

# Funkcionální a logické programování



Dušan Kolář

*kolar@fit.vutbr.cz*

+420-54-114 1238



# Bodový zisk

■ Půlsemestrální zkouška	20
Projekt 1 (FP)	12
Projekt 2 (LP)	8
<i>Zápočet</i>	<i>20</i>
Závěrečná zkouška	60 ( <b>25</b> )
■ <b>CELKEM</b>	100 ( <b>45</b> )



# Důležitá data

- Odevzdání projektů (*další info za chvíli*)
  - projekt 1: 10.4.2016 (*neděle*)
  - projekt 2: 1.5.2016 (*neděle*)
- Půlsestrální zkouška 7. týden  
21.3.2016
  -
- Závěrečná zkouška (semestr 6.5.)
  - termín 1: *bude určen globálně*
  - termín 2: *bude určen globálně*
  - termín 3: *bude určen globálně*



# Rozvrh

## ■ Přednášky:

*pondělí*

– učebna

E112

– čas

13:00-14:50

## ■ Počítače:

*pondělí, úterý*

– učebna

N103-N105

– čas

08:00-09:50 (po)

19:00-20:50 (út)

**výuka již běží!**



# Počítače

1. 8.2. + 9.2. a 15.2. + 16.2.
  2. 22.2. + 23.2. a 29.2. + 1.3.
  3. 7.3. + 8.3. a 14.3. + 15.3.
- 
4. 21.3. + 22.3. a **28.3.** + 29.3.
  5. 4.4. + 5.4. a 11.4. + 12.4.
  6. 18.4. + 19.4. a 25.4. + 26.4.



# Záznamy přednášek

- Implicitní nastavení
  - Nevysílá se
  - Neukládá se
- Důvod
  - Ztratily původní význam – není o přednášky zájem
- Budeme ukládat
  - Zveřejnění po zpracování (pozor i 3 týdny!)
  - **Nemá význam, jaký se tomu přikládá!**
  - ***Programování je třeba si odprogramovat!***

# Důvody pro studium FJ (1)

- S programy se lépe matematicky pracuje
  - referenční transparence
    - v daném kontextu představuje proměnná vždy tutéž hodnotu
      - »  $x := x + 1$        $\{ x = 3 \}$       **imperativní styl**
      - »  $x := x + 1$        $\{ x = 4 \}$       proměnná  $\equiv$  adresa hodnoty
  - stejné lze vyjádřit stejným (optimalizace)
    - »  $x - x = f(1) - f(1)$

# Důvody pro studium FJ (2)

- FJ mají lepší mechanismus abstrakce
  - díky možnosti abstrakce jsou programy ve FJ stručnější  
(problém: kryptografické programy)
  - funkce vyššího řádu (funkce jako hodnota, ne ukazatel na funkci)
    - »  $\Sigma = 0 + n_1 + n_2 + \dots$   
 $\Pi = 1 * n_1 * n_2 * \dots$   
 $\heartsuit = C \heartsuit n_1 \heartsuit n_2 \heartsuit \dots$
    - » `compute(C,  $\heartsuit$ ,  $n_1$ ,  $n_2$ , ...)`





# Důvody pro studium FJ (3)

- FP umožňuje nové algoritmické přístupy
  - lazy evaluation X eager/strict evaluation
    - » možnost práce s potenciálně nekonečnými datovými strukturami
    - » možnost oddělení dat od řízení (nemusíme se starat o to, jak proběhne vyhodnocení)
    - » sdílení mezivýsledků/memoization



# Důvody pro studium FJ (4)

- FP umožňuje nové přístupy k vývoji programů
  - možnost dokazování programů
  - možnost transformace programů na základě algebraických vlastností



# Důvody pro studium FJ (5)

- FP umožňují využít (masivně) paralelního výpočtu
  - „jednoduchá“ dekompozice programu na části, které lze vyhodnocovat paralelně
  - problém: paralelismu je potenciálně příliš mnoho



# Důvody pro studium FJ (6)

- FP je důležité pro některé oblasti aplikace informatiky
  - umělá inteligence
  - specifikace (jazyk Z), modelování, rychlé prototypování
- Možnost využít i pro „tradiční“ oblasti informatiky



# Důvody pro studium FJ (7)

- FP má blízký vztah k teoretické informatice
  - sémantika programovacích jazyků
  - teorie složitosti algoritmů
  - teorie vyčíslitelnosti



# Důvody pro studium FJ (8)

- FP je zajímavou a myšlení rozvíjející činností pro studenty informatiky!
  - Jiný pohled na programování
  - Odstranění zažitých dogmat z procedurálních jazyků
    - Java
    - C++
    - Scala



# Doporučení

- Sledujte [www stránky](#)
  - Všechny údaje nejprve tam
  - Sám od tam opisují data
- Chodte na přednášky, cvičení
  - v průběhu si zkoušejte, co se říká
- Programujte, programujte, programujte
- ***Nejde o algoritmizaci, ale o zápis!***
  - Algoritmy znáte/jsou triviální
- Projekt  $\neq$  jedno řešení všech



# Individuální projekt

## ■ Není to

- že odjedu na Saharu a tam to spáchám
- že to stáhnu z netu a upravím
- že se o práci podělím s kolegou/y

## ■ Může být to

- že s kolegy provedu diskusi na dané téma, ale naimplementuje každý **sám**
- podívám se na net, jak to kdo řešil, ale své řešení vytvořím **sám**





# Hodnocení individuálního projektu

- Bere v potaz rozdílnost složitosti jednotlivých zadání
- Je mírnější, než u týmových projektů
- Umožňuje získ bonusových bodů za zvolení přístupu k řešení problému, který pozitivně vybočuje z řady
- *Uznání z loňska 16+*

# Individuální projekt

- Je výrazně jednodušší, než týmový
  - žádný „parsec“
- Odbourává závislost na kolezích
  - špatné zkušenosti, aspoň 2 týmy ročně
- Týmové jsou v jiných předmětech
- Umožňuje lepší organizaci
- **Přinutí si to všechny osahat 😊**



# Literatura - FP

- Bird R., Wadler P.: *Introduction to Functional Programming*, Prentice Hall, 1988, ISBN 0-13-484189-1
- Thompson S.: *Haskell, The Craft of Functional Programming*, Addison Wesley, 1999, ISBN 0-201-34275-8
- **WWW stránky:**
  - <http://www.haskell.org>



# Literatura - LP

- Hill P., Lloyd J.: *The Gödel Programming Language*, MIT Press, 1994, ISBN 0-262-08229-2
- Bieliková M., Návrat P.: *Funkcionálne a logické programovanie*, 2000, FEI STU Bratislava
- **WWW stránky:**
  - <http://www.swi-prolog.org>
  - <http://www.csupomona.edu/~jrfisher/www/>
  - [http://cs.wwc.edu/~cs\\_dept/KU/PR/Prolog.html](http://cs.wwc.edu/~cs_dept/KU/PR/Prolog.html)



# Přehled přednášek

1. Úvod do FP
2. Haskell - úvod, seznamy
3. Uživatelské datové typy, typové třídy, pole
4. Vstupy/výstupy
5. Praktické úkoly
6. Dokazování ve FP
7. Denotační sémantika
8. Prolog – úvod
9. Seznamy, operátor řezu, řazení
10. Praktiky v Prologu 1
11. Praktiky v Prologu 2
12. Praktiky v Prologu 3
13. Gödel
14. CLP



# Úvod do funkcionálního programování

# Cíle oddílu



## ■ Úvod do FP

- Imperativní a deklarativní programování

## ■ Důvody studia FP

## ■ Lambda kalkul – rekapitulace

- Pro začátečníky je vhodné nastudovat např. Češka, Motyčková, Hruška: Vyčíslitelnost a složitost, či jiné publikace o lambda kalkulu

## ■ Důležité pojmy z algebry

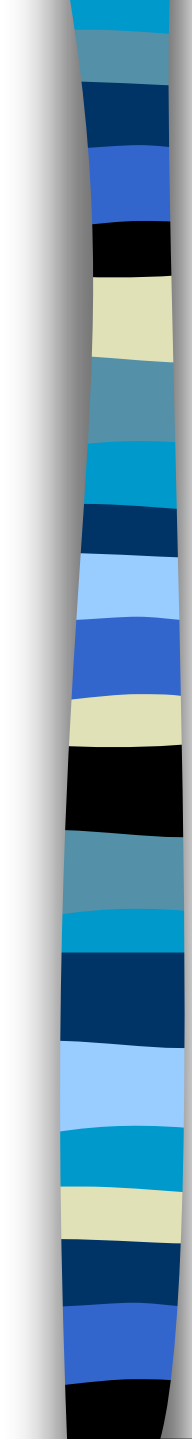


# Klasifikace programovacích jazyků

## ■ Imperativní jazyky

- program má implicitní **stav**, který se modifikuje
- explicitní pojem **pořadí** příkazů
- vyjadřujeme to, **jak** se má program vyhodnocovat
- vhodné pro tradiční architektury (von N.)
- většina současných jazyků





# Klasifikace programovacích jazyků (pokr.)

## ■ Deklarativní jazyky

- program **nemá** implicitní stav, stavové informace se udržují explicitně
- program je tvořen **výrazy** (termy) - nikoliv příkazy
- opakované provádění se vyjadřuje **rekurzí**
- vyjadřujeme **co** se má spočítat



# Funkcionální (aplikativní) jazyky

## ■ Základní model:

- matematický pojem funkce aplikované na argumenty a vypočítávající jediný výsledek  
⇒ výsledek je deterministický

## ■ Čistě funkcionální jazyky:

- FP, **Haskell**, Miranda, Hope, Orwell

## ■ Hybridní jazyky (imperativní):

- Lisp, Scheme, SML (a jeho deriváty)



# Relační (logické) jazyky

## ■ Základní model:

- **relace** (predikát)
- výpočet probíhá „nedeterministicky“ (s navracením)

## ■ Jazyky:

- **Prolog**, Parlog, **Goedel**
- CLP



# Historie funkcionálních jazyků

1930 Alonzo Church

čistý  $\lambda$ -kalkul

typovaný  $\lambda$ -kalkul

1958 Mac Carthy

LISP

1965 P. Landin

ISWIM

předchůdce ML

1978 R. Milner

ML - metajazyk pro  
systém podporující  
dokazování  
původní typový  
systém

1985 D. Turner

Miranda  
lazy evaluation



# Historie funkcionálních jazyků (pokr.)

1987

Haskell - modulární,  
lazy evaluation,

1989 A. Appel  
& D. Mac Queen  
Standard ML

1990 X. Leroy  
Caml Light

1998

Haskell 98  
standardizovaná  
verze jazyka  
(sloučení 4 hlavních  
implementací)

2003/4

Haskell - knihovny

# $\lambda$ -kalkul

## ■ Matematický model FJ

- FJ lze na něj převést, bez výhrady

## ■ Syntax:

- výraz  $\rightarrow$  proměnná  
| (výraz výraz)  
| ( $\lambda$  proměnná.výraz)

# Proměnná

- Označuje jakoukoliv hodnotu
  - typovaný  $X$  beztypový přístup
- **Vázaná a volná proměnná**
  - $\lambda f. (f\ x)$ 
    - $x$  je v  $(f\ x)$  volná
    - $f$  je v  $(f\ x)$  vázaná

# Aplikace

- $(e_1 \ e_2)$ 
  - představuje aplikaci funkce  $e_1$  na výraz  $e_2$
  - výsledek přitom může být dále aplikovatelný





# Abstrakce

- $(\lambda x.e)$ 
  - představuje funkci s parametrem  $x$  a tělem  $e$

# Substituce

■  $e_1[e_2/x]$

- substituce výrazu  $e_2$  za všechny volné výskyty proměnné  $x$  ve výrazu  $e_1$

# Platná substituce

- $e_1[e_2/x]$ 
  - libovolná **volná** proměnná v  $e_2$  se nesmí stát vázanou
- $(\lambda f.f\ x)[y/x] = (\lambda f.f\ y)$
- $(\lambda f.f\ x)[(f\ y)/x]$  **je neplatná substituce**

# $\alpha$ - konverze

- Přejmenování proměnných
  - $(\lambda x.e) \leftrightarrow (\lambda x'.e)[x'/x]$
  - substituce musí být platná!
- Využití
  - unifikace (porovnání)
  - zlepšení čitelnosti výrazů

# $\beta$ - konverze

- Vyhodnocení abstrakce
  - $(\lambda x. e_1) e_2 \leftrightarrow e_1[e_2/x]$
  - substituce musí být platná
- Operační charakter
  - mění formu výrazu, ale ne jeho hodnotu
    - ta je dána prvotním zápisem
  - globální strategie vyhodnocení (postupu)

# $\eta$ - konverze

## ■ Odstranění abstrakce

- $(\lambda x. f\ x) \leftrightarrow f$
- $x$  nesmí být volná v  $f$ 
  - Funkce jsou ekvivalentní pokud pro všechny hodnoty dávají stejný výsledek
- Umožňuje definovat **řezy** funkcí
  - $(+1) = (\lambda x. x + 1)$

# Normalizační teorémy

- Church-Rosserovy teorémy
- **redex** - **re**ducible **ex**pression: výraz, který lze dále redukovat
  - $\alpha/\beta$  - redex
- **normální forma** - výraz je v normální formě pokud neobsahuje žádný  $\beta$  - redex

# Teorém I

- Pokud  $e_1 \Leftrightarrow e_2$ , pak existuje výraz  $e$  takový, že
  - $e_1 \rightarrow e$
  - $e_2 \rightarrow e$



# Teorém II

- Pokud  $e_1 \rightarrow e_2$  a  $e_2$  je v normální formě, pak existuje taková redukční posloupnost, že z  $e_1$  je možné se „bezpečně“ dostat do  $e_2$   
**(normální redukční posloupnost)**



# Důsledek

- Pokud existuje normální forma, lze k ní dojít normální redukční posloupností (leftmost outermost redex)
  - Určuje způsoby vyhodnocení ve FP
  - Nevyhodnocuje se pod  $\lambda$ -abstrakcemi
    - využívá se zejména v programování u striktních jazyků

# Funkce - matematický pojem

## ■ Zobrazení množiny $A$ do množiny $B$

–  $f: A \rightarrow B$

- $A$  - definiční obor
- $B$  - obor hodnot
- $f(a) = b$

–  $f(x_1) = y_1 \ \&\& \ f(x_1) = y_2 \Rightarrow y_1 = y_2$

# Vybrané typy funkcí

## ■ Injektivní

- různé prvky  $A$  se zobrazují na různé prvky  $B$   
 $f(a) = f(a') \Leftrightarrow a = a'$

## ■ Surjektivní

- všechny prvky oboru hodnot mají svůj vzor  
 $\forall b \in B \exists a \in A: f(a) = b$

## ■ *Bijektivní*

- injektivní + surjektivní

# Kompozice a inverze funkcí

## ■ Kompozice

- $f: A \rightarrow B, g: B \rightarrow C$
- $(g \circ f)(a) = g(f(a))$

## ■ Inverze

- $f: A \rightarrow B, f^{-1}: B \rightarrow A$
- $f^{-1}(f(a)) = a$ 
  - existuje ke každé injektivní funkci
  - každá bijektivní funkce je invertibilní

# Binární operace na $A$

■  $\oplus: (A \times A) \rightarrow A$

- asociativita
- komutativita
- levá/pravá identita

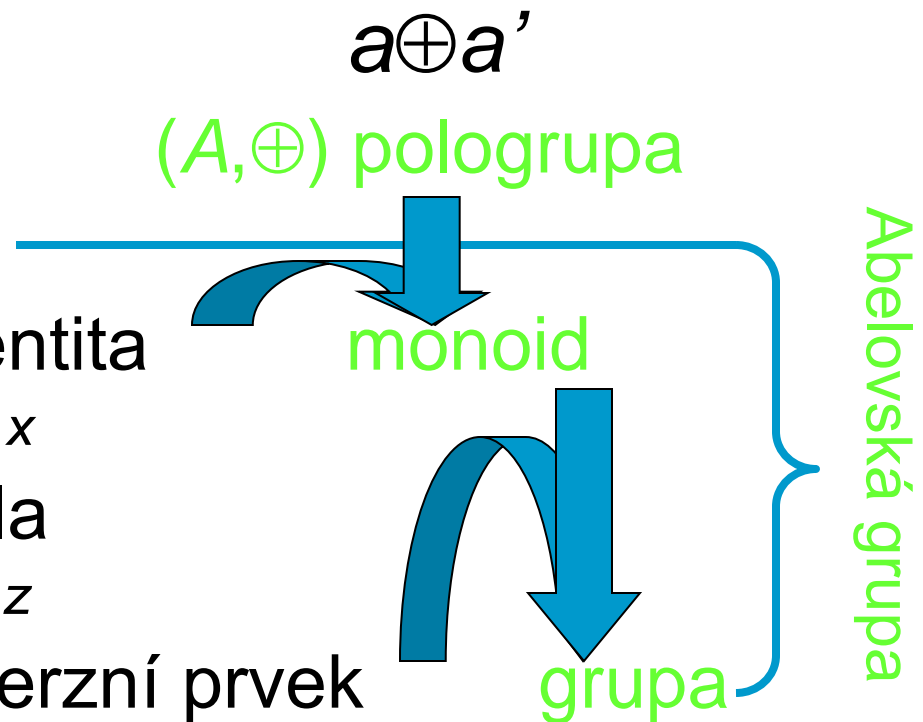
$$\gg e \oplus x = x$$

- levá/pravá nula

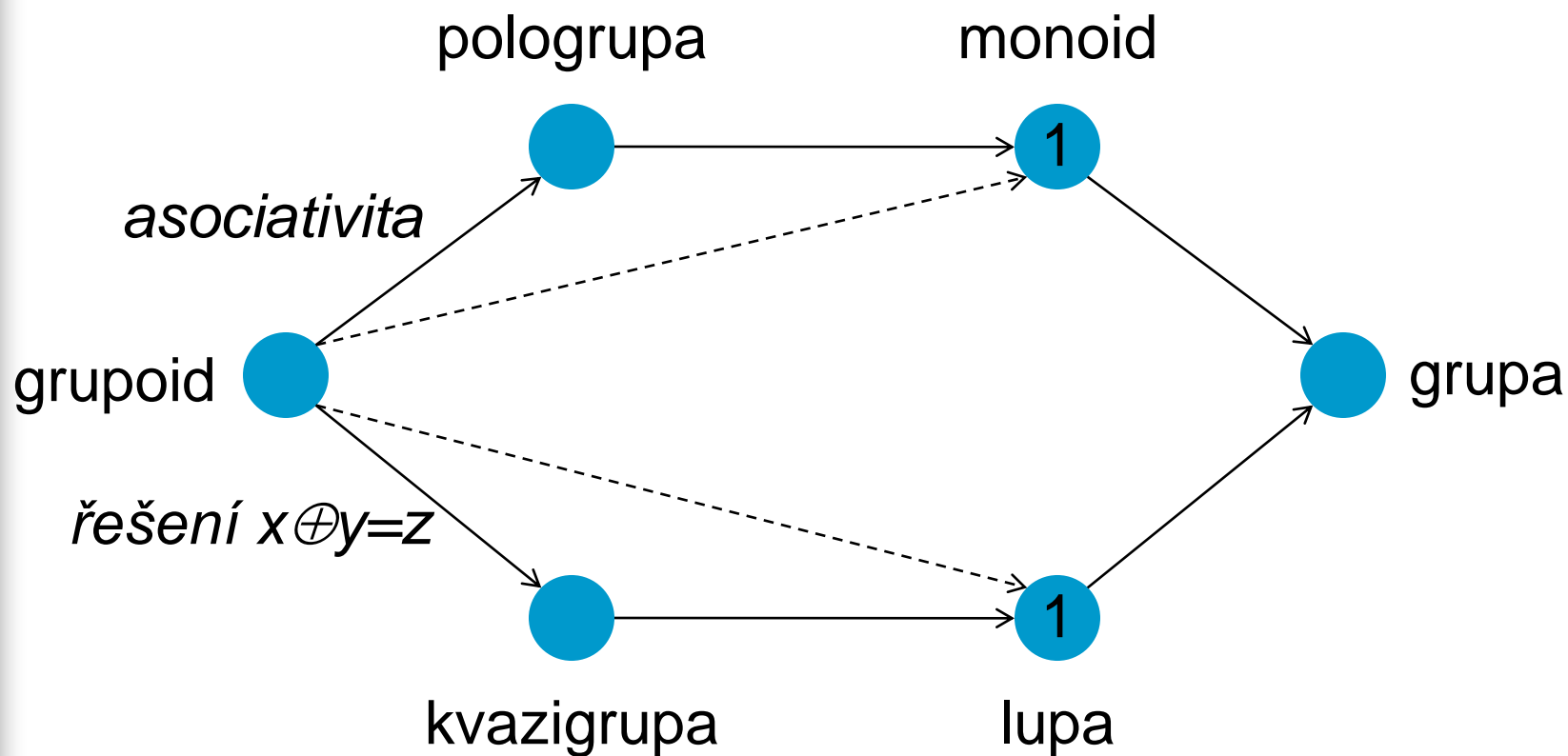
$$\gg z \oplus x = z$$

- levý/pravý inverzní prvek

$$\gg x \oplus y = e$$



# Od grupoidu ke grupě



# Důležité pojmy

- Imperativní, deklarativní
- Lambda kalkul
  - Definice
  - Konverze/redukce
  - Substituce
- Kompozice, inverze, asociativita, komutativita
- Pologrupa, monoid, grupa





# Úkoly



- Navrhňte reprezentaci čísel v lambda kalkulu
- Navrhňte běžné operace nad těmito čísly
- Ukažte, že platí obecně známé rovnosti:
  - $x+y = y+x$
  - $x+0 = x$
  - ...



# Haskell - úvod, seznamy

# Cíle oddílu



- Přehled základních datových typů jazyka Haskell
- Deklarace funkcí v jazyce Haskell
- Rekurze
- Práce se seznamy

# Co a kde ...

## ■ Haskell

- překladač (kompilátor) – **ghc**, **nhgc**, **jhc**
- interaktivní kompilátor – **ghci**
- <http://www.haskell.org>

## ■ *Hugs*

- *interpret virtuálního kódu*
- <http://www.haskell.org>

## ■ *Linux, Q:\FLP\...*



# Základní datové typy

## ■ Int

- 45, 890

## ■ Char

- 'a', 'X'

## ■ Bool

- True, False

## ■ Float

- 3.14159, 234.2

# Seznamy

## ■ Konstruktor

- $:$
- $[]$

## ■ Prázdný seznam

- $[] :: [a]$

## ■ Neprázdný seznam

- $1:2:3:[] \sim [1,2,3] :: [\text{Int}]$
- $(x:xs)$

# N-tice

## ■ Konstruktor

- ,
- (a,b,c,d, ... )

## ■ Příklad

- $(1, 2) :: (\text{Int}, \text{Int})$
- $(1, ['a', 'b']) :: (\text{Int}, [\text{Char}])$



# Funkce

## ■ Typ

- $f :: a \rightarrow b$

## ■ Signatura typu

- $1 :: \text{Int}$
- $(1, ['a', 'b']) :: (\text{Int}, [\text{Char}])$



# Deklarace funkcí

- Využití
  - *Rovnic*
  - *Unifikace vzorů (pattern matching)*
- $f \text{ <pat1 1> <pat1 2> \dots = <rhs1>}$   
 $f \text{ <pat2 1> <pat2 2> \dots = <rhs2>}$   
...



# Proměnná ve funkci

- `square x = x * x`
- `add x y = x + y`

# Konstanta ve funkci

- $\text{not True} = \text{False}$   
 $\text{not False} = \text{True}$
- $\text{sgn } 0 = 0$   
 $\text{sgn } n = \text{if } n < 0 \text{ then } -1 \text{ else } 1$

# Seznam ve funkci

- $\text{length } [] = 0$

$\text{length } (x:xs) = 1 + \text{length } xs$

- závorky  
určují  
prioritu

priorita aplikace  
je nejvyšší



# N-tice ve funkci

- plus  $(x,y) = x+y$
- swap  $(x,y) = (y,x)$



# Vzor typu $n+k$ ve funkci

- $\text{fact } 0 = 1$   
 $\text{fact } (n+1) = (n+1) * \text{fact } n$

# Pojmenování části vzoru ve funkci

- $\text{duphd } p@(x:xs) = x:p$
- $\text{merge } [] \text{ } l2 = l2$   
 $\text{merge } l1 \text{ } [] = l1$   
 $\text{merge } l1 @(x:xs) \text{ } l2 @(y:ys) =$   
    if  $x < y$   
        then  $x:\text{merge } xs \text{ } l2$   
        else  $y:\text{merge } l1 \text{ } ys$

# Anonymní proměnná ve funkci

- $\text{hd } (x:_) = x$
- $\text{ziphd } (x:_) (y:_) = (x,y)$



# Strážené rovnice ve funkci

- fact n | n < 2 = 1  
| otherwise = n \* fact (n-1)
- sgn n | n < 0 = -1  
| n > 0 = 1  
| otherwise = 0

# Lokální funkce ve funkci

- $\text{sumsqr } x \ y = xx + yy$   
    where  $\text{sqr } a \ b = a*b$   
         $xx = \text{sqr } x \ x$   
         $yy = \text{sqr } y \ y$

# Prefixový & infixový zápis funkce

- $\text{mod} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

- $(+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

- Infix

- $5 \text{ `mod` } 3$   
 $1 + 1$

- Prefix

- $\text{mod } 5 \ 3$   
 $(+) \ 1 \ 1$

# Částečná aplikace funkce

- $\text{inc} = (+) 1$ 
  - $\text{inc } x = (+) 1 \ x$
  - $\backslash V . E \ V = E$
- $\text{inc}' = (1+)$



# Odvození typů

- Unifikace – kontrola výskytu
- Známé „pravdy“
  - obecný typ funkce
  - typ konstant
  - typy již odvozené
  - pravidla odvození typu pro danou situaci
- Hledá se nejobecnější typ

# Příklad 1

- $f :: a \rightarrow b \rightarrow (a, b)$ 
  - $(f (3::Int)) :: a \rightarrow (Int, a)$
  - $(f (3::Int) (3.5::Float)) :: (Int, Float)$
- $g :: a \rightarrow a \rightarrow (a, a)$ 
  - $(if \dots then f else g) :: a \rightarrow a \rightarrow (a, a)$
  - $(\backslash x \rightarrow if \dots then f x else g \text{ „Ahoj“}) ::$   
 $String \rightarrow String \rightarrow (String, String)$

## Příklad 2

- $f :: a \rightarrow [a]$   
 $g :: [a] \rightarrow [a]$ 
  - $(\text{if } \dots \text{ then } f \text{ else } g) :: \text{error}$
- $\text{if } f \ x \ y \text{ then } g \ y \ x \text{ else } h \ (g \ y)$ 
  - $x :: a$   
 $y :: b$   
 $f :: a \rightarrow b \rightarrow \text{Bool}$   
 $g :: b \rightarrow a \rightarrow c$   
 $h :: (a \rightarrow c) \rightarrow c$

# Rekurzivní programování

## ■ Zpětně rekurzivní funkce

- $f\ x = e(f\ x')$
- po návratu z rekurzivního volání se ještě něco počítá

## ■ Dopředně rekurzivní funkce

- $f\ x = f(E)$
- rekurzivní volání je poslední část výpočtu





# Rekurzivní programování (pokr.)

## ■ Lineární rekurze

- ve výrazu je jen jedno rekurzivní volání
- lze převést na cyklus

## ■ Koncová rekurze

- dopředně rekurzivní +lineární
- lze jednoduše přeložit na efektivní cyklus (není třeba uchovávat stav nedokončeného výpočtu)

# Rekurzivní programování (pokr.)

## ■ Příklad

–  $\text{rev} :: [a] \rightarrow [a]$

$O(n^2)$

$\text{rev} [] = []$

$\text{rev} (x:xs) = \text{rev} xs ++ [x]$

–  $\text{reverse}' :: [a] \rightarrow [a]$

$O(n)$

$\text{reverse}' xs = \text{rev}' xs []$

where  $\text{rev}' [] ys = ys$

$\text{rev}' (x:xs) ys = \text{rev}' xs (x:ys)$

# Funkce pro práci se seznamy

- $f [] = \dots$   
 $f (x:xs) = \dots$
- $g [x] = \dots$   
 $g (x:y:ys) = \dots$



# Úkol

- *Vytvořte funkci „sumlist“ pro součet hodnot všech prvků v seznamu celých čísel*
- Jaký je typ funkce?
  - $\text{sumlist} :: [\text{Int}] \rightarrow \text{Int}$
- Jaký je součet prázdného seznamu?
  - $\text{sumlist } [] = 0$

# Úkol (pokr.)

- Jaký je součet neprázdného seznamu (jako funkce součtu seznamu o jeden kratšího)?
  - $\text{sumlist } (x:xs) = x + \text{sumlist } xs$
- Je zaručen konečný počet kroků? Jestliže ano, potom jak?
  - *Domácí úkol*

# Redukce seznamu na hodnotu

- $\text{sumlist} :: [\text{Int}] \rightarrow \text{Int}$   
 $\text{sumlist } [] = 0$   
 $\text{sumlist } (x:xs) = x + \text{sumlist } xs$ 
  - $x_1 + (x_2 + (x_3 + \dots))$
- $\text{concat}' :: [[a]] \rightarrow [a]$   
 $\text{concat}' [] = []$   
 $\text{concat}' (xs:xss) = xs ++ \text{concat}' xss$ 
  - $xs_1 ++ (xs_2 ++ (xs_3 ++ \dots))$

# Změna položek seznamu

- $\text{squareAll} :: [\text{Int}] \rightarrow [\text{Int}]$   
 $\text{squareAll} [] = []$   
 $\text{squareAll} (x:xs) = (x*x) : \text{squareAll } xs$ 
  - $x_1*x_1 : (x_2*x_2 : (x_3*x_3 : \dots))$
- $\text{lengthAll} :: [[a]] \rightarrow [\text{Int}]$   
 $\text{lengthAll} [] = []$   
 $\text{lengthAll} (xs:xss) =$   
 $\quad (\text{length } xs) : \text{lengthAll } xss$ 
  - $\text{length } xs_1 : (\text{length } xs_2 : (\text{length } xs_3 : \dots))$



# Funkce vyššího řádu

- Co mají funkce společného funkce
  - sumlist a concat'
  - squareAll a lengthAll ?
- Čím se tyto funkce liší?
- Polymorfismus + funkce vyššího řádu  
⇒ **abstrakce metody výpočtu**



# Abstrakce (1) - folds

- $\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$   
 $\text{foldr } f \ z \ [] = z$   
 $\text{foldr } f \ z \ (x:xs) = f \ x \ (\text{foldr } f \ z \ xs)$
- $\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$   
 $\text{foldl } f \ z \ [] = z$   
 $\text{foldl } f \ z \ (x:xs) = \text{foldl } f \ (f \ z \ x) \ xs$
- $\text{concat'' } xss = \text{foldr } (++) \ [] \ xss$
- $\text{sumlist' } xs = \text{foldr } (+) \ 0 \ xs$



# Abstrakce (1) - úkol

- $\text{foldr } (\oplus) \text{ z xs} = \text{foldl } (\oplus) \text{ z xs}$
- KDY ?
- Která varianta je lepší?

# Abstrakce (2) - map

- $\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$   
 $\text{map } f [] = []$   
 $\text{map } f (x:xs) = f x : \text{map } f xs$
- $\text{squareAll}' \text{ } xs = \text{map } \text{sq } xs$   
    where  $\text{sq } x = x * x$
- $\text{lengthAll}' \text{ } xss = \text{map } \text{length } xss$

# Abstrakce - příklady

- $\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$   
 $\text{filter } \_ [] = []$   
 $\text{filter } p (x:xs)$ 
  - $\quad | p x \quad \quad \quad = x:xs'$
  - $\quad | \text{otherwise} = xs'$ $\quad \text{where } xs' = \text{filter } p \ xs$
- $\text{getEven} :: [\text{Int}] \rightarrow [\text{Int}]$   
 $\text{getEven } xs = \text{filter even } xs$

# Generátory seznamů

- Matematický zápis definice množiny
  - $\{x \mid x \in \mathbf{N}, \text{ even } x\}$
- Zápis v jazyce Haskell
  - `[výraz | kvalifikátory]`



# Kvalifikátory

## ■ Generátor

- $\text{pattern} \leftarrow \text{expression}$ 
  - vybírá postupně prvky seznamu *expression* a unifikuje je se vzorem *pattern*

## ■ Filtr

- predikát ( $a \rightarrow \text{Bool}$ )

# Příklady

- $[x \mid x \leftarrow xs] \equiv xs$
- $[f\ x \mid x \leftarrow xs] \equiv \text{map } f\ xs$
- $[x \mid x \leftarrow xs, p\ x] \equiv \text{filter } p\ xs$
- $[e \mid x \leftarrow xs, y \leftarrow ys, \dots] \equiv$   
 $\text{concat } [[e \mid y \leftarrow ys, \dots] \mid x \leftarrow xs]$

# Příklady (pokr.)

- $[d \mid d \leftarrow [1..n], n \text{ `mod` } d == 0] =$   
 $[d \mid d \leftarrow \text{filter } (\lambda d \rightarrow n \text{ `mod` } d == 0) [1..n]]$   
 $=$
- $\text{filter } (\lambda d \rightarrow n \text{ `mod` } d == 0) [1..n]$



# Příklady (pokr.)

- $[(m,n) \mid m \leftarrow [1..3], n \leftarrow [4,5]]$   
 $= [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]$
- $[' ' \mid n \leftarrow [1..10]] = \text{“} \text{---} \text{”}$
- spaces  $n = [' ' \mid i \leftarrow [1..n]]$

# Důležité pojmy

- Typ, jeho signatura
  - Typová proměnná
- Vzor
- Formy rekurze
  - Lineární, koncová, dopředná, zpětná
- Zastavení rekurze, podmínka
- Funkce vyššího řádu
- Generátory seznamů



# Úkoly



- Vytvořte funkci, která dostane jako argument seznam a vrátí ho tak, že vždy dvojice prvků bude prohozena
- Bude vaše implementace fungovat i nad nekonečnými seznamy?
- Vytvořte nekonečný seznam všech lichých členů Fibonacciho posloupnosti a ověřte správnost řešení



# Uživatelské datové typy, typové třídy, pole

# Cíle oddílu



- Definice a využití vlastních datových typů
- Typové třídy
  - Definice
  - Užití
  - Význam



# Bázové typy

- V každém jazyce
  - `2 :: Int`
  - `'a' :: Char`
  - `3.45 :: Float`
  - *`True :: Bool`*

# Základní - odvozené zázázových

## ■ Běžně v jazyce

– homogenní - seznam

- $[1,2] :: [\text{Int}]$
- $['a','b'] :: [\text{Char}]$
- $[[],[1]] :: [[\text{Int}]]$

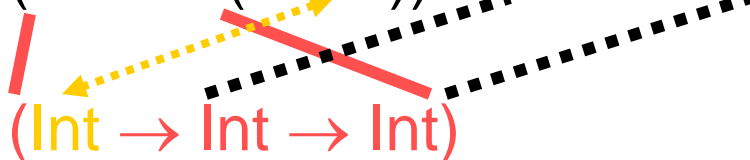
– heterogenní - n-tice

- $(1,'a',5650.4) :: (\text{Int}, \text{Char}, \text{Float})$
- $([23,42],'A') :: ([\text{Int}], \text{Char})$

# Monomorfní funkce

- Operátor aplikace funkce  $\rightarrow$ 
  - zprava asociativní

- $\text{addInt} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \sim \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$

- $(\text{addInt } (1 :: \text{Int})) :: \text{Int} \rightarrow \text{Int}$   




# Polymorfní funkce

## ■ Je číslo $n$ v seznamu?

- $\text{elem} :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Bool}$   
 $\text{elem} \_ [] = \text{False}$   
 $\text{elem } n (x:xs) = \text{if } n == x$   
                            then True  
                            else  $\text{elem } n xs$

# Polymorfní funkce (pokr.)

## ■ Je znak $c$ v seznamu?

- $\text{elemC} :: \text{Char} \rightarrow [\text{Char}] \rightarrow \text{Bool}$   
 $\text{elemC } \_ [] = \text{False}$   
 $\text{elemC } c (x:xs) = \text{if } c == x$   
                          then True  
                          else  $\text{elemC } c xs$

# Polymorfní funkce (pokr.)

- Je objekt daného typu v seznamu?  
(použití algoritmu pro všechny entity)
  - $\text{elem} :: a \rightarrow [a] \rightarrow \text{Bool}$   
 $\text{elem } \_ [] = \text{False}$   
 $\text{elem } y (x:xs) = \text{if } y == x$   
                                  then True  
                                  else elem y xs
  - **$a$  je typová proměnná**

# *Přetěžování?*

- `elemI`      $\sim$  if **n** == x :: Int  $\rightarrow$  Int  $\rightarrow$  Bool
- `elemC`     $\sim$  if **c** == x :: Char  $\rightarrow$  Char ...
- `elem`       $\sim$  if **y** == x :: a  $\rightarrow$  a  $\rightarrow$  Bool
  
- C, Pascal - přetěžování
- Haskell - typové třídy
  - `class Eq a where`  
    `(==) :: a  $\rightarrow$  a  $\rightarrow$  Bool`

# Typové třídy

- Typ ***a*** je instancí třídy ***C***, je-li pro něj definována vymezená množina operací
  - class Eq a where
$$(==) :: a \rightarrow a \rightarrow \text{Bool}$$
- Potom
  - elem :: (Eq a) => a → [a] → Bool
  - Pro všechny typy a, které jsou instancí třídy Eq, má operace elem typ  $a \rightarrow [a] \rightarrow \text{Bool}$

# Instance typové třídy

- instance Eq Int where  
    x==y = intEq x y  
          *// metoda (definice operace)*
- instance (Eq a) => Eq [a] where  
    []==[] = True  
    (x:xs)==(y:ys) = x==y && xs==ys  
    \_==\_ = False

# Nezbytné/odvozené metody

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

# Dědičnost

## ■ Jednoduchá

- `class (Eq a) => Ord a where`  
    `(<),(<=),(>=),(>) :: a -> a -> Bool`  
    `max,min :: a -> a -> a`

## ■ Vícenásobná

- `class (Eq a, Show a) => C a where`  
    ...



# Typové třídy vyššího řádu

- $\text{map } f [] = []$   
 $\text{map } f (x:xs) = f x : \text{map } f xs$
- `class Functor f where`  
 $\text{fmap} :: (a \rightarrow b) \rightarrow f a \rightarrow f b$
- $f$  je označení typu (jméno typu)
- pro případ seznamu jsou to hranaté závorky



# Uživatelské datové typy

## ■ Typová synonyma

- `type ComplexF = (Float, Float)`
- `type Matrix a = [[a]]`

## ■ Jednoduché typy

- `newtype ComplexC = ReIm (Float,Float)`
- `newtype MatrixC a = Matrix [[a]]`

# Komplexní datové typy

- **data** Datový\_typ  $a_1 \dots a_n =$   
Constr<sub>1</sub> | ... | Constr<sub>n</sub>
  - Datový typ = *jméno typu*
  - $a_i$  = *typové proměnné (nepovinné)*
  - Constr<sub>i</sub> = *konstruktory typu*

# Výčtové typy

- data Color = Red | Green | Blue
  - Red, Green, Blue jsou nulární konstruktory
- isRed :: Color → Bool
  - isRed Red = True
  - isRed \_ = False



# Rozšířené typy

- data Color' =  
    Red |  
    Green |  
    Blue |  
    Grayscale Int
- Grayscale :: Int  $\rightarrow$  Color'



# Parametrické typy

- data Point a = Pt a a
- (Pt 2 4) :: Point Int
- (Pt 3.4 4.5) :: Point Float
- (Pt (Pt 2 5) (Pt 4 5)) :: Point (Point Int)
- **Pt 4 5.6 - error**

# Rekurzivní datové typy

- data Tree a =  
    Lf |  
    Nd a (Tree a) (Tree a)
- inOrd :: Tree a → [a]  
    inOrd Lf = []  
    inOrd (Nd x l r) = inOrd l ++ (x:inOrd r)

# Datové typy a typové třídy

- instance Eq a => Eq (Tree a) where  
Lf == Lf = True  
(Nd x l1 r1) == (Nd y l2 r2) =  
x==y && l1==l2 && r1==r2  
\_==\_ = False
- instance Functor Tree where  
fmap \_ Lf = Lf  
fmap f (Nd x l1 l2)=  
Nd (f x)(fmap f l1)(fmap f l2)



# Datové typy a typové třídy

- instance Show a => Show (Tree a) where  
showsPrec \_ Lf = (++) "Lf"  
showsPrec \_ (Nd a Lf Lf) =  
    ("Nd "++).shows a . ((++) " Lf Lf")  
showsPrec \_ (Nd a Lf r) =  
    ("Nd "++).shows a . ((++) " Lf (") . shows r . (')' :)  
showsPrec \_ (Nd a l Lf) =  
    ("Nd "++).shows a . ((++) " (").shows l . ((++) ")" Lf")  
showsPrec \_ (Nd a l r) =  
    ("Nd "++).shows a . ((++) " (") . shows l .  
        ((++) ")" (") . shows r . (')' :)

# *Poznámka*

- *showsPrec p t*
  - *p ~ precedence*
  - *t ~ typ*
- *p udává prioritu a podle toho řadí do tisku závorky či nikoliv*
  - *vyšší p tím vyšší je priorita ,kolem', tj. je třeba to, co převádím, dát do závorek*
- *podobně existuje readsPrec pro třídu Read*



# Pro/proti vlastního přístupu

## ■ Pro

- vše je pod kontrolou
- možno zavést instanci i u složitých typů

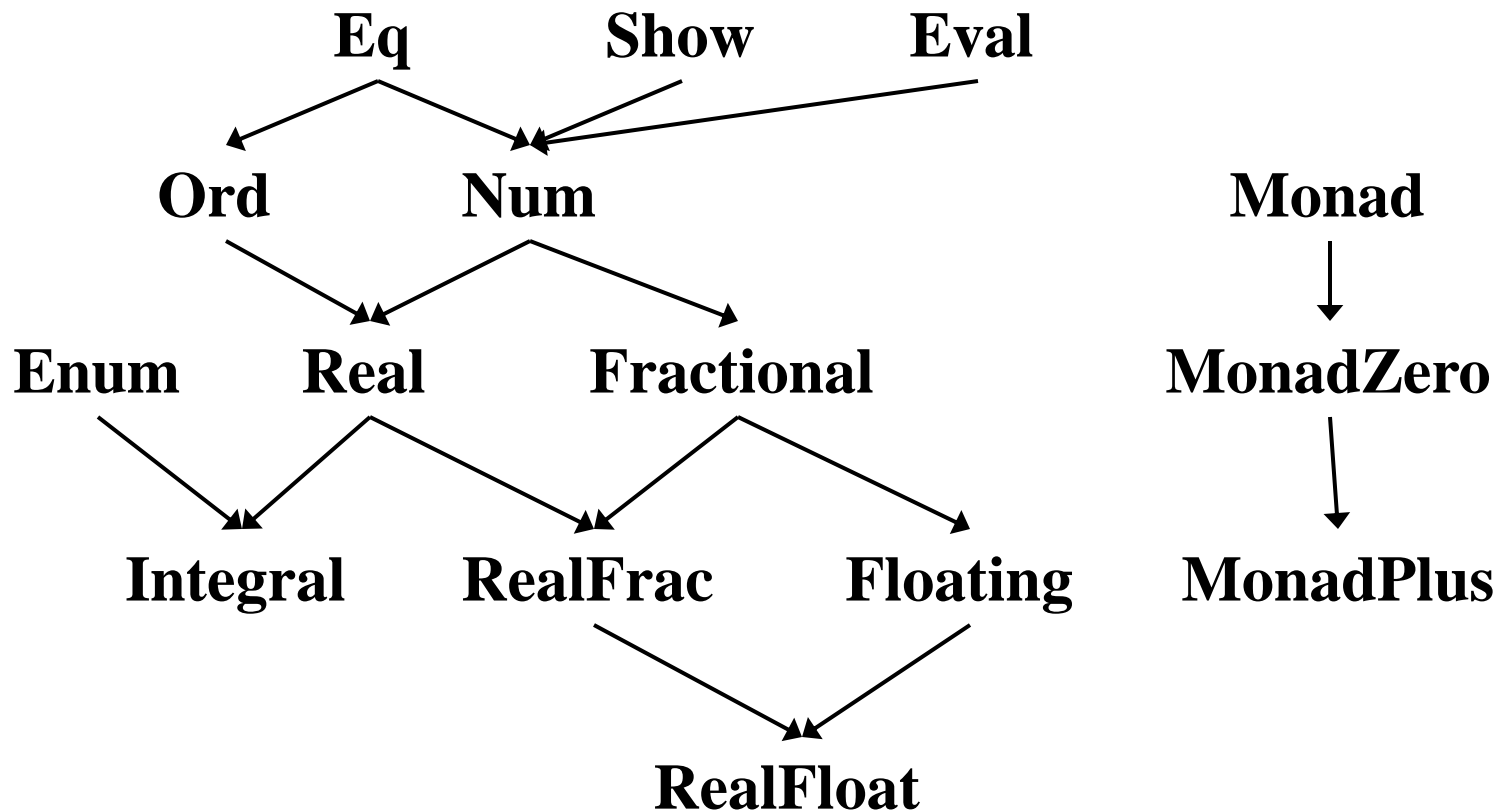
## ■ Proti

- často pracné
- implementace s využitím priorit není triviální
- klíčové slovo **deriving**

# Použití deriving

- `data Color = Red | Green | Blue`  
`deriving (Eq,Ord,Enum,Bounded,Show,Read)`
- `data Tree a =`  
    `Lf |`  
    `Nd a (Tree a) (Tree a)`  
`deriving (Eq,Show)`

# Přehled tříd v Haskellu



**Read   Functor   Bounded**

# Pole

## ■ Indexy

- class (Ord a) => Ix a where  
    range :: (a,a) → [a]  
    index :: (a,a) → a → Int  
    inRange :: (a,a) → a → Bool

- range (0,4) ~> [0,1,2,3,4]  
    range ((0,0),(1,2)) ~>  
        [(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]
- index (1,9) 2 ~> 1  
    index ((0,0),(1,2)) (1,1) ~> 4

# Vytvoření pole

## ■ Vytvoření monolitického pole

- $\text{array} :: (\text{Ix } a) \Rightarrow (a,a) \rightarrow [(a,b)] \rightarrow \text{Array } a \ b$ 
  - spodní/horní limit
  - inicializace pole, páry index hodnota

## ■ Příklad

- $\text{squares} = \text{array } (1,100) [(i,i*i) \mid i \leftarrow [1..100]]$
- $\text{squares}!7 \sim> 49$
- $\text{bounds squares} \sim> (1,100)$

# Pole s kumulovanými hodnotami

## ■ Funkce pro vytvoření

- `accumArray :: (Ix a) =>`  
`(b → c → b) → b → (a,a) → [Assoc a c] → Array a b`
  - kumulační funkce
  - inicializační hodnota (pro každou buňku stejná)
  - spodní/horní limit pole
  - seznam asociací (index-hodnota)

## ■ Histogram

- `hist bnds is = accumArray (+) 0 bnds [(i,1) | i ← is, inRange bnds i]`



# Změny hodnot prvků

## ■ Operátor změny

- $(//) :: (Ix\ a) \Rightarrow \text{Array } a\ b \rightarrow [(a,b)] \rightarrow \text{Array } a\ b$

## ■ Příklad (záměna řádků matice)

### ■ `swapRows i i' a =`

```
a // ([ ((i , j), a!(i' , j)) | j ← [jLo..jHi] ] ++  
      [ ((i' , j), a!(i , j)) | j ← [jLo..jHi] ] )  
where ((iLo, jLo), (iHi, jHi)) = bounds a
```

# Důležité pojmy



- Polymorfismus, přetěžování
- Typová třída, instance
- Dědičnost v typových třídách
- Typy:
  - Synonyma, jednoduché, **komplexní**
    - Výčtové, rozšířené, parametrické, rekurzivní

# Úkoly



- Navrhňte vlastní systém tříd a operátorů práci s celými a desetinnými čísly tak, aby operace  $3+4,5$  nevedla na implicitní konverzi
- Implementujte a ověřte funkčnost toho systému
- Navrhňte a implementujte třídu pro práci s vyhledávacími stromy



# Vstupy/výstupy

# Cíle oddílu



- Práce s monadickými třídami
- V/V operace jako monadické třídy



# Monadické třídy

- Základní třídy spojené s monádami (nikoliv monoidy)
  - Functor, Monad, MonadZero, MonadPlus
- Členové těchto tříd
  - IO, seznamy (`[]`), výjimky (`Maybe`)
    - výjimky jsou zkrácené seznamy  
Nothing ~ `[]`  
Just x ~ `[x]`



# Functor

- Definuje funkci *fmap*, která aplikuje jednu operaci na všechny elementy struktury beze změny tvaru struktury
  - $fmap\ id = id$   
 $fmap\ (f \ .\ g) = fmap\ f \ .\ fmap\ g$

# Monad

- class Monad m where  
    ( $\gg=$ )  $:: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$   
    ( $\gg$ )  $:: m\ a \rightarrow m\ b \rightarrow m\ b$   
    return  $:: a \rightarrow m\ a$   
  
     $m\ \gg\ k = m\ \gg= (\backslash\_ \rightarrow k)$



# Syntaktické ekvivalenty

- $[ (x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5,6] ]$
- ```
do  x ← [1,2,3]
   y ← [4,5,6]
  return (x,y)
```
- $[1,2,3] \gg = (\backslash x \rightarrow [4,5,6] \gg = (\backslash y \rightarrow \text{return } (x,y)))$

# Sémantické ekvivalenty

- $\text{return } a \gg= k \quad = k \ a$
- $m \gg= \text{return} \quad = m$
- $\text{map } f \ xs \quad = xs \gg= \text{return} . f$
- $m \gg= (\backslash x \rightarrow k \ x \gg= h)) \quad = (m \gg= k) \gg= h$

# MonadZero

## ■ Definice

- `class (Monad m) => MonadZero m where  
 zero :: m a`

## ■ Vlastnosti

- `m >>= \x → zero`     `= zero`
- `zero >>= m`             `= zero`

# MonadPlus

## ■ Definice

- $\text{class } (\text{MonadZero } m) \Rightarrow \text{MonadPlus } m \text{ where}$   
     $(++) \quad \quad \quad :: m\ a \rightarrow m\ a \rightarrow m\ a$

## ■ Vlastnosti

- $m ++ \text{zero} = m$
- $\text{zero} ++ m = m$

# Akce ve FP

- V imperativních jazycích je program tvořen posloupností *akcí* (čtení a nastavování globálních proměnných, čtení a zápis souborů)
- Haskell: oddělení akcí od čistě funkcionálního kódu pomocí *monadických operátorů*
- Akce je funkce s výsledkem typu **IO** a

# Příklady akcí

- `getChar :: IO Char`                      `-- akce vracející znak`
- `putChar :: Char → IO ()`                `-- akce nevracející nic`
- `return :: a → IO a`                      `-- hodnota => akce`
- `ready :: IO Bool`  
    `ready = do c ← getChar`  
                    `return (c=='y')`                `-- !! return !!`

# Funkce main

- představuje hlavní program; je to akce, která nic nevrací
  - `main :: IO ()`  
`main = do    c ← getChar`  
`putChar c`

# Čtení řádku textu

- Přečteme znak, je-li to konec řádku, vrátíme prázdný řetězec; jinak přečteme zbytek řádku, spojíme ho s načteným znakem a vrátíme
  - `getLine :: IO String` -- akce vracející řetězec  
`getLine = do x ← getChar`  
    if `x == '\n'` then return ""  
    else do `xs ← getLine`  
            return (x:xs)



# Vypsání řetězce

- Na všechny řetězce aplikujeme funkci *putChar*, např. funkcí *map*
  - *map putChar s* je typu `[IO()]`, tj. seznam akcí => **není to akce**
  - je třeba ještě použít funkci
    - `sequence :: Monad m => [m a] → m [a]`
  - `putStr :: String → IO()`  
`putStr s = sequence (map putChar s)`

# Výjimky

- Datový typ a konstruktor *GHC.IO.Exception*
  - **IOError** ~ **IOException**
    - `ioe_type` – položka
  - **IOErrorType**
    - `EOF`, `NoSuchThing`, ...



# Výjimky

- Ke každé výjimce existuje funkce *System.IO.Error*
  - **isXXXError :: IOError → Bool**
    - isEOFError

# Zachycení výjimky

- Výjimku zachytává funkce *catch*
  - $\text{catch} :: \text{IO } a \rightarrow (\text{IOError} \rightarrow \text{IO } a) \rightarrow \text{IO } a$ 
    - první parametr je akce, během které se mají výjimky odchyťovat
    - druhý parametr je funkce, která se zavolá, pokud nastane výjimka - funkce musí vrátit náhradní výsledek
- ! *catch* definováno 2x !
  - Prelude (pro IO) x Exception

# Generování výjimky

- Výjimku akce generuje funkce *ioError*
  - $\text{ioError} :: \text{IOError} \rightarrow \text{IO a}$   
*System.IO.Error*  
*IO*
- Výsledek funkce *ioError* je kompatibilní s **libovolnou akcí** (nezáleží na vrácené hodnotě, typ se přizpůsobí kontextu)



# Příklad č. 1

- Čtení znaku s ignorováním chyb
  - `getChar` = `getChar` 'catch' (`\_ → return '\n'`)
  - pokud při čtení nastane chyba (např. konec souboru), vrátí se znak konce řádku
  - neodlišuje konec souboru od ostatních chyb

## Příklad č. 2

- Čtení znaku s rozlišením konce souboru
  - `getChar' = getChar 'catch' eofHandler`  
    where `eofHandler e =`  
        if `isEOFError e` then return `'\n'`  
        else `ioError e`
  - pokud se při čtení dostaneme na konec souboru, vrací se znak konce řádku, jinak se vzniklá výjimka vygeneruje znovu

# Práce se soubory

## ■ Otevření a uzavření souboru

- `type FilePath = String`
- `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode`
- `openFile :: FilePath → IOMode → IO Handle`
- `hClose :: Handle → IO ()`





# Práce se soubory

## ■ Čtení ze souboru

- `hGetChar :: Handle → IO Char`
- `stdin, stdout, stderr :: Handle`
- `getChar = hGetChar stdin`
- `hGetContents :: Handle → IO String`



# Práce se soubory

- Funkce začínající na ,h‘ dostávají jako první parametr handle na otevřený soubor, varianty bez ,h‘ pracují se standardními soubory
- Funkce *hGetContents* (jako i ostatní) se vyhodnocuje ,lazy‘, tj. skutečný vstup dat je požadován až tehdy, když se výsledná hodnota začne vyhodnocovat



# Příklad

- Program pro kopírování souboru
- Funkce *opf* se zeptá na jméno vstupního, resp. výstupního souboru a tento soubor otevře v odpovídajícím režimu
- Pokud se soubor nepodaří otevřít, opakuje se dotaz na jméno

# Příklad (pokr.)

```
import IO
```

```
main = do fromHandle ← opf "Copy from: " ReadMode  
          toHandle ← opf "Copy to: " WriteMode  
          contents ← hGetContents fromHandle  
          hPutStr toHandle contents  
          hClose toHandle  
          hClose fromHandle  
          putStr "Done"
```

# Příklad (pokr.)

`opf :: String → IO Mode → IO Handle`

`opf prompt mode =`

`do putStr prompt`

`name ← getLine`

`catch (openFile name mode)`

`(\_ → do putStr ("Can't open " ++ name ++ "\n")`

`opf prompt mode)`



# Case study - kalkulátor

## ■ Zadání:

- Sestrojte program, který čte ze vstupu jednoduché výrazy (i přiřazovací) a hned je vyhodnocuje.
  - paměť
  - *parser - příští přednáška*
  - zpracování vstupu

# Výrazy

- module Expr  
    (Expr(Lit, Var, Op),  
    Ops(Add, Mul, Sub, Div, Mod),  
    Var) where

data Expr = Lit Int | Var Var | Op Ops Expr Expr

data Ops = Add | Sub | Mul | Div | Mod

type Var = Char

# Paměť

- module Store  
    ( Store, initial, value, update ) where

import Expr

newtype Store = Sto (Var  $\rightarrow$  Int)

initial :: Store

initial = Sto ( $\backslash v \rightarrow 0$ )



# Paměť (pokr.)

- $\text{value} :: \text{Store} \rightarrow \text{Var} \rightarrow \text{Int}$   
 $\text{value} (\text{Sto sto}) v = \text{sto } v$

$\text{update} :: \text{Store} \rightarrow \text{Var} \rightarrow \text{Int} \rightarrow \text{Store}$   
 $\text{update} (\text{Sto sto}) v n$   
 $= \text{Sto } (\backslash w \rightarrow \text{if } v == w \text{ then } n \text{ else } \text{sto } w)$

# Parser

- module Parser  
    (Command(Eval, Assign, Null),  
      commLine) where

import Expr

data Command = Eval Expr | Assign Var Expr |  
              Null

commLine :: String → Command

# Kalkulátor

- module Calc(mainCalc) where  
import Expr  
import Store  
import Parser

```
while :: IO Bool -> (a -> IO a) -> a -> IO ()  
while test action initarg = do  
    res <- test  
    if res  
        then action initarg >>=  
            while test action  
        else return ()
```

# Kalkulátor (pokr.)

- $\text{eval} :: \text{Expr} \rightarrow \text{Store} \rightarrow \text{Int}$   
     $\text{eval} (\text{Lit } n) \_ = n$   
     $\text{eval} (\text{Var } v) \text{ st} = \text{value st } v$   
     $\text{eval} (\text{Op op } e1 \ e2) \text{ st} = \text{opVal op } v1 \ v2$   
        where  $v1 = \text{eval } e1 \ \text{st}$   
               $v2 = \text{eval } e2 \ \text{st}$   
     $\text{opVal Add } v1 \ v2 = v1 + v2$   
     $\text{opVal Mul } v1 \ v2 = v1 * v2$   
     $\text{opVal Sub } v1 \ v2 = v1 - v2$   
     $\text{opVal Div } v1 \ v2 = v1 \ `div` v2$   
     $\text{opVal Mod } v1 \ v2 = v1 \ `mod` v2$

# Kalkulátor (pokr.)

- $\text{command} :: \text{Command} \rightarrow \text{Store} \rightarrow (\text{Int}, \text{Store})$   
   $\text{command Null st} = (0, \text{st})$   
   $\text{command (Eval e) st} = (\text{eval e st}, \text{st})$   
   $\text{command (Assign v e) st} = (\text{val}, \text{newSt})$   
    where  $\text{val} = \text{eval e st}$   
           $\text{newSt} = \text{update st v val}$

# Kalkulátor (pokr.)

- $\text{calcStep} :: \text{Store} \rightarrow \text{IO Store}$   
calcStep st =  
    do line  $\leftarrow$  getLine  
        let comm = commLine line  
            (val, newSt) = command comm st  
        print val  
        return newSt

# Kalkulátor (pokr.)

- $\text{calcSteps} :: \text{Store} \rightarrow \text{IO } ()$   
     $\text{calcSteps st} =$   
        while notEOF calcStep st

$\text{notEOF} :: \text{IO Bool}$

$\text{notEOF} = \text{do res} \leftarrow \text{isEOF}$   
            return (not res)

$\text{mainCalc} = \text{calcSteps initial}$

# Důležité pojmy

- Monadická třída
- Akce
- Výjimka





# Úkoly



- Prostudujte možnosti práce s binárními soubory
- Navrhňte diskovou i paměťovou reprezentaci pro jednoduchý DB systém v Haskellu
- Implementujte jej, zaměřte se na V/V operace, indexaci DB apod.



# Praktické úlohy

# Cíle oddílu

- Ukázka práce v jazyku Haskell





# Reprezentace relační databáze

- Navrhněte reprezentaci relační databáze a implementujte operace pro dotazování

# Tabulky

| Author     |             |
|------------|-------------|
| <i>ssn</i> | <i>name</i> |
| 1          | Čapek       |
| 2          | Němcová     |
| 3          | Clarke      |

| Book        |                       |
|-------------|-----------------------|
| <i>isbn</i> | <i>title</i>          |
| 100         | R.U.R.                |
| 101         | Válka s mloky         |
| 102         | Babička               |
| 103         | 2001: Vesmírná odysea |
| 104         | Rajské fontány        |
| 105         | Setkání s Rámou       |

| Wrote      |             |
|------------|-------------|
| <i>ssn</i> | <i>isbn</i> |
| 1          | 100         |
| 1          | 101         |
| 2          | 102         |
| 3          | 103         |
| 3          | 104         |
| 3          | 105         |

# Návrh reprezentace DB

- Data v tabulce mohou být řetězce, čísla, logické hodnoty nebo nedefinované hodnoty

- data Attribute = 

|      |        |
|------|--------|
| St   | String |
| Num  | Int    |
| Bool | Bool   |
| Null |        |



# Návrh reprezentace DB (pokr.)

- Řádek tabulky je tvořen seznamem hodnot
  - type Tuple = [Attribute]



# Návrh reprezentace DB (pokr.)

- Struktura (schéma) tabulky je dána jmény sloupců
  - type Schema = [String]





# Návrh reprezentace DB (pokr.)

- Tabulka je tvořena schématem a seznamem řádků
  - type Table = (Scheme, [Tuple])



# Návrh reprezentace DB (pokr.)

- Databáze je seznam pojmenovaných tabulek
  - `type Database = [(String, Table)]`

# Příklad databáze

- authors :: Table  
authors = ( ["ssn", "name"],  
          [[Num 1, St "Capek"],  
          [Num 2, St "Nemcova"],  
          [Num 3, St "Clarke"]] )

# Příklad databáze (pokr.)

- books :: Table  
books = ( ["isbn", "title"],  
[[Num 100, St "R.U.R."],  
[Num 101, St "Valka s mloky"],  
[Num 102, St "Babicka"],  
[Num 103, St "2001: Vesmirna ..."],  
[Num 104, St "Rajske fontany"],  
[Num 105, St "Setkani s Ramou"]])

# Příklad databáze (pokr.)

- wrote :: Table  
wrote = ( ["ssn", "isbn"],  
[[Num 1, Num 100],  
[Num 1, Num 101],  
[Num 2, Num 102],  
[Num 3, Num 103],  
[Num 3, Num 104],  
[Num 3, Num 105]] )



# Příklad databáze (pokr.)

- `db :: Database`  
`db = [ ("Author", authors),`  
      `("Book", books),`  
      `("Wrote", wrote) ]`

# Návrh reprezentace dotazu

- Dotaz je výraz nad jmény tabulek s operátory sjednocení, rozdíl, kartézský součin, projekce, selekce, join
  - data Query = Table String
    - | Union Query Query
    - | Difference Query Query
    - | Cross Query Query
    - | Project Schema Query
    - | Select Condition Query
    - | Join Query Query
    - ...

# Návrh reprezentace dotazu (pokr.)

- Podmínka je logický výraz s operátory AND, OR, NOT a relačními operátory
  - data Condition = NOT Condition
    - | AND Condition Condition
    - | OR Condition Condition
    - | EQU Expr Expr
    - | NEQ Expr Expr
    - ...



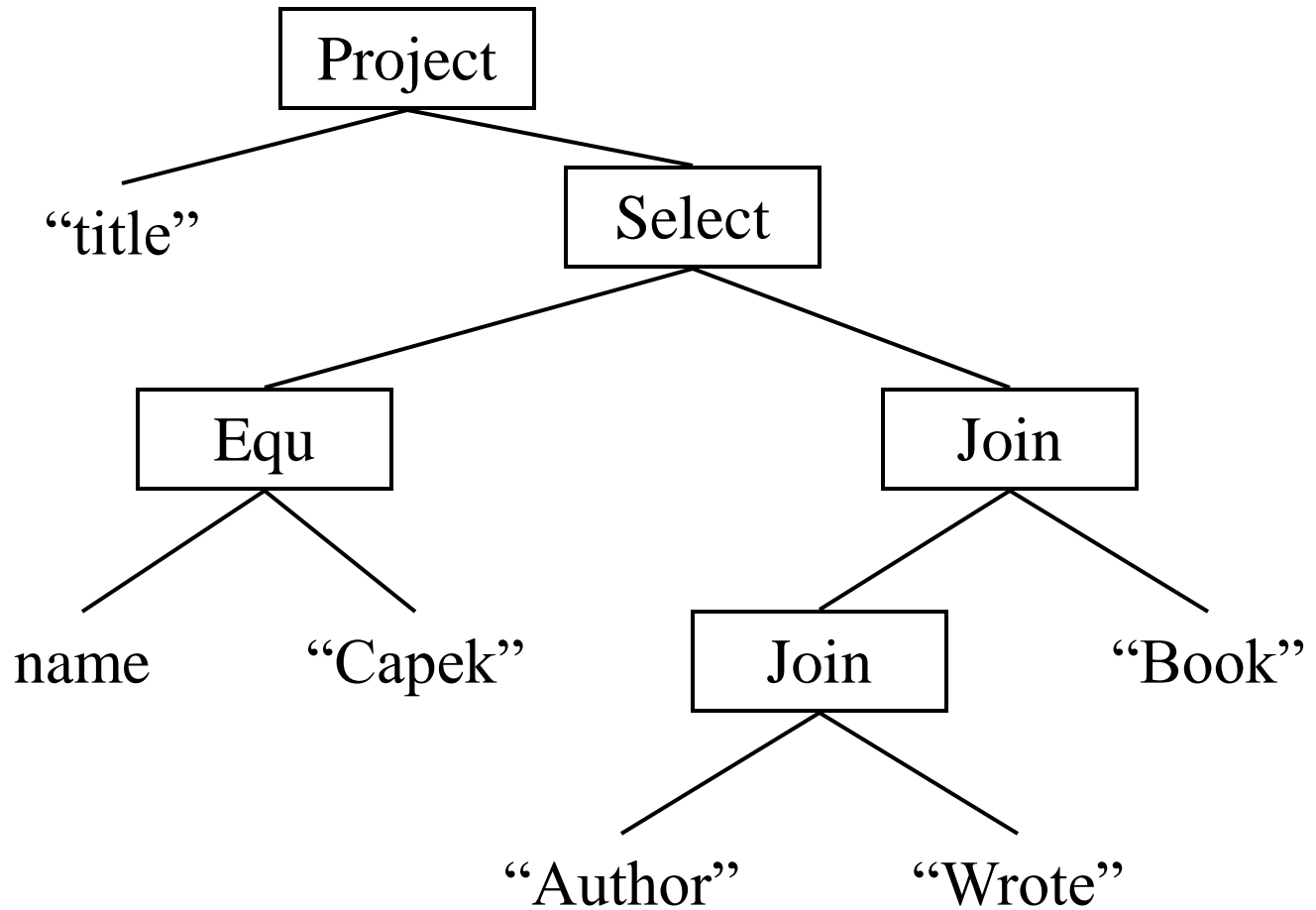
# Návrh reprezentace dotazu (pokr.)

- Aritmetický výraz je tvořen jmény atributů, konstantami a aritmetickými operátory
  - data Expr = Attr String  
| Const Attribute  
| Plus Expr Expr  
| Minus Expr Expr  
...

# Příklad dotazu

- SELECT title  
FROM author JOIN wrote JOIN book  
WHERE name="Capek"
- qry :: Query  
  qry :: Project ["title"]  
    (Select (EQU (Attr "name")  
                (Const (St "Capek"))))  
    (Join (Join (Table "Author")  
                (Table "Wrote"))  
          (Table "Book")))

# Reprezentace dotazu



# Realizace funkcí pro operátory

- $\text{union} :: \text{Table} \rightarrow \text{Table} \rightarrow \text{Table}$   
 $\text{cross} :: \text{Table} \rightarrow \text{Table} \rightarrow \text{Table}$   
 $\text{difference} :: \text{Table} \rightarrow \text{Table} \rightarrow \text{Table}$   
 $\text{join} :: \text{Table} \rightarrow \text{Table} \rightarrow \text{Table}$   
 $\text{project} :: \text{Schema} \rightarrow \text{Table} \rightarrow \text{Table}$   
 $\text{select} :: (\text{Schema} \rightarrow \text{Tuple} \rightarrow \text{Bool})$   
 $\quad \rightarrow \text{Table} \rightarrow \text{Table}$

# Sjednocení relací

- Schémata obou relací musí být totožná
- Výsledkem je seznam řádků obsahující řádky z obou tabulek
  - `union (atts1, tuples1) (atts2, tuples2) =`  
    `(atts1,`  
    `if atts1==atts2 then tuples1 ++ tuples2`  
    `else error "union: incompatible relation")`

# Interpretace dotazu

- Vstupem je dotaz a databáze, výstupem tabulka reprezentující výsledek dotazu
- Vyhodnocení post-order průchodem stromu dotazu
  - $\text{retrieve} = \text{Query} \rightarrow \text{Database} \rightarrow \text{Table}$   
 $\text{retrieve query db} = \text{ret query}$   
where  $\text{ret (Union } q1 \ q2) = \text{union (ret } q1) \ (\text{ret } q2)$   
 $\text{ret (Cross } q1 \ q2) = \text{cross (ret } q1) \ (\text{ret } q2)$   
 $\text{ret (Table } t) = \dots$

# Překlad výrazu

- `type Var = Char`
- `data Expr = Lit Int | Var Var | Op Ops Expr Expr`  
deriving (Show,Read,Eq)
- `data Ops = Add | Sub | Mul | Div | Mod`  
deriving (Enum,Bounded,Eq,Show,Read)
- `2 + 3 ~>`  
`Op Add (Lit 2) (Lit 3)`

# Typy překladačů 1

- type Parse1 a b = [a] → b
- bracket “(xyz” ~> ‘(‘
- number “234” ~> 2 či 23 či 234 ?
- bracket “234” ~> ???



# Typy překladačů 2

- type Parse2 a b = [a] → [b]
- bracket “(xyz” ~> [‘(‘]
- number “234” ~> [2, 23, 234]
- bracket “234” ~> []

# Typy překladačů 3

- type Parse a b = [a]  $\rightarrow$  [(b, [a])]
- bracket "(xyz"  $\sim\>$  [(('(', "xyz")]
- number "234"  $\sim\>$  [(2,"34"), (23,"4"), (234,"")]
- bracket "234"  $\sim\>$  []

# Některé základní překladače

- `none :: Parse a b`  
`none inp = []`
- `succeed :: b → Parse a b`  
`succeed val inp = [(val,inp)]`
- `token :: Eq a => a → Parse a a`  
`token t (x:xs)`
  - `| t==x` `= [(t,xs)]`
  - `| otherwise` `= []``token t []` `= []`

# Některé základní překladače (pokr.)

- $\text{spot} :: (a \rightarrow \text{Bool}) \rightarrow \text{Parse } a$   
   $\text{spot } p \ (x:xs)$   
     $\quad | \ p \ x \quad \quad = [(x,xs)]$   
     $\quad | \ \text{otherwise} \quad = []$   
   $\text{spot } \_ \ [] \quad \quad = []$
- $\text{bracket} = \text{token '('}$
- $\text{dig} = \text{spot isDigit}$
- $\text{token } t = \text{spot } (==t)$

# Spojování základních překladačů

- $\text{alt} :: \text{Parse } a \rightarrow \text{Parse } b \rightarrow \text{Parse } a \rightarrow b$   
 $\text{alt } p1 \ p2 \ \text{inp} = p1 \ \text{inp} ++ p2 \ \text{inp}$
- infixr 5  $>^*>$   
 $(>^*>) :: \text{Parse } a \rightarrow \text{Parse } b \rightarrow \text{Parse } a \rightarrow (b,c)$   
 $(>^*>) \ p1 \ p2 \ \text{inp}$   
 $= [ ((y,z), \text{rem2}) \mid (y, \text{rem1}) \leftarrow p1 \ \text{inp},$   
 $(z, \text{rem2}) \leftarrow p2 \ \text{rem1} ]$

# Příklad

- (bracket `alt` dig) "234"  
~> [] ++ [(2, "34")]  
~> [(2, "34")]
- number "24("  
~> [(2, "4(", (24, "(")]  
bracket "  
~> [("'", ")]  
(number >\*> bracket) "24("  
~> [ ((24, '(', ")]

# Výsledek překlada

- $\text{build} :: \text{Parse } a \ b \rightarrow (b \rightarrow c) \rightarrow \text{Parse } a \ c$   
 $\text{build } p \ f \ \text{inp} = [(f \ x, \text{rem}) \mid (x, \text{rem}) \leftarrow p \ \text{inp}]$
- $(\text{number } \text{'build'} \ \text{digsToNum}) \text{"21a"}$   
 $\sim> [(2, \text{"1a3"}), (21, \text{"a3"})]$
- $\text{list} :: \text{Parse } a \ b \rightarrow \text{Parse } a \ [b]$   
 $\text{list } p = (\text{succeed } []) \text{'alt'}$   
 $((p >^* > \text{list } p) \text{'build'} \ (\text{uncurry } ()))$

# Další užitečné ‘kombinátory’

- `neList :: Parse a b → Parse a [b]`  
`neList p x = filter isEmpty (list p x)`  
    where `isEmpty :: ([a],b) -> Bool`  
          `isEmpty ((_:_),_) = True`  
          `isEmpty _ = False`
- `optional :: Parse a b → Parse a [b]`  
`optional p = (p `build` (\x -> [x])) `alt``  
              `(succeed [])`



# Překladač výrazů

- `parser :: Parse Char Expr`  
`parser = litParse `alt` varParse`  
``alt` opExprParse`
- `varParse :: Parse Char Expr`  
`varParse = spot isVar `build` Var`  
`where isVar :: Char → Bool`  
`isVar x = 'a' <= x && x <= 'z'`

# Překladač výrazů (pokr.)

- $\text{opExprParse} :: [\text{Char}] \rightarrow [(\text{Expr}, [\text{Char}])]$

$\text{opExprParse} =$

    (token '('           >\*>

        parser           >\*>

        spot isOp       >\*>

        parser           >\*>

    token ')' )

    `build` makeExpr

- $\text{makeExpr} :: (\text{a}, (\text{Expr}, (\text{Char}, (\text{Expr}, \text{b})))) \rightarrow \text{Expr}$

$\text{makeExpr } \_ , (\text{e1}, (\text{bop}, (\text{e2}, \_))) =$

    Op (charToOp bop) e1 e2

# Překladač výrazů (pokr.)

- $\text{charToOp} :: \text{Char} \rightarrow \text{Ops}$   
     $\text{charToOp} \text{ '+' } = \text{Add}$   
     $\text{charToOp} \text{ '-' } = \text{Sub}$   
     $\text{charToOp} \text{ '*' } = \text{Mul}$   
     $\text{charToOp} \text{ '/' } = \text{Div}$   
     $\text{charToOp} \text{ '%' } = \text{Mod}$
- $\text{isOp} :: \text{Char} \rightarrow \text{Bool}$   
     $\text{isOp } x = \text{elem } x \text{ "+-*/\%"}$

# Překladač výrazů (pokr.)

- `litParse = ((optional (token '~')) >*>  
              (neList (spot isDigit))) `build`  
              (charListToExpr . uncurry (++))`
- `string2num :: [Char] -> Int`  
`string2num l = doit 0 l`  
    `where doit x [] = x`  
        `doit x (c:cs) = doit (x*10 + toNum c) cs`  
        `toNum c = ord c - ord '0'`
- `charListToExpr :: [Char] -> Expr`  
`charListToExpr ('~':cs) = Lit (- string2num cs)`  
`charListToExpr cs = Lit (string2num cs)`

# Překladač výrazů (pokr.)

- `asgParser :: Parse Char Command`  
`asgParser =`  
    `(varParse >*> token '=' >*> parser) `build``  
        `(\ (var, (_, ex)) -> Assign (unVar var) ex)`  
    where  
        `unVar (Var v) = v`

# Překladač výrazů (pokr.)

- `commLine :: String -> Command`  
`commLine str =`  
    if null pars then Null  
    else if isOKparse pars  
            then Eval \$ fst \$ head pars  
    else if isOKparse asgp then fst \$ head asgp  
    else error "Error in input,"  
    where  
        pars = parser str  
        isOKparse [(\_,l)] = length l == 0  
        isOKparse \_ = False  
        asgp = asgParser str

# Důležité pojmy

- Reprezentace dat
- Modelování
- Skládání elementárních operací
  - Základní překladač



# Úkoly



- Rozšiřte váš DB systém o základní operace jazyka SQL, zejména příkaz SELECT
- Prostudujte HSQL – <http://htoolkit.sourceforge.net>
- Vyberte si jednoduchou stolní hru (á la Člověče nezlob se) a implementujte ji v jazyce Haskell





# Dokazování ve FP

# Cíle oddílu



- Formální důkaz vlastnosti programu v jazyce Haskell
- Analýza efektivity programu
- Využívání „triků“



# Základní principy

- Základem je vlastnost **referenční transparence**
  - „equals can be replaced by equals“
- Princip **strukturální indukce**



# Strukturální indukce nad seznamy

- Nejprve dokážeme, že tvrzení platí pro prázdný seznam []
- Za předpokladu, že tvrzení platí pro seznam  $xs$  (indukční předpoklad) dokážeme, že platí také pro seznam  $(x:xs)$



# Praktické rady

- Krok důkazu je **vždy** tvořen aplikací některé definiční rovnice a to **v libovolném směru**
- Důležitá je volba indukční proměnné
- Závorky hrají jednu z nejvýznamnějších rolí

# Příklad

- Mějme dáno:

- $(++) :: [a] \rightarrow [a] \rightarrow [a]$

- $[] ++ ys = ys$  (1)

- $(x:xs) ++ ys = x:(xs++ys)$  (2)

- **Asociativita:**

- Pro všechny konečné seznamy  $xs$ ,  $ys$  a  $zs$  platí:

- $xs++(ys++zs) = (xs++ys)++zs$

# Důkaz

- Použijeme strukturální indukci přes proměnnou  $xs$ , neboť definice operátoru  $++$  rozvíjí pouze první argument

- $xs = []$

$[] ++ (ys ++ zs) =$  // (1) zleva doprava  
 $ys ++ zs$

$([] ++ ys) ++ zs =$  // (1) zprava doleva  
 $ys ++ zs$

# Důkaz (pokr.)

- **$xs = (a:as)$**

$as++(ys++zs) = (as++ys)++zs$     *// předpoklad*

$(a:as)++(ys++zs) = ((a:as)++ys)++zs$

*// dokázat*

$(a:as)++(ys++zs) =$

*// (2) zleva doprava*

$a:(as++(ys++zs)) =$

*// indukční předpoklad*

$a:((as++ys)++zs) =$

*// 2 zprava doleva*

$(a:(as++ys))++zs =$

*// 2 zprava doleva*

$((a:as)++ys)++zs$

**Q.E.D.**



# Další příklady

- $[] ++ xs = xs ++ []$
- $\text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$
- $\text{take } n \text{ } xs ++ \text{drop } n \text{ } xs = xs$
- $\text{rev } xs = \text{reverse } xs$ 
  - $\text{rev } [] = []$   
 $\text{rev } (x:xs) = \text{rev } xs ++ [x]$
  - $\text{reverse } xs = \text{rev}' \text{ } xs \text{ } []$   
where  $\text{rev}' \text{ } [] \text{ } ys = ys$   
 $\text{rev}' \text{ } (x:xs) \text{ } ys = \text{rev}' \text{ } xs \text{ } (x:ys)$

# Zobecnění cíle důkazu

- Mějme danou funkci:

- $\text{rev} :: [a] \rightarrow [a]$

- $\text{rev } xs = \text{shunt } xs []$

1

- where  $\text{shunt } [] \text{ } ys = ys$

2

- $\text{shunt } (x:xs) \text{ } ys = \text{shunt } xs (x:ys)$

3

- Dokažte, že:

- $\text{rev } (\text{rev } xs) = xs$

# První přiblížení

## ■ `rev (rev (x:xs))`

`= shunt (shunt (x:xs) []) []` // 1

`= shunt (shunt xs [x]) []` // 3

- **Není to vhodný cíl**

## ■ Efekt funkce *shunt*

- `shunt xs ys = (reverse xs) ++ ys`
- dalším otočením: `(reverse ys) ++ xs`

– **Tudíž:** `shunt (shunt xs ys) [] = shunt ys xs`

# Druhý pokus

■  $XS = []$

- $\text{shunt} (\text{shunt} [] \text{ys}) [] = \text{shunt} \text{ys} []$  // 2

■ Indukční krok:

- $\text{shunt} (\text{shunt} (x:xs) \text{ys}) [] = \text{shunt} (\text{shunt} xs (x:\text{ys})) []$  // 3

~ což by mělo dát  $\text{shunt} (x:\text{ys}) xs$

# Druhý pokus (pokr.)

- Potom ale hypotéza musí poskytnout výsledek:
  - $\text{shunt}(\text{shunt } xs \ (x:ys)) \ [] = \text{shunt } (x:ys) \ xs$
- spíše než:
  - $\text{shunt}(\text{shunt } xs \ ys) \ [] = \text{shunt } ys \ xs$
- proto...

# Zesílení indukční hypotézy

- Pro všechny konečné listy **zs**:
  - $\text{shunt}(\text{shunt } xs \text{ } zs) [] = \text{shunt } zs \text{ } xs$
- Dostáváme se tak do místa, kdy **zs** nahrazuje **(x:ys)**

# Úspěšný pokus

- Co nyní chceme dokázat?
  - Pro všechna  $zs$ :
    - $\text{shunt}(\text{shunt } xs \text{ } zs) [] = \text{shunt } zs \text{ } xs$
  - pro všechny konečné seznamy  $xs$  pomocí indukce

# S čím začneme

## ■ První krok

- $\forall zs \text{ (shunt (shunt [] zs) [] = shunt zs [])}$

## ■ Indukční krok

- $\forall zs \text{ (shunt (shunt (x:xs) zs) [] = shunt zs (x:xs))}$

## ■ Indukční hypotéza

- $\forall zs \text{ (shunt (shunt xs zs) [] = shunt zs xs)}$



# První krok

- $\forall zs \text{ (shunt (shunt [] zs) [] = shunt zs [])}$

## ■ Důkaz:

- $\text{shunt (shunt [] zs) []} = \text{shunt zs []}$  // 2

# Indukce

- $\forall zs \text{ (shunt (shunt (x:xs) zs) [] = shunt zs (x:xs))}$

## ■ Důkaz:

- $\text{shunt (shunt (x:xs) zs) []} =$  // 3  
 $\text{shunt (shunt xs (x:zs)) []} =$  // hyp  
 $\text{shunt (x:zs) xs} =$  // 3  
 $\text{shunt zs (x:xs)}$

***Q.E.D.***

# Efektivita programů

## ■ Asymptotická analýza

–  $T_f(x)$       počet redukcí potřebných  
pro výpočet hodnoty ( $f\ x$ )

– O-notation      „je řádu nejvýše...“

# Sémantika zápisu

- $T_{\text{reverse}}(xs) = O(n^2)$   
 $T_{\text{rev}}(xs) = O(n)$
- Ze zápisu nijak nevyplývá, že *rev* je vždy rychlejší než *reverse*!
- $g(n) = O(h(n))$   
 $\equiv \exists M: \forall n > 0: |g(n)| \leq M \cdot |h(n)|$

# Co je $g(n)$

- $g(n) =$   
 $a_0 + a_1n + \dots + a_m n$   
 $\sim \rightarrow g(n) = O(n_m)$
- M z výrazu:  
 $M \cdot |h(n)|$
- je například:  
 $|a_0| + |a_1| + \dots + |a_m|$

# Význam asymptotické analýzy

- Dává výsledky nezávislé na implementaci
  - program nižšího řádu **obvykle** *jede* rychleji
- Někdy je nutný podrobnější rozbor (například u stejného řádu)
  - velikosti konstant úměrnosti, přesný počet redukcí
  - testování se stopkami



# Úkol

- Určete řády funkcí
  - hd
  - last
  - length

# Věty o dualitě (1)

- Tvoří-li + a a monoid a je-li xs konečný seznam, pak platí:
  - $\text{foldr } (\underline{+}) \underline{a} \text{ xs} = \text{foldl } (\underline{+}) \underline{a} \text{ xs}$
  - foldl a foldr definují nad monoidy stejnou funkci
  - mohou se ale lišit **efektivitou**



## Věty o dualitě (2)

- Necht'  $\underline{+}$  a  $\underline{x}$  a  $\underline{a}$  jsou takové, že pro každé  $x$ ,  $y$  a  $z$  platí:

$$x \underline{+} (y \underline{x} z) = (x \underline{+} y) \underline{x} z$$

$$x \underline{+} \underline{a} = \underline{a} \underline{x} x$$

Potom pro každý konečný seznam  $xs$  platí:

$$\text{foldr } (\underline{+}) \underline{a} \text{ } xs = \text{foldl } (\underline{x}) \underline{a} \text{ } xs$$

# Resumé

- Věta o dualitě (1) je jen speciálním případem věty o dualitě (2), kdy platí:

$$\underline{+} = \underline{X}$$

# Zamyslete se/vyzkoušejte

- $\text{sum}' = \text{foldl } (+) 0$   
 $\text{sum}'' = \text{foldr } (+) 0$
- $\text{reverse}' = \text{foldl } (\backslash l \ e \rightarrow e:l) []$   
 $\text{reverse}'' = \text{foldr } (\backslash l \ e \rightarrow l++[e]) []$
- Naznačte i prostorovou složitost
- Vysvětlete!

# Důležité pojmy

- Referenční transparence
- Strukturální indukce
  - Indukční proměnná
- Asymptotická analýza



# Úkoly



- Studujte problematiku paměťové/prostorové složitosti
- Určete časovou a paměťovou složitost operací ve vaší implementaci DB systému
- Z implementovaných úkolů vyberte 2 – 3 rekurzivní funkce pracující se seznamy a dokažte, že pracují správně



# Paralelismus v jazyce Haskell

# Cíle oddílu



- Mechanismy paralelismu v Haskellu
  - Spuštění paralelního vlákna
  - Synchronizace vláken
  - Převzetí výsledků výpočtu



# Základní vlastnosti

- Vláknoový paralelismus
  - Procesový je dán vlastnostmi OS
  - Podpora OS pro jisté operace nutná
    - Vazba na funkce v jiném jazyce
- Vazba na monadickou třídu IO
  - *Možnost využití i třídy STM*
  - `import Control.Concurrent`
- Využití modifikovatelných paměťových míst





# Oddělení vlákna

- `forkIO :: IO () -> IO ThreadId`
  - Parametrem je operace, jejíž výsledkem je IO operace nad jednotkou
  - Výsledkem je identifikátor vlákna, které se od hlavního oddělilo



# Problém

- Jak zjistím, že vlákno doběhlo
- Jak získám výsledek z daného vlákna
- Jak uvolním ukončená, či běžící vlákna
- ...



# Uvolnění vlákna

- I běžící vlákno (tvrdé uvolnění)
- `killThread :: ThreadId -> IO ()`
  - Identifikátor vlákna
  - IO operace s výsledkem datové jednotky

# Získání výsledku

- Modul: `Control.Concurrent.MVar`
- `newEmptyMVar :: IO (MVar a)`
  - Vrací IO operaci obsahující prázdnou proměnnou, čerstvou, prázdnou
- *`newMVar :: a -> IO (MVar a)`*
  - *Vytvoří proměnnou obsahující již dodanou hodnotu*

# Získání výsledku (pokr.)

- `takeMVar :: MVar a -> IO a`
  - Proměnná s hodnotou
  - IO operace obsahující výslednou hodnotu
  - *Je-li proměnná MVar prázdná, vlákno čeká na jeho naplnění*
  - *Je-li čekajících vláken více, je po naplnění vybráno jedno z nich náhodně*

# Neblokovaný přístup

- `tryTakeMVar :: MVar a -> IO (Maybe a)`
  - Proměnná s očekávanou hodnotou
  - IO akce vracející možný výsledek
    - Nothing
    - Just a



# Vytvoření vlákna – funkce vlákna

- Jeden z parametrů funkce, která tvoří vlákno je MVar
- Funkce vlákna vrátí IO ()
- Funkce vlákna nevolá cizí funkce
  - Viz konec kapitoly
- Poslední operací vlákna je zápis výsledku do předané MVar
  - Chci-li pak vlákno zrušit

# Zápis výsledné hodnoty vlákna

- `putMVar :: MVar a -> a -> IO ()`
  - Proměnná `MVar`, musí být prázdná, jinak vlákno čeká
  - Hodnota, na kterou se má nastavit
  - IO akce s výsledkem datová jednotka



# Zápis se selháním

- Více vláken se zdrojem dat
- Všechny výsledky platné
- Jedno jediné sběrné místo
- `tryPutMVar :: MVar a -> a -> IO Bool`
  - Proměnná
  - Hodnota, na kterou se má nastavit
  - IO akce – pravda znamená úspěšný zápis



# Další operace s proměnnými MVar

- Čtení
  - Modifikace
  - Test na prázdnotu
  - ...
- 
- *Viz dokumentaci jazyka Haskell*

# Příklad

- Balastní funkce

- Simulace zátěže výpočtu
- Lze dosadit libovolnou jinou

- ```
sumAllSums [] = 0
sumAllSums l@(_:xs) = sumlist 0 l + sumAllSums xs
    where  sumlist res [] = res
           sumlist sr  (v:vs) = sumlist (sr+v) vs
```

# Příklad (pokr.)

## ■ Hlavní funkce programu – začátek:

```
■ main = do
    putStrLn "Starting..."
    mv1 <- newEmptyMVar
    mv2 <- newEmptyMVar
```

Jedná se  
monadickou  
operaci

Dvě výpočetní  
vlákna, dva  
výsledky

# Příklad (pokr.)

## ■ Hlavní funkce programu – spuštění:

■ `main = do`

`...`

`t1 <- forkIO $ mkSum1 mv1`

`t2 <- forkIO $ mkSum2 mv2`

Oddělení vlákn

Vazba na  
proměnné jako  
parametr

# Příklad (pokr.)

- Hlavní funkce programu – sběr a tisk výsledků:

- `main = do`

```
...  
s2 <- takeMVar mv2  
s1 <- takeMVar mv1  
putStrLn $ "Suma1: " ++ show s1  
putStrLn $ "Suma2: " ++ show s2
```

Dva výsledky

Tisk  
výsledných  
hodnot  
(zpracování)

# Příklad (pokr.)

- Hlavní funkce programu – ukončení programu:

```
■ main = do
    ...
    forkIO $ do    killThread t1
                  killThread t2
    putStrLn "Done!"
```

Dvě vlákna

Poslední  
operace  
programu

# Příklad (konec)

- Hlavní funkce programu – lokální funkce:

```
■ main =  
  ...  
  where
```

Uložení  
výsledku

Balastní funkce

Různé  
parametry pro  
porovnání

```
mkSum1 mv = do  
  let res = sumAllSums [1..10000]  
  res `seq` putMVar mv res  
  
mkSum2 mv = do  
  let res = sumAllSums [1..10001]  
  res `seq` putMVar mv res
```





# Odložení vlákna

- `yield :: IO ()`
  - IO akce poskytující datovou jednotku
  - Vynucuje změnu kontextu
  - Kooperativní vícevláknové zpracování

# Příklad

## ■ Tělo funkce main (drobná úprava)

```
■ main = do
    putStrLn "Starting..."
    mv1 <- newEmptyMVar
    mv2 <- newEmptyMVar
    t1 <- forkIO $ mkSum1 mv1
    t2 <- forkIO $ mkSum2 mv2
    yield
    takeRes mv1 mv2
    forkIO $ do    killThread t1
                  killThread t2
    putStrLn "Done!"
```

V hlavním  
vlákně se  
neodehrává  
výpočet

# Příklad (konec)

## ■ Funkce takeRes (vyzvednutí výsledku):

```
■ takeRes mv1 mv2 = do
    mr1 <- tryTakeMVar mv1
    case mr1 of
        Just r1 -> takeMVar mv2
        Nothing -> do
            yield
            mr2 <- tryTakeMVar mv2
            case mr2 of
                Just r2 -> takeMVar mv1
                Nothing -> do
                    yield
                    takeRes mv1 mv2
```

Stále se asi počítá, ale pokud bych potřeboval, mohu i já, nejsem blokován

# Využití u datových struktur

- `data Tree a = Node a (Tree a) (Tree a)`  
                  | Leaf
- Úkol: spočtete hloubku stromu
- Sekvenčně:
  - `sdepth Leaf = 0`
  - `sdepth (Node _ l r) = if ld>rd then ld else rd`  
    where  
      `ld = 1 + sdepth l`  
      `rd = 1 + rdepth r`

# Paralelně

## ■ Špatný přístup:

- ```
pdepth Leaf = return 0
pdepth (Node _ l r) = do
    mv1 <- newEmptyMVar
    mv2 <- newEmptyMVar
    t1 <- forkIO $ pdepth l >>= putMVar mv1
    t2 <- forkIO $ pdepth r >>= putMVar mv2
    ld <- takeMVar mv1
    rd <- takeMVar mv2
    killThread t1
    killThread t2
    return $ 1 + if ld>rd then ld else rd
```

# Paralelně

## ■ Korektní přístup:

- `pdepth' Leaf = return 0`  
`pdepth' (Node _ l r) = do`  
    `mv1 <- newEmptyMVar`  
    `mv2 <- newEmptyMVar`  
    `t1 <- forkIO $ putMVar mv1 $! sdepth l`  
    `t2 <- forkIO $ putMVar mv2 $! sdepth r`  
    `ld <- takeMVar mv1`  
    `rd <- takeMVar mv2`  
    `killThread t1`  
    `killThread t2`  
    `return $ 1 + if ld>rd then ld else rd`



# Zhodnocení

- Špatný přístup
  - Příliš mnoho vláken
  - Využití cílového HW malý
- Korektní přístup
  - Právě 2 výpočetní vlákna
    - Třetí čeká na hodnotu, šlo by vylepšit
  - Vyšší využití HW pro vlastní výpočet

# Vazba na OS

- Pro vazbu na vlákna OS
  - Volání cizích funkcí
  - Vlastnosti OS
  - ...
- `forkOS :: IO () -> IO ThreadId`
  - Funkce vlákna vracející IO akci s výsledkem datové jednotky
  - IO akce s výsledkem identifikátoru vlákna



# Důležité pojmy

- Vlákno
  - Oddělení
  - Synchronizace
  - Předání výsledku
  - Změna kontextu



# Úkoly



- Prostudujte knihovnu Concurrent
- Které z dosud implementovaných funkcí se nabízejí pro paralelizaci (2 výpočetní vlákna)?
- Implementujte paralelní verze pro 2 vámi dosud vytvořené funkce a ověřte jejich správnou činnost i zrychlení výpočtu.



# Denotační sémantika

# Cíle oddílu



- Porozumění denotační sémantice
  - Čtení zápisu
  - Tvorba velmi jednoduchých popisů



# Co je denotační sémantika

- Denotační sémantika programovacího jazyka popisuje význam programu jako *funkci*
  - program: *Input* → *Output*



# Princip kompozicionality

- Význam složené konstrukce je dán kombinací významů jednotlivých složek



# Popisné prostředky

- Pro popis se používá  $\lambda$ -**kalkul**
- Lze jednoduše vyjádřit také pomocí *funkcionálního jazyka*



# Abstraktní syntax

- Popisuje zjednodušenou strukturu programu bez detailů, které nenesou žádnou sémantickou informaci
  - priorita a asociativita operátorů
  - závorky
  - ...





# Abstraktní syntax (pokr.)

- Pro datovou realizaci abstraktní syntaxe programu se používá obvykle stromová struktura - **abstraktní syntaktický strom** (AST)



# Domény

- Syntaktické konstrukce jsou strukturovány do **syntaktických domén**
  - Doména:
    - výrazů - **Exp**
    - příkazů - **Com**
    - deklarací - **Dec**
    - programů - **Prog**
    - ...

# Sémantické funkce

- Přiřazují význam jednotlivým syntaktickým konstrukcím
- Jsou definovány pro každou doménu zvlášť:
  - $e[1 + 1] = 2$
  - $p[i = \text{read}; \text{write } 2 \times i] = \lambda (x : \_).[2 \times x]$



# Výrazy s konstantami

- Významem výrazu je jeho hodnota:

- $e[1+1]=2$

# Reprezentace ve FJ

- `data Exp = Add Exp Exp`  
                  | `Mul Exp Exp`  
                  | `Neg Exp`  
                  | `Num Int`
- `e :: Exp -> Int`  
  `e (Add e1 e2) = (e e1) + (e e2)`  
  `e (Mul e1 e2) = (e e1) * (e e2)`  
  `e (Neg e1)     = 0 - (e e1)`  
  `e (Num x)      = x`



# Výrazy s proměnnými

- Význam výrazu (jeho hodnota) závisí na konkrétních hodnotách proměnných
- Funkce přiřazující proměnným hodnotu (valuační funkce) musí být parametrem sémantické funkce  $e$  - tato funkce modeluje paměť počítače s pojmenovanými buňkami

# Reprezentace ve FJ

- `type Store = String -> Int`  
  `data Exp = ...`  
      `| Var String`

- `e :: Exp -> Store -> Int`

```
e (Add e1 e2) s = (e e1 s) + (e e2 s)
e (Num x)      _ = x
e (Var v)      s = s v
```



# Výrazy s přiřazením

- Hodnoty proměnných (a tedy i valuační funkce) se mohou během vyhodnocení výrazu změnit
- Změna valuační funkce musí být součástí funkční hodnoty sémantické funkce  $e$  (t.j. výsledek musí být uspořádanou dvojicí hodnot)



# Reprezentace ve FJ

- `data Exp = ...  
          | Asgn String Exp`
- `e :: Exp -> Store -> (Int, Store)`
- `e (Asgn v e1) s =  
    let (v1, s') = e e1 s  
        s' ' v' = if v' == v then v1  
                  else s v'  
    in (v1, s' ' )`

# Reprezentace ve FJ (pokr.)

- $e \text{ (Add } e1 \text{ } e2) \text{ } s =$   
     $\text{let } (v1, s') = e \text{ } e1 \text{ } s$   
     $\text{in let } (v2, s'') = e \text{ } e2 \text{ } s'$   
     $\text{in } (v1+v2, s'')$

$e \text{ (Num } x) \quad s = (x, s)$



# Příkazy

- Příkaz neprodukuje (na rozdíl od výrazu) hodnotu, jeho významem je vedlejší efekt - změna stavu programu (např. hodnot proměnných)

# Reprezentace ve FJ

- $\text{Data Com} = \text{Eval Exp}$ 
    - |  $\text{If Exp Com}$
    - |  $\text{While Exp Com}$
    - |  $\text{Seq Com Com}$
  - $c :: \text{Com} \rightarrow \text{Store} \rightarrow \text{Store}$
- $c \text{ (Eval } e1) \text{ } s =$   
     $\text{let } (\_, s') = e \text{ } e1 \text{ } s$   
     $\text{in } s'$

# Reprezentace ve FJ (pokr.)

- $c \text{ (If } e1 \text{ } c1) \text{ } s =$   
     $\text{let } (v1, s') = e \text{ } e1 \text{ } s$   
     $\text{in if } v1 == 0 \text{ then } s'$   
     $\text{else } c \text{ } c1 \text{ } s'$
- $c \text{ (While } e1 \text{ } c1) \text{ } s =$   
     $\text{let } (v1, s') = e \text{ } e1 \text{ } s$   
     $\text{in if } v1 == 0 \text{ then } s'$   
     $\text{else } c \text{ (While } e1 \text{ } c1) (c \text{ } c1 \text{ } s')$
- $c \text{ (Seq } c1 \text{ } c2) \text{ } s = c \text{ } c2 \text{ } (c \text{ } c1 \text{ } s)$



# Vstup a výstup

- Stav programu je tvořen nejen okamžitými hodnotami proměnných, ale také nezpracovaným vstupem a již vyprodukovaným výstupem
- Místo typu **Store** je třeba v předchozích definicích používat typ **State** zahrnující i vstup a výstup programu

# Reprezentace ve FJ

- `type Input = [Int]`  
  `type Output = [Int]`  
  `type State = (Store, Input, Output)`
- `data Exp = ...`  
    `| Read`
- `e :: Exp -> State -> (Int, State)`  
  
  `e Read (s, x:xs, o) = (x, (s, xs, o))`

# Reprezentace ve FJ (pokr.)

- `data Com = ...  
          | Write Exp`

- `c :: Com -> State -> State`

```
c (Write e1) s =  
  let (v1, (s', i, o)) = e e1 s  
  in (s', i, v1:o)
```





# Kontinuace

- Význam „zbytku programu“ od aktuálního místa v programu až po jeho ukončení lze modelovat **kontinuací** - funkcí, která na základě aktuálního stavu vrátí výsledek celého programu



# Kontinuace v denotační sémantice

- Sémantická funkce obdrží jednu nebo více kontinuací, kterým po vyhodnocení může (a nemusí) předat řízení

# Kontinuace výrazu

- Obdrží hodnotu výrazu, stav programu po jeho výpočtu a vrátí výsledek programu
  - `type ECont = Int -> State -> Output`



# Kontinuace příkazu

- Obdrží stav po provedení příkazu a vrátí výsledek programu

- `type CCont = State -> Output`



# Sémantická funkce pro výraz

- Obdrží zpracovávaný výraz, stav před jeho výpočtem a kontinuuaci, které má předat nový stav a výsledek; vrací výsledek celého programu („vyrobený“ obvykle kontinuuací)

# Reprezentace ve FJ

- $e :: \text{Exp} \rightarrow \text{State} \rightarrow \text{ECont} \rightarrow \text{Output}$
- $e \text{ (Add } e1 \text{ } e2) \text{ } s \text{ } ec =$   
     $e \text{ } e1 \text{ } s \text{ } (\backslash v1 \text{ } s' \rightarrow$   
         $e \text{ } e2 \text{ } s' \text{ } (\backslash v2 \text{ } s'' \rightarrow$   
             $ec \text{ } (v1+v2) \text{ } s''))$



# Sémantická funkce pro příkaz

- Obdrží zpracováváný příkaz, stav před jeho provedením a kontinuaci, které má předat nový stav; vrací výsledek celého programu

# Reprezentace ve FJ

- `c :: Com -> State -> CCont -> Output`
- `c (Seq c1 c2) s cc =  
 c c1 s (\s' ->  
 c c2 s' (\s'' -> cc s''))`
- `c (If e1 c1) s cc =  
 e e1 s (\v1 s' ->  
 if v1 == 0 then cc s'  
 else c c1 s' cc)`





# Použití kontinuací

- *Reakce na chybové stavy* - funkce nepředá řízení žádné kontinuaci a tím ukončí vyhodnocení
- *Definice významu skokových příkazů* - při skoku se předá řízení kontinuaci odpovídající cílovému místu skoku



# Použití kontinuací (pokr.)

- *Definice významu volání podprogramu* - kontinuační odpovídající zbytku programu za příkazem volání se uschová pro použití v příkazu **return**



# Modely výpočtu

- Program nejprve provede výpočet a pak zobrazí výsledek nebo chybu
- Program vypisuje průběžně výsledky a na konci oznámí úspěch nebo chybu

# Syntax a sémantika programu

- Program je tvořen posloupností příkazů
  - `type Prog = Com`
- Vstupem programu je posloupnost čísel, výstupem posloupnost čísel zakončená znakem OK nebo chybovou zprávou
  - `type Input = [Int]`  
`data Value =`  
`I Int | OK | Err String`  
`type Output = [Value]`



# Syntax a sémantika programu ...

- Stav programu je tvořen obsahem proměnných a nepřečteným vstupem; na počátku není žádná proměnná definována

# Reprezentace ve FJ

- `type Store = String -> Maybe Int`  
`data State = State {store::Store,`  
`input::Input}`
- `emptyStore :: Store`  
`emptyStore id = Nothing`
- `initialState :: Input -> State`  
`initialState inp =`  
`State store=emptyStore, input=inp`

# Syntax a sémantika programu ...

## ■ Vyhodnocení programu

- $p :: \text{Prog} \rightarrow \text{Input} \rightarrow \text{Output}$

```
p body inp =  
  c body (initialState inp)  
    (\_ -> [OK])
```

# Syntax a sémantika programu ...

## ■ Ošetření nedefinované proměnné

- $e :: \text{Exp} \rightarrow \text{State} \rightarrow \text{ECont} \rightarrow \text{Output}$

```
e (Var v) s ec =  
  case (store s) v of  
    Just v1 -> ec v1 s  
    Nothing  -> [Err ("Undef: "++v)]
```



# Syntax a sémantika programu ...

## ■ Realizace příkazu **Write**

- $c :: \text{Exp} \rightarrow \text{State} \rightarrow \text{CCont} \rightarrow \text{Output}$

```
c (Write e1) s cc =  
  e e1 s (\v1 s' -> (I v1) :  
    (cc  
      (State (store s'), (input s')))))
```

# Důležité pojmy

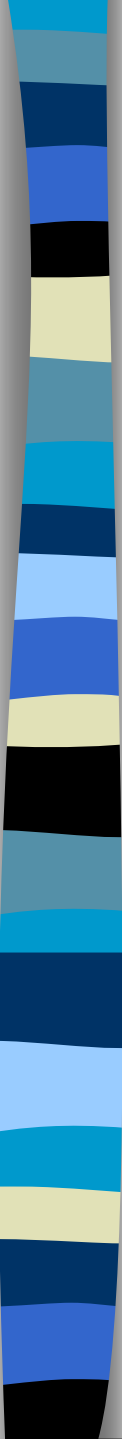
- Denotační sémantika
  - Syntaktická doména
  - Sémantická funkce
- Kontinuace
- Model výpočtu



# Úkoly



- Definujte abstraktní syntaxi a sémantiku podmíněného výrazu if-then-else
  - bez použití kontinuí
  - s kontinuí
- Definujte abstraktní syntaxi a sémantiku příkazu repeat-until
  - bez použití kontinuí
  - s kontinuí





# Prolog - úvod

# Cíle oddílu



- Teoretické pozadí jazyka
- Syntaxe
- Sémantika základních vestavěných operací
- Způsob vyhodnocení u jednoduchých obrátů

# Logické programovací jazyky

## ■ Jsou části predikátové logiky

– program            -        klauzule         $D$

– dotazy            -        cíle                 $G$

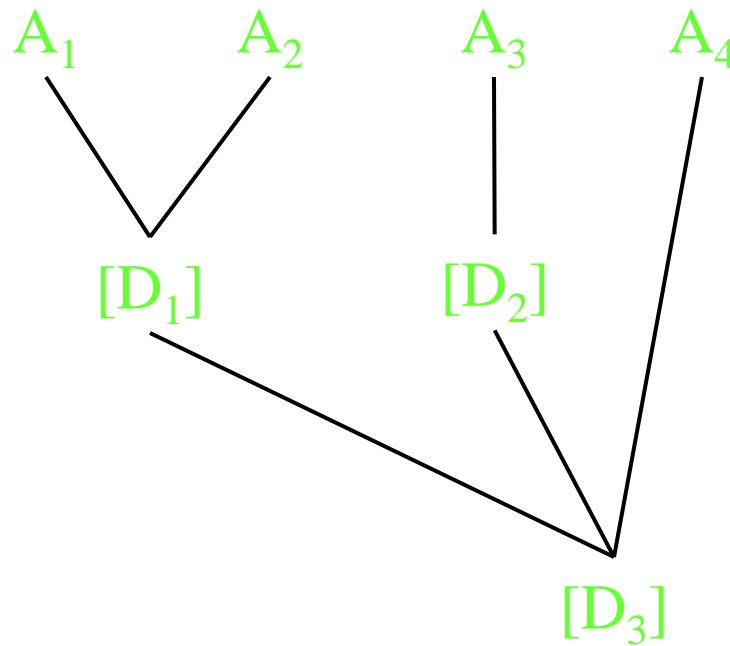
– platné důsledky  $P \vdash Q$ , kde  $P \in D, Q \in G$

– dedukční pravidla

– axiomy

# Důkaz v LP

## ■ Je tvořený důsledky



*axiomy*

*instance  
dedukčních  
pravidel*

*teorém  
(dokazované)  
tvrzení*



# Standardní Prolog

- $D...$  pozitivní klauzule

- $a_0 \sqsubset a_1 \wedge a_2 \wedge \dots \wedge a_n$

- $G...$  konjunkce atomů

- $a_1 \wedge a_2 \wedge \dots \wedge a_n$

- *Hornovy klauzule*

# Syntaxe

## ■ Logická

- $D ::= A \mid A \subset G \mid \forall x.D \mid D \wedge D$   
 $G ::= A \mid G \wedge G$   
 $A ::= \text{atomické formule}$

## ■ Abstraktní

- $D ::= A \mid A :- G. \mid D D$   
 $G ::= A \mid G, G$

# Příklad

- $\text{app}([], X, X).$   
 $\text{app}([E|X], Y, [E|Z]) \text{ :- } \text{app}(X, Y, Z).$
- $\text{?- app}([1], [2], [1, 2]).$

■ *Jak určíme platnost/neplatnost dotazu?*

# Dedukční pravidla

$$\frac{P_1 \vdash A}{P_1 \wedge P_2 \vdash A} \quad \frac{P_2 \vdash A}{P_1 \wedge P_2 \vdash A}$$

$\wedge_L$

*vyhledání  
pravidla*

$$\frac{P \vdash G_1 \quad P \vdash G_2}{P \vdash G_1 \wedge G_2}$$

$\wedge_R$

$$\frac{P \wedge D[\frac{t}{X}] \vdash A}{\forall X. D \in P, P \vdash A}$$

$\forall_L$

*t je  
libovolný  
term*

# Dedukční pravidla (pokr.)

$$\frac{P \vdash G}{A \subset G \in P, P \vdash A}$$

$\supset_L$

$$\frac{}{A \in P, P \vdash A}$$

*axiom*

# Příklad

- $P = \{C_1, C_2\}$   
     $C_1 = \forall x.(\text{app } [] \ x \ x)$   
     $C_2 = \forall exyz.((\text{app } [e|x] \ y \ [e|z]) \subset (\text{app } x \ y \ z))$
- $Q = (\text{app } [1] \ [2] \ [1,2])$
- Ukažte, že  $P \vdash Q$  (najděte důkaz)

# Příklad (pokr.)

- $C_1 = \forall x.(\text{app } [] \ x \ x)$   
 $C_2 = \forall exyz.((\text{app } [e|x] \ y \ [e|z]) \subset (\text{app } x \ y \ z))$
- $Q = (\text{app } [1] \ [2] \ [1,2])$   
 $P = \{C_1, C_2\}$

$$\begin{array}{c}
 \hline
 P, (\text{app } [] [2] [2]) \vdash (\text{app } [] [2] [2]) \\
 \hline
 P \vdash (\text{app } [] [2] [2]) \\
 \hline
 P, (Q \subset \text{app } [] [2] [2]) \vdash Q \\
 \hline
 P \vdash Q
 \end{array}$$

*axiom*  
 $\forall_L \text{pro } C_1$   
 $\supset_L$   
 $\forall_L^4 \text{pro } C_2$

$1/e \ []/x$   
 $[2]/y \ [2]/z$

# Poznámka

## ■ Axiom

- $A \in P, P \vdash A$
- rovnost termů je zajištěna unifikací

## ■ Pravidlo $\forall_L$ :

- $D[t/x]$
- volba termu odložena (lazy)





# Logické programování

## ■ Hlavní myšlenka

- využití počítače k vyvozování důsledků na základě deklarativního popisu

## ■ Postup

- reálný svět → zamýšlená interpretace  
→ logický model  
→ program

## ■ Výpočet

- určení (ne)splnitelnosti **cíle** včetně **substitucí**

# Tvar programu

## ■ Množina klauzulí

- $H \leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_n \quad n \geq 0$ 
  - $H \dots$  hlava
  - $B_1 \wedge B_2 \wedge \dots \wedge B_n \dots$  tělo
- $H \leftarrow \blacksquare$ 
  - fakt

# Tvar programu

## ■ Množina důsledků

- je nekonečná
- Vybíráme si je pomocí dotazů - cílů
- $\square \leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_n \quad n \geq 0$

# Příklad

- `child(tom, john).`  
`child(ann, tom).`  
`child(john, mark).`  
`child(alice, john).`
- $\forall X \forall Y (\text{grandchild}(X, Y)$   
 $\leftarrow \exists Z (\text{child}(X, Z) \wedge \text{child}(Z, Y)))$



- $\forall X \forall Y \forall Z (\text{grandchild}(X, Y)$   
 $\leftarrow \text{child}(X, Z) \wedge \text{child}(Z, Y))$



# Problém

- $\leftarrow \text{child}(\text{mary}, X)$ 
  - nelze odvodit z databáze (programu)
  - odpověď je „no“
  - předpoklad uzavřeného světa
  - lze vyjádřit pouze pozitivní informace



# Prolog

## ■ počátek 70. let

- první implementace v jazyce Fortran
- G. Battani, H. Meloni (Marseille)

## ■ 2. polovina 70. let

- interpret a kompilátor DEC10
- D.H.D. Warren
- rychlost srovnatelná s Lispem



# Vlastnosti jazyka Prolog

- Jazyk s operační sémantikou danou SLD rezolucí prováděnou expanzí podcílů do hloubky
- Výběr podcílů v klauzulích zleva doprava
- Operátor řezu pro další řízení výpočtu

# Struktura jazyka Prolog

## ■ Termy

- konstanty, proměnné, struktury
- $3, X, \_ , f(t_1, \dots, t_n)$

## ■ Klauzule

- $H \text{ :- } B_1, B_2, \dots, B_n.$

## ■ Cíle

- $? \text{ - } B_1, B_2, \dots, B_n.$



# Příklad

- `male(john). male(paul). male(bob).`
- `female(mary). female(jane). female(linda).`
- `father(john,bob). father(john,jane).`
- `mother(mary,bob). mother(linda,jane).`
- `parent(X,Y) :- father(X,Y).`
- `parent(X,Y) :- mother(X,Y).`
- `is_mother(X) :- mother(X,_).`
- `aunt(X,Y) :- female(X), sibling(X,Z), parent(Z,Y).`
- *`sister_of(X, Y), sibling(X, Y), grandpa_of(X, Y).`*



# Vestavěné predikáty

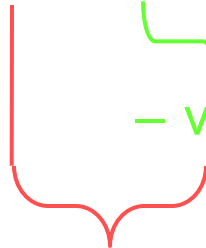
- mimologický, operačně definovaný význam, případně zcela bez deklarativního významu – pouze vedlejší účinky (vstup/výstup, modifikace databáze, ...)
- obvykle nejsou znovusplnitelné
- nezbytné pro praktické programování

# Aritmetické operace

– X is  $1+2$

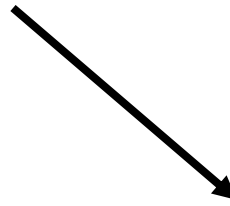


– vyhodnocení



– unifikace

–



termy se sami nevyhodnocují

– X is  $X+1$  *vždy selže*



# Testování typu dat

- `integer(X)`
  - X je číslo
- `atom(X)`
  - X je identifikátor (konstanta)
- `atomic(X)`
  - X je atom nebo číslo



# Metalogické predikáty

- $\text{var}(X)$ 
  - $X$  je nenavázaná proměnná
- $\text{nonvar}(X)$ 
  - $X$  je navázána na hodnotu
- $X == Y$ 
  - totožnost objektů



# Operace s klauzulemi

- `assert(C)`  
`asserta(C)`  
`assertz(C)`
  - vložení klauzule či faktu
- `clause(H,B)`
  - výběr
- `retract(C)`
  - zrušení



# Změna databáze

- Okamžitá (*immediate update view*)
- Logická (*logical update view*)
  - generace DB (s každou operací)
  - cíl uzavřený v intervalu ‚vložení‘...,zrušení‘  
klauzule s ní může pracovat

# *Immediate update view*

- | ?- assert(a(1)).  
yes
- | ?- a(X), Y is X + 1, assert(a(Y)).  
X = 1,  
Y = 2 ? ;  
X = 2,  
Y = 3 ? ;  
...



# *Logical update view*

- | ?- assert(a(1)).  
yes
- | ?- a(X), Y is X + 1, assert(a(Y)).  
X = 1,  
Y = 2 ? ;  
no
- | ?- a(X), Y is X + 1, assert(a(Y)).  
X = 1,  
Y = 2 ? ;  
X = 2,  
Y = 3 ? ;  
no

# Rozebírání struktury termů

- $T = ..L$ 
  - převod termu na seznam
- $call(P)$ 
  - zavolání predikátu  $P$  (jako by byl v místě zapsán)
- $functor(T, F, C)$ 
  - funktor + četnost
- $arg(N, T, A)$ 
  - $N$ -tý argument termu  $T$



# Vstup a výstup

- read
  - načti ze vstupu
- write
  - zapiš na výstup
- see/seen/seeing
- tell/told/telling



# Pomocné predikáty

- $A=B$ 
  - explicitní unifikace
- fail
  - vždy selhávající predikát
- true
  - vždy uspívající predikát
- repeat
  - vždy znovu-splnitelný predikát
- not
  - negace jako neúspěch

# Unifikace v Prologu

- základní a jediná operace
- probíhá při volání procedur zadáním cíle, nebo explicitně v predikátu `,=‘`
- neprovádí se kontrola výskytu
  - $X = f(X)$ 
    - naváže  $X$  na  $f(X)$ 
      - » **krach**
      - » **nesprávný výsledek**

# Využití unifikace

- Přiřazení hodnoty proměnné
  - $X = T$
- Test na rovnost
  - $\text{Term1} = \text{Term2}$
- Selektor ze seznamů
  - $L = [X|_ ]$
- Vytváření struktur
  - $Y_s = [a,b,c]$

# Využití unifikace (pokr.)

- Předávání parametrů hodnotou
  - term jako argument
- Předávání parametrů odkazem
  - proměnná jako argument
- Sdílení proměnných, vytváření synonym
  - $a(P, Q)$   
    ↓     ↓  
     $a(X, X)$

# Důležité pojmy

- Predikátová logika
  - Predikát, důkaz, dedukční pravidla
- Hornova klauzule, term
- SLD rezoluce
  - Prohledávání do hloubky
- Unifikace





# Úkoly



- Ve vašem oblíbeném časopise najděte nějakou zábavní logickou hříčku, řešte ji pomocí jazyka Prolog



# Seznamy, operátor řezu, řazení

# Cíle oddílu



- Typy seznamů v logických jazycích a práce s nimi
- Funkce a využití operátoru řezu
- Zvládnutí práce a programovacích technik na praktických příkladech

# Prologovské seznamy

## ■ Konstruktory: ‘.’ nil

- kind list      type  $\rightarrow$  type.  
type []      (list A).  
type ‘.’       $A \rightarrow (\text{list } A) \rightarrow (\text{list } A)$

- $.(1,.(2,[]))$        $\sim$       [1,2]
- $.(1,.(2,X))$        $\sim$       [1,2|X]

# Příklad

## ■ member(X,L)

- member(X, [X|\_]).  
member(X, [Y|Tail]) :- member(X,Tail).

## ■ append(L1,L2,L12)

- append([], L, L).  
append([H|T1], L2, [H|T2]) :- append(T1,L2,T2).

## ■ last(X,L)

- last(X, [X]).  
last(X, [\_|Tail]) :- last(X, Tail).



# Náměty k zamyšlení

- `permutation(L1, L2)`
  - L2 je permutací L1
- `reverse(L1, L2)`
  - L2 je L1 v opačném pořadí
- `sublists(XS, XSS)`
  - XSS je seznam všech podseznamů XS

# Predikáty „vyššího řádu“ 1

- `map(_, [], []).`  
`map(F, [H|T], [NH|NT]) :-`  
    `P =.. [F,H,NH], call(P), map(F, T, NT).`
- `inc(X,Y) :- var(Y), Y is X+1, !.`  
    `inc(X,Y) :- Z is Y-1, Z=X.`
- `?- map(inc,[1,2,3],X).`       $\sim>$       `X=[2,3,4]`
- `?- map(inc,X,[2,3,4]).`       $\sim>$       `X=[1,2,3]`

# Predikáty „vyššího řádu“ 2

- `foldr(_, B, [], B).`  
`foldr(F, B, [H|T], BB) :-`  
    `foldr(F, B, T, BT),`  
    `P =.. [F,H,BT,BB], call(P).`
- `foldl(_, A, [], A).`  
`foldl(F, A, [H|T], AA) :-`  
    `P =.. [F,A,H,AT], call(P),`  
    `foldl(F, AT, T, AA).`



# Příklad na „folds“

- `add(X,Y,Z) :- ZZ is Y+X, ZZ=Z.`
- `conS(T,H,[H|T]).`
- `sum(L, S) :- foldr(add, 0, L, S).`
- `rev(L, RL) :- foldl(conS, [], L, RL).`

# Diferenční seznamy

## ■ Idea

- $X - X \sim []$   
   $[1,2,3|X] - X \sim [1,2,3]$

## ■ Operace jsou často destruktivní

- nefungují „obousměrně“
- `dappend(A-ZA, ZA-ZB, A-ZB)`

# Příklad

- $\text{dtwice}(L, LL) \text{ :- dappend}(L, L, LL).$
- $\text{dtwice}([1|X]-X, Y)$   
     $\leadsto \text{dappend}([1|X]-X, \underbrace{[1|X]-X}_{\text{unifikace}}, Y)$   
     $\leadsto Y = [1|X] - X$   
         $X = [1|X]$

? nekonečný term  
? kontrola výskytu

# Další příklady

- `rev1([], Y-Y).`  
  `rev1([A|L], Z-Y) :- rev1(L, Z-[A|Y]).`
- `rev(L, RL) :- rev1(L, RL-[]).`
- `dlist2list(L-[], L).`
- `list2dlist(L, AL-ZL) :- dappend(L, ZL, AL).`

# Funkcionální seznamy

## ■ Idea

- $z \setminus z \sim []$   
   $z \setminus [1,2,3 \mid z] \sim [1,2,3]$

## ■ Není destruktivní

- lze použít ve všech směrech
- `fappend(L, R, z \ (L (R z)))`.

# Příklad

- `ftwice(L, LL) :- fappend(L, L, LL).`
- `fmember(E, z\(_ [E|(_z)]))`.
- `fnrev(z\z, z\z).`  
`fnrev(z\[A|(L z)], z\ (RL [A|z])) :- fnrev(L, RL).`
- `frev1([], z\z).`  
`frev1([A|L], z\ (RL [A|z])) :- frev1(L, RL).`  
`frev(L, (RL [])) :- frev1(L, RL).`
- `flist2list(F, (F []))`.
- `list2flist(L, FL) :- pi list\ (append(L, list, (FL list))).`

# Operátor řezu

- **foo :- a, b, c, !, d, e, f.**  
– **navracení**      **navracení**
- po splnění **c** může probíhat navracení pouze mezi **d, e, f**
- pokud selže **d**, selže celý cíl **foo**  
(body návratu až po **foo** včetně se zapomenou)

# Použití operátoru řezu 1

- Chceme sdělit Prologu, že již byla nalezena správná varianta pravidla

- ```
sum_to(1,1) :- !.  
  sum_to(N, Res) :-  
    N1 is N-1,  
    sum_to(N1,Res1),  
    Res is Res1+N.
```



# Použití operátoru řezu 2

- Chceme, aby cíl ihned selhal
  - `!, fail`
  - `not(P) :- call(P), !, fail.`  
`not(P).`

# Použití operátoru řezu 3

## ■ Chceme ukončit generování alternativních řešení

- `is_integer(0).`  
`is_integer(X) :- is_integer(Y), X is Y+1.`
- `divide(N1, N2, Result) :-`  
    `is_integer(Result),`  
    `Product1 is Result*N2,`  
    `Product2 is (Result+1)*N2,`  
    `Product1 =< N1, Product2 > N1, !.`



# Pozor!

- Zavedením řezu lze zamezit získání všech správných řešení nebo obdržet řešení nesprávná
- Pokud začneme pravidla používat jiným způsobem, mohou dávat nesprávné výsledky



# Příklad

- `num_of_parents(adam, 0) :- !.`  
  `num_of_parents(eve, 0) :- !.`  
  `num_of_parents(X, 2).`
- `?- num_of_parents(eve, 2).`  
  `yes`

# Příklady s operátorem řezu 1

- `remove(A, [A|L], L) :- !.`  
`remove(A, [B|L], [B|M]) :- remove(A,L,M).`  
`remove(_, [], []).`
- `delete(_, [], []).`  
`delete(X, [X|L], M) :- !, delete(X, L, M).`  
`delete(X, [Y|L1], [Y|L2]) :- delete(X, L1, L2).`

# Příklady s operátorem řezu 2

- `intersection([], X, []).`  
`intersection([X|R], Y, [X|Z]) :-`  
    `member(X,Y),`  
    `!,`  
    `intersection(R,Y, Z).`  
`intersection([X|R], Y, Z) :- intersection(R, Y, Z).`
- `union([], X, X).`  
`union([X|R], Y, Z) :-`  
    `member(X, Y), !, union(R, Y, Z).`  
`union([X|R], Y, [X|Z]) :- union(R, Y, Z).`

# Predikáty „vyššího řádu“ s operátorem řezu 1

- `range(S, S, [S]) :- !.`  
`range(S, E, [S|T]) :-`  
    `S < E, SS is S+1, range(SS, E, T), !.`  
    `range(_, _, []).`
- `?- range(1, 10, L).`  
    `~> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

# Predikáty „vyššího řádu“ s operátorem řezu 2a

- `range(S, _, S, [S]) :- !.`  
`range(S, N, E, [S|T]) :-`  
    `ST is N-S,`  
    `(ST < 0, S >= E, rangeD(N, ST, E, T);`  
    `ST >= 0, S <= E, rangeU(N, ST, E, T)), !.`  
`range(_, _, _, []).`
- `rangeD(S, ST, E, [S|T]) :-`  
    `S >= E, SS is S+ST,`  
    `rangeD(SS, ST, E, T), !.`  
`rangeD(_, _, _, []).`



# Predikáty „vyššího řádu“ s operátorem řezu 2b

- $\text{rangeU}(S, ST, E, [S|T]) :-$   
     $S \leq E$ ,  $SS$  is  $S+ST$ ,  
     $\text{rangeU}(SS, ST, E, T)$ , !.  
     $\text{rangeU}(\_, \_, \_, [])$ .
- $?- \text{range}(1, 3, 10, L).$   
     $\sim \rightarrow [1, 3, 5, 7, 9]$
- $?- \text{range}(10, 9, 5, L).$   
     $\sim \rightarrow [10, 9, 8, 7, 6, 5]$

# Predikáty „vyššího řádu“ s operátorem řezu 3

- `filter(_, [], []).`  
`filter(P, [H|T], [H|TT]) :-`  
    `PP =.. [P,H], call(PP), !, filter(P, T, TT).`  
`filter(P, [_|T], TT) :-`  
    `filter(P, T, TT).`
- `odd(X) :- Y is X // 2, YY is Y * 2, YY \= X.`
- `?- filter(odd, [1, 2, 3, 4, 5, 6], L).`  
    `~> [1, 3, 5]`

# Predikáty „vyššího řádu“ s operátorem řezu 4

- `take(_, [], []).`  
`take(N, [H|T], [H|TT]) :-`  
    `N > 0, !, NN is N-1, take(NN, T, TT).`  
`take(_, _, []).`
- `?- take(5, [1, 2, 3, 4, 5, 6, 7, 8], X).`  
    `~> X=[1, 2, 3, 4, 5]`

# Predikáty „vyššího řádu“ s operátorem řezu 5

- `takeWhile(_, [], []).`  
`takeWhile(P, [H|T], [H|TT]) :-`  
    `PP =.. [P,H], call(PP), !, takeWhile(P, T, TT).`  
`takeWhile(_, _, []).`
- `dropWhile(_, [], []).`  
`dropWhile(P, [H|T], TT) :-`  
    `PP =.. [P,H], call(PP), !, dropWhile(P, T, TT).`  
`dropWhile(_, L, L).`

# Predikáty „vyššího řádu“ s operátorem řezu 6

- `split(_, [], ([],[])) :- !.`  
`split(P, L, R) :- split(P, L, [], R).`
- `split(_, [], W, (RW,[])) :- reverse(W, RW).`  
`split(P, [H|T], W, R) :-`  
    `PP =.. [P,H], call(PP), !, split(P, T, [H|W], R).`  
`split(_, R, L, (RL,R)) :- reverse(L, RL).`
- `?- split(mensi_jak_3, [1,2,3,4,5], L).`  
    `~> X = [1, 2] , [3, 4, 5]`

# Řazení - generuj a testuj

- `gtsort(L1, L2) :-`  
    `permutation(L1,L2), sorted(L2), !.`
- `sorted([]).`  
    `sorted([_]).`  
    `sorted([A,B|L]) :- A =< B, sorted([B|L]).`
- `permutation([], []).`  
    `permutation(L, [H|T]) :-`  
        `append(V, [H|U], L),`  
        `append(V, U, W),`  
        `permutation(W, T).`

# Řazení na principu vkládání 1

- `insert(L1, L2) :- insert(L1, [], L2).`
- `insert([], X, X).`  
`insert([X|X1], Y, Z) :-`  
    `ins(X, Y, Z1), insert(X1, Z1, Z).`
- `ins(X, [], [X]).`  
`ins(X, [Y|Y1], [X,Y|Y1]) :- X <= Y, !.`  
`ins(X, [Y|Y1], [Y|Z1]) :- ins(X, Y1, Z1).`

# Řazení na principu vkládání 2

- `insert(L1, L2, ○) :- insert(L1, [], L2, ○).`
- `insert([], X, X, ○).`  
`insert([X|X1], Y, Z, ○) :-`  
    `ins(X, Y, Z1, ○), insert(X1, Z1, Z, ○).`
- `ins(X, [], [X], _).`  
`ins(X, [Y|Y1], [X,Y|Y1], ○) :-`  
    `P =.. [○, X, Y], call(P), !.`  
`ins(X, [Y|Y1], [Y|Z1], ○) :- ins(X, Y1, Z1, ○).`



# Řazení na principu „bubble sort”

- `bubble(L1, L2) :-`  
    `append(X, [A,B|Y], L1), A > B,`  
    `append(X, [B,A|Y], Z), bubble(Z,L2).`  
    `bubble(L,L).`
- nelze použít optimalizované verze `append`  
    `append([], X, X) :- !.`  
    `append([H|X], Y, [H|Z]) :- append(X, Y, Z).`

# Řazení metodou „quicksort“

- `quick([], []).`  
`quick([H|T], S) :-`  
    `split(T, H, A, B),`  
    `quick(A, A1), quick(B, B1),`  
    `append(A1,[H|B1],S).`
- `split([], _, [], []).`  
`split([X|X1], Y, [X|Z1], Z2) :-`  
    `X < Y, split(X1, Y, Z1, Z2).`  
`split([X|X1], Y, Z1, [X|Z2]) :-`  
    `X >= Y, split(X1, Y, Z1, Z2).`

*možno  
zpracovat  
paralelně*

# Důležité pojmy



- Seznamy
  - Prologovské, diferenční, funkcionální
- Řez
- Operátor řezu
  - Případy použití

# Úkoly



- Implementujte i další řadicí algoritmy nad seznamy
- Vložte údaje z DB systému řešeného ve funkcionální části do Prologu a implementujte jednoduché dotazování – využijte seznamy, rekurze a operátor řezu



# Praktiky v Prologu 1

# Cíle oddílu

- Data a datové struktury
- Řetězce (SWI Prolog)
- Formátované výstupní operace
- Práce s operátory



# Nejjednodušší data

## ■ Pojmenované n-tice

- často uložené jako fakty
- i přímé použití ~ pattern matching
- `isRGB(color(X)) :-  
    member(X,[red,green,blue]).`
- `?- isRGB(color(brown)).`                      ~> No.
- `?- isRGB(color(red)).`                        ~> Yes.



# Rekurzivně definovaná data

## ■ Význam typů malý

- Kontrola se provádí jen v určitých případech

## ■ Druhy

- libovolné hierarchické struktury
- tzn. i rekurzivní





# Stromy 1

## ■ Vyhledávací stromy

– klíč

- typicky číslo
- pro ‚nečísla‘ nutno nadefinovat vlastní porovnávací predikát

– data

- cokoliv

# Příklad

- `emptyTree(leaf).`
- `add2tree(K:V, leaf, node(K:V,leaf,leaf)).`  
`add2tree(K:_, node(K:V,X,Y), node(K:V,X,Y)) :- !.`  
`add2tree(K:V, node(K:_,X,Y), node(K:V,X,Y)) :- !.`  
`add2tree(Kn:Vn, node(K:V,L,R), node(K:V,LL,R)) :-`  
    `Kn < K, !,`  
    `add2tree(Kn:Vn, L, LL).`  
`add2tree(Kn:Vn, node(K:V,L,R), node(K:V,L,RR)) :-`  
    `add2tree(Kn:Vn, R, RR).`

# Příklad (pokr.)

- `inOrdTree(leaf, []).`  
`inOrdTree(node(Key:Value,L,R), List) :-`  
    `inOrdTree(L, LL),`  
    `inOrdTree(R, RL),`  
    `append(LL, [Key:Value|RL], List).`
- `list2tree([], leaf).`  
`list2tree([I|IS], NewTree) :-`  
    `list2tree(IS, Tree),`  
    `add2tree(I:I, Tree, NewTree).`



# Data jako DB

- Ukládání a definice (výše)
- Výpis dat
  - write(+Term)
  - write\_ln(+Term)
  - nl



# Příklad

- `author('John','Smith',1956,living).`  
`author('Peter Paul','Willis',1895,1940).`
- `book('Happy Luke',1980,none).`  
`book('Sad Jane',1983,45-234234-5).`  
`book('Lonely Lady',1939,none).`
- `b_a(1,1).`  
`b_a(1,2).`  
`b_a(2,3).`

# Příklad (pokr.)

- `display(author(Name,Surname,Born,living)) :-`  
    `write('Author: '),`  
    `write(Name), write(' '), write_In(Surname),`  
    `write('Born on: '),`  
    `write_In(Born), !.`  
`display(author(Name,Surname,Born,Died)) :-`  
    `write('Author: '),`  
    `write(Name), write(' '), write_In(Surname),`  
    `write('Born on: '), write_In(Born),`  
    `write('Died on: '), write_In(Died).`

# Příklad (pokr.)

- `display(book(Title,Year,ISBN)) :-`  
    `write('Book title: '),`  
    `write(Title), nl,`  
    `write('Published in year: '),`  
    `write(Year), nl,`  
    `write('ISBN: '),`  
    `write(ISBN), nl.`

# Příklad (pokr.)

- `displayAllAuthors :-`  
    `author(N,S,B,D),`  
    `display(author(N,S,B,D)),`  
    `fail.`  
    `displayAllAuthors.`
- `displayAllBooks :-`  
    `book(T,Y,I),`  
    `display(book(T,Y,I)),`  
    `fail.`  
    `displayAllBooks.`



# Formátovaný výstup

- Podobné jazyku C
- Ne všechny mutace Prologu
  - `writeln(+Format,+[Term])`
    - `%w`      prostý výpis
    - `%r`      term, kolikrát zopakovat
    - `%Nl/r/c`    zarovnat na N pozic (! **atomy** !)
    - řada dalších (viz WWW, ref. man., ...)
  - `?- writeln('%10l%w', ['Ahoj', 'Karle']).`  
~> Ahoj      Karle

# Řetězce

## ■ Literál

- převedeno na seznam znaků

## ■ Typ ‚string‘ (**není seznam, SWI**)

- string\_to\_atom(?String, ?Atom)
- string\_to\_list(?String, ?List)
- string\_concat(?String1, ?String2, ?String3)
- sub\_string(+String, ?Start, ?Length, ?After, ?Sub)
- string\_length(+String, -Length)
- swritef(-String, +Format, +[Term])

# Příklad - DB

- `writeL(SP, ToWrite) :-`  
    `swritef(S, '%w ', [ToWrite]),`  
    `string_length(S, L),`  
    `SS is SP-L,`  
    (  
        `SS>0,`  
        `writeln('%w%', [ToWrite, ' ', SS])`  
    ;  
    `write(ToWrite)`  
), !.

# Příklad - DB (pokr.)

- `displayf(author(Name,Surname,Born,living),J) :-`  
    `writeL(J,'Author:'),`  
    `write(Name), write(' '), write_In(Surname),`  
    `writeL(J,'Born on:'), write_In(Born), !.`
- `displayf(author(Name,Surname,Born,Died),J) :-`  
    `writeL(J,'Author:'),`  
    `write(Name), write(' '), write_In(Surname),`  
    `writeL(J,'Born on:'), write_In(Born),`  
    `writeL(J,'Died on:'), write_In(Died).`

# Příklad - DB (pokr.)

- `displayf(book(Title,Year,ISBN),J) :-`  
    `writeL(J,'Book title:'),`  
    `write_In(Title),`  
    `writeL(J,'Published in year:'),`  
    `write_In(Year),`  
    `writeL(J,'ISBN:'),`  
    `write_In(ISBN).`
- `?- displayf(book('R.U.R',1935,none), 19).`

# Přístup ke klauzulím

## ■ Predikáty 'clause'

- `clause(?Head, ?Body)`
- `clause(?Head, ?Body, ?Reference)`
- `nth_clause(?Pred, ?Index, ?Reference)`
- `?- nth_clause(member(_, _), 2, Ref),  
                    clause(Head, Body, Ref).`  
~ Ref = 160088  
Head = system : member(G575, [G578|G579])  
Body = member(G575, G579)

# Příklad DB

- displayAll :-

```
writeln('%r\n', ['=', 18]),  
b_a(Author, Book),  
nth_clause(author(_,_,_,_), Author, RA),  
clause(A,_,RA),  
nth_clause(book(_,_,_), Book, RB),  
clause(B,_,RB),  
displayf(B,19), displayf(A,19),  
writeln('%r\n', ['- ', 18]), fail.
```

- displayAll :-

```
nl, write_ln('END...').
```

# Operátory

## ■ Definice nového operátoru

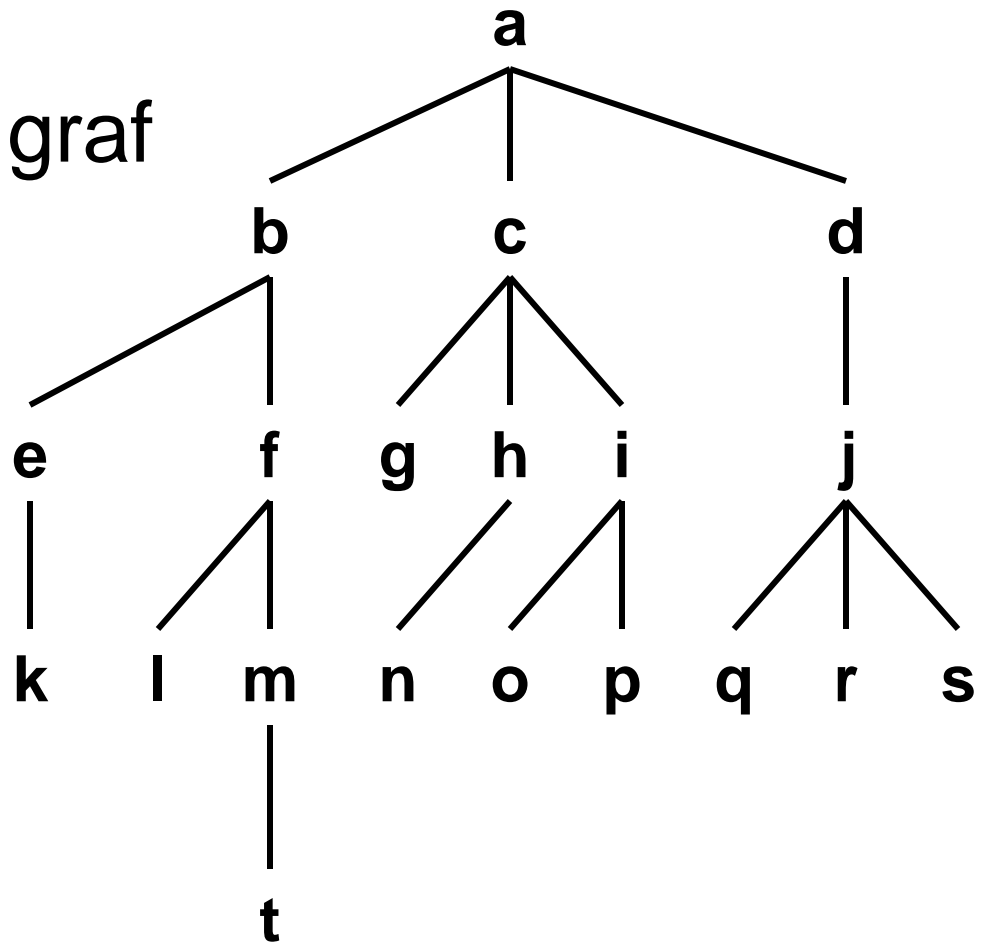
- `op(+Precedence, +Type, :Name)`
  - 0-1200, 0 ruší deklaraci, 1200 nejnižší
  - `xf`, `yf`, `xfx`, `xfy`, `yfx`, `yfy`, `fy`, `fx`
  - může být i seznam



1200	xfx	--> :-
1200	fx	:- ?-
1150	fx	dynamic multifile module_transparent discon-tiguous volatile initialization
1100	xfy	;
1050	xfy	->
1000	xfy	,
954	xfy	\
900	fy	\+
900	fx	~
700	xfx	< = =.. =@= =:= =< == =\= > >= @< @=< @> @>= \= \== is
600	xfy	:
500	yfx	+ - ^ V xor
500	fx	+ - ? \
400	yfx	* / // << >> mod rem
200	xfx	**
200	xfy	^

# Příklad

- Strom jako graf  
*case study*



# Definice grafu

- :- op(500, xfx, 'is\_parent').
- root(a).
- a is\_parent b.      a is\_parent c.      a is\_parent d.  
b is\_parent e.      b is\_parent f.  
c is\_parent g.      c is\_parent h.      c is\_parent i.  
d is\_parent j.  
e is\_parent k.  
f is\_parent l.      f is\_parent m.  
h is\_parent n.  
i is\_parent o.      i is\_parent p.  
j is\_parent q.      j is\_parent r.      j is\_parent s.  
m is\_parent t.

# Uzly se stejným rodičovským uzlem

- :- op(500, xfx, 'is\_sibling\_of').
- X is\_sibling\_of Y :-  
    Z is\_parent X,  
    Z is\_parent Y,  
    X \== Y.
- ?- X is\_sibling m.  
    ~> X = I  
    Yes.

# Uzly na stejné úrovni

- :- op(500, xfx, 'is\_same\_level\_as').
- X is\_same\_level\_as X .  
X is\_same\_level\_as Y :-  
    W is\_parent X,  
    Z is\_parent Y,  
    W is\_same\_level\_as Z.
- ?- g is\_same\_level p.  
    ~> No.



# Hloubka uzlu ve stromu

- `:- op(500, xfx, 'has_depth').`

- `Node has_depth 0 :-  
    root(Node), !.`

`Node has_depth D :-  
    Mother is_parent Node,  
    Mother has_depth D1,  
    D is D1 + 1.`

- `?- k has_depth D.  
    ~> D = 3`

# Cesta od kořene k uzlu

- `locate(Node) :-`  
    `path(Node),`  
    `write(Node),`  
    `nl.`
- `path(Node) :-`  
    `root(Node).`  
    `path(Node) :-`  
        `Mother is_parent Node,`  
        `path(Mother),`  
        `write(Mother),`  
        `write(' --> ').`

# Příklad

- ?- locate(p).  
~> a --> c --> i --> p

Yes

- ?- locate(a).  
~> a

Yes



# Zjištění variant podcíle

## ■ Posloupnost dle databáze

- bagof(+Var, +Goal, -Bag)
  - proměnná, která má být testována
  - podcíl obsahující testovanou proměnnou
  - seznam možných hodnot
- setof(+Var, +Goal, -Set)
  - výsledkem je uspořádaná množina (seznam) termů-jedinečný výskyt opakovaných hodnot

# Příklad

- ?- listing(foo).

foo(a, b, c).

foo(a, b, d).

foo(b, c, e).

foo(b, c, f).

foo(c, c, g).

- ?- bagof(C, foo(A, B, C), Cs).

A = a, B = b, C = G308, Cs = [c, d] ;

A = b, B = c, C = G308, Cs = [e, f] ;

A = c, B = c, C = G308, Cs = [g] ;

- ?- bagof(C, A^foo(A, B, C), Cs).

A = G324, B = b, C = G326, Cs = [c, d] ;

A = G324, B = c, C = G326, Cs = [e, f, g] ;

# Funkční if-then-else

## ■ Zápis

- (podmínka  $\rightarrow$  splněná ; nesplněná)
- $\text{max}(X, Y, M) :-$   
     $(X > Y \rightarrow M = X ; M = Y).$

# Výška uzlu (nejdelší cesta k listu)

- $\text{height}(N, H) :-$   
     $\text{setof}(Z, \text{ht}(N, Z), \text{Set}),$   
     $\text{max}(\text{Set}, 0, H).$
- $\text{ht}(\text{Node}, 0) :-$   
     $\text{leaf}(\text{Node}), !.$   
     $\text{ht}(\text{Node}, H) :-$   
         $\text{Node is\_parent Child},$   
         $\text{ht}(\text{Child}, H1),$   
         $H \text{ is } H1 + 1.$



# Výška uzlu (nejdelší cesta k listu - pokr.)

- `leaf(Node) :-  
    not(is_parent(Node, _)).`
- `max([], M, M).  
max([X|R], M, A) :-  
    (X > M -> max(R, X, A) ; max(R, M, A)).`



# Příklad

- ?- height(f, H).  
~> H = 2
- ?- height(g, H).  
~> H = 0
- ?- height(N, 4).  
~> N = a

# Důležité pojmy

- (Pojmenovaná) n-tice
- Řetězec
- DB Prologu
- Operátor



# Úkoly



- Navrhnete systém operátorů pro uložení mapy do DB Prologu – využijte zejména zmíněné techniky a predikáty.
- Implementujte jednoduché dotazování nad mapami – využijte predikátů pro práci a vyhodnocení DB





# Praktiky v Prologu 2

# Cíle oddílu

- Změna DB
- Klíč s hodnotou
- Dynamické klauzule
- Vlastní funkce
- Moduly





# Operace s databází

## ■ Změna databáze

- změna programu
  - náročné výpočty - simulace, ...
  - Vojnar, Janoušek
- uložení faktů o stavu výpočtu
  - nalezené ‚znalosti‘
  - pozice ve stavovém prostoru
  - ...



# Standardní operace

- assert(+Term)  
asserta(+Term)  
assertz(+Term)
- retract(+Term)  
retractall(+Head)
- *Logical/Immediate View !*



# Databáze s klíčem

- recorda(+Key, +Term, -Reference)  
recorda(+Key, +Term)
- recordz(+Key, +Term, -Reference)  
recordz(+Key, +Term)
- recorded(+Key, -Value, -Reference)  
recorded(+Key, -Value)
- erase(+Reference)

# Klíč s hodnotou

- `flag(+Key, -Old, +New)`
- `countTo(Times,Goal) :-`  
    (  
        `flag(cx,_,0),`  
        `Goal,`  
        `flag(cx,N,N+1),`  
        `fail`  
    ;  
        `flag(cx,Times,0)`  
    ).

# Dynamické klauzule

- Pokud měníme DB, je vhodné označit odstraňované/přidávané klauzule.
  - při dotazu na neexistující klauzuli je signalizován prostý neúspěch
  - :- dynamic  
pos/2,  
tmp/2.

# Prohledávání s. prostoru 1

- `searchPathL(start(X,Y), end(X,Y), [X:Y]) :- !.`  
`searchPathL(start(X,Y), E, [X:Y|T]) :-`  
    `assert(pos(X,Y)),`  
    `nextStep(X, Y, XX, YY),`  
    `not( pos(XX,YY) ),`  
    `searchPathL(start(XX,YY), E, T).`  
`searchPathL(start(X,Y), _, _) :-`  
    `pos(X, Y),`  
    `retract(pos(X,Y)),`  
    `fail.`



# Určení dalšího kroku

- `nextStep(X, Y, XX, YY) :-`  
    `XX is X+1, YY is Y+1, test(XX, YY).`  
`nextStep(X, Y, XX, YY) :-`  
    `XX is X+1, YY is Y-1, test(XX, YY).`  
`nextStep(X, Y, XX, YY) :-`  
    `XX is X-1, YY is Y+1, test(XX, YY).`  
`nextStep(X, Y, XX, YY) :-`  
    `XX is X-1, YY is Y-1, test(XX, YY).`
- `test(X, Y) :- X>0, Y>0, X<10, Y<10.`

# Vyčištění stavového prostoru

- clearPos :-  
    pos(X, Y),  
    retract( pos(X,Y) ),  
    clearPos, !.  
clearPos.

# Prohledávání s. prostoru 2

- `searchPath(start(X,Y), end(X,Y)) :-  
    assert( pos(X,Y) ).  
searchPath(start(X,Y), end(X,Y)) :-  
    pos(X, Y),  
    retract( pos(X,Y) ), !,  
    fail.`



...

- `searchPath(start(X,Y), E) :-`  
    `assert( pos(X,Y) ),`  
    `nextStep(X, Y, XX, YY),`  
    `not( pos(XX,YY) ),`  
    `searchPath(start(XX,YY), E).`  
`searchPath(start(X,Y), _) :-`  
    `pos(X, Y),`  
    `retract( pos(X,Y) ),`  
    `fail.`

# Vypsání nalezené cesty

- writePath :-  
    not( pos( \_, \_ ) ),  
    write\_In('None'), !.  
writePath :-  
    pos(X, Y),  
    write('Moved to: '),  
    write(X), write(','), write\_In(Y),  
    fail.  
writePath.

# Uložení cesty do seznamu 1

```
■ listPos([]) :-  
    not( pos(_,_) ), !.  
listPos([X:Y|T]) :-  
    pos(X, Y),  
    not( tmp(X,Y) ),  
    assert( tmp(X,Y) ),  
    listPos(T), !.  
listPos([]) :-  
    clearTmp.                                /* analogie s clearPos */
```

# Uložení cesty do seznamu 2

- `xlistPos(L) :-  
listPos(1, L).`
- `listPos(N, [X:Y|T]) :-  
nth_clause(pos(_, _), N, R),  
clause(pos(X, Y), _, R),  
NN is N+1,  
listPos(NN, T), !.  
listPos(_, []). :- !.`

# Uložení cesty do seznamu 3

- `slistPos(L) :-`  
    `bagof(X, Ye^pos(X,Ye), Xs),`  
    `bagof(Y, Xe^pos(Xe,Y), Ys),`  
    `zip(Xs, Ys, L).`
- `zip([], _, []) :- !.`  
    `zip(_, [], []).`  
    `zip([X|XS], [Y|YS], [X:Y|XYS]) :-`  
        `zip(XS, YS, YYS).`



# Prohledávání s. prostoru 3

- `searchPathR(start(X,Y), end(X,Y)) :-  
    recordz(pos, X:Y).  
searchPathR(start(X,Y), end(X,Y)) :-  
    recorded(pos, X:Y, Ref),  
    erase(Ref), !,  
    fail.`



...

- `searchPathR(start(X,Y), E) :-`  
    `recordz(pos, X:Y),`  
    `nextStep(X, Y, XX, YY),`  
    `not( recorded(pos, XX:YY) ),`  
    `searchPathR(start(XX,YY), E).`  
`searchPathR(start(X,Y), _) :-`  
    `recorded(pos, X:Y, Ref),`  
    `erase(Ref),`  
    `fail.`

# Uložení cesty do seznamu 4

- `listPosR(L) :-`  
    `bagof(V, recorded(pos,V), L).`



# Definice vlastních funkcí

## ■ Pouze aritmetické funkce

- vyhodnocení operátorem **is**

## ■ Zápis

- `:- arithmetic_function(pred/arity).`
  - `pred` označuje jméno predikátu
  - `arity` označuje počet parametrů (bez výsledku)

# Příklad

- `:- arithmetic_function(max2/2).`  
  `:- arithmetic_function(max3/3).`
- `max2(X, Y, M) :-`  
    `(X > Y -> X=M ; Y=M).`
- `max3(X, Y, Z, M) :-`  
    `MM is max2(X, Y),`  
    `M is max2(MM, Z).`
- `?- X is max2(4, 5).`  
  `~> X=5`



# Moduly

- Výhody modulárního programování jsou zřejmé
- Modulární přístup v Prologu
  - Vazba modulů přes jména
  - Vazba modulů přes predikáty

# Vazba modulů přes jména

- `:- module(extend, [add_extension/3]).`
- `add_extension(Extension, Plain, Extended) :-  
    maplist(extend_atom(Extension), Plain, Extended).`
- `extend_atom(Extension, Plain, Extended) :-  
    concat(Plain, Extension, Extended).`
  - vyhodnotí se korektně
  - *extend\_atom* je označen jménem modulu
  - toto ale udělá se všemi lokálními (nikoliv *public*) atomy

# Problém s vazbou přes jména

- `:- module(action, [action/3]).`  
    `action(Object, sleep, Arg) :- ....`  
    `action(Object, awake, Arg) :- ....`
- `:- module(process, [awake_process/2]).`  
    `awake_process(Process, Arg) :-`  
        `action(Process, awake, Arg).`
- vzhledem k lokálním datům se neprovede





# Vazba přes predikáty

- Všechna data jsou globální a viditelná mezi moduly
- Řeší uvedené problémy
- Problém s
  - meta-predikáty (pokročilí uživatelé)
  - nejsou lokální data

# Definice modulu

## ■ Soubor *rever.pl*

- `:- module(rever, [revs/2]).`

```
revs([],[]) :- !.
```

```
revs(L,RL) :-
```

```
    rev(L,[],RL).
```

```
rev([], L, L).
```

```
rev([X|XS], RS, L) :-
```

```
    rev(XS, [X|RS], L).
```

# Užití modulu

## ■ Soubor *main.pl*

- `:-use_module([rever]).`

```
palin(RL) :-  
    append(L, [S|R], RL),  
    (  
        revs(L, [S|R])  
        ;  
        revs(L, R)  
    ).
```

# Predikáty pro použití modulu

## ■ `use_module(+File)`

- nahraje modul (seznam modulů)

## ■ `use_module(+File, +ImportList)`

- nahraje modul a importuje dané predikáty (i ty, co nebyly exportovány - zjednodušuje ladění)

## ■ `import(+Head)`

- nahraje predikát *Head* do aktuálního modulu
  - *Head* ~ *<module>:<term>*



# Rezervovaná jména modulů

- system
  - předdefinované predikáty
- user
  - uživatelský pracovní prostor

# Příklad - OO implementace

- `:- module(oop, [invoke/4, create/3, oid/1]).`
- `:- dynamic(oid/1).`
- `oid(-1).`
- `:- op(500, xfx, 'is_instance_of').`
- `invoke(Instance, MethodName, Arguments, MID) :-`  
    `Instance is_instance_of Class,`  
    `Class:getMethod(MethodName, MethodPred, MID), !,`  
    `ToInvoke =.. [MethodPred,Instance | Arguments],`  
    `Class:ToInvoke.`



# Dokončení modulu *oop*

- `create(Class, MID, Instance) :-`  
    `Class:getMethod(create, MethodPred, MID), !,`  
    `ToInvoke =.. [MethodPred, Instance],`  
    `Class:ToInvoke.`
- `instance(Class,_) is_instance_of Class.`

# Třída *point*

- `:- module(point, []).`  
`:- dynamic(attr/4).`
- `mid('point_asd354422gfsdf').`
- `attribute( x, 0, private).`      `% private/read/public`  
`attribute( y, 0, private).`
- `method( create, create, public).`      `% public/private`  
`method( getX, getX, public).`  
`method( getY, getY, public).`  
`method( writeX, writeX, public).`  
`method( writeY, writeY, public).`



# Třída *point* - pokračování

- `getMethod(Name, Pred, MID) :-`  
    `mid(MID), !,`  
    `method(Name, Pred, _).`  
`getMethod(Name, Pred, _) :-`  
    `method(Name, Pred, public).`
- `readAttr(instance(point,OID), Name, Val, MID) :-`  
    `mid(MID), !,`  
    `attr(OID, Name, Val, _).`  
`readAttr(instance(point,OID), Name, Val, _) :-`  
    (`attr(OID, Name, Val, read)`  
    ;`attr(OID, Name, Val, public)`  
    ), !.

# Třída *point* - pokračování

- `writeAttr(instance(point,OID), Name, Val, MID) :-`  
    `mid(MID),`  
    `attr(OID, Name, V, Acc),`  
    `retract(attr(OID, Name, V, Acc)),`  
    `assert(attr(OID, Name, Val, Acc)), !.`  
`writeAttr(instance(point,OID), Name, Val, _) :-`  
    `attr(OID, Name, V, public),`  
    `retract(attr(OID, Name, V, public)),`  
    `assert(attr(OID, Name, Val, public)).`

# Třída *point* - pokračování

- `create(instance(point,OID)) :-`  
    `oid(Last),`  
    `OID is Last+1,`  
    `not( oid(OID) ),`  
    `asserta( oid(OID) ),`  
    `mkAttrs(OID).`
- `mkAttrs(OID) :-`  
    `attribute(N, V, A),`  
    `assert( attr(OID, N, V, A) ),`  
    `fail.`  
    `mkAttrs(_).`



# Třída *point* - pokračování

- `getX(I, X) :-`  
    `mid(MID),`  
    `readAttr(I, x, X, MID).`
- `getY(I, Y) :-`  
    `mid(MID),`  
    `readAttr(I, y, Y, MID).`



# Třída *point* - dokončení

- `writeX(I, X) :-`  
    `mid(MID),`  
    `writeAttr(I, x, X, MID).`
- `writeY(I, Y) :-`  
    `mid(MID),`  
    `writeAttr(I, y, Y, MID).`

# Užití obou modulů

- `:- use_module([oop, point]).`
- `mid('main_sdfj24kj345kj').`
- `demo :-`
  - `mid(MID),`
  - `create(point, MID, P1),`
  - `create(point, MID, P2),`
  - `pr('P1', P1),`
  - `pr('P2', P2),`
  - `invoke(P1, writeX, [5], MID),`
  - `invoke(P2, writeY, [12], MID),`
  - `pr('P1', P1),`
  - `pr('P2', P2).`

# Užití - tisk

- `pr(Id, P) :-`  
    `write_ln(Id),`  
    `mid(MID),`  
    `invoke(P, getX, [X], MID),`  
    `writeln('%w%5r\n', ['X:', X]),`  
    `invoke(P, getY, [Y], MID),`  
    `writeln('%w%5r\n', ['Y:', Y]).`

# Chybné užití

- demofail :-

```
mid(MID),  
create(point, MID, P1),  
create(point, MID, P2),  
pr('P1', P1),  
pr('P2', P2),  
point:writeAttr(P1, x, 5, MID),  
point:writeAttr(P2, y, 12, MID),  
pr('P1', P1),  
pr('P2', P2).
```



# Zjednodušení syntaxe

- `:- op(610, xfx, '>>').`
- `Instance >> Method : Args :-`  
    `mid(MID),`  
    `invoke(Instance, Method, Args, MID).`
- `demoX :-`  
    `mid(MID),`  
    `create(point, MID, P1),`  
    `create(point, MID, P2),`  
    `pr('P1', P1),`  
    `pr('P2', P2),`  
    `P1 >> writeX : [5],`  
    `P2 >> writeY : [12],`  
    `pr('P1', P1),`  
    `pr('P2', P2).`

# Důležité pojmy

- Klauzule
  - Dynamická x statická
- Matematické funkce
- Vazba mezi moduly



# Úkoly



- Rozšiřte systém s mapou tak, aby se daly dynamicky měnit informace uložené v DB
- Celou aplikaci vhodně modularizujte
  - Zkuste si nějaký modul, u něhož víte, že kolega má stejný, jej prohodit a odzkoušet funkčnost



# Praktiky v Prologu 3

# Cíle oddílu

- Pravidla pro zpracování gramatik





# Využití pravidel pro zpracování gramatik

## ■ *DCG Grammar rules*

- Definite Clause Grammar
- Založeno na rozdílových seznamech
- **Není** součástí standardního Prologu
- Velmi často je implementováno
  - s určitými drobnými rozdíly

# Zápis pravidel

## ■ Použití operátoru -->

- nepoužívá se :-
- pro převod pravidla do DB se používá predikát *expand\_term/2*
  - provádí překlad gramatických pravidel tak, že přidává další argumenty pro zápis rozdílových seznamů

# Vyhodnocení pravidel

- `phrase(+RuleSet, +InputList)`
- `phrase(+RuleSet, +InputList, -Rest)`
  - seznam pravidel - dán startovacím pravidlem/nonterminálem
  - seznam symbolů, který má být analyzován
  - seznam neanalyzovaných/odmítnutých symbolů



# Příklad

- integer(I) -->  
    digit(D0),  
    digits(D),  
    { number\_chars(I, [D0|D]) }.
- digits([D|T]) -->  
    digit(D), !,  
    digits(T).  
    digits([]) --> [].
- digit(D) -->  
    [D],  
    { code\_type(D, digit) }.



# Příklad - vyhodnocení

- ?- phrase(integer(X), "42 times", Rest).

$X = 42$

$\text{Rest} = [32, 116, 105, 109, 101, 115]$

# Case study - kalkulátor

## ■ Zadání:

- Vytvořte interpret jazyka umožňující definovat jednoduché aritmetické výrazy, pojmenovávat je, tisknout jejich hodnoty.
  - Klíčová slova: if, then, else, let, print
  - Operátory: &&, ||, :=, /=, ==, <=, >=, <, >, +, -, \*, /, %, :, (, ), ~

# Lexikální analýza - kategorie

- `isDig(X) :-`  
    `atom_char('0',Zero), atom_char('9',Nine),`  
    `X >= Zero, X =< Nine.`
- `isSpc(X) :- member(X,[9,10,13,32]).`
- `isChar(X) :-`  
    `atom_char('a',Lowa), atom_char('z',Lowz),`  
    `atom_char('A',Upa), atom_char('Z',Upz),`  
    `(X >= Lowa, X =< Lowz ;`  
    `X >= Upa, X =< Upz).`
- `isANum(X) :- isDig(X),!.`  
    `isANum(X) :- isChar(X).`
- `isOp(O) :- member(O,"/<>:+-*% ;=()&|~").`

# Lexikální analýza - operátory

- `keyops("&&", 'and').`  
`keyops(":= ", 'asg').`  
`keyops("/=", 'neq').`  
`keyops("<=", 'leq').`  
`keyops("<", 'lt').`  
`keyops("+", 'add').`  
`keyops("*", 'mul').`  
`keyops("%", 'mod').`  
`keyops("(", 'lp').`  
`keyops("~", 'not').`

`keyops("||", 'or').`

`keyops("==", 'eq').`  
`keyops(">=", 'geq').`  
`keyops(">", 'gt').`  
`keyops("-", 'sub').`  
`keyops("/", 'div').`  
`keyops(";", 'sem').`  
`keyops(")", 'rp').`

# Lexikální analýza - složení op.

- `mkops([X,Y|T],[O|TT]) :-  
    keyops([X,Y],O),!,  
    mkops(T,TT).  
mkops([X|T],[O|TT]) :-  
    keyops([X],O),  
    mkops(T,TT).  
mkops([],[]).`
- `mkop(X,op(X)).`



# Lexikální analýza - kl. slova

- `keyword("if", 'if').`  
`keyword("then", 'then').`  
`keyword("else", 'else').`  
`keyword("let", 'let').`  
`keyword("print", 'print').`

# Lexikální analýza - číslice

- `digit(D) -->`  
    `[D],`  
    `{ isDig(D) }.`
- `digits([D|T]) -->`  
    `digit(D), !,`  
    `digits(T).`  
`digits([]) -->`  
    `[].`



# Lexikální analýza - mezery

- `spaces([S|T]) -->`  
    `space(S), !,`  
    `spaces(T).`  
`spaces([]) -->`  
    `[].`
- `space(S) -->`  
    `[S],`  
    `{ isSpc(S) }.`

# Lexikální analýza - čísla+znaky

- `anums([AN|T]) -->`  
    `anum(AN), !,`  
    `anums(T).`  
`anums([]) -->`  
    `[].`
- `anum(AN) -->`  
    `[AN],`  
    `{ isANum(AN) }.`

# Lexikální analýza - operátory

- $\text{ops}([\text{AN}|\text{T}]) \rightarrow$   
     $\text{op}(\text{AN}), !,$   
     $\text{ops}(\text{T}).$   
 $\text{ops}([]) \rightarrow$   
     $[].$
- $\text{op}(\text{AN}) \rightarrow$   
     $[\text{AN}],$   
     $\{ \text{isOp}(\text{AN}) \}.$

# Lexikální analýza - znaky

- `charac(C) -->`  
    `[C],`  
    `{ isChar(C) }.`

# Lexikální analýza - celá čísla, klíčová slova

- `integer(I) -->`  
    `digit(D0),`  
    `digits(D),`  
    `{ number_chars(I, [D0|D]) }.`
- `keyword(K) -->`  
    `charac(C),`  
    `anums(CC),`  
    `{ keyword([C|CC], K) }.`

# Lexikální analýza - identifikátory, operátory

- `identifier(I) -->`  
    `charac(C),`  
    `anums(CC),`  
    `{ not(keyword([C|CC], _)), atom_chars(I,[C|CC]) }.`
- `operator(OP) -->`  
    `op(O),`  
    `ops(OO),`  
    `{ mkops([O|OO],OP) }.`

# Lexikální analýza - mezery, konce řádků, ...

- $ws(W) \rightarrow$   
     $space(S),$   
     $spaces(SS),$   
     $\{ W = [S|SS] \}.$

# Lexikální analýza - komplet 1

- `tokens(T) -->`  
    `integer(I), !,`  
    `tokens(TT),`  
    `{ T = [num(I)|TT] }.`
- `tokens(T) -->`  
    `keyword(K), !,`  
    `tokens(TT),`  
    `{ T = [K|TT] }.`
- `tokens(T) -->`  
    `identifier(I), !,`  
    `tokens(TT),`  
    `{ T = [id(I)|TT] }.`



# Lexikální analýza - komplet 2

- `tokens(T) -->`  
    `operator(O), !,`  
    `tokens(TT),`  
    `{ maplist(mkop,O,OO), append(OO,TT,T) }.`
- `tokens(T) -->`  
    `ws(_), !,`  
    `tokens(TT),`  
    `{ T = TT }.`
- `tokens([]) -->`  
    `[].`

# Syntaktická analýza (shora dolů)

## - program

- $\text{prog}(P) \rightarrow$   
     $\text{letpart}(L), !,$   
     $\text{prog}(PP),$   
     $\{ P = [L|PP] \}.$
- $\text{prog}(P) \rightarrow$   
     $\text{printpart}(L), !,$   
     $\text{prog}(PP),$   
     $\{ P = [L|PP] \}.$
- $\text{prog}([]) \rightarrow$   
     $[].$

# Syntaktická analýza - definice

- letpart(P) -->  
    ['let',  
      id(I),  
      op(asg)],!,  
    log(E),  
    [op(sem)],  
    { P = let(I,E) }.

# Syntaktická analýza - výstup

- `printpart(P) -->`  
    `['print'],!,`  
    `log(E),`  
    `[op(sem)],`  
    `{ P = print(E) }.`

# Syntaktická analýza - logické operace

- `log(E) -->`  
    `neg(M), llog(MM),`  
    `{ rebind(nxn,MM,M,[],[],E) }.`  
`log(E) -->`  
    `neg(M),`  
    `{ E = M }.`
- `rebind(S,F,S,B,N,G) :-`  
    `F=..[H,P1,P2], S=P1, !, X=..[N,S,B], G=..[H,X,P2].`  
`rebind(S,F,L,_,_,G) :-`  
    `F=..[H,P1,P2], S=P1, !, G=..[H,L,P2].`  
`rebind(S,F,L,B,N,G) :-`  
    `F=..[H,P1,P2], rebind(S,P1,L,B,N,R), G=..[H,R,P2].`

# Syntaktická analýza - logické binární operátory

- $\text{llog}(E) \rightarrow$   
     $[\text{op}(\text{and})], \text{neg}(B), \text{llog}(EE),$   
     $\{ \text{rebind}(\text{nxn}, EE, \text{nxn}, B, \text{and}, E) \}.$
- $\text{llog}(E) \rightarrow$   
     $[\text{op}(\text{and})], \text{neg}(B),$   
     $\{ E = \text{and}(\text{nxn}, B) \}.$
- $\text{llog}(E) \rightarrow$   
     $[\text{op}(\text{or})], \text{neg}(B), \text{llog}(EE),$   
     $\{ \text{rebind}(\text{nxn}, EE, \text{nxn}, B, \text{or}, E) \}.$
- $\text{llog}(E) \rightarrow$   
     $[\text{op}(\text{or})], \text{neg}(B),$   
     $\{ E = \text{or}(\text{nxn}, B) \}.$

# Syntaktická analýza - logická negace

- $\text{neg}(E) \rightarrow$   
     $\text{equ}(E).$   
 $\text{neg}(E) \rightarrow$   
     $[\text{not}], \text{neg}(EE),$   
     $\{ E = \text{not}(EE) \}.$

# Syntaktická analýza - rovnost/nerovnost

- `equ(E) -->`  
    `ord(M),`  
    `eequ(MM),`  
    `{ rebind(nxn,MM,M,[],[],E) }.`  
`equ(E) -->`  
    `ord(M),`  
    `{ E = M }.`



# Syntaktická analýza - operátory rovnosti

- `eequ(E) -->`  
    `[op(eq)], ord(B), eequ(EE),`  
    `{ rebind(nxn,EE,nxn,B,eq,E) }.`  
`eequ(E) -->`  
    `[op(eq)], ord(B),`  
    `{ E = eq(nxn,B) }.`  
`eequ(E) -->`  
    `[op(neq)], ord(B), eequ(EE),`  
    `{ rebind(nxn,EE,nxn,B,neq,E) }.`  
`eequ(E) -->`  
    `[op(neq)], ord(B),`  
    `{ E = neq(nxn,B) }.`

# Syntaktická analýza - uspořádání

- `ord(E) -->`  
    `expr(M), oord(MM),`  
    `{ rebind(nxn,MM,M,[],[],E) }.`  
`ord(E) -->`  
    `expr(M),`  
    `{ E = M }.`

# Syntaktická analýza - operátory uspořádání 1

- `oord(E) -->`  
    `[op(lt), expr(B), oord(E),`  
    `{ rebind(nxn,EE,nxn,B,lt,E) }.`
- `oord(E) -->`  
    `[op(lt), expr(B),`  
    `{ E = lt(nxn,B) }.`
- `oord(E) -->`  
    `[op(gt), expr(B), oord(E),`  
    `{ rebind(nxn,EE,nxn,B,gt,E) }.`
- `oord(E) -->`  
    `[op(gt), expr(B),`  
    `{ E = gt(nxn,B) }.`

# Syntaktická analýza - operátory uspořádání 2

- $\text{oord}(E) \rightarrow$   
     $[\text{op}(\text{leq})], \text{expr}(B), \text{oord}(EE),$   
     $\{ \text{rebind}(\text{nxn}, EE, \text{nxn}, B, \text{leq}, E) \}.$
- $\text{oord}(E) \rightarrow$   
     $[\text{op}(\text{leq})], \text{expr}(B),$   
     $\{ E = \text{leq}(\text{nxn}, B) \}.$
- $\text{oord}(E) \rightarrow$   
     $[\text{op}(\text{geq})], \text{expr}(B), \text{oord}(EE),$   
     $\{ \text{rebind}(\text{nxn}, EE, \text{nxn}, B, \text{geq}, E) \}.$
- $\text{oord}(E) \rightarrow$   
     $[\text{op}(\text{geq})], \text{expr}(B),$   
     $\{ E = \text{geq}(\text{nxn}, B) \}.$

# Syntaktická analýza - aritmetické výrazy s aditivními operátory

- $\text{expr}(E) \rightarrow$   
     $\text{muls}(M), \text{eexpr}(MM),$   
     $\{ \text{rebind}(nxn, MM, M, [], [], E) \}.$   
 $\text{expr}(E) \rightarrow$   
     $\text{muls}(M),$   
     $\{ E = M \}.$

# Syntaktická analýza - aditivní operátory

- `eexpr(E) -->`  
    `[op(add)], muls(B), eexpr(EE),`  
    `{ rebind(nxn,EE,nxn,B,add,E) }.`  
`eexpr(E) -->`  
    `[op(add)], muls(B),`  
    `{ E = add(nxn,B) }.`  
`eexpr(E) -->`  
    `[op(sub)], muls(B), eexpr(EE),`  
    `{ rebind(nxn,EE,nxn,B,sub,E) }.`  
`eexpr(E) -->`  
    `[op(sub)],muls(B),`  
    `{ E = sub(nxn,B) }.`

# Syntaktická analýza - aritmetické výrazy s multiplikativními op.

- `muls(E) -->`  
    `base(B), mmuls(BB),`  
    `{ rebind(nxn,BB,B,[],[],E) }.`  
`muls(E) -->`  
    `base(B),`  
    `{ B = E }.`

# Syntaktická analýza - multiplikativní operátory 1

- $\text{mmuls}(E) \rightarrow$   
     $[\text{op}(\text{mul})], \text{base}(B), \text{mmuls}(EE),$   
     $\{ \text{rebind}(\text{nxn}, EE, \text{nxn}, B, \text{mul}, E) \}.$   
 $\text{mmuls}(E) \rightarrow$   
     $[\text{op}(\text{mul})], \text{base}(B),$   
     $\{ E = \text{mul}(\text{nxn}, B) \}.$   
 $\text{mmuls}(E) \rightarrow$   
     $[\text{op}(\text{div})], \text{base}(B), \text{mmuls}(EE),$   
     $\{ \text{rebind}(\text{nxn}, EE, \text{nxn}, B, \text{div}, E) \}.$   
 $\text{mmuls}(E) \rightarrow$   
     $[\text{op}(\text{div})], \text{base}(B),$   
     $\{ E = \text{div}(\text{nxn}, B) \}.$



# Syntaktická analýza - multiplikativní operátory 2

- $\text{mmuls}(E) \rightarrow$   
     $[\text{op}(\text{mod})], \text{base}(B), \text{mmuls}(EE),$   
     $\{ \text{rebind}(\text{nxn}, EE, \text{nxn}, B, \text{modulo}, E) \}.$   
 $\text{mmuls}(E) \rightarrow$   
     $[\text{op}(\text{mod})], \text{base}(B),$   
     $\{ E = \text{modulo}(\text{nxn}, B) \}.$

# Syntaktická analýza - bazové výrazy

- $\text{base}(E) \rightarrow$   
     $[\text{num}(N)],$   
     $\{ E = \text{num}(N) \}.$   
 $\text{base}(E) \rightarrow$   
     $[\text{id}(I)],$   
     $\{ E = \text{id}(I) \}.$   
 $\text{base}(E) \rightarrow$   
     $[\text{op}(lp)], \log(E), [\text{op}(rp)], !.$   
 $\text{base}(E) \rightarrow$   
     $[\text{if}], \log(C), [\text{then}], !, \log(E1), [\text{else}], !, \log(E2),$   
     $\{ E = \text{if}(C, E1, E2) \}.$

# Vyhodnocení výrazů 1

- `eval(_, num(N), N).`  
`eval(M, id(I), N) :-`  
    `search(I, M, N).`  
`eval(M, if(C,E1,E2), N) :-`  
    `eval(M, C, CC),`  
    `(CC==0, eval(M, E2, N) ; CC\=0, eval(M, E1, N)).`  
`eval(M,mul(E1,E2),N) :-`  
    `eval(M, E1, N1), eval(M, E2, N2), N is N1 * N2.`  
`eval(M, div(E1,E2), N) :-`  
    `eval(M, E1, N1), eval(M, E2, N2), N is N1 // N2.`  
`eval(M, mod(E1,E2), N) :-`  
    `eval(M, E1, N1), eval(M, E2, N2), N is N1 mod N2.`

# Vyhodnocení výrazů 2

- $\text{eval}(M, \text{add}(E1, E2), N) :-$   
     $\text{eval}(M, E1, N1), \text{eval}(M, E2, N2), N \text{ is } N1 + N2.$
- $\text{eval}(M, \text{sub}(E1, E2), N) :-$   
     $\text{eval}(M, E1, N1), \text{eval}(M, E2, N2), N \text{ is } N1 - N2.$
- $\text{eval}(M, \text{lt}(E1, E2), N) :-$   
     $\text{eval}(M, E1, N1), \text{eval}(M, E2, N2),$   
     $(N1 < N2, N=1 ; N1 \geq N2, N=0).$
- $\text{eval}(M, \text{gt}(E1, E2), N) :-$   
     $\text{eval}(M, E1, N1), \text{eval}(M, E2, N2),$   
     $(N1 > N2, N=1 ; N1 \leq N2, N=0).$
- $\text{eval}(M, \text{leq}(E1, E2), N) :-$   
     $\text{eval}(M, E1, N1), \text{eval}(M, E2, N2),$   
     $(N1 \leq N2, N=1 ; N1 > N2, N=0).$

# Vyhodnocení výrazů 3

- `eval(M, geq(E1,E2), N) :-`  
    `eval(M, E1, N1), eval(M, E2, N2),`  
    `(N1>=N2, N=1 ; N1<N2, N=0).`  
`eval(M, eq(E1,E2), N) :-`  
    `eval(M, E1, N1), eval(M, E2, N2),`  
    `(N1==N2, N=1 ; N1\=N2, N=0).`  
`eval(M, neq(E1,E2), N) :-`  
    `eval(M, E1, N1), eval(M, E2, N2),`  
    `(N1\=N2, N=1 ; N1==N2, N=0).`  
`eval(M, not(E) ,N) :-`  
    `eval(M, E, N1),`  
    `(N1==0, N=1 ; N1\=0, N=0).`

# Vyhodnocení výrazů 4

- $\text{eval}(M, \text{and}(E1, E2), N) :-$   
     $\text{eval}(M, E1, N1), \text{eval}(M, E2, N2),$   
     $(N1 \neq 0, N2 \neq 0, !, N = 1 ; N = 0).$   
 $\text{eval}(M, \text{or}(E1, E2), N) :-$   
     $\text{eval}(M, E1, N1), \text{eval}(M, E2, N2),$   
     $((N1 \neq 0 ; N2 \neq 0), !, N = 1 ; N = 0).$

# Tabulka symbolů

- `search(_, [], 0).`  
`search(I, [(I,N)|_], N) :- !.`  
`search(I, [_|T], N) :-`  
    `search(I, T, N).`
- `change(I, [], V, [(I,V)]).`  
`change(I, [(I,_)|T], V, [(I,V)|T]) :- !.`  
`change(I, [H|T], V, [H|L]) :-`  
    `change(I,T,V,L).`

# Příkazy

- `command(M, let(I,E), nothing, MM) :-  
    eval(M, E, N), change(I, M, N, MM).`  
`command(M, print(E), N, M) :-  
    eval(M, E, N).`
- `commands([], []) :- !.`  
`commands(C, R) :- cml([], C, R).`
- `cml(_, [], []).`  
`cml(M, [C|T], R) :-  
    command(M, C, nothing, MM),!, cml(MM, T, R).`  
`cml(M, [C|T], [V|R]) :-  
    command(M, C, V, MM), cml(MM,T,R).`



# Hlavní predikát

- `calc(S, R) :-`  
    `phrase(tokens(T), S, []),`  
    `phrase(prog(P), T, []),`  
    `commands(P,R), !.`  
`calc(S, 'Lexical error') :-`  
    `phrase(tokens(_), S, [_|_]), !.`  
`calc(S, 'Syntax error') :-`  
    `phrase(tokens(T), S, []),`  
    `phrase(prog(_), T, [_|_]), !.`  
`calc(_, 'Semantic error').`

# Důležité pojmy

- DCG
- Operátor -->



# Úkoly



- Navrhňte jednoduchý systém pro zadávání příkazů počítači pomocí krátkých jednoduchých vět přirozeného jazyka (asi ideálně anglicky, nebo v češtině bez skloňování, časování, ...), implementujte rozpoznávač těchto vět pomocí DCG, vyzkoušejte nejednoznačnost gramatiky



# Vlákna v Ciao Prologu

# Cíle oddílu



- Užití vláken a paralelismu v Prologu
  - Oddělení vláken
  - Synchronizace
  - Rušení vláken
  - ...



# Základní pravidla

- Obdobná, jako v jazyku Haskell  
:- use\_module(library(concurrency)).
- Problém
  - Skutečný paralelismus vyžadující GC



# Dynamické predikáty

- Předávání parametrů
- Předávání výsledků
- Synchronizace
  
- `:- concurrent res/1, run/1.`

# Vytvoření vlákna

- `eng_call(+Goal, +EngineCreation, +ThreadCreation, -GoalId)`
  - Cíl
  - Způsob vytvoření stroje (wait, create)
  - Způsob vytvoření vlákna (self, wait, create)
  - ID
  - Selže, pokud selže i cíl





# Čekání na dokončení výpočtu

- `eng_wait(+GoalId)`
  - ID cíle
  - Čeká, až stroj s daným ID dokončí výpočet



# Měkké ukončení vlákna

- `eng_release(+GoalId)`
  - ID cíle
  - Pokud je stroj s daným ID ve stavu idle, tak je uvolněn (jinak pád, test pomocí `wait`)
- *Viz `eng_kill`*



# Paralelní/rychlá údržba faktů

- `asserta_fact`
- `assertz_fact`
- `retract_fact`



# Synchronizace

## ■ Užití faktů

- Operace `*_fact` jsou blokující, pokud jsou fakta definována jako paralelní
- Může být ale zrádné



# Příklad

- Paralelní mergesort s maximálně 2 vlákny
- ```
:- use_module(library(concurrency)).  
:- use_module(library(lists)).  
:- use_module(library(system)).
```

# Příklad (pokr.)

- :- concurrent res/1, run/1.
- merge([],L,L) :- !.  
merge([H|T],[],[H|T]) :- !.  
merge([H1|T1],[H2|T2],[H1|T]) :-  
    H1 =< H2,  
    merge(T1,[H2|T2],T), !.  
merge([H1|T1],[H2|T2],[H2|T]) :-  
    H1 > H2,  
    merge([H1|T1],T2,T).

# Příklad (pokr.)

- `separate(L,0,[],L) :- !.`  
`separate([H|T], N, [H|L],R) :-`  
    `N>0,`  
    `NN is N - 1,`  
    `separate(T,NN,L,R).`

- *Obdoba splitAt*

# Příklad (pokr.)

- `msrts([],[]) :- !.`  
`msrts([X],[X]) :- !.`  
`msrts(L,SL) :-`  
    `length(L,X), M is X // 2,`  
    `separate(L,M,LL,RL),`  
    `msrts(LL,SLL),`  
    `msrts(RL,SRL),`  
    `merge(SLL,SRL,SL).`
- *Sekvenční verze, nutný vstup*



# Příklad (pokr.)

■ `msrtp([],[]) :- !.`  
`msrtp([X],[X]) :- !.` }  
`msrtp(L,SL) :-`  
    `length(L,X), M is X // 2,`  
    `separate(L,M,LL,RL),`  
    `eng_call(msrtphx,create,create,GID1),`

Shodné  
jako u  
sekvenční  
verze

Vytvoření  
vlákna

# Příklad (pokr.)



...

```
assertz_fact(run(LL)),  
msrts(RL,SRL),  
{ retract_fact(res(SLL)),  
  eng_wait(GID1),
```

Odstartuje  
paralelní vlákno

Seřadí druhou  
polovinu v  
daném vlákně

Měly by jít  
prohodit?

Čeká na  
zastavení  
činnosti vlákna

Vezme  
polovinu  
seřazenou  
jiným vláknem

# Příklad (pokr.)



...

```
eng_release(GID1),  
merge(SLL,SRL,SL),  
!..
```

Odstraní  
paralelní vlákno

Složí výsledek

# Příklad (konec)

■ msrtphx :-  
    retract\_fact(run(L)),  
    msrts(L,R),  
    assertz\_fact(res(R)).

Odstartuje  
vlákno, jak je  
výsledek  
dostupný

Uloží výsledek  
pro hlavní  
vlákno

# Důležité pojmy

- Synchronizace
- Fakty pro paralelní zpracování
- Čekání na vlákno
- Problematika paralelní změny dat (?)



# Úkoly



- Prostudujte další predikáty knihovny Concurrency
- Navrhňte již vámi implementované predikáty pro paralelizaci
- Paralelizujte je tak, že hlavní vlákno bude čekat na další 2 pracující



# Gödel

# Cíle oddílu

## ■ Jazyk Gödel

- Základní syntaktická pravidla
- Sémantika vyhodnocení
- Datové typy
- Moduly
- ...







# Programovací jazyk Gödel

## ■ Autoři:

- P.M.Hill, J.W. Lloyd
- MIT/Bristol

## ■ Implementace:

- SUN SPARC
- Linux

## ■ Implementační a cílový jazyk:

- SICStus Prolog



# Hlavní prvky jazyka

- **typový systém**
  - lepší vyjádření sémantiky (než Prolog)
  - efektivnější cílový kód
  - účinnější ladění
- **moduly**
- **prostředky pro řízení**
- **meta-programování**
- **vstup/výstup**

# Porovnání s jazykem Prolog

## ■ Požadavky na programovací jazyk:

- vysoká úroveň specifikace
- vysoká vyjadřovací schopnost
  - » nemá typy
- efektivita
- praktická použitelnost
- jednoduchá (matematická) sémantika
  - » problémy!

# Porovnání s jazykem Prolog

## ■ Problémy:

- mnoho programů nemá deklarativní sémantiku
  - » ?- var(X), solve(p(X)).                      p(a).
  - » ?- solve(p(X)), var(X).
- procedurální sémantika je velmi složitá
- operátor řezu
- unifikace bez kontroly výskytu
- negace není bezpečná

■ *Prolog je vhodným mezijazykem pro překlad Gödelu*



# Typový systém jazyka Gödel

- typové systémy vhodné pro logiku vyšších řádů
  - typovaný lambda kalkul
  - pro FPJ a LPJ vyšších řádů
- *více-druhov<sup>á</sup> logika prvního řádu*  
+  
*parametrický polymorfismus*

# Zavedení typových proměnných

## ■ Prolog

- `append([1,2,3],[po,ut,st],X).`
- `X = ...`

## ■ Gödel

- `append([1,2,3],[Po,Ut,St],x).`
- `????`



# Typy v jazyku Gödel

- silně typovaný jazyk
  - všechny konstanty, funkce, tvrzení a predikáty v programu a jejich typy je třeba specifikovat
  - proměnné mají typy odvozené z kontextu

# Vícedruhovost 1

- MODULE M1.

BASE Day, ListOfDay.

CONSTANT Nil: ListOfDay;  
Sunday, Monday, Tue... : Day.

PREDICATE

Append: ListOfDay\*ListOfDay\*ListOfDay;  
Append3: ListOfDay\*ListOfDay\*ListOfDay  
\*ListOfDay.



# Vícedruhovost 2

- $\text{Append}(\text{Nil}, x, x).$   
 $\text{Append}(\text{Cons}(u,x),y,\text{Cons}(u,z)) \leftarrow$   
 $\text{Append}(x,y,z)$
- $\text{Append3}(x,y,z,u) \leftarrow \text{Append}(x,y,w) \ \& \ \text{Append}(w,z,u)$



# Vícedruhovost 3

## ■ symboly

- Day, Nil, Cons, Append, ...

## ■ kategorie

- báze, konstruktor, konstanta, funkce, výrok, predikát

## ■ cíle

- $\leftarrow \text{Append3}(\text{Cons}(\text{Monday}, \text{Nil}), \text{Cons}(\text{Tuesday}, \text{Nil}), \text{Cons}(\text{Wednesday}, \text{Nil}), x).$

## ■ typy

- Day, ListOfDay

# Konstruktor 1

- MODULE M2.  
BASE Day.  
CONSTRUCTOR List/1.  
CONSTANT Nil : List(Day);  
Sunday, Monday, Tue... : Day.  
FUNCTION Cons: Day \* List(Day) → List(Day).  
PREDICATE  
Append: List(Day)\*List(Day)\*List(Day);  
Append3: List(Day)\*List(Day)\*List(Day)  
\*List(Day).



# Konstruktory 2

## ■ typy

- Day, List(Day), List(List(Day)), ...
  - BASE Day, Person.
- typy: Day, Person, List(Day), List(Person), ...

## ■ pracujeme s typy (sortami), které mohou (ale nemusí) mít strukturu

# Polymorfismus 1

- MODULE M3.  
BASE Day, Person.  
CONSTRUCTOR Nil: List(a);  
Sunday, Monday, Tue... : Day;  
Fred, Bill, Mary : Person.  
FUNCTION Cons: a \* List(a) → List(a).  
PREDICATE  
Append: List(a)\*List(a)\*List(a);  
Append3: List(a)\*List(a)\*List(a)\*List(a).

# Polymorfismus 2

- **typy**

- `a`, `List(a)`, `List(List(Day))`, ...

- **základní typy (monotypy):**

- `Day`, `Person`, `List(Day)`, `List(List(Person))`, ...

- **polymorfní symboly**

- `Nil`
  - `NilDay`, `NilPerson`, ...

# Seznamy

- CONSTRUCTOR List/1.  
CONSTANT Nil: List(a).  
FUNCTION Cons :  $a * \text{List}(a) \rightarrow \text{List}(a)$ .
- Cons(Fred, Cons(Bill, Cons(Mary, Nil)))  
= [Fred, Bill, Mary]
- Cons(Fred, Cons(Bill, x))  
= [Fred, Bill | x]

# Append3

- MODULE M4.  
IMPORT Lists.  
BASE Day, Person.  
CONSTANT Sunday, Moday, Tue, ... : Day;  
Fred, Bill, Mary : Person.  
PREDICATE Append3: ... .  
...
- ← Append3([Monday],  
[Tuesday], [Wednesday], x).
- ← Append3(x, y, z, [Fred, Bill, Mary]).





# Modul Lists

- EXPORT Lists.

IMPORT Integers.

CONSTRUCTOR List/1.

CONSTANT Nil: List(a).

FUNCTION Cons :  $a * \text{List}(a) \rightarrow \text{List}(a)$ .

# Modul Lists (pokr.)

- PREDICATE

Member:  $a * \text{List}(a)$ ;

Append:  $\text{List}(a) * \text{List}(a) * \text{List}(a)$ ;

Permutation:  $\text{List}(a) * \text{List}(a)$ ;

Delete:  $a * \text{List}(a) * \text{List}(a)$ ;

DeleteFirst:  $a * \text{List}(a) * \text{List}(a)$ ;

Reverse:  $\text{List}(a) * \text{List}(a)$ ;

Prefix:  $\text{List}(a) * \text{Integer} * \text{List}(a)$ ;

Suffix:  $\text{List}(a) * \text{Integer} * \text{List}(a)$ ;

Length:  $\text{List}(a) * \text{Integer}$ ;



# Modul Lists (pokr.)

- PREDICATE

Sorted: List(Integer);

Sort: List(Integer)\*List(Integer);

Merge: List(Integer)\*List(Integer)  
\*List(Integer).

# Kvantifikátory a spojky

- $\&$  konjunkce
- $\vee$  disjunkce
- $\sim$  negace
- $\rightarrow$  implikace
- $\leftarrow$  implikace
- $\leftrightarrow$  ekvivalence
- ALL [x,y,...] e univerzální kvantifikátor
- SOME [x,y,...] e existenční kvantifikátor

# Podseznamy

- MODULE Inclusion.  
IMPORT Lists.  
PREDICATE IncludedIn : List(a) \* List(a).

IncludedIn(x,y)  $\rightarrow$   
ALL [z] (Member(z,y)  $\leftarrow$  Member(z,x)).

# Podseznamy (pokr.)

- $\leftarrow \text{IncludedIn}([1,3,2],[4,3,2,1]).$
- $\leftarrow \text{IncludedIn}([1,5,2],[4,3,2,1]).$
- $\text{ALL } [x] (\text{SOME } [y] \text{ Mother}(y,x) \rightarrow \text{SOME } [z] \text{ Father}(z,x)).$
- $\text{ALL } [x] (\text{Mother}(\_,x) \rightarrow \text{Father}(\_,x)).$
- $x: \text{Parent}(x,y) \ \& \ \text{Parent}(y,\text{Jane}).$

# Operátory

- FUNCTION *indikátor*  
^ : yFx (540) : Integer \* Integer \* Integer;  
*asociativita* *priorita*  
xFx    xFy  
yFx

# Operátory (pokr.)

- FUNCTION

- :  $Fy$  (530) : Integer  $\rightarrow$  Integer

$Fx$   $Fy$

$xF$   $yF$

- PREDICATE

> :  $zPz$  : Integer  $\rightarrow$  Integer

$zPz$

$Pz$   $zP$

- *operátor = funkce nebo predikát obsahující v deklaraci indikátor*



# Rovnost

- rovnost je zcela základní predikát v logice

- není nutno importovat
- $\text{PREDICATE } = : zPz : a^*a.$   
 $\text{PREDICATE } \sim = : zPz : a^*a.$   
 $x \sim = y \leftrightarrow \sim(x=y).$

– zabudované výroky True a False

- True.

*definice False je prázdná!*

# Čísla

## ■ Modul Integers

- $\wedge$  - \* Div Mod Rem + -  
Abs Sign Max Min

## BASE Integer

} *funkce*

- $> < >= =<$

Interval : Integer \* Integer \* Integer

–  $r \leq s \leq t \sim \rightarrow \text{Interval}(r,s,t).$

} *predikáty*



# Čísla (pokr.)

## ■ Modul Rationals

BASE Rational.

- $2//3$   
StandardRational(r,p,q).

## ■ Modul Floats

BASE Float.

- Sqrt, Round, Sin, Exp, ...

# Formule s podmínkou

- IF Podmínka THEN Formule
- IF SOME  $[x_1, \dots, x_n]$  Podmínka THEN Formule
- IF Podmínka THEN Formule<sub>1</sub> ELSE Formule<sub>2</sub>
  - $\text{Max}(x,y,z) \leftarrow \text{IF } x \leq y \text{ THEN } z=y \text{ ELSE } z=x.$
- IF SOME  $[x_1, \dots, x_n]$  Podmínka  
THEN Formule<sub>1</sub> ELSE Formule<sub>2</sub>

# Příklad

- ```
MODULE AssocList.  
  IMPORT Strings.  
  BASE PairType.  
  FUNCTION Pair: Integer*String → PairType.  
  PREDICATE  
    Lookup: Integer*String*List(PairType)  
      *List(PairType).
```

# Příklad (pokr.)

- `Lookup(key, value, assocl, new_assocl) ←`  
    `IF SOME[v] Member(Pair(key,v),assocl)`  
    `THEN`  
        `value = v`  
        `& new_assocl = assocl`  
    `ELSE`  
        `new_assocl =`  
            `[Pair(key,value) | assocl]`

# Poznámky

- Gödel pracuje s omezeními (constraints)
  - vyhodnocování konstantních výrazů s celými a racionálními čísly
  - práce s intervaly
    - $1 < x \leq 5$
  - řešení lineárních rovnic
    - $2 \cdot x + 1 = y + 2 \ \& \ 3 \cdot y - 2 = 1.$
    - $x/2 + y/3 = 5/6 \ \& \ y/2 + 1/3 = 5/6.$
  - hledání všech možností
    - $x^2 + y^2 = z^2 \ \& \ 1 < x < 50 \ \& \ 1 < y < 50 \ \& \ 0 < z.$



# Poznámky (pokr.)

- Omezení se mohou řešit v libovolném pořadí
- Pořadí výběru klauzulí není definováno
- Implementace bezpečné negace





# Moduly

- podpora vývoje velkých a složitých programů
  - nezávislý vývoj komponent
  - řešení problémů s konflikty jmen
  - možnost zakrytí implementačních detailů

# Příklad

- **EXPORT** M5.  
IMPORT Lists.  
BASE Day, Person.  
CONSTANT Sunday, Monday, Tue... : Day;  
Fred, Bill, ... : Person.  
PREDICATE Append3: List(a)\*List(a)\*... .
- **LOCAL** M5.  
Append3(x,y,z,u)  $\leftarrow$  Append(x,y,w),App... .
- **MODULE** M6.  
IMPORT M5.  
...

# Příklad - GCD

- MODULE GCD.  
 IMPORT Integers.

PREDICATE Gcd : Integer \* Integer \* Integer.

Gcd(i,j,d)  $\leftarrow$   
 ComDiv(i,j,d)  
 &  $\sim$  SOME [e] (ComDiv(i,j,e) & e>d).

# Příklad - GCD (pokr.)

- PREDICATE ComDiv: Integer\*Integer\*Integer.

```
ComDiv(i,j,d) ←  
  IF (i=0 ∨ j=0)  
  THEN  
    d = Max(Abs(i),Abs(j))  
  ELSE  
    1 ≤ d ≤ Min(Abs(i),Abs(j))  
    & i Mod d = 0  
    & j Mod d = 0.
```

# Důležité pojmy

- Modul
- Typy a jejich druhy
- Konstruktory a jejich kategorie



# Úkoly



- Nainstalujte si systém Gödel a vyzkoušejte příklady v něm
- Vytvořte modul pro uspořádání seznamu dle zadaných kritérií a odzkoušejte
- V rámci toho modulu implementuje různé řadicí algoritmy – implementace porovnejte s těmi v Prologu



# CLP

# Cíle oddílu

- Co to je CLP?
- Použití
- Srovnání s Prologem
- Přehled systémů







# Constraint Logic Programming

- Motivace - nevýhody čistě logických jazyků
  - všechno musí být reprezentováno prvky Herbrandova univerza (neinterpretované termy)
  - programování na nízké úrovni rozlišení
  - nejsou plně implementovány
  - explicitní reprezentace objektů/výsledků
  - prohledávání do hloubky
  - metoda „generuj a testuj“ (mnohdy se nehodí)

# Jak to dělá Prolog

- $\text{fact}(N,F) :- N>0, N1 \text{ is } N-1,$   
 $\text{fact}(N1,M), F = N*M.$   
 $\text{fact}(0,1).$
- ?-  $\text{fact}(3,F).$ 
  - ~  $C1 = \{3>0, F=3*M1\}$
  - ~  $C2 = C1 \cup \{2>0, M1=2*M2\}$
  - ~  $C3 = C2 \cup \{1>0, M2=1*M3\}$
  - ~  $C4 = C3 \cup \{0=0, M3=1\}$

$$\sim \mathbf{F} = 6$$



# Unifikace v Prologu

## ■ Unifikace

- je rovnost  $t=s$  řešitelná?
- Reprezentace explicitního řešení ( $\text{mgu}(t,s)$ )
  - speciální případ omezení (rovnost nad termy Herbrandova univerza)

## ■ CLP

- nahrazení unifikace splněním omezení!

# Unifikace a omezení

- **UNIFIKACE**

**x**

- **OMEZENÍ**

teorie  $\tau$  a (alg.)  
struktura  $A$

- rovnostní teorie

- 

*Otázka*

- je  $t=s$  řešitelné?  
jaká jsou řešení?

Je  $C$  řešitelné?

- 

*Výsledek*

- úplná množina mgu
- Problémy: nespočetné struktury, nekonečné sjednocení mgu, nemusí existovat toto sjednocení

ANO-NE



# CLP

# X

# LP

- *Doména*
- struktura  $A$  (včetně  $R$ ) Herbrandovo univerzum
- *Syntaxe*
- klauzule s omezeními klauzule
- *Operační interpretace*
- řešení omezení (nejen rovnost) unifikace  
SLD rezoluce SLD rezoluce
- *Deklarativní interpretace*
- Pravdivost získána úspěšným odvozením  
Nepravdivost získána konečně selhávajícím odvozením
- *Výstup*
- omezení nejobecnější unifikátor



# Účel omezení

- Manipulace s *interpretovanými* objekty
  - CLP(A)
- Vylepšení prohledávacích metod
  - constraint satisfaction



# Výhody

- Efektivnější vyhledávání - doplnění *backtrackingu* technikami zachování konzistence
- Omezení se vyhodnocují deterministicky před nedeterministickým prohledáváním
- „Constraint and Generate“
  - nikoliv „Generate and Test“

# Program v CLP(A)

## ■ konečná množina pravidel

- $A_0 \text{ :- } C_1, C_2, \dots, C_n, A_1, A_2, \dots, A_m$   
omezení nad atomy nad  
relacemi  $A$  a  $A$ -termy  
 $A$ -termy

## ■ dotaz

- $?- C_1, C_2, \dots, C_n, A_1, A_2, \dots, A_m.$

$n+m>0$



# Operační sémantika

## ■ cíl $G =$

- ( atomy,  $a_1, \dots, a_n$   
řešená omezení,  $c_1, \dots, c_m$   
odložená omezení )  $d_1, \dots, d_k$

## ■ derivační krok

- vybereme omezení  $d_i$  a přidáme do množiny řešených omezení, přičemž tato množina musí zůstat řešitelná
- vybereme atom  $a_i$  a pravidlo  $h:- e_1 \dots e_s, b_1 \dots b_r$ ;  
 $a_i$  nahradíme atomy  $b_1 \dots b_r$  a k odloženým omezením přidáme  $e_1 \dots e_s$

# Příklad - Prolog

- `fib(0,1).`  
`fib(1,1).`  
`fib(N,F) :- N>1, N1 is N-1, N2 is N-2,`  
`fib(N1,F1), fib(N2,F2), F is F1+F2.`
- `?- fib(5,X).`  
`~> X = 8`
- `?- fib(X,8).`  
`~> ?`
- `? B >= 80, B <= 90, fib(A,B).`  
`~> ?`

**FUNKCE**

# Příklad - CLP

- `fib(0,1).`  
`fib(1,1).`  
`fib(N,X1+X2) :- N>1, fib(N-1,X1), fib(N-2,X2).`
- `?- fib(5,X).`  
`~> X = 8`
- `?- fib(X,8).`  
`~> X = 5`
- `? B >= 80, B <= 90, fib(A,B).`  
`~> A = 10, B = 89`

**RELACE**

# Způsob vyhodnocení

- $\text{equal}(\text{pair}(X,Y)) \text{ :- } X = Y.$
- $\text{?- equal}(\text{pair}(2+5,7)), p(X).$
- $\sim \text{pair}(2+5,7) = \text{pair}(X_0,Y_0), X_0=Y_0, p(X).$ 
  - $f(t_1, \dots, t_n) = f(s_1, \dots, s_n)$   
 $\Downarrow$   
 $t_1 = s_1, \dots, t_n = s_n$
- $\sim 2+5 = X_0, 7 = Y_0, X_0 = Y_0, p(X).$
- $\sim \text{> } p(X).$

# Existující CLP systémy

## ■ CLP( $\mathcal{R}$ )

- nejstarší
- University of Monash, Australia (IBM)
- UNIX, DOS, Windows
  - pro výuku a výzkum zdarma

# Existující CLP systémy (pokr.)

## ■ CHIP

- European computer-industry research center
  - ICL + Bull + Siemens
- velmi propracovaný systém, racionální čísla s libovolnou přesností
- základ několika komerčních systémů

# Existující CLP systémy (pokr.)

## ■ CHARME

- Bull (Francie?)
- nemá neinterpretované termy
  - integer + matice + destruktivní přiřazení
  - syntaxe jako Pascal nebo C
- mechanismus démonů (pozastavení funkce)
- z uváděných asi nejpropracovanější systém
  - debugger
  - X-Window interface



# Existující CLP systémy (pokr.)

## ■ PROLOG III

- University of Marseille
- lineární racionální aritmetika
- booleovské termy
- konečné seznamy





# CLP( $\mathbb{R}$ ) - aproximace CLP(A)

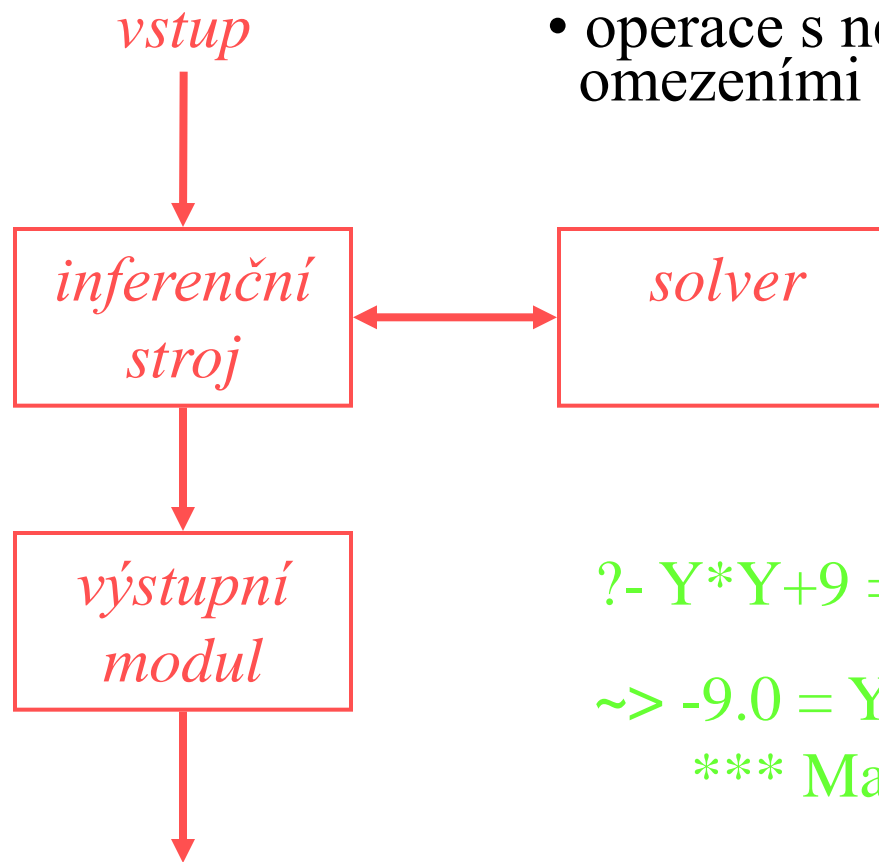
- ne-aritmetické rovnice se řeší unifikací bez „kontroly výskytu“
- podcíle se vybírají zleva doprava
- pravidla se vybírají shora dolů
- reprezentace reálných čísel v pohyblivé čárce
- implementace lineárního solveru (úplná)

# CLP( $\mathcal{R}$ ) - aproximace CLP(A)

- typová kontrola
- syntakticky není rozdíl mezi programy v Prologu a v CLP( $\mathcal{R}$ )
  - *rozdíl je v interpretaci*

# CLP( $\mathbb{R}$ )

- řešení lineárních rovnic
- řešení lineárních nerovnic
- operace s nelineárními omezeními



?-  $Y*Y+9 = 0.$

~>  $-9.0 = Y*Y$

\*\*\* Maybe

# Příklad 1

- $p(10,10).$   
   $q(W, c(U,V)) :-$   
     $W-U+V = 10,$   
     $p(U,V).$
- $?- q(X, c(X+Y, X-Y)).$

# Příklad 2

•

**SEND**  
**+ MORE**  

---

**MONEY**



# Příklad 3

- Rezistory  $R_1$  a  $R_2$ :
  - zapojeny sériově
    - $R = R_1 + R_2$
  - zapojeny paralelně
    - $R = R_1 * R_2 / (R_1 + R_2)$

## Příklad 3 (pokr.)

- `res(r(X), X).`  
`res(cell(X), 0).`  
`res(serial(X,Y),  $R_X+R_Y$ ) :-`  
    `res(X, $R_X$ ), res(Y, $R_Y$ ).`  
`res(parallel(X,Y),  $(R_X*R_Y)/(R_X+R_Y)$ ) :-`  
    `res(X, $R_X$ ), res(Y, $R_Y$ ).`
- `?- res(serial(r(10), r(20)), X).`  
    `~> X=30`

# Příklad 3 (pokr.)

- $\text{res}(\text{serial}(r(X), r(Y)), 30).$

$\langle \text{res}(\text{serial}(X_0, Y_0), R_{X_0} + R_{Y_0}) : -\text{res}(X_0, R_{X_0}), \text{res}(Y_0, R_{Y_0}). \rangle$

$\sim 30 = R_{X_0} + R_{Y_0}, \text{res}(r(X), R_{X_0}), \text{res}(r(Y), R_{Y_0}).$

$\langle \text{res}(r(X_1), X_1) \rangle$

$\sim 30 = R_{X_0} + R_{Y_0}, X = R_{X_0}, \text{res}(r(Y), R_{Y_0}).$

$\langle \text{res}(r(X_2), X_2) \rangle$

$\sim 30 = R_{X_0} + R_{Y_0}, X = R_{X_0}, Y = R_{Y_0}.$

$\sim \rightarrow X = 30 - Y$



## Příklad 3 (pokr.)

- `res(parallel(serial(r(X), cell(5)), r(Y)), 5).`

$$\sim> 5*X + 5*Y = X*Y$$

# Příklad 4

- mortgage(P, Time, IntRate, Bal, MP) :-  
    Time > 0, Time <= 1,  
    Bal =  $P * (1 + \text{Time} * \text{IntRate} / 1200) - \text{Time} * \text{MP}$ .  
mortgage(P, Time, IntRate, Bal, MP) :-  
    Time > 1,  
    mortgage( $P * (1 + \text{IntRate} / 1200) - \text{MP}$ ),  
            Time-1,  
            IntRate,  
            Bal,  
            MP).

## Příklad 4 (pokr.)

- ?- mortgage(100000, 180, 12, 0, MP).  
~> MP = 1200.17
- ?- mortgage(P, 180, 12, 0, 1200.17).  
~> P = 100000
- ?- mortgage(P, 180, 12, Bal, MP).  
~>  $P = 0.166783 * \text{Bal} + 83.3217 * \text{MP}$

# Důležité pojmy

- Omezení
- Operační sémantika



# Úkoly



- Zkuste pro jednoduché formy omezení implementovat v Prologu nadstavbu, která by něco podobného umožňovala