IB015 Neimperativní programování

Organizace a motivace kurzu, programovací jazyk Haskell

Jiří Barnat

Organizace kurzu

Cíle kurzu

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

- Rozumí funkcionálnímu a logickému výpočetnímu paradigmatu.
- Je schopen dekomponovat výpočetní problém na jednotlivé funkce a tuto schopnost používá při vytváření vlastních kódů i v imperativních programovacích jazycích.
- Chápe nedostatky techniky COPY-PASTE programování a umí tuto techniku programování efektivně nahradit.
- Má základní znalost programovacích jazyků Haskell a Prolog.

Požadavky a předpoklady pro úspěšné absolvování

Předpoklady

- Možné úspěšně absolvovat bez znalosti programování.
- Schopnost abstraktního myšlení.

Znalost imperativního programování

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

Požadavky na ukončení

Forma ukončení

- Závěrečný písemný test.
- Vnitrosemestrální písemný test (! nelze opakovat).
- Možnost získat až 10 bodů za domácí úlohy ze cvičení.
- Možnost získat až 10 bodů za aktivitu na cvičení.

Požadavky na úspěšné ukončení

- Nutno získat 48 bodů ze 100+.
- Body za aktivitu na cvičení se přičtou pouze pokud součet ostatních bodů je alespoň 48.

Kontext v rámci FI

Obory

- IB015 je povinná součást studijního základu.
- Doporučen převážně do 3. semestru studia.

Navazující předměty

- IB016 Seminář z funkcionálního programování
- IB013 Logické programování
- IA014 Funkcionální programování (???)
- IA008 Výpočetní logika

Zdroje a učební materiály

Funkcionální paradigma

- http://haskell.cz/
- Thompson, Simon. Haskell: the craft of functional programming.
- Structure and Interpretation of Computer Programs [http://mitpress.mit.edu/sicp/full-text/book/book.html]

Logické paradigma

- http://www.learnprolognow.org
- Nerode, Shore: Logic for Applications

Co znamená programovat?

Programování a programovací jazyk

Programování

- Vytváření 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.

Programovací jazyk

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

Co znamená umět programovat?

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 moderní doby

Riziko

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

Riziko moderní doby

Riziko

- 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

Riziko moderní doby

Riziko

- 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

Programovací jazyky

Klasifikace

- Imperativní — C/C++, Java, Perl, php, ...
- Funkcionální – **Haskell**, OCaML, Erlang, Lisp, ...
- Logické – **Prolog**, ...
- Kombinované – C#, Scala, C++, ...
- ...

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 **interpretru**.
- Pro každý programovací jazyk je potřeba jiný překladač/interpretr.

Překladače a interpretry

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.

Příklad překladu a interpretace

Programovací jazyk Haskell

- Překladač – `ghc`.
- Interaktivní interpretr – `ghci` (dříve též `hugs`).
- Neinteraktivní interpretace – runghc.

Překladače programovacího jazyka C/C++

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

Programujeme pomocí funkcí

Co je to funkce?

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)`
- `objemkvadru a b c = a*b*c`
- ...

S čím funkce pracují

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.
  `rotate90r :: Obrázek -> Obrázek`
- Objem kvádru.
  `objemkvadru :: Číslo` $\times$ `Číslo` $\times$ `Číslo -> Číslo`
- Počet hran polygonu.
  `hranypolygonu :: Polygon -> Celé_číslo`

Předpoklady

`rotate90r :: Obrázek -> Obrázek`

`hranypolygonu :: Obrázek -> Celé_číslo`

$\bigtriangleup$ `:: Obrázek`

Aplikace funkcí

`rotate90r` $\bigtriangleup$ `=` $\rhd$

`hranypolygonu` $\bigtriangleup$ `= 3`

Funkce jako základní stavební kameny

Pozorování

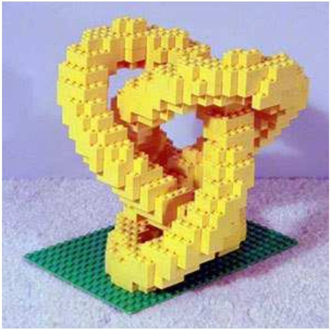
- Složitější úkony lze realizovat pomocí jednodušších operací.
- Složitější funkce lze definovat **složením** jednodušších.

Funkce jako základní stavební kameny

Pozorování

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

Skládání – cesta ke složitějším objektům a funkcím



Skládání funkcí

Operátor .

- `f1 ( f2 x ) = (f1 . f2) x`
- Čteme jako „f1 po f2".

Příklad

- Mějme funkci `double`, která vezme obrázek a vytvoří nový obrázek zkopírováním původního obrázku dvakrát vedle sebe.

```
double :: Obrázek -> Obrázek

double △ = △△
```

- Novou funkci `rotate_and_double` můžeme definovat takto:

```
rotate_and_double :: Obrázek -> Obrázek
rotate_and_double x = (double . rotate90r) x

rotate_and_double △ = ▷▷
```

Skládání funkcí a priority operací v Haskellu

Složené funkce a η-redukce

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

  ```
  rotate_and_double x = (double.rotate90r) x
  ```

  lze zapsat také jako

  ```
  rotate_and_double = double.rotate90r
  ```

POZOR na prioritu vyhodnocování v Haskellu

- Aplikace funkce na parametry má nejvyšší prioritu.

  `double.rotate90r` $\triangle$ `=` `double.(rotate90r` $\triangle$`)` $\rightsquigarrow$ **ERROR**

- Závorky kolem výrazu `double.rotate90r` jsou při aplikaci na hodnotu $\triangle$ nutné.

Příklady

`(rotate_and_double . rotate_and_double)` $\triangle$ `=`

`((double . double) . double)` $\triangle$ `=`

`(double . hranypolygonu)` $\triangle$ `=`

Příklady

(rotate_and_double . rotate_and_double) $\bigtriangleup$ = $\begin{matrix} \bigtriangledown & \bigtriangledown \\ \bigtriangledown & \bigtriangledown \end{matrix}$

((double . double) . double) $\bigtriangleup$ =

(double . hranypolygonu) $\bigtriangleup$ =

Příklady

```
(rotate_and_double . rotate_and_double) △ = ▽▽
                                            ▽▽

((double . double) . double) △ = △△△△△△△△

(double . hranypolygonu) △ =
```

Příklady

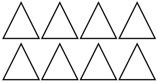
`(rotate_and_double . rotate_and_double)` △ = ▽▽ ▽▽

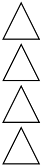
`((double . double) . double)` △ = △△△△△△△△

`(double . hranypolygonu)` △ = **ERROR**

Příklady – pokračování

Jak lze pomocí `double`, `rotate90r` a  vyrobit následující?

a) 

b) 

Terminologie

Typová signatura:

`rotate_and_double ::` <u>**Obrázek ->Obrázek**</u>

Jméno funkce

<u>**rotate_and_double**</u> `x = (double.rotate) x`

Tělo funkce

`rotate_and_double x =` <u>**(double.rotate) x**</u>

Definice funkce

<u>**rotate_and_double x = (double.rotate) x**</u>

Terminologie – pokračování

Formální parametr

```
rotate_and_double x = (double.rotate) x
```

Aktuální parametr

```
rotate_and_double △
```

Výraz

rotate_and_double △

Podvýraz

rotate_and_double (**rotate_and_double △**)

Funkcionální programování v Haskellu

Funkcionální výpočetní paradigma

- program = výraz + definice funkcí
- výpočet = úprava (zjednodušování) 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
```

Výpočet funkcionálního programu

Program

- definice funkcí

```
square x = x * x
pyth a b = square a + square b
```

- výraz

```
pyth 3 4
```

Výpočet

$$\underline{\texttt{pyth 3 4}} \leadsto \underline{\texttt{square 3}} \texttt{ + square 4} \leadsto \texttt{3 * 3 + } \underline{\texttt{square 4}} \leadsto$$

$$\leadsto \underline{\texttt{3 * 3}} \texttt{ + 4 * 4} \leadsto \texttt{9 + } \underline{\texttt{4 * 4}} \leadsto \underline{\texttt{9 + 16}} \leadsto$$

$$\leadsto 25$$

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 Haskellu pomocí <u>let ... in</u>

- **<u>let</u>** definice **<u>in</u>** výraz

```
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
```

Základy Haskellu – Základní datové typy

Čísla

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

Základy Haskellu – Víceřádkové definice

Příklad

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

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

Víceřádkové definice funkcí

- Na místě formálních parametrů se použijí tzv. vzory.
- Použije se první vzor, který vyhovuje.
- 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` ⇝ `False`
- `3 = 4` ⇝ **Error**

Základy Haskelu – Infix, Prefix, Parametry

Možnosti zápisu binárních funkcí

- Infixový zápis binárních funkcí: `3+4`, `4*5`
- Prefixový zápis binárních funkcí: `(+) 3 4`, `(*) 4 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 ozávorkovat.
- `(*) 3 4 + 5` $\leadsto$ 17
- `(*) 3 + 4 5` $\leadsto$ **Error**
- `(*) 3 (+) 4 5` $\leadsto$ **Error**
- `(*) 3 ( (+) 4 5 )` $\leadsto$ 27

Vstup výstupní operace

- Operace, které čtou a zapisují data z/do souborů, či terminálu.
- Program v Haskellu je reprezentován výrazem `main`.
- Program je sekvence vstup-výstupních akcí.
- Funkcionální princip vstup/výstupních akcí je složitý, bude vysvětlen později.

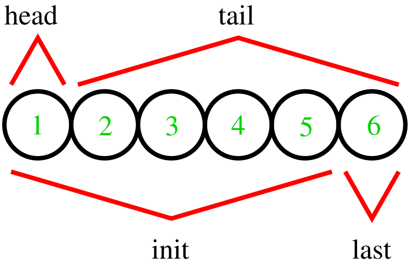
do notace programu v Haskellu

- ```
  main = do putStr "Zadej celé číslo: "
            x <-getLine
            print ((+) 1 (read x::Int))
  ```
- Textové zarovnání je důležité!
  ```

Práce na doma

Zadání

- Napište a spusťte program v Haskellu, který bude řešit dělitelnost dvou čísel, tj. zeptá se uživatele na dělence, načte ho, pak se zeptá na dělitele, kterého také načte, a sdělí uživateli, zda je zadaný dělenec dělitelný beze zbytku zadaným dělitelem.

IB015 Neimperativní programování

Seznamy, Typy a Rekurze

Jiří Barnat
Libor Škarvada

Uspořádané n-tice a seznamy

Programování a data

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.
- Pro uchování informací používáme různé **datové struktury**.

Strukturovaná data

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**

Uspořádané n-tice

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", "majen10cm")`
- 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).

Seznamy

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: `[1,2,3,4,5,6,...]`
- 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.

Příklad dekompozice dat

Aplikace – Diář squashových partnerů.

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

Datová dekompozice

- Seznam kontaktů

  `[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 není typ

Typ není váš nepřítel

Typ není váš nepřítel

Co je to typ a k čemu je

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. (Typy se komponují stejně jako data.)

Konstrukce typů

Základní datové typy

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

Složené typy

- Uspořádané n-tice:

  `(Bool,Int)`

- Seznamy:

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

  `[Char]` $\equiv$ `String`

Funkcionální typy

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

Funkcionální typy

Konstrukce

- Jsou-li σ a τ nějaké typy, tak σ `->` τ je typ všech funkcí s parametrem typu σ a funkční hodnotou typu τ.

Typ n-árních funkcí

- Jsou-li σ_1, σ_2, $\sigma_3 \ldots \sigma_n$ a τ nějaké typy, tak

  $$\sigma_1 \texttt{ -> } \sigma_2 \texttt{ -> } \sigma_3 \texttt{ -> } \ldots \texttt{ -> } \sigma_n \texttt{ -> } \tau$$

  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.

Funkcionální typ a aplikace

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
```

Polymorfní typy

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
```

Typové třídy

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
```

Uspořádané n-tice a seznamy v Haskellu

Uspořádané n-tice 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: `()`

Předdefinované funkce pro uspořádané dvojice

Datové konstruktory

- `(,,...,)` – datový konstruktor uspořádané n-tice
- `(,)` – datový konstruktor uspořádané dvojice

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

Projekce

- `fst` , `snd` – projekce na první a druhou složku

```
fst ::  (a,b) -> a
fst (x,y) = x

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

Seznamy

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 ]
```

Konstrukce seznamů

Datové 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
```

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
```

Význam datových konstruktorů

Datové 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
```

Rekurze

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 parametrů.
- V čistě funkcionálním jazyce nahrazuje cykly známé z imperativních programovacích jazyků.

Příklad

- Funkci `length`, která při aplikaci na seznam vrací 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 [])))

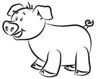
                  ⇝ 1 + ( 1 + ( 1 + ( 1 + 0 )))
                  ⇝ 1 + ( 1 + ( 1 + 1 ))
                  ⇝ 1 + ( 1 + 2 )
                  ⇝ 1 + 3
                  ⇝ 4
```

Další příklady použití rekurze

Číselné funkce

```
factorial :: Integer -> Integer

factorial 0 = 1
factorial x = x * factorial (x-1)
```

Nekonečné opakování (teoreticky)

```
main :: IO()

main = do putStrLn "Vloz text:"
          x <- getLine
          putStrLn ( "Zadal jsi:" ++ x )
          main
```

Rekurze je mentálně náročná

Pozorování

- Použití rekurze je možné pouze tehdy, je-li podproblém, na nějž se rekurzivně aplikuje dané řešení, **stejného typu**.

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 funkci? Jaký je typ této rekurzivní funkce?

Pozorování

- Použití rekurze je možné pouze tehdy, je-li podproblém, na nějž se rekurzivně aplikuje dané řešení, **stejného typu**.

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 funkci? Jaký je typ této rekurzivní funkce?

Řešení

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

Práce se seznamy v Haskellu

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
```



null, length, (!!)

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)
```

take, drop

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 : take (n-1) s   =   x : (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
```

takeWhile, dropWhile, map

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
```

zip,unzip

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
```

zipWith

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 (,)
```

Práce na doma

Technická cvičení

- Vyzkoušejte si všechny odpřednášené funkce na modelových seznamech v prostředí interpretru jazyka Haskell.

Mentální cvičení

- 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.
- 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

Funkce vyšších řádů

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

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

N-ární funkce jako funkce vyšších řádů

Pozorování

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

Příklad

- Funkci

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

  lze číst jako

  ```
  (*) :: Num a => a -> (a -> 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 -> a`.

Částečná aplikace

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

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

- Výsledkem aplikace `(*)` na hodnotu `3` je funkce.

```
(*) 3 :: Num a => a -> a
```

- Tuto funkci je možné označit, a posléze použít.

```
f = (*) 3
f :: Num a => a -> a
f 4 ⇝ ((*) 3) 4 ⇝ 12
```

Implicitní závorkování typu a aplikace

Připomenutí

- Typový konstruktor `->` je binární.
- Typový konstruktor `->` se používá pouze infixově.

Implicitní závorkování

- Typový konstruktor `->` implicitně sdružuje zprava

  $\mathbf{f} :: a_1 \texttt{ -> } (a_2 \texttt{ -> } (a_3 \texttt{ -> } \ldots \texttt{ -> } (a_{n-1} \texttt{ -> } (a_n \texttt{ -> } a)) \ldots))$

- Aplikace funkce na argumenty implicitně sdružuje zleva

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

Pozorování

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

Odvozování typu

S každou aplikací ubude jeden výskyt -> v typu výrazu

```
(+)         :: Num a => a -> a -> a
(+) 2       :: Num a =>      a -> a
((+) 2) 3 :: Num a =>           a
```

Odvozování typu

S každou aplikací ubude jeden výskyt `->` v typu výrazu

```
(+)         :: Num a => a -> a -> a
(+) 2       :: Num a =>      a -> a
((+) 2) 3   :: Num a =>           a
```

Specializací typové proměnné však mohou `->` přibýt.

- Vezměme například funkci identity

```
id :: a -> a
id x = x
```

- Při aplikaci `id` na `(+)` ubude `->` z typu `id`, tj.

```
id (+) :: a
```

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

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

Pořadí parametrů a částečná aplikace

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`

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, částečná aplikace a funkce `flip`

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
```

Jak bránit částečné aplikaci

Pozorování

- Funkce vyššího řádu a částečná aplikace souvisí s násobným použitím funkcionálního typového konstruktoru `->` .
- Chceme-li zabránit částečné aplikaci, musíme definovat funkci tak, aby v jejím typu byl pouze jeden výskyt `->` .
- 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
```

curry, uncurry

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 – curry a uncurry

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
```

- Uveďte 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
```

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

Řešení

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

Operátorové sekce

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 `⊕` a hodnoty `p` a `q`

```
⊕ :: a -> b -> c
p :: a
q :: b
```

- Částečnou aplikaci na první argument zapíšeme jako `(p⊕)`

```
(p⊕) = (⊕) p
```

- Částečnou aplikaci na druhý argument zapíšeme jako `(⊕q)`

```
(⊕q) = flip (⊕) q
```

Částečná aplikace – příklady

Příklad1

- Jednoduché použití operátorových sekcí.

```
(*2) 34 ⇝ 68
(++".") ['V','ě','t','a'] ⇝ "Věta."
```

Příklad2

- Odvoďte typ následujících funkcí, popište jejich význam:

```
(1.0/)
(`mod` 3)
(!!0)
(>0)
```

Částečná aplikace – příklady

Příklad1

- Jednoduché použití operátorových sekcí.

  (*2) 34 $\rightsquigarrow$ 68

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

Příklad2

- Odvoďte typ následujících funkcí, popište jejich význam:

```
(1.0/)     :: Fractional a => a -> a
(`mod` 3)  :: Integral a => a -> a
(!!0)      :: [a] -> a
(>0)       :: (Num a, Ord a) => a -> Bool
```

Binární operátor pro skládání funkcí

Skládání funkcí

- Základní operátor funkcionálního programování

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f ( g x )
```

Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

```
(.) :: (b -> c) -> (a -> b) -> (a -> 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 -> b) -> a -> c) -> d) -> (b -> c) -> d
```

Binární operátor pro skládání funkcí

Skládání funkcí

- Základní operátor funkcionálního programování

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f ( g x )
```

Pozorování

- Operátor pro skládání funkcí lze chápat také jako binární.

```
(.) :: (b -> c) -> (a -> b) -> (a -> 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 -> b) -> a -> c) -> d) -> (b -> c) -> d
```

Kombinátory

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*)
```

Nepojmenované funkce

Jednorázové použité pojmenované 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]
map f [1,2,3,4,5] where f x = x*x + 1 ⇝ [2,5,10,17,26]
```

Nepojmenované funkce

Co to je

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

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

Původ a všeobecné označení

- Myšlenka a teoretický základ pochází z Lambda kalkulu, proto se též nepojmenovaným 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, ...

Princip tvorby lambda funkcí

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

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

Lambda funkce v Haskellu

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
```

Nepojmenované funkce vyšších řádů

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

```
\ x -> (\ y -> x - y)
( \ x -> (\ y -> x - y) ) 3 4 ⇝ ... ⇝ -1
```

- Zkrácený z důvodu čitelnosti kódu a pohodlí programátorů

```
\ x y -> x - y
( \ x y -> x - y) 3 4 ⇝ ... ⇝ -1
```

Nepojmenované funkce a typy

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 konstruktoru `->` 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.

Nepojmenované funkce a jejich typy – příklady

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
```

Domácí cvičení

Mentální cvičení

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

IB015 Neimperativní programování

Akumulační funkce, Definice typů, Vstup/výstup

Jiří Barnat
Libor Škarvada

Akumulační funkce

Akumulační funkce `foldl`, `foldr`

Pozorování

- Seznam je posloupnost oddělených prvků.
- Motivací akumulačních 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

$\texttt{foldl1} \oplus [x_1, x_2, \ldots, x_n] \leadsto^* ((\ldots(x_1 \oplus x_2)\ldots) \oplus x_{n-1}) \oplus x_n$

$\texttt{foldr1} \oplus [x_1, x_2, \ldots, x_n] \leadsto^* x_1 \oplus (x_2 \oplus (\ldots(x_{n-1} \oplus x_n)\ldots))$

Akumulační funkce na neprázdných seznamech

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
```

Akumulační funkce `foldl`, `foldr`

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

$$\texttt{foldl} \oplus \mathbf{v}\ [\mathtt{x}_1,\mathtt{x}_2,\ldots,\mathtt{x}_n] \rightsquigarrow^* ((\ldots((\mathbf{v}\oplus\mathtt{x}_1)\oplus\mathtt{x}_2)\ldots)\oplus\mathtt{x}_{n-1})\oplus\mathtt{x}_n$$

$$\texttt{foldr} \oplus \mathbf{v}\ [\mathtt{x}_1,\mathtt{x}_2,\ldots,\mathtt{x}_n] \rightsquigarrow^* \mathtt{x}_1\oplus(\mathtt{x}_2\oplus(\ldots(\mathtt{x}_{n-1}\oplus(\mathtt{x}_n\oplus\mathbf{v}))\ldots))$$

Akumulační funkce na libovolných seznamech

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 akumulačních 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
```

Uživatelem definované typy

Riziko nepoužití uživatelem definovaných typů

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.**

Riziko nepoužití uživatelem definovaných typů – příklad

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** `=` **Datové_konstruktory**
- Jednotlivé datové konstruktory se oddělují znakem `|`
- Syntaktické omezení Haskellu: nově definovaný typ i datové 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 datové konstruktory – konstanty.

Definice n-árních datových konstruktorů

Uživatelem definované

- Obecná šablona pro n-ární datový konstruktor:

  Jméno Typ$_1$ ... Typ$_n$

- Příklad typu s ternárním datovým konstruktorem:

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

- Hodnoty typu Barva:

  ```
  RGB 42 42 42
  RGB 12 (-23) 45
  ```

Částečná aplikace datového konstruktoru

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

Typové konstruktory

Typové konstanty

- Definicí dle šablony:

  `data` **Název_typu** `= Datové_konstruktory`

  zavádíme nový typ s označením `Název_typu`.
- `Název_typu` je nulární typový konstruktor, 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]**

Hodnoty a typy

Tvorba nových hodnot

- Aplikace datových 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 datové konstruktory!

```
'a' :: Char
[('a','a'),('a','a')] :: [(Char,Char)]          -- datové
[('a','a'), ('a','a')] :: [(Char,Char)]         -- typové
```

Příklad použití uživatelem definovaných typů

```
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)
```

Definice polymorfních typových konstruktorů

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 ... a_n** `= ...`

- Typové proměnné lze použít pro definici datových konstruktorů.

Kompletní obecná šablona

$$
\begin{array}{lll}
\texttt{data Tcons } \texttt{a}_1 \ \dots \ \texttt{a}_n & = & \texttt{Dcons}_1 \ \texttt{typ}_{(1,1)} \ \texttt{typ}_{(1,2)} \ \dots \ \texttt{typ}_{(1,arita_1)} \\
 & & \vdots \\
 & & \texttt{Dcons}_m \ \texttt{typ}_{(m,1)} \ \texttt{typ}_{(m,2)} \ \dots \ \texttt{typ}_{(m,arita_m)}
\end{array}
$$

Typový konstruktor `Maybe`

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` ?

Typový konstruktor Maybe

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)
```

Vstup/výstup

Vstup/výstup a referenční transparentnost

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.

Typy `IO()` , `IO a`

Vstup-výstupní akce

- Interakce programu s uživatelem nebo operačním systémem.
- Například výpis textu na terminálu, vytvoření nového souboru, načtení hodnoty proměnné prostředí, ...

Myšlenka

- Pro vstup-výstupní akce je zaveden speciální typ – **IO a** .
- Tento typ má formálně jednu jedinou textově nereprezentovatelnou hodnotu, a to **vstup-výstupní akci**.
- Výstupní akce mají typ **IO ()** .

  ```
  putStrLn "Ahoj!" :: IO ()
  ```

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

  ```
  getLine :: IO String
  ```

Vnitřní výsledek a vedlejší efekt

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 potom by nás zajímalo, kde je onen načtený řetězec?

Vnitřní výsledek a vedlejší efekt

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 potom by nás zajímalo, kde je onen 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`.

Operátor `>>=`

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)
```

Nesprávné použití operátoru `>>=`

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.
- Správné použití:

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

Řazení vstup-výstupních akcí

Funkce `return`

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

```
return ::  a -> IO a
return ['A','h','o','j'] >>= putStr
```

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`
- Příklady použití:

```
putStr "Jeje"  >> putChar '!'
getLine >> putStr "nic"
```

Základní funkce pro výstup

```
putChar :: Char -> IO ()
```

- Zapíše znak na standardní výstup.

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

- Zapíše řetězec na standardní výstup.

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

- Zapíše řetězec na standardní výstup a přídá 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.

Práce se soubory

```
type FilePath = String
```

- Definuje typový alias.

```
readFile ::  FilePath -> IO String
```

- Načte obsah souboru jako řetězec. Soubor je čten líně.

```
writeFile ::  FilePath -> String -> IO ()
```

- Zapíše 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` .

Další vstup-výstupní akce

Moduly System a Directory

- Existují další vstup-výstupní funkce pro práci s adresáři či systémovými proměnnými.
- Tyto funkce jsou předdefinovány v modulech `System` a `Directory`.

Použití modulu

- Moduly, jejichž funkce chceme použít, je třeba označit.
- Lze to učinit v souboru s globálními definicemi použitím klíčového slova `import`.
- Příklad:

```
import Char
import Directory
main = getDirectoryContents ".." >>=
       print .  map (\x -> (toUpper.head) x :  tail x)
```

Zápis pomocí `do`

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)
```

Funkce `sequence`

- Máme-li seznam vstup-výstupních akcí, můžeme je pomocí funkce `sequence` provést dávkově naráz.
- ```
  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
  ```
  ```

Funkce `mapM`

Funkce `mapM`

- Aplikuje unární funkci, jejíž výsledkem je vstup-výstupní akce, 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) [1,2] >>= print`
  po zadání dvou řádků s obsahem `radek1` a `radek2`
  vypíše `["radek1","radek2"]`

Domácí cvičení

Mentální cvičení

- Zdůvodněte (uvědomte si) proč je typ funkcí `foldr` a `foldl` takový, jaký je.
- Zadefinujte funkci `sequence` bez použití notace `do`.

Programování v Haskellu

- Definujte funkci, která pro 12 měsíčních platů zadaných seznamem vypočítá 15% daň z příjmu s přihlédnutím k tomu, že z celkové výše ročního příjmu se daní pouze část, která převyšuje nezdanitelné minimum ve výši 24 600 Kč.
- Vyhledejte popis funkcí obsažených v modulech `Char`, `Directory` a `IO` a vyzkoušejte je ve svých programech.
- Napište program, který vyzve uživatele, aby zadal 16 čísel oddělených mezerou, a poté tato čísla vypíše v matici velikosti 4x4.

IB015 Neimperativní programování

Redukční strategie, Seznamy program QuickCheck

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.

Základní redukční strategie

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

Líná redukční strategie

- Normální redukční strategii, při níž si pamatujeme hodnoty upravených podvýrazů a žádný s opakovaným výskytem nevyhodnocujeme více než jednou.
- Využívá referenční transparentnost.
- Nelze aplikovat na výrazy s vedlejším efektem.

Haskell

- Používá línou redukční strategii.
- Též označováno jako **líné vyhodnocování**.

Příklady vyhodnocování s různou strategií

Definice funkce

- `cube x = x * x * x`

Striktní redukční strategie

- `cube (3+5)` ⇝ `cube 8` ⇝ `8 * 8 * 8` ⇝ `64 * 8` ⇝ `512`

Normální redukční strategie

- `cube (3+5)` ⇝ `(3+5) * (3+5) * (3+5)` ⇝ `8 * (3+5) * (3+5)`
  ⇝ `8 * 8 * (3+5)` ⇝ `64 * (3+5)` ⇝ `64 * 8` ⇝ `512`

Líná redukční strategie

- `cube (3+5)` ⇝ `(3+5) * (3+5) * (3+5)` ⇝ `8 * 8 * 8` ⇝
  `64 * 8` ⇝ `512`

Vliv strategie na výsledek vyhodnocování

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 líném vyhodnocování k němu nedojde

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

Vliv strategie na výsledek vyhodnocování

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 líném vyhodnocování k zacyklení nedojde:

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

Obecné vlastnosti redukčních strategií

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 cyklit. Nebo je výpočet podle jedné strategie delší než podle jiné. Nikdy však **nemůže skončit dvěma různými výsledky**.

Obecné vlastnosti redukčních strategií

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.

Obecné vlastnosti redukčních strategií

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 líné/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.

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**.

Příklad

- Seznam nekonečně mnoha jedniček:

```
jednicky = 1 :  jednicky
```

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

$$\texttt{jednicky} \rightsquigarrow \texttt{1:jednicky} \rightsquigarrow \texttt{1:1:jednicky} \rightsquigarrow \cdots$$

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

$$\underline{\texttt{head jednicky}} = \texttt{head}\ \underline{\texttt{jednicky}} \rightsquigarrow \underline{\texttt{head (1:jednicky)}} \rightsquigarrow \texttt{1}$$

Příklady definice nekonečných seznamů

Nekonečný rostoucí seznam všech přirozených čísel

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

	0	1	2	3	4	5	...	nats
+	1	1	1	1	1	1	...	jednicky
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
0	1	1	2	3	5	8	13	...	

Funkce generující nekonečné seznamy

Nekonečné opakování jednoho prvku

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

Nekonečné opakování seznamu

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

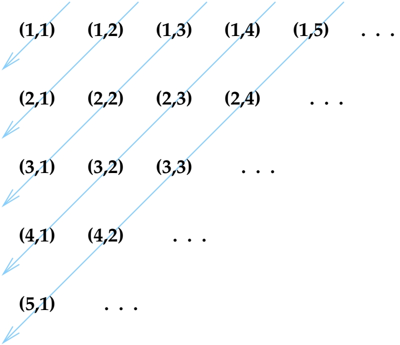
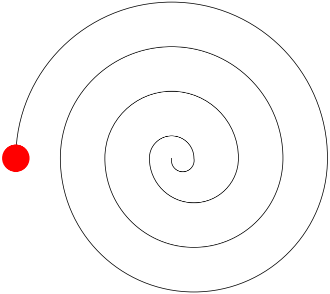
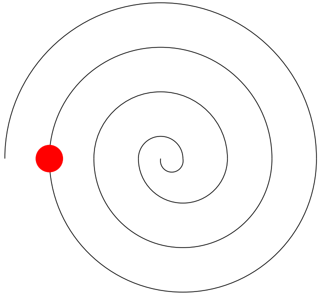
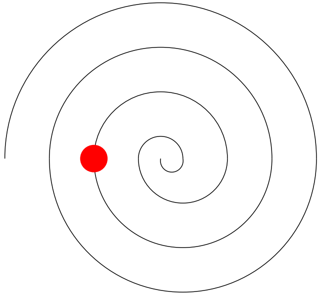
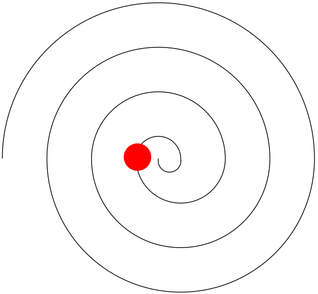
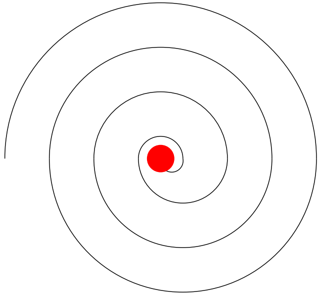
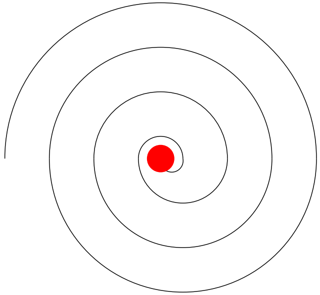
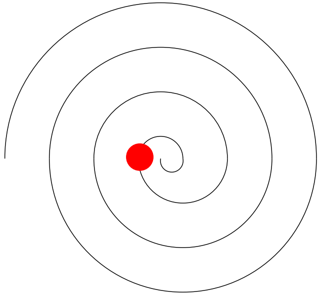
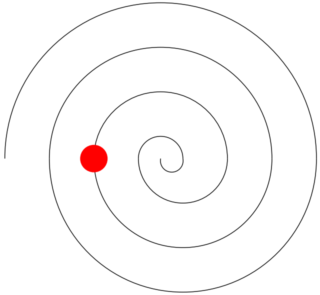
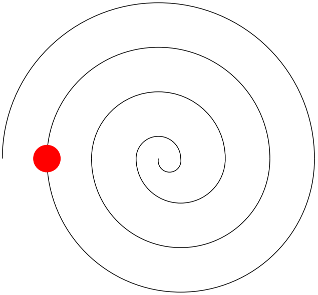
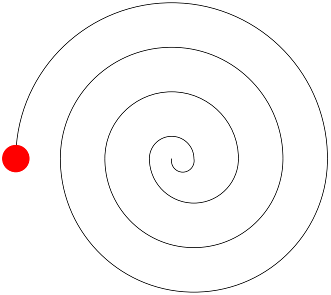
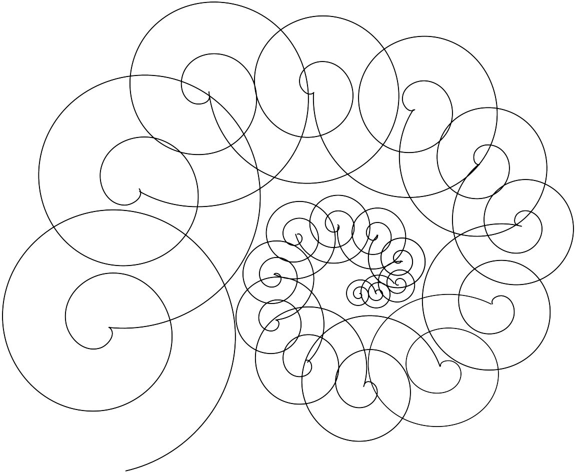
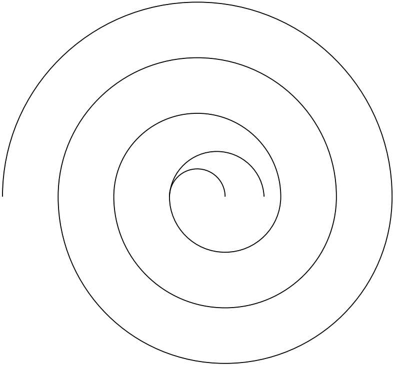
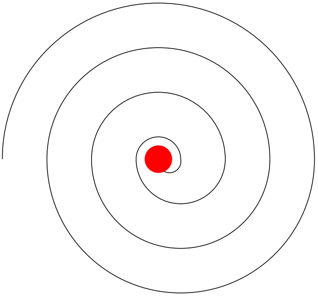
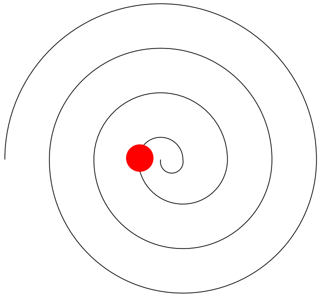
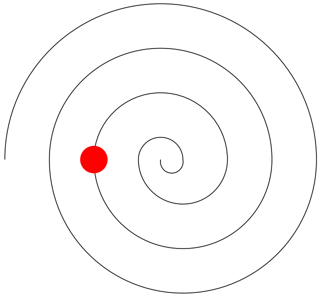
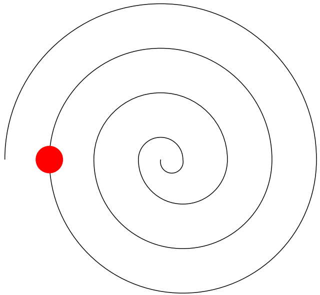
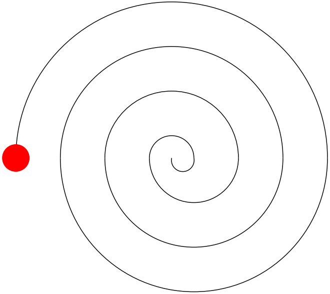
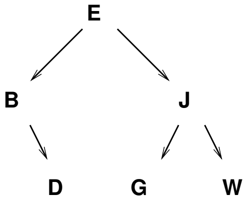
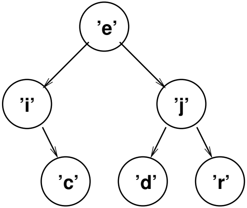
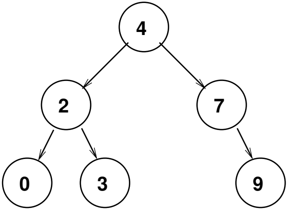
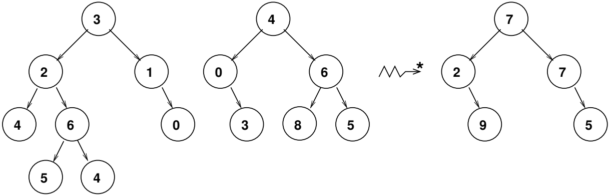
Opakovaná aplikace funkce

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

Příklady

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)` $\rightsquigarrow^*$
- `take 5 (iterate ('a':)  [])` $\rightsquigarrow^*$
- `take 10 (iterate (*(-1)) 1)` $\rightsquigarrow^*$
- `take 8 (cycle "Ha ")` $\rightsquigarrow^*$

Příklady

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

Zápis seznamů

Zápis seznamu výčtem

Prostý výčet

- Zápis pomocí základních datových 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` $\rightsquigarrow^*$ `[1,2,3,4,5,6,7,8,9,10,11,12]`

  `enumFromTo 'A' 'Z'` $\rightsquigarrow^*$ `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`

- Všechny uspořádatelné typy jsou enumerovatelné.

Enumerační funkce a syntaktické zkratky

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)` $\leadsto^*$ `[0,2,4,6,8,10,12,14,16,18]`

`enumFromThenTo 0 3 15` $\leadsto^*$ `[0,3,6,9,12,15]`

Přehled enumeračních funkcí a syntaktických zkratek

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

Intenzionální zápis seznamu

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 | generátor 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**.

Kvalifikátory – Generátory

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.
- Nejvyšší váhu má generátor vlevo, směrem doprava váha klesá.

Kvalifikátory – Predikáty a lokální definice

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 proměnou použitelnou buď v definičním výrazu nebo v libovolném kvalifikátoru vyskytujícím se vpravo.
- Výraz může využít proměnné definované vlevo.

Příklady 1/3

- `[ n^2 | n <-[0..3] ]`

  $\rightsquigarrow^*$ `[0,1,4,9]`

- `[ (c,k) | c<-"abc", k<-[1,2] ]`

  $\rightsquigarrow^*$ `[('a',1),('a',2),('b',1),('b',2),('c',1),('c',2)]`

- `[ 3*n | n<-[0..6], odd n ]`

  $\rightsquigarrow^*$ `[3,9,15]`

- `[ (m,n) | m<-[1..3], n<-[1..3], n<=m ]`

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

- `[ (m,n) | m<-[1..3], n<-[1..m] ]`

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

- `[ (x,y) | z<-[0..2], x<-[0..z], let y=z-x ]`

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

Příklady 2/3

- `[ replicate n c | c<-"xyz", n<-[2,3] ]`

  $\leadsto^*$ `["xx","xxx","yy","yyy","zz","zzz"]`

- `[ replicate n c | n<-[2,3], c<-"xyz" ]`

  $\leadsto^*$ `["xx","yy","zz","xxx","yyy","zzz"]`

- `[ x^2 | [x]<-[[],[2,3],[4],[1,1..],[],[7],[0..]]  ]`

  $\leadsto^*$ `[16,49]`

- `[ 0 | []<-[[],[2,3],[4],[0..],[],[5]] ]`

  $\leadsto^*$ `[0,0]`

- `[ x^3 | x<-[0..10], odd x ]`

  $\leadsto^*$ `[1,27,125,343,729]`

- `[ x^3 | x<-[0..10], odd x, x < 1 ]`

  $\leadsto^*$ `[]`

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		7				11		13				17		19				23		25				29
					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
																											29

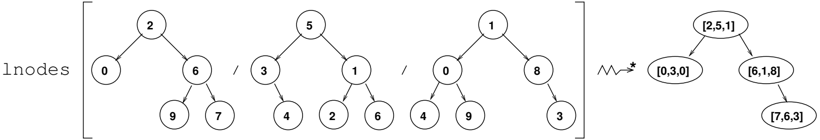
Prvočísla

- 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..]
  ```
  ```

QuickCheck

Testování programů programem QuickCheck

Myšlenka programu QuickCheck

- Generování náhodných hodnot.
- Testování chování programu na daném počtu těchto hodnot.

Testovaná vlastnost

- **Unární funkce**, která vrací hodnotu typu `Bool` .
- Hodnota `True` indikuje správný výsledek, `False` nesprávný.
- Může volat jiné funkce.

Standardní použití

```
import Test.QuickCheck
quickCheck (\z -> muj_program z == ocekavany_vysledek)
```

Testované vlastnosti

- `prop1 :: Int -> Bool`
  `prop1 x = (x+1)*(x+1) == x*x + 2*x + 1`
- `prop2 :: Float -> Bool`
  `prop2 x = (x+1)*(x+1) == x*x + 2*x + 1`

Použití programu quickCheck v interpretru

- `quickCheck prop1`
  `OK, passed 100 tests.`
- `quickCheck prop2`
  `Falsifiable, after 9 tests:`
  `-2.166667`

Pokročilé použití programu QuickCheck

Počet testů

- Přednastavený počet testů může být nedostatečný.
- Definice procedury s větším počtem testů:

```
myCheck = quickCheckWith stdArgs { maxSuccess = 5000 }
```

- Použití nové testovací procedury:

```
myCheck (\z->z/='a')
Falsifiable, after 212 tests:
'a'
```

"Ukecané" testování

- Při testování vypisuje použité hodnoty

```
verboseCheck (\z->z/="aa")
...
```

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.

Domácí cvičení 2/2

QuickCheck

- Programem QuickCheck ověřte, že dvojice $(3, 4)$ a $(4, 3)$ jsou v seznamu uvedeny ve správném pořadí.
- Programem QuickCheck ověřte, že náhodně generované dvojice dvojic přirozených čísel jsou v seznamu uvedeny ve správném pořadí.

  Nápověda: dvojice s čísly < 1 automaticky vyhovují testu.

Verze s lexikografickým uspořádáním

- Opakujte celé zadání s tím, že dvojice jsou v seznamu uspořádány lexikograficky:

$$(a, b) < (c, d) \Longleftrightarrow (a < c) \vee (a = c \wedge b < d)$$

- Vysvětlete nastalý problém s testováním.

IB015 Neimperativní programování

A zase ta REKURZE ...
(Rekurze a indukce, Rekurzivní datové typy)

Jiří Barnat
Libor Škarvada

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
```

Rekurze a zacyklení výpočtu

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 cyklí**.

Příklad

- Nesprávné použití rekurze ve funkci `length'` :

```
length' :: [a] -> Integer
length' [] = 0
length' x = length' x
```

- Při aplikaci `length'` na neprázdný seznam výpočet cyklí.
- Chybu neodhalí typová kontrola, definice je typově správně.

Kruh versus spirála

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
```



Kruh versus spirála

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
```



Kruh versus spirála

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
```



Kruh versus spirála

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
```



Kruh versus spirála

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
```



Kruh versus spirála

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
```



Kruh versus spirála

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
```



Kruh versus spirála

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
```



Kruh versus spirála

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
```



Rekurzivní definice funkcí

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 zbývající části 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.

Definice rekurzivní funkce

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 odebírání 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 rekurzivní funkce s více bázovými případy

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]` $\leadsto^*$ `[1,3,5,7]`
- `oddMembers "Trol ej ej schomoula."` $\leadsto^*$ `"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

Stejné principy rekurze

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**.

Seznam

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]
```



Seznam

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]
```



Rekurzivní datové struktury

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.

Binární stromy

Rekurzivní definice binárního stromu

- Prázdný strom je **binární strom**
- Hodnota 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.

Binární stromy v Haskellu

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))
```



Funkce pro práci s rekurzivními datovými strukturami

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.

Definice funkce `treezipwith`

- ```
  treezipwith :: (a -> b -> c) -> BinTree a -> BinTree b -> BinTree c
  treezipwith op (Node v1 l1 r1) (Node v2 l2 r2)
              = Node (v1 `op` v2) (treezipwith op l1 l2)
                                  (treezipwith op r1 r2)
  treezipwith _ _ _ = Empty
  ```

Příklad

treezipwith (+)


  ```

Popis funkce `lnodes`

- Funkce `lnodes` vytvoří ze seznamu stromů jeden strom seznamů, tj. strom, jehož struktura bude průnikem všech stromů ze seznamu a v jehož uzlech budou seznamy hodnot z uzlů na odpovídajících pozicích.

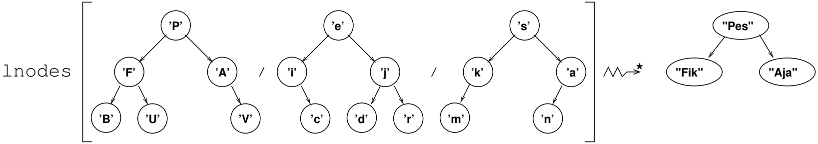
Příklad



Definice funkce `lnodes`

```
lnodes ::  [ BinTree a ] -> BinTree [a]
lnodes = foldr (treezipwith (:))  niltree
   where niltree = Node [] niltree niltree
```

Příklad



Dokazování rekurzivních programů

Dokazování správnosti programů

Fakta

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

Důkaz koreknosti algoritmu

- Dokazujeme, že pokud výpočet algoritmu na platných vstupech skončí, tak algoritmus vrací 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**.

Rekurze a matematická indukce

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.

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

Příklad

Pozorování

- Klíčovým problémem při použití matematické indukce je identifikace toho, podle čeho má být indukce vedena.
- Napovědět může místo rekurzivního volání funkce, neboť rekurze 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)
```

Příklad

Pozorování

- Klíčovým problémem při použití matematické indukce je identifikace toho, podle čeho má být indukce vedena.
- Napovědět může místo rekurzivního volání funkce, neboť rekurze 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.**
  ```

Příklad – Bázový krok

Bázový krok, T(0)

- Nechť y=0, a nechť x je libovolné.
- `fpow x y` se redukuje dle `fpow x 0 = 1` na hodnotu `1` .
- $x^0 = 1$, pro libovolné x.
- Tudíž pro $y = 0$ tvrzení platí.

Příklad – Indukční krok

Indukční krok, T(i)=>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 `fpow x i` $\rightsquigarrow^* x^i$ pro libovolnou hodnotu x.
- `fpow x (i+1)`

  $\rightsquigarrow$ `x * fpow x (i+1-1)`

  $\rightsquigarrow$ `x * fpow x i` $\stackrel{dleIP}{=}$ `x *` x^i

  $\rightsquigarrow$ x^{i+1}
- Ukázali jsme, že pokud tvrzení platí pro i, pak platí i pro $i + 1$.
- Z platnoti bázového kroku a vlastností matematické indukce plyne, že pro libovolnou hodnotu x tvrzení platí pro všechny hodnoty y.

Další příklady

Věta 1

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

```
length (s ++ t) = (length s) + (length t).
```

- Důkaz veden indukcí podle délky seznamu s.

Věta 2

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

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

- Důkaz veden indukcí podle délky seznamu s.

Věta 3

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

```
take m s ++ drop m s = s.
```

- Důkaz veden indukcí podle m.

Mentální cvičení

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)
```

IB015 Neimperativní programování

Časová složitost, Typové třídy, Moduly

Jiří Barnat
Libor Škarvada

Časová složitost

Časová složitost algoritmu

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.

Časová složitost algoritmu

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.

Na délce záleží!

Funkce pro reverzi seznamu

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
  ```
  ```

Reverze 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]
⇝   (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])
```

Reverze seznamu funkcí `reverse’`

- 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])
```

Reverze seznamu funkcí `reverse'`

- 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])
```

Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky *n*.
- n+1 + 1 + 2 **+ 3 + … + 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])
```

Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky *n*.
- n+1 + 1 + 2 + 3 + ... + 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])
```

Reverze seznamu funkcí `reverse`

```
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]
```

Reverze seznamu funkcí `reverse`

- 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]
```

Reverze seznamu funkcí `reverse`

- 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]
```

Reverze seznamu funkcí `reverse`

- 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]
```

Reverze seznamu funkcí `reverse`

- 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 multiplikativní konstantu.
- Podle toho hovoříme o funkcích lineárních, kvadratických, exponenciálních apod.

Přehled asymptotických funkcí

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

reverse‘

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

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

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

reverse

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

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

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

Časová složitost algoritmu a problému

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.

Počítání mocniny

Definice funkcí

```
mocnina' :: Int -> Int -> Int
mocnina' m 0 = 1
mocnina' m n = m * mocnina' m (n-1)

mocnina :: Int -> Int -> Int
mocnina m 0 = 1
mocnina m n = if even n then r else m * r
          where r = mocnina (m * m) (n `div` 2)
```

Složitost výpočtu vzhledem k exponentu

- Složitost funkce `mocnina'` je **lineární**.
- Složitost funkce `mocnina` je **logaritmická**.

Výpočet Fibonacciho číselné řady

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**.

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 [x`$_1$`, ..., x`$_n$`]` je různý.
- Nejkratší výpočet má délku 3:

  `ins 1 [2,4,6,8]` $\leadsto^{3}$ `[1,2,4,6,8]`
- Nejdelší výpočet má délku $3n + 1$:

  `ins 9 [2,4,6,8]` $\leadsto^{3*4+1}$ `[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 funkcí pro uspořádání prvků v seznamu pomocí postupného zařazování.

```
insSort :: Ord a => [a] -> [a]
insSort = 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 `insSort`

$$
\begin{array}{ll}
 & \texttt{insSort [}x_1\texttt{, }x_2\texttt{, ..., }x_{n-1}\texttt{, }x_n\texttt{]} \\
\rightsquigarrow & \texttt{foldr ins [] [}x_1\texttt{, }x_2\texttt{, ..., }x_{n-1}\texttt{, }x_n\texttt{]} \\
\rightsquigarrow^{n+1} & \texttt{ins }x_1\texttt{ (ins }x_2\texttt{ (...(ins }x_{n-1}\texttt{ (ins }x_n\texttt{ []))...))}
\end{array}
$$

Příklad závislosti časové složitosti na redukční strategii

Definice funkce

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

Striktní vyhodnocování (nejhorší případ – seznam je klesající)

$$
\begin{array}{ll}
 & \texttt{minim [x}_1\texttt{, ..., x}_n\texttt{]} \\
\leadsto & \texttt{(head.inssort) [x}_1\texttt{,...,x}_n\texttt{]} \\
\leadsto & \texttt{head (inssort [x}_1\texttt{,...,x}_n\texttt{])} \\
\leadsto & \texttt{head (foldr ins [] [x}_1\texttt{,...,x}_n\texttt{])} \\
\leadsto^{n+1} & \texttt{head (ins x}_1\texttt{ (...(ins x}_{n-2}\texttt{ (ins x}_{n-1}\texttt{ (ins x}_n\texttt{[]))))...))} \\
\leadsto^{3\cdot 0+1} & \texttt{head (ins x}_1\texttt{ (...(ins x}_{n-2}\texttt{ (ins x}_{n-1}\texttt{ [x}_n\texttt{]))...))} \\
\leadsto^{3\cdot 1+1} & \texttt{head (ins x}_1\texttt{ (...(ins x}_{n-2}\texttt{ [x}_n\texttt{,x}_{n-1}\texttt{])...))} \\
\leadsto^{3\cdot 2+1} & \texttt{head (ins x}_1\texttt{ (...[x}_n\texttt{,x}_{n-1}\texttt{,x}_{n-2}\texttt{]...))} \\
\vdots & \\
\leadsto^{3\cdot (n-2)+1} & \texttt{head (ins x}_1\texttt{ [x}_n\texttt{,...,x}_2\texttt{])} \\
\leadsto^{3\cdot (n-1)+1} & \texttt{head [x}_n\texttt{,...,x}_1\texttt{]} \\
\leadsto & \texttt{x}_n
\end{array}
$$
  ```

Příklad závislosti časové složitosti na redukční strategii

Definice funkce

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

Líné vyhodnocování (nejhorší případ – nejmenší prvek na konci)

$$
\begin{array}{ll}
 & \texttt{minim [}x_1\texttt{,...,}x_n\texttt{]} \\
\rightsquigarrow & \texttt{(head.inssort) [}x_1\texttt{,...,}x_n\texttt{]} \\
\rightsquigarrow & \texttt{head (inssort [}x_1\texttt{,...,}x_n\texttt{])} \\
\rightsquigarrow & \texttt{head (foldr ins [] [}x_1\texttt{,...,}x_n\texttt{])} \\
\rightsquigarrow^{n+1} & \texttt{head (ins } x_1 \texttt{ (...(ins } x_{n-2} \texttt{ (ins } x_{n-1} \texttt{ (ins } x_n \texttt{ [])))...))} \\
\rightsquigarrow & \texttt{head (ins } x_1 \texttt{ (...(ins } x_{n-2} \texttt{ (ins } x_{n-1} \texttt{ (} x_n \texttt{ : [])))...))} \\
\rightsquigarrow^{3} & \texttt{head (ins } x_1 \texttt{ (...(ins } x_{n-2} \texttt{ (} x_n \texttt{ :(ins } x_{n-1} \texttt{ [])))...))} \\
\rightsquigarrow^{3} & \texttt{head (ins } x_1 \texttt{ (...(} x_n \texttt{ : (ins } x_{n-2} \texttt{ (ins } x_{n-1} \texttt{ [])))...))} \\
\vdots & \\
\rightsquigarrow^{3} & \texttt{head (ins } x_1 \texttt{ (} x_n \texttt{ :(ins } x_2 \texttt{ (ins } x_3 \texttt{ (...(ins } x_{n-1} \texttt{ [])...)))))} \\
\rightsquigarrow^{3} & \texttt{head (} x_n \texttt{ :(ins } x_1 \texttt{ (ins } x_2 \texttt{ (... (ins } x_{n-1} \texttt{ [])...))))} \\
\rightsquigarrow & x_n
\end{array}
$$

Příklad závislosti časové složitosti na redukční strategii

Striktní vyhodnocování

- Počet redukčních kroků výrazu `minim [x1,...,xn]` 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 `minim [x1,...,xn]` je:

$$3 + n + 1 + 1 + 3.(n-1) + 1 = 4n + 3$$

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

Vliv redukční strategie na 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 = const n (fib' n)
```

- Lineární líně i striktně:

  `length` $[\mathtt{a}_1,\ldots,\mathtt{a}_n]$

Typové třídy

Připomenutí klasifikace typů

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

Význam

- Identifikují společné vlastnosti různých typů.
- Umožňují definici funkcí polymorfních typů zúžených na typy požadovaných vlastností.

Programátorský význam

- Definice a použití typových tříd umožňují 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.

Příklad typové třídy a jejího využití

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
  ```

Využití typové třídy jinou typovou třídou

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)
  ```
  ```

Přenos vlastností typu na složený typ

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)
  ```
  ```

Obecná definice typové třídy a deklarace instance

Definice typové třídy

- `class` $\left[\ (C_1\ a, \ldots, C_k\ a)\ \texttt{=>}\ \right]\ C\ a$

$$\left[\begin{array}{ll} \texttt{where} & op_1 :: typ_1 \\ & \vdots \\ & op_n :: typ_n \\ & \left[\begin{array}{l} default_1 \\ \vdots \\ default_m \end{array}\right] \end{array}\right]$$

Deklarace instance

- `instance` $\left[\ (C_1\ a_1, \ldots, C_k\ a_k)\ \texttt{=>}\ \right]\ C\ typ$

$$\left[\begin{array}{ll} \texttt{where} & valdef_1 \\ & \vdots \\ & valdef_n \end{array}\right]$$

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.

Příklad přetížení operace

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)
  ```
  ```

Moduly a modulární návrh programů

Modulární návrh

Motivace

- Oddělení nezávislých, znovupoužitelných, logicky ucelených částí kódu do separátních celků – **modulů**.

Zapouzdření

- Při definici modulu je nutné explicitně vyjmenovat funkce, které mají být viditelné a použitelné mimo rozsah modulu, tzv. **exportované** funkce.
- Ostatní funkce a datové typy definované v modulu nejsou z vnějšku modulu viditelné.
- Moduly by měly exportovat jen to, co je nutné.
- Modul může exportovat hodnoty, typy a typové konstruktory, typové a konstruktorové třídy, jména modulů.

Definice Modulu v Haskellu

Obecná definice

- [module Jméno [(export$_1$, ..., export$_n$)] where]

  [import M$_1$ [spec$_1$]

  ⋮

  import M$_m$ [spec$_m$]]

  [globální_deklarace]

Automatické doplnění definice

- Není-li uvedena hlavička, doplní se

  `module Main (main) where`

- Nevyskytuje-li se mezi importovanými moduly M$_1$,...M$_m$ modul `Prelude`, doplní se

  `import Prelude`

Main

Hlavní funkce

- Program musí mít definovanou hlavní funkci – funkci `main` .
- Právě jeden modul v programu musí být `Main` .

Modul `Main`

- Modul `Main` musí exportovat hodnotu

  `main :: IO` τ

  pro nějaký typ τ, (obvykle $\tau =$ `()`).

Příklad modulu – Datový typ Fifo

Datový typ Fifo

- Datový kontejner (struktura, která uchovává prvky) přistupovaný 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) [])
```

IB015 Neimperativní programování

Ukázky funkcionálně řešených problémů

Jiří Barnat
Libor Škarvada

Užitečné konstrukce Haskellu

Aplikační operátor $

Pozorování

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

Aplikační operátor $

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

Aplikační operátor $ – příklady

Ekvivalentní zápisy pomocí operátoru $

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

Příklady

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

Pojmenovaný vzor – symbol `@`

Použití symbolu `@`

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

Příklady

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

Syntaktická konstrukce case

Pozorování

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

Syntaktická konstrukce `case`

- ```
  case expression of
      pattern₁ -> expression₁
      ...
      patternₙ -> expressionₙ
  ```
- Textové zarovnání vzorů je nutné.
- Všechny výrazy na pravých stranách musí být stejného typu.
  ```

Syntaktická konstrukce case – příklady

Úkol 1

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

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

Úkol 2

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

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

Strážené definice funkcí

Stráž

- Stráž je výraz typu Bool přidružený k definičnímu přiřazení.
- Při výpočtu bude tato definice funkce realizována, pouze pokud bude asociovaný výraz vyhodnocen na `True` .
- ```
  function args
          | guard1 = expression1
          ...
          | guardn = expressionn
          | otherwise = default expression
  ```

Pozorování

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

Strážené definice funkcí – příklady

Příklad 1

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

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

Příklad 2

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

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

Použití akumulátoru

Akumulátor a rekurze

Obecný princip

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

Pozorování

- Přímé použití rekurzivní funkce má tendenci být čitelnější.

Akumulátor a rekurze - připomenutí

Pozorování

- Neefektivita opakovaným voláním rekurzivní funkce

Připomeňte si

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

Akumulátor a rekurze - připomenutí

Pozorování

- Neefektivita přímým použitím rekurzivní funkce.

Připomeňte si

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

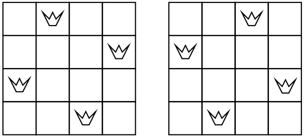
Prohledávání a prořezávání

Problém n dam

Problém n dam

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

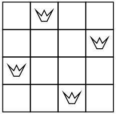
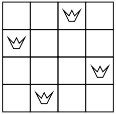
Všechna řešení pro $n = 4$

- 

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

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

Úkol

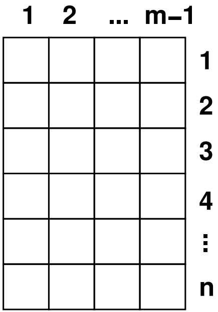
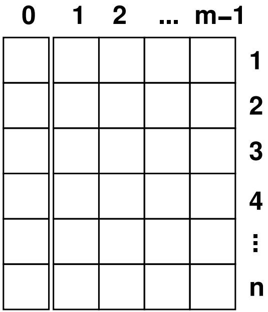
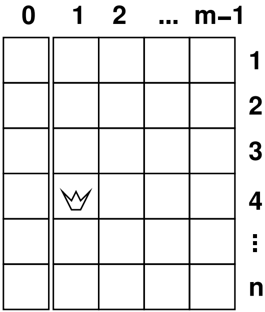
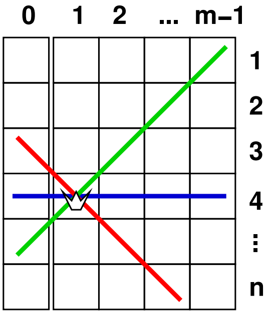
- Naprogramujte funkci `damy :: Int -> [[Int]]` , která pro zadané n vrátí seznam všech možných řešení.

Problém *n* dam – myšlenka řešení

Postupné rozšiřování šachovnice

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

Cesta k celkovému řešení

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

Problém *n* dam – pomocná funkce

Účel

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

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

Princip

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



Problém *n* dam – pomocná funkce

Účel

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

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

Princip

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



Problém *n* dam – pomocná funkce

Účel

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

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

Princip

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



Problém *n* dam – pomocná funkce

Účel

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

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

Princip

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


  ```

Problém *n* dam – pomocná funkce

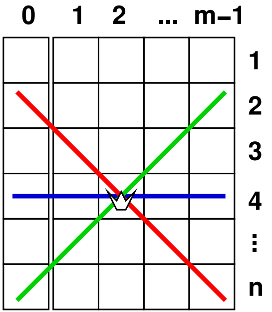
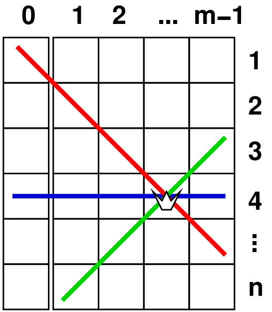
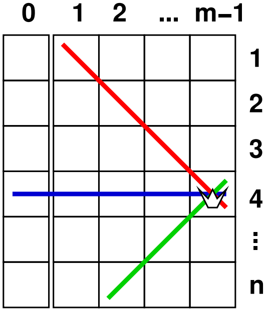
Účel

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

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

Princip

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


  ```

Účel

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

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

Princip

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



Problém *n* dam – pomocná funkce

Účel

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

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

Princip

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


  ```

Problém *n* dam – pomocná funkce

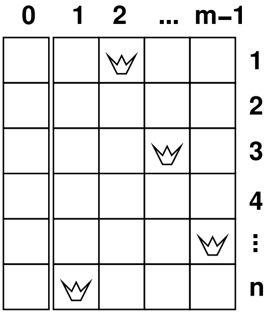
Účel

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

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

Princip

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



Problém *n* dam – pomocná funkce

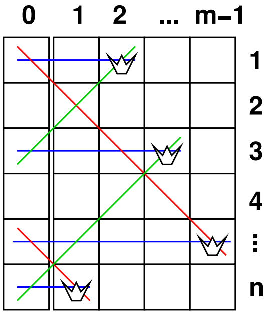
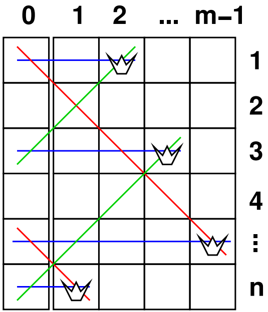
Účel

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

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

Princip

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


  ```

Problém *n* dam – pomocná funkce

Účel

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

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

Princip

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


  ```

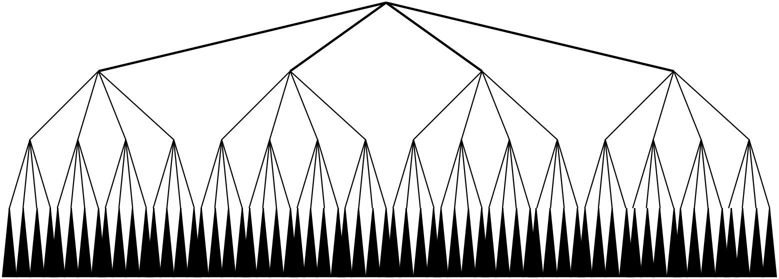
Problém n dam – řešení

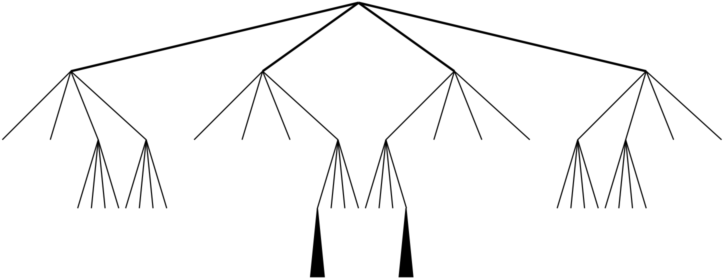
```
type Reseni = [Int]

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

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

Backtracking, Prožezávání

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

Problém n dam – efekt prořezávání (n=4)





Nejmenší nepoužité přirozené číslo

(Richard Bird: Pearls of Functional Algorithm Design)

Nejmenší nepoužité přirozené číslo

Zadání

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

Řešení 1.

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

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

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

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

`minfree` – kvadratické

Funkce `listminus`

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

Funkce `listminus`

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

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

`minfree` – kvadratické

Funkce `listminus`

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

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

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]`

`minfree` – kvadratické

Funkce `listminus`

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

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

minfree – kvadratické

Funkce `listminus`

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

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

`minfree` – kvadratické

Funkce `listminus`

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

Vlastnosti řešení

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

Otázka

- Je možné problém řešit s lineární časovou složitostí?

`minfree` – lépe než kvadraticky

Pozorování

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

Idea lepšího řešení

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

Pozorování

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

Idea lepšího řešení

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

`minfree` – lépe než kvadraticky

Pozorování

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

Idea lepšího řešení

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

minfree – lépe než kvadraticky

Pozorování

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

Idea lepšího řešení

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

Pozorování

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

Idea lepšího řešení

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

`minfree` – lépe než kvadraticky

Pozorování

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

Idea lepšího řešení

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

  `[0,3,1,4]`

`minfree` – lépe než kvadraticky

Pozorování

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

Idea lepšího řešení

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

Předpoklady

- Konstantní operace pro přístup k hodnotám omezeně dlouhého seznamu.

Haskell efektivně, aneb od seznamů k polím

Pole v Haskellu

Pole

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

Poznámka:

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

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

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

Pole v Haskellu

Použití

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

Typový konstruktor `Array`

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

Příklad hodnoty a typu

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

Pole v Haskellu – indexace

Typová třída `Ix`

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

Meze pole

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

Pole v Haskellu – meze (příklady)

Instance třídy Ix

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

Příklad 1

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

Příklad 2

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

Pole v Haskellu – přístup k prvkům

Přístupy k prvkům pole

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

Příklady přístupů k prvkům

- `array (5,6) [(5,"ANO!"),(6,"NE!")] ! 5`

  $\leadsto^*$ `"ANO!"`
- `array ('D','E') [('D',2014),('E',1977)] ! 'D'`

  $\leadsto^*$ `2014`
- `array (1,2) [(1,1),(2,2)] // [(1,3),(2,3)]`

  $\leadsto^*$ `array (1,2) [(1,3),(2,3)]`

Pole v Haskellu – Opearace s poli

Meze pole

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

Seznam indexů pole

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

Převody mezi poli a seznamy

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

Převody mezi poli a seznamy dvojic

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

accumArray

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

Příklady

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

Nejmenší nepoužité přirozené číslo II

(Richard Bird: Pearls of Functional Algorithm Design)

minfree – lineárně

Postup

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

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

- Nebudeme uvažovat čísla mimo očekávaný rozsah.

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

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

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

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

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

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

Mezivýsledek

- `checklist s` $\rightsquigarrow^*$

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

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
```

minfree – lineárně (řešení)

Řešení

- `minfree = search . checklist`

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

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

Složitost algoritmu

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

Lineární řazení některých seznamů

(Richard Bird: Pearls of Functional Algorithm Design)

Lineární řazení čísel z omezeného rozsahu

Pozorování

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

Algoritmus

- ```
  max = 1000

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

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

Příklad

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

USS Haskell

IB016 – Placená reklama

Zaujal vás Haskell?

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

IB016 Seminář z funkcionálního programování

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

IB015 Neimperativní programování

Neimperativní programování v Prologu

Jiří Barnat

Logické programování

- Deklarativní programovací paradigma
- Mechanická dedukce nových informací na základě uvedených údajů a jejich vzájemných vztahů.
- Výpočet = logické dokazování.
- Nejpoužívanější (jediný) programovací jazyk

Prolog

Logické programování

- Deklarativní programovací paradigma
- Mechanická dedukce nových informací na základě uvedených údajů a jejich vzájemných vztahů.
- Výpočet = logické dokazování.
- Nejpoužívanější (jediný) programovací jazyk

Prolog

- Třešnička na dortu neimperativního programování.

Četnost, oblasti a způsoby použití

Četnost použití Prologu

- Logické programování se masově nerozšířilo.
- Okrajový, specificky používaný programovací prostředek.
- `http://langpop.com/` (2011) – Prolog ani není uveden.

Typické oblasti použití Prologu

- Umělá inteligence, zpracování jazyka, rozvrhování.
- Řešení deklarativně zadaných úloh.

Moderní způsoby použití Prologu

- Vložené použití ve větší (i imperativní) aplikaci.
- Deduktivní databáze.

Logické výpočetní paradigma

- program = databáze faktů a pravidel + cíl
- výpočet = dokazování cíle metodou unifikace a SLD rezoluce
- výsledek = pravda/nepravda, případně seznam hodnot volných proměnných, pro které je cíl pravdivý vzhledem k databázi faktů a pravidel.

Příklad programu

- databáze faktů a pravidel

```
pocasi(prsi).
stav(mokro) :- pocasi(prsi).
```

- cíl a výsledek

```
?-stav(mokro).                ?-stav(X).
true.                         X = mokro.
```

Implementace Prologu

SICStus Prolog

- "State-of-the-art" implementace prologu.
- Komerční produkt.
- `http://sicstus.sics.se/`

SWI-Prolog

- Relativně úplná a kompatibilní implementace.
- Freeware.
- `http://swi-prolog.org`

A další ...

- YAP: Yet Another Prolog
  `http://www.dcc.fc.up.pt/~vsc/Yap/`

Úvodní intuitivní příklady

Interaktivní uživatelské prostředí Prologu

SWI-Prolog

- Spouštěno příkazem `swipl`.

Textové interaktivní prostředí

- Standardní výzva: `?-`
- Veškeré povely uživatele musí být zakončeny tečkou.

Základní povely

- Ukončení prostředí: `halt.`
- Nápověda: `help.`
- Načtení souboru `jmeno.pl`: `consult(jmeno).`
- Též alternativně: `[jmeno].`

Notace databáze fakt

Struktura jednoduchých příkladů

- Databáze fakt a pravidel uvedena v externím souboru.
- Dotazy kladené skrze interpreter.
- Přípona souboru `.pl` nebo `.pro`.

Notace fakt

- Fakta začínají malým písmenem a končí tečkou.
- Fakty 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 */ .`

Jednoduché dotazy na fakta a relace

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.
  ```
  ```

Dotazy s využitím proměnných

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 */
```

Databáze s pravidly

Pravidla v databázi

- Zápis: `clovek(X) :- zena(X).`
- Význam: Pokud platí `zena(X)`, pak platí `clovek(X)`.

Disjunkce

- Zápis: `clovek(X) :- zena(X); muz(X).`
- Alternativní zápis:

```
clovek(X) :- zena(X).
clovek(X) :- muz(X).
```

- Význam: $(\mathit{zena}(X) \vee \mathit{muz}(X)) \implies \mathit{clovek}(X)$.

Konjunkce

- Zápis: `unikat(X) :- zena(X), muz(X).`
- Význam: $(\mathit{zena}(X) \wedge \mathit{muz}(X)) \implies \mathit{unikat}(X)$.

Dotazy na databáze s pravidly

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.
  ```
  ```

Dotazy na databáze s pravidly – lokalita proměnných

Rozsah platnosti proměnných

- Použití proměnné je lokalizováno na dané pravidlo.

Příklad

- ```
  clovek(X) :- zena(X); muz(X).
  unikat(X) :- zena(X), muz(X).
  zena(bozena_nemcova).
  zena(serena_will).
  muz(jara_cimrman).
  muz(serena_will).
  ```

- ```
  ?- clovek(X).
  X = bozena_nemcova ;
  X = serena_will ;
  X = jara_cimrman ;
  X = serena_will.
  ```

  ```
  ?- unikat(X).
  X = serena_will.
  ```

Termy – Základní stavební kameny

Syntax Prologu

Pozorování

- Fakta, pravidla a dotazy jsou tvořeny z termů.

Termy

- Atomy.
- Čísla.
- Proměnné.
- Strukturované termy.

Atomy

- Řetězce začínající malým písmenem, obsahující písmena číslice a znak podtržítko.
- 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 atom.

Objekty Prologu – Čísla

Čísla

- 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.

Objekty Prologu – Strukturované termy

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

Unifikace

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`.

Příklady

- ```
  ?- =(slovo,slovo).
  true.

  ?- =(slovo,X).
  X = slovo.
  ```

  ```
  ?- a(A,[ble,ble]) = a(b(c(d)),B).
  A = b(c(d)),
  B = [ble, ble].
  ```
  ```

Algoritmus unifikace

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 instaciována 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 instanciace 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.`

Unifikace rekurzivních výrazů

Příklad

- Uvažme následující dotaz na Prolog:

```
?- X = 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 algorimu na přechozím slajdu, však k unifikaci dojde:

```
?- X = otec(X).
X = otec(X).
```

Poznámka

- Některé implementace Prologu mohou při této unifikaci cyklit.

Unifikace s kontrolou sebevýskytu.

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

Výpočet v Prologu

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

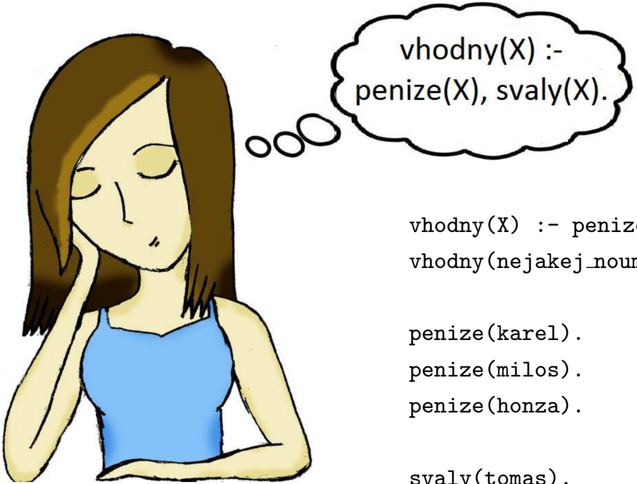
- Princip výpočtu s využitím příkladů a neformální demonstrace postupu bez intelektuální zátěže odpovídajícího teoretického fundamentu.

Pořadí faktů v databázi

Pozorování

- Při výpočtu Prolog vždy využívá fakta v tom pořadí, v jakém jsou uvedeny v programu.

Příklad

- ```
  players(flink,gta5).
  players(flink,world_of_tanks).
  players(flink,master_of_orion).
  ```
- ```
  ?- players(flink,X).
  X = gta5 ;                          /* uživatel vložil ; */
  X = world_of_tanks ;                /* uživatel vložil ; */
  X = master_of_orion.
  ```
- ```
  ?- players(flink,X).
  X = gta5 .                          /* 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).

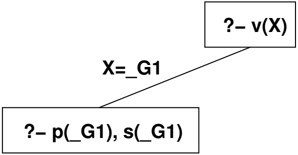
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
```

?– v(X)

Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

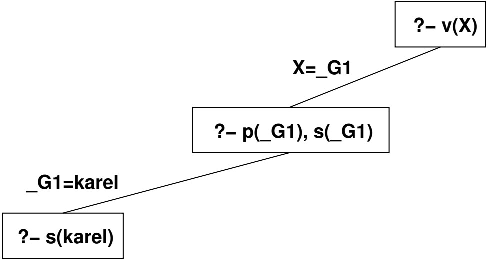
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
```



Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

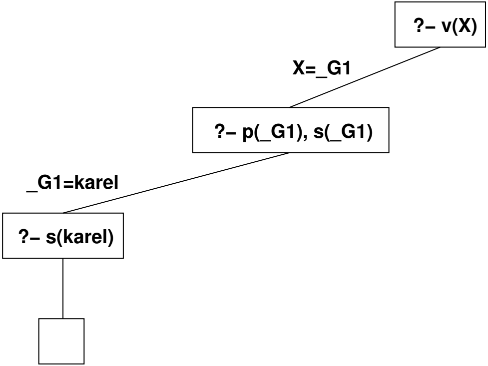
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
```



Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

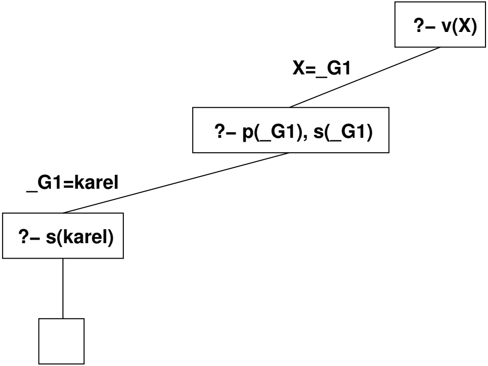
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
```



```
v(X) :- p(X), s(X).
v(nouma).

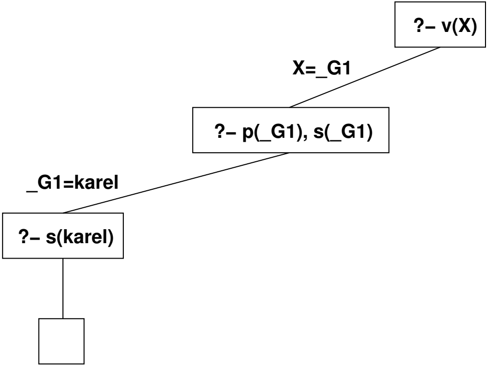
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
karel
```



Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

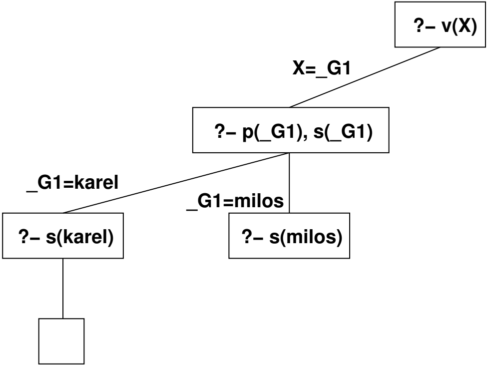
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
karel  ;     /* uživatel vložil ; */
```



Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

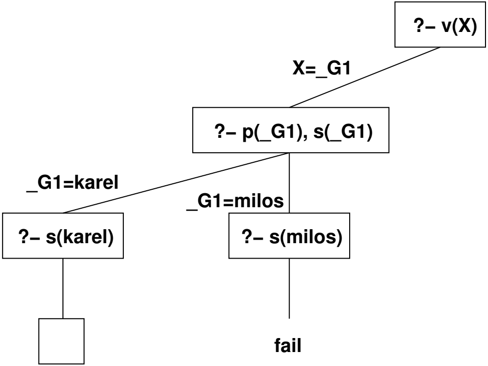
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
karel ;     /* uživatel vložil ; */
```



Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

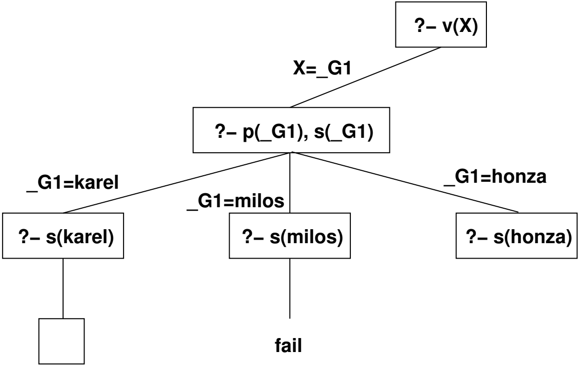
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
karel ;     /* uživatel vložil ; */
```



Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

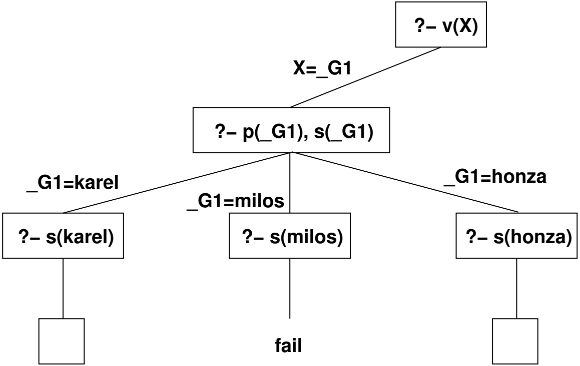
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
karel  ;     /* uživatel vložil ; */
```



Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

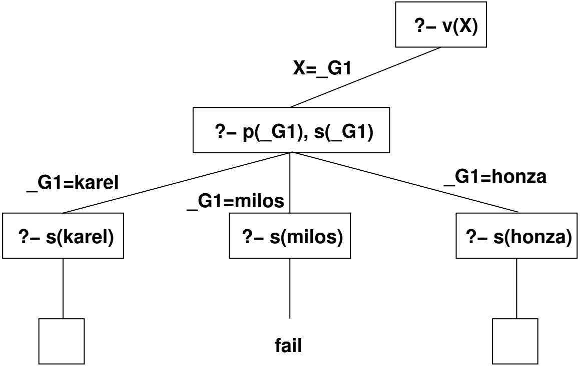
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
karel ;      /* uživatel vložil ; */
```



Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

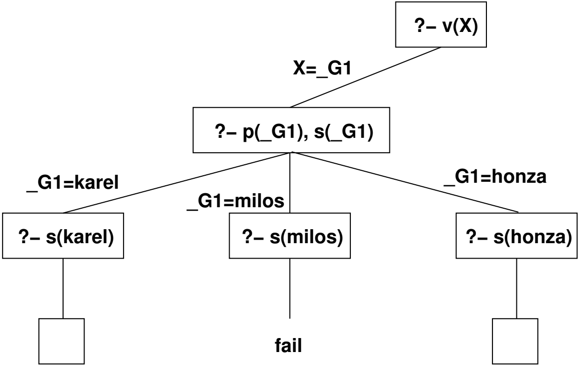
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

?-v(X).
karel  ;     /* uživatel vložil ; */
honza
```



Příběh slečny Prology

```
v(X) :- p(X), s(X).
v(nouma).

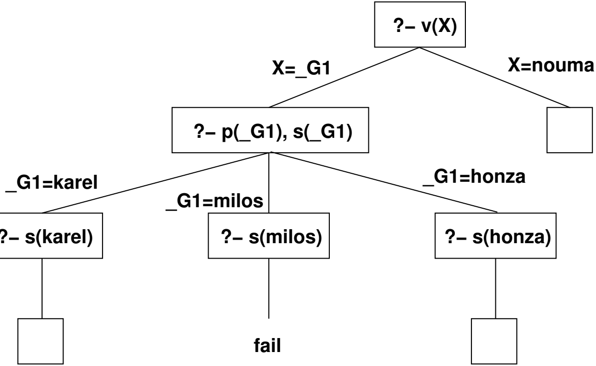
p(karel).
p(milos).
p(honza).

s(tomas).
s(honza).
s(karel).

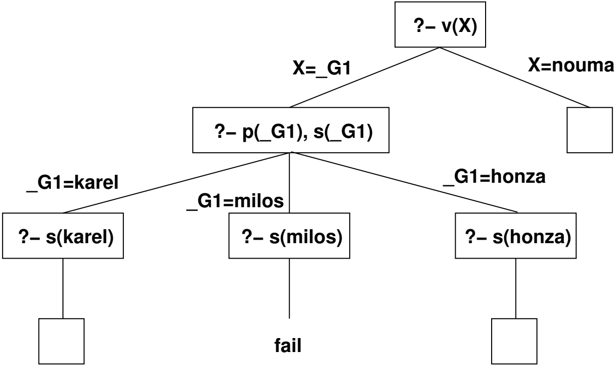
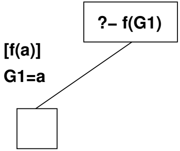
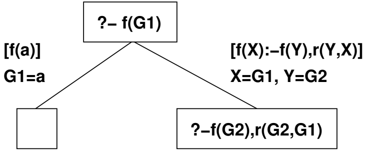
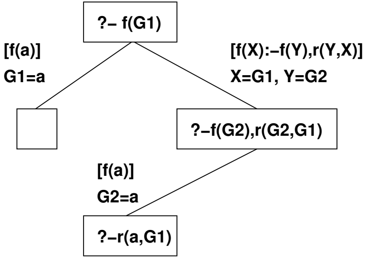
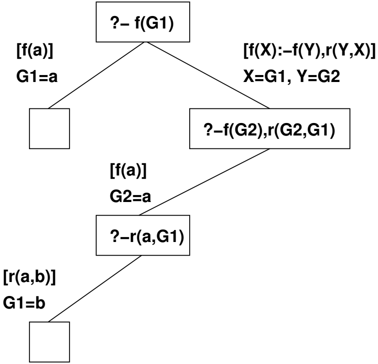
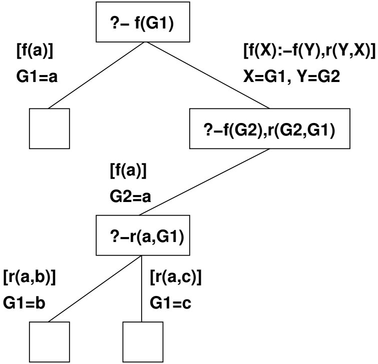
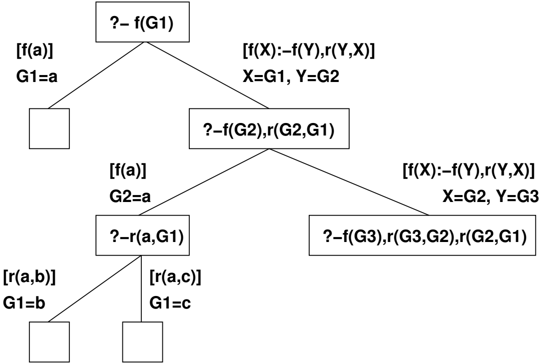
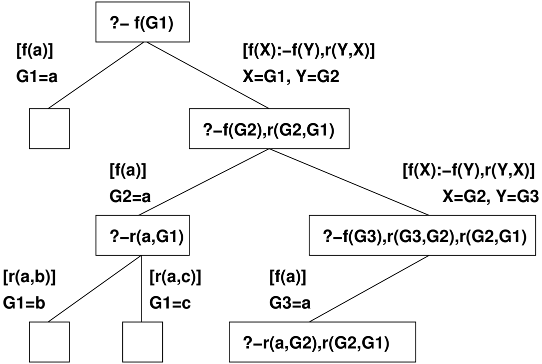
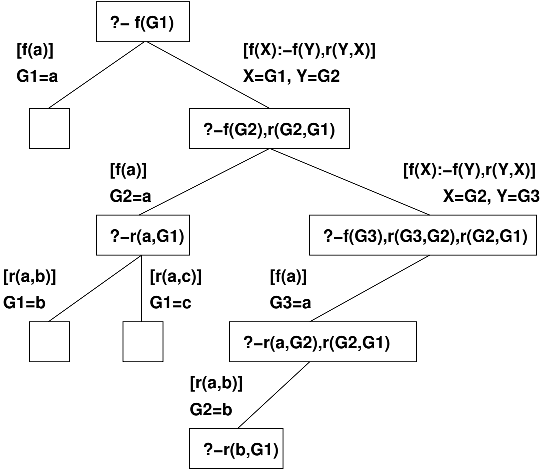
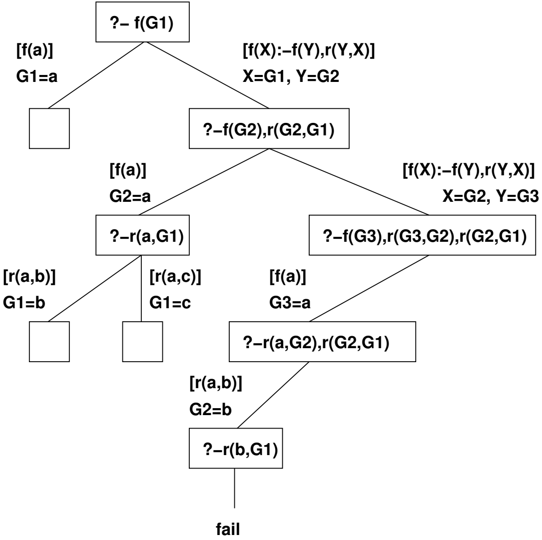
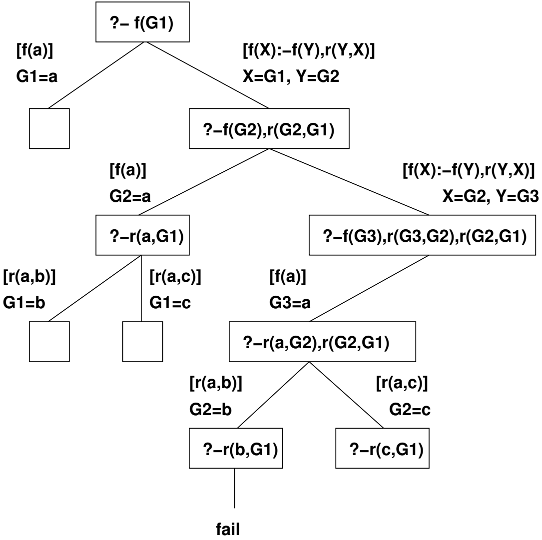
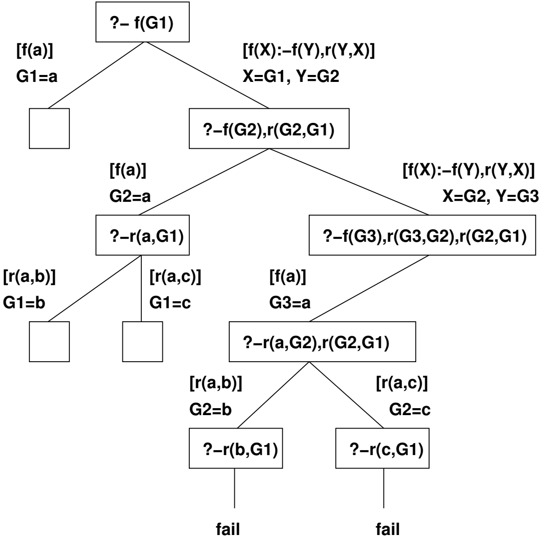
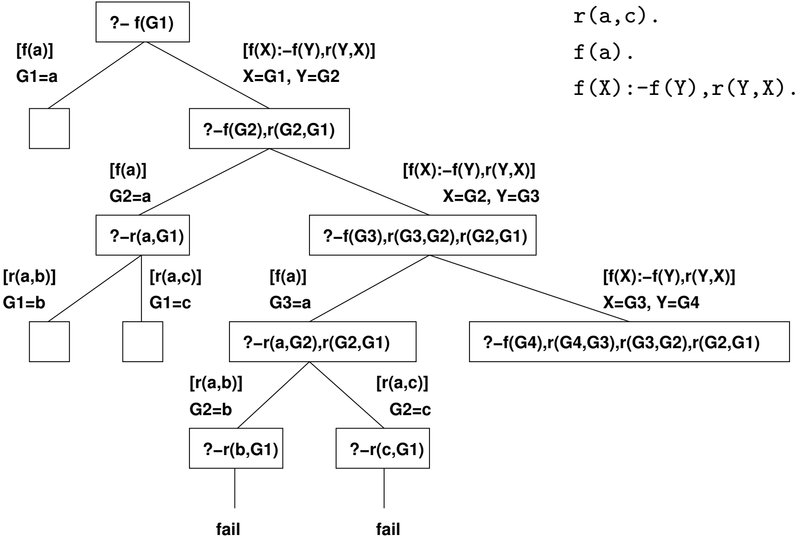
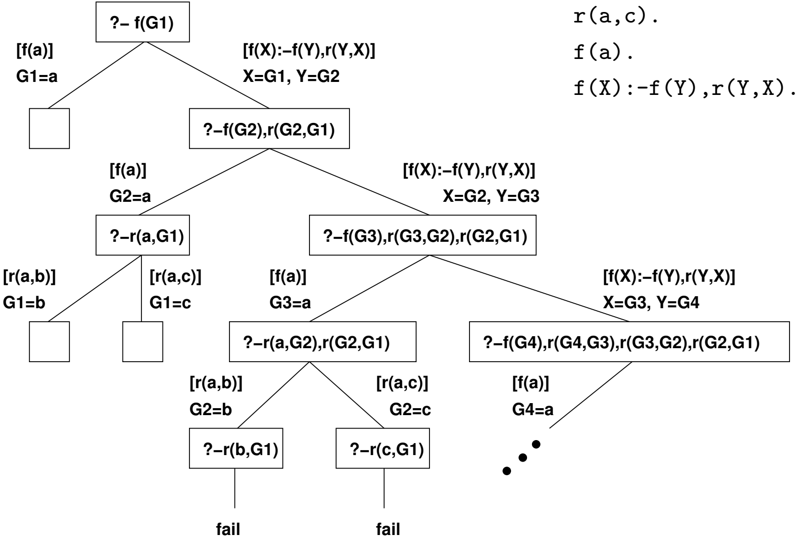
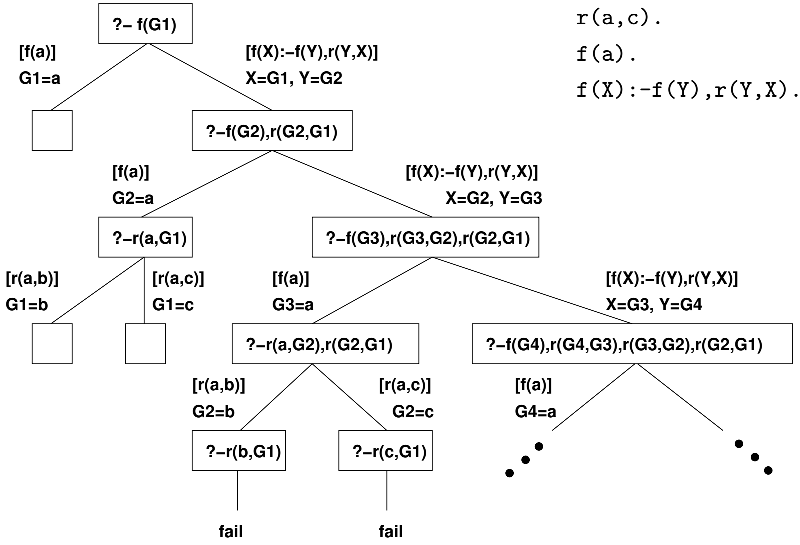
?-v(X).
karel  ;     /* uživatel vložil ; */
honza  ;     /* uživatel vložil ; */
```



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 ; */
```



```
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 přiřazenních na cestě od listu ke kořeni stromu.
- Vrcholy, pro které se nepodařilo nalézt položku v databázi vyhovující prvnímu podcíli jsou označeny podvrcholem **fail**.

Výpočet s rekurzí

?– 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).
```




```
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).
```



Rizika výpočtu a doporučení

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.

Domácí cvičení

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

- Narhněte 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, Tail rekurze v Prologu

Jiří Barnat

Seznamy a unifikace

Seznam v Prologu

- Konečná sekvence prvků.
- Různorodé (typově nestejné) prvky.
- Délkou seznamu se chápe počet prvků nejvyšší úrovně.

Zápis

- Hranaté závorky, prvky oddělené čárkou.

  `[marek, 2, matous, lukas(jan)]`
- Prázdný seznam:

  `[]`
- Zanořený seznam:

  `[ [], a, [c,[h]], [[[jo]]] ]`

Neprázdné seznamy

Struktura

- Neprázdný seznam se dekomponuje na hlavu seznamu (head) a tělo seznamu (tail).
- Prázdný seznam nemá interní strukturu.

Operátor `|`

- Pro dekompozici seznamů na hlavu a tělo.

Příklad

- ```
  ?- [H|T] = [marek,matous,lukas,jan].
  H = marek,
  T = [matous, lukas, jan].
  ```
- ```
  ?- [X|Y] = [].
  false.
  ```

Neprázdné seznamy – rozšířená hlava

Hlava jako prefix seznamu

- Hlavu lze zobecnit na neprázdnou konečnou sekvenci prvků.
- Prvky v hlavě seznamu jsou oddělovány čárkou.
- V jednom nezanořeném seznamu je smysluplné pouze jedno použití operátoru `|`.

Použití

- Správné použití

```
?- [X,Y|Z] = [1,2,3,4].
X = 1,
Y = 2,
Z = [3, 4].
```

- Nesprávné použití

```
?- [X|Y|Z] = [1,2,3,4].
```

ERROR

Anonymní proměnná

- Označená znakem podtržítko.
- 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|_]= Seznam.
```

Příklady unifikace těla seznamu.

- ```
  ?- [X,Y] = [1,2,3,4].
  false.
  ```
- ```
  ?- [X,Y|[Z]] = [1,2,3,4].
  false.
  ```
- ```
  ?- [_,_|[Z]] = [1,2,3].
  Z = 3.
  ```
- ```
  ?- [1,2|[Z,Y]] = [1,2,3,4].
  Z = 3,
  Y = 4.
  ```
- ```
  ?- [1,2|[Z,Y]] = [1,2,3,4,5].
  false.
  ```
- ```
  ?- [1,2|[Z|Y]] = [1,2,3,4,5].
  Z = 3,
  Y = [4, 5].
  ```

Od funkcí k relacím

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)`.

Příklad

- Vytvořte seznam, který začíná dvěma uvedenými prvky, a to v libovolném pořadí.
- Imperativní pohled, funkce vracející požadované výsledky (samostatnou otázkou je, jak reprezentovat více výsledků)

  `fStartsWith(X,Y)` $\rightsquigarrow^*$ `[X,Y|_]`

  $\rightsquigarrow^*$ `[Y,X|_]`

- V Prologu kódujeme jako relaci s o jedna větší aritou:

```
rStartsWith(X,Y,[X,Y|_]).
rStartsWith(X,Y,[Y,X|_]).
```

Příklad – rekurze na seznamech

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).
X = [b,b,b,b].
```

```
?- a2b(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].
```

Operace nad seznamy

Délka seznamu a test na bytí seznamem

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.
  ```

Operátor ., Otevřené seznamy

Operátor .

- Předřazení prvku k seznamu (aka : z Haskellu).
- Nemá infixový zápis.
- Příklady

```
?- X = .(b,[1,2,3]).
X = [b, 1, 2, 3].
```

```
?- X = .([a],[a]).
X = [[a], a].
```

Seznamy jako neúplná datová struktura

- Prolog umí pracovat i se seznamy, které nejsou kompletně definovány, tzv. **otevřené seznamy**.
- ```
  ?- Seznam = [1|Telo], Telo = [2|X].
  Seznam = [1,2|X],
  Telo = [2|X].
  ```
- ```
  ?- length([a,b|_],6).
  true.
  ```

Prvky seznamu – predikát member/2

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]).`

  $\rightsquigarrow$

  `?- member(lukas,[matous,lukas,jan]).`

  $\rightsquigarrow$

  `?- member(lukas,[lukas,jan]).`

  `true.`

Predikát `member/2` – použití

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.
  ```

Konkatenace seznamů predikátem append

append/3

- Dotaz `append(L1,L2,L3)` se vyhodnotí na pravda, pokud seznam `L3` je zřetězením 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` · `L2` = `L3`.
- Test na rovnost seznamů.
- Výpočet zřetězení dvou seznamů.
- Odvození prefixu, nebo sufixu v daném seznamu.
- Generování seznamů, jejichž zřetězení 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)
  ```

Predikáty nth

nth0(Index, List, Elem)

- `Index` prvku `Elem` v seznamu `List`, počítáno od 0, tj. první prvek má index 0.

nth1(Index, List, Elem)

- Totéž co `nth0/3`, ale počítáno od 1.

nth0(N, List, Elem, Rest)

- `Index` prvku `Elem` v seznamu `List`, počítáno od 0, tj. první prvek má index 0, `Rest` je seznam vzniklý ze seznamu `List` odebráním dotčeného prvku.

nth1(N, List, Elem, Rest)

- Totéž co `nth0/4`, ale počítáno od 1.

Cyklické seznamy

Pozorování

- Nedokonalost unifikačního algoritmu lze "zneužít" pro definici cyklických seznamů, tj. seznamů, které jsou periodicky nekonečné.

Příklady

- ```
  ?- unify_with_occurs_check(X,[1,2,3|X]).
  false.
  ```
- ```
  ?- X=[1,2,3|X], nth1(7,X,Z).
  X = [1, 2, 3|X],
  Z = 1.
  ```
- ```
  ?- X=[a,b|X], append([a,b,a,b,a],Z,X).
  X = [a,b|X],
  Z = [b|X].
  ```
  ```

Aritmetika

SWI Prolog - Aritmetické typy

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.**

Aritmetické funkce a operátory

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

Aritmetické operace v relacích

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

- V případě použití aritmetické operace takovéto chování vyžaduje inverzní aritmetickou funkci.
- Prolog při unifikaci a rezoluci nepočítá inverzní funkce, v okamžiku požadavku na takovouto operaci ohlásí interpret chybu (nedostatečná instanciace).
- Porovnejte:

`?- X is 3*3.`	`?- 9 is 3*X.`
`X = 9.`	**ERROR**
  ```

Příklady nedostatečné instanciace

Vyzkoušejte

- `?- 9 is X + 1.` ERROR
  `?- X > 3.` ERROR
  `?- X = 2, X > 3.` `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).
X = 2.
```

```
?- length1([a,b],X).
```
ERROR

Instanciace parametrů

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 instaciovanými parametry, říkáme, že predikát je použit v `(+,+)` módu.

Příklady z dokumentace

between(+Low, +High, ?Value)

- `Low and High are integers, High >=Low. If Value is an integer, Low=< Value=< 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>= 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.`

Tail rekurze

Rekurze a efektivita

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.
- Akumulátor = pomocný shromažďovací parametr.
- Zavedení akumulátoru vyžaduje definici pomocného predikátu.

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 `(?,+,?)`.

Iniciální hodnota akumulátorového parametru

Pozorování

- V některých případech je obtížné zvolit iniciální hodnotu akumulačního parametru.
- Uvažme následující definici pomocného predikátu `accMax/3`, který s využitím akumulátoru pro volání `accMax(List,A,Max)` počítá největší číslo v seznamu:

```
accMax([],A,A).
accMax([H|T],A,Max) :- H > A, accMax(T,H,Max).
accMax([H|T],A,Max) :- H =< A, accMax(T,A,Max).
```

Úkol

- S využitím `accMax/3` definujte predikát `max_member/2`.
- Jakou zvolit výchozí hodnotu akumulátoru?

Iniciální hodnota akumulátorového parametru

Pozorování

- V některých případech je obtížné zvolit iniciální hodnotu akumulačního parametru.
- Uvažme následující definici pomocného predikátu `accMax/3`, který s využitím akumulátoru pro volání `accMax(List,A,Max)` počítá největší číslo v seznamu:

```
accMax([],A,A).
accMax([H|T],A,Max) :- H > A, accMax(T,H,Max).
accMax([H|T],A,Max) :- H =< A, accMax(T,A,Max).
```

Úkol

- S využitím `accMax/3` definujte predikát `max_member/2`.
- Jakou zvolit výchozí hodnotu akumulátoru?

```
max_member(List,Max) :- List = [H|_], accMax(List,H,Max).
```

Řetězce, operátor syntaktické ekvivalence

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 ekvivalence (identita) - operátor ==

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.
  ```

Konverze řetězců a jejich délka

string_to_atom(?String, ?Atom)

- Logical conversion between a string and an atom. At least one of the two arguments must be instantiated. Atom can also be an integer or floating point number.

string_to_list(?String, ?List)

- Logical conversion between a string and a list of character code characters. At least one of the two arguments must be instantiated.

string_length(+String, -Length)

- Unify Length with the number of characters in String. This predicate is functionally equivalent to atom_length/2 and also accepts atoms, integers and floats as its first argument.

Zřetězení a detekce podřetězců

string_concat(?String1, ?String2, ?String3)

- Similar to atom_concat/3, but the unbound argument will be unified with a string object rather than an atom. Also, if both String1 and String2 are unbound and String3 is bound to text, it breaks String3, unifying the start with String1 and the end with String2 as append does with lists.

sub_string(+String, ?Start, ?Length, ?After, ?Sub)

- Sub is a substring of String starting at Start, with length Length, and String has After characters left after the match. See also sub_atom/5.

Příklady

- ```
  ?- sub_string('Nenene!',X,Y,Z,'ne').
  X = Y, Y = 2, Z = 3 ;
  X = 4, Y = 2, Z = 1 ;
  false.
  ```
- ```
  ?- sub_string('Nenene!',4,2,1,X).
  X = "ne".
  ```

Příklady

Příklad: násobky daného čísla

Zadání

- Definujte predikát `nasobky(+Cislo,+Pocet,?Seznam)`, který je pravdivý, pokud `Seznam` je prvních `Pocet` násobků čísla `Cislo`.
-
```
?- nasobky(3,6,[3,6,9,12,15,18]).
true.
```

Řešení

-
```
nasobky(N,P,L) :- aNas(N,P,[],L).
aNas(_,0,Acc,Acc).
aNas(N,P,A,L) :- P>0,            /* prevent infinite recursion */
                 P1 is P-1, K is N*P,
                 aNas(N,P1,[K|A],L).
```

Příklad: packList

Zadání

- V prologu naprogramujte predikát `packList/2`, který realizuje vynechání sousedních identických termů v seznamu, predikát předpokládá mód použití `(+,?)`.
- ```
  ?- packList([a,a,a,1,1,c,c,c,[a],[a]],X).
  X = [a,1,c,[a]].
  ```

Řešení

- ```
  packList(L,P) :- aPL(L,RP,L,[]), reverse(RP,P).
  aPL([],P,_,P).
  aPL([H|T],P,L,Acc) :- H \== L, aPL(T,P,H,[H|Acc]);
                        H == L, aPL(T,P,L,Acc).
  ```

Domácí úloha

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

Řezy, vstup-výstup, všechna řešení

Jiří Barnat

Řez

Motivace pro operaci řezu

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.

Vestavěný predikát ! (vykřičník)

Operátor řezu – `!/0`

- Vždy jako podcíl uspěje.
- 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

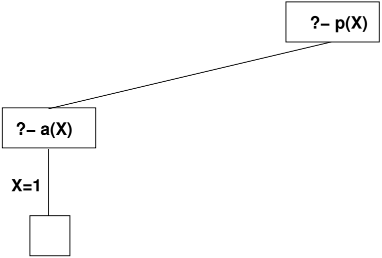
- Prořezává výpočetní strom.
- Rychlejší výpočet.
- Riziko odřezání větví výpočtu, které vedou k dalším (stejným, či jiným) řešením.

Příklad fungování řezu – bez řezu

?– p(X)

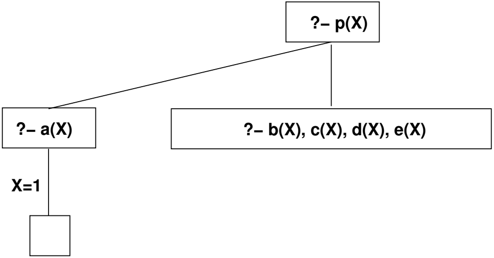
```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



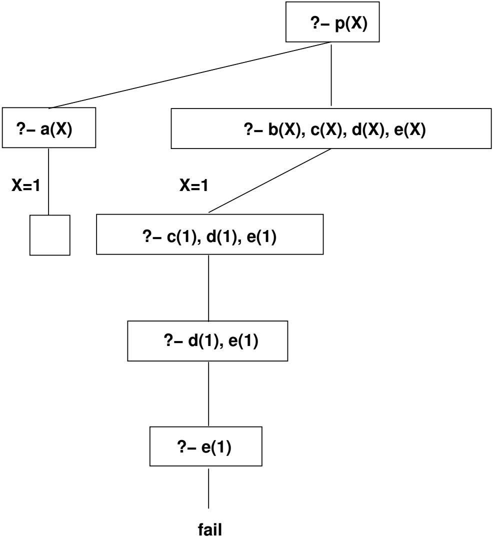

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



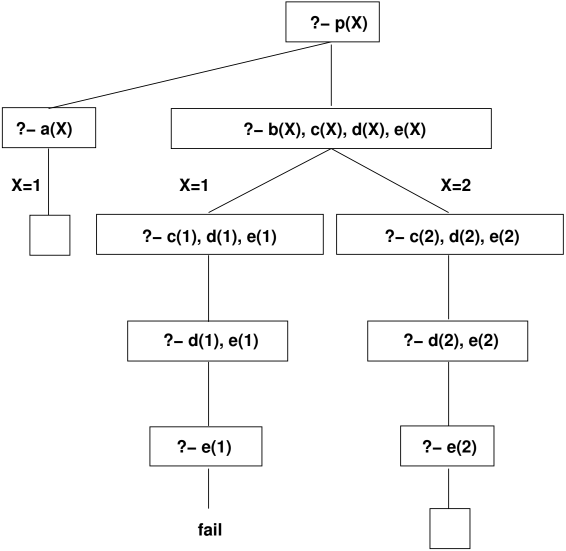

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



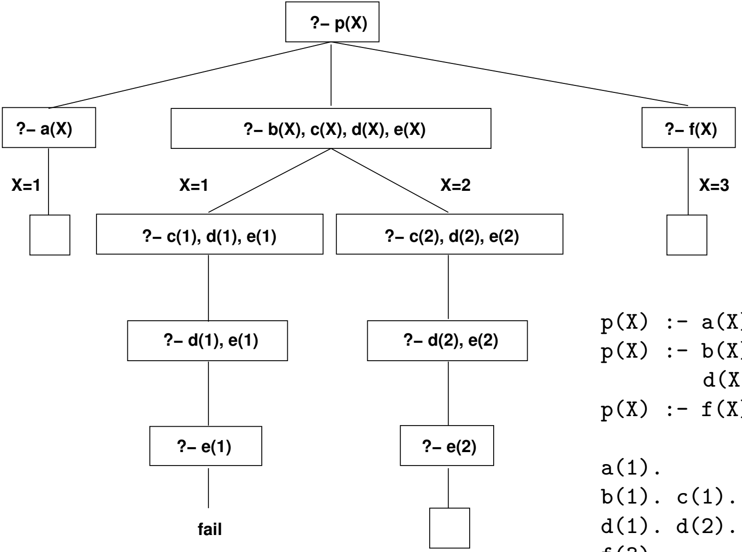

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



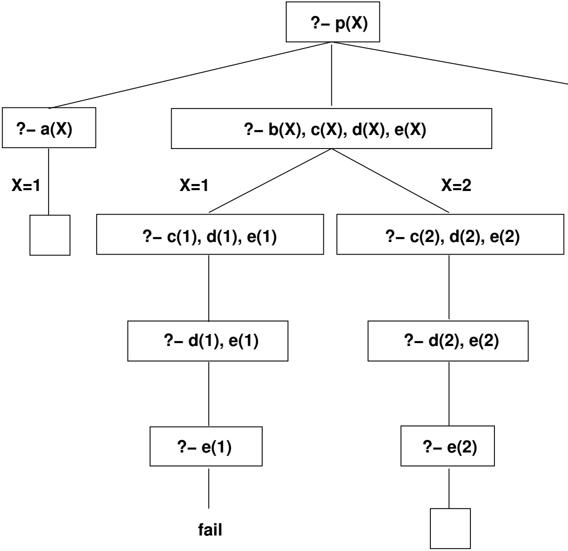

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – bez řezu



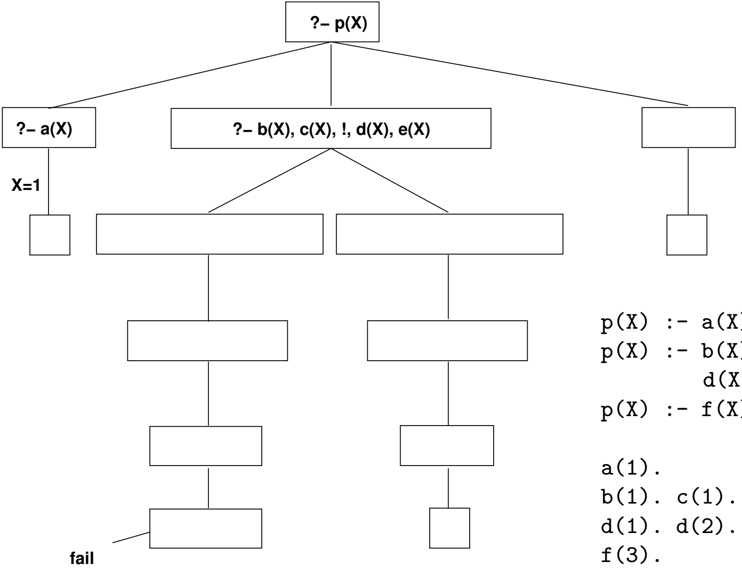

```
p(X) :- a(X).
p(X) :- b(X),c(X),
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem



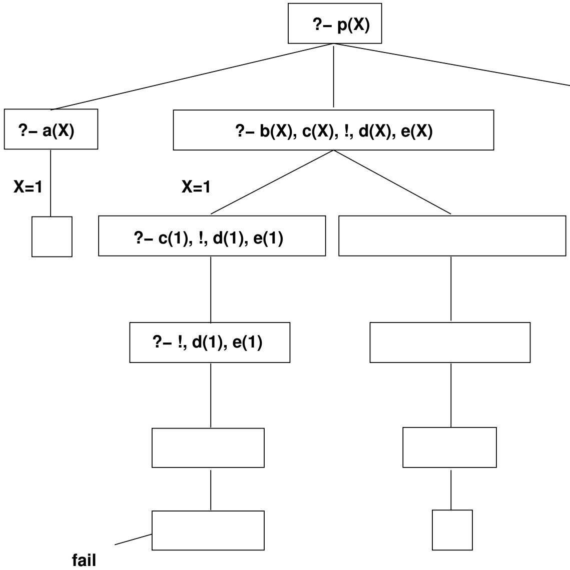

```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem



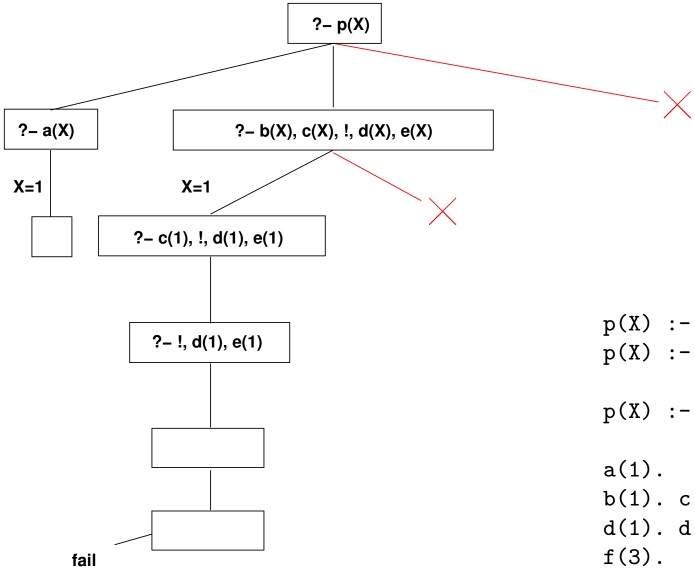

```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad fungování řezu – s řezem



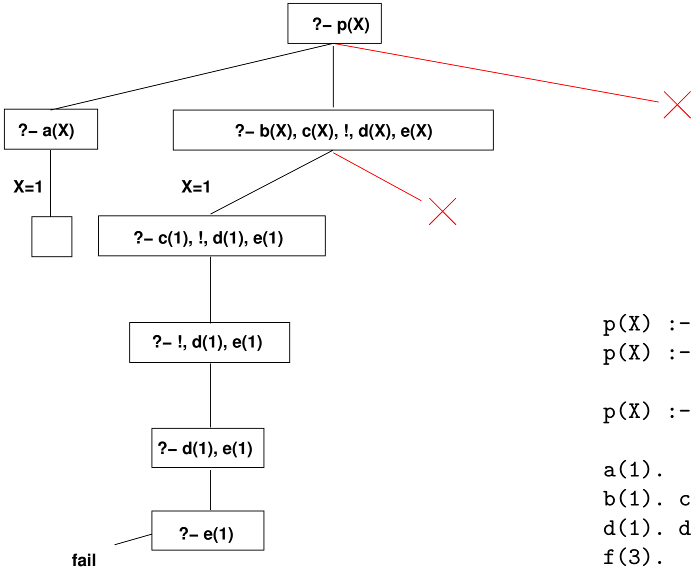

```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

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).
```

Vedlejší efekty – Upnutí

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.
```

Vedlejší efekty – Prořezání

Popis

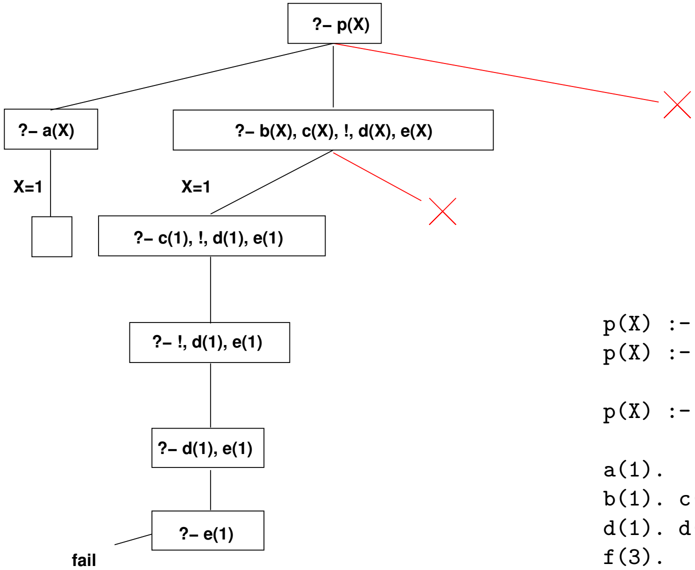
- Pokud se při řešení podcíle narazí v těle pravidla na operátor řezu, všechny unifikace vyplývající z podcílů vyskytujících se v těle pravidla před operátorem `!` se fixují (jiné možnosti unifikace těchto podcílů se neuvažují).

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.
```

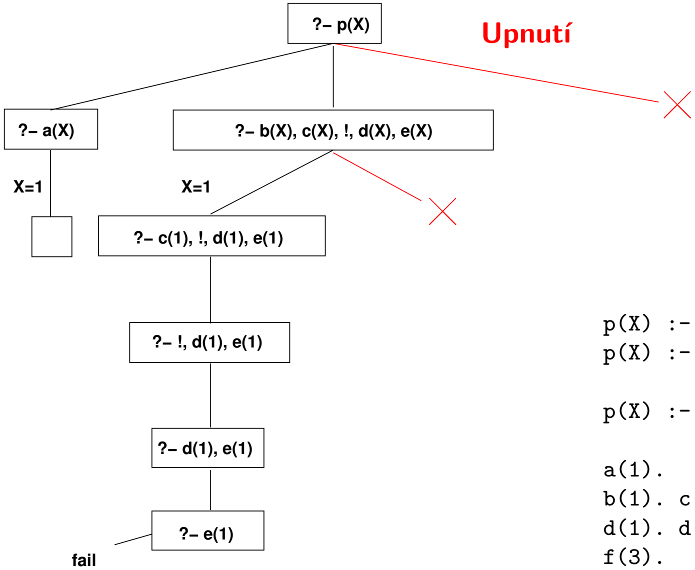
b)
```
a(X) :- X = 0.
a(X) :- X = 1.
b(X,Y) :- a(X), !, a(Y).

?- b(X,Y).
X = 0, Y = 0 ;
X = 0, Y = 1.
```

Příklad fungování řezu – vedlejší efekty



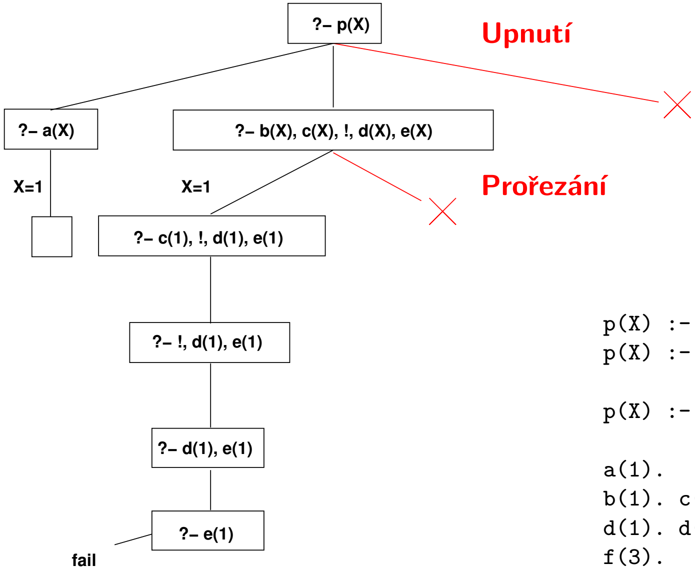

```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

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



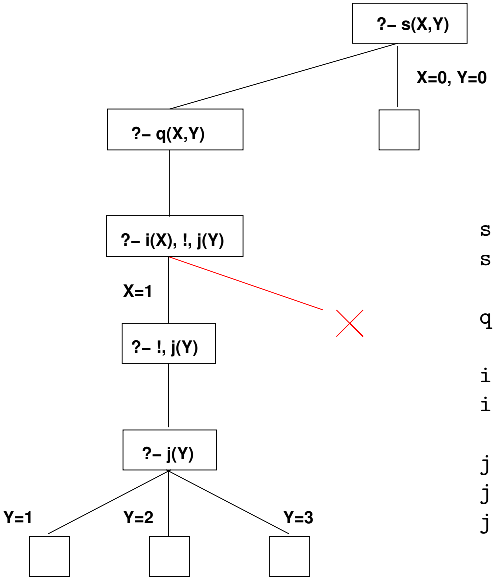

```
p(X) :- a(X).
p(X) :- b(X),c(X), !,
        d(X),e(X).
p(X) :- f(X).

a(1).
b(1). c(1).
d(1). d(2). e(2).
f(3).
b(2). c(2).
```

Příklad

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).
  ```
  ```

Příklad

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.
```

Příklad

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).
```

Příklad

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).
```

Použití řezu – max/3

Predikát `max/3`

- Uvažme predikát `max(+N1,+N2,?Max)`, který se pro číselné argumenty vyhodnotí na pravda, pokud třetí číslo je maximem prvních dvou.

Řešení bez použití řezu

- ```
  max(X,Y,Y):- X =< Y.
  max(X,Y,X):- X > Y.
  ```

Pozorování

- Neefektivita v případě dotazu:
  ```
  ?- max(3,4,X).
  X = 4 ;     /* následuje úplně zbytečný výpočet */
  false.
  ```
- Uvedené klauzule jsou vzájemně výlučné, pokud výpočet na jedné uspěje, vyhodnocovat druhou klauzuli je zcela zbytečné.
  ```

Řešení s použitím řezu

- ```
  max(X,Y,Y) :- X =< Y, !.
  max(X,Y,X) :- X > Y.
  ```
- Korektní řešení, nerealizuje nadbytečný výpočet při rekurzivním prohledávání stromu (díky upnutí).

Pozorování a otázka

- Pokud `X>Y`, dochází ke dvěma aritmetickým porovnáním.
- Podmínky jsou vzájemně výlučné.
- Je test `X > Y` v těle druhého pravidla vůbec nutný?

Odpověď

- V módu (+,+,-) je test nadbytečný.
- V módu (+,+,+) je test nutný, kdyby v klauzuli nebyl, tak:
  ```
  ?- max(2,3,2).
  true. /* CHYBA */
  ```
  ```

Efektivnější, ale nesprávné řešení

- ```
  max(X,Y,Y) :- X =< Y, !.
  max(X,Y,X).
  ```
- Problém: cíl `max(2,3,2)` se neunifikuje s hlavou prvního pravidla, přestože platí podmínka `2=<3`.

Opravené správné řešení

- ```
  max(X,Y,Z) :- X =< Y, !, Z = Y.
  max(X,Y,X).
  ```
- K unifikaci s hlavou prvního pravidla dojde vždy, podmínka je vždy vyhodnocena, a je-li třeba, dojde k upnutí.
- Funguje korektně v módu `(+,+,?)`.
- Vždy provede pouze jedno aritmetické porovnání.

Typy řezů

Zelené řezy

- 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 řezy eliminující duplicity.

Červené řezy

- Odstraněním operátoru řezu se mění sémantika programu (po odstranění řezu, je možné nalézt další jiná řešení).

Obecná doporučovaná strategie

- Vyrobit funkční řešení bez řezů.
- Zvýšit efektivitu použitím „zelených“ řezů.
- Využít „červené řezy“ pouze pokud není vyhnutí, dobře okomentovat.

Negace v Prologu

Predikát `fail`

Predikát `fail/0`

- Vestavěný predikát, nikdy neuspěje.
- Pokus o dokazování `fail` způsobí „backtracking“ ve výpočtu.

Pozorování/Připomenutí

- Pokud všechny větve výpočetního stromu skončí neúspěchem, interpretr ohlásí `false`, tj. že požadovaný cíl nelze dokázat.

Vysvětlete

- ```
  ?- fail.
  false.
  ```
- ```
  a(_) :- fail.
  a(_).
  ?- a(cokoliv).
  true.
  ```
- ```
  a(_) :- !, fail.
  a(_).
  ?- a(cokoliv).
  false.
  ```
  ```

Kombinace `fail` a upnutí (cut-fail combo)

Kombinace fail a upnutí

- V kombinaci s řezy, konkrétně mechanismem upnutí, může predikát `fail` sloužit jako negace.

Příklad

- V Prologu zapište: „Hezké je vše, co není škaredé.“

```
hezke(X) :- skarede(X), !, fail.
hezke(_).
skarede(strasidlo).
```

- Pokud je možné dokázat podcíl `skarede(X)`, pak predikát `hezke(X)` pro totéž `X` se vyhodnotí na `false`.

```
?- hezke(strasidlo).        ?- hezke(cokoliv_jineho).
false.                      true.
```

Predikát negace \+/1

Význam predikátu `\+/1`

- Pokud $\exists$(X$_1$,...,X$_n$) takové, že P(X$_1$,...,X$_n$) je dokazatelné, pak

  ```
  ?- \+P(X1,...,Xn)
  false.
  ```

Definice predikátu `\+/1`

- Definován následujícími pravidly:

  ```
  \+(P) :- P, !, fail.
  \+(_).
  ```

- Známa jako „Negation as failure.“

Použití \+/1 na termy s volnou proměnnou

Pozorování

- Při aplikaci na cíl s proměnnou, je negace vůči faktu, zda pro původní cíl existuje splňující přiřazení.

Neintuitivní chování – neodpovídá logické negaci

- ```
  barva(cervena).
  barva(modra).
  ```

  ```
  ?- X=zelena, \+barva(X).
  X = zelena.
  ```

  ```
  ?- \+barva(X), X=zelena.
  false.
  ```

Doporučení

- Operátor \+ používat pouze na podcíle s plně instanciovanými argumenty.
  ```

Neintuitivní chování negace predikátem \+

Pozorování

- Negace aplikovaná na termy s volnou proměnnou je nebezpečná zejména pokud se vyskytuje jako podcíl na pravé straně pravidla pro jiný term.

Příklad

- Uvažme následující program:

```
barva(cervena).
barva(modra).
foo(X) :- \+barva(X).
```

- Zajímá nás, zda existuje `X` takové, že platí `foo(X)`, tedy:

```
?- foo(X).
false.
```

- Logický závěr by mohl být, že takové X neexistuje, ale:

```
?- foo(fialova).
true.
```

Další predikáty pro řízení běhu programu: podmínka

If ->Then; Else

- Definováno následovně:

```
(If -> Then; Else) :- If, !, Then.
(If -> Then; Else) :- !, Else.
```

- Pokud není větev `Else` chová se jako:

```
If -> Then; fail.
```

Příklad použití podmínky

- `min(X,Y,Z) :- X =< Y -> Z = X ; Z = Y.`

Vstup, Výstup

Proudy a SWI Prolog

Proud (Stream)

- Místo, odkud program může číst, nebo kam může program zapisovat posloupnost znaků.
- Proudy realizují čtení z klávesnice, výpisy na obrazovku, čtení a zápis do souborů.

Předdefinované proudy `user_*`

- `user_input`, `user_output`, `user_error`
- Pro přímou interakci s uživatelem.
- Iniciálně svázány s proudy `stdin`, `stdout` a `stderr`.

Předdefinované proudy `current_*` (SWI-Prolog)

- `current_input`, `current_output`
- Iniciálně svázány s odpovídajícími proudy `user_*`.
- Definují místo čtení a zápisu pro predikáty, které neberou konkrétní proud jako svůj argument.

Manipulace s proudy

Poznámka

- V Prologu se ustálily dvě sady funkcí pro manipulaci se vstup-výstupními proudy.
- SWI Prolog podporuje oba módy a umí mezi nimi přepínat.

Edinburghský styl

- `tell/1`, `see/1`, ...
- Jednoduché rozhraní, snadné použití.

ISO standard

- `open/3`, `close/1`, ...
- Pro komplexní použití.

Edinburghský styl

see(+SrcDest)

- Otevře `SrcDest` pro čtení a nastaví aktuální vstupní proud.

tell(+SrcDest)

- Otevře `SrcDest` pro zápis a nastaví aktuální výstupní proud.

append(+File)

- Jako `tell/2` ale nastaví pozici místa zápisu na konec souboru.

seeing(-Stream)/telling(-Stream)

- Vrací aktuálně používané proudy pro čtení/zápis.

seen a **told**

- Uzavírá aktuální vstupní resp. výstupní proud.

Predikáty realizující vstup/výstup znaků

Forma čteného/zapisovaného znaku

- `byte` – číslo 0 až 255.
- `char` – znak.
- `code` – ASCII kód znaku.

put_char(+Char)
put_char(+Stream, +Char)

- Realizuje zápis znaku do aktuálního resp. zadaného proudu.
- Podobně `put_byte` a `put_code`.

Predikáty

- `nl` – zapíše znak nového řádku.
- `get_*` – načte znak v dané formě.
- `peek_*` – znak čekající na přečtení v dané formě.
- `tab(+A)` – zapíše `A` mezer.
- `flush_output` – vyprázdní buffer operačního systému.

Predikáty pro parsování vstupu unifikací

read(-Term)

- Přečte vstup až do další tečky, a přečtené se pokusí unifikovat s argumentem `Term`.
- Při čtení z konce souboru vrací atom `end_of_file`.

Příklad

-
```
?- read(name:N), read(adresa:[X,Y,Z]).
|: name: jirik. adresa:['u shnile tresne', 42, atlantida].
N = jirik,
X = 'u shnile tresne',
Y = 42,
Z = atlantida.
```

Predikáty pro vstup/výstup termů

write(+Term)
write(+Stream, +Term)

- Zápis termu do aktuálního/zadaného výstupního proudu.

writeln(+Term)

- Ekvivalentní zápisu `write(Term), nl`.

read_term(-Term, +Options)
write_term(+Term, +Options)

- Komplexní čtení zápis, viz dokumentace.

Predikát repeat/0 a zpracování vstupů

repeat/0

- Vždy uspěje, vytváří neomezený počet větvení výpočetního stromu pro „backtrackování“.
- ```
  repeat.
  repeat :- repeat.
  ```

Použití `repeat`

- Typickým použitím predikátu je zpracování vstupů.

```
Head :- repeat,
          ctiZeVstupu(X),
          zpracujVstup(X),
          jeKonecVstupu(X),          /* X == end_of_file. */
        !.
```

- Mimo toto použití se v podstatě nevyskytuje.
```

Sekce

Seznamy všech řešení

Seznamy všech řešení

Pozorování

- Dotazem s volnou proměnnou instruujeme Prolog, aby nalezl jedno vyhovující přiřazení volným proměnným.
- Uživatel může vynutit systematické hledání dalších řešení.
- Prolog ale umí vrátit seznam všech řešení najednou.

bagof(+Template, :Goal, -Bag)

- Vrací seznam `Bag` všech alternativ unifikovaných s `Template` vyhovujících cíli `Goal`.
- Vrací `false` pokud `Goal` nemá řešení.

Jednoduchý příklad

- `bagof(X,barva(X),Barvy).`

  `Barvy = [modra, cervena].`

bagof/3 - Příklad

Databáze

- ```
  slevy(albert,mleko,leden).
  slevy(albert,mleko,unor).
  slevy(billa,cukr,cerven).
  slevy(billa,cukr,prosinec).
  slevy(tesco,cukr,duben).
  ```

Dotazy

- ```
  ?- bagof(Z,slevy(X,Y,Z),R).
  X = albert, Y = mleko, R = [leden, unor] ;
  X = billa,  Y = cukr,  R = [cerven, prosinec] ;
  X = tesco,  Y = cukr,  R = [duben].
  ```
- ```
  ?- bagof(Z,slevy(_,_,Z),R).
  R = [leden, unor] ;
  R = [cerven, prosinec] ;
  R = [duben].
  ```
  ```

bagof/3 a existenční kvantifikace

Pozorování

- Při použití `bagof` různé hodnoty proměnných, které nejsou součástí výsledného seznamu, vedou na různé varianty výsledku.
- Pro sloučení těchto variant nestačí použít anonymní proměnnou.

Existenční kvantifikace

- Zápisem `Var^` před cíl vyjádříme, že různé hodnoty v této proměnné se nemají rozlišovat, stačí že existuje nějaké vyhovující přiřazení.
- Je možné takto kvantifikovat více proměnných:

```
X^Y^Cil(X,Y,Z)          [X,Y]^Cil(X,Y,Z)
```

Existenčně kvantifikované dotazy – Příklad

Databáze

- ```
  slevy(albert,mleko,leden).
  slevy(albert,mleko,unor).
  slevy(billa,cukr,cerven).
  slevy(billa,cukr,prosinec).
  slevy(tesco,cukr,duben).
  ```

Existenčně kvantifikované dotazy

- ```
  ?- bagof(Z,X^slevy(X,Y,Z),R).
  Y = cukr,  R = [cerven, prosinec, duben] ;
  Y = mleko, R = [leden, unor].
  ```
- ```
  ?- bagof(Z,Y^X^slevy(X,Y,Z),R).
  R = [leden, unor, cerven, prosinec, duben].
  ```
- ```
  ?- bagof(Z,[X,Y]^slevy(X,Y,Z),R).
  R = [leden, unor, cerven, prosinec, duben].
  ```

`findall/3`, `setof/3`

findall(+Template, :Goal, -Bag)

- Seznam všech vyhovujících řešení.
- V případě že `Goal` nemá řešení vrací prázdný seznam.
- Jinak funguje stejně jako `bagof/3` s tím, že všechny volné proměnné jsou existenčně kvantifikovány.

setof(+Template, :Goal, -Set)

- Využívá predikát `bagof/3`, ale výsledek seřadí s použitím predikátu `sort/2`. Výsledek je tedy seřazený seznam všech možných řešení, s tím že každé řešení je uvedeno pouze jednou (duplicitní řešení jsou odstraněna).

IB015 Neimperativní programování

Ukázky použití Prologu a závěrečné zhodnocení

Jiří Barnat

Einsteinova hádanka

Zadání hádanky

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.

Řešení v Prologu

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`

Programování s omezujícími podmínkami

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.
- ...

Obecný postup řešení a domény proměnných

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

Constraint Programming a SWI-Prolog

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).

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^E2` (umocnění), `min(E1,E2)`, `max(E1,E2)`,
  `E1/E2` (celočíselné dělení ořezáním),
  `E1 rem E2` (zbytek po dělení /)
  jsou výrazy v celočíselné doméně.

Omezující podmínky

- Relační operátory předřazené znakem `#`.
- `E1 #>= E2`, `E1 #=< E2`,
  `E1 #= E2`, `E1 #\= E2`,
  `E1 #> E2`, `E1 #< E2`,

`clpfd` – logické spojky

Logické spojky

- `#\ Q` – Negace
- `P #\/ Q` – Disjunkce
- `P #/\ Q` – Konjunkce
- `P #<==> Q` – Ekvivalence
- `P #==> Q` – Implikace
- `P #<== 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 =< I =< 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`.

clpfd – jednoduché dotazy

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.
  ```

SEND + MORE = MONEY

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.
  ```
  ```

`clpfd` – hledání řešení

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.

`clpfd` – parametry hledání řešení

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.

SEND + MORE = MONEY

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.
  ```

`clpfd` – vybrané predikáty omezujících podmínek

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 =< SJ` nebo `SJ+DJ =< 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.
  ```

Deklarativní versus imperativní

Deklarativní programovací paradigma

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.

Imperativní programovací paradigma

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.

Průnik deklarativních a imperativního světa

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.

Příklad použití deklarativního stylu

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);
  ```

Příklad použití deklarativního stylu

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);
  ```

Příklad použití deklarativního stylu

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);
  ```

Příklad použití deklarativního stylu

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<=N`).

Nově deklarativním stylem

- ```
  int fact(int n)
  {
    if (n==0) return 1;
    else return n*fact(n-1);
  }
  print fact(N);
  ```

Příklad použití deklarativního stylu

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<=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);
  ```

Příklad použití deklarativního stylu

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<=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);
  ```

- Jasné chování pro `N=0`.

Příklad použití deklarativního stylu

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<=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);
```

- Jasné chování pro `N=0`.
- Pojmenovaná funkce, syntaktická indicie pro sémantický význam.

A to je konec ...

Závěrem

Co si odneseme do života ...

- Funkcionální výpočetní paradigma.
- Solidní základy programovacího jazyka Haskell.
- Solidní 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.

Závěrem

Co si odneseme do života ...

- Funkcionální výpočetní paradigma.
- Solidní základy programovacího jazyka Haskell.
- Solidní 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.

Poděkování

Přednášejícího studentům

- Za vzornou docházku a přípravu jak na přednášky, tak i na cvičení, vnitrosemestrální 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.