PROLOG: OSNOVE IN PRINCIPI STRUKTURIRANJA PODATKOV

IVAN BRATKO, MATJAŽ GAMS

FAKULTETA ZA ELEKTRONIKO IN INSTITUT JOŽE STEFAN, LJUBLJANA

UDK:519.682

Prolog je programski jezik, osnovan na formalni logiki. Program v Prologu definiramo z množico trditev in ne s postopkom kot v običajnih, postopkovno orientiranih jezikih. Zato ga štejemo med t.i. nepostopkovne ali deklarativne jezike. Izvajanje programa ustreza formalnemu dokazovanju izreka, da iz danih trditev (podatkov) sledijo ustrezni rezultati. Nekatere značilnosti Prologa močno olajšujejo programiranje operacij na logično kompleksnih podatkovnih strukturah in pa sistemov umetne inteligence. V uvodnem delu tega članka so opisane nekatere osnovne komponente Prologa. Sledijo primeri implementacije podatkovnih struktur, ki kažejo, kako Prologova sintaksa omogoča strukturiranje podatkov brez uporabe kazalcev, kot je to običajno v drugih jezikih, npr. v Pascalu.

PROLOG: FUNDAMENTALS AND PRINCIPLES FOR DATA STRUCTURING. Prolog is a programming language based on formal logic. In Prolog, programs are defined by a set of assertions and not by a procedure as in usual, procedurally oriented languages. Therefore it is called a nonprocedural or declarative language. The execution of a Prolog program corresponds to a formal proof of a theorem that the results of the program logically follow from the given assertions (data). Some of Prolog features greately facilitate the programming of operations on logically complex data structures and of artificial intelligence systems. This paper presents the basics of Prolog followed by a number of examples of implementation of data structures in Prolog. The examples illustrate how the Prolog syntax can be used for data structuring without the use of pointers as e.g. in Pascal or other "usual" languages.

1. ZNAČILNOSTI PROLOGA IN PRIMER PROGRAMA

V zgodnjih sedemdesetih letih je veljala naslednja groba klasifikacija programskih jezikov (Sammet 1969):
"Vse programske jezike lahko razdelimo v dve skupini: v eni je Lisp, v drugi pa vsi ostali jeziki."

S tem je bila podčrtana razlika med programskim jezikom umetne inteligence Lispom (npr. Siklossy 1976) in ostalimi "normalnimi" programskimi jeziki, kot so Algol, Fortran, Cobol, PL1 idr. Odkar eksistira Prolog pa bi bilo umestno to klasifikacijo popraviti takole: Vse jezike lahko razdelimo v dve skupini: v prvi je Prolog, v drugi pa vsi ostali jeziki (vključno z Lispom).

Prolog je enostaven, vendar močan programski jezik. Njegova matematična osnova je formalna logika, v podobnem smislu kot tvorijo matematično osnovo Fortrana in njemu sorodnih jezikov aritmetični izrazi. Osnova Prologa je bila razvita na univerzi v Marseilleu (Roussel 1975) kot praktična realizacija ideje, da je mogoče matematično logiko uporabiti tudi kot programski jezik (npr. Kowalski 1974).

Izrazi predikatnega računa prvega reda, ki tvorijo sintakso Prologa, omogočajo enostavno definiranje relacij med podatki in pa strukturiranje podatkov. Zato je Prolog posebno močan pri simboličnem, nenumeričnem procesiranju in pri delu z bogatimi podatkovnimi strukturami. Mnoge formalizme iz teorije podatkovnih baz, npr. relacijski model, lahko uporabimo praktično neposredno kot program v Prologu. Močna komponenta Prologa je tudi nedeterminizem in z njim povezano avtomatsko vračanje(automatic backtracking), ki močno olajšuje programiranje algoritmov kombinatorične narave.

Tisto, kar tako močno razlikuje Prolog od ostalih pomembnejših jezikov je, da je Prolog nepostopkovni ali deklarativni jezik, za razliko od drugih, ki so vsi v bistvu postopkovni (ali proceduralni). V "konvencionalnih" jezikih zato opisujemo postopke, torej kako pridemo od danih podatkov do rezultatov. Ideja Prologa pa je, da opišemo samo, kakšna je relacija med podatki in rezultati. Prologov prevajalnik pa mora sam poiskati postopek operacij, ki prevede podatke v rezultate tako, da le-ti ustrezajo zahtevani relaciji.

To razliko med postopkovnimi in nepostopkovnimi jeziki opisujeta naslova dveh znamenitih publikacij. Prva (Wirth 1975) se nanaša na postopkovne jezike (Pascal), druga pa na nepostopkovne (Kowalski 1979):

Wirth: "Programs = Algorithms + Data Structures"

Kowalski: "Algorithm = Logic + Control".

Logika pove, kaj naj program naredi, medtem ko (v Prologu skromna) kontrolna komponenta določa, kako bo interpreter nalogo dejansko izvedel.

Za Prolog je značilno, da programa ne definiramo z zaporedjem operacij (tako kot v običajnih, postopkovnih jezikih), temveč z množico relacij oz. predikatov. Tako je npr. v Pascalu značilna operacija prireditveni stavek, npr.:

```
Y:= f(X)
```

Ta stavek se izvrši tako, da vzamemo vrednost X, izračunamo vrednost funkcije f(X) in to vrednost priredimo spremenljivki Y. Torej lahko gornji stavek ponazorimo z diagramom

kjer je X vhod in Y izhod.

Ekvivalentno definicijo bi v Prologu napisali z izrazom:

f(X,Y)

ki bi ga lahko prebrali kot : X in Y sta v relaciji f. Taka izjava deluje med izvajanjem programa takole: če je znan X, potem se izračuna Y tako, da je ta Y v relaciji f z X. Če pa je znan Y, potem se izračuna X tako, da je spet Y v relaciji f z X. Taka izjava f(X,Y) deluje v obeh smereh: enkrat je vhod X in izhod Y, drugič pa obratno:

Obstajajo pa še druge možnosti, npr.:

(1) Znana sta oba, X in Y; potem se izjava f(X,Y) obnaša kot neke vrste kretnica, ki testira, ali sta dana X in Y v relaciji s f.

(2) Noben od X in Y ni znan; v tem primeru se program obnaša tako, kot da bi Prolog priredil spremenljivkama X in Y vse možne pare vrednosti, ki so v relaciji f.

Drug primer značilnega stavka v konvencionalnem jeziku je npr.:

```
if N>0 then N := N-1;
```

kar lahko beremo kot: "Če je vrednost spremenljivke na lokaciji N večja od 0, potem zamenjaj vrednost na lokaciji N s staro vrednostjo minus 1".

Značilen primer stavka v Prologu pa je npr.:

```
potomec(X,Y):-otrok(X,Y).
```

kar beremo takole:

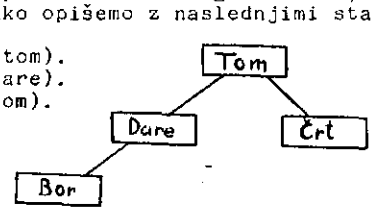
"X je eden izmed potomcev od Y, če je X eden izmed otrok od Y" ali pa

"Če je X otrok od Y, iz tega sledi, da je X potomec od Y".

Ta trditev velja za vsak X in Y. Operator :- deluje kot logična implikacija z desne v levo (v matematiki navadno znak ⟵).

Kot primer programa v Prologu vzemimo implementacijo nekaterih sorodstvenih relacij. Slika 1 kaže primer družinskega drevesa, ki ga v Prologu lahko opišemo z naslednjimi stavki:

```
otrok(dare,tom).
otrok(bor,dare).
otrok(črt,tom).
```



Slika 1. Primer družinskega drevesa.

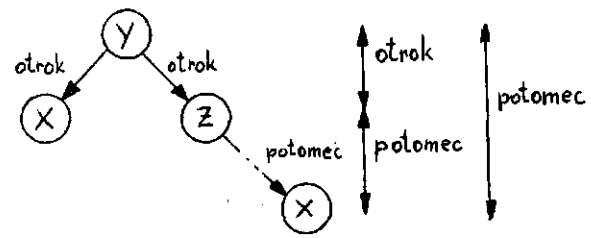
Te stavke lahko beremo npr. "Dare je otrok od Tom" ali pa "Dare in Tom sta v relaciji otrok".

Napišimo še stavke v Prologu, ki definirajo relacijo "potomec", na osnovi naslednjega razmišljanja (glej sliko 2).

(a) X je potomec od Y, če je X otrok od Y, ali
(b) X je potomec od Y, če eksistira Z, tak, da
    (1) Z je otrok od Y in
    (2) X je potomec od Z.

Ti dve izjavi opišemo v Prologu z:

```
potomec(X,Y) :- otrok(X,Y).
potomec(X,Y) :- otrok(Z,Y), potomec(X,Z).
```



Slika 2. Definicija relacije "potomec".

Zdaj lahko postavimo Prologovemu sistemu že nekaj vprašanj, npr: "Kdo je otrok od Toma in kdo je Borov oče?" To storimo z naslednjim stavkom v Prologu:

```
?-otrok(X,tom),otrok(bor,Y).
```

Odgovor, ki ga dobimo, je:

```
X=dare, Y=dare;
X=črt, Y=dare
```

Kot vidimo, sta bila na vprašanje možna dva odgovora (ker ima Tom dva otroka). Prologov interpreter je poiskal oba.

Vprašajmo še, kdo so Tomovi potomci:

```
?- potomec(X,tom).

X=dare;
X=črt;
X=bor;
```

Ta primer ilustrira način programiranja v Prologu, ki je močno različen od programiranja v konvencionalnih jezikih. V primerjavi s temi jeziki v Prologu ne eksistirajo konstrukti, kot so:

- krmilni konstrukti, npr. goto, while, repeat;
- prirejanje vrednosti spremenljivkam in spreminjanje vrednosti spremenljivk;
- dznake stavkov, globalna imena spremenljivk.

Nekatere implementacije Prologa so:

- interpreter v fortranu (napisan na univerzi v Marseillu);
- interpreter v assemblerju za DEC-10 (univerza v Edinburghu);
- kompilator, napisan v Prologu in deloma v assemblerju za DEC-10 (Edinburgh);
- interpreter v assemblerju za PDP 11/34 (Edin.)
- interpreter v Lispu za CDC CYBER (Linköping).

2. NEKATERI ELEMENTI JEZIKA IN ENOSTAVNI STAVKI

Programer v Prologu opiše svoj problem na deklarativen način, medtem ko mora Prologov sistem pri izvajanju programa vendarle izvršiti določen postopek, torej ga mora interpretatirati "postopkovno"; zato ločimo dva pomena (oz. semantiki) programov v Prologu: deklarativni in postopkovni pomen programa.

Tako npr. stavek v Prologu

```
P :- Q, R.
```

(ki je ekvivalenten zapisu v matematični logiki $P \Leftarrow Q \wedge R$) lahko beremo na naslednje načine:

(1) Deklarativno:
"Q in R implicira P" ali
"Iz Q in R sledi P"

(2) Postopkovno:
"Za to, da rešimo P moramo rešiti Q in R"
ali "En način za izvršitev procedure P je:
pokliči proceduro Q in zatem proceduro R."

Program v Prologu lahko zato razumemo kot množico logičnih trditev, ki jih mora Prologov sistem ovreči ali pa dokazati njihovo resničnost. Zato je izvajanje programa v Prologu dejansko dokazovanje trditev, ki so postavljene v programu. S tega stališča je Prologov interpreter avtomatski dokazovalnik izrekov. Stavki v Prologu imajo obliko

```
P :- Q, R, S.
```

glava stavka telo stavka

Ta stavek pomeni: "P je res, če so Q in R in S res". Q, R in S se imenujejo cilji. Število ciljev v stavku je poljubno, lahko tudi enako nič. Stavek brez ciljev je enotin stavek, npr.

```
P.
```

kar pomeni: "P je vedno res".

Z enostavnimi stavki opisujemo enostavna dejstva, npr.

```
otrok(dare,tom).
```

Vprašalni stavki se začenjajo z vprašajem,

```
?-P, Q.
```

To pomeni "ali sta P in Q resnična" oz. "reši P in Q".

Elementarni podatkovni objekti so:

(1)"atomi", npr.:
david, a, nil, tom, dare

(2)cela števila, npr.:
5, 0, -973

(3)spremenljivke (se ločijo od atomov po veliki začetnici) npr.:
X,Y, B23, List, Pilot

Spremenljivke lahko dobijo vrednost med izvajanjem programa. Sestavljeni podatkovni objekti so izrazi. Primer izraza, ki opisuje točko v tridimenzionalnem prostoru, je

```
točka(X,Y,Z)
```

funktor argumenti

Z uporabo funktorjev lahko gradimo izraze, v katerih so vgnezdeni podizrazi. Na ta način lahko strukturiramo naše podatkovne objekte. Podatkovno strukturo, ki jo imenujemo seznam, (znano predvsem iz programskega jezika Lisp) lahko npr. zgradimo z uporabo funktorja "." (pike). Seznam je zaporedje podatkov, ki ga lahko definiramo takole: seznam je bodisi (1)prazno zaporedje, ki ga po dogovoru navadno označimo z nil (no-item-list), bodisi (2) ima prvi element (imenovan glava) in pa preostanek (imenovan rep); pri tem je glava lahko karkoli, rep pa mora biti seznam (lahko tudi prazen). Primer seznama imen je npr.:

```
[ana,teja,maja]
```

Glava tega seznama je "ana", rep pa je "teja, maja".

V Prologu smemo uporabljati za sezname prav tako notacijo, kot smo jo uporabili v gornjem primeru. Vendar je to le sintaksna nadgradnja Prologove predstavitve seznamov s funktorjem ".". Ta funktor ima dva argumenta, od katerih je prvi glava seznama, drugi pa rep seznama. Tako je

```
[ana,teja,maja]
```

ekvivalentno zapisu

```
.(ana, [teja,maja] )
```

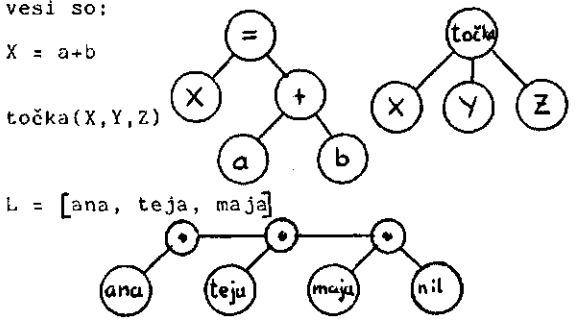
kar je ekvivalentno

```
.(ana,.(teja, [maja] ))
```

oz. naprej

```
.(ana,.(teja,.(maja,nil)))
```

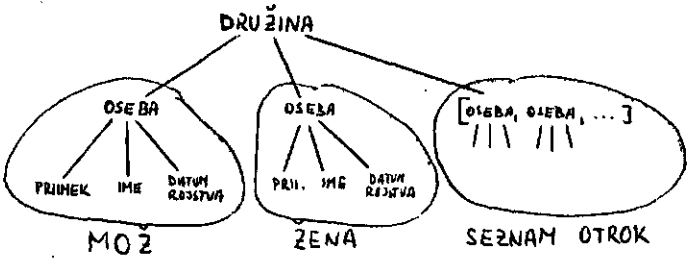
Nazorno lahko ponazarjamo izraze grafično z uporabo dreves. V takih drevesih so v notranjih vozliščih funktorji, v zunanjih pa elementarni podatki. Primeri ponazarjanja z drevesi so:



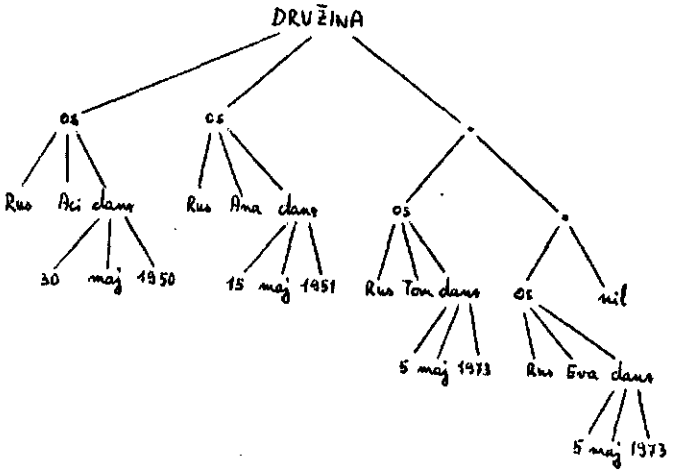
L = [ana, teja, maja]

Če pišemo Rep = [teja,maja] , potem je koristna uporaba operatorja "|", ki povezuje glavo in rep:

L = [ana | Rep]



Primer:



Slika 3. Strukturiranje podatkov o družini.

Dejstvo, da eksistira družina na sliki 3, lahko opišemo z naslednjim enotinim stavkom v Prologu:

```
druž
   (os (rus, aci, danr(30, maj, 1950)),
    os (rus, ana, danr(15, maj, 1951)),
    os (rus, tom, danr( 5, maj, 1973)),
    os (rus, eva, danr( 5, maj, 1973))).
```

Pri tem smo uvedli funktorje: druž(družina), os(oseba), danr(dan rojstva).

Leksikalni doseg spremenljivk v stavkih Prologa je omejen na stavek sam. Tako lahko v dveh stavkih nastopa ime spremenljivke X, vendar to ni ena sama spremenljivka, temveč dve povsem različni spremenljivki. Poglejmo nekaj primerov, ki ilustrirajo to pravilo:

1. Trditev "Vsakdo zaupa samemu sebi" napišemo lahko z naslednjim stavkom:

```
zaupa(X,X).
```

2. Trditev "Vsak moški X je poročen, če eksistira družina, v kateri je X mož":

```
poročen(X) :- druž(X, _, _).
```

Črtice označujejo "slepe" spremenljivke.

3. Trditev:"Katerikoli osebi A in B sta poročeni med seboj, če eksistira družina, v kateri je A mož in B žena", ali pa "če eksistira družina, kjer je B mož in A žena. " To trditev opisujeta naslednja stavka:

```
zakonca(A,B) :- druž(A,B,_).
zakonca(A,B) :- druž(B,A,_).
```

Ta stavek lahko zapišemo krajše:

```
zakonca(A,B):- druž(A,B,_); druž(B,A,_).
```

Podpičje označuje disjunkcijo med dvema ciljema.

3. DEKLARATIVNA IN POSTOPKOVNA SEMANTIKA TER IZVAJANJE PROGRAMOV

Kot smo že omenili, lahko programe v Prologu gledamo na dva načina: prvič deklarativno in drugič postopkovno. Ker imajo programi v Prologu deklarativni in postopkovni pomen, eksistirata tudi dve semantiki Prologa. Deklarativni pomen Prologovega programa govori o relacijah med podatki oz. strukturami. Torej v nekem smislu opisuje zakonitosti, ki se jim podrejajo objekti, ki nastopajo v programu, neodvisno od samega postopka. Ta nepostopkovni, deklarativni vidik Prologovih programov dviga Prolog izredno visoko v hierarhiji programskih jezikov.

Vendar pa mora Prologov interpreter priti do iskanih rezultatov vseeno po nekem postopku, zato je potrebna (kot neke vrste nujno zlo) tudi postopkovna semantika, ki natančno predpisuje, po kakšnem postopku se program izvaja. S stališča programiranja pa je nadvse pomembno to, da lahko programer v načelu razmišlja na en ali drug način, saj se programerju ni treba spuščati v detalje samega postopka. Izkaže se, da je pogosto mnogo lažje rešiti problem na deklarativen način. Deklarativna semantika Prologa definira, kdaj je kaka izjava v Prologovem programu resnična, tj. da je izjava dejstvo: "Dani cilj je dejstvo, če je ta cilj "primer" glave kakega stavka in so pri tem vsi cilji (če eksistirajo) v telesu tega stavka tudi dejstva. Primer stavka (ali izraza) dobimo s substitucijo vsake od njegovih spremenljivk z nekim drugim izrazom na vseh mestih, kjer se spremenljivka pojavlja."

Postopkovna semantika Prologa pa je dana z naslednjimi pravili:

1. Za izpolnitev cilja je treba med seboj "prilagoditi" ta cilj in glavo kakega stavka in zatem izpeljati vse cilje v telesu tega stavka ("prilagajanje" izrazov je pojasnjeno kasneje).
2. Cilje v telesu izvršuj od leve proti desni.
3. Cilj poskušaj prilagoditi glavam stavkov po vrsti od zgoraj navzdol.
4. Kadar ni mogoče (več) prilagoditi cilja nobeni glavi, se "vrni nazaj" in poskušaj izpolniti prejšnji cilj na kakšen drug način (angl. backtrack).
5. Če niti prilagajanje niti vračanje ni več mogoče, potem cilja ni mogoče izpolniti.

Prilagajanje dveh izrazov je proces, ki spremenljivkam v obeh izrazih priredi take vrednosti, da postaneta po tej substituciji spremenljivk oba izraza identična. Prilagoditev je torej množica prireditev vrednosti spremenljivkam tako, da se oba izraza izenačita. Npr. izraza

```
točka(X,Y,3) in
točka(2,Y1,Z)
```

se prilagodita takole: X=2, Y=Y1, Z=3.

Pri izvajanju Prologovega programa iščemo vedno najbolj splošno prilagoditev. Najsplošnejša prilagoditev je tista prilagoditev, ki izenači oba izraza in pri tem čimmanj natančno specificira vrednosti spremenljivk. V gornjem primeru je možna tudi naslednja, manj splošna prilagoditev :

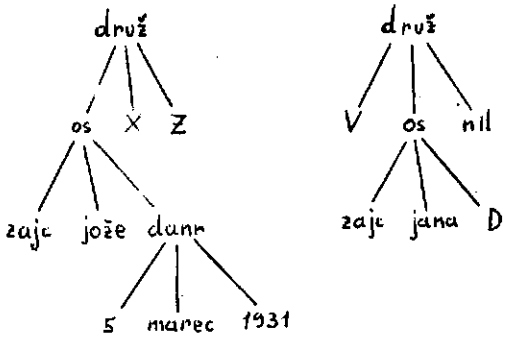
```
X=2, Y=5, Y1=5, Z=3.
```

Ta izenači oba izraza in rezultat prilagajanja je točka(2,5,3). Vendar pa ni najsplošnejša, ker sta vrednosti spremenljivk Y in Y1 po nepotrebnem povsem specificirani. Za prilagoditev zadošča namreč že omejitev Y=Y1.

Slika 4 kaže še en primer prilagajanja.

1. izraz
   druž(os(zajc,jože,danr(5,marec,1931)),X,Z)

2. izraz
   druž(V,os(zajc,jana,D),nil)



Najsplošnejša prilagoditev:
V= os(zajc,jože,danr(5,marec,1931))
X= os(zajc,jana,D)
Z= nil

Slika 4. Primer prilagajanja izrazov.

Izkaže se, da je prilagajanje izrazov izredno močno programirno orodje in da lahko že s samim prilagajanjem rešujemo probleme. Naslednji program npr. izračuna s samim prilagajanjem, katere daljice so hkrati horizontalne in vertikalne. Slika 5 ilustrira način, ki smo ga izbrali za popis nekaterih pojmov iz ravninske geometrije.

Točka je dana z dvema koordinatama, npr. t(X,Y), daljica pa z dvema točkama: daljica(T1, T2). Naslednji program definira, kdaj je kaka daljica vertikalna oz. horizontalna:

```
vertikalna(daljica(t(X,Y1),t(X,Y2))).
horizontalna(daljica(t(X1,Y),t(X2,Y))).
```

Ta program že zadošča, da lahko zastavimo nekaj vprašanj v obliki ciljev za Prologov

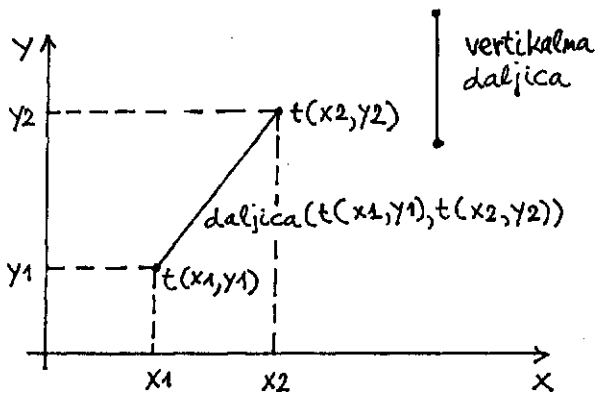
interpreter (odgovori računalnika so podčrtani, komentarji pa v oklepajih):

```
?-vertikalna(daljica(t(0,1),t(0,2))).
Yes; (odgovor je da)
?-vertikalna(daljica(t(1,2),t(2,3)).
No; (odgovor je: ta daljica ni vertikalna).
```

Vprašajmo še, ali eksistira daljica, ki je hkrati vertikalna in horizontalna:

```
?-vertikalna(D),horizontalna(D).
D=daljica(t(X,Y),t(X,Y))
```

Odgovor pomeni: da, to je vsaka daljica, ki je degenerirana v eno samo točko.



Slika 5: Opis nekaterih geometrijskih pojmov v Prologu.

Naslednji primeri bodo ilustrirali postopkovni pomen Prologovih programov.

Primer 1: relacija"ded-vnuk"

To relacijo lahko opišemo deklarativno takole: X je ded od Y, če je X roditelj od Z in obenem Z roditelj od Y, ter X je moški. To, skupaj s še nekaterimi enostavnimi trditvami, zapišemo v Prologu takole:

```
ded(X,Y) :- roditelj(X,Z),
            roditelj(Z,Y),
            moški(X).
moški(tone).
moški(peter).
ženska(ana).
ženska(eva).
moški(rok).
roditelj(ana,eva).
roditelj(tone,eva).
roditelj(eva,rok).
roditelj(peter,rok).
```

Zastavimo vprašanje : Kdo je Rokov ded?

```
?-ded(X,rok).
```

Zasledujmo izvajanje programa, to je postopkovni pomen programa:
cilj je ded(X,rok)
Ta cilj se prilagodi z glavo stavka

```
ded(X,Y):-....
```

Pri tem postane Y=rok, generirajo pa se podcilji (v telesu stavka, s katerim se je prilagodil tekoči cilj):

```
podcilj 1: roditelj(X,Z)
podcilj 2: roditelj(Z,rok)
podcilj 3: moški(X)
```

Podcilj 1 lahko izpolnimo tako, da ga prilagodimo prvemu izmed stavkov "roditelj(...)". Tako dobimo prilagoditev
X=ana, Z=eva
Po tej prilagoditvi je podcilj 1 izpolnjen (ker v telesu stavka "roditelj..." ni bilo več nobenega cilja). Zato je na vrsti podcilj 2, ki je po prilagoditvi postal
roditelj(eva,rok)
Ta podcilj je trivialen, saj je že shranjen kot dejstvo v našem programu. Naslednji, tj. tretji podcilj, je
moški(ana)
Ta podcilj se ne prilega glavi nobenega stavka, zato ga ni mogoče izpolniti. Zato se izvajanje programa vrne nazaj in skuša prejšnje podcilje izpolniti na drug način. Podcilja 2 ni mogoče izpolniti na noben drug način, zato pa je to mogoče s podciljem 1:
roditelj(X,Z)
in sicer takole:
X=tone, Z=eva
Podcilj postane zdaj roditelj(eva,rok) .
To je že enotin stavek v programu. Preostane še cilj moški(tone), ki je prav tako trivialno izpolnjen. S tem so vsi podcilji izpolnjeni. Izpiše se rešitev, ki je definirana s prilagoditvami:

X=tone

Primer 2: konkatenacija seznamov

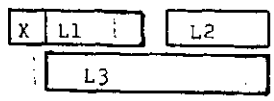
Napišimo proceduro za konkatenacijo (spajanje) seznamov, tako da bo predikat

conc(L1,L2,L3) izpolnjen,

če je seznam L3 konkatenacija seznamov L1 in L2, npr.:

conc([a,b] , [c,d,e] , [a,b,c,d,e])

Idejo za to proceduro lahko izrazimo deklarativno:
(1)Konkatenacija kateregakoli seznama L s praznim seznamom je seznam L.
(2)Če konkateniramo seznam oblike [X|L1] (glava je X, rep pa L1) s seznamom L2, dobimo seznam, katerega glava je X, rep pa konkatenacija seznamov L1 in L2 :



Ti dve trditvi izrazimo v Prologu takole:

```
conc([],L,L)
conc( [X|L1] ,L2, [X|L3] ) :- conc(L1,L2,L3).
```

Proceduro conc lahko uporabljamo na razne načine. Npr. tako, da podamo vrednosti prvih dveh argumentov in kot rezultat dobimo njeno konkatenacijo. Bolj zanimiv primer uporabe je, če je podan tretji argument, kot rezultat pa dobimo prva dva seznama, tako da je njuna konkatenacija enaka tretjemu. Če je možno tretji (dani) seznam sestaviti iz dveh seznamov na več načinov, potem dobimo kot rezultat vse možne rešitve:

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

```
?-conc(L1,L2,[a,b,c]).
L1=[],L2=[a,b,c];
L1=[a],L2=[b,c];
L1=[a,b],L2=[c];
L1=[a,b,c],L2=[]
```

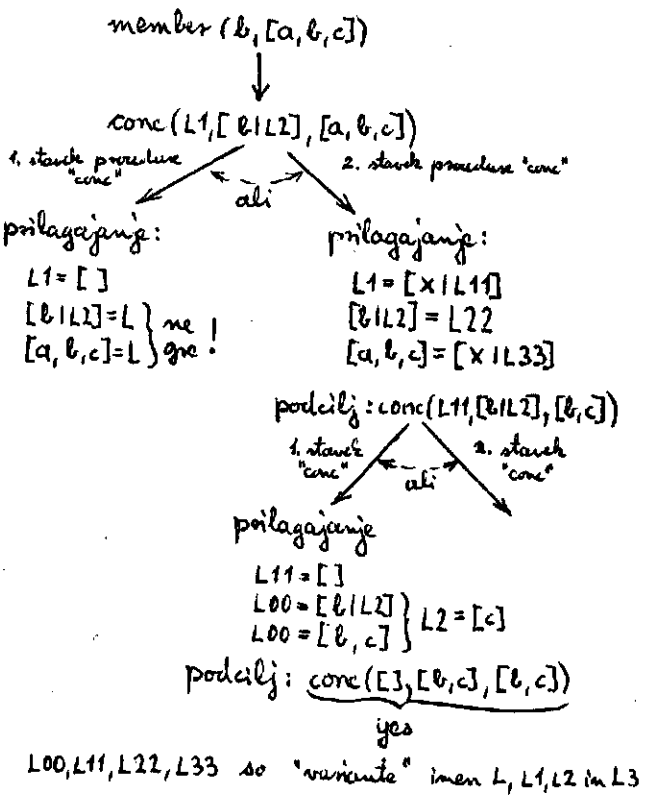
Ta zgled ilustrira nedeterminizem v Prologu: če je možnih več rešitev, Prologov interpreter generira vse, razen če tega na poseben način ne prepovemo. Definirajmo še relacijo

```
member(X,L)
```

ki je izpolnjena, če je X element seznama L. To relacijo lahko na nepostopkoven način definiramo takole: X je element seznama L, če eksistirata dva podseznama seznama L tako, da je glava drugega podseznama enaka X:

```
member(X,L) :- conc(L1,[X|L2],L).
```

Kako deluje ta presenetljiva rešitev, kaže slika 6.



Slika 6. Izvajanje procedure member ob vprašanju:
?- member(b,[a,b,c]).

Včasih že vnaprej vemo, da je generiranje alternativnih rešitev nepotrebno oz. nesmiselno. V takih primerih bi Prologov program po nepotrebnem trošil čas in prostor, zato imamo na voljo dodatni krmilni element, ki prepreči iskanje alternativnih rešitev. Ta krmilni element je t.i. "cut operator", ki ga označujemo s klicajem. Za primer definirajmo relacijo

```
max(A,B,M)
```

tako, da je M vrednost večjega od A in B. To lahko storimo z:

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

Kako se obnaša ta procedura, če zastavimo cilj:

```
?-max(2,3,M),M<2.
```

Prvi podcilj uspe tako, da M postane 3. Drugi podcilj postane 3<2 in ne uspe. Zato se izvajanje vrne na prvi podcilj in ga poskusi zadovoljiti še na kak drug način, torej poskusi z:

```
max(2,3,M) :- 3<2.
```

Seveda tudi to ne uspe in s tem ne uspe tudi celotni cilj. Iz naravne relacije max vemo, da lahko uspe kvečjemu eden izmed stavkov procedure max. Ko je uspel v gornjem poizkusu drugi stavek te procedure, je jasno, da tretji stavek ne more več uspeti. Zato je vračanje na ta stavek brezplodno. To vračanje lahko preprečimo z naslednjo dopolnitvijo procedure max:

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

Natančen pomen operatorja "!", imenovanega "cut", je: "!" se obnaša kot cilj, ki vedno uspe, pri tem pa povzroči kot stranski učinek, da se prepreči vračanje na prejšnje podcilje pred klicajem.

Naslednji primer nadalje pojasnjuje obnašanje operatorja "cut".

```
P :- Q1,Q2.
P :- R.
```

Ta procedura definira naslednjo logično zvezo med izjavami P,Q1,Q2 in R:

$$P \Leftrightarrow Q1 \wedge Q2 \vee R$$

Zdaj postavimo "cut" takole:

```
P :- Q1,!,Q2.
P :- R.
```

Logična zveza med izjavami se spremeni v:

$$P \Leftrightarrow Q1 \wedge Q2 \vee \neg Q1 \wedge R$$

4. ZAKLJUČEK

V sestavku smo pokazali nekatere značilnosti Prologa, predvsem tiste, zaradi katerih je način razmišljanja pri programiranju v Prologu drugačen kot pri programiranju v postopkovnih jezikih. Sintaksa predikatnega računa, na kateri temelji Prolog, je zelo posrečena za opis relacij med podatki. Po eni strani ta sintaksa omogoča delo s podatkovnimi strukturami, ne da bi pri tem eksplicitno uporabljali kazalce. Vsak sestavljeni podatkovni objekt v Prologu lahko ponazorimo z drevesom. Po drugi strani pa lahko to sintakso uporabljamo direktno kot jezik za postavljanje vprašanj (query language) za relacijski model podatkovnih baz. Že sam mehanizem prilagajanja zadošča za dokaj zahtevno iskanje relacijskih odnosov v bazi podatkov (to je v množici trditev, zapisanih v programu v Prologu).

Opisani primeri so povečini nakazali možnosti strukturiranja in iskanja podatkov. Nekatere druge operacije, ki zahtevajo več procesiranja na seznamskih, drevesnih in grafskih strukturah, so opisane v (Bratko, Gams 1981). Tam je poudarek na primerjavi Prologa z običajnimi jeziki, zato so te operacije implementirane še v Pascalu kot predstavniku konvencionalnih jezikov.

LITERATURA

1. Bratko I., Gams M. (1981, v pripravi) Prolog: procesiranje podatkovnih struktur in primerjava s Pascalom. Ljubljana: Informatika.
2. Kowalski R. (1974) Predicate logic as a programming language. Proc. IFIP Congress 74. North-Holland.
3. Kowalski R. (1979) Algorithm = Logic + Control. CACM, Vol. 22, No. 7, 572-595.
4. Pereira L., Pereira F., Warren D.H.D. (1978) User's Guide to DEC System 10 Prolog. University of Edinburgh, Department of Artificial Intelligence.
5. Roussel P. (1975) Prolog: manual d'utilisation. Raport interne GIAVER de Luminy, Universite d'Aix, Marseille.
6. Sammet J. (1969) Programming Languages: History and Fundamentals. Prentice-Hall.
7. Wirth N. (1976) Programs = Algorithms + Data Structures. Prentice-Hall.
8. Siklossy L. (1976) Let's Talk Lisp. Prentice-Hall.