+E2$  (součet),  $E1 * E2$  (součin),  $E1 - E2$  (rozdíl),
  - $E1 \wedge E2$  (umocnění),  $\min(E1, E2)$ ,  $\max(E1, E2)$ ,
  - $E1 / E2$  (celočíselné dělení ořezáním),
  - $E1 \text{ rem } E2$  (zbytek po dělení /)jsou výrazy v celočíselné doméně.

## Omezující podmínky

- Relační operátory předřazené znakem #.
- $E1 \#>= E2$ ,  $E1 \#<= E2$ ,
- $E1 \#= E2$ ,  $E1 \#\backslash= E2$ ,
- $E1 \#> E2$ ,  $E1 \#< E2$ ,

## Logické spojky

- $\neg Q$  – Negace
- $P \vee Q$  – Disjunkce
- $P \wedge Q$  – Konjunkce
- $P \iff Q$  – Ekvivalence
- $P \implies Q$  – Implikace
- $P \Leftarrow Q$  – Implikace

## Číselná reprezentace logických hodnot

- Pravda/Nepravda jsou realizovány hodnotami 1 a 0.
- Relační operátory jsou aplikovatelné na tyto celočíselné hodnoty.

# Domény volných proměnných

?Var in +Domain

- Proměnná `var` má hodnotu z domény `Domain`.

+Vars ins +Domain

- Proměnné v seznamu `Vars` mají hodnotu z domény `Domain`.

**all\_different**(Vars)

- Každá proměnná ze seznamu `Vars` má jinou hodnotu.

## Specifikace domény

- $N$  — jednoprvková množina obsahující celé číslo  $N$ .
- `Lower..Upper` — všechna celá čísla  $I$  taková, že  $Lower \leq I \leq Upper$ , `Lower` musí být celé číslo, nebo term `inf` označující záporné nekonečno, podobně `Upper` musí být celé číslo, nebo term `sup` označující kladné nekonečno.
- `Domain1 \ / Domain2` — sjednocení domén `Domain1` a `Domain2`.

## Pozorování

- Následující dotazy jsou řešeny pouze fází propagace omezujících podmínek (neprochází se systematicky prostor všech možných přiřazení hodnot volným proměnným).

## Příklady dotazů na clpfd

- `?- X #\= 20.`  
`X in inf..19\21..sup.`
- `?- X*X #= 144.`  
`X in -12\12.`
- `?- 4*X + 2*Y #= 24, X + Y #= 9, X #>= 0, Y #>= 0.`  
`X = 3, Y = 6.`
- `?- X #= Y #<==> B, X in 0..3, Y in 4..5.`  
`B = 0, X in 0..3, Y in 4..5.`

## Popis

- Kryptoaritmetické puzzle, každé písmeno představuje jednu cifru, žádná dvě různá písmena nepředstavují tutéž cifru. Jaké je mapování písmen na číslice?

## Zadání pro clpfd

- ```
puzzle([S,E,N,D]+ [M,O,R,E] = [M,O,N,E,Y]) :-  
  Vars = [S,E,N,D,M,O,R,Y],  
  Vars ins 0..9,  
  all_different(Vars),  
          S*1000 + E*100 + N*10 + D +  
          M*1000 + O*100 + R*10 + E #=  
M*10000 + O*1000 + N*100 + E*10 + Y,  
M #\= 0, S #\= 0.
```

label(+Vars)

- Zahájí hledání vyhovujících hodnot proměnných `Vars`.
- Totéž, co `labeling([],Vars)`.

labeling(+Options,+Vars)

- Zahájí hledání vyhovujících hodnot proměnných `Vars`.
- Parametry uvedené v seznamu `Options` ovlivňují způsob enumerace hledaných hodnot.

Parametry hledání

- Pořadí fixace proměnných.
- Směr prohledávání domén.
- Strategie větvení prohledávaného stromu.

Pořadí fixace proměnných

- `leftmost` — přiřazuje hodnoty proměnným v tom pořadí, ve kterém jsou uvedeny.
- `ff` — preferuje proměnné s menšími doménami.
- `ffc` — preferuje proměnné, které participují v největším počtu omezujících podmínek.
- `min` — preferuje proměnná s nejmenší spodní závorou.
- `max` — preferuje proměnná s největší horní závorou.

Směr prohledávání domén

- `up` — zkouší prvky domény od nejmenších k největším.
- `down` — zkouší prvky domény od největších k nejmenším.

Odpověď clpfd bez prohledávání

- Vars = [9, E, N, D, 1, 0, R, Y],
S = 9, M = 1, O = 0,
E in 4..7, N in 5..8, D in 2..8, R in 2..8, Y in 2..8,
all_different([9, E, N, D, 1, 0, R, Y]),
1000*9+91*E+ -90*N+D+ -9000*1+ -900*0+10*R+ -1*Y#=0.

Požadavek na prohledávání

- Uvedením podcíle label([S,E,N,D]).

Odpověď clpfd s vyhledáním valuací proměnných S,E,N a D

- Vars = [9, 5, 6, 7, 1, 0, 8, 2],
S = 9, E = 5, N = 6, D = 7,
M = 1, O = 0, R = 8, Y = 2 ;
false.

sum(+Vars,+Rel,?Expr)

- Součet hodnot proměnných v seznamu Vars je v relaci Rel s hodnotou výrazu Expr.

scalar_product(+Cs,+Vs,+Rel,?Expr)

- Skalární součin seznamu čísel Cs s čísly, nebo proměnnými v seznamu Vs, je v relaci Rel s hodnotou výrazu Expr.

serialized(+Starts,+Durations)

- Pro hodnoty Starts=[S1,...,SN] a Durations=[D1,...,DN], platí, že úlohy začínající v čase SI a trvající dobu DI se nepřekrývají, tj. $SI+DI \leq SJ$ nebo $SJ+DJ \leq SI$.

Jiné použití clpfd v Prologu

- Aritmetické vyhodnocování v celých číslech bez nutnosti instanciace argumentů aritmetických operací (propagace hodnot všemi směry).

Příklad

- `n_factorial(0,1).`
`n_factorial(N,F) :-`
 `N #> 0, N1 #= N - 1, F #= N * F1,`
 `n_factorial(N1,F1).`
- `?- n_factorial(N,1).`
`N = 0 ;`
`N = 1 ;`
`false.`

Einsteinova hádanka

Popis situace

- Je 5 domů, z nichž každý má jinou barvu.
- V každém domě žije jeden člověk, který pochází z jiného státu.
- Každý člověk pije nápoj, kouří jeden druh cigaret a chová jedno zvíře.
- Žádný z nich nepije stejný nápoj, nekouří stejný druh cigaret a nechová stejné zvíře.

Otázka

- Kdo chová rybičky?
- Za následujících předpokladů ...

Zadání hádanky – nápovědy

- 1 Brit bydlí v červeném domě.
- 2 Švéd chová psa.
- 3 Dán pije čaj.
- 4 Zelený dům stojí hned nalevo od bílého.
- 5 Majitel zeleného domu pije kávu.
- 6 Ten, kdo kouří PallMall, chová ptáka.
- 7 Majitel žlutého domu kouří Dunhill.
- 8 Ten, kdo bydlí uprostřed řady domů, pije mléko.
- 9 Nor bydlí v prvním domě.
- 10 Ten, kdo kouří Blend, bydlí vedle toho, kdo chová kočku.
- 11 Ten, kdo chová koně, bydlí vedle toho, kdo kouří Dunhill.
- 12 Ten, kdo kouří BlueMaster, pije pivo.
- 13 Němec kouří Prince.
- 14 Nor bydlí vedle modrého domu.
- 15 Ten, kdo kouří Blend, má souseda, který pije vodu.

Copy-paste, aneb programátorova smrt

- `einstein_0.pl`

Přeuspořádání, aneb optimalizace v praxi

- `einstein_1.pl`

Transformace na řešení absolventa FI

- `einstein_2.pl`
- `einstein_3.pl`
- `einstein_4.pl`
- `einstein_5.pl`

Deklarativní versus imperativní

Princip

- Programem je především formulace cíle a vztahu požadovaného výsledku výpočtu k daným vstupům.
- Popis postupu výpočtu není požadován, nebo je druhotným vstupem zadávaným kvůli zvýšení efektivity výpočtu.

Výhody a nevýhody

- + Kratší a srozumitelnější kód.
- + Méně skrytých chyb.
- Náročnější tvorba kódu, požaduje schopnost abstrakce.
- Riziko neefektivního řešení.
- Obtížná přímá kontrola výpočetního HW.

Princip

- Programem je popis transformace zadaných vstupů na požadovaný výsledek.
- Popis vztahů výsledku vzhledem ke vstupům není požadován, nebo je do programu vkládán za účelem kontroly korektnosti popisované transformace.

Výhody a nevýhody

- + Detailní kontrola nad postupem výpočtu.
- + Efektivní využití dostupného HW .
- + Snazší tvorba kódu.
- Více prostoru pro zanesení chyb.
- Skryté a dlouho neodhalené chyby.
- Nečitelnost významu programu.

Jazykové konstrukce

- Nepojmenované funkce (lambda funkce).
- Parametrický polymorfismus / generické programování.
- Silná typová kontrola.
- Sémantika jazyka oddělená od výpočetního HW.

Programátorský styl

- Přenos kontroly typů z doby za běhu programu do doby kompilace.
- Deklarace vzájemných vztahů vnitřních dat v imperativním programu.
- Programování bez pomocných přepisovatelných proměnných.

Původně imperativním stylem

- ```
int vysledek=1;
for (int i=1; i<=N; i++)
{
 vysledek=vysledek*i;
}
print vysledek;
```

## Nově deklarativním stylem

- ```
int fact(int n)
{
    if (n==0) return 1;
    else return n*fact(n-1);
}
print fact(N);
```

Původně imperativním stylem

- ```
int vysledek=1;
for (int i=1; i<=N; i++)
{
 vysledek=vysledek*i;
}
print vysledek;
```

- Co to vlastně počítá?

## Nově deklarativním stylem

- ```
int fact(int n)
{
    if (n==0) return 1;
    else return n*fact(n-1);
}
print fact(N);
```

Původně imperativním stylem

- ```
int vysledek=1;
for (int i=1; i<=N; i++)
{
 vysledek=vysledek*i;
}
print vysledek;
```

- Co to vlastně počítá?
- Přepisovatelná proměnná navíc, těžší optimalizace.

## Nově deklarativním stylem

- ```
int fact(int n)
{
    if (n==0) return 1;
    else return n*fact(n-1);
}
print fact(N);
```

Původně imperativním stylem

- ```
int vysledek=1;
for (int i=1; i<=N; i++)
{
 vysledek=vysledek*i;
}
print vysledek;
```

- Co to vlastně počítá?
- Přepisovatelná proměnná navíc, těžší optimalizace.
- Větší prostor pro zanesení chyb ( $i=1, i \leq N$ ).

## Nově deklarativním stylem

- ```
int fact(int n)
{
    if (n==0) return 1;
    else return n*fact(n-1);
}
print fact(N);
```


Původně imperativním stylem

- ```
int vysledek=1;
for (int i=1; i<=N; i++)
{
 vysledek=vysledek*i;
}
print vysledek;
```

- Co to vlastně počítá?
- Přepisovatelná proměnná navíc, těžší optimalizace.
- Větší prostor pro zanesení chyb ( $i=1, i \leq N$ ).
- Skryté chování pro  $N=0$ .

## Nově deklarativním stylem

- ```
int fact(int n)
{
    if (n==0) return 1;
    else return n*fact(n-1);
}
print fact(N);
```

Původně imperativním stylem

- ```
int vysledek=1;
for (int i=1; i<=N; i++)
{
 vysledek=vysledek*i;
}
print vysledek;
```

- Co to vlastně počítá?
- Přepisovatelná proměnná navíc, těžší optimalizace.
- Větší prostor pro zanesení chyb ( $i=1, i \leq N$ ).
- Skryté chování pro  $N=0$ .

## Nově deklarativním stylem

- ```
int fact(int n)
{
    if (n==0) return 1;
    else return n*fact(n-1);
}
print fact(N);
```

- Jasně chování pro $N=0$.

Původně imperativním stylem

```
• int vysledek=1;
  for (int i=1; i<=N; i++)
  {
    vysledek=vysledek*i;
  }
  print vysledek;
```

- Co to vlastně počítá?
- Přepisovatelná proměnná navíc, těžší optimalizace.
- Větší prostor pro zanesení chyb ($i=1, i \leq N$).
- Skryté chování pro $N=0$.

Nově deklarativním stylem

```
• int fact(int n)
  {
    if (n==0) return 1;
    else return n*fact(n-1);
  }
  print fact(N);
```

- Jasně chování pro $N=0$.
- Pojmenovaná funkce, syntaktická indicie pro sémantický význam.

A to je konec ...

Co si odneseme do života ...

- Funkcionální výpočetní paradigma.
- Solidní základy programovacího jazyka Haskell.
- Intuitivní základy programování v Prologu.

Čím ještě nám byl kurz prospěšný ...

- **Deklarativní návyky při návrhu programů a algoritmů mnohokrát využijeme v naší (převážně imperativní) informatické praxi.**
- Mentální posilovna.

Co si odneseme do života ...

- Funkcionální výpočetní paradigma.
- Solidní základy programovacího jazyka Haskell.
- Intuitivní základy programování v Prologu.

Čím ještě nám byl kurz prospěšný ...

- **Deklarativní návyky při návrhu programů a algoritmů mnohokrát využijeme v naší (převážně imperativní) informatické praxi.**
- Mentální posilovna.



Přednášejícího studentům

- Za vzornou docházku a přípravu jak na přednášky, tak i na cvičení, a zkouškové písemky, a za celkově poctivý přístup ke studiu.

Studentů přednášejícímu

- Formou zpětné vazby například vyplněním studentské ankety a upozorněním na zásadní, ale i okrajové nedostatky jak přednášejícího, tak i jím připravených studijních materiálů.

The goal of IB015

