

Neprocedurální programování

Prolog 1

Úvod



Robert Kowalski
*1941



Alain Colmerauer
1941 - 2017

Co je to „neprocedurální programování“ ?

Procedurální (imperativní) programování

- Pascal, 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 [[html](#)]
- programování s omezujícími podmínkami
[[NOPT042](#)]

Prolog : Implementace

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

Prolog : Následníci

Datalog

- dotazovací jazyk pro deduktivní databázové systémy
- syntakticky podmnožina Prologu

Mercury

- vychází z Prologu i Haskellu
- Zoltan Somogyi, U of Melbourne

Erlang

- funkcionální jazyk pro aplikace v reálném čase
- první implementace v jazyce Prolog

Programování s omezujícími podmínkami

- NOPT042

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**

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).
```

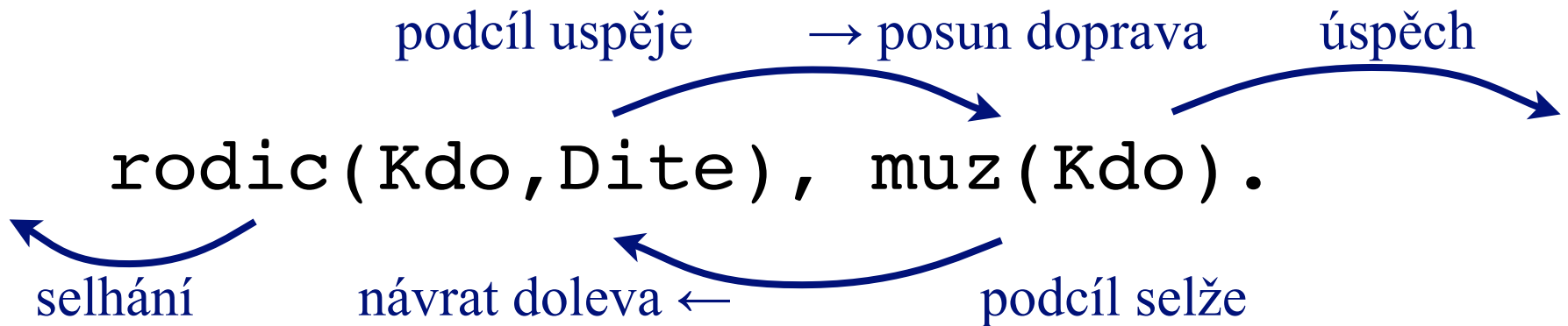
pravidlo

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 `Bratr \= Osoba` uspěje
- pokud cíl `Bratr = 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

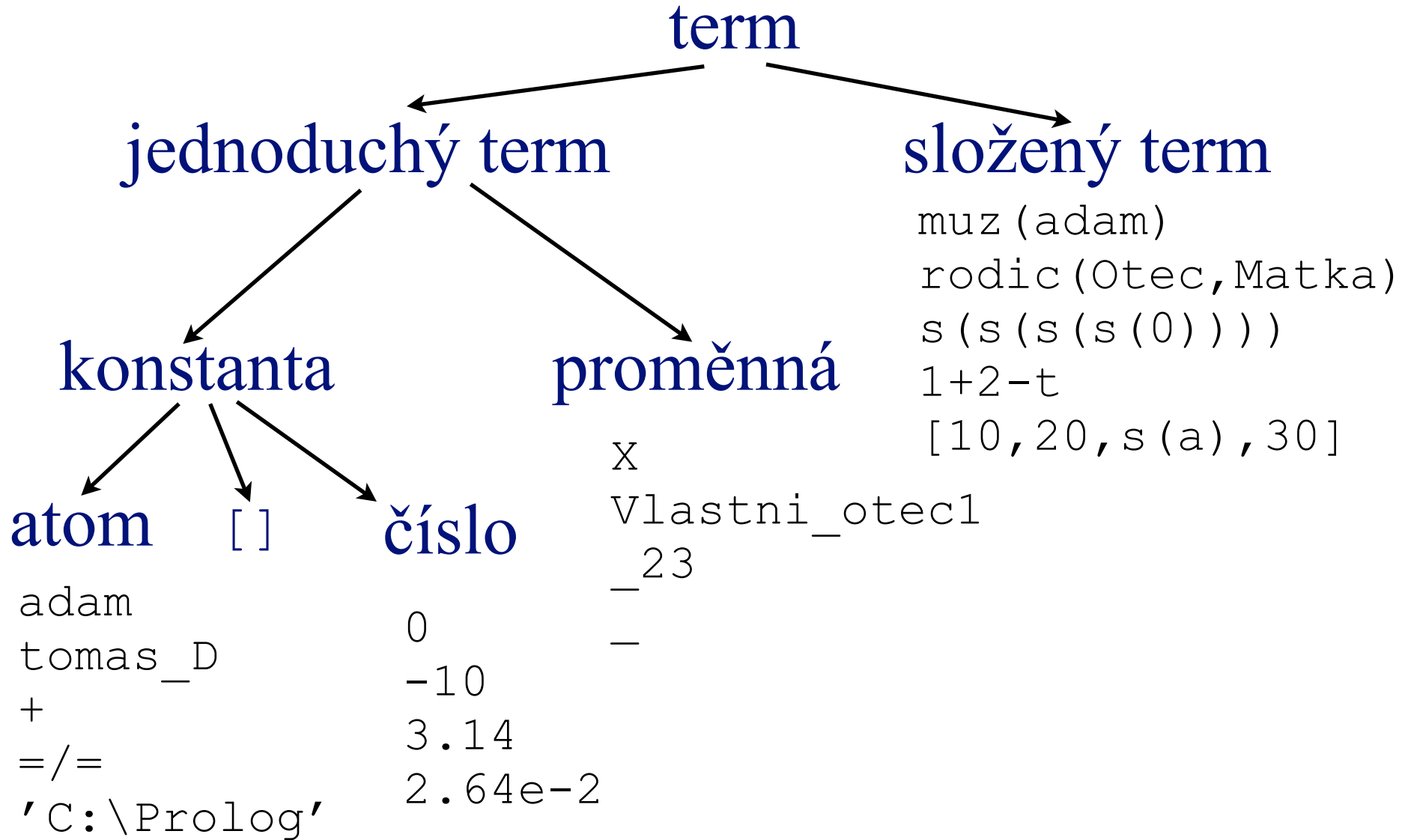
- ① V jazyce Prolog popište **fakta** z příběhu.
- ② Přidejte pravidla pro definici **příbuzenských vztahů**.
- ③ Formulujte dotazy, které ověří platnost tvrzení uvedených v příběhu (“můj syn se tak stal mým strýcem” apod.).
- ④ Formulujte dotazy, který dokáže tvrzení

“Moje žena je mojí babičkou”

a

“Já jsem svým dědečkem”

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

- $\text{funktor}(\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n)$
- 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(28,2,2020),kod('NPRG005'))`
- `s(s(s(s(0))))`

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 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

Je 5 domů, každý jiné barvy.

V každém domě bydlí osoba, která

- vystudovala obor informatika na jisté fakultě,
- má v oblibě jistý operační systém,
- pracuje pro jistou IT společnost,
- a ráda programuje v jistém jazyce.

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

Víme, že

Absolvent MU má v oblibě systém Linux.

Absolvent ZČU žije v červeném domě.

Absolvent ČVUT programuje v jazyce Haskell.

Absolvent UP pracuje pro Microsoft.

Absolvent MFF bydlí v posledním domě.

Fanoušek systému iOS pracuje pro Apple.

Obyvatel žlutého domu obdivuje Android.

Fanoušek systému macOS programuje v jazyce Swift.

Obyvatel zeleného domu pracuje pro Google.

Dále víme, že

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

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

Fanoušek **Windows** má **sousedu**, který pracuje pro **IBM**.

Zelený dům stojí (hned) **napravo** od **bílého** domu.

Absolvent **MFF** bydlí **vedle modrého** domu.

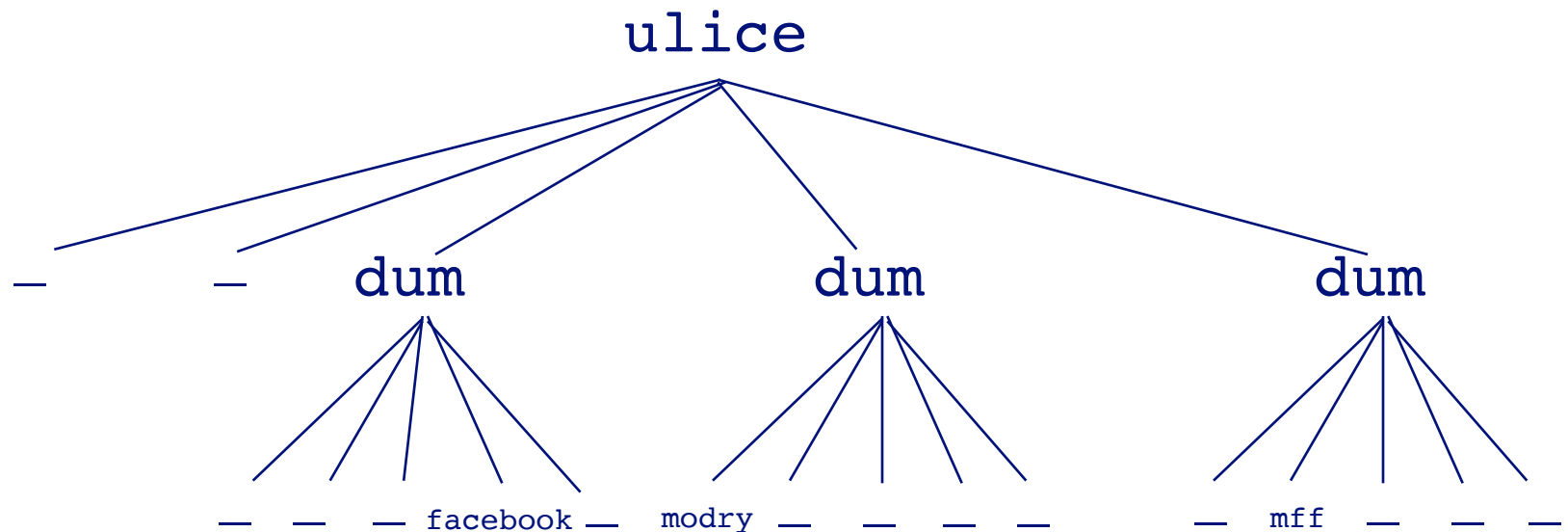
Obyvatel **prostředního** domu pracuje pro **Facebook**.

Problém

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

Einsteinova hádanka: datová struktura

```
ulice(_,  
  _,  
  dum(_,_,_,facebook,_),  
  dum(modry,_,_,_,_),  
  dum(_,mff,_,_,_))
```



Seznamy

[] prázdný seznam

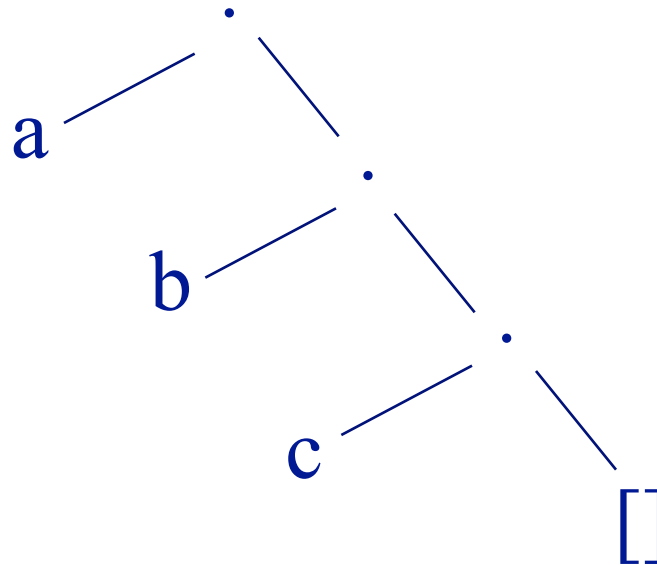
[a,b,c] příklad neprázdného seznamu

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]`
- notace tečka-dvojic už není viditelná
- `./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

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(a,[b,c],V).`
 - » není-li vypouštěný prvek v zadaném seznamu, selže
- předdefinován jako `select/3`

Problém: další varianty vypouštění

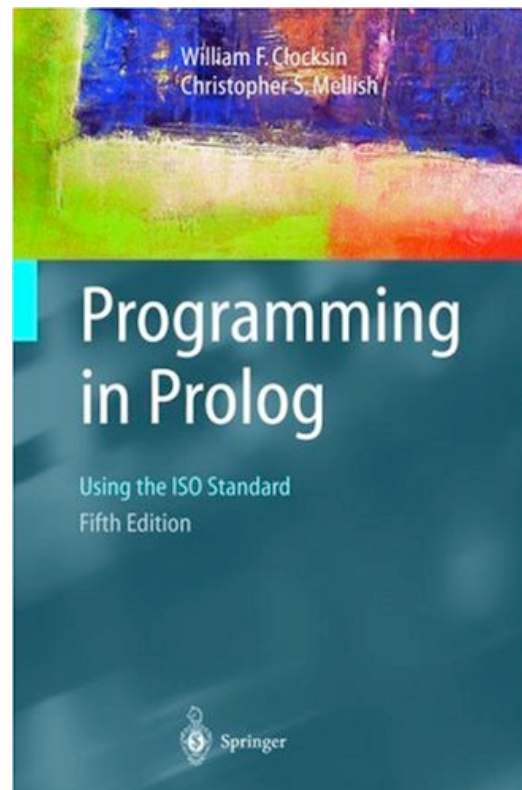
Definujte následující predikáty

- `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

Neprocedurální programování

Prolog 2

28.2.2020



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

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

```
SplňováníCíle(Program, SeznamCílů, Úspěch)  
if SeznamCílů je prázdný then Úspěch := true  
else Cíl := hlava(SeznamCílů)  
      Další := tělo(SeznamCílů)  
      Splněno := false  
      while not Splněno and Program obsahuje další klauzuli  
      do 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 then  
                  NovéCíle := zřetězení  $T_1, \dots, T_n$  se seznamem Další  
                  ve všech prvcích seznamu NovéCíle proved' substituci S  
                  SplňováníCíle(Program, NovéCíle, Splněno)  
      Úspěch := Splněno
```

Směr výpočtu

`% predek(X,Y) :- X je předkem Y.`

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

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

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

- syntaxe to nespecifikuje
- některé argumenty mohou hrát roli vstupu i výstupu
 - » relace “=” je symetrická
- `?- predek(premysl_orac, karel_ctvrty).`
- `?- predek(premysl_orac, Y).`
- `?- predek(X, karel_ctvrty).`
- `?- predek(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 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).
```

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: co bylo minule

$[]$ prázdný seznam

$[a,b,c]$ příklad neprázdného seznamu

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

Vestavěné predikáty **member/2**, **select/3**

- viz další stránka

Seznamy: vestavěné predikáty

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

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

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

- `?- select(a, [a,b,a,c,a],Ys).`
- `?- select(X, [a,b,a,c,a], Ys).`
- `?- select(x, Xs, [a,b,a,c,a]).`

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) .
```

První a poslední prvek

První prvek

- hlava seznamu
- přístupný přímo pomocí |

Poslední prvek

```
% posledni(+Seznam, ?X) :- X je posledním  
                           prvkem Seznamu.
```

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

```
posledni( [ X | 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(+Seznam, ?X) :-  
%      X je prostředním prvkem Seznamu.  
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(S1, S2, [a,b,c,d,e]).

Předdefinován jako standardní predikát

`append/3`

Využití predikátu zretez/3

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

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

 **Problém:** Využijte `zretez/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),  
                   zretez(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`

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(Preddek,Potomek) :- Preddek 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(Preddek,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

Aritmetika

?- $X = 1+1$.

?- X is $1+1$.

Vestavěný predikát $is/2$

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

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

- má jen procedurální význam
- 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

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`

Slévání setříděných seznamů

```
% 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

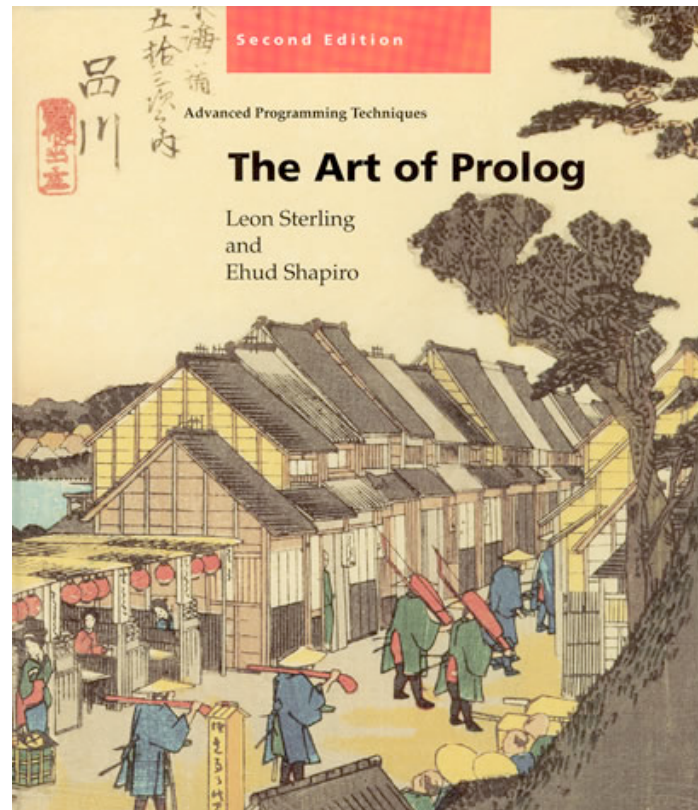
Standardní pořadí termů

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

Neprocedurální programování

Prolog 3

6.3.2020



Co bylo minule



- ✎ Unifikace, algoritmus splňování cíle
- ✎ Deklarativní a procedurální význam programu
 - příklad: predikát `předek / 2`
- ✎ Směr výpočtu
 - specifikační konvence `+`, `-`, `?`
- ✎ Seznamy
 - technika akumulátoru
- ✎ Aritmetika
 - `is / 2`
 - příklady: `délka / max seznamu`, QuickSort

Opakování: Unifikace

Základní operace na termech

Dva termy lze unifikovat, pokud

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

Unifikaci vyvolá operátor =

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

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.

?- $X = f(X)$.

Opakování: Směr výpočtu

`% predek(X,Y) :- X je předkem Y.`

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

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

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

- syntaxe to nespecifikuje
- některé argumenty mohou hrát roli vstupu i výstupu
 - » relace “=” je symetrická
- `?- predek(premysl_orac, karel_ctvrty).`
- `?- predek(premysl_orac, Y).`
- `?- predek(X, karel_ctvrty).`
- `?- predek(X, Y).`

Opakování: Směr výpočtu

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

Opakování: Specifikace vstupu / 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**

Opakování: Aritmetika

?- $X = 1+1$.

?- X is $1+1$.

Vestavěný predikát $is/2$

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

Opakování: Aritmetika

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

- má jen procedurální význam
- nepatří do čistého Prologu

Příklady

- **?- X is 4*asin(sqrt(2)/2).**
- **?- 1+2 == 2+1.**

Třídění sléváním: Mergesort

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

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

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

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


Nedeterminismus

```
msort([],[ ]).
```

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

```
msort(Xs,Ys) :- msplit(Xs,Xs1,Xs2),  
                msort(Xs1,Ys1),  
                msort(Xs2,Ys2),  
                merge(Ys1,Ys2,Ys).
```

Procedura `msort/2` je **nedeterministická**

- `?- msort([],Ys).` `?- msort([1],Ys).`
- unifikace s dvěma různými klauzulemi (fakt i hlava pravidla)!

 **Problém:** Jaké má nedeterminismus v tomto případě důsledky?

Užitečný nedeterminismus: 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)`

Nedeterminismus vs. determinismus

Deterministická procedura

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

Nedeterminismus

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

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`

Rekurze

Strukturální rekurze

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

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

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

```
% rekurzivní krok
```

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

```
% fak(+N,?F):- F je N faktoriál.  
fak(N,F) :- N > 0, N1 is N-1,  
            fak(N1,F1), F is N * F1.  
fak(0,1).
```

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

☀ Příklad koncové rekurze

```
% fak2(+N,?F):- F je N faktoriál.
```

```
fak2(N,F) :- fak2(N,1,F).
```

```
% fak2(N,A,F) :- F = N! · A .
```

```
fak2(N,A,F) :- N > 0, A1 is N*A,  
                N1 is N-1, fak2(N1,A1,F).
```

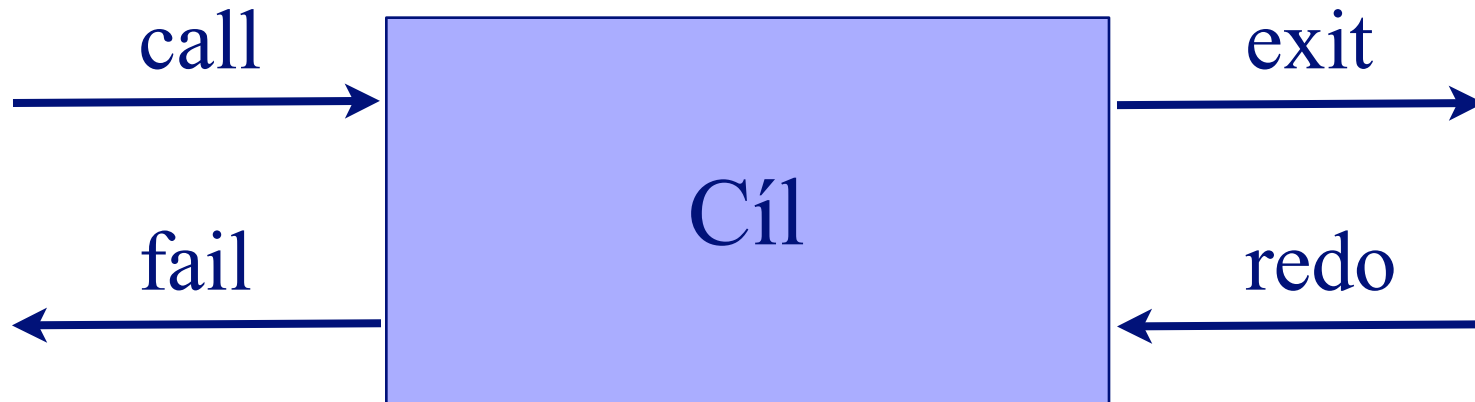
```
fak2(0,A,A).
```

Procedura `fak2/2` je **koncově rekurzivní**

- pouze konstatní prostor na zásobníku
- dodatečný argument `A` hraje roli **akumulátoru**

Jiný příklad: otočení seznamu v lineárním čase

Krabičkový model výpočtu



Ladění

Trasování výpočtu

- zastaví na každé bráně
- vypíše bránu, hloubku rekurze, cíl
- možnost zadávat příkazy
 - » výpis info nebo ovlivnění průběhu výpočtu
- `trace/0` zapne, `notrace/0` vypne

Grafický režim (SWI Prolog)

- `guitracer/0` zapne
- `noguitracer/0` vypne

Úplné trasování

- → nadměrný objem informací → selektivní trasování

Ladění

Sledování vybraných predikátů (“spypoint”)

- `spy(Pred)` nastaví sledování predikátu `Pred`
 - » `?- spy(soucet/3).`
 - » nebo `+` v režimu ladění
- `nospy(Pred)` zruší sledování `Pred`
 - » nebo `-` v režimu ladění
- `nospyall/0` zruší vše

Režim ladění

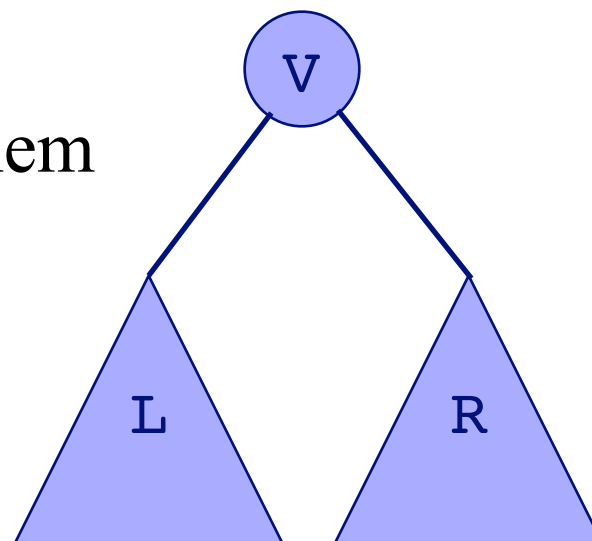
- sledování vybraných predikátů
- `debug/0` zapne ladění
- `nodebug/0` vypne ladění

Binární stromy

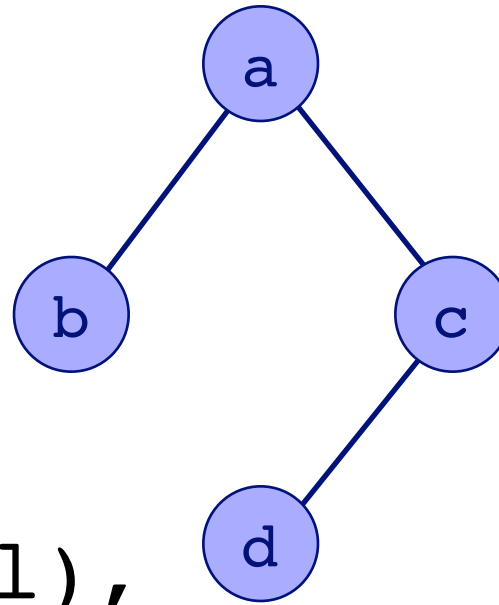
Prázdný strom: atom `nil`

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

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

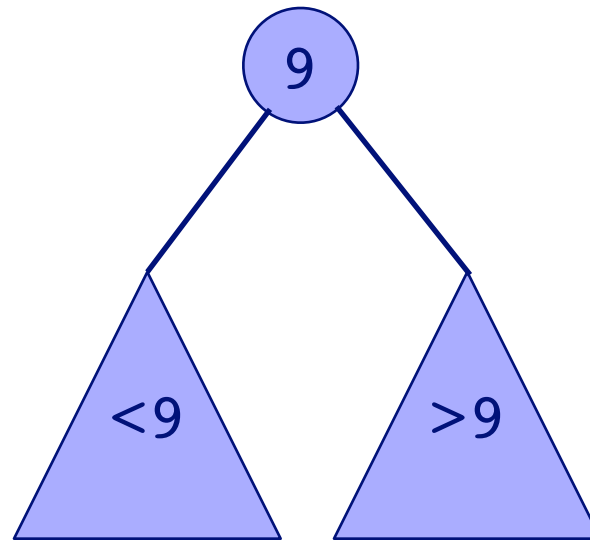


☀ Příklad binárního stromu

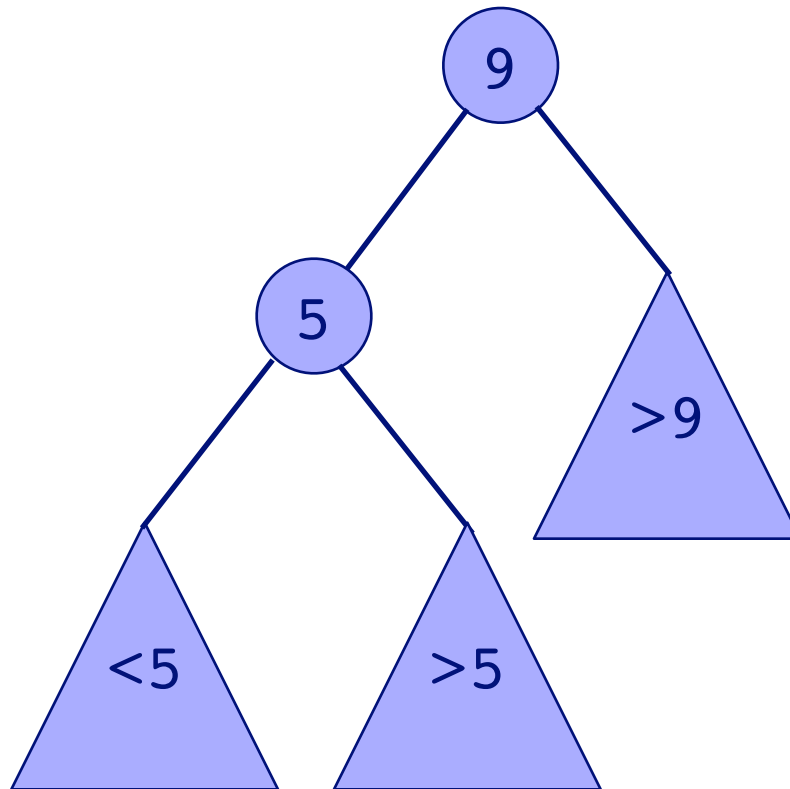


```
t(t(nil, b, nil),  
  a,  
  t(t(nil, d, nil), c, nil)  
)
```

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 vložení  
% 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)) :-  
                                delmax(L,L1,Y).
```

$t(A,B,C)$

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):-  
                                         delmax(R,R1,Y).
```



$t(A,B,C)$

Problémy se stromy

- ① Upravte predikáty `add/3` a `del/3` pro nějakou variantu **vyvážených** binárních stromů
 - AVL stromy
 - červeno-černé stromy
- ② Definujte predikáty pro
 - vložení prvku
 - odstranění kořenez **haldy**
- ③ Navrhněte reprezentaci pro obecné (kořenové) stromy

Operátory

`:- op(Priorita, Druh, Jmeno) .`

- deklarace operátoru

Priorita

- přirozené číslo $\in \langle 1, 1200 \rangle$
- nižší hodnota \rightarrow váže více

Druh

- určuje aritu (unární, binární)
- pozici (infix, prefix, postfix)
- asociativitu

Jmeno

- atom nebo seznam atomů

Předdefinované operátory

```
:- op( 200, fy, [+,-,\] ).
:- op( 200, xfy, ^ ).
:- op( 200, xfx, ** ).
:- op( 400, yfx, [*,/,//,<<,>>,mod,div,rem,xor] ).
:- op( 500, yfx, [-,+,\/,/\/,] ).
:- op( 700, xfx, [>, =, <, >=, =<, \=, @<, @=<,
    @>, @>=, =@=, ==, =\=, is, =.., ==, \==] ).
:- op( 900, fy, \+ ).
:- op(1000, xfy, ', ' ).
:- op(1050, xfy, -> ).
:- op(1100, xfy, ; ).
:- op(1105, xfy, '| ' ).
:- op(1200, ffx, [:-, ?-] ).
:- op(1200, xfx, [:-, -->] ).
```

Operátory: druh

f reprezentuje operátor

x y reprezentují operandy

- unární
 - » v prefixové notaci: **fx fy**
 - » v postfixové notaci: **xf yf**
- binární: **xfx xfy yfx**

Operátory: druh

Asociativita

- x reprezentuje operand s prioritou $<$ prioritě operátoru
- y reprezentuje operand s prioritou \leq prioritě operátoru
- prioritě operandu
 - » = prioritě hlavního funktoru, jde-li o složený term
 - » = 0 jinak

☀ **Příklad:** $a-b-c$

- $:- \text{ op}(500, yfx, -) .$
- $(a-b)-c$
- ~~$a-(b-c)$~~

Operátory: asociativita

yfx zleva asociativní

- :- op(500, yfx, +).
- $1+2+3 \rightarrow (1+2)+3$

xfy zprava asociativní

- :- op(200, xfy, ^).
- $1^2^3 \rightarrow 1^(2^3)$

xx není asociativní

- :- op(700, xx, =).
- $X = Y = Z \rightarrow \text{chyba}$

Operátory: asociativita

$fx \quad fy$ podobně

- $:- \text{op}(200, fy, -) .$
- $--1 \rightarrow -(-1)$
- $:- \text{op}(1200, fx, :-) .$
- $:- \text{op}(1200, xfx, :-) .$
- $:- p(X) :- q(X) . \rightarrow \text{chyba}$

☀ Příklad

- $1 + 2 + 3 * 4$
- $\rightarrow (1 + 2) + (3 * 4)$

Zjištění definovaných operátorů

- `current_op(?Pri, ?Druh, ?Jmeno)`

☀ Příklad: formule výrokového počtu

Můžeme vytvořit obvyklým způsobem z

- konstant **true/0**, **false/0**
- spojek **non/1**, **and/2**, **or/2**, **xor/2**, **imp/2**, **ekv/2**
- závorek
- výrokových proměnných **p/1**, např. **p(a)**

Spojky lze definovat jako operátory

- `:- op(200, fy, non).`
- `:- op(210, yfx, and).`
- `:- op(215, xfx, xor).`
- `:- op(220, yfx, or).`
- `:- op(230, xfy, imp).`
- `:- op(240, xfx, ekv).`

Výrokový počet: ověření správnosti

```
% je_fvp(+F) :- F je správně utvořená  
%                formule výrokového počtu.  
  
je_fvp(true).  
je_fvp(false).  
je_fvp(p(_)).  
je_fvp(non A):- je_fvp(A).  
je_fvp(A and B):- je_fvp(A), je_fvp(B).  
je_fvp(A or B):- je_fvp(A), je_fvp(B).  
je_fvp(A imp B):- je_fvp(A), je_fvp(B).  
je_fvp(A ekv B):- je_fvp(A), je_fvp(B).  
?- je_fvp(p(a) and non p(b)).  
true
```

Výrokový počet: vyhodnocení

```
% eval(+F,+S,-H) :- H je hodnota  
% formule F v interpretaci jejich  
% proměnných dané seznamem S  
?- eval(p(a) and non p(b),  
        [a-true,b-false], H).  
H = true
```

Vyhodnocení spojek

% predikáty pro vyhodnocení spojek

`eval_non(true, false).`

`eval_non(false, true).`

`eval_and(true, true, true).`

`eval_and(true, false, false).`

`eval_and(false, _, false).`

✎ **Problém:** Doplňte další spojky, např.

`eval_or/3`

Vyhodnocení formulí

```
eval(true,_,true).
```

```
eval(false,_,false).
```

```
eval(p(X),S,H) :- member(X-H,S).
```

```
eval(non F,S,H) :- eval(F,S,HF),  
                    eval_non(HF,H).
```

```
eval(F and G,S,H) :-  
    eval(F,S,HF), eval(G,S,HG),  
    eval_and(HF,HG,H).
```

 **Problém:** Dokončete definici eval/3

Problémy s formulemi VP

Formule je **splnitelná**

- pokud nabývá hodnoty pravda
- pro nějakou interpretaci svých proměnných

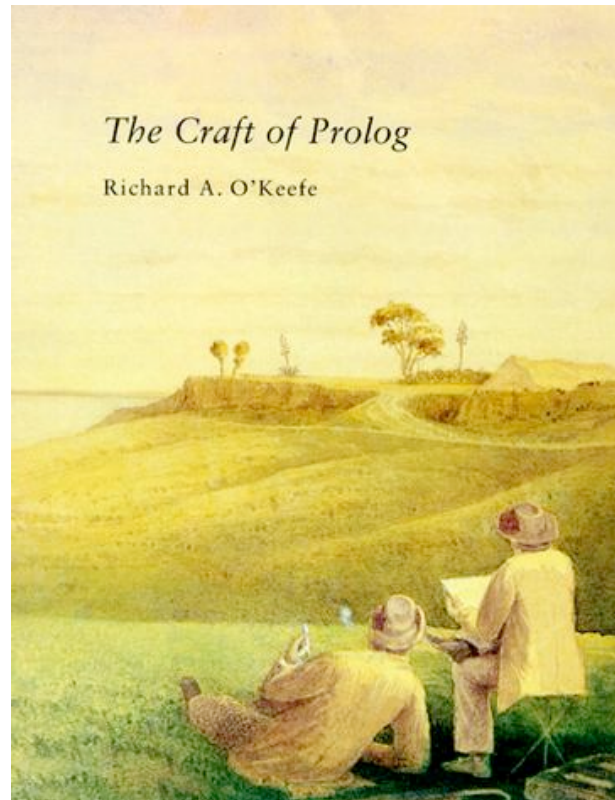
 Definujte predikát **sat/1**

```
% sat(+Formule) :-  
    Formule je splnitelná.
```

Neprocedurální programování

Prolog 4

13.3.2020



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
```

Problém logické pravdivosti ve výrokovém počtu

Formule je **tautologií**

- pokud nabývá hodnoty pravda
- pro všechny možné pravdivostní hodnoty svých proměnných

 Definujte predikát **tautologie/1**

```
% tautologie(+Formule):- Formule  
% výrokového počtu je tautologií.
```

Červený řez

Mění deklarativní význam programu

$p :- a, b.$

$p :- c.$

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

$p :- a, \textcolor{red}{!}, b.$

$p :- c.$

$$p \Leftarrow (a \wedge b) \vee (\textcolor{red}{\neg} a \wedge c)$$

☀ **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

Bill má rád počítače:

```
ma_rad(bill, X) :- pocitac(X).
```

Bill má rád počítače, ale ne od společnosti Apple:

```
ma_rad(bill, X) :- apple(X),  
                    !,  
                    fail.
```

```
ma_rad(bill, X) :- pocitac(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_rad(bill, X) :- pocitac(X),  
                  \+ apple(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

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

$\backslash +$ 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 $\backslash +$
 - » norma (ISO) doporučuje používat $\backslash +$

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(+X,+Y,-Z):- seznam Z je  
% sjednocením množin reprezentovaných  
% seznamy X a Y.
```

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 nahrad'te písmena číslicemi tak, aby platily rovnosti v řádcích i ve sloupcích. Každé písmeno nahrad'te 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.

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

Algebraamy

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

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

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

Algebrogramy

```
% soucet(S1,S2,S):- S1,S2,S jsou  
% seznamy číslíc tří přirozených  
% čísel takových, že třetí je  
% součtem prvních dvou.
```

```
soucet(S1,S2,S) :-
```

```
    s1(S1,S2,S,0,  
       [0,1,2,3,4,5,6,7,8,9],_).
```

přenos do vyššího

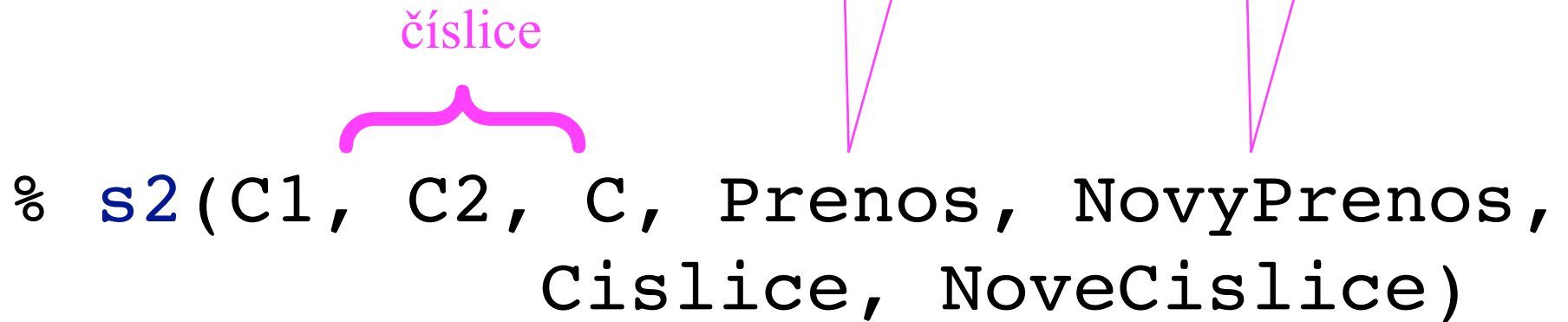
nabídka číslíc

zbylé číslice

Algebragramy

```
s1([], [], [], 0, Cisllice, Cisllice).  
s1([C1|S1], [C2|S2], [C|S],  
  Prenos, Cisllice, NoveCisllice) :-  
  s1(S1, S2, S, Prenos1,  
      Cisllice, Cisllice1),  
  s2(C1, C2, C, Prenos1, Prenos,  
      Cisllice1, NoveCisllice).
```

Algebragramy


The diagram shows the function signature `% s2(C1, C2, C, Prenos, NovyPrenos, Cislice, NoveCislice)`. A magenta bracket above `C1` and `C2` is labeled `číslice`. A magenta speech bubble above `Prenos` contains the text `přenos zprava`. Another magenta speech bubble above `NovyPrenos` contains the text `přenos doleva`. Below the signature, two magenta speech bubbles point to `Cislice` and `NoveCislice` respectively, containing the text `nabídka číslic` and `zbylé číslice`.

```
% s2(C1, C2, C, Prenos, NovyPrenos,  
    Cislice, NoveCislice)
```

Algebragramy

```
s2(C1, C2, C, Prenos, NovyPrenos,  
Cisllice, NoveCisllice) :-  
    vyber(C1, Cisllice, Cisllice1),  
    vyber(C2, Cisllice1, Cisllice2),  
    vyber(C, Cisllice2, NoveCisllice),  
    S is C1+C2+Prenos,  
    C ::= S mod 10,  
    NovyPrenos is S // 10.
```

Algebrogramy

```
% vyber(?C,+S,-S1):- Je-li C volná  
% proměnná, vybere do C číslíci ze  
% seznamu S, zbylé číslíce vrátí v  
% S1. Je-li C vázaná, S1 = S.  
vyber(C,S,S) :- nonvar(C).  
vyber(C,S,S1):- var(C),  
                  select(C,S,S1).
```

☀ Příklad: řez & negace

Pro řešení algebrogramu jsme využili predikát

`vyber`(?C,+S,-S1)

`vyber`(C,S,S) :- nonvar(C).

`vyber`(C,S,S1) :- var(C),
select(C,S,S1).

Alternativní definice s využitím řezu

`vyber`(C,S,S) :- nonvar(C), !.

`vyber`(C,S,S1) :- select(C,S,S1).

Algebrogramy: použití

ABC - AD = CF

-

*

+

JF + AG = CH

GD + AEJ = ACG

Seznamy v **soucet/3**

- musí být stejné délky
- mohou obsahovat číslice i volné proměnné

Algebramy: použití

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

```
algebragram1([A,C,D,E,F,G,H,J]) :-  
    soucet([J,F],[A,G],[C,H]),  
    soucet([0,G,D],[A,E,J],[A,C,G]),  
    soucet([0,C,F],[0,C,H],[A,C,G]),  
    A>0, C>0, J>0, G>0.
```

Algebramy: použití

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

?- `algebram1([A,C,D,E,F,G,H,J]).`

`A = 1, C = 9, D = 4, E = 6, F = 5,`
`G = 2, H = 7, J = 8 ;`

`false`

 **Problém:** zobecněte na další operátory

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,13),`

`arg(2,D,brezen),`

`arg(3,D,2020).`

`D = datum(13,brezen,2020)`

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

☀ 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

`der (F+G , X , DF+DG) :- der (F , X , DF) ,
der (G , X , DG) .`

`der (F-G , X , DF-DG) :- der (F , X , DF) ,
der (G , X , DG) .`

`der (F * G , X , F * DG + DF * G) :- der (F , X , DF) ,
der (G , X , DG) .`

`der (F / G , X , (G * DF - F * DG) / (G * G)) :-
der (F , X , DF) ,
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 `[]`

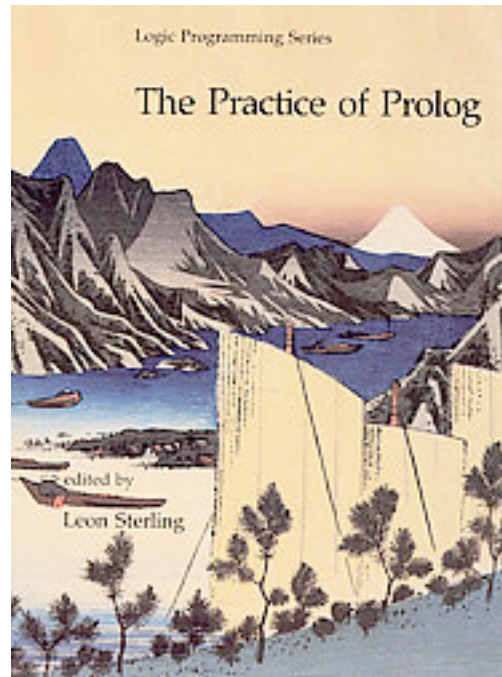
?- `findall(P, trida(P,T), Pismena).`

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




Neprocedurální programování

Prolog 5

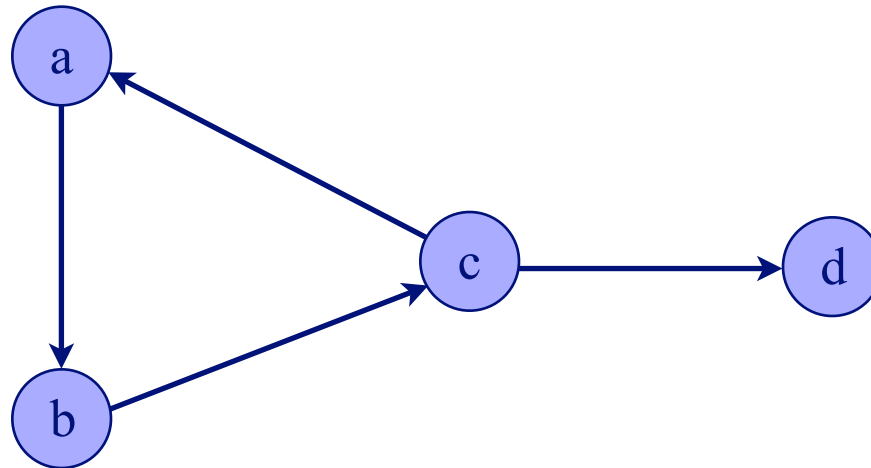
20.3.2020



Osnova

-  Grafové algoritmy
 - průchod do hloubky a do šířky
-  Prohledávání stavového prostoru
 - heuristické prohledávání, A^*
-  Zpracování přirozeného jazyka
 - příklad: Eliza

☀ Příklad: 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(+V1,?V2):- existuje cesta z  
%                vrcholu V1 do V2?
```

```
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],C),  
                  reverse(C,Cesta).
```

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

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

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).
```

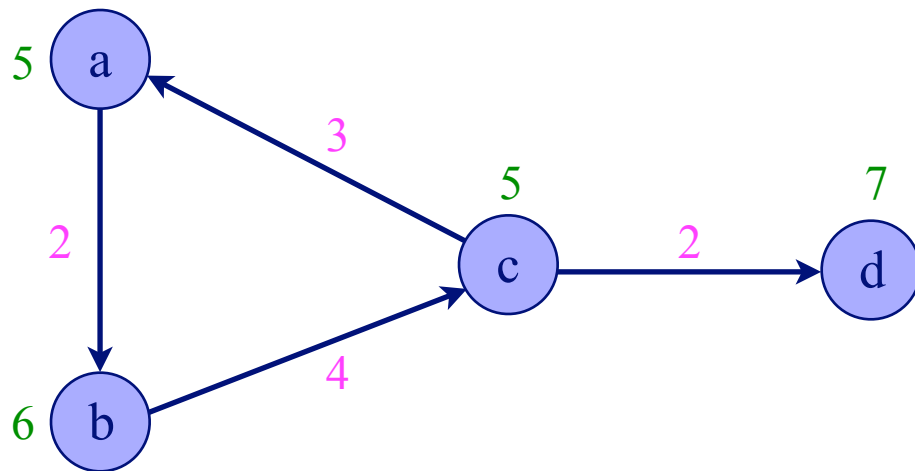
Grafy: průchod do šířky

```
% 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).
```

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**.

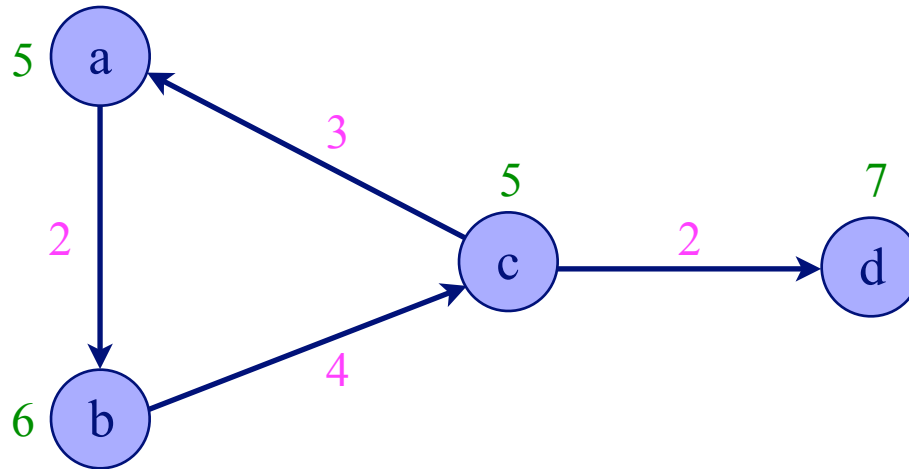
Grafy s ohodnocením



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 bfs/3 \Rightarrow dijkstra/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 zelí

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

Řešení úlohy: posloupnost stavů

Reprezentace stavu

- $s(\text{Farmer}, \text{Vlk}, \text{Kozu}, \text{Zelí})$
- 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

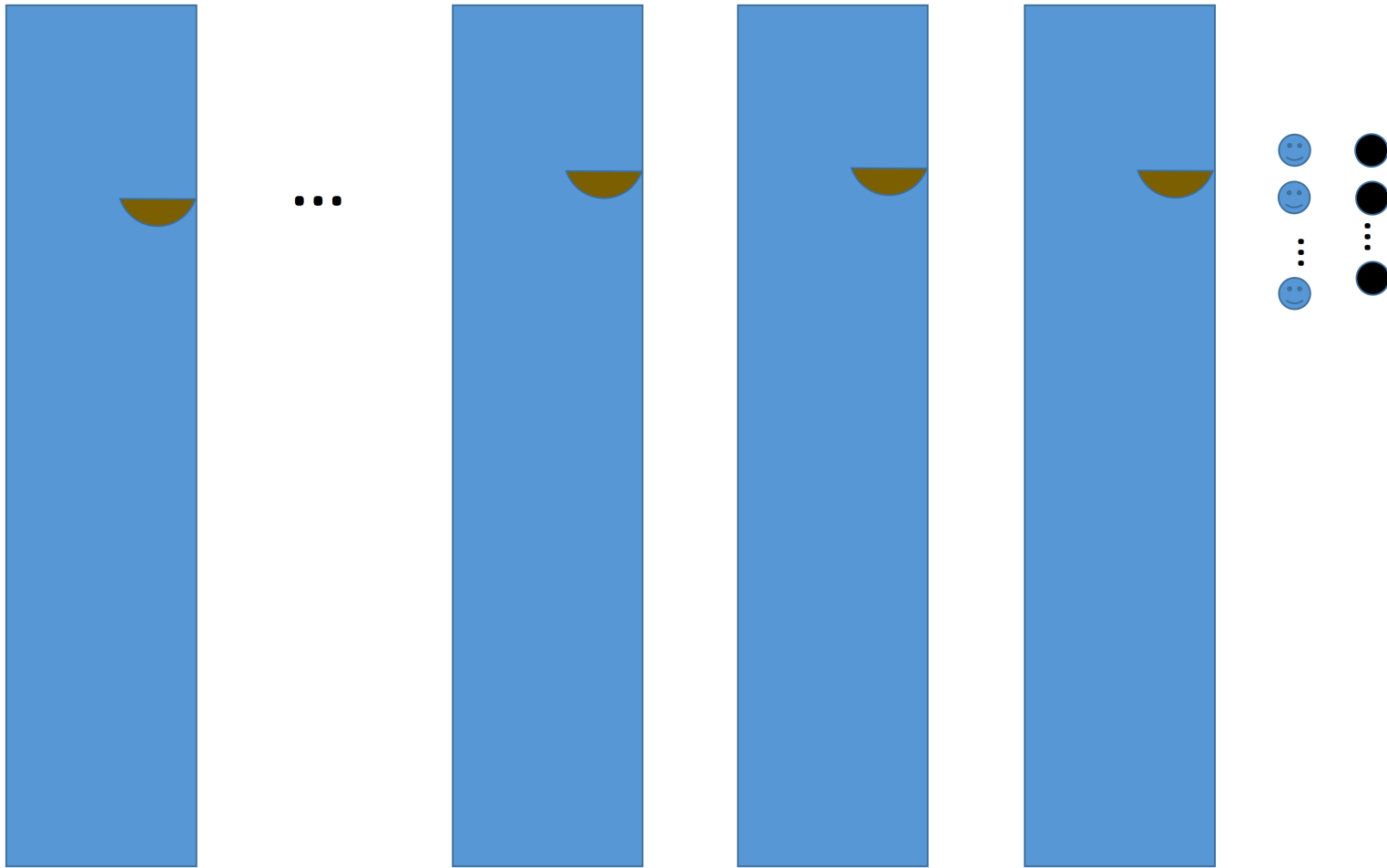
- s vrcholy $s(F, V, K, Z)$
- a hranami $hrana(X, Y) :- dalsi(X, Y)$
- z vrcholu $s(1, 1, 1, 1)$
- do vrcholu $s(p, p, p, p)$

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

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

nejkratší
řešení

Problém misionářů a lidojedů



Prohledávání stavového prostoru: uspořádaný výběr

Obecnější situace

- přechod ze stavu s_1 do stavu s_2 má **cenu $c(s_1, s_2)$**
- cena řešení $s_1, \dots, s_n = \sum c(s_i, s_{i+1})$
- hledáme optimální řešení = **řešení minimální ceny**
- příklad: problém obchodního cestujícího

Strategie prohledávání

- slepé: DFS, BFS
- uspořádaný výběr
 - » “expanze” stavu minimální ceny
 - » variace na téma Dijkstra

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*

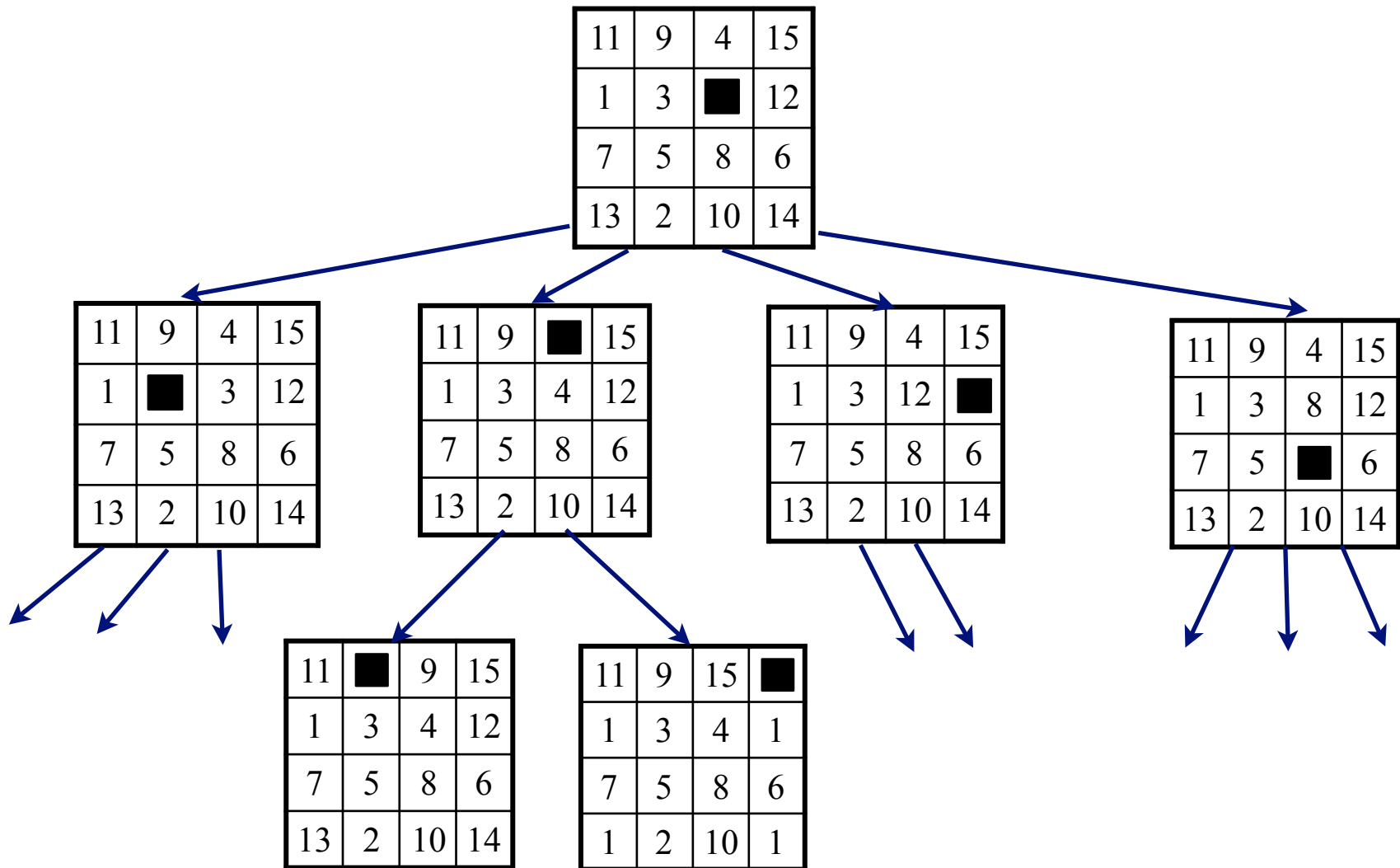
Algoritmus A*

- 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.*

Přednáška Umělá inteligence I NAIL069

☀ Příklad: Loydova “15”



Zpracování přirozeného jazyka



Eliza: Dialog s psychoanalytičkou

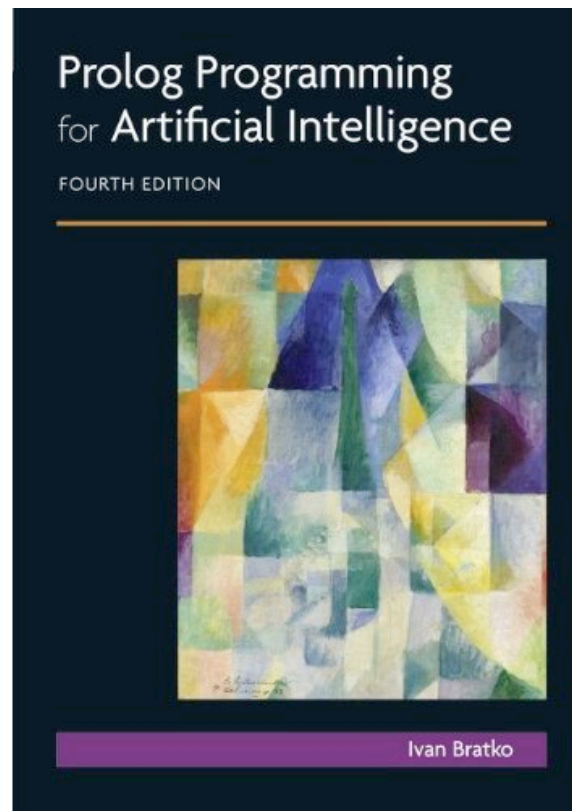
- J. Weizenbaum, ELIZA - A computer program for the study of natural language communication between man and machine. *Comm. of the ACM* 9 (1966), 36-45.
- Turingova imitační hra

```
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 6

27.3.2020



Osnova

- ✎ Vstup & výstup
 - zpracování přirozeného jazyka, Eliza
- ✎ Vestavěné predikáty pro modifikaci programu
 - `findall` pomocí `assert` & `retract`
- ✎ Práce s množinami řešení bez bagof
 - generování všech kombinací
- ✎ Predikáty vyšších řádů

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 ASCII kódem dalšího znaku na vstupu

- `get0/1` edinburská verze
- `get/1` jako `get0/1`, pouze přeskakuje řídicí znaky ($\text{ASCII} \leq 32$)

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

- `put(Z)` Edinburská verze

`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('bye', 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

```
% cti_pismena(+Pismeno,-S,-DalsiZnak):-  
% vrátí seznam S písmen slova, které  
% začíná písmenem Pismeno a za ním  
% následuje znak DalsiZnak.  
cti_pismena(46,[],46):- !.  
                        % znak '.' - konec věty  
cti_pismena(63,[],63):- !.  
                        % znak '?' - konec věty  
cti_pismena(32,[],32):- !.  
                        % znak ' ' - 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(Znak), cti_zbytek(Znak,SezSlov).  
  
cti_zbytek(46,[ ]):- !.                % konec věty  
cti_zbytek(63,[ ]):- !.                % konec věty  
cti_zbytek(32,SezSlov):- !,            % mezera  
    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 **dynamický**

- `:- 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,Vs):- N>0, N1 is N-1,
    skomb(Xs,N1,Ys),
    skomb(Xs,N,Zs),
    map_insert(X,Ys,Ws),
    append(Ws,Zs,Vs).
```

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/4`

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

```
transpose([], []).
```

```
transpose([Xs], Yss) :-
```

```
    maplist(listify, Xs, Yss), !.
```





```
transpose([Xs|Xss], TMat) :-
```

```
    transpose(Xss, Yss),
```

```
    maplist(insert, Xs, Yss, TMat).
```


Kam dále?

NOPT042 Programování s omezujícími podmínkami (Roman Barták)

-  Přehled o technikách splňování omezujících podmínek
-  Algoritmy splňování podmínek
 - prohledávací (prohledávání do hloubky, lokální prohledávání)
 - propagační (hranová konzistence, konzistence po cestě).
-  Řešení příliš omezených problémů, různé modelovací techniky
-  Předpokládány jsou základní programovací znalosti Prologu

Kam dále?

NAIL022 Metody logického programování (Jan Hric)

Přehled o logickém programování


- implementační a optimalizační techniky
- rozšíření a pokročilé metody tvorby programů


Zahrnuje

- WAM - Warrenův abstraktní stroj
- binarizace, abstraktní interpretace
- částečné vyhodnocování, typy
- programování s omezeními

Kam dále?

NAIL076-7 Logické programování I,II (J. Hric)

 Hornova logika, logické programy, procedurální interpretace logických programů, Prolog a jeho řídicí struktury. Semantika programů, ukončení práce programu, test konfliktu proměnných

 Domény a datové struktury. Konečnost výpočtů, stupňová zobrazení. Dokazování správnosti programů. Negativní informace, negace jako neúspěch, nemonotónní odvozování

Kam dále?

NAIL069 Umělá inteligence I (Roman Barták)

- ✎ Inteligentní agenti, jejich prostředí a základní struktury
- ✎ Řešení úloh prohledáváním
 - DFS, BFS, ID, A*, IDA*
 - lokální a on-line prohledávání, heuristiky
- ✎ Splňování omezujících podmínek
- ✎ Hry (minimax, alfa-beta)
- ✎ Reprezentace znalostí v logice, logické odvoz. techniky
 - dopředné a zpětné řetězení, rezoluční metoda
- ✎ Automatické plánování

Kam dále?

NOPT042 Programování s omezujícími podmínkami (Roman Barták)

NAIL022 Metody logického programování (Jan Hric)

NAIL076-7 Logické programování I,II (Jan Hric)

NAIL069 Umělá inteligence I (Roman Barták)