LOGICKE PROGRAMOVANI

A **logic program** is a set of axioms, or rules, defining relationships between objects. A **computation** of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its **meaning**. The **art of logic programming** is constructing concise and elegant programs that have the desired meaning.

Sterling and Shapiro: The Art of Prolog

– co je logicke programovani?

paradigma programovani

zalozeno na matematicke logice (automaticke dokazovani)

program = mnozina axiomu

vypocet = konstruktivni dukaz cile (dotazu), ktery zada uzivatel

– vyvoj a pouziti?

Robinson J. A.: A machine-oriented logic based on the resolution principle. *J. ACM* **12**(1965), 23–41: princip **rezoluce**

zac. 70. let 20. stol., Univerzita v Marseille: **PROLOG**

programovani pro **umelou inteligenci** (expertni systemy): LISP v USA, Prolog v Evrope

- rozsireni (neurcitost: fuzzy Prolog; Prolog zalozeny na linearni logice, ...)

– implementace

prekladace Prologu: interprety

priklady: mnoho interpretu

budeme pouzivat: SWI-Prolog (autor Jan Wielemaker, University of Amsterdam, jan@swi.psy.uva.nl, http://www.swi-prolog.org, verze pro Unix, MS-Windows)

HISTORIE LOGICKEHO PROGRAMOVANI

PRVNI LOGICKY PROGRAM

databaze Prologu

```
male(john).  male(george). male(bill). male(harold).
female(monica). female(jane).

isChildOf(john,george).  isChildOf(george,bill).
isChildOf(bill,monica).  isChildOf(jane,george).

isSonOf(X,Y) :- isChildOf(X,Y), male(X).

isDaughterOf(X,Y) :- isChildOf(X,Y), female(X).

isOffspringOf(X,Y) :- isChildOf(X,Y).
isOffspringOf(X,Y) :- isChildOf(X,Z), isOffspringOf(Z,Y).
```

uzivatelsky dotaz

```
Q- male(john).
Yes.

Q- female(john).
No.

Q- male(X).
X=john.

Q- isSonOf(john,george).
Yes.

Q- female(X).
X=monica ;
X=jane ;
No.
```

ZAKLADNI RYSY LOGICKEHO PROGRAMOVANI

- **logicky program** je konecna mnozina formuli (tvrzeni popisujici modelovanou realitu; formule maji specialni tvar)
- **vypocet** je zahajen zadanim formule-dotazu (tu zadava uzivatel)
- **cilem vypoctu je najit dukaz** potvrzujici, ze dotaz logicky vyplyva (je dokazatelny) z logickeho programu (konstruktivnost)
- pokud je takto zjisteno, ze dotaz z programu vyplyva, vypocet konci a uzivateli je oznameno `Yes` s hodnotami pripadnych promennych, ktere se v dotazu vyskytuji
- pokud neni zjisteno, ze dotaz z programu vyplyva, vypocet konci a uzivateli je oznameno `No`
- muze se stat, ze vypocet neskoci

Zakladni rysy, kterymi se logicke programovani odlisuje od ostatnich programovacich paradigmat:

- programovani: programator specifikuje, **co se ma vypocitat**, a **ne jak se to ma vypocitat a kam ulozit mezivysledky**
- **rizeni vypoctu**: Prolog nema prikazy pro rizeni behu vypoctu ani pro rizeni toku dat, nema prikazy cyklu, vetveni, prirazovaci prikaz
- **promenne**: neexistuje rozdeleni promennych na vstupni a vystupni, promenna muze byt jednou pouzita jako vstupni, jindy jako vystupni; promenna v Prologu oznacuje behem vypoctu objekt, ktery vyhovuje jistym podminkam
- nerozlisuje se mezi **daty a programem**.

Pozor na lakadlo: je treba mit na pameti, ze **programator neni zbaven zodpovednosti** za to, jak bude vypocet probihat.
Vypocet je rizen prologovskym prekladacem a programator musi pravidla, kterymi se vypocet ridi, znat a v souladu s nimi program v Prologu vytvaret.

ZAKLADNI POJMY LOGICKEHO PROGRAMOVANI

treba rozlisovat:

ciste logicke programovani (plne vysvetlitelne terminy matematicke logiky)

logicke programovani = ciste LP + prvky zvysujici komfort programatora

Prolog = konkretni programovaci jazyk zalozeny na principech LP

v dalsim:

pojem

pojem v terminech logickeho programovani

pojem v terminech logiky

logicky program a jeho spusteni

databaze a polozeni dotazu

mnozina formuli a hledani dukazu sporu s pridanou formuli

LOGICKY PROGRAM

logicky program = baze znalosti

databaze ... obsahuje fakty a pravidla

teorie ... obsahuje specialni formule

priklad: viz vyse

fakt = zakladni vztah

ma tvar **r(t1,. . . ,tn)**

r . . . relacni symbol, t1,. . . ,tn . . . termy (konstanty, promenne, slozene termy)

atomicka formule

priklad: male(john), plus(X,0,X), . . .

pravidlo = jestlize . . . pak

ma tvar **A :- B1,. . . , Bn**, A, B1,. . . , Bn . . . fakty

formule $B_1 \wedge \cdots \wedge B_n \Rightarrow A$

priklad: isSonOf(X,Y) :- isChildOf(X,Y), male(X).

PROMENNE, SUBSTITUCE, INSTANCE

promenna = bezny vyznam (hodnoty jsou termy, pres substituce)
promenna zacina **velkym pismenem**
promenna
priklad: X, Xyz, Promenna, . . .

substituce θ = navazani hodnoty
substituce = $\theta = \{(X_1, \theta_1), \ldots, (X_n, \theta_n)\}$
konecny soubor dvojic (Xi,ti), Xi promenna, ti term, a plati (1) pro $i \neq j$ je $X_i \neq X_j$, X_i se nevyskytuje v t_j
substituce ... odpovida $\{(X, t)\}$
priklad: $\theta = \{(X, john), (Y, Z), (U, john)\}$

pouziti substituce θ **na term/formuli** = nahrada promennych odpovidajicimi termy
pouziti θ **na term/formuli** A
kazdy vyskyt X_i v A se nahradi t_i
odpovida $A(X_1/t_1) \cdots (X_n/t_n)$
priklady: $\theta = \{(X, john), (Y, Z), (U, john)\}$
A=male(john); pak $A\theta$ je male(john)
$A\theta$ =male(U); pak $A\theta$ je male(john)

$A\theta$ = isOffspringOf(X,Y) :- isChildOf(X,Z), isOffspringOf(Z,Y); pak $A\theta$ je isOffspringOf(john,Z) :- isChildOf(john,Z), isOffspringOf(Z,Z)

instance = "specialni pripad" (po provedeni substituce)
B je instanci A, pokud existuje θ t.z. $B = A\theta$
B je instanci A ...
priklady: male(john) je instanci male(X)
male(Y) je instanci male(X)
isOffspringOf(john,george) :- isChildOf(john,Z), isOffspringOf(Z,george) je instanci isOffspringOf(X,Y) :- isChildOf(X,Z), isOffspringOf(Z,Y)

DOTAZY

dotaz = uzivatelem zadana otazka, spousti vypocet

C, popr. **C1,. . . ,Cn**, kde Ci jsou fakty

spec. formule ... jeji negace se prida k programu

priklady: Q- male(john)

Q- male(X)

Q- isOffspringOf(john,X), male(X)

FAKTY, PRAVIDLA, DOTAZY = HORNOVSKE KLAUZUL

Def. **Hornovska klauzule** je disjunkce atomickych formuli a negaci atomickych formuli, ve ktere se vyskytuje nejvyse jedna pozitivni (tj. nenegovana) atomicka formule.

Priklad Hornovska klauzule tedy neni male(X) $\vee$ female(Y)

fakty a **pravidla** odpovidaji hornovskym klauzulim s prave jednou pozitivni atomickou formuli, nebot
fakt A je hornovska formule
pravidlo A :- B1,. . . , Bn, tj. formule $B_1 \wedge \cdots \wedge B_n \Rightarrow A$, je (seman.) ekvivalentni Horn. klauz. $\neg B_1 \vee \cdots \vee \neg B_n \vee A$

dotaz: s dotazem $C_1, \ldots, C_n$ ve v LP (viz dale) pracuje tak, ze se negace odpovidajici formule, tj. negace $C_1 \wedge \cdots \wedge C_n$ prida k log. programu.
$\neg(C_1 \wedge \cdots \wedge C_n)$ je ale ekvivalentni $\neg C_1 \vee \cdots \vee \neg C_n$, coz je Hor. kl.

Naopak: kazda hornovska klauzule odpovida faktu nebo pravidlu (ma-li pozitivni atomickou formuli) nebo dotazu (nema-li pozitivni at. formuli)

PROMENNE A JEJICH KVANTIFIKACE

- fakty, pravidla i dotazy mohou obsahovat promenne
- promenne jsou chapany jakoby byly **vazany kvantifikatory**, a to nasledovne
- fakty a pravidla: promenne vazany obecnym kvantifikatorem $\forall$, tj. pravidlu A :- B1,. . . , Bn odpovida formule $(\forall X_1, \ldots, X_m)(B_1 \wedge \cdots \wedge B_n \Rightarrow A)$, kde $X_1, \ldots, X_m$ jsou vsechny promenne z $A, B_1, \ldots, B_n$
- dotazy: promenne vazany obecnym kvantifikatorem $\exists$, tj. dotazu $C_1, \ldots, C_n$ odpovida formule $(\exists X_1, \ldots, X_m)(C_1 \wedge \cdots \wedge C_n)$, kde $X_1, \ldots, X_m$ jsou vsechny promenne z $A, C_1, \ldots, C_n$

Poznamka S dotazem $C_1, \ldots, C_n$ se v LP pracuje tak, ze se negace odpovidajici prida k logickemu programu (a rezolucni metodou se hleda spor). Negaci odpovidajici formule je formule

$$\neg(\exists X_1, \ldots, X_m)(C_1 \wedge \cdots \wedge C_n)$$

a ta je (seman.) ekvivalentni s

$$(\forall X_1, \ldots, X_m)(\neg C_1 \vee \cdots \vee \neg C_n)$$

Poznamka Vidime: **v LP se pracuje s univerzalne kvantifikovanymi hornovskymi klauzulemi**. Proto kvantifikatory v LP nepiseme.

UNIFIKACE

Def. **Unifikace** (unifikacni substituce) mnoziny $\{\varphi_1, \ldots, \varphi_n\}$ formuli (termu) je substituce θ, ktera prevadi vsechny φ_i ve shodnou formuli (tj. $\varphi_i\theta = \varphi_j\theta$ pro lib. i, j).

Def. Substituce θ_1 je obecnejsi nez substituce θ_2, prave kdyz existuje substituce σ t.z. $\theta_1\sigma = \theta_2$ (tj. θ_2 vznikne "upresnenim z θ_1").

Def. **Nejobecnejsi unifikace** (mgu, most general unifier) mnoziny T formuli (termu) je takova unifikace mnoziny T, ktera je obecnejsi nez kazda jina unifikace mnoziny T.

Poznamka Nejobecnejsi unifikace neni urcena jednoznacne. Dve nejobecnejsi unifikace se mohou lisit ...

Priklad (prednasky)

TERMY V LOGICE VS. V LOGICKEM PROGRAMOVANI

logika: termy a formule (ruzne syntakticke objekty)

logicke programovani: (nekdy) jine pojeti, nasledovne:

- kazda konstanta i promenna je term
- jsou-li $t_1, \ldots, t_n$ termy a je-li f n-arni **funktor**, pak $f(t_1, \ldots, t_n)$ je term.

Za funktory se pritom povazuji, ktere se pouzivaji jako funkcni i jako relacni symboly v logice.

Poznamka To ma vyhody i nevyhody:

vyhody: povazujeme-li funkcni i relacni symboly za funktory, jsou termy a atomicke formule ve smyslu logiky termy v smyslu LP; to umoznuje jednoduse zavest nektere pojmy a operace s termy i formulemi logiky (zavedenim pro termy ve smyslu LP), napr. pouziti substituce, apod.

nevyhody: michani funkcnich a relacnich symbolu

JEDNODUCHY ABSTRAKTNI PREDKLADAC PROLOGU

situace: k logickemu programu T (konecna mnozina faktu a pravidel) je zadan dotaz C

ukol (pro prolog. prekladac): plyne z T dotaz C (tj. $T \models C$)?

jak prekladac postupuje: prokazuje, ze $T \cup \{\neg C\}$ je sporna (nebot $T \cup \{\neg C\}$ je sporna, prave kdyz $T \models C$); spornosti rozumejme spornost definovanou v axiomatickem systemu predik. logiky, viz prednasky Matematicka logika

jak se hleda spornost? tzv. **rezolucni metodou** (Robinson, 1965), princip:

- fakty a pravidla a dotaz se chapou jako hornovske klauzule
- pomoci rezolucniho odvozovaciho pravidla se hleda prazdna klauzule (ta odpovida sporu), podrobneji viz prednasky Mat. logika
- v pripade hornovskych klauzuli se rezolucni metoda zjednodusuje (je treba zapojovat klauzuli odpovidajici cili)
- to je obsazeno v popisu abstraktniho prekladace Prologu

abstraktni prekladac Prologu

vstup: logicky program T, dotaz $C = \{C_1, \ldots, C_n\}$ (konjunkce)

vystup: "Yes" a substituce $C\theta$, pokud bylo odvozeno $T \models C\theta$; "No", pokud nebylo odvozeno $T \models C\theta$

algoritmus:

inicializuj $R := C$;
dokud $R \neq \emptyset$
    vyber cil $A' \in R$ a pravidlo nebo fakt $A : -B_1, \ldots, B_n$ $(n \geq 0)$ z T tak,
    ze mgu AaA' je θ
    nelze-li takovy vyber provest, vyskoc ze smycky
    z R vyjmi A a pridej tam $B_1, \ldots, B_n$
    na R a C pouzij θ (tj. proved $R := R\theta$, $C := C\theta$)
pokud $R = \emptyset$, vypis "Yes" a $C\theta$; jinak vypis "No"

co zustava nespecifikovano?

- jak najit mgu formuli A a A' (**unifikacni algoritmus**)
- vyber A' je **nedeterministicky**
- vyber $A : -B_1, \ldots, B_n$ je **nedeterministicky**

Poznamenejme predem: Skutecne prekladace Prologu vzniknou upresnenim vyse popsaneho abstraktniho prekladace nasledovne:

- mgu formuli A a A' se nalezne tzv. **unifikacnim algoritmem**
- "zdeterministicneni" vyberu A': dilci cile v R jsou usporadany (zleva doprava), bere se prvni cil A' zleva
- "zdeterministicneni" vyberu $A : -B_1, \ldots, B_n$: programove klauzule (fakty a pravidla) jsou usporadany (shora dolu), bere se prvni mozna klauzule shora

Vypoctovy strom

- duledek nedeterminisicnosti: z daneho stavu vypoctu je mozne pokracovat nekolika cestami (ty odpovidaji vyberum A' a $A : -B_1, \ldots, B_n$), graf postupnych moznych stavu vypoctu tedy neni linearni, ale je to strom (**vypoctovy strom**)

- **stav vypoctu**: je dan aktualni hodnotou R (aktualni cil, ktery je treba splnit) a C (uzivatelsky dotaz, na ktery byly postupne pouzity nalezene mgu θ), viz algoritmus abstraktniho prekladace

- podle algoritmu abstraktniho prekladace vypocet konci odpovedi Yes (spolu s vypsanim $C\theta$, tj. dotaz s hodnotami promennych, pro ktere dotaz vyplyva z programu), prave kdyz prekladac prejde do stavu s prazdnou mnozinou aktualnich cilu ($R = \emptyset$);

- **vypoctovy strom**:
  - strom; kazdy vrchol je ohodnocen stavem vypoctu, tj. dvojici (R, C)
  - korenem stromu je dvojice (UC, UC), kde UC je uzivatelem zadany cil
  - nasledovniky vrcholu ohodnoc. (R, C) jsou prave vrcholy ohodnoc. (R', C'), pro ktere existuji A' v R a pravidlo nebo fakt $A : -B_1, \ldots, B_n$

v programu tak ze θ je mgu formuli A' a A a plati
(a) R' vznikne aplikaci θ na mnozinu vzniklou z R vyjmutim A' a pridanim $B_1, \ldots, B_n$ (tj. $R' := (R - \{A'\} \cup \{B_1, \ldots, B_n\})\theta$)
(b) C' vznikne aplikaci θ na C (tj. $C' := C\theta$)

Lze dokazat tvrzeni: Uzivateluv cil UC plyne z programu, prave kdyz vypoctovy strom (tj. s korenem (UC, UC)) obsahuje vrchol $(\emptyset, C)$; v tom pripade je $C = UC\psi$, kde ψ je substituce urcujici, hodnoty promennych, pri kterych UC plyne z programu.

Detaily viz kurz Matematicka logika.

Tedy, abstraktni prekladac je navrzen tak, ze uziv. cil UC plyne z log. programu T ($T \models UC$, odpoved by mela byt Yes), prave kdyz to prekladac zjisti (ocitne se ve stavu $(\emptyset, C)$, a tedy vypise Yes a C). Pritom C je instanci UC, $C = UC\psi$, pro kterou $T \models C$ (tj. ψ udava hodnoty promennych v dotazu UC, pri kterych dotaz plyne z programu).

Abstraktni prekladac je ale nedeterministicky. Potrebujeme jeho (efektivni) deterministickou implementaci, tj. **konkretni prologovsky prekladac**. Ten ma za ukol simulovat abstraktni prekladac, tj. prohledavat vypoctovy strom. Postupuje se nasledovne:

- Abychom mohli hovorit o prvnim, druhem, . . . , nasledovniku uzlu ve vypoctovem strome, provedeme nasledujici:
  - dilci cile v UC a R se povazuji za usporadane (ocislovane, "zleva doprava", tj. napr. $UC = (UC_1, \ldots, UC_k)$)
  - programove klauzule (fakty) se povazuji za usporadane (ocislovane, "shora dolu", tj. muzeme mluvit o programove klauzuli s cilsem 1, 2, . . . , n,. . .)
  - prvni nasledovnik (R', C') uzlu (R, C), pro $R = (R_1, \ldots, R_m)$, vznikne z klauzule $A : -B_1, \ldots, B_n$ s nejmensim cislem (tj. z 1. pouzitelne klauzule brano shora dolu), pro kterou je A unifikovatelne s R_1 (R_1 je A' z abstr. prekladace). Pritom, je-li θ prisl. mgu, pak $R' := (B_1, \ldots, B_n, R_2, \ldots, R_m)\theta$, tj. dilci cile se pridaji na zacatek, a $C' := C\theta$.

- Takovy strom je mozne prohledavat (pripomenme, ze ve strome prekladac hleda uzel $(\emptyset, C)$) beznymi technikami prohledavani stromu (tj. v konkretnim prolog. prekladaci mohou byt tyto techniky pouzity). Zejmena jde o:

- Prohledavani do sirky (tj. po jednotlivych urovnich/patrech stromu). Vyhoda: Pokud hledany uzel $(\emptyset, C)$ ve stromu existuje, pak ho prekladac najde. Nevyhoda: Vypocetne narocne.

- Prohledavani do hloubky: proved projdi(UC, UC), kde pro uzel (R, C) je projdi(R, C) definovano nasledovne:
  - pokud $R = \emptyset$, vypis Yes a C, skonci cely vypocet; jinak
  - ma-li uzel (R, C) nasledovniky (R_1, C_1), $\ldots$ (R_n, C_n), provadej postupne projdi(R_1, C_1), $\ldots$ projdi(R_n, C_n) (vsimnete si: v projdi(R_i, C_i) se vypocet muze zastavit s uspechem, tj. Yes)
  - nema-li uzel (R, C) nasledovniky, neprovadej nic (tj. ukonci se projdi(R, C) a prejde se do predchudce uzlu (R, C))

- Nevyhody prohledavani do hloubky: Vypocet se muze vydat po nekonecne vetvi, uspesne reseni (tj. uzel $(\emptyset, C)$) nemusi byt nalezeno. Vyhody: Efektivita (prolog. programy se pisou s ohledem na to, ze prohledavat se bude do hloubky).

- Pouzivane prekladace Prologu pouzivaji prohledavani do hloubky s vyse uvedenym (tj. ocislovani programovych klauzuli a dilcich cilu). Prohledavani je implementovano prostrednictvim zasobniku (zasobnik realizuje rekurzivni prohledavani), tzv. prologivsky zasobnik.

- Z toho primo plyne: Pri psani prolog. programu zalezi na poradi faktu a pravidel (ovlivnuje to strom vypoctu a tedy vysledek prohledavani do

hloubky). Lze ukazat, ze na poradi dilcich cilu nezalezi (tj. reseni je nalezeno pri jednom poradi, prave kdyz je nalezeno pri jeho libovolne permutaci).

Souvislost popsaneho odvozovani a rezolucniho odvozovani

Odvozovaci krok v tom, co bylo popsane, ma tvar:

z (cile) $A', R_2, \ldots, R_m$ a (program. klauzule) $A :- B_1, \ldots, B_n$ odvod $B_1\theta, \ldots, B_n\theta, R$
kde θ je mgu A' a A

v terminologii rezolucniho odvozovani: z (cilove) horn. klauzule $\neg A' \vee \neg R_2 \vee \cdots \vee \neg R_m$ a horn. klauzule $A \vee \neg B_1 \vee \cdots \vee \neg B_n$ odvod (cilovou) horn. klauzuli $\neg B_1\theta \vee \cdots \vee \neg B_n\theta \vee \neg R_2\theta \vee \cdots \vee \neg R_m\theta$, kde θ je mgu A' a A

Pripomenme, ze cili $A', R_2, \ldots, R_m$ (konjunkce) odpovida klauzule $\neg A' \vee \neg R_2 \vee \cdots \vee \neg R_m$ (negace te konjunkce).

Tedy: Odvozovaci pravidlo (v prolog. prekladaci) je jen jinak zapsane rezolucni odvozovaci pravidlo a prolog. prekladac implementuje rezolucni odvozovani. Viz prednasky Mat. logika.

UNIFIKACNI ALGORITMUS

vstup: termy t_1 a t_2 (ve smyslu LP, tj. i formule)

vystup: θ . . . mgu termu t_1 a t_2, popr. "No", pokud mgu neexistuje

algoritmus:

inicializuj $\theta := \emptyset$; zasobnik obsahuje $t_1 = t_2$; fail:=false
dokud zasobnik$\neq \emptyset$ a not(fail)
    vyjmi ze zasobniku $X = Y$
    pokud
      X je promenna nevyskytujici se v Y: pouzij substituci (X, Y) na zasobnik a na θ (tj. substituuj tam Y za X) a pridej (X, Y) do θ
      Y je promenna nevyskytujici se v X: pouzij substituci (Y, X) na zasobnik a na θ (tj. substituuj tam X za Y) a pridej (Y, X) do θ
      X a Y jsou totozne (konstanty nebo promenne): pokracuj
      X je $f(X_1, \ldots, X_n)$ a Y je $f(Y_1, \ldots, Y_n)$: pro $i = 1, \ldots, n$ vloz na zasobnik $X_i = Y_i$
      jinak fail:=true
pokud fail=true, vypis "No"
jinak vypis θ

K unifikacnimu algoritmu: Lze dokazat, ze je korektni (pracuje spravne).

HLEDANI RESENI

```
/* follows(X,Y,L) ... Y follows X in a list L */

follows(X,Y,[X,Y|L]).

follows(X,Y,[Z|L]):-follows(X,Y,L).
```

dotazy a postupne generovani cilu:

```
follows(U,4,[1,2,3,4,5]).
```

prubeh vypoctu viz prednasky (pri prochazeni stromu nedojde k navraceni)

Hledani alternativnich reseni a zpetne navraceni (backtracking)

```
follows(1,U,[1,1,2,3,2]), follows(3,U,[1,1,2,3,2]).
```

- prvni cile lze unifikovat s faktem i pravidlem v definici follows
- po unifikaci s faktem vede na cil follows(3,1,[1,1,2,3,2])
- to vede postupne az na cil follows(3,1,[]), ktery nelze splnit a dochazi k navraceni zpet a hledani alternativnich reseni
- probehne navrat az k cili follows(1,U,[1,1,2,3,2]) a probehne unifikace s hlavou pravidla; to vede na cil follows(1,U,[1,2,3,2])
- to vede k nalezeni reseni pro U=2

prubeh vypoctu podrobneji viz prednasky

ZAKLADNI PRVKY JAZYKA PROLOG

Prolog je rozsirenim dosud predstaveneho cisteho logickeho programovani. S predstavenymi zaklady LP pristoupime k vykladu jazyka Prolog (spolecneho jadra).

Zaklady syntaxe

pouzivane znaky: bezne (alfanumericke a nektere specialni)

konstanty - zacinaji malymi pismeny (jmena relaci, objektu: petr, 12, isBrother, ...)

promenne - zacinaji velkymi pismeny

anonymni promenne - znak _ (podtrzitko)

Kazdy vyskyt je vyskytem jine promenne, tj. $r(\_,\_,\_)$ je jako $r(X,Y,Z)$. Hodnoty anonymnich promennych se nevypisuji (_ je jako "jakakoli hodnota").

Je-li v databazi r(a,b)., pak na dotaz ?-r(X,X). je odpoved No., na dotaz ?-r(_,_). je odpoved Yes (bez vypisovani hodnot promennych).

Rez

Rez je zvlastni predikat v Prologu, oznacovany !, slouzi k "orezani" vypoctoveho stromu.

! je pri prvnim pruchodu splnen, pri navraceni je nesplnen a zpusobi navrat o jednu uroven vys, tj. alternativni reseni aktualniho dilciho cile nebudou zkousena (prislusne vetve vypoctoveho stromu budou odrezany).

Priklad

```
r:-a,!,b,c.
r:-d,!.
r:-e.
```

Pri pokusu o splneni r bude pouzita nejvyse jedna ze tri uvedenych klauzuli: Nejprve nastane pokus pouzit prvni klauzuli. Pokud cil a neni splnen, nastane pokus pouzit druhou klauzuli. Pokud je cil a splnen, je splnen i ! a nastava pokus o splneni b. Pokud b neni splnen, zpusobi navraceni pres !, ze uz nenastava pokus o alternativni splneni a a splneni r konci neuspechem. Pokud b je splnen, napstava pokus o splneni c. Pokud neni c splnen, nastava pokus o alternativni splneni b, viz vyse. Pokud c je splnen, pokus o splneni r konci uspechem. Pri pripadnem navraceni (pri neuspechu nektereho nasledujiciho

cile) zpusobi !, ze pri navratu az k pokusu o znovusplneni r (druhou klauzuli) se tento pokus neuskutecni, tj. vetev odpovidajici splneni druhe klauzule se odrizne.

Dalsi priklad:

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

Dotaz prvek(X,[a,b,c]). vede na odpovedi X=a; X=b; X=c; No. (Pripomenme, ze strednik ; vede k vyvolani pokusu o alternativni reseni.)

Pri zmene na

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

zabrani ! po nalezeni reseni X=a navratu a dalsi reseni nebudou nalezena.

Rez ma tedy vliv na proceduralni vyznam programu. Rez byva prirovnava k pouzivani GOTO (prikaz skoku).

Zeleny a cerveny rez

Podle vlivu pouziti rezu na deklarativni vyznam programu rozlisujeme tzv. zeleny rez a cerveny rez. Rez v programu se nazyva zeleny, jestlize jeho

pouziti nema vliv na deklarativni vyznam programu (tj. pro kazdy uziv. dotaz se program chova stejne s tim rezem jako bez nej, pripadny vliv ma rez jen na efektivitu vypoctu). Jinak (tj. s vlivem na deklarativni vyznam programu) jde o rez cerveny.

Podrobneji viz cviceni.

Programovani cyklu

predikaty **repeat** (vzdy splnen), **fail** (vzdy nesplnen, vyvola navrat k predchozim cilum)

repeat jako by byl definovan nasledovne:

```
repeat.

repeat:-repeat.
```

Priklad:

```
% nekonecna smycka vyvolana uziv. dotazem
repeat, fail.
```

Cyklus s podminkou

Chceme vyjadrit konstrukt DO C1,...,CN UNTIL PODM. To lze nasledovne:

repeat, C1,...,CN,PODM,!.

Napr.

```
input:-repeat, write('input integer <50:'), read(N), integer(N),
N<50,!.
```

Smycka, ve ktere bude nacitan vstup, dokud budou zadavana cela cisla mensi nez 50.

Cyklus rizeny promennou

```
/*cyklus s promennou od do */
cycleC(I,I,J,_) :- I>J,!,fail.
cycleC(I,I,J,_).
cycleC(I,J,K,S) :- J1 is J+S, cycleC(I,J1,K,S).

doC(I,D,H,K,C) :- cycleC(I,D,H,K),call(C),fail.
doC(I,D,H,K,C).
```

Pr.: ?-doC(I,1,5,1,(write(I),nl)). Bude postupne vypisovat 1,2,3,4,5 (na novych radcich).

(Nektere) vestavene predikaty Prologu

Implementace Prologu se lisi. Nasledujici jsou zpravidla spolecne. Vice viz uzivatelske prirucky, jde o prehledove uvedeni moznosti, ktere v Prologu zpravidla mame.

klasifikace termu

atom(X) ... splneno, kdyz X je atom
integer(X) ... splneno, kdyz X je cele cislo
var(X) ... splneno, kdyz X je volna promenna
nonvar(X) ... splneno, kdyz X je vazana promenna
atd.

operatory

moznost definovat vlastni operatory (pro snazsi cteni), vcetne priorit a asociativity

aritmeticke operatory

obvykle $+$ $-$ $*$ / mod is

Obvykle pouziti (viz programy).

Pozor: “=” vs. “is”

2*1 = 2, ani 1*2=2*1 neni splneno, 2*1=X je splneno pro X=2*1; obecne t=s je splneno, pokud t a s jsou stejne termy.

Kdy se aritmeticky vyraz vyhodnocuje? Je-li druhym argumentem predikatu “is” nebo argumentem predikatu <, >, =<, >=, =:=, = \ =

Priklady:

rizeni databaze

assert(K) ... prida na zacatek databaze klauzuli, ktera je navazana na promennou K

retract(K) ... odstrani z databaze prvni klauzuli, ktera je unifikovatelna s K

podrobneji viz programy

vstup a vystup

write(X) ... vypise instanci, na kterou je navazana promenna X

read(X) ... X je navazana na term, ktery je na vstupu

podrobneji viz programy

ladeni programu

debug, trace

JEDNODUCHE PROLOGOVSKE PROGRAMY

dalsi prvky Prologu:

SEZNAMY

- $[a, b, c, d]$
- [] ... prazdny
- $[a]$ ... jednoprvkovy
- $[a, [b, c], d]$ ... triprvkovy
- $[a, [[b, c], [d, e]], f]$ ... triprvkovy
- $[H|T]$ ... alternativni zapis, hlava H, telo T
- $[a, b, c, d]$ ... take jako $[a|[b, c, d, e]]$, $[a, b|[c, d]]$, $[a, b, c|[d]]$,
- $[a, b, c, d|[]]$ ... jednoprvkovy

programy

```
/*faktorial*/

faktorial(0,1).

faktorial(N,F):-N>=1, N1 is N-1, faktorial(N1,F1), F is N*F1.
```

```
 /* member(X,Y) ... X je prvkem seznamu Y*/

memberL(X,[X|T]).

memberL(X,[Y|T]) :- memberL(X,T).
```

```
/* nthEl(X,N,Y) prave kdyz X je N-tym prvkem seznamu Y */

nthEl(X,1,[X|T]).

nthEl(X,N,[Y|T]):- nthEl(X,N1,T), N is N1+1.
```

```
/* lastEl(X,Y) prave kdyz X je poslednim prvkem seznamu Y*/

lastEl(X,[X]).

lastEl(X,[_|Y]):-lastEl(X,Y).
```

```
/* lengthL(X,N) ... N is the length of the list X*/

lengthL([],0).

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

```
/*occurrence(X,Y,N) pravek kdyz X se v seznamu Y vyskytuje prave
N-krat  */

occurrence(X,[],0).

occurrence(X,[X|T],N):-occurrence(X,T,N1), N is N1+1.

occurrence(X,[Y|T],N):-Y\==X, occurrence(X,T,N).

/*zde Y\==X ... term Y se nerovna termu X */
```

```
/* deleteFromList(X,Y,Z) ... Z vznikne odstranenim vsech vyskytu X
ze seznamu Y */

deleteFromList(X,[],[]).

deleteFromList(X,[X|T],S):-!,deleteFromList(X,T,S).

deleteFromList(X,[Y|T1],[Y|T2]):-deleteFromList(X,T1,T2).
```

```
/*appendL(X,Y,Z) ... list Z is list X followed by list Y */

appendL([],T,T).

appendL([X|T],K,[X|U]) :- appendL(T,K,U).
```

```
/* insertFirst(X,Y,Z) ... Z vznikne vlozenim X pred prvni prvek
seznamu Y*/

insertFirst(X,Y,[X|Y]).
```

```
/* insertLast(X,Y,Z) ... Z vznikne vlozenim X za posledni prvek
seznamu Y*/

insertLast(X,[],[X]).

insertLast(X,[Y|T1],[Y|T2]):-insertLast(X,T1,T2).
```

```
/*replaceFirst(X,Y,U,V) ... V vznikne z U nahrazenim prvniho
vyskytu prvku Y prvkem X */

replaceFirst(X,Y,[],[]).

replaceFirst(X,Y,[Y|T],[X|T]):-!.

replaceFirst(X,Y,[Z|T1],[Z|T2]):-replaceFirst(X,Y,T1,T2).
```

```
/* reverseL(X,Y) ... seznam X je obracenim seznamu Y */

reverseL([],[]).

reverseL([X|T],U) :- reverseL(T,L), appendL(L,[X],U).
```

```
/*followsL(X,Y,Z) ... Y nasleduje za X v seznamu Z*/

followsL(X,Y,[X,Y|T]).

followsL(X,Y,[Z|T]) :- followsL(X,Y,T).
```

```
/* prefix(X,Y) ... X je prefixem (tj. pocatecnim usekem)  Y */

prefix([],_).

prefix([X|T1],[X|T2]):-prefix(T1,T2).
```

```
/* subset(X,Y) ... kazdy prvek seznamu X je prvkem seznamu Y */

subsetL([X|T],U):-memberL(X,U), subsetL(T,U).

subsetL([],_).
```

```
/* sublist(X,Y) ... X je podseznamem Y, tj. Y vznikne vlozenim
nejakych prvku do X */

sublist([X|T1],[X|T2]):-sublist(T1,T2).

sublist(U,[_|T]):-sublist(U,T). sublist([],_).
```

```
/* listsToNumbers(X,Y) ... X is a list of lists and Y is a list of
numbers of elements of these lists, e.g.
                                listsToNumbers([[a],[],[c,f,[a,s,d]]],[1,0,3]

listsToNumbers([],[]).

listsToNumbers([X|T1],[N|T2]):-length(X,N), listsToNumbers(T1,T2).
```