

Neprocedurální programování

Prolog 1

Úvod



Robert Kowalski
*1941



Alain Colmerauer
1941 - 2017

Co je to „neprocedurální programování“ ?

Procedurální (imperativní) programování

- Python, C, C#, Java, assembler, ...
- přiřazovací příkaz
- popisujeme, jak úlohu vyřešit

Neprocedurální programování

- programování bez přiřazovacího příkazu

Neprocedurální programování

Logické programování

- popisujeme problém, který chceme řešit
- prostředky matematické logiky
- Prolog - Programmation en Logique

Funkcionální programování

- program = definice funkcí
- výpočet = aplikace funkce na argumenty
 - » skládání funkcí
 - » “matematické” funkce bez vedlejších efektů
- LISP - List Processing
- Haskell - Haskell Curry

Programmation en Logique

1971 Robert Kowalski (Edinburgh)
Alain Colmerauer (Marseille)

1972 první interpret Prologu

- A. Colmerauer, Philippe Roussel
- 1. program v Prologu = francouzský QA systém

1977 David Warren (Edinburgh)

- kompilátor Prologu
- edinburský dialekt

1983 Warren Abstract Machine (WAM)

- architektura paměti, instrukční sada
- standardní cíl kompilátorů Prologu

Prolog : Historie

1995 ISO Prolog standard [[html](#)]

(1995) ISO/IEC 13211-1 *General Core*

(2000) ISO/IEC 13211-2 *Modules*

Aplikace

- výuka a výzkum
- zpracování přirozeného jazyka
- AI, automatické dokazování vět
- expertní systémy, dotazovací systémy, systémy řízení
- webové aplikace, sémantický web
- programování s omezujícími podmínkami
[[NOPT042](#)]

Prolog : Implementace

- B-Prolog www.picat-lang.org/bprolog/
- BinProlog code.google.com/archive/p/binprolog/
- Ciao ciao-lang.org
- GNU Prolog www.gprolog.org
- SICStus Prolog sicstus.sics.se
- Strawberry Prolog dobrev.com
- SWI Prolog www.swi-prolog.org
- tuProlog tuprolog.apice.unibo.it
- Visual Prolog www.visual-prolog.com
- Win-Prolog www.lpa.co.uk/win.htm
- YAP Prolog github.com/vscosta/yap-6.3

Jednoduchý program v Prologu

```
muz(adam).           % Adam je muž.  
muz(kain).           % Kain je muž.  
muz(abel).           % Abel je muž.  
zena(eva).           % Eva je žena.  
rodic(adam,kain).    % Adam a Eva jsou  
rodic(eva,kain).     % rodiči Kaina.  
rodic(adam,abel).    % Oba jsou i  
rodic(eva,abel).     % rodiči Abela.
```

Jednoduchý program v Prologu

```
muz ( adam ) .  
muz ( kain ) .  
muz ( abel ) .  
zena ( eva ) .  
rodic ( adam , kain ) .  
rodic ( eva , kain ) .  
rodic ( adam , abel ) .  
rodic ( eva , abel ) .
```

f
a
k
t
a

k
l
a
u
z
u
l
e

Fakta v Prologu

unární
funktor

konstanta
(atom)

`muz (adam) .`



term



tečka

Jednoduchý program v Prologu

```
muz ( adam ) .  
muz ( kain ) .  
muz ( abel ) .  
zena ( eva ) .  
rodic ( adam , kain ) .  
rodic ( eva , kain ) .  
rodic ( adam , abel ) .  
rodic ( eva , abel ) .
```

Jednoduchý program v Prologu

muz (adam) .

muz (kain) .

muz (abel) .

zena (eva) .

rodic (adam , kain) .

rodic (eva , kain) .

rodic (adam , abel) .

rodic (eva , abel) .

atomy

Jednoduchý program v Prologu

```
muz ( adam ) .
```

```
muz ( kain ) .
```

```
muz ( abel ) .
```

```
zena ( eva ) .
```

```
rodic ( adam , kain ) .
```

```
rodic ( eva , kain ) .
```

```
rodic ( adam , abel ) .
```

```
rodic ( eva , abel ) .
```

atomy

funktory

Procedury v Prologu

```
muZ ( adam ) .  
muZ ( kain ) .  
muZ ( abel ) .  
zena ( eva ) .  
rodic ( adam , kain ) .  
rodic ( eva , kain ) .  
rodic ( adam , abel ) .  
rodic ( eva , abel ) .
```

} procedura
definuje predikát **muZ/1**

Procedury v Prologu

```
muZ ( adam ) .  
muZ ( kain ) .  
muZ ( abel ) .
```

} procedura
definuje predikát **muZ/1**

```
zena ( eva ) .
```

zena/1

```
rodic ( adam , kain ) .  
rodic ( eva , kain ) .  
rodic ( adam , abel ) .  
rodic ( eva , abel ) .
```

Procedury v Prologu

```
muz ( adam ) .  
muz ( kain ) .  
muz ( abel ) .
```

} procedura
definuje predikát **muz/1**

```
zena ( eva ) .
```

zena/1

```
rodic ( adam , kain ) .
```

```
rodic ( eva , kain ) .
```

```
rodic ( adam , abel ) .
```

```
rodic ( eva , abel ) .
```

rodic/2

SWI Prolog



<http://swi-prolog.org/>

- 1987 - nyní
- Jan Wielemaker, Vrije Universiteit Amsterdam
- Sociaal-Wetenschappelijke Informatica
- open source (BSD)
- Windows, Unix, macOS

XPCE

- nástroj pro tvorbu GUI
- nezávislý na platformě (a na jazyce)

SWI Prolog

Editor zdrojového kódu

- **PceEmacs**: vestavěný editor SWI - Prologu
 - » klon editoru Emacs
 - » implementován v Prologu + XPCE
 - » automatické odsazování, zvýrazňování syntaxe, ...
 - » `?- emacs.`
 - » `?- edit(file('test.pl')).` % nový
 - » `?- edit('test.pl').` % existující
 - » `?- edit(test).` % .pl lze vynechat
 - » menu **File / Edit**
- **Váš oblíbený editor**
 - » lze nastavit v **Settings / User init file**
- **Visual Studio Code**
 - » rozšíření **VSC-Prolog** či **Prolog**

Práce v SWI-Prologu

Spuštění SWI-Prologu

- `?-`

Editor → soubor `rodina.pl`

Překlad

- `?- consult(rodina) .`
- `?- [rodina] .`
- `?- ['C:/Prolog/rodina.pl'] .`
- menu **File / Consult**
- `?- make .`

Výpočet → zadání dotazu

- `?- muz(adam) .`

Dotazy a odpovědi

Uživatel položí dotaz

- zadá **cíl**
- **Prolog** se pokouší **cíl** splnit
- **unifikace & backtracking**

?- muz (adam) .

?- muz (eva) .

?- muz (X) . % X je proměnná

- hledáme všechny muže
- více řešení

Výpis všech řešení

Povely při výpisu násobných řešení

- pro další řešení zadej `;`
- pro návrat zadej `.` (`enter`)
- pro plný výpis Prologem zkráceného řešení zadej `w`

```
?- rodic(X,kain) . % Kainovy rodiče
```

```
?- rodic(adam,Y) . % Adamovy děti
```

- vstup a výstup není určen předem

```
?- rodic(X,kain) , muž(X) .  
      % Kdo je Kainův otec?
```

- složený dotaz s konjunkcí `(,)`

Jednoduchý program v Prologu II

```
muz(adam). muz(kain). muz(abel).  
zena(eva).  
rodic(adam,kain). rodic(eva,kain).  
rodic(adam,abel). rodic(eva,abel).
```

hlava



tělo



```
otec(Kdo,Dite) :- rodic(Kdo,Dite),  
                  muz(Kdo).
```

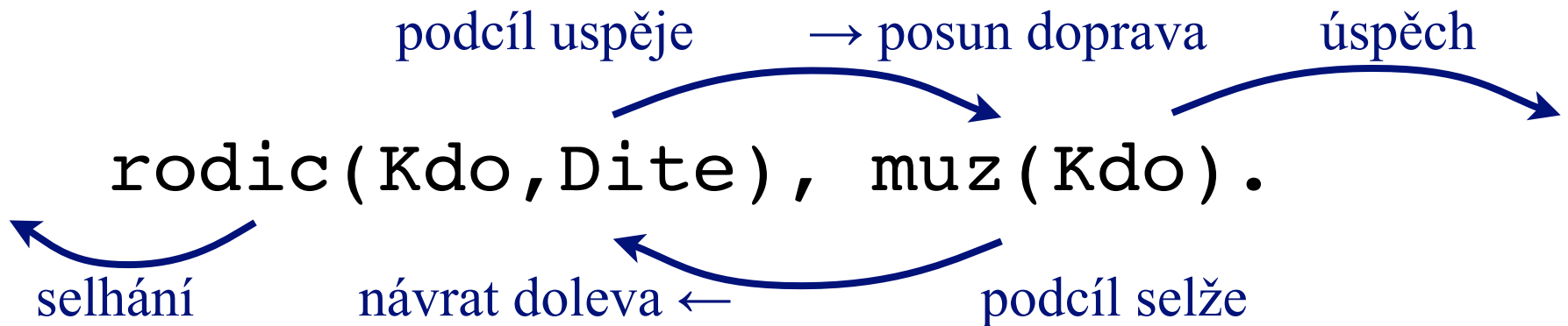
pravidlo

Vyhodnocení pravidla

? – `otec(X,kain)` . % Kdo je Kainův otec?

Jak splnit cíl, který je **hlavou** pravidla?

- je třeba splnit **tělo**
- **tělo** je konjunkcí podcílů \Rightarrow třeba splnit každý podcíl
 - » vyhodnocení podcílů zleva doprava



Procedura s více pravidly

`% clovek(C) :- C je člověk.`

`clovek(C) :- zena(C).`

`clovek(C) :- muz(C).`

`?- clovek(X).`

Poučení

- klauzule jsou při splňování cíle procházeny v pořadí, v jakém jsou zapsány

Disjunkce

```
% clovek(C) :- C je člověk.
```

```
clovek(C) :- zena(C).
```

```
clovek(C) :- muz(C).
```

Alternativní zápis

```
clovek(C) :- zena(C) ; muz(C).
```

Disjunkce

- středník **;** reprezentuje logickou spojku **nebo**
- konjunkce (**,**) “váže více” nežli disjunkce (**;**)

Další predikáty

% **bratr**(Bratr,Osoba):-

Bratr je bratrem Osoby.

bratr(Bratr,Osoba):- rodic(R,Bratr),
rodic(R,Osoba),
muz(Bratr).

☀ Je tato definice korektní?

✗ **Není!**

Potřebujeme, aby Bratr a Osoba byly různé osoby!

- cíl $\text{Bratr} \neq \text{Osoba}$ uspěje
- pokud cíl $\text{Bratr} = \text{Osoba}$ **neuspěje**

Procedura bratr/2

```
% bratr(Bratr,Osoba):-
```

```
    Bratr je bratrem Osoby.
```

```
bratr(Bratr,Osoba):-rodic(R,Bratr),  
                    rodic(R,Osoba),  
                    muz(Bratr),  
                    Bratr \= Osoba.
```

Problém

- přepište pravidlo jako formuli predikátového počtu
- a zkuste doplnit **kvantifikátory** (\forall , \exists)

Problém

Sestavte následující predikáty

% **tchyne**(Kdo, Čí) :-

Kdo je tchyní osoby Čí.

% **sestrenice**(Kdo, Čí) :-

Kdo je sestřenicí osoby Čí.

% **svagr**(Kdo, Čí) :-

Kdo je švagrem osoby Čí.

Anonymní proměnná

% **rodic**(X) :- X má dítě.

rodic(X) :- **rodic**(X,Y).

Ekvivalentní zápis

rodic(X) :- **rodic**(X, _).

- znak **_** označuje **anonymní proměnnou**
- ”na jménu této proměnné nezáleží”
- dva **různé** výskyty **_**
označují **různé** proměnné!

Problém genealogický

Vzal jsem si za ženu **vdovu**, která již měla dospělou **dceru**.

Můj **otec**, který nás často navštěvoval, se do mé (nevlastní) dcery zamiloval a oženil se s ní.

Tak se můj otec stal mým **zetěm** a má (nevlastní) dcera mojí (nevlastní) **matkou**.

Problém genealogický

O několik měsíců později má žena porodila **syna**, který se tak stal **švagrem** mého otce a současně mým **strýcem**.

Žena mého otce – tedy má (nevlastní) dcera – později také porodila **syna**, který se tak stal mým **bratrem** a současně i **vnukem** ...

Problém

- ① V jazyce Prolog popište **fakta** z příběhu.
- ② Přidejte pravidla pro definici **příbuzenských vztahů**.
- ③ Formulujte dotazy, které ověří platnost tvrzení uvedených v příběhu (“můj syn se tak stal mým strýcem” apod.).
- ④ Formulujte dotaz, který dokáže tvrzení

“Já jsem svým dědečkem”

Příklad: Einsteinova hádanka

Na kolejích je 5 pokojů, každý vyzdoben portrétem jistého velikána oboru informatika.

V každém bydlí student/ka informatiky, který/á

- tráví lockdown v jistém okrese,
- má v oblibě jistý operační systém,
- hraje jistou počítačovou hru
- a programuje v jistém jazyce.

 **Problém:** Kdo rád programuje v Prologu?

Víme, že

Obyvatel okresu Jičín má v oblibě systém Android.

Obyvatel okresu České Budějovice žije v pokoji Alonzo Churcha.

Obyvatel okresu Česká Lípa programuje v jazyce Swift.

Obyvatel okresu Louny hraje Kingdom Come: Deliverance.

Obyvatel okresu Brno bydlí v posledním pokoji.

Fanoušek systému iOS hraje Cyberpunk.

Dále víme, že

Nájemník pokoje **Alana Turinga** obdivuje **Linux**.

Fanoušek systému **macOS** programuje v jazyce **Swift**.

Nájemník pokoje **Larryho Page** hraje hru **Witcher 3**.

Fanoušek systému **macOS** bydlí **vedle** programátora v jazyce **Python**.

Programátor v **F#** bydlí **vedle** fanouška **Linuxu**.

Dále víme, že

Fanoušek **Windows** má **sousedu**, který hraje hru **Atentát**.

Pokoj **Larryho Page** je (bezprostředně) **napravo** od pokoje **Steva Jobse**.

Obyvatel okresu **Brno** bydlí **vedle** pokoje **Billa Gatese**.

Nájemník **prostředního** pokoje hraje **Skyrim**.

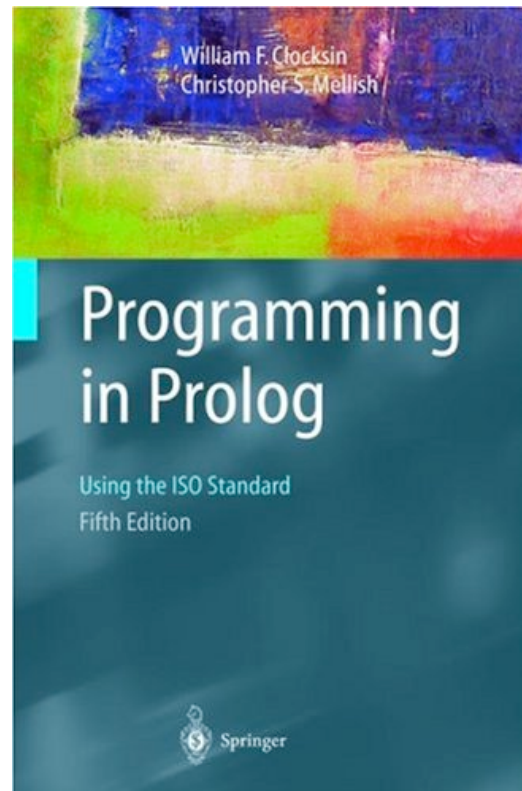
Problém

- ① Formulujte zadání hádanky jako program v jazyce Prolog.
- ② Formulujte dotaz, kterým zjistíte
kdo programuje v Prologu?






Neprocedurální programování

Prolog 2

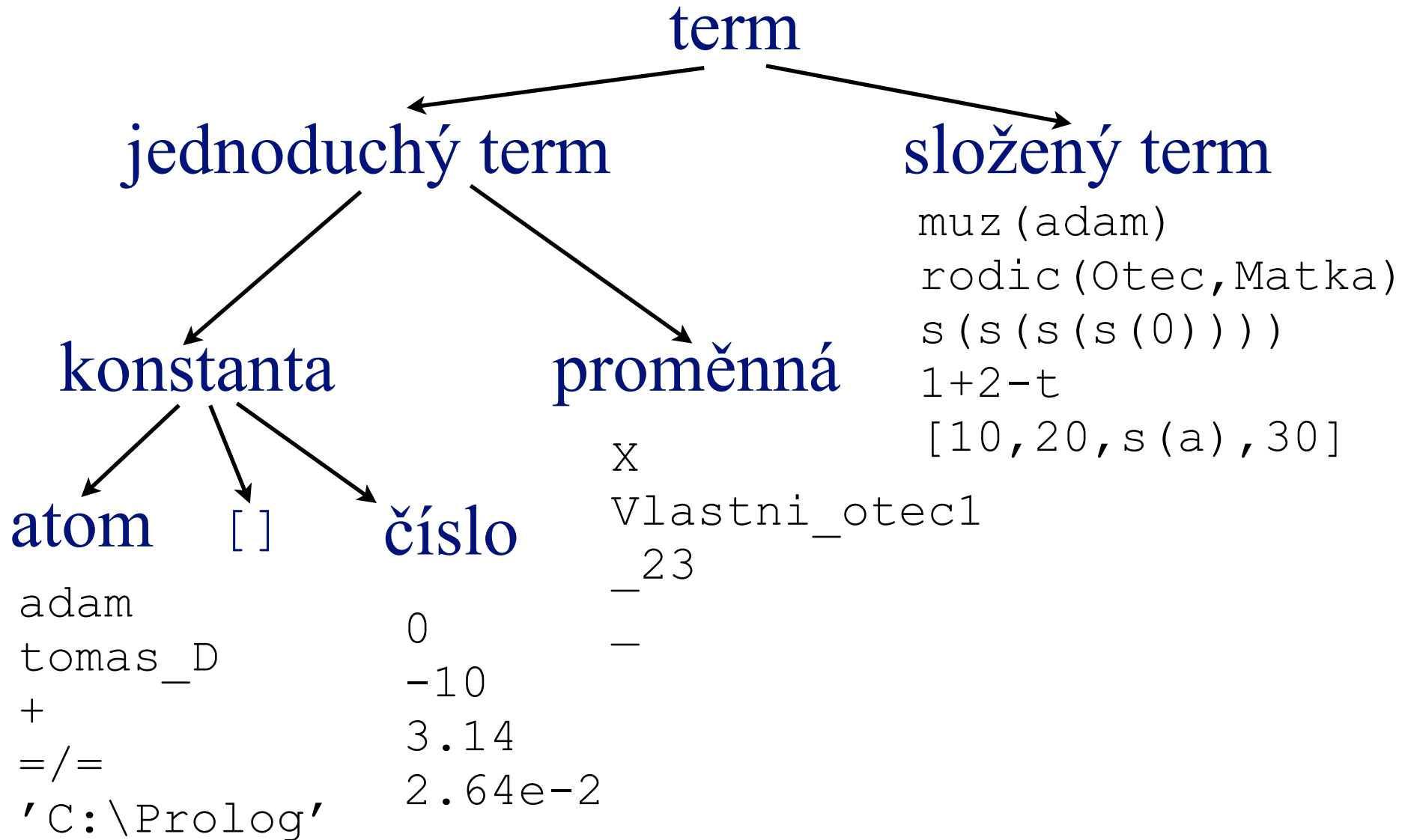
9.3.2021



Osnova

-  K syntaxi jazyka Prolog
 - Einsteinova hádanka
-  Unifikace, algoritmus splňování cíle
-  Rekurze
 - příklad aritmetický
-  Směr výpočtu
-  Seznamy

K syntaxi Prologu : Termy



Jednoduché termy

Atom

- začíná malým písmenem a obsahuje pouze písmena, číslice a `_`
 - např. `prednaska_Prolog1`
- obsahuje pouze tyto speciální znaky
 - `+ - * / \ ~ ^ < > = : . ? @ # $ &`
 - např. `?- <==> :- +`
 - kromě `/*` (začátek komentáře)
- středník `;` vykřičník `!`
- je tvořen znaky uzavřenými mezi apostrofy
 - např. `'C:\Prolog'` `'Adam'`

Jednoduché termy

Proměnná

- začíná velkým písmenem nebo _
- obsahuje pouze písmena, číslice a _

Složené termy (struktury)

Rekurzivní definice

- `funktor`($\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n$)
- n - ární `funktor` & n argumentů
- `funktor` je (syntakticky) atom
- argumenty jsou opět termy
- `funktor` je určen jménem i aritou: `rodic/1` i `rodic/2`

☀ Příklady

- `rodic(X, kain)`
- `prednaska(datum(9, 3, 2021), kod('NPRG005'))`
- `s(s(s(s(0))))`

hlavní
funktor

Tvar programu v Prologu

Program se skládá z *procedur*

Procedura $\stackrel{\text{def}}{=}$ posloupnost *klauzulí*
se stejným hlavním funktorem

Klauzule $\stackrel{\text{def}}{=}$ *pravidlo* nebo *fakt*

Pravidlo $\stackrel{\text{def}}{=}$ *hlava* :- *tělo* .

- :- $\stackrel{\text{def}}{=}$ operátor “jestliže”
- *hlava* $\stackrel{\text{def}}{=}$ term (\neq proměnná, číslo)
- *tělo* $\stackrel{\text{def}}{=}$ posloupnost termů (\neq číslo)
 - + logické spojky konjunkce (,) disjunkce (;)
 - + závorky

~~a;b :- c.~~

Tvar programu v Prologu

Fakt $\stackrel{\text{def}}{=}$ pravidlo s prázdným tělem

Direktiva $\stackrel{\text{def}}{=}$ pravidlo s prázdnou hlavou

- `:- consult(demo) .`
 - » nevypisují se hodnoty proměnných
 - » nehledají se alternativní řešení

Komentáře

- na jednom řádku: uvozené `%`
- na více řádcích: mezi `/*` a `*/`

Proměnné

Procedurální jazyky: proměnné

- jsou / mohou být deklarovány
- mohou být globální, lokální
- změna hodnoty přiřazovacím příkazem

Prolog

- dynamická alokace paměti
 - » garbage collection
- platnost proměnné je omezena na klauzuli, v níž se vyskytuje
- proměnná volná / vázaná
 - » může být vázána na hodnotu při splňování cíle
 - » neúspěch → návrat → odvolání vazby

Termy

Procedurální jazyky: datový typ záznam

- položky identifikovány jménem

Prolog: složený term

- položky identifikovány polohou
- stromová struktura

Příklad: Einsteinova hádanka

Na kolejích je 5 pokojů, každý vyzdoben portrétem jistého velikána oboru informatika.

V každém bydlí student/ka informatiky, která

- tráví lockdown v jistém okrese,
- má v oblibě jistý operační systém,
- hraje jistou počítačovou hru,
- a ráda programuje v jistém jazyce.

 **Problém:** Kdo rád programuje v Prologu?

Víme, že

Obyvatel okresu Jičín má v oblibě systém Android.

Obyvatel okresu České Budějovice žije v pokoji Alonzo Churcha.

Obyvatel okresu Česká Lípa programuje v jazyce Swift.

Obyvatel okresu Louny hraje Kingdom Come: Deliverance.

Obyvatel okresu Brno bydlí v posledním pokoji.

Fanoušek systému iOS hraje Cyberpunk.

Dále víme, že

Nájemník pokoje **Alana Turinga** obdivuje **Linux**.

Fanoušek systému **macOS** programuje v jazyce **Swift**.

Nájemník pokoje **Larryho Page** hraje hru **Witcher 3**.

Fanoušek systému **macOS** bydlí **vedle** programátora v jazyce **Python**.

Programátor v **F#** bydlí **vedle** fanouška **Linuxu**.

Dále víme, že

Fanoušek **Windows** má **souseda**, který hraje hru **Atentát**.

Pokoj **Larryho Page** je (bezprostředně) **napravo** od pokoje **Steva Jobse**.

Obyvatel okresu **Brno** bydlí **vedle** pokoje **Billa Gatese**.

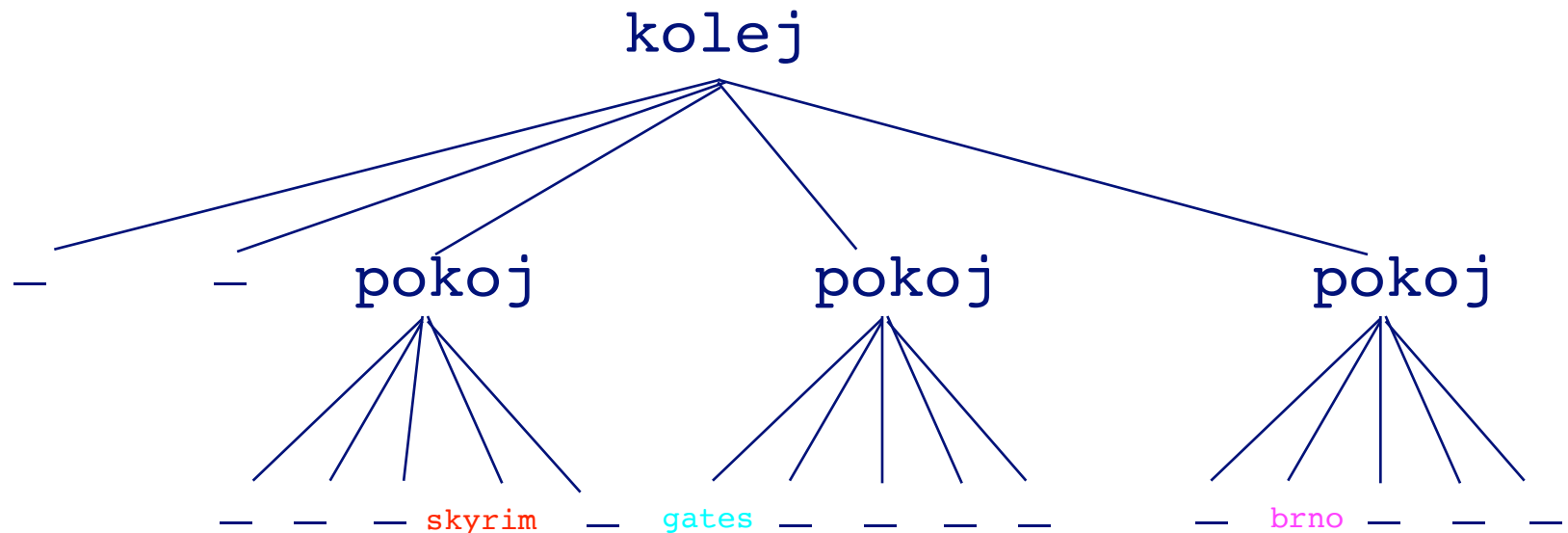
Nájemník **prostředního** pokoje hraje **Skyrim**.

Problém

- ① Formulujte zadání hádanky jako program v jazyce Prolog.
- ② Formulujte dotaz, kterým zjistíte
kdo programuje v Prologu?

Einsteinova hádanka: datová struktura

```
kolej(_,  
_/  
pokoj(_,_/_/_,skyrim,_),  
pokoj(gates,_/_/_/_),  
pokoj(_,_brno,_/_/_))
```



Operátory

Procedurální jazyky: $a+b*c-1$

- výrazy s operátory v infixové notaci

Prolog: $\text{adam} \backslash = \text{eva}$

- syntaktické pozlátka pro $\backslash = (\text{adam}, \text{eva})$
- $\backslash =$ je binární funktor
 - » definovaný jako operátor
 - » lze použít infixovou notaci
- $\text{display}/1$
 - » standardní predikát, vypíše term v kanonickém tvaru
- $?- \text{display}(a+b*c-1)$
- $-(+(a, *(b, c)), 1)$

Unifikace

Základní operace na termech

Dva termy lze unifikovat, pokud

- jsou identické, nebo
- se stanou identickými po substituci vhodné hodnoty proměnným v obou termech

☀ Příklad

- $\text{datum}(D1, M1, 2021) = \text{datum}(D2, \text{brezen}, R2)$
 - » např. $D1 = D2 = 1, M1 = \text{brezen}, R2 = 2021$
- nejobecnější unifikace
 - » $D1 = D2$
 - » $M1 = \text{brezen}$
 - » $R2 = 2020$

Unifikační algoritmus

Unifikaci vyvolá operátor =

- $term1 = term2$ uspěje, pokud oba termy lze unifikovat
- jinak selže

Při úspěchu provede nejobecnější unifikaci

Výsledkem úspěšné unifikace je substituce hodnot za proměnné

Poznámka: $term1 \neq term2$

- uspěje, pokud oba termy **nelze** unifikovat

Termy S a T lze unifikovat, pokud

S a T jsou identické konstanty

S a T jsou proměnné

- výsledkem unifikace je jejich ztotožnění

S je proměnná, T je term různý od proměnné

- výsledkem je substituce termu T za proměnnou S

T je proměnná, S je term různý od proměnné

- výsledkem je substituce termu S za proměnnou T

S a T jsou složené termy, které

- oba mají stejný hlavní funktor a
- odpovídající argumenty lze unifikovat

V ostatních případech S a T unifikovat nelze.

Unifikace: Příklad

?- $f(X, a(b, c)) = f(d, a(Y, c))$.

$X = d, Y = b$.

?- $f(X, a(b, c)) = f(Y, a(Y, c))$.

$X = Y = b$.

?- $f(c, a(b, c)) = f(Y, a(Y, c))$.

false.

?- $X = f(X)$.

Algoritmus splňování cíle

Unifikační algoritmus + backtracking

- průchod do hloubky s návratem při neúspěchu

Na pořadí záleží

- klauzule i termy jsou zpracovány v pořadí, v němž jsou v programu zapsány
- chronologický backtracking

Platnost proměnných

- platnost proměnné omezena na pravidlo, v němž se vyskytuje
- před použitím pravidla jsou v něm vždy všechny proměnné přejmenovány

Algoritmus splňování cíle

```
def splňování_cíle(program, seznam_cílů) :  
    """ program : seznam klauzulí  
        seznam_cílů : obsahuje cíle, které chceme splnit  
        vrátí : úspěch / neúspěch  
    """  
  
    if seznam_cílů je prázdný :  
        return True  
  
    else :  
        cíl = hlava(seznam_cílů) % první cíl v seznamu  
        další = tělo(seznam_cílů) % seznam ostatních cílů  
        splněno = False  
  
        while not splněno and program obsahuje další klauzuli :  
            nechť další klauzule je tvaru  $H :- T_1, \dots, T_n$ .  
            přejmenuj všechny proměnné v této klauzuli  
            if cíl lze unifikovat s termem  $H$   
                a výsledkem je substituce  $S$  :  
                nové_cíle = zřetězení  $T_1, \dots, T_n$  se seznamem další  
                ve všech prvcích seznamu nové_cíle proved' substituci  $S$   
                splněno = splňování_cíle(program, nové_cíle)  
  
        return splněno
```

☀ Příklad aritmetický

```
% prirozene_cislo(X) :- X je přirozené číslo.  
prirozene_cislo(0).  
prirozene_cislo(s(X)) :- prirozene_cislo(X).  
  
% mensi(X,Y):- X je ostře menší než Y.  
mensi(0,s(X)) :- prirozene_cislo(X).  
mensi(s(X),s(Y)) :- mensi(X,Y).
```

Rekurze

Strukturální rekurze

- řízena strukturou argumentů
- rekurzivní datová struktura
 - ☞ rekurzivní procedura, která s ní pracuje
- báze
 - » fakt nebo nerekurzivní pravidlo
- krok rekurze
 - » rekurzivní pravidlo

```
soucet(0,X,X) :- prirodzene_cislo(X). % báze
soucet(s(X),Y,s(Z)) :- soucet(X,Y,Z).
                             % rekurzivní krok
```

Směr výpočtu

```
mensi(0,s(X)) :- prirodzene_cislo(X).  
mensi(s(X),s(Y)) :- mensi(X,Y).
```

☞ Které argumenty jsou **vstupní** a které **výstupní**?

- syntaxe to nespecifikuje
- některé argumenty mohou hrát roli vstupu i výstupu
 - » relace “=” je symetrická
- ?- **mensi(s(0),s(s(0)))** .
- ?- **mensi(X,s(s(0)))** .
- ?- **mensi(s(0),Y)** .
- ?- **mensi(X,Y)** .

Specifikace vstupu a výstupu

Konvence pro specifikaci V/V v komentáři

- $+$ argument je **vstupní**
 - » při dotazu musí být konkretizován
 - » základní term = term bez volných proměnných
- $-$ argument je **výstupní**
 - » při dotazu nesmí být konkretizován
 - » volná proměnná
- $?$ argument může být **vstupní i výstupní**
- $+-$ argument obsahuje **volné proměnné**

☀ Příklad (aritmetický)

```
% soucet(+X,+Y,?Z) :- Z=X+Y.
```

```
soucet(0,X,X) :- prirodzene_cislo(X).
```

```
soucet(s(X),Y,s(Z)) :- soucet(X,Y,Z).
```

Dotazy

- ?- soucet(s(0),s(0),s(s(0))).
- ?- soucet(s(0),s(0),Z).

Co lze říci o dotazech typu

- % soucet(+X,-Y,+Z)
- % soucet(-X,+Y,+Z)

Směr výpočtu : závěr

Poučení

- predikáty v Prologu jsou často invertibilní
 - » vstup \leftrightarrow výstup \Rightarrow obrácení “směru výpočtu”
- ne vždy jsou oba směry stejně efektivní
- samostatný predikát pro každý směr

Seznamy

[] prázdný seznam

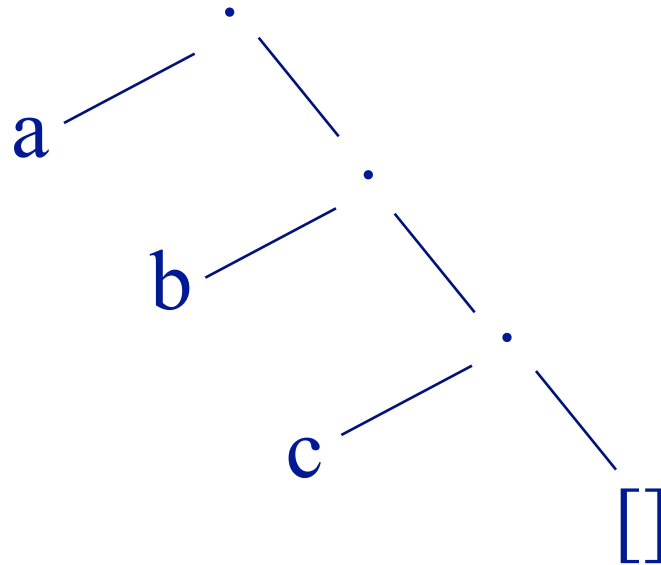
[a,b,c] příklad neprázdného seznamu

Neprázdný seznam

- tečka-dvojice *.(Hlava, Tělo)*
- *Hlava* - první prvek seznamu
- *Tělo* - seznam tvořený zbylými prvky
- navrženo pro jazyk LISP

Seznamy: Příklad

$[a,b,c] = .(a, .(b, .(c, []))) =$



Seznamy: notace

SWI-Prolog verze 6

- `?- display([a,b,c])`
- `.(a, .(b, .(c, [])))`

SWI-Prolog od verze 7

- `?- display([a,b,c])`
- `'[]'(a,'[]'(b,'[]'(c,[])))`
- `./2` má jiné využití

V jazyce Prolog pro oddělení hlavy a těla seznamu slouží operátor |

Seznamy: operátor |

.(Hlava, Telo) se v jazyce Prolog zapíše jako
[Hlava | Telo]

Operátor | má dokonce ještě obecnější význam

- umožňuje oddělit nejen hlavu
- ale i začátek seznamu
- *[Začátek | Tělo]*
- *Začátek* je **výčet** prvků na začátku seznamu oddělených **čárkami**
- *Tělo* je **seznam** zbývajících prvků seznamu

$$[a,b,c] = [a \mid [b,c]] = [a,b \mid [c]] = [a,b,c \mid []]$$

Seznamy: predikát prvek/2

```
% prvek(X,Seznam):- X je prvkem  
                        Seznamu.
```

```
prvek(X, [X|_]).
```

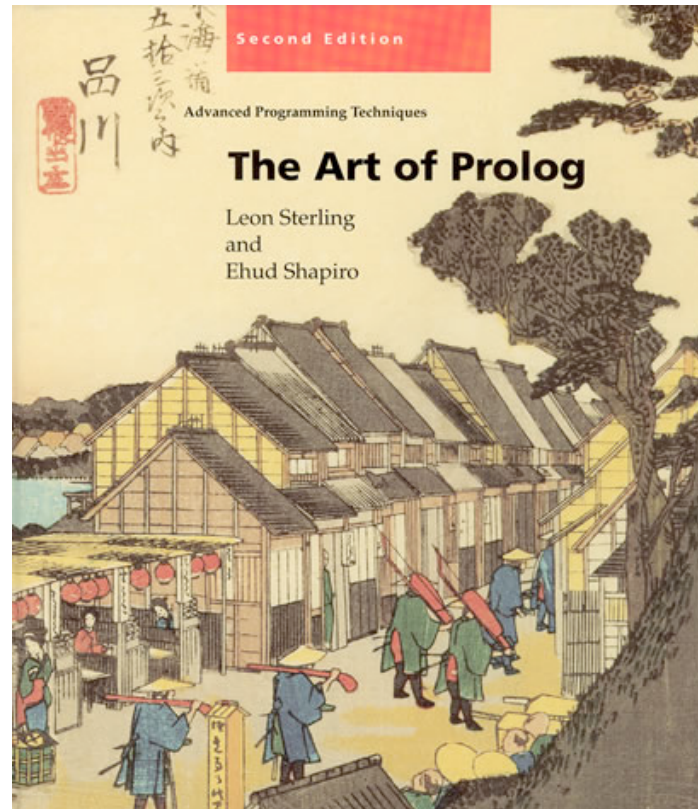
```
prvek(X, [_|T]) :- prvek(X,T).
```

- ?- prvek(a, [a,b,c]).
- ?- prvek(X, [a,b,c]).
- ?- prvek(a, S).
- předdefinován jako standardní predikát
member/2






Neprocedurální programování

Prolog 3

16.3.2021



Co bylo minule

-  K syntaxi jazyka Prolog
-  Unifikace, algoritmus splňování cíle
-  Rekurze
-  Směr výpočtu
 - příklad aritmetický
-  Seznamy

☀ Příklad (genealogický)

```
predek(X,Y) :- rodic(X,Y).
```

```
predek(X,Z) :- rodic(X,Y), predek(Y,Z).
```

Jak se vyhodnocují dotazy

- ?- predek(premysl_orac, karel_ctvrty).
- ?- predek(premysl_orac, Y).
- ?- predek(X, karel_ctvrty).
- % predek(+Predek, ?Potomek)
- vhodnější název by byl potomek/2

```
% predek1(?Predek, +Potomek)
```

```
predek1(X,Y) :- rodic(X,Y).
```

```
predek1(X,Z) :- rodic(Y,Z), predek1(X,Y).
```

Osnova

Seznamy

- technika akumulátoru
- koncová rekurze

Nedeterminismus

Aritmetika

Seznamy: predikát member/2

```
% member(?X, ?Xs) :- X je prvkem  
                        seznamu Xs.
```

```
member(X, [X|_]).
```

```
member(X, [_|Xs]) :- member(X, Xs).
```

- ?- member(a, [a,b,c]).
- ?- member(X, [a,b,c]).
- ?- member(a, Xs).
- ?- member(X, Xs).

Vyhledávání v asociativním seznamu

Seznam položek s klíčem a hodnotou, např.

- $p(\text{kod}, \text{ucitel})$
- kod - ucitel

?- ASeznam = [p(nprg031,holan), p(nprg031,pergel),
p(nprg005,hric), p(nprg005,dvorak), p(nmin112,topfer)],
member(p(nprg005,Ucitel), ASeznam).

Ucitel = hric ;

Ucitel = dvorak

?- ASeznam = [nprg031-holan, nprg031-pergel, nprg005-
hric, nprg005-dvorak, nmin112-topfer],
member(nprg005-Ucitel, ASeznam).

Vypuštění prvku ze seznamu

```
% vypust(+X,+Xs,?Ys):- Seznam Ys vznikne  
%      vypuštěním prvku X ze seznamu Xs.
```

```
vypust(X,[X|Xs],Xs).
```

```
vypust(X,[Y|Ys],[Y|Zs]):-vypust(X,Ys,Zs).
```

- ?- vypust(a, [a,b,a,c,a],V).
 - » vypustí vždy jen jeden výskyt
 - » postupně vrátí všechny možnosti
- ?- vypust(x, [a,b,a,c,a], V).
 - » není-li vypouštěný prvek v zadaném seznamu, selže
- předdefinován jako **select/3**

Problém: další varianty vypouštění

Definujte následující predikáty

- `vypust1(X,Xs,Ys):-` varianta `vypust/3`, která vždy uspěje
- `vypustvse(X,Xs,Ys):-` Seznam `Ys` vznikne vypuštěním všech výskytů prvku `X` ze seznamu `Xs`, vždy uspěje
» předdefinován jako `delete(Xs,X,Ys)`
- `vypustvsel(X,Xs,Ys):-` varianta `vypustvse/3`, není-li `X` prvkem `Xs`, selže

Vložení prvku do seznamu

```
vypust (X, [X | Xs], Xs) .
```

```
vypust (X, [Y | Ys], [Y | Zs]) :-
```

```
    vypust (X, Ys, Zs) .
```

- co lze říci o dotazu
- ?- vypust(z, Xs, [a,b,c]) .
 - » vloží prvek do seznamu
 - » postupně na všechna možná místa

```
% vloz(+X,+Xs,?Ys):- Seznam Ys vznikne  
%      vložením prvku X do seznamu Xs.
```

```
vloz(X,Xs,Ys) :- vypust(X,Ys,Xs) .
```

👉 vestavěný **select/3** lze použít i pro vkládání

První a poslední prvek

První prvek

- hlava seznamu
- přístupný přímo pomocí |

Poslední prvek

```
% posledni(+Xs,?X):- X je posledním  
                        prvkem seznamu Xs.
```

```
posledni( [X],X) .
```

```
posledni( [_|Xs],Y) :- posledni(Xs,Y) .
```

- předdefinován jako standardní predikát **last/2**

Prostřední prvek

① Naivní řešení

- odstran první a poslední prvek
- ve zbytku najdi prostřední prvek rekurzivně
- báze pro 1 a 2prvkové seznamy
- kvadratická časová složitost

② Řešení s aritmetikou

- spočítej délku seznamu n
- vrať prvek na pozici $\lceil (n+1)/2 \rceil$ nebo $\lfloor (n+1)/2 \rfloor$

③ Elegantní řešení

- pomocí 2 signálů
- když rychlý dorazí na konec, pomalý je uprostřed

Prostřední prvek pomocí 2 signálů

```
% prostredni(+Xs,?X):-  
%   X je prostředním prvkem seznamu Xs.  
prostredni(S,X) :- prostredni(S,S,X).  
prostredni([_],[X|_],X).  
prostredni([_,_],[X|_],X).  
prostredni([_,_|Xs],[_|Ys],X) :-  
    prostredni(Xs,Ys,X).
```

Časová složitost **lineární**

Problém

- který prvek bude vrácen ze seznamu sudé délky?

Zřetězení seznamů

```
% zretez(?Xs,?Ys,?Zs):- Zs je zřetězením  
                        seznamů Xs a Ys.
```

```
zretez([ ],Ys,Ys).
```

```
zretez([X|Xs],Ys,[X|Zs]):- zretez(Xs,Ys,Zs).
```

Dotazy

- ?- zretez([a,b,c], [d,e], [a,b,c,d,e]).
- ?- zretez([a,b,c], [d,e], S).
- ?- zretez(Xs, Ys, [a,b,c,d,e]).

Předdefinován jako standardní predikát

append/3

Využití predikátu zřetězení

`prvek(X,Ys) :- append(_, [X|_], Ys).`

`posledni(X,Ys) :- append(_, [X], Ys).`

 **Problém:** Využijte `append/3` k definici následujících predikátů

- `prefix(?Xs,+Ys) :-` Xs je předponou seznamu Ys
- `sufix(?Xs,+Ys) :-` Xs je příponou seznamu Ys
- `faktor(?Xs,+Ys) :-` Xs je (souvislý) podseznam seznamu Ys

Otočení seznamu

① Naivní řešení

```
% otoc(+Xs,-Ys):- Ys je otočením  
% seznamu Xs.
```

```
otoc([],[]).
```

```
otoc([X|Xs],Zs) :- otoc(Xs,Ys),  
                  append(Ys,[X],Zs).
```

- kvadratická časová složitost

② Otočení v lineárním čase

- řešíme obecnější problém
- technika akumulátoru

Otočení seznamu v lineárním čase

```
otocAk(Xs, Ys) :- otocAk(Xs, [], Ys).
```

```
% otocAk(+Xs, +A, -Ys) :- Ys je zřetězením  
%      otočeného seznamu Xs se seznamem A.
```

```
otocAk([], A, A).
```

```
otocAk([X|Xs], A, Ys) :- otocAk(Xs, [X|A], Ys).
```

Vlastnosti řešení

- řešíme obecnější problém
- technika akumulátoru
- lineární čas

Otočení seznamu v lineárním čase

```
otocAk(Xs,Ys) :- otocAk(Xs,[],Ys).  
% otocAk(+Xs,+A,-Ys):- Ys je zřetězením  
%      otočeného seznamu Xs se seznamem A.  
otocAk([],A,A).  
otocAk([X|Xs],A,Ys):-otocAk(Xs,[X|A],Ys).
```

Problém

- jaká bude odpověď na dotaz `otocAk(-Xs,+Ys)`?

Předdefinován jako standardní predikát `reverse/2`

Koncová rekurze

$p(\dots) :- \underbrace{\dots}, p(\dots).$

zde se p nevyskytuje
pouze deterministické cíle
(žádný backtracking)

procedura p
je deterministická

Koncová rekurze

- návrat z každého rekurzivního volání je triviální
 - » úspora paměti
 - ✓ konstatní prostor na zásobníku
 - » rychlost
- rekurzi lze nahradit iterací
 - » překladač provádí automaticky
 - » Tail Recursion Optimization / Last Call Optimization

Příklad koncové rekurze

```
% otoc(+Xs,-Ys):- Ys je otočením  
%                  seznamu Xs.  
  
otoc([],[]).  
  
otoc([X|Xs],Zs) :- otoc(Xs,Ys),  
                   append(Ys,[X],Zs).
```

Procedura `otoc/2` **není** koncově rekurzivní

☀ Příklad koncové rekurze

```
otocAk(Xs, Ys) :- otocAk(Xs, [], Ys).
```

```
otocAk([], A, A).
```

```
otocAk([X|Xs], A, Ys) :-
```

```
    otocAk(Xs, [X|A], Ys).
```

Procedura `otocAk/2` je **koncově rekurzivní**

- pouze konstatní prostor na zásobníku
- náhrada rekurze iterací

Permutace

```
% permutace(+Xs,-Ys) :- Seznam Ys je  
%                      permutací seznamu Xs.  
permutace([], []).  
permutace(Xs, [X|Zs]) :- select(X, Xs, Ys),  
                           permutace(Ys, Zs).
```

☀ Idea

- **nedeterministický** výběr prvního prvku permutace
- permutace zbylých prvků rekurzivně

Permutace: alternativní řešení

```
permutace2([ ], [ ]).
```

```
permutace2([X|Xs], Zs) :-  
    permutace2(Xs, Ys), select(X, Zs, Ys).
```

☀ Idea

- oddělení hlavy vstupního seznamu
- permutace těla vstupního seznamu
- **nedeterministické** vložení hlavy

Standardní predikát **permutation(?Xs, ?Ys)**

☀ Axiomatizace přirozených čísel

```
% prirozene_cislo(?X) :- X je přirozené číslo.  
prirozene_cislo(0).  
prirozene_cislo(s(X)) :- prirozene_cislo(X).  
  
% mensi(+X,+Y) :- X je ostře menší než Y.  
mensi(0,s(X)) :- prirozene_cislo(X).  
mensi(s(X),s(Y)) :- mensi(X,Y).
```

Procedura `mensi/2` je **deterministická**

- cíl lze unifikovat s hlavou nejvýše jednoho pravidla
- (dotaz `mensi(-X,+Y)` je ovšem nedeterministický)

Nedeterminismus

```
% mensi(+X,+Y):- X je ostře menší než Y.  
mensi(0,s(X)) :- prirodzene_cislo(X).  
mensi(s(X),s(Y)) :- mensi(X,Y).
```

```
% alternativní definice  
mensi2(X,s(X)) :- prirodzene_cislo(X).  
mensi2(X,s(Y)) :- mensi2(X,Y).
```

Procedura `mensi2/2` je **nedeterministická**

- `?- mensi2(s(0),s(s(0)))`.
- unifikace s hlavou obou pravidel

Nedeterminismus vs. determinismus

Deterministická procedura

- při neúspěchu není nutný návrat
 - » backtracking je triviální
- snadnější ladění

Nedeterminismus

- mocný nástroj
- někdy ho potřebujeme
 - » viz generování permutací

Aritmetika

?- $X = 1+1$.

?- X is $1+1$.

Vestavěný predikát $\text{is}/2$

- definovaný jako operátor
- ?- $\text{display}(X \text{ is } 1+1)$.
- $\text{is}(X, +(1, 1))$

Operátor is/2

S is T

- term **T** je vázán na aritmetický výraz
 - » hodnota **T** je vyhodnocena jako aritmetický výraz
 - » výsledek je unifikován s termem **S**
- term **T** **není** vázán na aritmetický výraz \Rightarrow **chyba**

Vestavěný systémový predikát

- nepatří do čistého Prologu

Aritmetické operátory

$+$, $-$, $*$, $/$, $//$, $^$, mod

- ?- $X \text{ is } 2^3 \text{ mod } 5$.

Relační: $>/2$, $</2$, $>=/2$, $=</2$

Rovnost a nerovnost: $:=$, $=\backslash$

- vyhodnocení operandů, porovnání výsledků
- operand **není** vázán na aritmetickou hodnotu
 \Rightarrow **chyba**
- ?- $1+2 := 2+1$.

Aritmetické operátory

- jsou deterministické
- “nebacktrackují”

Aritmetické funkce

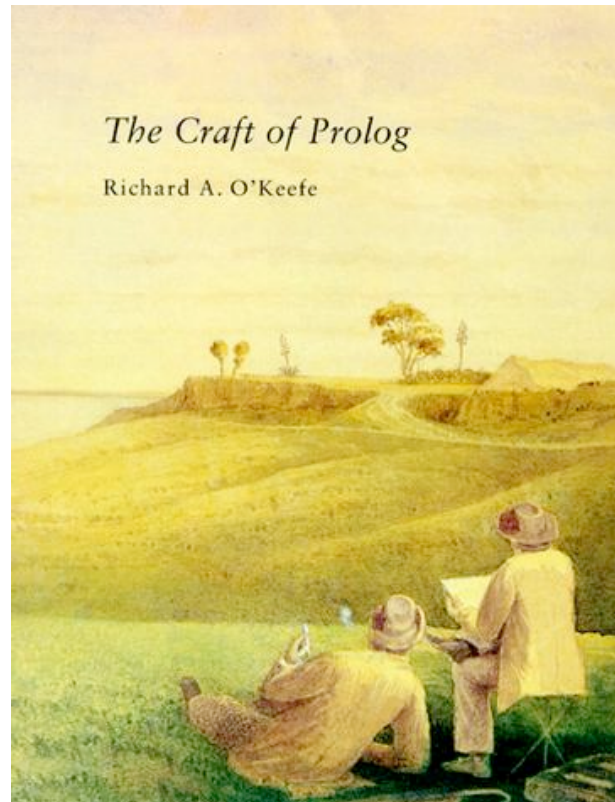
Lze použít v aritmetických výrazech

- $\max/2$, $\min/2$, $\text{abs}/1$...
- $\sin/1$, $\cos/1$, $\tan/1$, $\text{sqrt}/1$, $\log/1$...
» ?- X is $4*\text{asin}(\text{sqrt}(2)/2)$.
- bitové operace: $\wedge/2$, $\vee/2$, $\text{xor}/2$, $\gg/2$, $\ll/2$...
- vrací aritmetickou (nikoliv logickou) hodnotu

Neprocedurální programování

Prolog 4

22.3.2021



Co bylo minule

Směr výpočtu

- příklad: predikát `predek/2`





Seznamy

- technika akumulátoru
- koncová rekurze

Nedeterminismus

Aritmetika

Osnova

-  Deklarativní a procedurální správnost
-  Aritmetika
 - příklad: třídění
-  Stromy v Prologu
 - binární vyhledávací stromy
-  Řez a negace

Deklarativní význam programu

Pravidla

- $p :- q, r.$ $p :- q; r.$

mají význam formulí

- $q \wedge r \Rightarrow p$ $q \vee r \Rightarrow p$

Deklarativní význam programu

- množina formulí, které určují význam klauzulí programu
- nezávisí na pořadí klauzulí programu
- nezávisí na pořadí termů v těle pravidel
 - » \wedge a \vee jsou komutativní

Procedurální význam programu

Pravidlo

- $p :- q, r.$

lze interpretovat i takto

- pro splnění cíle p je třeba nejprve splnit podcíl q a potom podcíl r

Procedurální význam

- = procedura splňování cíle vzhledem k danému programu
- závisí na pořadí klauzulí programu i termů v těle pravidel

Deklarativní / procedurální správnost programu

Program může být

- **správný deklarativně**
 - » odpověď na dotaz existuje
- **leč nesprávný procedurálně**
 - » odpověď nelze nalézt procedurou splňování cíle
 - » popsaná procedura splňování cíle není úplná

Je-li program **správný deklarativně**

- nemůže dát chybný výsledek
- nemusí však dát vůbec žádný výsledek
 - » je-li procedurálně nesprávný, může dojít k zacyklení

☀ Příklad genealogický: předek/2

```
% predek(+Predek,?Potomek) :- Predek je  
%                             předkem Potomka.
```

```
predek(X,Y) :- rodic(X,Y).
```

```
predek(X,Z) :- rodic(X,Y), predek(Y,Z).
```

predek/2 je deklarativně i procedurálně správný

```
% predek2(+Predek,?Potomek) :- jiná varianta  
%                             předka.
```

```
predek2(X,Z) :- predek2(Y,Z), rodic(X,Y).
```

```
predek2(X,Y) :- rodic(X,Y).
```

predek2/2 se zacyklí

☀ Příklad genealogický: předek/2

```
predek3(X,Z) :- rodic(X,Y), predek3(Y,Z).  
predek3(X,Y) :- rodic(X,Y).
```

predek3/2 je deklarativně i procedurálně správný

```
predek4(X,Y) :- rodic(X,Y).  
predek4(X,Z) :- predek4(Y,Z), rodic(X,Y).
```

predek4/2 najde všechna řešení, pak se zacyklí

Jednoduché aritmetické predikáty

% **max**(+X,+Y,?Max) :- Max je maximum
z čísel X a Y.

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

max(X,Y,Y) .

Jaké budou odpovědi na následující dotazy?

- ?- **max**(2,1,M).

- ?- **max**(2,1,1).

» true.

chyba !!!

Korektní verze

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

max(X,Y,Y) :- X < Y.

Jednoduché aritmetické predikáty

```
% mezi(+X,+Y,-Z):- Postupně vrátí  
% celá čísla splňující  $X \leq Z \leq Y$ .
```

```
mezi(X,Y,X) :- X =< Y.
```

```
mezi(X,Y,Z) :- X<Y, NoveX is X+1,  
                mezi(NoveX,Y,Z).
```

```
?- mezi(1,3,Z).
```

- $Z = 1$;
- $Z = 2$;
- $Z = 3$.

Předdefinován jako standardní predikát
between/3

Délka seznamu

```
% delka(+Xs,?N) :- N je počet prvků  
% seznamu Xs.
```

```
delka([ ],0).
```

```
delka([_|Xs],N) :- delka(Xs,N1),  
N is N1+1.
```

☞ Alternativní řešení

- koncová rekurze
- akumulátor

Délka seznamu s akumulátorem

```
delkaAk(Xs,N) :- delkaAk(Xs,0,N).
```

```
delkaAk([],A,A).
```

```
delkaAk([_|Xs],A,N) :- A1 is A+1,  
                        delkaAk(Xs,A1,N).
```

 **Problém:** Jaké odpovědi obdržíme na dotazy

- ?- delkaAk(S,3).
- ?- delkaAk(S,N).

Předdefinován jako standardní predikát
`length/2`

Třídění sléváním: Mergesort

```
% mergesort(+Xs,-Ys):- Ys je vzestupně  
%                      setříděný seznam Xs.
```

```
mergesort([],[]).
```

```
mergesort([X],[X]).
```

```
mergesort(Xs,Ys) :- Xs = [_,_|_],  
                    msplit(Xs,Xs1,Xs2),  
                    mergesort(Xs1,Ys1),  
                    mergesort(Xs2,Ys2),  
                    merge(Ys1,Ys2,Ys).
```


Třídění sléváním: rozdělení

```
% msplit(+Xs,-Ys,-Zs):- Ys a Zs
% jsou seznamy lichých a sudých prvků
% seznamu Xs.

msplit([],[],[]).
msplit([X],[X],[]).
msplit([X,Y|Zs],[X|Xs],[Y|Ys]) :-
    msplit(Zs,Xs,Ys).
```

Třídění sléváním: slévání

```
% merge(+Xs,+Ys,-Zs) :- sloučí  
% uspořádané seznamy Xs a Ys do  
% uspořádaného seznamu Zs.
```

```
merge([ ],Ys,Ys).
```

```
merge([X|Xs],[ ],[X|Xs]).
```

```
merge([X|Xs],[Y|Ys],[X|Zs]) :- X <= Y,  
                                merge(Xs,[Y|Ys],Zs).
```

```
merge([X|Xs],[Y|Ys],[Y|Zs]) :- X > Y,  
                                merge([X|Xs],Ys,Zs).
```

Třídění: Quicksort

```
% quicksort(+Xs,-Ys):- Ys je vzestupně  
%                     setříděný seznam Xs.  
quicksort([],[]).  
quicksort([X|Xs],S) :-  
    qsplit(X,Xs,Ys,Zs),  
    quicksort(Ys,YsS),  
    quicksort(Zs,ZsS),  
    append(YsS,[X|ZsS],S).
```

Quicksort: rozdělení

```
% qsplit(+Pivot,+Xs,-Ys,-Zs):- Ys a Zs  
% jsou seznamy prvků  $\leq$ Pivot a  $>$ Pivot  
% seznamu Xs.
```

```
qsplit(_,[],[],[]).
```

```
qsplit(P,[X|Xs],[X|Ys],Zs):- X  $\leq$  P,  
                             qsplit(P,Xs,Ys,Zs).
```

```
qsplit(P,[X|Xs],Ys,[X|Zs]):- X  $>$  P,  
                             qsplit(P,Xs,Ys,Zs).
```

Třídění termů

Standardní predikát `sort/2`

- `sort(+Seznam, -UsporadanySeznam)`
- `Seznam` je seznam libovolných termů
- setřídí a odstraní duplicity

Standardní pořadí termů

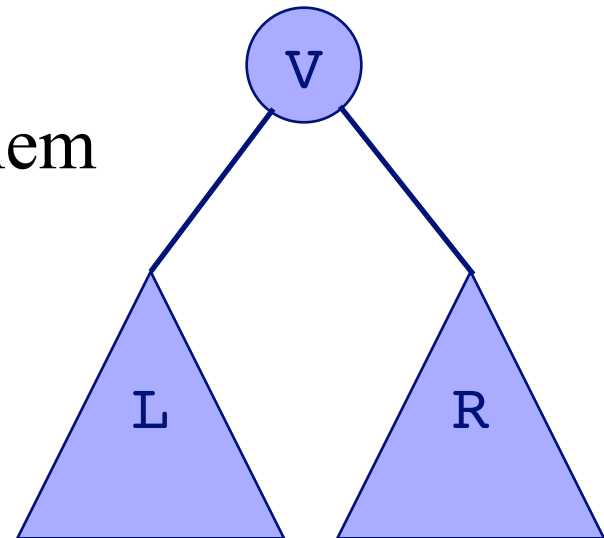
- `proměnné` < `čísla` < `atomy` < `složené termy`
- `proměnné`: dle adresy
- `atomy`: lexikograficky
- `složené termy`: arita, funktor, argumenty zleva doprava
- standardní operátory
 - » `@</2`, `@>/2`, `@=</2`, `@>=/2`

Binární stromy

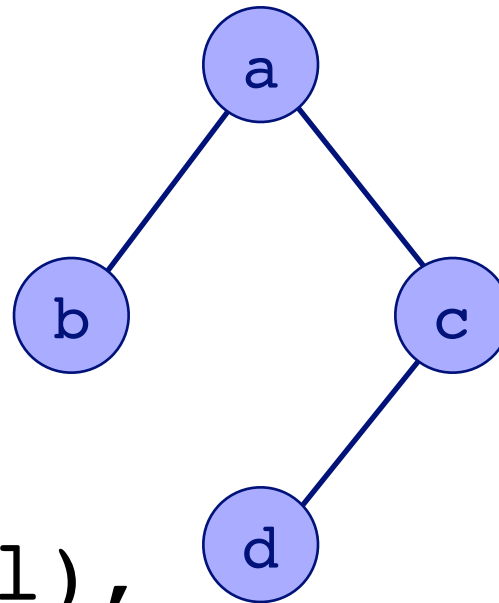
Prázdný strom: atom `nil`

Neprázdný strom: složený term $t(L, V, R)$

- `L` je levý podstrom
- `V` je vrchol, který je kořenem
- `R` je pravý podstrom

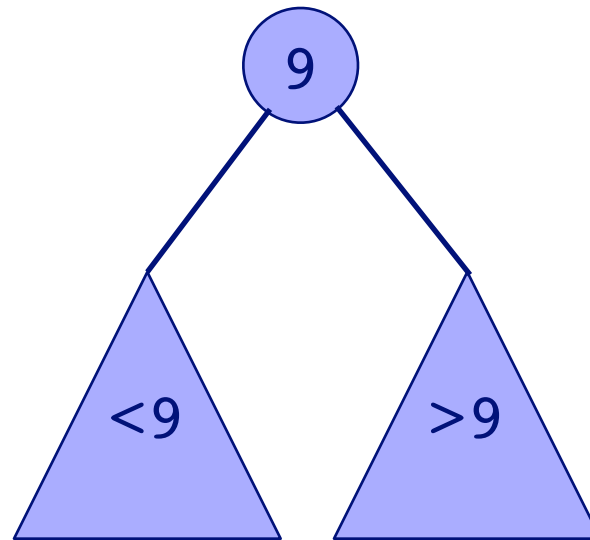


☀ Příklad binárního stromu

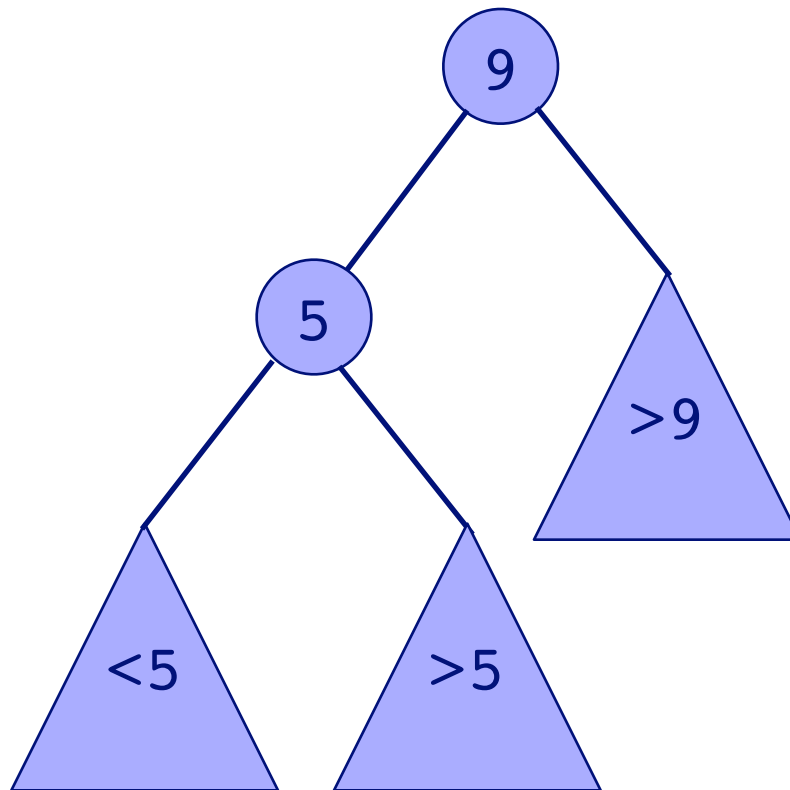


```
t(t(nil, b, nil),  
  a,  
  t(t(nil, d, nil), c, nil)  
)
```

Binární vyhledávací stromy



Binární vyhledávací stromy



Binární vyhledávací stromy: prvek

```
% in(+X,+B) :- X je prvkem binárního  
%               vyhledávacího stromu B.  
in(X, t(_,X,_)).  
in(X, t(L,K,_)) :- X<K, in(X,L).  
in(X, t(_,K,R)) :- X>K, in(X,R).
```

Binární vyhledávací stromy: vlož

```
% add(+X,+B,-B1) :- B1 vznikne vložním  
% X do binárního vyhledávacího stromu B.
```

```
add(X,nil,t(nil,X,nil)).
```

```
add(X,t(L,X,R),t(L,X,R)). % bez duplicit
```

```
add(X,t(L,V,R),t(L1,V,R)) :- X<V,  
                                add(X,L,L1).
```

```
add(X,t(L,V,R),t(L,V,R1)) :- X>V,  
                                add(X,R,R1).
```

Problém

- lze proceduru využít též k vypuštění prvku **X** z **B**
- dotazem typu **add(+X,-B1,+B)** ?

Binární vyhledávací stromy: vypust'

```
% del(+X,+B,-B1) :- B1 vznikne  
%                vypuštěním X z BVS B.  
del(X,t(nil,X,R),R).  
del(X,t(L,V,R),t(L1,V,R)) :- X<V,  
                                del(X,L,L1).  
del(X,t(L,V,R),t(L,V,R1)) :- X>V,  
                                del(X,R,R1).  
del(X,t(L,X,R),t(L1,Y,R)) :- L \= nil,  
                                delmax(L,L1,Y).
```

Binární vyhledávací stromy: vypust'

```
% delmax(+B,-B1,-Y) :- B1 vznikne  
% vypuštěním maximálního prvku Y  
% z BVS B.
```

```
delmax(t(L,X,nil),L,X).
```

```
delmax(t(L,X,R),t(L,X,R1),Y):- R \= nil,  
                                delmax(R,R1,Y).
```

Problémy se stromy

- ① Definujte predikáty pro
 - vložení prvku
 - odstranění kořenez **binární haldy**.
- ② Navrhněte efektivní algoritmus, který zjistí, zdali je zadaný binární strom symetrický dle svislé osy, procházející jeho kořenem.
- ③ Navrhněte reprezentaci pro obecné (kořenové) stromy.

Predikát !/0

- vždy uspěje
- při pokusu o návrat při backtrackingu způsobí okamžité selhání splňovaného cíle

$c_1 :- p_1, \dots, p_i, !, p_j, \dots, p_k.$

$c_2 :- p_m, \dots, p_n.$

- c_1 a c_2 jsou termy s hlavním funktorem c
- p_i uspěje \Rightarrow uspěje i $!$
- p_j selže \Rightarrow selže cíl c

☀ Příklad řezu

```
% prvek(X, Seznam) :- X je prvkem Seznamu.  
prvek(X, [X|_]).  
prvek(X, [_|Xs]) :- prvek(X, Xs).  
  
% prvek_det(X, Seznam) :-  
%           prvek/1 deterministicky.  
prvek_det(X, [X|_]) :- !.  
prvek_det(X, [_|Xs]) :- prvek_det(X, Xs).  
  
• prvek_det(-X, +S) vrátí první prvek X v S
```


Červený řez

Mění deklarativní význam programu

$p :- a, b.$

$p :- c.$

$p \Leftarrow (a \wedge b) \vee c$

$p :- a, !, b.$

$p :- c.$

$p \Leftarrow (a \wedge b) \vee (\neg a \wedge c)$

☀ **Příklad:** `prvek/2` vs. `prvek_det/2`

Zelený řez

Nemění deklarativní význam programu

- pouze “odřezává” neperspektivní větve výpočtu

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

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

☝ Ale pozor

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

```
max(X, Y, Y).
```

- ?- max(2, 1, 1).

true

chyba !!!

Negace neúspěchem

Lída má ráda muže:

```
ma_rada(lida, X) :- muz(X).
```

Lída má ráda muže, ale ne plešaté:

```
ma_rada(lida, X) :- plesaty(X),  
                    !,  
                    fail.
```

```
ma_rada(lida, X) :- muz(X).
```

Negace: operátor \+

```
% not(C) :- Cíl C nelze splnit.
```

```
not(C) :- C, !, fail.
```

```
not(C).
```



call(C)

Doporučená notace pro negaci

- operátor \+
- :- op(900, fy, \+).

```
ma_rada(lida, X) :- muz(X),  
                    \+ plesaty(X).
```

Neunifikovatelné ...

```
% neunif(X,Y) :- X a Y nelze  
%                  unifikovat.  
neunif(X,Y) :- X = Y, !, fail.  
neunif(X,Y).
```

Předdefinovaný operátor $\backslash=$.

```
neunif(X,Y) :- \+(X = Y).
```

... a různé

```
% ruzne(X,Y) :- X a Y jsou ruzne.
```

```
ruzne(X,Y) :- X == Y, !, fail.
```

```
ruzne(X,Y).
```

Předdefinovaný operátor `\==`.

```
ruzne(X,Y) :- \+(X == Y).
```

ekvivalence termů

Prolog: negace neúspěchem

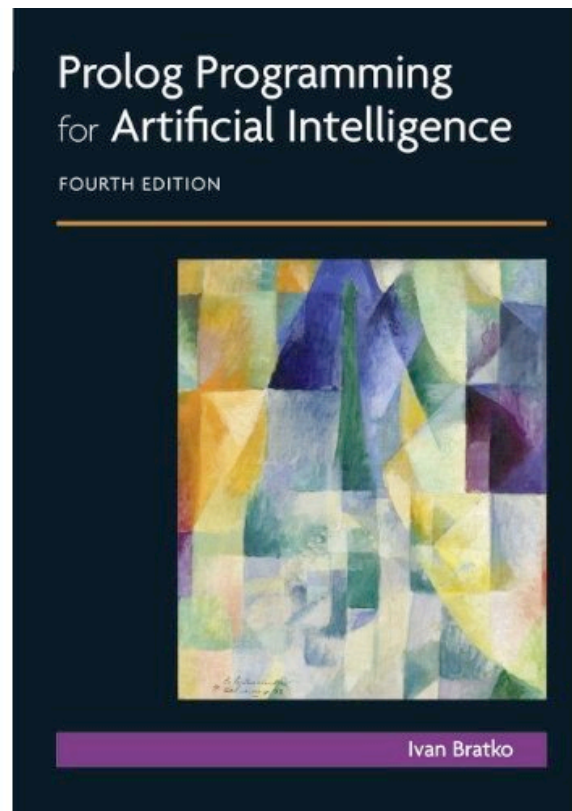
\+

- neodpovídá negaci v matematické logice
- negace neúspěchem
- předpoklad uzavřeného světa





Neprocedurální programování

Prolog 5

30.3.2021



Osnova

-  Negace nebo řez?
 - zkrocení řezu
-  Neúplně definované datové struktury
 - zřetězení v konstantním čase
-  Vestavěné predikáty : struktura termu
 - zjednodušování výrazů
 - symbolické derivování
-  Shromáždění všech výsledků dotazu

Negace nebo řez?

```
porazil(smid, panatta).
```

```
porazil(lendl, barazzutti).
```

```
porazil(barazzutti, smid).
```

Definujme predikat

```
kategorie(+Hrac, -Trida)
```

pro tridy

- vitez
- bojovnik
- sportovec

Negace ... ?

```
% kategorie(+Hrac,-Trida)
```

```
kategorie(X,vitez):-
```

```
    porazil(X,_), \+ porazil(_,X).
```

```
kategorie(X,bojovnik):-
```

```
    porazil(X,_), porazil(_,X).
```

```
kategorie(X,sportovec):-
```

```
    porazil(_,X), \+ porazil(X,_).
```

✗ Nevýhoda

- opakované vyhodnocení téhož cíle

... nebo řez?

```
kategorie(X,bojovnik):- porazil(X,_),  
                        porazil(_,X),!.  
kategorie(X,vitez):-   porazil(X,_),!.  
kategorie(X,sportovec):- porazil(_,X).
```

Idiom

```
p :- test1, !, tělo1.  
p :- test2, !, tělo2.  
p :- tělo3.
```

... nebo řez?

```
kategorie(X,bojovnik):- porazil(X,_),  
                        porazil(_,X),!.  
kategorie(X,vitez):-   porazil(X,_),!.  
kategorie(X,sportovec):- porazil(_,X).
```

 **Problém:** Jak dopadnou dotazy typu

- `kategorie(+Hrac,+Trida)`
- `kategorie(-Hrac,+Trida)?`

Prolog: negace neúspěchem

\+

- neodpovídá negaci v matematické logice
- negace neúspěchem
- předpoklad uzavřeného světa

☀ Příklad negace neúspěchem

```
jazyk(c).  
jazyk(python).  
jazyk(prolog).  
jazyk(haskell).
```

```
proc(c).  
proc(python).
```

```
?- proc(X).
```

```
    X = c ;
```

```
    X = python
```

☀ Příklad negace neúspěchem

```
jazyk(c).  
jazyk(python).  
jazyk(prolog).  
jazyk(haskell).
```

```
proc(c).  
proc(python).
```

```
?- \+ proc(X).
```

```
    false
```

```
?- jazyk(X), \+ proc(X).
```

```
    X = prolog ;
```

```
    X = haskell
```


☀ Příklad negace neúspěchem

```
jazyk(c).  
jazyk(python).  
jazyk(prolog).  
jazyk(haskell).
```

```
proc(c).  
proc(python).
```

```
?- \+ proc(X).
```

```
false
```

```
?- \+ proc(X), jazyk(X).
```

```
false
```

Negace: volné proměnné

`\+ C`

- `C` může obsahovat volné proměnné

Možné řešení

- definovat negaci (`not/1`) pouze pro základní termy
 - » term bez volných proměnných
- SWI Prolog
 - » `not/1` ekvivalentní `\+`
 - » norma (ISO) doporučuje používat `\+`

Zkrocení řezu

```
% once(Cíl) :- vrátí první řešení,  
%               které splní Cíl  
  
once(C) :- C, !.  
  
% forall(+Podminka, +Cíl):-  
%     uspěje, pokud Cíl lze splnit  
%     pro všechny hodnoty proměnných  
%     pro než lze splnit Podminku  
  
forall(Podminka, Cíl) :-  
    \+ (Podminka, \+ Cíl).
```

If -> Then ; Else

If -> Then ; _ :- If, !, Then.

If -> _ ; Else :- !, Else.

If -> Then :- If, !, Then.

✓ Podmínka **If** se vyhodnocuje jen jednou

Uvnitř “větví” **Then** a **Else** možný backtracking

☀ Příklad

```
% sjednoceni(+Xs,+Ys,-Zs):- seznam Zs  
%                je sjednocením množin  
%    reprezentovaných seznamy Xs a Ys.
```

Sjednocení pomocí negace

```
sjednoceni([ ], Ys, Ys) .
```

```
sjednoceni([X | Xs], Ys, Zs) :-  
    member(X, Ys),  
    sjednoceni(Xs, Ys, Zs) .
```

```
sjednoceni([X | Xs], Ys, [X | Zs]) :-  
    \+ member(X, Ys),  
    sjednoceni(Xs, Ys, Zs) .
```

Sjednocení pomocí řezu

```
sjednoceni([ ], Ys, Ys) .
```

```
sjednoceni([X|Xs], Ys, Zs) :-  
    member(X, Ys), !,  
    sjednoceni(Xs, Ys, Zs) .
```

```
sjednoceni([X|Xs], Ys, [X|Zs]) :-  
    sjednoceni(Xs, Ys, Zs) .
```

Sjednocení pomocí if-then-else

```
sjednoceni([ ], Ys, Ys) .
```

```
sjednoceni([X | Xs], Ys, Zs) :-
```

```
    (member(X, Ys) -> Zs=Zs1 ; Zs=[X | Zs1]) ,
```

```
    sjednoceni(Xs, Ys, Zs1) .
```

Predikáty pro řízení výpočtu

- ✓ nabízejí idiomy imperativního programování
- může existovat elegantnější řešení v neprocedurálním duchu

Neúplně definované datové struktury

$[a, b, c]$
seznam



$[a, b, c | S] - S$
rozdílový seznam

volná
proměnná

Zřetězení rozdílových seznamů

- v **konstantním** čase
- `zretez(A-B, B-C, A-C)`.

?- `zretez([a,b,c|X]-X, [d,e|Y]-Y, Z)`.

$X = [d, e | Y],$

$Z = [a, b, c, d, e | Y] - Y$

Rozdílové seznamy

obyčejný seznam \leftrightarrow rozdílový seznam

% `prevod1(+OS,-RS) :- RS je rozdílová
reprezentace obyčejného seznamu OS.`

?- `prevod1([a,b,c], RS).`

$RS = [a,b,c|S] - S$

`prevod1([], S-S).`

`prevod1([X|Xs], [X|S]-T) :- prevod1(Xs, S-T).`

% `prevod2(-OS,+RS)`

?- `prevod2(OS, [a,b,c|S]-S).`

$OS = [a,b,c]$

`prevod2(Xs, Xs-[]).`

Quicksort efektivně

```
quicksort([ ], S-S).
```

```
quicksort([X|Xs], -) :-
```

```
    qsplit(X, Xs, Ys, Zs),
```

```
    quicksort(Ys, -),
```

```
    quicksort(Zs, -),
```

```
    append(-, -).
```

Problém

- navrhnete efektivní verzi třídění quicksortem
- odstraňte explicitní volání predikátu `append/3`
- ke zřetězení využijte rozdílové seznamy

Vestavěné predikáty: test typu termu

`atom/1` argumentem je atom

`atomic/1` argumentem je konstanta

`number/1` `integer/1` `float/1`

`var/1` argumentem je volná proměnná

`nonvar/1` argumentem není volná proměnná

`ground/1` argumentem je základní term

- bez volných proměnných

`compound/1` argumentem je složený term

Příklad

3. V následujícím algebrogramu nahraďte písmena číslicemi tak, aby platily rovnosti v řádcích i ve sloupcích. Každé písmeno nahraďte jednou číslicí, různým písmenům odpovídají různé číslice. Kromě výsledku uveďte také postup úvah, které vedly k vyřešení úlohy. Nalezněte všechna řešení a zdůvodněte, proč jiné řešení neexistuje.

$$\begin{array}{rclcl} ABC & - & AD & = & CF \\ - & & * & & + \\ JF & + & AG & = & CH \\ \hline GD & + & AEJ & = & ACG \end{array}$$

Algebrogrammy

$$\begin{array}{rcccccc} & D & O & N & A & L & D \\ + & G & E & R & A & L & D \\ \hline R & O & B & E & R & T \end{array} \qquad \begin{array}{rcccccc} & 5 & 2 & 6 & 4 & 8 & 5 \\ + & 1 & 9 & 7 & 4 & 8 & 5 \\ \hline 7 & 2 & 3 & 9 & 7 & 0 \end{array}$$

?- `soucet([D,O,N,A,L,D],
[G,E,R,A,L,D],[R,O,B,E,R,T]).`

$D = 5, O = 2, N = 6, A = 4, L = 8,$
 $G = 1, E = 9, R = 7, B = 3, T = 0$

Rozbor struktury termu: univ

Vestavěný operátor `=..`

- `univ`
- `Term =.. Seznam`
 - » `Seznam = [HlavniFunktor | SeznamArgumentu]`
 - » `+Term =.. -Seznam`
 - » `-Term =.. +Seznam`

?- `f(a,b) =.. S.`

`S = [f,a,b]`

?- `T =.. [p,X,f(X,Y)].`

`T = p(X,f(X,Y))`

Rozbor struktury termu: functor

Specifičtější vestavěné predikáty

`functor(Term, F, A) :- Term` má hlavní
funktor `F` a aritu `A`.

- k termu určí funktor a aritu: `(+, ?, ?)`
- k funktoru a aritě vytvoří term: `(?, +, +)`

`?- functor(f(a,b), F, A).`

`F = f`

`A = 2`

`?- functor(Term, f, 2).`

`Term = f(_G328, _G329)`

Rozbor struktury termu: arg

`arg(+N,+Term,?A) :- A je N-tým
argumentem Termu.`

`?- arg(2,f(X,t(a),t(b)),A).`

`A = t(a)`

☀ Příklad

`?- functor(D,datum,3),`

`arg(1,D,30),`

`arg(2,D,brezen),`

`arg(3,D,2021).`

`D = datum(30,brezen,2021)`

Příklad zjednodušování výrazů

```
s(*, X, 1, X).
```

```
s(*, 1, X, X).
```

```
s(*, X, Y, Z) :- integer(X), integer(Y),  
                  Z is X*Y.
```

```
s(*, X, Y, X*Y). % zarážka pro *
```

Podobná tabulka pro další operátory

```
simp(V, V) :- atomic(V), !.
```

```
simp(V, ZV) :- V =.. [Op, La, Pa],  
                simp(La, ZL), simp(Pa, ZP),  
                s(Op, ZL, ZP, ZV).
```

Zjednodušování výrazů

?- `simp(2*3*a,Z).`

`Z = 6*a`

?- `simp(a*2*3,Z).`

`Z = a*2*3`

 **Problém:** Co s tím?

```
s(*,X*Y,Z,W*X) :- integer(Y),  
                    integer(Z),  
                    W is Y*Z.
```

☀ Příklad: symbolické derivování

```
?- der(x^3,x,D) .
```

```
    D = 3*x^2
```

```
% der(+Vyráz,+X,-Der):- Der je derivací
```

```
%           Vyrázu vzhledem k proměnné X
```

```
%           Vyráz a X jsou základní termy
```

```
der(X,X,1) .
```

```
der(Y,X,0) :- atomic(Y), X\=Y.
```

☀ Příklad: symbolické derivování

% derivace elementárních funkcí

`der(sin(X),X,cos(X)).`

`der(cos(X),X,-sin(X)).`

`der(e^X,X,e^X).`

`der(ln(X),X,1/X).`

% derivace mocniny

`der(X^N,X,N*X^N1):- number(N),
N1 is N-1.`

☀ Příklad: symbolické derivování

% pravidla pro různé operátory

$\text{der}(F+G, X, DF+DG) :- \text{der}(F, X, DF),$
 $\text{der}(G, X, DG).$

$\text{der}(F-G, X, DF-DG) :- \text{der}(F, X, DF),$
 $\text{der}(G, X, DG).$

$\text{der}(F * G, X, F * DG + DF * G) :- \text{der}(F, X, DF),$
 $\text{der}(G, X, DG).$

$\text{der}(F / G, X, (G * DF - F * DG) / (G * G)) :-$
 $\text{der}(F, X, DF),$
 $\text{der}(G, X, DG).$

☀ Příklad: symbolické derivování

```
?- der(sin(cos(x)), x, D).
```

```
D = cos(cos(x)) * -sin(x)
```

% derivace složené funkce

```
der(F_G_X, X, DF*DG) :- F_G_X =.. [_ , G_X],  
                        G_X \= X,  
                        der(F_G_X, G_X, DF),  
                        der(G_X, X, DG).
```

✎ Problém

- neumí zjednodušit výsledek

```
» ?- der(x*x, x, D).
```

```
»      D = x*1+1*x
```

Shromáždění všech výsledků dotazu

Vestavěné predikáty `bagof/3`, `setof/3`, `findall/3`

`bagof(±Objekt, ±Cil,
-SeznamObjektuSplnujicichCil)`

Pokud `Cil` nelze splnit, `bagof` selže

`Seznam` může obsahovat opakované výskyty

Pokud `Cil` obsahuje volnou proměnnou `X`,
která není obsažena v `Objektu`

- `bagof` postupně vrátí všechny výsledky
- pro všechny různé hodnoty `X`, pro něž `Cil` uspěje
- `X^Cil` všechna řešení bez ohledu na hodnoty `X`

Příklad

```
trida(b,sou). trida(a,sam).  
trida(c,sou). trida(e,sam).  
trida(d,sou).
```

Dotazy

```
?- bagof(P, trida(P,sou), Pismena).
```

```
    Pismena = [b,c,d]
```

```
?- bagof(P, trida(P,T), Pismena).
```

```
    T = sou, Pismena = [b,c,d] ;
```

```
    T = sam, Pismena = [a,e]
```

```
?- bagof(P, T^trida(P,T), Pismena).
```

```
    Pismena = [b,a,c,e,d]
```


Vestavěné predikáty: setof

setof/3

- jako bagof/3, ale
- vrátí **uspořádaný** seznam
- bez duplicit

?- **setof(T/P, trida(P,T), Pismena).**

**Pismena = [sam/a,sam/e,sou/b,
sou/c,sou/d]**

Vestavěné predikáty: findall

`findall/3`

- jako `bagof/3`, ale
- shromáždí všechna řešení **bez ohledu na volné proměnné**, nevyskytující se v `Cil`i
- **vždy** uspěje
 - » pokud `Cil` nelze splnit, vrátí `[]`

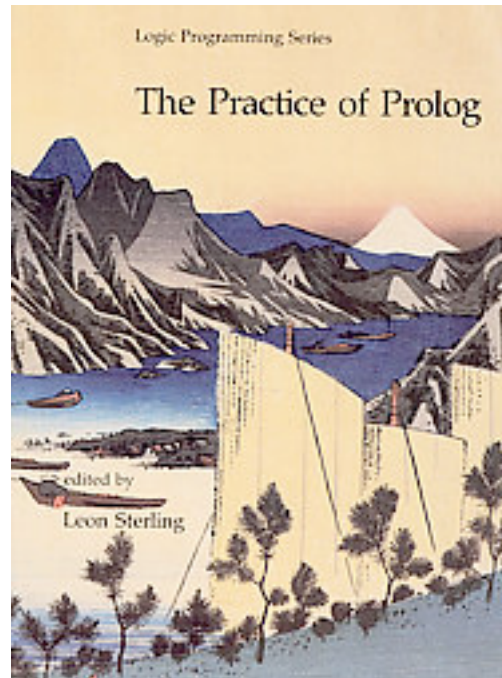
?- `findall(P, trida(P,T), Pismena).`

`Pismena = [b,a,c,e,d]`

Neprocedurální programování

Prolog 6

6.4.2021



Osnova

Grafové algoritmy

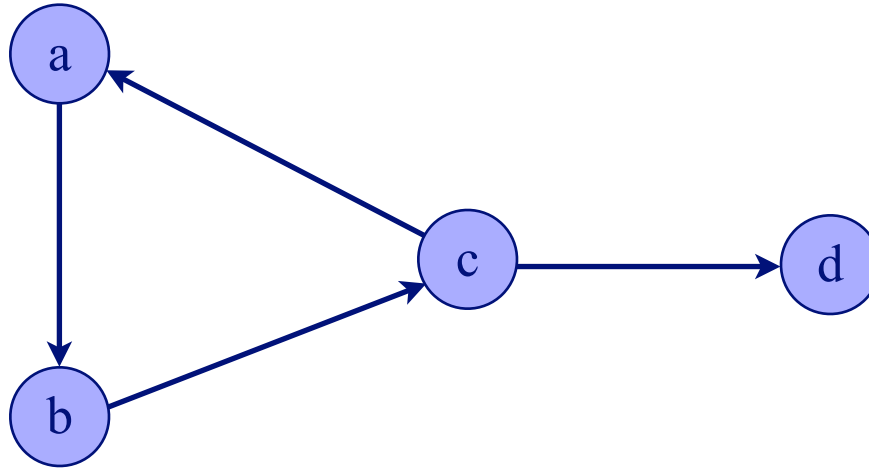
- průchod do hloubky a do šířky

Prohledávání stavového prostoru

Vstup & výstup

- zpracování přirozeného jazyka, Eliza

Grafové algoritmy



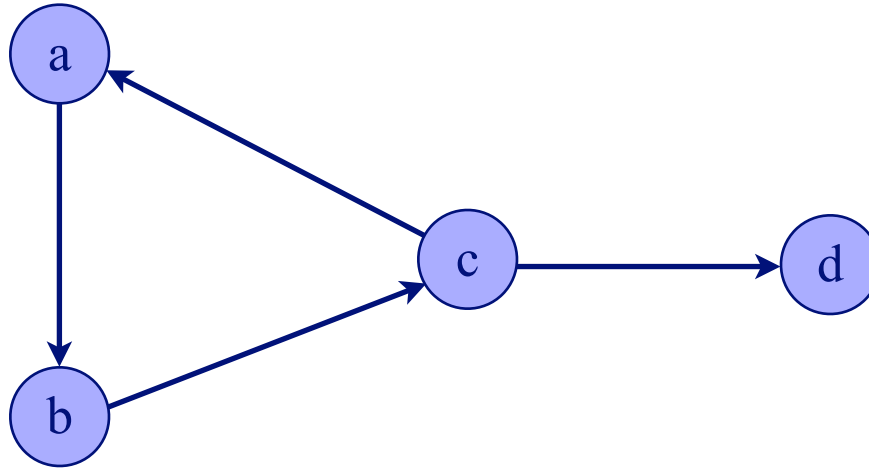
Reprezentace grafu

% pomocí faktů

```
vrchol(a). vrchol(b). vrchol(c). vrchol(d).
```

```
hrana(a,b). hrana(b,c). hrana(c,a). hrana(c,d).
```

Grafové algoritmy



Reprezentace grafu

- `graf([a,b,c,d],
[h(a,b),h(b,c),h(c,a),h(c,d)])`
- `[a->[b],b->[c],c->[a,d],d->[]]`

Grafy: reprezentace

Rozhraní

```
vrchol(?Vrchol, +Graf)
```

```
vrchol(V, graf(Vrcholy, Hrany)) :-  
    member(V, Vrcholy).
```

```
hrana(?Vrchol1, ?Vrchol2, +Graf)
```

```
hrana(V1, V2, graf(Vrcholy, Hrany)) :-  
    member(h(V1, V2), Hrany).
```

Dále jen: `hrana(Vrchol1, Vrchol2)`

Grafy: dosažitelnost

Hledání cesty v grafu průchodem do hloubky
(Depth First Search)

```
% dfs(+Start,?Cil):- existuje cesta  
%    z vrcholu Start do vrcholu Cil?
```

```
dfs(X,X).
```

```
dfs(X,Z):- hrana(X,Y), dfs(Y,Z).
```

✗ Korektní jen pro **acyklické** grafy!

Grafy: průchod do hloubky

```
dfs(X,Y):- dfs(X,Y,[X]).
```

```
% dfs(X,Y,Nav) :- Nav je seznam  
%                již navštívených vrcholů.
```

```
dfs(X,X,_).
```

```
dfs(X,Z,Nav):- hrana(X,Y),  
                \+ member(Y,Nav),  
                dfs(Y,Z,[Y|Nav]).
```

 **Problém:** `dfs/3` nevrací nalezenou cestu

Grafy: průchod do hloubky

Predikát, který vrátí i nalezenou cestu

```
% dfs(X,Y,Cesta):- Cesta je seznam  
% vrcholů na cestě z X do Y.
```

```
dfs(X,Y,Cesta):- dfs(X,Y,[X],Cesta).
```

```
dfs(X,X,_,[X]).
```


```
dfs(X,Z,Nav,[X|Ys]):- hrana(X,Y),  
                        \+ member(Y,Nav),  
                        dfs(Y,Z,[Y|Nav],Ys).
```

Grafy: průchod do šířky

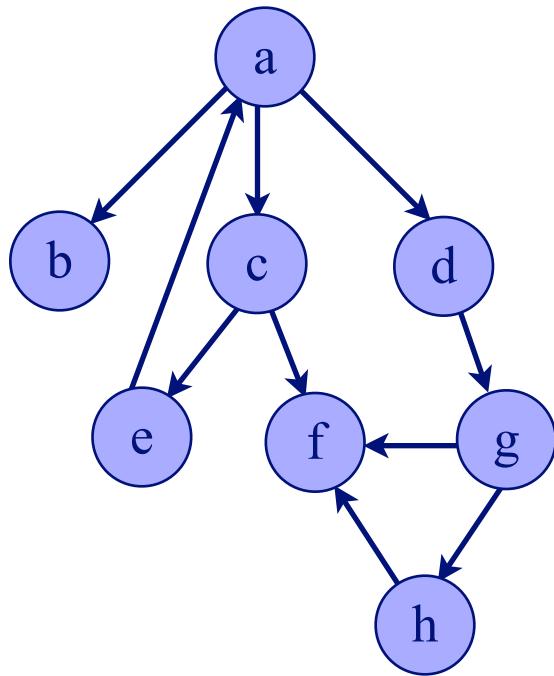
Hledání cesty v grafu **průchodem do šířky**
(Breadth First Search)

- použití fronty již nalezených cest
- která reprezentuje BFS-strom

```
% bfs(+Start,+Cil,-Cesta):- Cesta z
%      vrcholu Start do vrcholu Cil
%      nalezená průchodem do šířky.
bfs(Start,Cil,Cesta):-
    bfs1([[Start]],Cil,CestaRev),
    reverse(CestaRev,Cesta).
```



☀ Příklad – fronta cest



?– bfs(a,f,Cesta).

[[a]]

[[b,a], [c,a], [d,a]]

[[c,a], [d,a]]

[[d,a], [e,c,a], [f,c,a]]

[[e,c,a], [f,c,a], [g,d,a]]

[[f,c,a], [g,d,a]]

Grafy: průchod do šířky

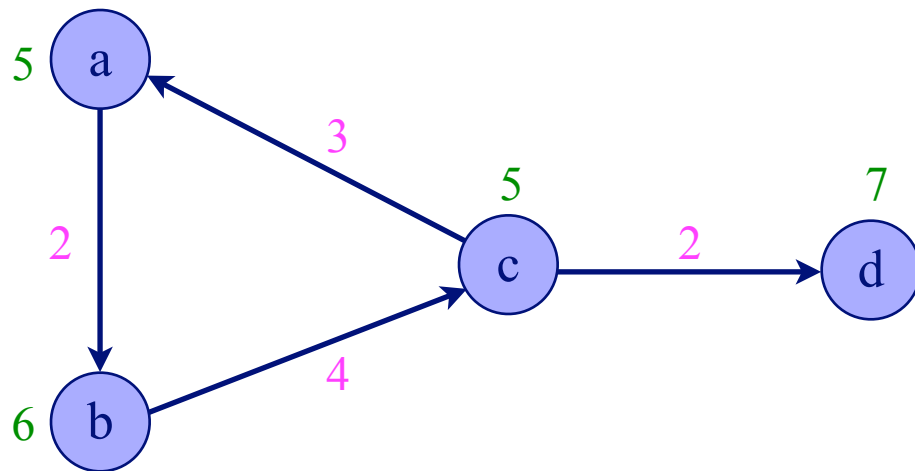
```
% bfs1(Fronta, Cil, CestaRev)
bfs1([Xs|_], Cil, Xs):- Xs=[Cil|_].
bfs1([[X|Xs]|Xss], Cil, CestaR):-
    findall([Y,X|Xs],
            (hrana(X,Y), \+member(Y,[X|Xs])),
            NoveCesty),
    append(Xss, NoveCesty, NovaFronta), !,
    bfs1(NovaFronta, Cil, CestaR).
```

acyklické prodloužení

Problémy

- ① Navrhněte efektivnější verzi predikátu **bfs1/3**, v níž bude **zřetězení** realizováno pomocí **rozdílových seznamů**.
- ② Implementujte verzi průchodu do šířky, v níž budeme cesty prodlužovat pouze vrcholy, které jsme dosud **vůbec nenavštívili**.

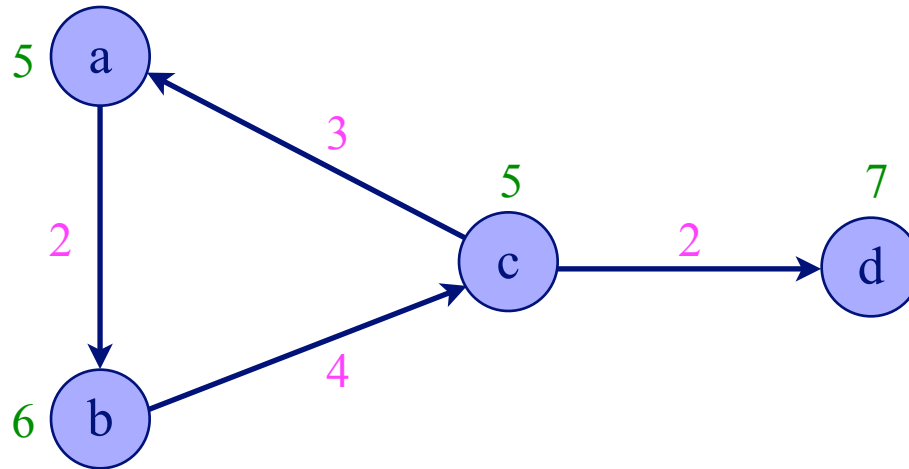
Grafy s ohodnocením



Reprezentace grafu

- `graf([a/5,b/6,c/5,d/7],
[h(a,b,2),h(b,c,4),h(c,a,3),h(c,d,2)])`
- `[a/5->[b/2],b/6->[c/4],
c/5->[a/3,d/2],d/7->[]]`

Grafy s ohodnocením



Rozhraní

- `vrchol(?Vrchol, ?Ohodnoceni, +Graf)`
- `hrana(?Vrchol1, ?Vrchol2, ?Ohod, +Graf)`
- dále jen `hrana(Vrchol1, Vrchol2, Ohod)`

Grafy: problém nejkratší cesty

Graf bez ohodnocení hran

- délka cesty = # hran
- nejkratší cesta \Rightarrow bfs/3

Graf s nezáporným ohodnocením hran

- cena cesty = \sum ohodnocení hran
- nejkratší cesta \Rightarrow dijkstra/3

Reprezentace cesty

- seznam $[a, b, c] \Rightarrow \text{term } c(\text{Cena}, [a, b, c])$

Grafy: Dijkstrův algoritmus

```
dijkstra(Start, Cil, Cesta) :-  
    dijkstra1([c(0, [Start])], Cil, C),  
    reverse(C, Cesta).
```

Modifikace bfs1/3 \Rightarrow dijkstra1/3

- fronta cest \Rightarrow prioritní fronta cest (s cenami)
- výběr nejdříve přidané cesty \Rightarrow výběr cesty minimální ceny

Prohledávání stavového prostoru

Příklad: Úloha o farmáři, vlku, koze a zeli

- farmář převáží vlka, kozu a zeli na druhý břeh
- do lodky se vejdou vždy jen dva objekty
- farmář nesmí zanechat na jednom břehu
 - » kozu & zeli
 - » vlka & kozu

Řešení úlohy: posloupnost stavů

Reprezentace stavu

- $s(\text{Farmer}, \text{Vlk}, \text{Kozu}, \text{Zeli})$
- počáteční stav: $s(1, 1, 1, 1)$
- cílový stav: $s(p, p, p, p)$

☀ Příklad: Farmář, vlk, koza, zelí

```
proti(l,p).      proti(p,l).  
prevoz(s(F,V,K,Z),s(F1,V,K,Z)):-  
    proti(F,F1).  
prevoz(s(F,F,K,Z),s(F1,F1,K,Z)):-  
    proti(F,F1).  
prevoz(s(F,V,F,Z),s(F1,V,F1,Z)):-  
    proti(F,F1).  
prevoz(s(F,V,K,F),s(F1,V,K,F1)):-  
    proti(F,F1).
```

Farmář, vlk, koza, zelí

Predikát **bezpecny/1**

- definuje "bezpečný" stav

bezpecny (s (F , V , F , Z)) .

bezpecny (s (F , F , K , F)) :- proti (F , K) .

Predikát **dalsi/2** generuje k zadanému stavu všechny stavy následující

dalsi (Stav1 , Stav2) :-
 prevoz (Stav1 , Stav2) ,
 bezpecny (Stav2) .

Farmář, vlk, koza, zelí

Zbývá nalézt cestu v grafu

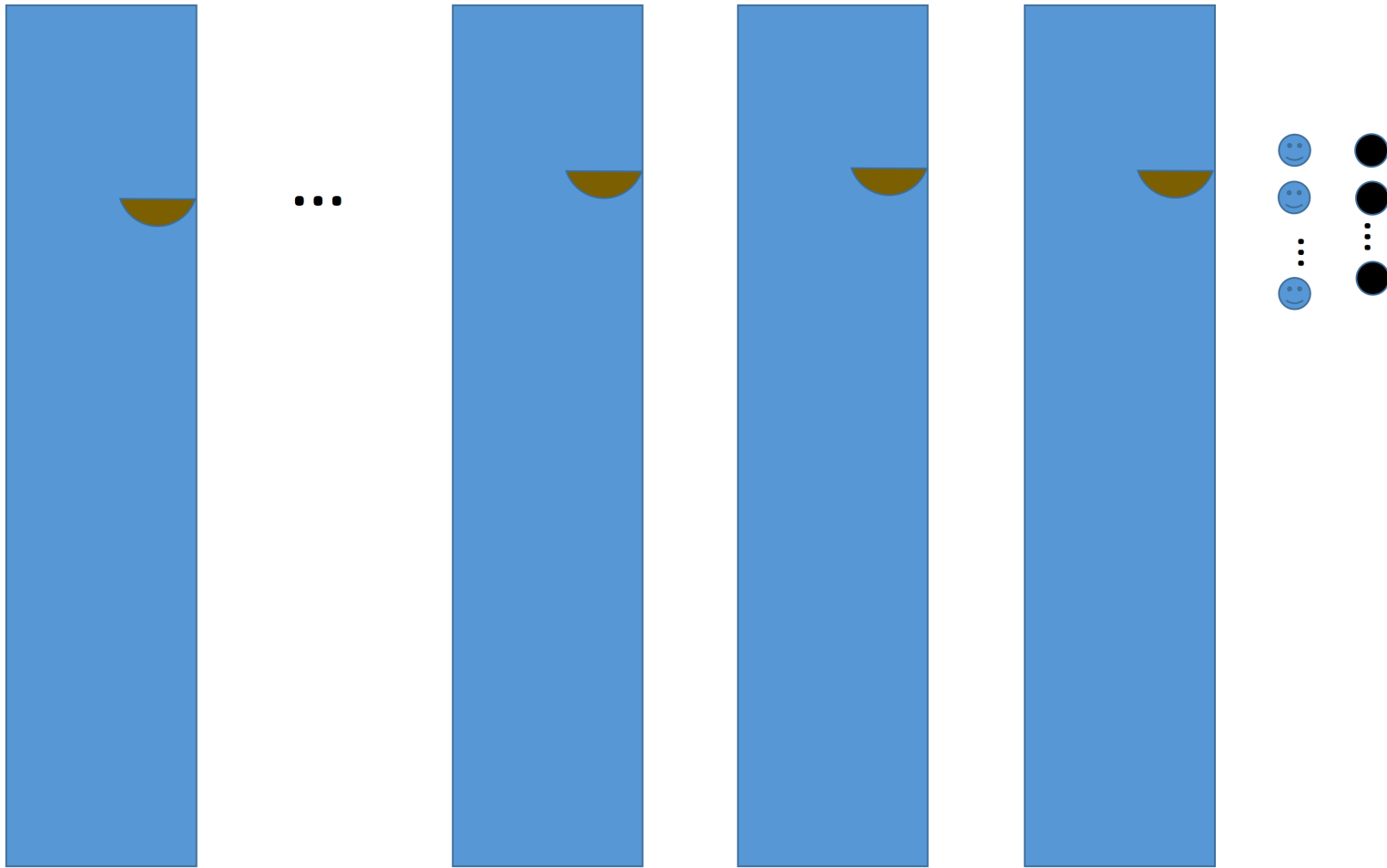
- s vrcholy $s(F, V, K, Z)$
- a hranami $hrana(X, Y) :- dalsi(X, Y)$
- z vrcholu $s(1, 1, 1, 1)$
- do vrcholu $s(p, p, p, p)$

```
fvkz(Resení) :- dfs(s(1, 1, 1, 1),  
                    s(p, p, p, p),  
                    Resení).
```

```
fvkz(Resení) :- bfs(s(1, 1, 1, 1),  
                   s(p, p, p, p),  
                   Resení).
```

nejkratší
řešení

Problém misionářů a lidojedů



Heuristické prohledávání

Best First Search

- expanze stavu, který má “největší šanci” na to, že povede k cíli
- jak takový stav najít?

Zavedeme ohodnocující funkci f


- $f(s) = g(s) + h(s)$
 - » $g(s)$ = cena optimální cesty ze startu do s
 - » $h(s)$ = cena optimální cesty z s do cíle
- g ani h neznáme \Rightarrow použijeme odhad

$$\hat{f}(s) = \hat{g}(s) + \hat{h}(s)$$

Heuristické prohledávání: A*

Algoritmus A*

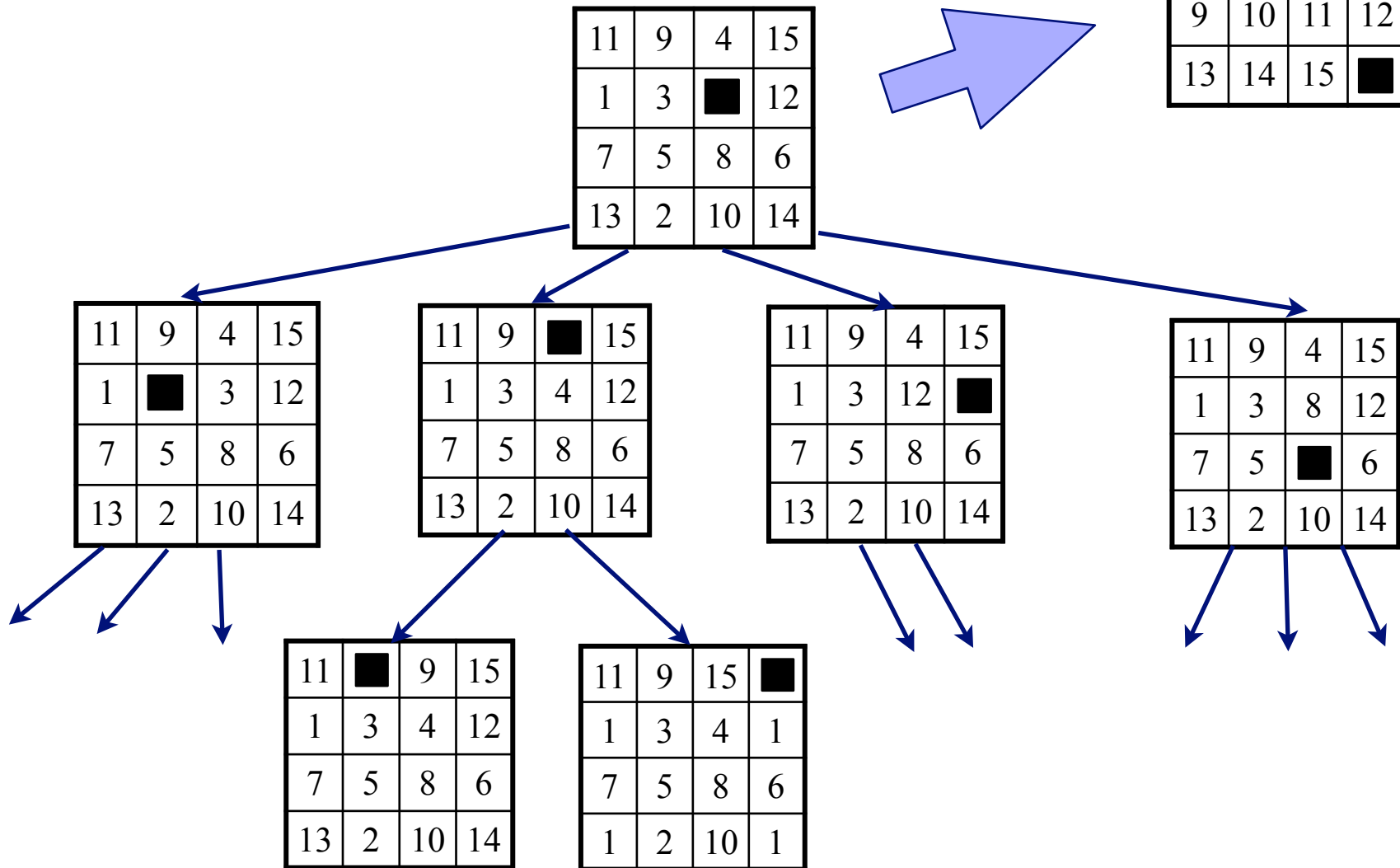
- používá ohodnocující funkci $\hat{f}(s) = \hat{g}(s) + \hat{h}(s)$
- kde $\hat{g}(s)$ = cena nalezené cesty se startu do s
- jak odhadnout $\hat{h}(s)$?

 **Věta.** *Pokud existuje $\delta > 0$ tak, že cena žádné hrany neklesne pod δ a $\hat{h}(s) \leq h(s)$ pro každý stav s , pak první řešení nalezené algoritmem A* je řešení optimální.*

Úvod do umělé inteligence NAIL120

☀ Příklad: Loydova “15”

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	■



Vstup a výstup: termy

V/V termů

`read(?T)` přečte z aktuálního vstupu jeden term (ukončený tečkou) a unifikuje jej s `T`

`write(+T)` vypíše na aktuální výstup hodnotu termu `T`

- s právě platnými hodnotami proměnných v termu `T` obsažených

Vstup a výstup: znaky

Znakový vstup

`get_char(?C)` unifikuje **C** s dalším znakem na vstupu

`get_code(?C)` unifikuje **C** s kódem dalšího znaku na vstupu

Vstup a výstup: znaky

Znakový výstup

`put_char(Z)` vypíše znak `Z`
na aktuální výstup

`put_code(C)` vypíše znak s kódem `C`
na aktuální výstup

`tab(N)` vypíše `N` mezer

`nl` nový řádek

Vstup a výstup: proudy

Implicitní vstup - klávesnice, výstup - obrazovka

- atom `user`

Edinburgský model

- `see(+F)` nastaví vstup ze souboru `F`
» `see('C:\prolog\data.pl')`
- `seen/0` uzavře aktuální vstup, `see(user)`
- `seeing(-F)` dotaz na aktuální vstupní soubor
- `tell/1`, `told/0`, `telling/1` analogicky pro výstup

Vstup a výstup: cykly

Standardní predikát `repeat/0`

```
repeat.
```

```
repeat :- repeat.
```

☀ **Příklad**

```
seeing(In), % zjistí a uschová
```

```
telling(Out), % aktuální V/V
```

```
see(F1), % otevře vstupní soubor
```

```
tell(F2), % otevře výstupní soubor
```

Vstup a výstup: příklad

```
repeat,    % opakuj
read(X),   % načti další term
( X=end_of_file, !,    % ukončení
  told,seen,          % uzavření souborů
  see(In),tell(Out) % obnovení V/V
;                    % není EOF
transformuj(X,Y) % vlastní zpracování
write(Y), % term do F2
fail % návrat na začátek cyklu
).
```


Vstup a výstup: ISO

Standard ISO

- `open(+Soubor, +Mode, ?Proud)`
 - » otevře `Soubor` v režimu `Mode` (`read`, `write`, `append`, `update`)
 - » proměnná `Proud` je vázána na číselnou identifikaci proudu
 - » atom `Proud` se stává identifikátorem proudu
 - » `open/4`
- `close(+Proud)`

Vstup a výstup: ISO

Standard ISO

- `set_input(+Stream)`

`open(file, read, Stream),`
`set_input(Stream) \cong see(file)`

- `set_output(+Stream)`
- `current_input(-Stream)`
- `current_output(-Stream)`

Zpracování přirozeného jazyka



Eliza: Dialog s psychoanalytičkou

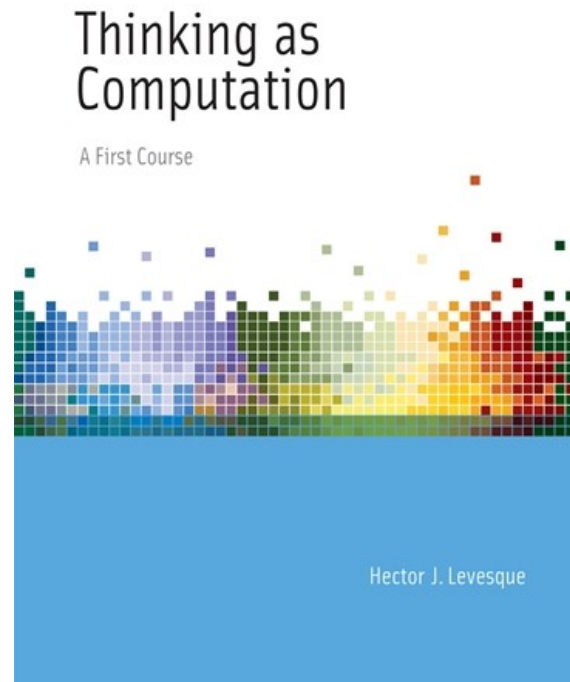
- J. Weizenbaum, ELIZA - A computer program for the study of natural language communication between man and machine. *Comm. of the ACM* 9 (1966), 36-45.
- Turingova imitační hra

```
eliza:- write('Hello. My name is Eliza.  
          How can I help you?'),  
        nl, cti_vetu(V), eliza(V), !.  
eliza(Vstup):- member('quit', Vstup),  
                write('Goodbye.  
My secretary will send you a bill. '), nl.
```





Neprocedurální programování

Prolog 7

13.4.2021



Osnova

-  Zpracování přirozeného jazyka
 - Eliza
-  Vestavěné predikáty pro modifikaci programu
 - `assert/1`, `retract/1`
-  Práce s množinami řešení
-  Predikáty vyšších řádů

Zpracování přirozeného jazyka



Eliza: Dialog s psychoanalytičkou

- J. Weizenbaum, ELIZA - A computer program for the study of natural language communication between man and machine. *Comm. of the ACM* 9 (1966), 36-45.
- Turingova imitační hra

```
eliza:- write('Hello. My name is Eliza.  
          How can I help you?'),  
        nl, cti_vetu(V), eliza(V), !.  
eliza(Vstup):- member('quit', Vstup),  
                write('Goodbye.  
My secretary will send you a bill. '), nl.
```

Eliza: podnět a odezva

```
vzor([ 'I' ,am,1],  
      [ 'How' ,long,have,you,been,1,?] ).  
vzor([ 1,you,2,me],  
      [ 'What' ,makes,you,think, 'I' ,2,you,?] ).  
vzor([ 'I' ,like,1],  
      [ 'Does' ,anyone,else,in,your,family,like,1,?] ).  
vzor([ 'I' ,feel,1],  
      [ 'Do' ,you,often,feel,that,way,?] ).  
vzor([ 1,X,2],  
      [ 'Can' ,you,tell,me,more,about,X,?] ):-  
                                          important(X).  
vzor([ 1],[ 'Please' ,go, 'on.' ] ).
```

Eliza: pomocné predikáty

👉 Klíčová slova

```
important(father).
```

```
important(mother).
```

```
important(brother).
```

```
important(son).
```

```
important(daughter).
```

```
important(sister).
```

Predikát **hledej/3** pro hledání ve asociativním seznamu

```
hledej(Klic, [Klic-Hodnota|_], Hodnota).
```

```
hledej(Klic, [Klic1-_|Slovník], Hodnota):-  
    Klic \= Klic1,  
    hledej(Klic, Slovník, Hodnota).
```


Eliza: komunikační smyčka

```
eliza(Vstup):-  
    vzor(Podnet, Reakce),  
    match(Podnet, Slovník, Vstup),  
    match(Reakce, Slovník, Vystup),  
    reply(Vystup),  
    cti_vetu(Vstup1),  
    !,  
    eliza(Vstup1).  
reply([H|T]):- write(H), write(' '), reply(T).  
reply([]):- nl.
```

Eliza: reakce na podnět

```
match([Slovo|Vzor],Slovník,[Slovo|Cíl):-  
    atom(Slovo),  
    match(Vzor,Slovník,Cíl).  
match([N|Vzor],Slovník,Cíl):-  
    integer(N),  
    hledej(N,Slovník,LevýCíl),  
    append(LevýCíl,PravýCíl,Cíl),  
    match(Vzor,Slovník,PravýCíl).  
match([],_,[]).
```

Eliza: zpracování vstupu

```
% typ_znaku(+KodZnaku,?Typ):-  
% Typ znaku se zadany kodem KodZnaku,  
% Typ je oddelovac, konec_vety nebo jiny.  
typ_znaku(Z,konc_vety) :-  
    member(Z,[33,46,63]), !.  
    % vykřičník, tečka, otazník  
typ_znaku(Z,oddelovac) :- Z =< 32, !.  
    % mezery apod.  
typ_znaku(_,jiny).
```

Eliza: zpracování vstupu

```
% cti_pismena(+Pismeno,-S,-DalsiZnak):-  
% vrátí seznam S písmen slova, které  
% začíná písmenem Pismeno a za ním  
% následuje znak DalsiZnak.  
  
cti_pismena(Z,[ ],Z):-  
    typ_znaku(Z,konec_vety),!.  
    % konec věty  
  
cti_pismena(Z,[ ],Z):-  
    typ_znaku(Z,oddelovac),!.  
    % oddelovač - konec slova  
  
cti_pismena(Pis,[Pis|SezPis],DalsiZnak):-  
    get_code(Znak),  
    cti_pismena(Znak,SezPis,DalsiZnak).
```

Eliza: načtení věty

```
% cti_vetu(-SeznamSlov):- přečte na vstupu  
% větu a vrátí SeznamSlov věty.  
  
cti_vetu(SezSlov):-  
    get_code(Znak), cti_zbytek(Znak,SezSlov).  
  
cti_zbytek(Z,[ ]):-  
    typ_znaku(Z,konec_vety), !.    % konec věty  
  
cti_zbytek(Z,SezSlov):-  
    typ_znaku(Z,oddelovac), !, % mezera apod.  
    cti_vetu(SezSlov).  
  
cti_zbytek(Pismeno,[Slovo|SezSlov]):-  
    cti_pismena(Pismeno,SezPis,DalsiZnak),  
    name(Slovo,SezPis),  
    cti_zbytek(DalsiZnak,SezSlov).
```

Predikáty pro modifikaci programu

Umožní **přidávat** nové či **vyřazovat** existující klauzule programu

- mění deklarativní význam programu
- zpomalení výpočtu
- možnost simulace přiřazovacího příkazu

Predikát definovaný modifikovanou procedurou je třeba označit jako **dynamiccký**

- `:- dynamic predikat/2,
jiny_predikat/1.`

Predikáty `assert/1`, `retract/1`

`asserta(+T)` přidá term **T** jako novou klauzuli na začátek programu v paměti

`assertz(+T)` přidá term **T** jako novou klauzuli na konec programu v paměti

- `assert/1` ekvivalentní `assertz/1`

`retract(?T)` odstraní z programu v paměti první výskyt klauzule, kterou lze unifikovat s **T**

`retractall(?T)` odstraní z programu v paměti všechny klauzule, jejichž hlavu lze unifikovat s termem **T**

findall pomocí assert & retract

```
findall(X,Cil,SezVys):- zapis(X,Cil),  
                        seber([ ], SezVys).  
  
zapis(X,Cil):- Cil,  
               asserta(data999(X)),  
               fail.  
  
zapis(_,_) .  
  
seber(S,SezVys) :- data999(X),  
                  retract(data999(X)),  
                  seber([X|S], SezVys), !.  
  
seber(SezVys, SezVys) .
```


Práce s množinami řešení bez bagof

```
% komb(+Mnozina,+N,-Komb):- Komb je  
%      kombinace radu N z Mnoziny.  
komb(_,0,[ ]).  
komb([X|Xs],N,[X|Ys]):- N>0,  
                           N1 is N-1,  
                           komb(Xs,N1,Ys).  
komb([_|Xs],N,Ys):- N>0,  
                     komb(Xs,N,Ys).
```

Všetchny kombinace

```
% skomb(+Mnozina,+N,-SKomb):- SKomb
%           je seznam vsech kombinaci
%           radu N z Mnoziny.

skomb(_,0,[[ ]]).
skomb([ ],N,[ ]):- N>0.
skomb([X|Xs],N,SKomb):- N>0, N1 is N-1,
    skomb(Xs,N1,Yss),
    skomb(Xs,N,Zss),
    map_insert(X,Yss,Yss1),
    append(Yss1,Zss,SKomb).
```

Pomocný predikát map_insert

```
% map_insert(?X,?Xss,?Yss):-  
%           vlozi X do hlavy kazdeho  
%           seznamu v Xss a vrati v Yss.  
map_insert(_,[],[]).  
map_insert(X,[Xs|Xss],[[X|Xs]|Yss]):-  
           map_insert(X,Xss,Yss).
```

Predikáty vyšších řádů: `maplist/3`

```
maplist(_, [], []).
```

```
maplist(P, [X|Xs], [Y|Ys]):-
```

```
    Q=..[P,X,Y], Q, maplist(P,Xs,Ys).
```

```
?- maplist(reverse, [[1,2,3],[a,b]], V).
```

```
V = [[3,2,1],[b,a]]
```

```
% posledni(+Matice, ?Sloupec):- vrátí
```

```
%      posledni Sloupec Matice.
```

```
posledni(Matice, Sloupec):-
```

```
    maplist(last, Matice, Sloupec).
```

Predikáty vyšších řádů

```
% call(Cil,X,Y):- zavolá Cil  
%                  s argumenty X,Y
```

```
call(reverse,[1,2,3],S)
```

- reverse([1,2,3],S)

```
call(plus(1),2,X)
```

- plus(1,2,X)

Alternativní definice predikátu `maplist/3`

```
maplist(_,[],[]).
```

```
maplist(P,[X|Xs],[Y|Ys]):-
```

```
    call(P,X,Y), maplist(P,Xs,Ys).
```

map_insert/3 pomocí maplist/3

```
insert(X,Xs,[X|Xs]).
```

```
map_insert(X,Xss,Yss):-  
    maplist(insert(X),Xss,Yss).
```

Transpozice matice pomocí `maplist/3`

```
hlava([X|_],X).    % vrátí hlavu seznamu
telo([_|Xs],Xs).   % vrátí tělo seznamu

transponuj([[]|_],[]):- !. % báze
transponuj(Xss,[Xs|Zss]):-
    maplist(hlava,Xss,Xs), % první s.
    maplist(telo,Xss,Yss), % ostatní s.
    transponuj(Yss,Zss).
```

Srovnání: kombinace ve Scheme

≡ komb.scm ✕

sources > ≡ komb.scm

```
1 #lang scheme
2 (define (komb rad seznam)
3   (cond ((zero? rad) '(()))
4         ((null? seznam) '())
5         (else (append (map (lambda (rad-1) (cons (car seznam) rad-1))
6                                (komb (- rad 1) (cdr seznam)))
7                         (komb rad (cdr seznam)))
8         )
9   )
10 )
11 )
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
(base) Tom-MacBook-Pro:sources td$ racket
Welcome to Racket v8.0 [cs].
> (enter! "komb.scm")
"komb.scm"> (komb 2 '(1 2 3))
'((1 2) (1 3) (2 3))
"komb.scm"> □
```


Srovnání: kombinace v Haskellu

✖ komb.hs ×

sources > ✖ komb.hs

```
1  -- kombinace bez opakovani
2  komb :: Int -> [a] -> [[a]]
3
4  komb 0 _ = [[]]
5  komb _ [] = []
6  komb n (x:xs) | n>0 = map (x:) (komb (n-1) xs) ++ komb n xs
7  komb _ _ | otherwise = error "wrong argument"
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
(base) Tom-MacBook-Pro:sources td$ ghci
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
Prelude> :l komb
[1 of 1] Compiling Main                    ( komb.hs, interpreted )
Ok, one module loaded.
*Main> komb 2 [1,2,3]
[[1,2],[1,3],[2,3]]
*Main> 
```

Neprocedurální programování.

Funkcionální programování. Haskell Př. 8–13

Jan Hric

1. června 2021

Obsah

1 Úvod

2 Haskell 2 Dodatky

Outline

1 Úvod

2 Haskell 2 Dodatky

Funkcionální programování

- Programování s pomocí funkcí
- program = definice funkcí
- výpočet = aplikace funkce na argumenty
- v Hs (čistý jazyk): "matematické" funkce, bez vedlejších efektů
- datové struktury: λ -termy (abstrakce, aplikace)
- β -pravidlo: $(\lambda x.M)N \rightarrow_{\beta} M[x := N]$ ("beta", "lambda")
- výsledek: vyhodnocený tvar, normální forma (pokud existuje)
 - vyhodnocený tvar je jednoznačný (i díky teorii)
 - vyhodnocování je deterministické
- dnešní jazyky integrují části FP (nějak)
- z hlediska logiky: rovnostní teorie (pouze predikát rovnosti)

Příklad, Robinsonova aritmetika (část)

```

x + Z      = x      -- x+0 = x; Z jako Zero
x + (S y)  = S (x+y) -- x+S(y)=S(x+y)
--
x+y = if y==Z then x else S(x+ unS y)
      where unS (S y) = y

```

- Haskell, definice funkce + (pro numerály)
- case-sensitive (konstruktory termu vs. funkce), operátory
- symbolické výrazy (termy, ale jinak se píšou), i jako návratová hodnota fce.
- „=” : levou stranu přepisujeme pravou stranou
- S, Z: "vyhodnocené" funkce - (datové) konstruktory
- $\frac{(S\ Z) + (S\ (S\ Z))}{S\ (S\ ((S\ Z) + S\ Z))} \rightarrow S\ ((S\ Z) + (S\ Z)) \rightarrow S\ (S\ (S\ Z))$

Historie FP a příklad použití

- λ -kalkulus: 30. léta (teorie vyčíslitelnosti)
- Lisp 1959: s-výrazy
- Scheme 1975
- ML 1985: typovaný jazyk
- Haskell 1989
- Haskell 2010
- ...a další: Erlang (pro realtime apl.), Scala (kompiluje do JVM), Clojure, OCaml
- FFTW: knihovna FFT in the West - cena za numerickou matematiku 1999. V čem je napsána: no, (samozřejmě) v C. Ale generovaném pomocí OCaml.

Haskell

- <https://www.haskell.org> ; ekosystém
- - sw: Haskell Platform
- kompilátor GHC: Glasgow Haskell Compiler (GHCi, WinGHCi IDE)
- HUGs (WinHugs)

Literatura

- Graham Hutton, Programming in Haskell, Cambridge University Press 2007, Cambridge, Veľká Británie
- Richard Bird, Thinking Functionally with Haskell, Cambridge University Press 2014, Cambridge, Veľká Británie
- Bryan O'Sullivan, Don Stewart, John Goerzen, Real World Haskell, O'Reilly Media 2008,
<http://book.realworldhaskell.org/>

Charakteristiky Haskellu

- Statický typový systém (parametrický polymorfizmus, typové odvozování)
- Rekurzivní funkce, funkce vyšších řádů (f. jako param. i výsl.)
- Uživatelsky definované typy, i rekurzivní
- "Stručné" seznamy (list comprehensions)
- Líné vyhodnocování
- Čistý funkč. jazyk, referenční transparentnost, neměnné (immutable) dat.strukt.
- Monády, i pro V/V
- Odvozování a dokazování vlastností programů
- Operátory, Přetěžování (pomocí typových tříd), ...
- ... moduly, balíčky, paralelizmus

Základy práce

- REPL: Read-Eval-Print Loop, interpretační prostředí
- - lze i kompilovat (do .exe)
- program ve skriptu, obvyklá přípona .hs
- - příprava skriptu v textovém editoru
- `--` jednořádkové komentáře
- `{- vnořené víceřádkové komentáře -}`
- `Prelude.hs` - standardně natahovaný soubor, obsahuje předdefinované funkce

Prostředí

- `prompt >`
- `:quit` : ukončí prostředí
- `:? :help`
- `:load "myfile.hs"` : načtení souboru
- `:type map` : vypíše typ výrazu, čtyřtečka : :
výraz : : má typ
`map :: (a -> b) -> [a] -> [b]`
- `:set +t` : nastavení options, např. výpis typu

Hodnoty a typy

Příklady zápisu hodnot a jejich typů

- `5 :: Int` – celá čísla
- `1000000000000 :: Integer` – dlouhá čísla
- `3.0 :: Float` **nebo** `3.0 :: Double`
- `'a' :: Char` `'\t'`, `'\n'`, `'\\'`, `'\''`, `a "\""`
- `True :: Bool`, `False :: Bool` – v prelude
- `[1,2,3] :: [Int]`, **totéž** `1:(2:(3:[])) :: [Int]`
- `"abc" :: String` – řetězce, `String = [Char]`
- ***špatně:** `[2, 'b'] :: [?]` – kompilátor odmítne
- `(2, 'b') :: (Int, Char)` – dvojice, n-tice, `i () :: ()`
- `succ :: Int -> Int` – typ funkce, nepovinný při def. fce
- `succ n = n+1` – definice funkce

Typy

- Každý výraz má typ; nemusíme uvádět (typy funkcí), systém si typy (většinou) odvodí.
- `> ((read "5") :: Float) + 3`
- Konkrétní typy začínají **velkým** písmenem, typové prom. **malým**.
- Nekonzistentní typy: typová chyba, při překladu (tj. při `:load`)
- Haskell nemá implicitní přetypování, nutno explicitně
`fromInteger :: Num a => Integer -> a`
- Motivace pro zavedení typů:
 - ochrana proti některým druhům chyb
 - dokumentace
 - Ale: typy (v Hs) neodchytí speciální sémantiku hodnot:
`1/0, head []`

Příklady (i se seznamy)

```
-- elem :: (..) => a -> [a] -> Bool  -- typ, funkce
-- je [a1] prvek seznamu [a2]?
elem x []      = False
elem x (y:ys) = x==y || elem x ys  -- vs. Prolog

rev :: [a] -> [a]
rev xs        = rev1 xs []  -- akumulator
rev1 xs acc = if null xs then acc
              else rev1 (tail xs) (head xs :acc)

expR x e = -- rychlé umocňování  $x^e$ 
  if e == 0 then 1 else
  if e == 1 then x else
  if even e then expR (x*x) (div e 2)
  else x*expR x (e - 1)
```

Funkce

- Definice v souboru, tj. skriptu; nutno natáhnout
`:load/:reload`
- Funkce se aplikuje na *argumenty* a vrací *výsledek* (případně složený); aplikace je "selektor"
- Jména (alfanumerických) funkcí a proměnných začínají malým písmenem (konstruktory velkým)
- Definice funkce je "laciná"
→ mnoho krátkých funkcí, které skládáme a specializujeme
→ funkce píšeme tak, aby šly skládat a specializovat
- Currifikovaná forma (curried, podle Haskell B. Curry)
 - "argumenty se aplikují po jednom" - ale budeme mluvit o funkcích více proměnných
 - funkce dostane jeden argument a vrací funkci ve zbylých argumentech, $f(x, y) = f_x(y)$

Funkce 2

- `f :: Int -> Char -> Bool`
- funkční typ `->` je asociativní doprava:
`f :: Int -> (Char -> Bool)`
- necurifikovaná funkce `f' :: (Int, Char) -> Bool` volaná na dvojici, která nedovoluje **částečnou aplikaci**
- `> f 3 'a' ;` volání funkce, také `f 3`
- volání/aplikace je asociativní doleva: `(f 3) 'a'`, místo necurifikované `f' (3, 'a')`
- typování částečných aplikací sedí: `((f 3) :: (Char -> Bool)) ('a' :: Char)`
- konkrétní výskyt fce v programu může mít stejně, míň nebo víc arg. vůči definici (jen jediná def., na rozdíl od Prologu)
→ syntakticky, tj. závorkami, určíme aktuální počet arg.

Funkce 3

- Složené argumenty funkce jsou v závorkách.
- typové odvozování unifikací: z $f :: \underline{b} \rightarrow c$ a $x :: \underline{b}$ odvod' $f\ x :: c$
- typicky, výsledek funkce, tj. hodnota, se hned použije jako argument, případně *pojmenuje* lokálním jménem (tj. nepřirazuje se, nemáme příkazy)
proměnné reprezentují hodnoty (i složené), ne místa v paměti

Stručně vestavěné funkce

... stručně a zjednodušeně

- `Int`: typ celých čísel, `Integer`: dlouhá čísla
- běžné aritmetické funkce:
- `+, *, -, div, mod :: Int -> Int -> Int` – zj.
- `abs, negate :: Int -> Int`
- formálně: `(+) :: Num a => a -> a -> a`
- typ `Bool`, výsledky podmínek, stráží
- `== /= > >= <= < :: (..) => a -> a -> Bool` – zj.
- `(&&), (||) :: Bool -> Bool -> Bool` – binární and a or
- `not :: Bool -> Bool`

Syntax - layout

- V .hs souborech definice globálních funkcí začínají v 1. sloupci
- 2D layout, offside pravidlo: definice ze stejné skupiny začínají ve stejném sloupci
 - co začíná víc vpravo, patří do stejné definice jako minulý řádek
 - co začíná víc vlevo, ukončuje skupinu definic
 - aplikuje se na lokální definice (let, where), strážce, case
 - umožňuje vynechání oddelovačů “,” a závorek “{”
- Víc definic na jednom řádku:
`let {x=1;y=2} in ...`

Definice funkcí: if, stráže

- `even x = mod x 2 == 0` – jednořádková
- `abs1 x = if x>=0 then x else -x` – if-then-else
 - podmínka je výraz typu `Bool`
 - i.t.e. má vždy `else` větev: co vrátíte, když selže podmínka
- stráže (svislítko): výpočet ve FP je deterministický, bere se první výraz za rovnítkem, u kterého uspěje stráž `:: Bool` ;
otherwise ve významu `True`

```
nsd n m
  | m == 0      = n
  | n >= m      = nsd m (mod n m)
  | otherwise   = nsd m n
abs2 x = (if x>=0 then id else \y-> -y) x
id = \x -> x -- id x = x
```

Porovnávání se vzorem 1

- anglický termín: pattern matching
- Vyhodnocovaný výraz ve FP je bez proměnných (na rozdíl od Prologu), ale ne nutně vyhodnocený. V def. funkce na místě formálních parametrů může být nejen proměnná, ale i term, který vyjadřuje implicitní podmínku na tvar dat (po nezbytném vyhodnocení). Př. `ordered (x1:x2:xs) = ...`
- Aktuální parametry se musí dostatečně vyhodnotit, aby šla podmínka rozhodnout. Vyhodnotí se pouze potřebné části struktury, u složených typů. Jednoduché typy se vyhodnotí na konstantu.
- Lze použít i pro uživatelsky definované typy.
- P.m. pojmenuje složky aktuálních argumentů, proto (obvykle) nepotřebujeme selektory (resp. rozebírání struktury, např. `head`, `tail`)
- Jméno proměnné ve vzoru je definiční výskyt (může být jen jednou, na rozdíl od Prologu), prom. porovnáváme (`==`)

Porovnávání se vzorem 2

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

- `length1` je parametricky polymorfní (`[a] -> ...`), přijímá seznamy s lib. prvky `: a` ; zpracovává pouze strukturu seznamu, nikoli prvků (!typické pro FP)
- argument v druhé klauzuli je složený, proto jsou nutné závorky
- Neodmítnutelný vzor `"_"` (podtržítka): úspěje, nevyhodnocuje arg., může být víckrát

```
and1 :: Bool -> Bool -> Bool
and1 False _ = False
and1 _ x = x -- x se bude vyhodnocovat až v kontextu
and2 True True = True
and2 _ _ = False
```

- - `and1` je líné v 2. arg., `and2` vyhodnocuje někdy i 2. arg.

Porovnávání se vzorem 3

vzor @: přístup na celek i části

```
-- ordered :: [a] -> Bool
ordered (x:xs@(y:_)) = x<=y && ordered xs
ordered _             = True
```

- matching @ neselže, ale matching podčástí může selhat
- default s _ v příkladu se použije pouze pro [x] a []. Použití koncové klauzule na 2. místě nevadí (ve FP, na rozdíl od Prologu) pro LCO/TRO.
- implementačně: použití @ ušetří novou stavbu struktury v těle funkce (zde :)
- složené vzory musí být v závorkách, jinak se špatně naparsují (Př. sémantické chyby: length x:xs = ...)

Seznamy

- - jsou vestavěné, jako speciální syntax
- `data [a] = [] | a : [a]` – pseudokód
- Konstruktory (odvozené z typu):
- `[] :: [a]` – polymorfní konstanta
- `(:)` `:: a -> [a] -> [a]` – datový konstruktor
- syntax: `[1, 2, 3]` je `1:2:3:[]` asoc. doprava
tj. `1:(2:(3:[]))`
- *nelze `[1,2:xs]`, nutno `1:2:xs` pro seznamy s tělem `xs`
- hranaté (seznamové) závorky se používají pro další dva konstrukty - příště (stručné seznamy a generátory posl.)

map, filter, reverse

map a filter jsou funkce vyššího řádu

```
map :: (a->b) -> [a] -> [b]
map _ []          = []
map f (x:xs) = f x : map f xs
filter (a->Bool)->[a]->[a]
filter p [] = []
filter p (x:xs) = if p x then x:filter p xs
                  else    filter p xs

reverse xs = rev1 xs []
rev1 [] acc = acc
rev1 (x:xs) acc = rev1 xs (x:acc)

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

- reverse: rychlé, pomocí akumulátoru
- DC: `zip :: [a] -> [b] -> [(a,b)]`
- DC: `zipWith :: (a->b->c) -> [a] -> [b] -> [c]`

Lokální definice: let (a where)

- `let` tvoří výraz, možno vkládat do výrazů
- Definice v `let (a where)` jsou vzájemně rekurzivní, můžeme def. hodnoty i (lokální) funkce
- V `let` vlevo lze použít pattern matching

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

- typické použití `let`: lokální zapamatování hodnoty, když ji nebo její části potřebujeme víckrát použít
- `++` je append, spojení seznamů

Konstrukt where

- Speciální syntax, netvoří výraz; lze použít jen na vnější úrovni definice funkcí
- Definice ve where jsou vzájemně rekurzivní, můžeme def. hodnoty i funkce
- `qs/2`: quicksort, s parametrickým komparátorem `cmp/2` (!typické pro FP: -))
- `f: where` přes několik stráží, nelze pomocí `let`

```
qs _cmp [] = []
qs  cmp (x:xs) = qs cmp l ++ (x:qs cmp u)
  where
    (l,u) = split _cmp x xs
f x y
  | y > z = ...
  | y == z = ...
  | y < z = ...
  where z = x*x
```

Lexikální konvence 1

Alfanumerické identifikátory:

- posloupnosti písmen, číslíc, ' (apostrofu), _ (podtržítka)
- jména funkcí a proměnných: začínají malým písmenem nebo podtržítkem
- (velkým písmenem začínají konstruktory: True, Bool)
- vyhrazeno: klíčová slova: case of where let in if then else data type infix infixl infixr primitive class instance module default ...
- funkce se zapisují v prefixním zápisu: `mod 5 2`
- alfanumerický identifikátor jako (binární infixní) operátor, uzavření v ' (zpětný apostrof): `5 `mod` 2`

Lexikální konvence 2

Symbolové identifikátory:

- jeden nebo víc znaků: `:` `!` `*` `+` `-` `.` `<` `=` `>` `@` `^` `|` `/` `\` `&`
- speciální význam: `:` `~`
- symbolové konstruktory začínají znakem `:`
- standardní zápis: jako infixní operátor: `3+4`
- pro zápis v prefixním nebo neoperátorovém kontextu: v závorkách: `(+)` `3 4`, `map (+) [5,6]`, `(+) :: ...`
- sekce *(op)*, *(op arg)*, *(arg op)* jako částečně aplikované funkce, pro oba druhy identifikátorů:
$$(+) = \lambda x\ y \rightarrow x+y$$
$$(1/) = \lambda y \rightarrow 1/y$$
$$(\text{'div' } 2) = \lambda x \rightarrow x \text{'div' } 2$$

Literate Haskell

Je to varianta zdrojové syntaxe

Obvyklá přípona je `.lhs`

Za platný kód jsou považovány pouze řádky začínající znakem `'>'`,
všechno ostatní je komentář

Okolo řádků kódu musí být prázdné řádky

Příklad funkce `(++)`, tj. `append`

```
> (++) :: [a] -> [a] -> [a]
> []      ++ ys = ys
> (x:xs) ++ ys = x:(xs++ys)
```

Obvyklé použití: soubor jiného určení (blog v HTML, TeX) je také
platný Literate Haskell (po změně přípony na `.lhs`)

Standardní funkce z prelude 1

```
head :: [a] -> a
head (x:_) = x -- pro [] chyba
tail :: [a] -> [a]
tail (_:xs) = xs -- pro [] chyba
null :: [a] -> Bool
null xs = xs==[] -- nepoužívat if :- ( , ale bool. spojky
id :: a -> a
elem :: Eq a => a -> [a] -> Bool -- relace jako charakte-
ristická fce
elem x [] = False
elem x (y:ys) = x==y || elem x ys
(!!) :: [a] -> Int -> a -- n-tý od 0
fst :: (a,b) -> a
snd :: (a,b) -> b
(,) :: a -> b -> (a,b) -- spec. syntax
```


Standardní funkce 2

- `take n xs` - vrátí prvních `n` prvků z `xs` nebo všechny, když je jich málo
- chceme fci rozumně dodefinovat (pro `length xs < n`)
- ukázka použití selektorů: `(head, tail)` a stráží: `-`

```
take :: Int -> [a] -> [a]
take n xs =
  | n <= 0 || xs == [] = []
  | otherwise = head xs : take (n-1) (tail xs)
```

- `drop n xs` - stejný typ, zahodí prvních `n` prvků a vrátí zbytek (i `[]`)
- `takeWhile p xs` - vrátí nejdelší úvodní úsek, kde pro prvky platí podmínka `p`; porovnejte typy `take` a `takeWhile`

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

Standardní funkce 3

- map, filter, (bude: foldr, unfold)
- zipWith f xs ys - paralelní zpracování 2 seznamů xs a ys fcí f
- zipWith/3 je polymorfní, protože prvky vstupních seznamů zpracovává pouze funkcionální parametr $f/2$. (!typické pro FP)

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

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

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

```
-- totéž:
```

```
zip xs ys = zipWith (,) xs ys -- stejné posl. arg. lze vypustit
```

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

- zip - jednořádková definice získaná specializací, !typické pro FP

Generátory posloupností

- speciální syntax pro generování aritmetických posloupností, dvě tečky
- převádí se na volání funkcí z typové třídy Enum → jde použít i pro uživatelské typy
- jiné posloupnosti než aritmetické získáme transformací (např. map)
- první dva členy určují "krok", generuje i nekonečné seznamy
 - `[1..5] ~> [1,2,3,4,5]`
 - `[1,3..10] ~> [1,3,5,7,9]`
 - `[1..] ~> [1,2,3,4,5 ...]`
 - `[1,3..] ~> [1,3,5,7,9,11 ...]`
 - `[5,4..1] ~> [5,4,3,2,1]`

Stručné seznamy 1

- list comprehensions
- motivace z teorie množin: $\{x^2 - y^2 \mid x, y \in \{1..9\} \wedge x > y\}$
- vybíráme prvky ze seznamů a pro každý úspěšný zkombinovaný výběr generujeme 1 prvek výstupního seznamu
- před svislítkem: hodnota, která se vygeneruje na výstup
- za svislítkem, oddělené čárkami:

❶ generátory $x \leftarrow xs$, vyhodnocované zleva

❷ testy/stráže $:: \text{Bool}$,

❸ let $mez = 100$

```
[f x | x <- xs] -- map f xs, generátor x <- xs  
[x | x <- xs, p x] -- filter p xs, test p x  
[(x, kv) | x <- xs, let kv = x * x] -- let
```

Stručné seznamy 2: příklady

```
kart :: [a] -> [b] -> [(a,b)]
kart xs ys = [(x,y)|x<-xs, y<-ys]
concat :: [[a]] -> [a]
concat xss = [x|x<-xss, x<-xs]
length xs = sum [1| _<-xs] -- nepoužijeme hodnotu
klíce pary = [klic|(klic,hodn)<-pary] -- pattern matching
delitele n = [d| d<-[1..n], n`mod`d==0]
prvocislo n = delitele n == [1,n]
horniTrojuh n=[(i,j)|i<-[1..n], j<-[i..n]] -- použijeme i
```

```
> kart [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```

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

Příklad: quicksort

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

```
qs1 :: (a->a->Bool) -> [a] -> [a]
qs1 cmp xs = qs xs
  where qs [] = []
        qs (x:xs) = qs [y | y<-xs, y `cmp` x] ++ [x] ++
                    qs [y | y<-xs, not (y `cmp` x)]
> qs1 (<) [9,7..0]
```

- qs1 parametrizovaný uspořádáním cmp, obvyklé v FP: funkce píšeme co nejobecněji (funkc. param. jsou nejobecnější způsob předání arg.)
- qs1 je polymorfní, univerzální (třídí i reverzně, podle 1./2. složky ...)
- relace reprezentována jako charakteristická fce :: ... -> Bool
- "schovaný" parametr cmp v rekurzi

Seznamy výsledků

- Programátorský idiom, nemáme backtracking (na rozdíl od Prologu), pracujeme se všemi výsledky v seznamu a vydáváme jeden seznam všech výsledků
- díky línému vyhodnocování se seznam (při dobré implementaci) postupně generuje a zpracovává, tj. nemáme v paměti celou strukturu najednou. Příklad: `length (komb 6 [1..11])`
- Příklad: kombinace, tj. seznam všech kombinací

```
komb 0 ps = [[]]
```

```
komb _ [] = []
```

```
komb k (p:ps) = [p:k1 | k1 <- komb (k-1) ps] ++ komb k ps
```

- Příklad: vrácení prvku a zbytku seznamu všemi způsoby (bez ==)

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

```
vyber [] = []
```

```
vyber (x:xs) = (x,xs) : [(y,x:ys) | (y,ys) <- vyber xs]
```

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

```
[(1,[2,3]), (2,[1,3]), (3,[1,2])]
```

Generuj a testuj

- Programy typu "generuj a testuj" se lehce *píší* (včetně NP-úplných), někdy s použitím stručných seznamů
- výstup může být seznam výsledků, případně zpracovaný (minimum), existence výsledku ...
- PŘ: přesný součet podmnožiny (mírně optimalizováno):

`batoh1 b xs` platí, pokud existuje $I \subset xs$, tž. $\sum_{x \in I} x = b$.

```
batoh1 :: Int -> [Int] -> Bool
```

```
batoh1 0 _ = True
```

```
batoh1 _ [] = False
```

```
batoh1 b (x:xs) = b > 0 && (batoh1 (b-x) xs || batoh1 b xs)
```

```
batoh2 :: (Num a, Ord a) => a -> [a] -> [a] -> [[a]]
```

```
batoh2 0 _ acc = [acc]
```

```
batoh2 _ [] acc = []
```

```
batoh2 b (x:xs) acc = if b <= 0 then []
```

```
    else batoh2 (b-x) xs (x:acc) ++ batoh2 b xs acc
```


Kartézský součin víc seznamů

- Trade-off čas vs. paměť

```
kartn1, kartn2 :: [[a]] -> [[a]]
kartn1 [] = [[]]
kartn1 (xs:xss) = [x:ks|x<-xs, ks<-kartn1 xss]
kartn2 [] = [[]]
kartn2 (xs:xss) = [x:ks|ks<-kartn2 xss, x<-xs]
kartn3 [] = [[]]
kartn3 (xs:xss) = [x:ks|let kk=kartn3 xss,x<-xs,ks<-kk]
> kartn1 [[1,2],[3],[5,6]]
[[1,3,5],[1,3,6],[2,3,5],[2,3,6]]
```

- Chování: kartn1 generuje (stejně) ks opakovaně, jako při prohledávání stavového prostoru do hloubky.
- Fce kartn2 vygeneruje *velkou* d.s. obsahující ks jednou. Pokud si ji potřebuje pamatovat, tak je to (až neúnosně) paměťově náročné.
- Fce kartn3 generuje d.s. kk jednou a pamatuje si ji.

Operátory, deklarace

- Operátory jsou syntaktický cukr, pro přehlednost a pohodlí zápisu
- Haskell má pouze binární operátory, priority 9-0 (0 nejnižší)
- Vyšší priorita váže víc, jako v matematice
- Aplikace funkce váže nejvíc, proto musí být složené argumenty v závorkách (i při porovnávání se vzorem)
- př.: `fact 3+4` vs. `fact (3+4)`
- Lze použít i pro alfanumerické operátory

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

Operátory

- funkční volání váže těsněji než nejvyšší priorita 9
- 9,doleva `!!` – n-tý prvek, `(mat!!1)!!2`
- 9,doprava `.` – tečka, skládání funkcí
- 8,doprava `^^^` `**`
- 7,doleva `*` `/` ``div`` ``mod`` ``rem`` ``quot``
- 6,doleva `+` `-` `-i` unární `-`
- 5,doprava `:` `++` `-1:(2:[])`
- 5,neassoc `\\` – delete
- 4,neassoc `==` `/=` `<` `<=` `>` `>=` ``elem`` ``notElem``
- 3,doprava `&&` – doprava vhodnější pro líné vyh.
- 2,doprava `||`

Na asociativitu záleží, pokud jsou arg. různého typu (`!!` `:` `elem`)

Uživatelské typy, nerekurzivní

- Klíčové slovo `data`, vlevo od `=` typ (s parametry), vpravo datové konstruktory (s parametry), jednotlivé varianty oddělené svislítkem `|`

```
data Bool = False | True -- z prelude
data Ordering = LT | EQ | GT
data Point a = Pt a a -- polymorfní
data Maybe a = Nothing | Just a -- pro chyby
data Either a b = Left a | Right b -- sjednocení typů
```

- `Bool`, `Ordering`, `Point` ... jsou jména typových konstruktorů
- `False`, `LT`, `Pt`, ... jsou jména datových konstruktorů
- Vyhodnocený tvar výrazu obsahuje (pouze) datové konstruktory, vyhodnocené datové konstruktory mohou obsahovat nevyhodnocené argumenty
- Datové konstruktory jsou funkce (odvozené z typu) a mají svůj (i polymorfní) typ: `True :: Bool`, `Just :: a -> Maybe a`

Uživatelské typy 2

- Datové konstruktory generují výraz určitého typu. D.k. nejsou přetížené (nelze overloading), ale mohou být polymorfní (př. `Just`, `Left`)
- Každá varianta typu má dat. konstruktor (`Either`)
- D.k. pomáhají typovému systému při odvozování typů a slouží k rozlišení variant hodnoty (jako tagy)
- Porovnávat se vzorem lze i uživatelské typy

```
jePocatek (Pt 0 0) = True
jePocatek _         = False
unJust (Just x) = x
```

- Konkrétní hodnoty jsou konkrétního typu:

```
Pt 1 2 :: Point Int
Pt "ab" "ba" :: Point String
```

Typ Maybe

- Typ `Maybe` a slouží na přidání speciální hodnoty `Nothing` k typu daném parametrem `a`
- Hodnota `Nothing` je mimo typ `a` a proto typ `a` můžeme používat celý (tj. `Nothing` nezabírá místo)
- Typ `Maybe` je polymorfní a proto jedna funkce slouží všem typům
- Typické použití: `lookup`, je polymorf. a pro `[]` mám co vrátet
`:: Maybe b`

```
mapMaybe :: (a->b) -> Maybe a -> Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup _ [] = Nothing
lookup k ((h,v):xs)
  | k == h      = Just v -- zabalení v
  | otherwise = lookup k xs
```

Typ Either

- Umožňuje disjunktí sjednocení typů
- "map" je jiné, potřebuje 2 fnc. arg.
- swap vyměňuje složky
- left aplikuje funkci pouze na levé složky
- swap a left jsou "pomocné" funkce

```
mapEither (a->b)->(c->d)->Either a c -> Either b d
mapEither f g (Left x)=Left(f x)
mapEither f g (Right y)=Right(g y)
swap :: Either a b -> Either b a
swap (Left x) = Right x
swap (Right y)= Left y
left f ei = mapEither f id ei
```

```
> map (zpJablka `mapEither` zpHrusky) es
```

- funkcionální parametr zkonstruuji z jednodušších funkcí (bez λ -funkce)

(Pojmenované položky)

- Dosud: položky podle polohy

```
data PointF = Pt Float Float
pointx :: PointF -> Float
pointx (Pt x _) -> x
```

- Hs98/Hs2010: pojmenované položky, jiná syntax pattern matchingu

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


Rekurzivní uživ. typy

```
data Nat = Z | S Nat -- monomorfní, numerály
data Tree a = Leaf a -- polymorfní
              | Branch (Tree a) (Tree a)
data Tree2 a = Void
              | Node (Tree2 a) a (Tree2 a)
data Tree3 a b = Leaf3 b
                | Branch3 (Tree3 a b) a (Tree3 a b)
data NTree a = Tr a [NTree a] -- n-ární stromy
```

- různé typy stromů pro různé použití (Tree2 pro BVS, NTree pro n-ární)
- omezení (na regulární typy): konstruktory typu na pravé straně definice mají stejné argumenty jako na (definující) levé straně

Rekurzivní typy

- pozorování: rekurzivní typy se dobře zpracovávají rekurzivními funkcemi
- přístup na složky: (zatím) porovnání se vzorem, (uživatелеm definované) selektory jako `leftSub`

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

ozdobT2 :: Int -> Tree2 a -> (Int, Tree2 (Int, a))
ozdobT2 i Void = (i, Void)
ozdobT2 i (Node l x r) = (iR, Node tL (iL, x) tR)
  where (iL, tL) = ozdobT2 i l
        (iR, tR) = ozdobT2 (iL+1) r
test = Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
```

- `ozdobT2`: předávání stavu, `test`: testovací data

Typ NTree

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

- nepřímá rekurze při definici typu
- struktura končí prázdným seznamem, nemá konstruktor pro ukončení rekurze
- typ konstruktoru: `Tr :: a -> [NTree a] -> NTree a`
- typické zpracování: 2 vzájemně rekurzivní funkce, jedna pro stromy, druhá pro seznam stromů

```
hloubkaNT (Tr x ts) = 1+maximum (hloubky ts)
  where hloubky [] = [0]  -- zarážka pro maximum
        hloubky ts = map hloubkaNT ts
```

Typová synonyma 1

- klíčové slovo `type`
- na pravé straně od '=' jsou jména typů
- neobsahuje datový konstruktor
- systém při výpisech `t.synonyma` nepoužívá, vrací rozepsané typy

```
type RGB = (Int,Int,Int)
type Complex = (Double,Double)
type Matice a = [[a]]
```

Typová synonyma 2

- klíčové slovo `newtype`
- definování nekompatibilního typu stejné struktury (typový systém nedovolí typy míchat, např. `Int` a `Euro`)
- datový konstruktor na pravé straně má právě jeden argument, a to původní typ
- typový konstruktor `Euro` vlevo a datový konstruktor `Euro` vpravo jsou v různých namespace
- datový konstruktor se používá pouze při kompilaci, run-time je bez overheadu
- časté použití: nová verze typu, která má jiné standardní operace, např. `== a >`

```
newtype Euro = Euro Int
plusEu :: Euro -> Euro -> Euro -- přetížení + později
Euro x `plusEu` Euro y = Euro (x+y)
```

...

- obecnější pomocná funkce:

```
lift2Eu :: (Int->Int->Int) -> Euro->Euro->Euro
lift2Eu op (Euro x) (Euro y) = Euro (x `op` y)
plusEu = lift2Eu (+)
```

- fce `lift2Eu` je analogická fci `zipWith` pro (nerekurzivní) typ `Euro`

Case

- výraz `case` je základní metoda pro rozlišování konstruktorů
- výraz `case` je obecné porovnávání se vzorem, použitelné jako výraz, i pro uživatelské typy
- aktivuje se deterministicky první vzor, který vyhovuje
- používá 2D layout (nebo `{}` a `;`)
- lze použít také pro 1 zpracováváný výraz (bez závorek okolo)

```
case (vyraz,...) of  
  vzor _-> vyraz  
  vzor2 -> vyraz2  
  ...
```

Case: příklad

- `take2 n xs` vybírá prvních `n` prvků z `xs`, pokud existují, jinak všechny
- na nejvyšší úrovni definice funkce se typicky používají pattern matching místo `case`, viz `take`

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

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


Funkce 1

- funkce se dají "za běhu" konstruovat lambda-abstrakcí
- formální parametry mají rozsah platnosti tělo definice

```
succ x = x+1 -- obvyklý zápis
succ = \x -> x+1 -- ekvivalentní
add = \x y -> x+y -- víc parametrů
add = \x -> \y -> x+y
```

- aplikace funkce (mezerou) je (jediný) selektor funkce. Hodnotu funkce lze zjistit v jednotlivých bodech.
- Anonymní funkce se často používají jako argumenty funkcí vyšších řádů.
- *referenční transparentnost*: volání funkce na stejných parametrech vrátí stejnou hodnotu. Protože nemáme globální proměnné, přiřazení a sideefekty.
- Následně výsledek nezávisí na pořadí vyhodnocování a proto je možné líné vyhodnocování

Funkce 2

- na funkcích není definována rovnost (`==`)
- skládání funkcí, identita `id` jako neutrální prvek pro skládání

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

- platí `id.f = f = f.id`
- aplikace, pro pohodlnější zápis

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

- `$` je asoc. doprava: `f3 $ f2 $ f1 x`

Funkce 3

- definice funkcí lze psát bez posledních argumentů na obou stranách; typový systém si poradí; tzv. bezbodový zápis (v teorii η -redukce $\lambda x.Fx = F$, "eta"-r.)
- typicky používáme při specializaci funkcí \rightarrow proto funkcionální argumenty chceme jako první
- zafixování argumentu neznamena optimalizaci, funkce čeká na všechny své argumenty a až potom se spustí

```
zip = zipWith (,)
odd :: Int -> Bool
odd = not . even -- bezbodový zápis, bez x
```

- př.: fce `filter` s opačnou podmínkou:

```
negFilter f = filter (not.f)
```

Funkce 4

- transformace funkcí: někdy potřebujeme upravit funkce pro plynulé použití
- přehození dvou argumentů, `flip`, v prelude

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

- funkce `curry` a `uncurry`, v prelude

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

Funkce 5

- př: data jsou dvojice, ale funkce očekává samostatné argumenty
→ přizpůsobíme funkci

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

- *Closure*, uzávěr: částečně aplikovaná funkce si zapamatuje svoje argumenty z lexikálního rozsahu platnosti, i když ji předáte mimo lexikální rozsah
- nejsou problémy s dealokací proměnných, používané nelokální proměnné jsou dostupné z closure

```
const :: a -> b -> a
const x y = x
let z = ..., cF = const z in cF
```

- Díky closure se hodnota `z` uvnitř `cF` zachová a může se použít i mimo blok, kde byla definována.

Nekonečné d.s. 1

- líné vyhodnocování vyhodnocuje pouze to, co je potřeba, shora (a jen jednou): pro výpis, pro pattern matching, pro arg. vestavěných funkcí ...
- líné vyhodnocování umožňuje potenciálně nekonečné datové struktury
- použije se jen konečná část (anebo přeteče paměť)
- n.s. má konečnou reprezentaci v paměti

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

Nekonečné d.s. 2

- Vytvoření nekon. struktur, z prelude

```
repeat x = x: repeat x
```

```
cycle xs = xs ++ cycle xs
```

```
iterate f x = x:iterate f (f x)
```

```
jednMat = iterate (0:) (1:repeat 0)
```

- např. při sčítání `jednMat` s konečnou maticí pomocí `zipWith` se vygeneruje a použije pouze konečná část

```
plusMat m1 m2 = zipWith (zipWith (+)) m1 m2  
                ... (\r1 r2-> zipWith (+) r1 r2) ...
```

Nekonečné struktury a funkce

- psát funkce kompatibilně s líným vyhodnocováním:
 - mít funkce které odeberou pouze potřebnou část d.s. (`take`, `takeWhile` ...), tj. nestriktní funkce
 - mít funkce, které zpracují nekon. d.s. na nekon. d.s., tj. které na základě části vstupu vydají část výstupu. (`map`, `zipWith`, `filter` ...). Ale `reverse` takto nefunguje.
- def: *striktní funkce* vyhodnotí vždy svůj argument (úplně)
- pro striktní funkce platí: $f \perp = \perp$ (nedefinovaná hodnota, bottom)
- jazyk s hladovým/dychtivým vyhodnocováním (eager evaluation), např. Scheme, dovoluje psát striktní funkce; ale líné vyhodnocování (lazy eval.) umožňuje psát i nestriktní fce, např. vlastní "řídící" struktury (např. `maybeif`)
- `past`: `[x | x <- [1..], x < 4]`
- nedá očekávaný výsledek `[1,2,3]`, ale `1:2:3:⊥`, tj. cyklí; systém neodvozuje "limitní" chování

Líné vyhodnocování

- Líné vyhodnocování umožňuje nekonečné struktury, ale:
- Nevýhoda líného vyhodnocování: *memory leak*, *únik paměti*: nespočítání výrazu a neuvolnění paměti, dokud hodnota není potřeba
- očekáváte číslo, ale v paměti se hromadí nevyhodnocený výraz
- (Hackerský koutek: uživatel může požádat o striktní vyhodnocení `x` pomocí `$! : př. \mathbb{f} $\$!$ x .`)

Příklady

- Pascalův trojúhelník (po uhlopříčkách) ; idiom FP: zpracování celých struktur

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

- Stavové programování: návrhový vzor *iterátor* funkcí `until`
- ve FP lze (někdy) napsat kód s funkcionálními parametry vs. pseudokód v OOP

```
until2 done next init =
  head (dropWhile (not.done) -- mezivýsl. se průběžně
        (iterate next init) ) -- zahazují
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f = go where -- z knih., paměť. 2,8x lepší, čas 1,1x
  go x | p x          = x
       | otherwise    = go (f x)
```

- vrací celý interní stav (se kterým se počítá), nemá výstupní projekci
- protože se mezivýsledky průběžně zahazují, nemám v paměti celý seznam najednou

Polymorfní typy

- Pro odvozování typů se používá Hindleyův-Milnerův algoritmus, na části polymorfního λ -kalkulu.
- T: Funkce má v polymorfním λ -kalkulu jeden nejobecnější typ.
- Typ je polymorfní nebo monomorfní. Další typy získáme z nejobecnějšího polymorfního typu substitucí.
- Odvozování typů: unifikací. Pro funkci $f : a \rightarrow b$ a výraz $x : a'$ se odvodí substitute $\sigma = mgu(a, a')$ a typ výrazu $f \ x : b\sigma$
- Pragmaticky: při odvozování se zjistí nekonzistence, ne místo nebo důvod chyby. Příklad: při neúspěšné unif. typů $t1 = [(a, b)]$ a $t2 = ([c], d)$ chybí head na $t1$ nebo fst na $t2$?
- V Haskellu se používají typová rozšíření. Někdy musí uživatel typovému systému napovědět (tj. explicitně uvést typ). Pak se *typové odvozování* (type inference) mění na jednodušší *typovou kontrolu* (type checking) `read "1" :: Int, const (1 :: Int)`
- Typy se používají pouze při kompilaci. Běh je bez overheadu.

Polymorfizmus a typové třídy

- 1. Parametrický p., pomocí typových proměnných. Př:
`length :: [a] -> Int`
- Na typu argumentu nezáleží. Stejná impl. pro všechny typy. Kód 1x a funguje i pro budoucí typy.
- 2. Podtypový p. (přetížení, overloading), pomocí typových tříd.
Př: `(==) :: Eq a => a -> a -> Bool`
- Na typu argumentu záleží. Různé implementace pro různé typy, jedna instance pro jeden typ. Pro nové typy nutno přidat kód.
- Porovnání: V Hs funkce mají typový kontext. V OOP objekty mají TVM-tabulku virtuálních metod.
- Hs podle (odvozeného) typu argumentu (správně) určí, kterou instanci typové třídy použít. Případně chyba: 'žádná' nebo 'nejednoznačná' třída.

Typové třídy

- Ne všechny operace jsou definovány na všech typech. Typová třída je abstrakce těch typů, které mají definovány dané operace.
- T.třídy: `Eq Ord Show Read Enum Num Integral Fractional Float Ix ...`
- Tento mechanismus odpovídá přetížení, tj. ad hoc polymorfizmu.
- class - zavedení typové třídy
- instance - definování typu jako prvku typové třídy, spolu s definováním operací.
- Typový kontext funkce: podmínky na typy, tj. na typ. proměnné
- PŘ: funkci `eqpair` lze použít pouze pro typy, na kterých je definována rovnost. Funkce není přetížená (tj. kompilace 1x), pouze využívá přetížení funkce `(==)` z typové třídy.

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

Přetížení - detaily

- Přetížené konstanty:

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

- implementace využívá `fromInteger`
- analogie k `[]::[a]`
- Nastavení přepínače `t`, aby Haskell vypisoval typy

```
> :set +t
```

- Chybné použití `(+)` na typ `Char`

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

- Pro jeden typ lze mít pouze jednu instanci. Ale pomocí `newtype` lze získat izomorfní typ pro novou instanci.

Typové třídy, Eq

- Deklarace typové třídy (class) obsahuje signaturu, tj. jména a typy funkcí. Tyto funkce budou přetížené.
- Dále může obsahovat defaultní definice některých funkcí. Ty se použijí, pokud nejsou předdefinovány.
- Třída Eq: typy, které lze porovnávat (na rovnost). Např. Bool, Int, Integer, Float, Char. Taky seznamy a n-tice, pokud položky jsou instance Eq.

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

- typ rovnosti: `(==) :: Eq a => a -> a -> Bool`
- při použití `==` na typ `a` musí být pro typ `a` definována instance třídy `Eq`.

Instance

- pro vestavěný typ

```
instance Eq Int where
    x==y = intEq x y
```

- pro uživatelský neparametrický typ: def. rozpisem

```
instance Eq Bool where
    False==False = True
    True ==True  = True
    _      == _   = False
```

- pro parametrický uživ. typ , tj. typový konstruktor; typ prvků je instance Eq

```
instance (Eq a)=> Eq (Tree a) where
    Leaf a      == Leaf b      = a==b
    Branch l1 r1 == Branch l2 r2 = l1==l2 && r1==r2
    _           == _           = False
```


Eq 2

- víc podmínek v typovém kontextu

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

- první rovnost je na dvojicích, druhá na typu *a*, třetí na typu *b*

Deriving

- Některé typové třídy, `Eq`, `Ord`, `Show`, `Read`, `Enum` si lze nechat odvodit při definici nového typu
- Používá se klíčové slovo `deriving`
- Vytvoří se standardní definice: rovnost konstruktorů a složek, uspořádání podle definic konstruktorů v definici typu a dále lexikograficky, v `Read` a `Show` konstruktory identickými řetězci

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

Třída Ord

- Hodnoty typu jsou lineárně uspořádané
- Musíme mít pro typ už definovanou rovnost, typ je instance Eq

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

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

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

Třída Ord 2

- Instance Ord pro typy a typové konstruktory.

```
instance Ord Bool where
  True  <= False = False
  _      <= _      = True
```

```
instance (Ord a, Ord b) => Ord (a,b) where
  (a1,b1) <= (a2,b2) = a1<a2 || a1==a2 && b1<=b2
```

```
instance (Eq [a], Ord a) => Ord [a] where
  []      <= _ = True -- lexikograficky
  (x:xs) <= (y:ys) = x<y || x==y && xs<=ys
  _      <= _ = False
```

- Pro jeden typ pouze jedno uspořádání se jménem <=. Jiné třídění lze definovat na novém typu s využitím `newtype`.

Třídy Show, Read, Enum

- Třída Show a: hodnoty typu a lze převést na znakové řetězce. (Pouze převod na String, vlastní výstup je samostatně.)

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

- Třída Read a: hodnoty typu a lze převést ze znakového řetězce.

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

- při použití `read` je často nutno uvést typ: `read "1" :: Int`
- Třída Enum a: výčtové typy - dovoluje používat speciální syntax pro typ a

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

Čísla

- Třída Num: potřebuje Eq a Show, instance pro Int, Integer, Float ...

```
(+), (-), (*) :: a->a->a  
abs, negate, signum :: a->a  
fromInt :: Int -> a -- převod ze standardních čísel  
fromInteger :: Integer -> a
```

- Třída Integral a: potřebuje Num, instance Int, Integer

```
div, mod, quot, rem :: a->a->a  
toInteger :: a-> Integer
```

- Třída Fractional a: potřebuje Num, hodnoty jsou neceločíselné.
Instance: Float, Double (a přesné zlomky Ratio a).

- Třída Floating a: potřebuje Fractional, instance: Float, Double

```
exp, log, sin, cos, sqrt :: a->a  
(**) :: a->a->a -- umocňování
```

- Třída Bounded a

```
minbound, maxbound: a
```

- Třída Ix a: pro indexy polí

```
range :: (a,a) -> [a]  
index :: (a,a) -> a -> Int
```

Př: zbytkové třídy mod 17 (neúplné)

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

- Motivace: FFT napsaná pomocí + - * (a fromInt) funguje v \mathbb{C} (Data.Complex) i \mathbb{Z}_{17} (Z17)
- tj. stejný kód parametrizovaný operacemi z typových tříd funguje pro různé typy

Třída Functor

- Třída pro typový konstruktore (ne pro konkrétní typ), na kterém lze definovat analogii `map`: struktura typu zůstane, jednotlivé prvky se změní podle funkcionálního parametru `fmap`.

```
class Functor a where
  fmap :: (b->c) -> a b -> a c

instance Functor [] where
  fmap f xs = map f xs
instance Functor Tree2 where
  fmap _f Void = Void
  fmap f (Node l x r) = Node (fmap f l) (f x)
                        (fmap f r)
instance Functor Maybe where
  fmap f x = mapMaybe f x
instance Functor ((->) r) where
  fmap = (.) -- (a->b)->(r->a)->(r->b)
```

Strukturální rekurze pro seznamy: foldr

- Funkce foldr (svinutí) pro seznamy, doprava rekurzivní
- Nahrazuje konstruktory funkcemi, výsledek je lib. typu (vs. map, filter)
- **Př:** `foldr f z (1:2:3:[])` počítá `1 `f` (2 `f` (3 `f` z))`

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

```
foldr f z [] = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

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

- někdy potřebujeme výslednou hodnotu dodatečně zpracovat (prumer), někdy potřebujeme jiný druh rekurze (maximum)

Použití foldr ($12x + 1x$)

```

sum xs = foldr (\x s->x+s) 0 xs
product xs = foldr (*) 1 xs
faktorial n = product [1..n]
reverse xs = foldr (\x rs->rs++[x]) [] xs -- v  $O(n^2)$ 
concat xss = foldr (++) [] xss
xs ++ ys = foldr (:) ys xs
map f xs = foldr (\x ys->f x:ys) [] xs {- ((:).f)
iSort cmp xs = foldr (insert cmp) [] xs -- vs. jiné sort
  where insert = ...
and xs = foldr (&&) True xs
  Funkce and "zdědí" líné vyhodnocování (zleva) od &&
or xs = foldr (||) False xs
any p xs = foldr (\x b->p x||b) False xs -- duálně: all
all p xs = foldr (\x b->p x&& b) True xs
prumer xs = s/fromInt n where
  (s,n) = foldr (\x (s1,n1) -> (x+s1,1+n1)) (0,0) xs

```

Složenou hodnotu potřebujeme postzpracovat.

Varianty fold

- Varianty: na neprázdných seznamech `foldr1`; doleva rekurzivní `foldl` (`a foldl1`)

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

- Zpracování prvků zleva, vlastně pomocí akumulátoru, pro konečné seznamy

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

Ještě k fold

- V GHC v.8 je `foldr` a varianty definováno obecněji:
- Typ/typový konstruktor `t` považujeme za kontejner a přidáváme prvky postupně do výsledné hodnoty typu `b`.

```
foldr :: Foldable t => (a->b->b)->b-> t a ->b
```

- Zpracování předpon a přípon: `scanr`, `scanl`. "fold" vydával pouze konečný výsledek, "scan" vydává seznam mezivýsledků (jiný druh rekurze)

```
scanr :: (a->b->b)->b-> [a] -> [b]
scanl :: (a->b->a)->a-> [b] -> [a]
> foldr (:) [0] [1,2,3]
[1,2,3,0] :: [Integer]
> scanr (:) [0] [1,2,3] -- zpracování přípon
[[1,2,3,0],[2,3,0],[3,0],[0]] :: [[Integer]]
> scanl (flip (:)) [0] [1,2,3] -- zpracování předpon
[[0],[1,0],[2,1,0],[3,2,1,0]] :: [[Integer]]
```

Obecná rekurze: stromy

- Myšlenka nahrazovat konstruktory d.s. uživatelskými funkcemi jde použít i pro jiné struktury.

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

- Nahrazuju konstruktory `Leaf :: a->Tree a` a `Branch`, kde typ `b` je místo `Tree a`

```
foldT fL fB (Leaf a) = fL a
```

```
foldT fL fB (Branch l r) = fB (foldT fL fB l)
                             (foldT fL fB r)
```

```
hloubka t = foldT(\_>1)(\x y -> 1+max x y) t
```

```
sizeT t = foldT(\x->size x)((+).(1+))t --\x y->1+x+y
```

```
mergeSort :: Ord a => Tree a -> [a] -- fáze 2 mergesortu
```

```
mergeSort t = foldT (\x->[x]) merge t
```

- Mergesort používá strukturální rekurzi podle stromu bez vnitřních vrcholů (`Tree a`)

... i pro nerekurzivní typy

- Myšlenka nahrazovat konstruktory d.s. uživatelskými funkcemi jde použít i pro nerekurzivní typy: zpracování hodnoty zabalené v `Maybe`.

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing  = n
maybe _ f (Just x) = f x
```

```
> maybe 0 (*2) (readMaybe "5")
10
> maybe 0 (*2) (readMaybe "")  -- při chybě default 0
0
```

- **typ výstupu** `maybe` je libovolný typ `b`, pro porovnání u `fmap` je to `Maybe b`

Unfold (myšlenka), pro seznamy

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

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

- Převod čísla do binární repr., od nejméně významných bitů

```
int2bin n = unfold (0==) (\x->(x`mod`2,x`div`2)) n
> int2bin 11 = 1:i 5=1:1:i 2=1:1:0:i 1=1:1:0:1:i 0=
  1:1:0:1:[]
```

- Pro struktury, pokud má datový typ víc konstruktorů, použijeme Either – případně opakovaně.

```
unfoldT2 :: (b->Either () (b,a,b)) -> b -> Tree2 a
```


Unfoldr, z knih. Data.List

- Pro seznamy můžeme použít `Maybe` pro analýzu hodnoty.

```
unfoldr :: (b->Maybe(a,b)) -> b -> [a]
unfoldr f b0 =
    let go b = case f b of
                    Just (a,b1) -> a : go b1
                    Nothing      -> []
    in go b0
```

- Příklady použití, `b` je stav

```
> unfoldr (\b -> if b == 0 then Nothing
                  else Just (b, b-1)) 10
[10,9,8,7,6,5,4,3,2,1]
iterate f == unfoldr (\x -> Just (x, f x))
selectSort cmp xs = unfoldr selectMin xs
    where selectMin []      = Nothing
          selectMin (x:xs) = Just (selectMin' x xs)
...
◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ↺ ↻
```

Počítání s programy, rovnostní odvozování

- Typické použití odvozování při dokazování vlastností programu:
 - 1 pro částečnou správnost vůči specifikaci (a požadavkům),
 - 2 optimalizaci/efektivitu.
- V bezstavovém programování se s programy lehce počítá. (Čistý λ -kalkul je rovnostní teorie.)
- Příklad: asociativita ($++$), pro konečné seznamy x, y, z .
- Chceme dokázat: $x++(y++z) = (x++y)++z$, levá strana je efektivnější
- Rozbor případů pro $x=[]$ a $x=a:v$. Podtržená část se upravuje:

$$\begin{aligned}
 & \frac{[] ++ (y++z)}{=} \{ \text{def } ++.1 \} \\
 & \quad \underline{y++z} \\
 & = \{ \text{def } ++.1 \} \\
 & \quad ([] ++ y) ++ z
 \end{aligned}$$

$$\begin{aligned}
& \frac{(a:v)++(y++z)}{=} \{ \text{def } ++.2 \} \\
& \quad a : \frac{(v++(y++z))}{=} \{ \text{indukční předpoklad} \} \\
& \quad \frac{a : ((v++y)++z)}{=} \{ \text{def } ++.2 \} \\
& \quad \frac{(a : (v++y))++z}{=} \{ \text{def } ++.2 \} \\
& \quad ((a:v)++y)++z
\end{aligned}$$

- DC: $(\text{map } f.\text{map } g) \ x = \text{map } (f.g) \ x$, pravá strana je efektivnější, protože netvoří mezilehlý seznam
- Pozn. Např. pro `fmap` z `Functor` má (podle teorie) platit
 - 1 `fmap id = id`
 - 2 `fmap f . fmap g = fmap (f.g)`

Haskell takové rovnosti neumí ověřit (automaticky, zatím), ale umožňuje přidat optimalizační přepisovací pravidlo.

Monády, idea

- Monáda $M\ a$: hodnoty typu a jsou zabaleny v "kontejneru" M , který obsahuje přidanou informaci.
- $M\ a$ je typ výpočtu: $[a]$ pro nedeterminizmus, $Maybe\ a$ pro výpočty s chybou, se stavem, s kontextem, s výstupem, IO , Id pro "prázdnou" monádu ...
- Hodnotu a nemůžeme "vybrat" z monády a pracovat s ní přímo. Monadická hodnota se zpracuje na monadickou hodnotu.
- Monadický kód určuje pořadí vyhodnocování (důležité pro IO)
- Místo fce `zdvih` podobné `map` se používá `(>>=)`, tzv. `bind`, které popisuje pořadí vyhodnocování (jednovláknovost)
- Druhá funkce je `return`, která vyrábí hodnotu $M\ a$.

```
zdvih :: (a -> M b) -> M a -> M b
```

```
map    :: (a -> M b) -> M a -> M(M b)
```

```
(>>=) :: M a -> (a -> M b) -> M b
```

```
return :: a -> M a
```

(Monády, alternativně)

- Funkce $f :: a \rightarrow \underline{M} \ b$ a $g :: \underline{b} \rightarrow M \ c$ máme problém složit (na rozdíl od nemonadických $f' :: a \rightarrow b$ a $g' :: b \rightarrow c$). Funkce $(>>=)$ pomůže: (jinak bychom potřebovali $g'' :: \underline{M} \ b \rightarrow M \ c$)

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

- Monády mají v teorii vlastnosti (asociativita $>>=$, return je neutrální prvek pro bind), které překladač nekontroluje (zatím).
- `return`, `join` a `map` (konkrétně `fmap` z třídy `Functor`) je alternativní definice monády, místo `return a (>>=)`

```
map  ::                (a->b) -> (M a->M b)
fmap :: Functor M => (a->b) -> (M a->M b)
join :: M(M a) -> M a
join mma = mma >>= id
ma >>= f = join $ map f ma
```

Monády

- Monády pro typové konstruktory jsou definované pomocí typových tříd (v GHC 8 postupně `Functor M`, `Applicative M`, `Monad M`), tj. `>=> a` `return` jsou přetížené. (Následně jsou ze základních funkcí odvozené obecnější funkce, které jsou potom použitelné pro všechny monády.)
- Jednotlivé monády mají specifické výkonné funkce. (Každá monáda má jiné funkce, tj. nejsou součástí interface monády.)
- Používané monády nad typem `a`:
 - výpočet s chybou: `Maybe a`,
 - nedeterministický výpočet: `[a]`,
 - vstup/výstup: `IO a`,
 - výpočet s kontextem: `r->a`, (jako `Reader r`)
 - výpočet s výstupem: `(w, a)`, (jako `Writer w`)
 - výpočet se stavem: `s->(s, a)`, (jako `State s`)
 - continuation monad: `(a->r)->r`, (jako `Cont r`)
 - ...

Monáda: Maybe

- Typ `Maybe a`: monáda s chybou. Specifická funkce je vyvolání chyby, tj. vrácení `Nothing`.

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

```
data PlusT a = ContT a
              | PlusT a :+ PlusT a
              | VarT String
evalM :: Num a => PlusT a -> Maybe a
evalM (ContT a)    = Just a
evalM (va :+ vb) = evalM va >=> \a ->
                    evalM vb >=> \b ->
                    return (a+b)
evalM _            = Nothing
```

Monády: seznamy

- Seznam je monáda pro nedeterminizmus:
- Specifická akce je vrácení dvou (a víc) výsledků.

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

- Zpracování výsledků probíhá zleva a do hloubky. Pořadí ve výstupním seznamu je určeno funkcí `>>=`.

Monadický vstup a výstup

- Potřebujeme určit pořadí operací, což při líném vyhodnocování nejde přímo.
- Pro V/V se používá typový konstruktor `IO`, kde `IO a` je typ vstupně-výstupních akcí, které mají vnitřní výsledek typu `a`. (`IO` je podobné monádě pro stav)

```
type IO a = World -> (a, World) -- pro představu
getChar :: IO Char -- načte znak do vnitřního stavu
putChar :: Char -> IO () -- vypíše znak, výsledek je ne-
zajímavý
getLine :: IO String -- načte řádek
putString :: String -> IO () -- vypíše řetězec na std. výstup
```

- Na vnitřní stav se nedostaneme přímo, např. `upIO :: IO a -> a`, porušila by se referenční transparentnost: `upIO getChar` vrací různé výsledky.

- Připomínám pro převody:

```
show :: Show a => a -> String,  
read :: Read a => String -> a
```

- Čisté funkce nepracují s IO. Pragmaticky: používáme čisté funkce co nejvíc.
- Dříve uvedené funkce pracují se standardním vstupem a výstupem. Lze pracovat i se soubory, pomocí handle (otevření, načtení, výpis do, uzavření): knihovna `System.IO` se standardně načítá.

IO 2

- V IO máme $(>=>) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$
- příklad: převod na velké písmeno:

```
(getChar >=> putChar . toUpper) :: IO ()  
getChar >=> \c -> putChar (toUpper c)
```

- Operátor $(>>)$ ignoruje výsledek první akce, např. `putChar :: IO ()` (když nás zajímají pouze "sideefekty")

```
(>>) :: IO a -> IO b -> IO b  
o1 >> o2 = o1 >=> \_ -> o2  
> putStr "Hello, " >> putStr "world"  
sequence_ :: [IO ()] -> IO ()  
sequence_ = foldr (>>) (return ())  
> sequence_ (map putStr ["Hello, ", " ", "world"])
```

- IO poskytuje jednoznačný výstup i při líném vyhodnocování

Program s IO, pro kompilaci

- příklad: převod 1 řádku na velká písmena
- Proveditelný program se jménem `Main` a typem `IO ()` lze skompilovat a spustit.

```
-- import System.IO -- v Prelude
main :: IO () -- main pro kompilované programy
main =
    putStr "Zadej vstupní řádek:\n" >>
    getLine >=>
    \vstup -> putStr "Původní řádek:\n" >>
    putStr vstup >>
    putStr "\nPřevezeno na velká písmena:\n" >>
    putStr (map toUpper vstup)
```

Práce v monádě Maybe

- Definujeme `safeHead` a analogicky `safeTail`.

```
safeHead :: [a] -> Maybe a -- výst. typ Maybe
safeHead []      = Nothing  -- hlava neexistuje, chyba
safeHead (x:xs) = Just x   -- hlava zabalená v Just
```

- Bezpečný součet prvních dvou čísel ze seznamu v monádě `Maybe`. Vrací `Nothing`, pokud čísla neexistují.

```
safePlus xs =
  safeHead xs  >>= \x1 ->
  safeTail xs  >>= \xs1 ->
  safeHead xs1 >>= \x2 ->
  return (x1+x2)
```

- Monáda ošetřuje informace navíc, zde `Nothing` a zabalení v `Maybe`, tj. pattern matching je schovaný.

Maybe 2

- Funkce `safeHead` a `safeTail` se při monadickém zpracování nemění: dostávají nechybové vstupy. Naopak, kontrolují vstup a případně chybu vyvolávají.

Do-notation

- Speciální syntax pro monády: do-notation: podobná stručným seznamům (bez `let`)

```
> do {x<-[1,2];y<-[3,5]; return(x+y)}  
[4,6,5,7]
```

- do-notation umožňuje psát kód podobný procedurálním jazykům, proto je důležitá a oblíbená.
- Monadické zpracování běží na pozadí.
- příklad výše:

```
safePlus xs =  
  do {x1 <- safeHead xs; xs1 <- safeTail xs;  
      x2 <- safeHead xs1; return(x1+x2)}
```

Do-notation

- Do-notation je použitelná pro lib. monádu: je přetížená. Používá často 2D syntax.

```
do v1 <- e1
   v2 <- e2
   e3  -- když návr. hodn. nepotřebujeme, ale 2D-zarovnané
   ...
   vn <- en
   return (f v1 v2 ... vn)
```

- Je syntaktický cukr pro

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


Outline

1 Úvod

2 Haskell 2 Dodatky

Moduly, knihovny

- Použití hierarchických knihoven a modulů:
- Načte se celý soubor, případně pouze definice (funkcí, datových typů, ...) uvedené v závorkách. Knihovny jsou kompilované a proto rychlejší.
- Každý import na samostatném řádku

```
import Data.List (sort)
import qualified Data.List (sort) as DL
```

- V 2. případě používáme volání `DL.sort` pro rozlišení, tzv. *kvalifikované jméno*
- Definice modulu, v závorkách případně seznam exportů: (jménoTypu(konstruktory), funkce)

```
module Z17 (Z17(Z17),mInv) where
...
```

- Modul `Z17` se hledá jako soubor `Z17.hs` nebo `Z17.lhs`

Striktní vyhodnocení

- Měli jsme (\$) jako aplikaci: $f \$ x = f x$
- Funkce (\$) je striktní aplikace
- - volání $f \$! x$ vyhodnotí nejvyšší konstruktor x , potom volá $f x$
- Pokud je x elementárního typu, vyhodnotí se *úplně*; po vyhodnocení víme, že není \perp ; x se vyhodnocuje do tzv. hlavové normální formy (HNF).
- Motivace: předcházení memory leak

Příklad: foldl'

- Příklad: foldl nevyhodnocuje akumulátor hned, tj. nastává memory leak.

```
length xs = foldl (\d x->d+1) 0 xs
> length [1,2]
==> foldl .. 0 [1,2]
==> foldl .. (1+1+0) []
```

- Akumulátor chceme vyhodnocovat hned: definujeme foldl' pomocí \$!

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

- Používá se primitivní funkce (nedefinovatelná v Hs): seq :: a -> b -> b : vyhodnotí nejvyšší konstruktor prvního arg. (tj. HNF), pak vrátí 2. arg.

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

Třídy pro monády

- Třídy `Functor f`, `Applicative f`, `Monad m`

```
class Functor f where
    fmap      :: (a -> b) -> f a -> f b
class Functor f => Applicative f where
    pure      :: a -> f a
    (<*>)     :: f (a -> b) -> f a -> f b
class Applicative m => Monad m where
    (>>=)     :: forall a b. m a -> (a -> m b) -> m b
    (>>)      :: forall a b. m a -> m b -> m b
    return    :: a -> m a
    m >> k    = m >>= \_ -> k
    return    = pure
```

- varianta `bind (>>=)` s přehozenými arg. pro porovnání typů
`(>>=)'` :: (a -> f b) -> f a -> f b

FFT

- Rychlá Fourierova transformace

```
fft :: Num a => a -> [a] -> [a]
fft _ [x] = [x]
fft e xs = vs where
    (sude,liche) = rozdel xs
    vs1 = fft (e*e) sude
    vs2 = fft (e*e) liche
    c2 = zipWith (*) vs2 (iterate (e*) 1)
    vs  = zipWith (+) vs1 c2 ++ zipWith (-) vs1 c2

t2fft = fft (Z17 2) [0,1,0,0,0,0,0,0] -- v Z17
t3fft = fft (0:+1)  [0,1,0,0]         -- v Complex
t4fft = fft (cis (pi/4)) [0,1,2,3,4,3,2,1]
```

Funkcionální styl 1 2

-
-
-
-
-
-

-
-
-
-
-
-