SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Inteligentni sustavi

Uvod u programski jezik Prolog

Siniša Šegvić

Sadržaj

Poglavlje 1

Uvod

1.1 Organizacija vježbi

Cilj vježbi je upoznavanje sa programskim jezikom Prolog te mogućnostima njegove primjene na području umjetne inteligencije. Vježbe su zamišljene tako da se gradivo izloženo na auditornim vježbama, primjenjuje na rješavanje konkretnih zadataka u okviru laboratorijskih vježbi.

Svi izvedbeni detalji laboratorijskih vježbi, početak izvođenja, termini, raspored studenata po terminima, upute, dokumentacija, reference, preporučena implementacija Prologa te načini kolokviranja vježbi, se mogu naći na adresi:

`http://www.zemris.fer.hr/education/is/lv`

Molim obratiti posebnu pažnju na rokove za kolokviranje vježbi, jer nepoštivanje tih rokova povlači za sobom obavezan ponovni upis kolegija!

Konačno, konzultacije će se moći obavljati nakon auditornih vježbi, za vrijeme laboratorijskih vježbi, dok je za sve ostale termine potreban prethodni dogovor e-mailom.

1.2 Raspored tema

tjedan	auditorne vježbe	laboratorijske vježbe
1	uvod u Prolog, činjenice	-
2	pravila, sintaksa jezika	-
3	semantika jezika	uvodna vježba
4	liste	uvodna vježba
5	operatori	liste
6	aritmetika	liste
7	rez	operatori i aritmetika
8	baza podataka	operatori i aritmetika
9	osam kraljica	jednostavni problemi
10	ekspertni sustavi	jednostavni problemi
11	ekspertni sustavi, zadaci	pretraživanje
12	akumulatorski par, kontekst	pretraživanje
13	nepotpuno instancirani objekti	wumpus
14	-	wumpus
15	-	-

1.3 Prolog kao programski jezik za inteligentne sustave

Umjetna inteligencija je grana računarstva koja se bavi problemima koje ljudi još uvijek rješavaju bolje od računala. Primjer takvog teškog problema je razumijevanje prirodnog jezika. Tokom niza godina, istraživači su pokušavali razumjeti i formalizirati procese koji čine ono što nazivamo inteligencijom. Pokazalo se da su za prirodno i jednostavno izražavanje otkrivenih formalizama vrlo pogodni deklarativni programski jezici, koji omogućuju izražavanje problema u obliku skupa uvjeta, dok računalo pronalazi rješenje kao skup vrijednosti za koje su uvjeti zadovoljeni.

Deklarativni programski jezici (npr. SQL, Prolog) podrazumijevaju programiranje u dvije faze i to (1) opisivanje problema na način svojstven jeziku, i (2) upita kojim programu "kažemo" *što* točno želimo doznati. Naravno, da bi takav mehanizam bio moguć, deklarativni jezici moraju u sebi sadržavati implicitne algoritme za obradu što ograničava područje primjene jezika. Deklarativni programski jezici posebno su prikladni za izradu prototipova jer apstrahiraju izvedbene detalje niske razine kao što su rukovanje memorijom ili redoslijed izvođenja. Kratka usporedba loših i dobrih strana oba pristupa programiranju dana je u tablici 1.1.

deklarativno programiranje	**postupkovno programiranje**
• nekim problemima je prirodno pristupiti deklarativnim stilom jer se time postiže kraći izvorni kôd, kraći razvoj i lakše održavanje; • brzina izvođenja u eksperimentalnoj fazi razvoja često nije kritična pa je deklarativni pristup posebno prikladan za izradu prototipa.	• na postojećim arhitekturama računala, postupkovni programi u pravilu postižu brži i kraći izvršni kôd; • postupkovni jezici imaju veće područje primjene.

Tablica 1.1: Usporedba deklarativnog i postupkovnog stila programiranja

Deklarativni programski jezici se dijele na funkcijske i logičke. Prolog je najčešće korišteni logički jezik, dok su često korišteni funkcijski jezici Lisp, SML i Haskell. Tradicionalno, za pisanje "inteligentnih" programa u Americi često se koristi Lisp, dok se u Evropi uglavnom koristio Prolog.

Čisti deklarativni jezik po definiciji ne bi smio sadržavati tzv. popratne pojave (*engl.* side-effect) Popratna pojava u ovom smislu znači bilo kakvo mijenjanje stanja računalnog sustava na kojem se izvode, npr, pridruživanje (!). Iz toga proizlaze mnoga lijepa svojstva, npr, jednostavna formalna verifikacija programa ili mogućnost njegovog paralelnog izvođenja. Zbog takvih ambicija, programiranje u deklarativnim jezicima zahtijeva poseban mentalni napor, koji je tim veći što student ima veće iskustvo sa konvencionalnim, postupkovnim programiranjem. Mnoge konstrukcije koje su temelj postupkovnog programiranja, u deklarativnim jezicima naprosto su ilegalne (npr, "i=i+1"). Postavlja se pitanje, kako išta isprogramirati u jeziku koji ne podržava promjenu vrijednosti varijabli? Odgovor je najčešće rekurzija.

Dakle, kako bismo izračunali $X^n, n \in \mathbb{Z}$ u Prologu? Pa, rekli bismo da je $X^0 = 1$, te da za svaki $n > 0$ vrijedi da je $X^n = X * X^{n-1}$. Rješenje oblikujemo u obliku ternarne relacije, koja vrijedi ako je treći argument rezultat potenciranja prvog argumenta drugim.

Ispravnost programa provjerili bismo zadavanjem upita, npr, `xnan(5,3,X)`, na što bismo očekivali odgovor `X=125`.

```
xnan(X,0,1).   % X na 0 =1
xnan(X,N,Rez):-% X na N = X * X na N-1
  N > 0,
  N1 is N-1,
  xnan(X,N1,Rez1),
  Rez is X*Rez.
```

Kao ilustraciju široke zastupljenosti deklarativnog pristupa programiranju, razmotrimo implementaciju istog zadatka u jednom funkcijskom jeziku:

```
template <int X, int N>
struct xnan{
  enum {val=X*xnan<X,N-1>::val};
};
template <int X>
struct xnan<X,0>{
  enum {val=1};
};
```

Naravno, ne radi se o jeziku C++ nego o njegovom prevodiocu koji je Turing kompletan i može obavljati vrlo složene proračune za vrijeme prevođenja programa u C++-u. Zainteresirani čitatelj može prevesti gornji primjer uz sljedeći glavni program:

```
#include <iostream>
int main(){
  std::cerr <<xnan<5,3>::val <<std::endl;
  std::cerr <<xnan<4,2>::val <<std::endl;
}
```

Razmotrimo i rješenje istog problema izraženog u funkcijskom jeziku SML (*engl.* standard meta language). Opet, rekurzija je glavni mehanizam obavljanja repetitivnih zadataka.

```
n xnan2 (x, 0, r) = r
| xnan2 (x, n, r) = xnan2(x, n-1, r*x)
```

Na kraju valja reći da iz praktičnih razloga, gotovo niti jedan "deklarativni" jezik nije potpuno deklarativan, tj, dozvoljava stanoviti broj popratnih pojava. Ova odstupanja od ideala su vrlo važno svojstvo takvih jezika i iznimno ih je važno dobro znati jer će ona u mnogome određivati potrebni način razmišljanja i stil pri izražavanju problema u danom jeziku. U konkretnom slučaju, Prolog podržava dualnu semantiku po kojoj se program tumači i na deklarativan i na postupkovni način. Programi u Prologu se pišu uglavnom u deklarativnom stilu ali je uvijek potrebno imati na umu i postupkovni aspekt, kako interni mehanizam Prologa ne bi pri traženju rješenja ušao u beskonačnu petlju.

1.4 Uvod u Prolog - činjenice i pravila

Prolog je programski jezik za simboličku nenumeričku obradu. Temelji se na predikatnoj logici prvog reda i koristi ograničenu verziju klauzalnog oblika poznatu kao Hornov klauzalni oblik. Prolog je posebno pogodan za rješavanje problema koji se mogu opisati objektima i

relacijama među njima. Dva osnovna koncepta u Prologu su *činjenice* kojima se opisuju pojedinačne elementarne relacije, te *pravila* koja definiraju nove relacije na temelju postojećih. Činjenice i pravila se formalno opisuju temeljnim jedinicama programa, *stavcima*.

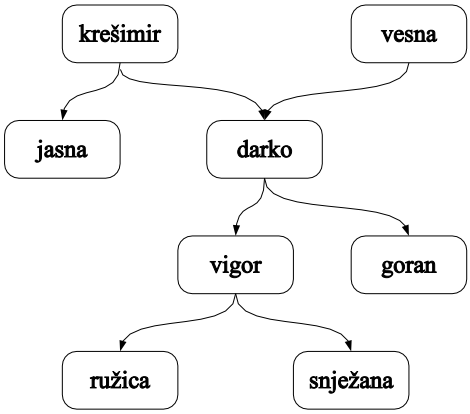
1.4.1 Opisivanje relacija

Upotreba činjenica u Prologu se može ilustrirati na primjeru relacije *roditelj*. Tako se, primjerice, činjenica da je Krešimir Jasnin roditelj, može predstaviti slijedećim stavkom:

```
roditelj(kresimir, jasna).
```

U tom stavku *roditelj* označava relaciju, dok *kresimir* i *jasna* označavaju objekte nad kojima ta relacija vrijedi. Definicija relacije može se proširiti dodatnim stavcima. Tako se porodično stablo neke obitelji da prikazati slijedećim programom:

```
roditelj(kresimir, jasna).
roditelj(kresimir, darko).
roditelj(vesna, darko).
roditelj(darko, vigor).
roditelj(darko, goran).
roditelj(vigor, ruzica).
roditelj(vigor, snjezana).
```



Prethodnom programu se u fazi upita mogu postavljati razni upiti. Primjerice, ako želimo saznati da li je Krešimir Ružičin roditelj, možemo postaviti upit:

```
?- roditelj(kresimir, ruzica).
No
```

Slijed znakova "?-" (prompt) označava da je Prolog spreman za upit, dok je "No" dobiveni odgovor. Prolog može davati i složenije odgovore od "da" ili "ne", što se može vidjeti na slijedećim primjerima (znak "%" označava da je ostatak linije komentar):

```
?- roditelj(X, ruzica).  % Tko je Ruzicin roditelj?
X=vigor
?- roditelj(kresimir, Y). % Tko je Kresimirovo dijete?
Y=jasna
Y=darko
```

Doseg varijable je jedan stavak, što se zajedno sa operatorom konjunkcije "," može koristiti za slaganje složenih uvjeta:

```
?- roditelj(kresimir, X), roditelj(X,Y). % Tko je Kresimirov unuk?
?- roditelj(X, ruzica), roditelj(X, snjezana).  % Da li su sestre?
```

Slijedi kratki sažetak korištenih svojstava Prologa:

- relacije se definiraju eksplicitnim navođenjem n-torki koje je zadovoljavaju;
- programiranje se odvija u fazama opisa problema i upita;
- opis problema (program) se sastoji od stavaka koji se zaključuju točkom;
- argumenti relacija mogu biti atomi i varijable;
- upiti se sastoje od više *ciljeva*; ako su ciljevi odvojeni zarezom, upit je njihova konjunkcija;
- Prolog može pronaći višestruke odgovore na zadani upit.

1.4.2 Opisivanje pravila

Pravilima se na temelju postojećih relacija definiraju nove. Može se reći da svako pravilo definira jednu uzročno posljedičnu vezu. Tako se na temelju relacije *roditelj* iz prethodnog odjeljka te novih dviju elementarnih unarnih relacija *musko* i *zensko*, mogu opisati nove relacije *majka* i *otac*.

```
% (uvrstiti relaciju roditelj
%  iz prethodnog odjeljka)
...
musko(kresimir).
zensko(vesna).
zensko(jasna).
musko(darko).
...
majka(X,Y):-
  roditelj(X,Y), zensko(X).
otac(X,Y):-
  roditelj(X,Y), musko(X).
```

```
faza upita:

?- otac(X,darko).
X=kresimir

?- otac(darko,X).
X=vigor
X=goran

?- otac(X,X).
No
```

Opći oblik stavka "`G :- T.`" se sastoji od *glave* `G` i *tijela* `T`. Glava stavka sadrži relaciju i objekte nad kojima se relacija definira, dok tijelo može biti prazno ili sadržavati niz ciljeva (kao kod upita) koji definiraju uvjete pod kojima relacija vrijedi. Kažemo da stavak koji ima nepraznu glavu i tijelo definira pravilo. Ako stavak nema tijela kažemo da definira činjenicu; konačno, s obzirom na svoju sintaksu, upit se može nazvati stavkom bez glave.

Kao i kod drugih programskih jezika, stavci Prologa se mogu proizvoljno formatirati. Ipak, obično se radi čitkosti programa glave stavaka počinju pisati od nultog stupca tekstualne datoteke, dok se pojedini ciljevi iz tijela uvlače za nekoliko znakovnih mjesta u desno. U nastavku su prikazani daljnji primjeri stavaka koji definiraju pravila:

```
% (uvrstiti relacije
% roditelj, musko i zensko)
...
djed(X,Z):-
  musko(X),
  roditelj(X,Y),
  roditelj(Y,Z).
```

```
% sestra nije simetrična relacija!
sestra(X,Y):-
  roditelj(Z,X),
  roditelj(Z,Y),
  razlicit(X,Y),
  zensko(X).
```

```
% Neke standardne relacije se mogu
% zapisivati i u operatorskom obliku.
% Jedna od takvih relacija je \=
% koja vrijedi ako joj se argumenti
% međusobno ne mogu unificirati.
% Tako je X \= Y ekvivalentno
% \=(X,Y).
razlicit(X,Y):-
  X \= Y.
```

Tokom *razrješavanja* upita, varijable se mogu nadomjestiti konkretnim objektima. Taj proces se naziva *vezivanje* varijabli i ireverzibilan je: za razliku od konvencionalnih programskih jezika, jednom vezana varijabla se ne može vezati za neki drugi objekt. Varijable koje se javljaju u glavi stavka su univerzalno kvantificirane, pravilo vrijedi **za svaku** kombinaciju varijabli iz glave stavka za koje vrijede uvjeti specificirani tijelom. Suprotno, varijable koje se javljaju *samo* u tijelu stavka su egzistencijalno kvantificirane, pravilo će vrijediti ako *postoji* neka kombinacija tih varijabli za koje su ciljevi iz tijela stavka zadovoljeni.

1.4.3 Rekurzivna pravila

Velik broj problema u računarstvu se rješava iterativnim pristupom. U konvencionalnim jezicima, ti problemi se opisuju programskim petljama. U Prologu se, međutim, oni prirodno opisuju rekurzijom, pa tako Prolog i nema jezične konstrukte za programske petlje. Najjednostavnija primjena rekurzije je modeliranje rekurzivnih relacija. Slijedeći ilustrativni program iz prošlog odjeljka, na temelju relacije *roditelj* može se definirati rekurzivna relacija *predak*.

```
% predak je roditelj...
predak(X,Y):-
  roditelj(X,Y).
% ...ali i roditelj od bilo kojeg drugog pretka.
predak(X,Z):-
  roditelj(X,Y),
  predak(Y,Z),
```

Relacija *predak* je opisana sa dva stavka što znači da vrijedi ako je zadovoljen bilo koji od njih. Više stavaka koji definiraju jednu relaciju čine *proceduru*. Redosljed stavaka u proceduri je *bitan*, što je definirano postupkovnom semantikom programa o kojoj će biti više riječi u sljedećim odjeljcima.

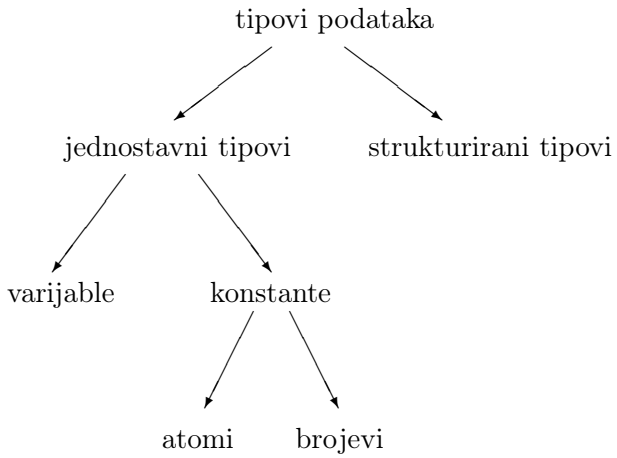
Poglavlje 2

Formalna definicija Prologa

2.1 Sintaksa jezika

2.1.1 Tipovi podataka

Prolog definira relativno malo tipova podataka, kao što se može vidjeti na sl.2.1. Tip podatka je jedinstveno određen sintaksom. Varijable se od atoma razlikuju po velikom početnom slovu, i mogu biti vezane za objekt bilo kojeg tipa uključujući i neku drugu nevezanu varijablu.



Slika 2.1: Klasifikacija tipova podataka Prologa

Atomi

Atomi mogu imati jedan od slijedeća tri oblika:

1. niz slova, brojeva i podvlaka koji započinje malim početnim slovom (`jedan`, `dva`, `x_Z5_y`);
2. niz specijalnih znakova (`<-->`, `...`, `::=`);
3. niz znakova unutar literala (`'Marjan'`, `'Slavonija'`)

Brojevi

Brojevi mogu biti cijeli ili s pomičnim zarezom, uz prikaz koji je uobičajen u ostalim programskim jezicima.

Varijable

Dozvoljeni oblik varijabli je niz slova, brojeva i podvlaka koji počinje ili velikim početnim slovom ili podvlakom. Leksički doseg varijable je jedan stavak. Ako se varijabla u stavku javlja točno jednom, može se označiti specijalnom neimenovanom varijablom "_". Ako se neimenovana varijabla pojavi u upitu, Prolog pri odgovoru na upit neće navesti njenu vrijednost.

```
imaDjecu(X):-
  roditelj(X,_).
netkoImaDjecu:-
  roditelj(_,_).  % nije isto sto i roditelj(X,X)!!!
?- roditelj(X,_).
X=kresimir.
...
```

Strukture

Strukture su objekti koji se sastoje od više komponenata (drugih objekata). Primjer takvog objekta bio bi `datum(17,listopad,2001)`. U tom primjeru, `datum` predstavlja ime strukture (tzv. *funktor*), dok su objekti odvojeni zarezom u zagradama komponente složenog objekta. Funktor je definiran imenom koje ima sintaksu atoma i mjesnošću, odnosno brojem argumenata. Strukture imaju istu sintaksu kao i ciljevi stavaka, a Prolog ih razlikuje prema kontekstu.

2.2 Semantika jezika

2.2.1 Unifikacija

Unifikacija je najčešće korištena operacija nad podacima. Dva izraza se međusobno mogu unificirati (prilagoditi) ako su ili identični, ili mogu postati identični uz odgovarajuća vezivanja varijabli. Slijede dva primjera unifikacije struktura u kojima `_GXXX` predstavlja novu varijablu koju u cilju unifikacije uvodi prevodioc:

```
?-datum(D,M,1983) = datum(D1,veljaca,G).
D=D1=_G283
M=veljaca
G=1983
```

```
?-datum(D,M,1983) = datum(D1,veljaca,G),
  datum(D,M,1983) = datum(15,M,G1).
D=15
D1=15
M=veljaca
G=1983
G1=1983
```

Preciznije, tok unificiranja dvaju podataka S i T je slijedeći:

1. ako su S i T konstante, prilagođavanje je moguće ako su S i T identični;
2. ako je S varijabla a T proizvoljan, prilagođavanje je moguće, i S se vezuje za T;
3. ako su S i T strukture, prilagođavanje je moguće ako su funktori identični, a pojedine komponente se mogu međusobno prilagoditi.

Unifikacija se može koristiti kao mehanizam za opisivanje odnosa među strukturiranim objektima. To se može ilustrirati na primjeru unarnih relacija `okomit` i `vodoravan`, čiji jedini argument je dužina opisana parom rubnih točaka:

```
okomita(duzina(t(X,Y), t(X,Y1))).
vodoravna(duzina(t(X,Y), t(X1,Y))).
```

2.2.2 Postupkovna semantika programa

Postupkovno značenje programa definira *kako* Prolog dolazi do odgovora na upit. Taj postupak je neformalno već opisan, kroz primjere rezultata obrade jednostavnijih upita, dok će se u ovom odjeljku on opisati detaljnije i formalno.

Predikati i relacije

Prilikom analize deklarativnog značenja nekog programa, pogodno je razmišljati na način da stavci Prologa modeliraju relacije. Međutim, u kontekstu izvođenja programa, prikladnije je razmišljati u terminima predikata — funkcija koje vraćaju istinu ili laž, tj. "vrijedi" ili "ne vrijedi". Odnos između relacije i odgovarajućeg predikata je jasan: ako je vrijednost predikata za neku n-torku parametara istinita, tada se može reći da za te parametre odgovarajuća relacija vrijedi. Prolog je jezik u kojem se deklarativno i postupkovno značenje isprepliću na način da je prilikom pisanja ili analize iste procedure često potrebno razmišljati i u terminima predikata i u terminima relacija.

Konceptualni opis izvođenja programa

Neka je, sasvim općenito, zadan upit $C_1, C_2, \ldots, C_n$. Tada se razrješavanje upita konceptualno može prikazati sljedećim postupkom:

1. formiraj stog ciljeva, tako sa se C_1 nađe na vrhu;
2. ako je stog prazan, vrati "vrijedi" (upit je uspješno razriješen);
3. a) skini prvi cilj sa stoga,

b) traži prvi stavak programa čija glava se s tim ciljem može unificirati,

c) primijeni unifikaciju i na tijelo pronađenog stavka,

d) stavi modificirano tijelo stavka na stog, tako da prvi cilj bude na vrhu stoga;

4. idi na korak 2.

Intuitivno najjasniji prikaz navedenog postupka dobiva se pomoću stabla izvođenja. Vrhovi stabla izvođenja prikazuju aktivne upite koji se predstavljaju stogom ciljeva, a bridovi njihove transformacije. Prilikom svake transformacije, cilj s vrha stoga se nadomješta tijelom stavka programa sa čijom glavom je taj cilj moguce unificirati, dok ostali ciljevi ostaju na stogu i prenose se u slijedeci vrh grafa. Radi jednostavnosti prikaza, u odredišni čvor se ne upisuje prvi cilj upita ukoliko mu je razrješavanje trivijalno. Za ilustraciju, razmotrimo stablo izvođenja za razrješavanje upita prema relaciji `teta`.

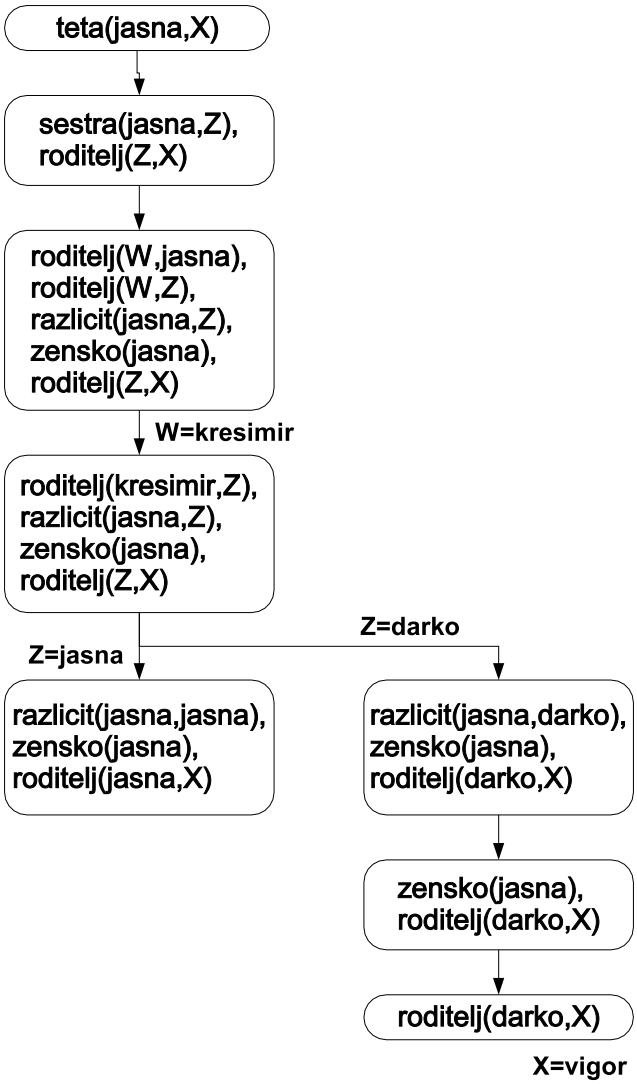
```
roditelj(kresimir, jasna).
roditelj(kresimir, darko).
roditelj(darko, vigor).
roditelj(darko, goran).


musko(kresimir).
zensko(jasna).
musko(darko).
musko(vigor).


sestra(X,Y):-
  roditelj(Z,X),
  roditelj(Z,Y),
  razlicit(X,Y),
  zensko(X).


teta(X,Y):-
 sestra(X,Z).
 roditelj(Z,Y).


% faza upita
?-teta(jasna,X).
X=vigor
```



Formalni opis izvođenja programa

Formalno, razrješavanje upita se može prikazati sljedećim pseudokôdom:

```
// procedura pokušava razriješiti upit zadan listom ciljeva C;
// ako uspije, vraća i listu unifikacija uz koje upit vrijedi.
// varijable koje počinju velikim slovom su kolekcije objekata.
bool razriješi(const Lista<Cilj>& C, Lista<Unifikacija>& U){
  if (C.prazan())
    return true;
  Cilj c0=C.prvi();
  C.odbaci_prvog();

  // pretpostavka:
  //   postoji globalni objekt Polje<Stavak> S
  //   koji sadrži sve stavke programa
  Kazalo k=S.početak();

  Lista<Cilj> Tijelo;
  Lista<Unifikacija> Uc0;
  while (pretraži(c0, k, Tijelo, Uc0){
    Lista<Cilj> Cpostojeci;
    primijeni_unifikaciju(Uc0, C, Cpostojeci);
    Lista<Cilj> Cnovi;
    spoji(Tijelo, Cpostojeci, Cnovi);

    Lista<Unifikacija> Unovi;
    if (razriješi(Cnovi, Unovi)){
      // uzimamo samo relevantne unifikacije - one u kojima
      // figuriraju varijable iz početne liste ciljeva
      reduciraj_unifikaciju(c0,Cpostojeci, Unovi);
      spoji(Uc0,Unovi, U);
      return true;
    }
  }
  return false;
}
// procedura traži stavak programa čija glava se može unificirati
// sa ciljem c, počevši od stavka određenog kazalom programa k:
// ako ga nađe vraća važeću unifikaciju U, novodobiveno kazalo k
// i prilagođeno tijelo stavka T
bool pretraži(
  const Cilj& c, Kazalo& k,
  Lista<Cilj>& T, Lista<Unifikacija>& U)
{
  while (k<S.kraj()){
    if (nađi_unifikaciju(S[k].glava(), c, U)){
      primijeni_unifikaciju(U, S[k].tijelo(), T);
      return true;
    }
    ++k;
  }
  return false;
}
```

```
// primjer

C: [
 sestra(jasna,Z)
 roditelj(Z,X)]
c0:
 sestra(jasna,Z)
C: [
 roditelj(Z,X)]


k: 0



k: 13
Tijelo:[
 roditelj(Z,X)
 roditelj(Z,Y)
 razlicit(X,Y)
 zensko(X)]
Uc0:[]
Cnovi:[
 roditelj(Z,X)
 roditelj(Z,Y)
 razlicit(X,Y)
 zensko(X)
 roditelj(Z,X)]









//---
```

Opisani postupak je nepotpun utoliko što u slučaju uspješnog razrješavanja upita ne predviđa mogućnost da korisnik zahtijeva traženje eventualnih daljnjih rješenja. Taj nedostatak nije odviše teško ukloniti pa se njegovo rješenje ostavlja kao zadatak za vježbu.

Praćenje postupka pretraživanja može se prikazati redosljedom pozivanja procedura `razriješi` i `pretraži`. U predloženom prikazu, svaki novi rekurzivni poziv procedure `razriješi` je radi jasnoće uvučen u desno za dva znakovna mjesta. Pozivi na istoj udaljenosti od desnog ruba tako opisuju izvođenje unutar jednog poziva procedure, odnosno unutar jednog okvira na stogu izvršavanja.

```
razriješi({teta(jasna,X)}, U)
 pretraži(teta(jasna,X), k, {sestra(jasna,Z), roditelj(Z,X)}, {})
 razriješi({sestra(jasna,Z), roditelj(Z,X)}, U)
  pretraži(sestra(jasna,Z), k,
     {roditelj(W,jasna), roditelj(W,Z),
      različit(jasna,Z), zensko(jasna)}, {})
  razriješi(
      {roditelj(W,jasna), roditelj(W,Z), različit(jasna,Z),
       zensko(jasna), roditelj(Z,X)}, U)
   pretraži(roditelj(W,jasna), k, {}, {W=kresimir})
   razriješi(
       {roditelj(kresimir,Z), različit(jasna,Z),
        zensko(jasna), roditelj(Z,X)}, U)
    pretraži(roditelj(kresimir,Z), k, {}, {Z=jasna})
    razriješi(
        {različit(jasna,jasna), zensko(jasna),
         roditelj(jasna,X)}, U)
     pretraži(različit(jasna,jasna), k, {}, {})
     (pretraži vraća false!)
    pretraži(roditelj(kresimir,Z), k', {}, {Z=darko})
    razriješi(
        {različit(jasna,darko), zensko(jasna),
         roditelj(darko,X)}, U)
     pretraži(različit(jasna,darko), k, {}, {})
     razriješi(zensko(jasna), roditelj(darko,X)}, U)
      pretraži(zensko(jasna), k, {}, {})
      razriješi({roditelj(darko,X)}, U)
       pretraži(roditelj(darko,X), k, {}, {X=vigor})
      razriješi vraća true uz U={X=vigor}.
     razriješi vraća true uz U={X=vigor}.
    razriješi vraća true uz U={X=vigor}.
   razriješi vraća true uz U={Z=darko, X=vigor}.
  razriješi vraća true uz U={W=kresimir,Z=darko,X=vigor}.
 razriješi vraća true uz U={Z=darko,X=vigor}.
razriješi vraća true uz U={X=vigor}.
```

Poglavlje 3

Temeljne tehnike programiranja

3.1 Liste

Lista je rekurzivno definirana podatkovna struktura koja se u prologu koristi vrlo često. Formalno, lista može biti ili prazna (oznaka `[]`), ili dvočlana struktura `.(Glava,Rep)`, gdje je `Glava` prvi element liste, a `Rep` ponovo lista koja sadrži ostale elemente početne liste. Elementi liste mogu biti proizvoljni objekti.

Radi jednostavnosti, dozvoljavaju se i alternativni zapisi, korištenjem uglatih zagrada i uspravne crte. Tako su slijedeći izrazi ekvivalentni:

- `.(G,T)` $\equiv$ `[G|T]`
- `[`l_1`|[`l_2`|[`l_3`|[]]]]` $\equiv$ `[`l_1`,`l_2`,`l_3`]`
- `[`l_1`,`l_2`|[`l_3`|[]]]` $\equiv$ `[`l_1`,`l_2`,`l_3`]`

3.1.1 Relacija pripadnosti elemenata listi

Listom se često modeliraju skupovi i operacije nad njima. Pripadnost je stoga temeljna relacija koju bi trebalo modelirati. Neka tražena relacija ima oblik `element(X, L)`, i neka vrijedi ako lista `L` sadrži objekt `X`. Tako bi prva dva upita iz sljedeće liste trebali biti razriješeni, a treći ne.

```
?-element(b,[a,b,c]).
yes
?-element([b,c], [a,[b,c]]).
yes
?-element(b, [a,[b,c]]).
no
```

Pokazuje se da je izvedba relacije `element` jednostavna:

```
element(X,[X|_]).
element(X,[_|Ostali]):-
  element(X,Ostali).
```

Alternativno i ekvivalentno, može se pisati i:

```
element(X,L):-
  L=[X|_].
element(X,L):-
  L=[_|Ostali],
  element(X,Ostali).
```

3.1.2 Povezivanje listi

Neka predikat za povezivanje listi ima oblik `povezi(L1,L2,L3)` i neka vrijedi ako se lista `L3` dobiva povezivanjem listi `L1` i `L2`. Tako bi upit `?-povezi([a,b], [c,d], [a,b,c,d]).` trebao uspjeti, za razliku od upita `?-povezi([a,b], [c,d], [a,b,a,c,d]).`. Kao i kod velikog broja ostalih problema, rješenje se gradi na način da se u prvom stavku ispituje trivijalan slučaj, dok se u drugom stavku dimenzionalnost problema smanjuje rekurzijom.

```
% L1=[] => L3=L2
povezi([],L,L).
% L1=[G1|R1] => L3=[G1|R1+L2]
povezi([G1|R1], L2, [G1|R3]):-
  povezi(R1,L2,R3).
```

Pokazuje se da predloženi predikat ima široku primjenu, kao što se može vidjeti iz slijedećih primjera:

```
?- povezi([a,[b,c],d], [a,[],b], L):-
L=[[a,[b,c],d,a,[],b]].
```

```
?- povezi(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=[]
```

```
?- povezi(Prije,[5|Poslije],[0,1,2,3,4,5,6,7,8,9]).
Prije=[0,1,2,3,4]
Poslije=[6,7,8,9]
```

```
% nova definicija predikata element
element1(X,L):-
  povezi(_,[X|_],L).
```

3.1.3 Dodavanje elemenata u listu

Sasvim općenito, liste se nalakše proširuju sprijeda. Odgovarajući predikat ima sučelje `dodaj(X,L,Nova)`, a zadatak mu je dodati element `X` na početak liste `L`, te rezultat unificirati sa listom `Nova`. Pokazuje se da je izvedba predikata trivijalna:

```
dodaj(X,L,[X|L]).
```

3.1.4 Brisanje elemenata liste

Neka predikat ima oblik `obrisi(X,Pocetna,Nova)`. U programiranju je uobičajena konvencija da se izlazni argumenti stave na začelje liste argumenata pa je semantika predikata jasna: lista `Nova` se gradi brisanjem objekta `X` iz liste `Pocetna`. U implementaciji ponovo razlikujemo trivijalni slučaj koji opisuje brisanje prvog elementa liste, te općeniti slučaj koji modelira postupno svođenje početnog upita na trivijalni:

```
obrisi(X,[X|Rep],Rep).
obrisi(X,[Y|Rep],[Y|RepBezX]):-
  obrisi(X,Rep,RepBezX).

% faza upita
?- obrisi(a,[a,b,a,a,],L).
L=[b,a,a]
L=[a,b,a]
L=[a,b,a]

?- obrisi(a,L,[1,2,3]).
L=[a,1,2,3]
L=[1,a,2,3]
L=[1,2,a,3]
L=[1,2,3,a]
```

Posljednji primjer upućuje na mogućnost korištenja predikata `obrisi` u izvedbi predikata `umetni`:

```
umetni(X,Lista,Nova):-
  obrisi(X,Nova,Lista).
```

3.1.5 Pronalaženje podlisti

Neka predikat za pronalaženje podlisti ima oblik `podlista(Podlista,Lista)` tako da bi upit `?-podlista([c,d,e],[a,b,c,d,e,f])` trebao uspjeti. Pokazuje se da je predikat moguće jednostavno ostvariti s dva poziva predikata poveži:

```
podlista(Podlista,Lista):-
  povezi(_,Sufiks,Lista),
  povezi(Podlista,_,Sufiks).
```

Opet se pokazuje da je predikat moguće koristiti i u kontekstu određivanja nepoznatih entiteta relacije:

```
?- podlista(P,[a,b,c]).
P=[]
P=[a]
P=[a,b]
...
```

3.1.6 Permutacije

Ponekad se javlja potreba za generiranjem permutacija zadane liste. Izvedba odgovarajućeg predikata se ponovo sastoji od trivijalnog slučaja koji je opisan prvim stavkom te općenitog slučaja u kojem permutacije zadane liste svodimo na permutacije njenog repa i tako smanjujemo dimenziju problema za jedan:

```
permutacija([],[]).
permutacija([Prvi|Ostali],P):-
  permutacija(Ostali,P1),
  umetni(Prvi,P1,P).

% faza izvođenja:
?- permutacija([a,b,c],P).
P=[a,b,c]
P=[b,a,c]
P=[b,c,a]
P=[a,c,b]
P=[c,a,b]
P=[c,b,a]
```

3.2 Operatorska sintaksa predikata i struktura

Operatorski prikaz je posebno pogodan za čitko predstavljanje čestih operacija:

$$2 * a + b * c, \tag{3.1}$$

U izrazu (3.1) + i $*$ predstavljaju operatore, dok su $2, a, b, c$ njihovi argumenti. Izraz (3.1) se u Prologu standardno može predstaviti strukturiranim objektom, u tzv. prefiksnom obliku sa zagradama:

```
+(*(2,a), *(b,c))
```
(3.2)

gdje su `+` i `*` funktori, a `2,a,b,c` atomi.

Zbog bolje čitkosti, Prolog dozvoljava i operatorski zapis kako strukturiranih objekata tako i predikata. Prologu se može reći da operatorski zapis pojedinih funktora sintaksno interpretira kao da su zadani prefiksnim oblikom sa zagradama. Neki operatori su unaprijed definirani kao npr. `+,-,*,/,=`, pa su izrazi (3.1) i (3.2) za Prolog ekvivalentni, ako programer ne odredi drukčije. Omogućavanje operatorskog zapisa utječe samo na dozvoljenu sintaksu izraza i nije povezano sa njegovom interpretacijom. U tom smislu je omogućavanje operatorskog zapisa donekle slično pretprocesorskim direktivama programskog jezika C.

Operatorski zapis funktora se omogućuje pozivom ugrađenog predikata `op`, čiji argumenti definiraju prioritet operatora, njegov tip, te funktor za kojeg se želi omogućiti operatorska interpretacija.

```
% stavak bez glave se izvršava prilikom učitavanja programa;
% 600 označava prioritet, a xfx tip operatora
:-op(600,xfx,ima).

% slijedeći stavak je ekvivalentan sa stavkom
% ima(marko,kesten).
marko ima kesten.

% deklaracija vrijedi za predikate...
?- X ima Y.
X=marko
Y=kesten

%...ali i za funktore struktura
?- ima(mirko,X)=Y ima loptu.
X=loptu
Y=mirko
```

Prioritet operatora je broj [1..1200] — veći broj označava viši prioritet u smislu da operator s najvećim prioritetom postaje glavni funktor izraza.

```
?- help(op).
  ⋮
(svi predefinirani operatori)
  ⋮
500 yfx +, -, ...
400 yfx *, /, ...
  ⋮

?-a+b*c=+(a,*(b,c)).
yes
```

Tako se izraz `a+b*c` interpretira kao `+(a,*(b,c))` jer `+` ima "veći" prioritet od `*`. Alternativno, može se reći da operator s manjim prioritetom "jače" veže (prioritet $\neq$ snaga vezanja).

Tip operatora može biti jedan od slijedečih:

- postfiksni: `xf`, `yf`;
- prefiksni: `fx`, `fy`;
- infiksni: `xfx`, `xfy`, `yfx`.

Kao što se može i pretpostaviti na temelju prethodne tablice, slovo `f` u oznaci tipa označava položaj operatora u odnosu na argumente. Slovo `x` u oznaci tipa se interpretira tako da prioritet odgovarajućeg argumenta izraza mora biti manji od proriteta operatora (inače će biti prijavljena sintaksna greška). Slovo `y` naprotiv dozvoljava proizvoljan prioritet odgovarajućeg argumenta. Prioritet argumenta je 0 osim ako se radi o drugom operatorskom izrazu; u tom slučaju prioritet argumenta jednak je prioritetu njegovog glavnog funktora. Izraz u zagradama također ima prioritet 0.

Za ilustraciju, može se razmotriti izraz `a-b-c`, kojeg obično razumijevamo kao `(a-b)-c`, a ne kao `a-(b-c)`. To se u Prologu može modelirati tako da se operator "`-`" definira s tipom `yfx`, tj. da mu desni argument ne može biti izraz istog prioriteta.

Upotrebom operatora je moguće značajno povećati čitkost programa. To se može ilustrirati na zapisu De Morganovog pravila $\overline{AB} \equiv \overline{A} + \overline{B}$. Bez operatorskog zapisa, to pravilo bi se moglo zapisati stavkom:

```
equivalence(not(and(A,B)), or(not(A), not(B))).
```

Upotrebom slijedećih definicija operatora,

```
:-op(800,xfx,<==>).
:-op(700,xfy,v).
:-op(600,xfy,&).
:-op(500,fy, ).
```

omogućuje se značajno kraći zapis stavka:

```
~ (A & B) <==> ~A v ~B.
```

3.3 Aritmetičke operacije

Prolog nije namijenjen za opisivanje numeričkih postupaka. Zato je podrška za numeričke operacije u Prologu relativno skromna, a temelji se na evaluacijskom predikatu `is`/2.

Predikat `is`/2 vraća istinu ako mu se prvi argument može unificirati sa aritmetičkom vrijednošću drugog argumenta. Pri tome drugi argument može biti i složeni aritmetički izraz, čija vrijednost se određuje ugrađenim procedurama koje su pridružene predefiniranim aritmetičkim funktorima (`+`, `*`, `pi`, `random`, `**`, `sqrt`). Budući da se vrijednost aritmetičkog izraza određuje konvencionalnim metodama, eventualne varijable u drugom argumentu predikata `is`/2 moraju biti vezane u trenutku poziva predikata. Nepoštivanje tog ograničenja dovodi do iznimke, koja obično rezultira nasilnim prekidom izvođenja programa.

```
% koje sve aritmetičke funkcije se mogu koristiti
% kao argumenti predikata is/2?
?- help(current_arithmetic_function).
?- current_arithmetic_function(X).

?- X=1+2
X=1+2
?- X is 1+2
X=3
```

Skup funktora koje predikat `is`/2 može evaluirati se može proširiti ugrađenim predikatom `arithmetic_function`/1.

```
% deklaracija aritmetičke funkcije
:-arithmetic_function(kvadrat/1).
% definicija aritmetičke funkcije
kvadrat(X,Rez):-
  Rez is X*X.

% kvadrat se pojavljuje na popisu
% poznatih aritmetickih funkcija ...
?-current_arithmetic_function(X).

?- X is kvadrat(1.2).
X=1.44
```

Predikati usporedbe

Pored predikata `is`/2, aritmetičku vrijednost izraza određuju i predikati usporedbe: `>`, `<`, `>=`, `=<`, `=\=`, `=:=`. Predikati usporedbe imaju dva argumenta, a vrijednost im je istinita ako su im aritmetičke vrijednosti operanada u odgovarajućoj relaciji.

Neka je primjerice programom definirana relacija `godiste(X,G)`, koja ljudima `X` pridružuje godine rođenja `G`. Tada bismo sve ljude koji su rođeni između 1950 i 1960 godine mogli pronaći slijedećim upitom:

```
?- godiste(Ime,Godina),
  Godina >= 1950,
  Godina < 1960.
```

Računanje najvećeg zajedničkog djelitelja

Euklidov algoritam za računanje NZD glasi:

$$nzd(x,y) = \begin{cases} x, & y = 0 \\ nzd(x \bmod y, y), & x > y \\ nzd(y,x), & x < y \end{cases}$$

Odgovarajući program bi bio:

```
nzd(X,0,X).
nzd(X,Y,D):-
  X>=Y, Y>0,
  X1 is X mod Y,
  nzd(Y,X1,D).
nzd(X,Y,D):-
  X<Y,
  nzd(Y,X,D).
```

Računanje duljine liste

Skica postupka je jednostavna: za svaku listu ćemo reći da joj je duljina za jedan veća od duljine njenog repa.

```
duljina([],0).
duljina([_|Rep],N):-
  duljina(Rep,N1),
  N is 1+N1.

?- duljina([a,b,[c,d],e],N).
N=4
```

Broj ponavljanja elementa u listi

Neka je zadatak odrediti broj pojavljivanja zadanog elementa u listi koja može imati proizvoljnu razinu gniježdenja. Primjerice, za upit `brojnost(a,[a,b,[c,[e],a],d,[]],X)`, očekujemo odgovor `X=2`.

Sučelje tražene procedure može biti: `brojnost(+Element, +Lista, -Rezultat)`. Oznaka '+' definira ulazni argument koji mora biti instanciran u trenutku poziva procedure. Suprotno, oznaka '-' dokumentira da je odgovarajući argument izlazni, tj. da može biti i nevezana varijabla.

```
brojnost(X,[Glava|Rep],Rez):-
  brojnost(X,Glava,Dg),
  brojnost(X,Rep,Dr),
  Rez is Dg+Dr.

brojnost(X,X,1).
brojnost(X,Y,0):-
  Y \= X,
  Y \= [_|_].
```

Različiti predikati "jednakosti"

Prolog definira cijeli niz predikata koji oblikom ili semantikom podjećaju na operator pridruživanja koji je uobičajen u konvencionalnim programskim jezicima. Kako bi se izbjegla zabuna, značenje svakog od tih predikata je sažeto u slijedećoj tablici:

notacija	naziv	semantika	napomena
`X = Y` (`X \= Y`)	unifikacija	vrijedi ako se `X` i `Y` mogu međusobno prilagoditi (unificirati)	simetričan
`X is Y` (-)	evaluacija	vrijedi ako se `X` može prilagoditi aritmetičkoj vrijednosti izraza `E`	asimetričan, sve varijable u `E` moraju biti vezane
E_1=:=E_2 (E_1=\=E_2)	aritmetička jednakost	vrijedi ako su aritmetičke vrijednosti izraza E_1 i E_2 jednake	simetričan, sve varijable u E_1 i E_2 moraju biti vezane
`X == Y` (`X \== Y`)	identičnost	vrijedi ako su `X` i `Y` identični	simetričan, unifikacija nad izrazima se ne obavlja

3.4 Upravljanje postupkom vraćanja

Kad Prolog tijekom traženja odgovora na zadani upit ne uspije razriješiti neki cilj, on se automatski vraća na mjesto poslednjeg izbora te pokušava taj cilj razriješiti na drugi način, svođenjem na tijelo sljedećeg, još neisprobanog stavka programa koji je na taj cilj primjenljiv. Iako je taj mehanizam vraćanja temeljno svojstvo Prologa, ponekad ga je korisno moći zabraniti. Zabrana vraćanja se najčešće koristi kada su alternative za rješenje problema međusobno isključive, kao što je to slučaj u primjeru modeliranja sljedeće matematičke funkcije:

$$f(x) = \begin{cases} 0, & x < 3 \\ 2{,}3 \leq & x < 5 \\ 3{,}5 \leq & x < 6 \\ 4{,}6 \leq & x \end{cases}$$

Pokazuje se da je jednostavna programska implementacija funkcije dosta neefikasna.

```
f(X,0):-X<3.
f(X,2):-3=<X, X<5.
f(X,3):-5=<X, X<6.
f(X,4):-6=<X.
```

Stavci koji definiraju predikat $f/2$ se međusobno isključuju, pa tako u slučaju da je zadan X veći od šest program razrješava dva redundantna cilja (`3=<X` i `5=<X`), dok se u slučaju kad je X manji od 3 izvršavaju 3 redundantna cilja (`3=<X`, `5=<X` i `6=<X`).

Spomenuti problemi se jednostavno rješavaju specijalnim predikatom reza ($!/0$), koji označava zabranu povratka po grani stabla u kojoj se predikat nalazi:

```
f(X,Y):-X<3,!,Y=0.\\
f(X,Y):-X<5,!,Y=2.\\
f(X,Y):-X<6,!,Y=3.\\
f(X,Y):-Y=4.\\
```

3.4.1 Formalna definicija reza

Neka je polazni cilj razriješen stavkom u čijem se tijelu nalazi rez. Ako cilj `!` dođe na red za izvođenje, cilj će uspjeti, a pored toga će se zabraniti vraćanje u sve dotada razriješene ciljeve stavka, te isto tako i u sve slijedeće stavke tog predikata.

Neka je zadan program:

```
c:-p,q,r,!,s,t,u.
c:-v.
a:-b,c,d.
```

Prilikom izvođenja prvog stavka, vraćanje na početak je moguće samo iz ciljeva `p`, `q`, `r`. Jednom kad se taj niz razriješi, neće se više razmatrati rješenje niza na neki drugi način, a također će biti onemogućena primjena drugog stavka programa; vraćanje unutar ciljeva `s`, `t`, `u` će pri tome biti dozvoljeno. Rez utiče samo samo na izvođenje predikata u čijoj se definiciji nalazi. U zadanom primjeru, rez je nevidljiv za treći stavak programa pa se unutar tog stavka vraćanje odvija normalno.

Kao druga ilustracija semantike reza, može se proučiti donji primer i odgovarajuće stablo izvođenja:

```
pred(X,Y):-p(X,Y).
pred(d,5).

p(X,Y):-q(X),!,r(X,Y).
p(c,b).

q(a).
q(b).

r(a,1).
r(a,2).
r(b,3).
```

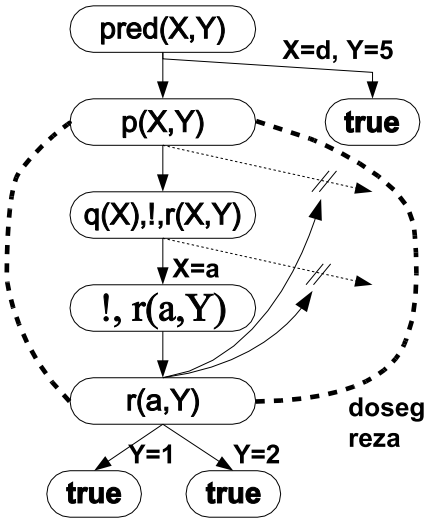
```
?- pred(X,Y).

X = a
Y = 1

X = a
Y = 2

X = d
Y = 5

No
```



Slika 3.1: Stablo izvođenja po zadanom upitu (vidi program)

3.4.2 Područje primjene reza

Glavna primjena reza je poboljšanje učinkovitosti programa. Prologu se rezom može izričito reći da ne pokušava razriješiti alternative za koje poznato da ne mogu uspjeti ili vode na

beskonačnu petlju. Rezom se prirodno izriču pravila koja se međusobno isključuju, dakle ekvivalent if-then-else konstrukcije iz konvencionalnih jezika:

```
procedura f:
  ako je P istina,
    onda pozovi Q;
    inače pozovi R.
```

$\equiv$

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

Međutim, rezove je potrebno vrlo pažljivo koristiti jer se njima mijenjaju ionako zamršeni odnosi između deklarativnog i postupkovnog značenja programa. U programu bez rezova, deklarativno znanje ovisi isključivo o sadržaju stavaka, a ne o njihovom redoslijedu. Unošenjem rezova u program, i deklarativno značenje programa postaje ovisno o redoslijedu, što čini programe teže razumljivim kao što se vidi iz priloženih primjera. Stoga rez valja koristiti samo kad je nužno potreban.

značenje bez reza		**značenje sa rezom**	
`p:-a,b.` `p:-c.`	$\equiv p \leftarrow (a \wedge b) \vee c$	`p:-a,!,b.` `p:-c.`	$\equiv p \leftarrow (a \wedge b) \vee (\overline{a} \wedge c)$
`p:-c.` `p:-a,b.`	$\equiv p \leftarrow c \vee (a \wedge b)$	`p:-c.` `p:-a,!,b.`	$\equiv p \leftarrow c \vee (a \wedge b)$

3.4.3 Primjeri

Predikat *not*/1

Dosada je predikat *not*/1 korišten bez naznaka o tome kakva bi bila njegova izvedba. Taj predikat se izvodi upotrebom predikata reza, te specijalnih predikata *fail*/0 i *true*/0. Kao što bi se moglo i pretpostaviti, vrijednost predikata *true*/0 je uvijek istinita dok predikat *fail*/0 uvijek vraća neuspjeh.

```
not(P):-
  P, !, fail;
  true.
```

Razvrstavanje u klase

Pretpostavimo da je zadatak razvrstati igrače nekog teniskog kluba u kategorije na temelju rezultata susreta održanih jedne nedjelje po sljedećem principu:

kategorija	**opis**
borac	igrači koji su i dobili i izgubili barem po jedan susret
pobjednik	igrač koji je u svim susretima pobijedio
sportaš	igrač koji je u svim susretima izgubio

Neka su rezultati susretâ opisani relacijom `pobijedio(X,Y)`, koja vrijedi ako je igrač X u izravnom dvoboju pobijedio igrača Y.

```
pobijedio(iva,tin).
pobijedio(boris,iva).
pobijedio(vesna,tin).
...

klasa(X,Y):-
  pobijedio(X,\_),
  pobijedio(\_,X), !,
  Y=borac.
klasa(X,pobjednik):-
  pobijedio(X,\_).
klasa(X,sportas):-
  pobijedio(\_,X).
```

Poglavlje 4

Jednostavne primjene Prologa

4.1 Baratanje strukturiranim podacima

Prolog je posebno pogodan za baratanje složenim strukturirama podataka jer omogućava jednostavnu implementaciju asocijativnog pristupa podacima. Upotrebu Prologa na tom području ilustrirat ćemo primjerom baze podataka o obiteljima te odgovarajućim operacijama pristupa zapisima te baze. Neka zapisi baze budu strukturirani objekti oblika:

```
porodica(Majka-Otac-Djeca).
```

Neka pri tome `Majka`, `Otac` i elementi liste `Djeca` imaju oblik:

```
osoba(Ime, Prezime, DatumRodenja, Zaposlenje).
```

Konačno, neka Zaposlenje ima oblik `posao(Tvrtka,Dohodak)`, te neka datum rođenja ima oblik `datum(d,m,g)`. Prirodni način predstavljanja baze podataka u Prologu je skup činjenica, pa ćemo svaku porodicu iz željenog skupa opisati jednim stavkom:

```
porodica(
  osoba(mirko,peric,datum(21,ožujak,1950),posao(mljekara,4000))-
  osoba(vesna,tomic,datum(23,rujan,1947),posao(toplana,4000))-
  [ osoba(suncana, peric, datum(28,listopad,1971), nezaposlen),
    osoba(darko, peric, datum(6,studeni,1975), nezaposlen)]).
```

Prolog je posebno prikladan za asocijativni pristup podacima, kada se objektu baze pristupa na temelju njegovog djelomično specificiranog sadržaja, neovisno o njegovoj fizičkoj lokaciji u bazi. Za ilustraciju se može razmotriti upit koji dohvaća sve obitelji iz baze u kojima se i majka i otac prezivaju Zoranić:

```
% pomoćni predikat prezime/2:
prezime(osoba(_,Prezime,_,_), Prezime).

?- porodica(O-M-_), prezime(O,zoranic), prezime(M,zoranic).
```

Slijedeći upit dohvaća imena i prezimena svih žena koje imaju troje ili više djece:

```
% pomoćni predikat ime/2:
prezime(osoba(Ime,_,_,_), Ime).

?- porodica(_-M-[_,_,_|_]), ime(M,Ime), prezime(M,Prezime).
```

U većim programima, fleksibilnost se postiže skrivanjem izvedbenih detalja podsustava od ostalih komponenti u sustavu. Podatkovna apstrakcija je važna tehnika za ostvarivanje takvih podsustava jer se njom omogućuje da korisnik podsustava može obavljati operacije nad podacima bez da mora znati detalje o fizičkoj izvedbi tih podataka. Time se postiže mogućnost nezavisnog razvijanja struktura podataka i procedura za njihovu obradu. U opisanom primjeru, pored već spomenutih `ime`/2 i `prezime`/2, u cilju postizanja podatkovne apstrakcije korisno bi bilo dodati i slijedeće predikate:

```
otac(P,O):-
  porodica(P),
  P=(O-_-_).
majka(porodica(_-O-_),O).
djeca(porodica(_-_-D),D).
dijete(P,D):-
  djeca(P,djeca),
  clan(D,Djeca).

otac(O):- otac(_,O).
majka(O):- majka(_,O).
dijete(O):- dijete(_,O).
postoji(O):-otac(O);
  majka(O);
  dijete(O).
```

```
dan(datum(D,_,_),D).
mjesec(datum(_,M,_),M).
godina(datum(_,_,G),G).

roden(osoba(_,_,D,_),D).
placa(osoba(_,_,_,posao(_,P)),P).
placa(osoba(_,_,_,nezaposlen,0).

% ukupni dohodak
% osoba iz liste
dohodak([],0).
dohodak([Prvi|Ostali],Suma):-
  placa(Prvi,Pp),
  dohodak(Ostali,Po),
  Suma is Pp+Po.
```

Koristeći opisano sučelje prema bazi podataka, mogu se postaviti slijedeći upiti:

```
% trazenje djece koja su mlada od 14 godina:
?- dijete(X), roden(X,D), godina(D,G), G>1987.

% trazenje ljudi koji su stariji od 30 godina,
% a zaraduju manje od 3000
?- postoji(O), roden(O,D), godina(D,G), G<1970, placa(O,P), P<3000.

% odredivanje ukupnog dohotka u porodici Mirka Perića
?- porodica(P), otac(P,O), ime(O,mirko), prezime(O,peric),
  majka(P,M), djeca(P,D), dohodak([O,M|D], Dohodak).

% traženje porodica s prosječnim dohotkom većim od 5000
?- ...
```

4.2 Problem osam kraljica

Razmatra se zadatak u kojem je potrebno razmjestiti n kraljica na šahovskoj ploči dimenzija $n \times n$, tako da se niti jedan par kraljica ne nalazi u položaju međusobnog napadanja. Skup svih razmještaja kraljica na ploči je ogroman — ima $\binom{n^2}{n}$ elemenata koje nije moguće sve isprobati. Razmatrat će se dva pristupa rješavanju problema koji uspijevaju reducirati skup rješenja s različitom efikasnošću.

4.2.1 Smanjenje dimenzionalnosti problema

Za svaki dozvoljeni razmještaj vrijedi da se u svakom stupcu i svakom retku nalazi točno po jedna kraljica. Rješenje stoga možemo predstaviti listom koordinata redaka kraljica, uz konvenciju da koordinata stupca odgovara položaju kraljice u listi.

(ilustracija za razmještaj $[2, 0, 3, 1]$ na ploči 4×4)

U traženom razmještaju, niti jedan par kraljica ne zauzima isti redak. Skup svih analiziranih razmještaja za ploču $n \times n$ se stoga može reducirati na skup svih permutacija liste $[0, 1, \ldots, n-1]$.

```
rješenje1(S):-
  permutacija([0,1,2,3,4,5,6,7],S),
  rješenje1a(S).

rješenje1a([]).
rješenje1a([Prva|Ostale]):-
  rješenje1a(Ostale),
  ne_napada(Prva,Ostale,1).
```

Prostor svih ispitivanih razmještaja je smanjen na $n!$ elemenata, koji se slijedno ispituju predikatom *rjesenje1a*/1. Predikat *rjesenje1a*/1 provjerava da li mu argument može biti parcijalno rješenje problema. Predikat `ne_napada(Redak,Lista,Udaljenost)` vrijedi ako kraljica u retku `Redak` ne napada neku od kraljica čiji su redci predstavljeni listom `Lista`, a stupci su im od stupca ispitivane kraljice udaljeni za `Udaljenost`, `Udaljenost`+1 itd.

```
ne_napada(_,[],_).
ne_napada(Redak,[Prva|Ostale],Udaljenost):-
  Redak-Prva =\= Udaljenost, %uzlazna dijagonala
  Prva-Redak =\= Udaljenost, %silazna dijagonala
  U1 is Udaljenost+1,
  ne_napada(Redak,Ostale,U1).
```

4.2.2 Otkrivanje nedozvoljenih djelomičnih razmještaja

Temeljna ideja ovog pristupa je rano otkrivanje nedozvoljenih položaja upotrebom tablica nepotrošenih redaka, stupaca te rastućih i padajućih dijagonala. Za svaku novu kraljicu koja se postavlja na ploču, provjerit će se da li se njeni redak stupac i dijagonale nalaze u listama koje predstavljaju spomenute tablice. Ako to nije slučaj, kraljica napada neku od već postavljenih kraljica pa razmještaj nije valjan te se može uštediti na ne razmatranju kraljica

koje još nisu postavljene. Prilikom postavljanja kraljice, naravno, koordinate redka, stupca i dijagonala se brišu iz odgovarajućih listi. Sva polja na nekoj uzlaznoj dijagonali imaju istu razliku koordinata retka i stupca Zato će uzlazne dijagonale biti označavane u skladu s tom razlikom, cijelim brojevima od -7 do 7. Analogno pravilo vrijedi za silazne dijagonale i zbroj koordinata retka i stupca polja koja toj dijagonali pripadaju. Silazne dijagonale će tako biti označene cijelim brojevima 0 do 14.

(ilustracija numeracije dijagonala, redaka i stupaca)

Problem se formalno može opisati kako slijedi: zadatak je naći osam legalnih četvorki (i, j, d_u, d_s) za koje vrijedi da se ni za jedan element četvorke ne koristi dva puta ista vrijednost. Izvedba u Prologu se temelji na proceduri `rješenje2(Razmještaj, Redci, Uzlazne, Silazne)`. Lista slobodnih stupaca se ne pojavljuje kao argument procedure, jer se zahtjev da se dvije kraljice ne nalaze u istom stupcu može zadovoljiti konvencijom koja ne smanjuje općenitost programa. Naime, za svako rješenje vrijedi da se u svakom stupcu javlja točno po jedna kraljica, pa se kraljice u listi `Razmještaj` mogu poredati u skladu sa rastućim vrijednostima koordinate stupca. Procedura *rješenje2*/4 će se pozivati iz sučeljne procedure *rješenje2*/1, čiji zadatak je izgraditi strukturu liste `Razmještaj`, odnosno odrediti početne vrijednosti listi `Redci`, `Uzlazne` i `Silazne`.

```
rješenje2(S):-
  S=[_/0,_/1,_/2,_/3,_/4,_/5,_/6,_/7],
  Redci=[0,1,2,3,4,5,6,7],
  Dijag_uzl=[-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],
  Dijag_sil=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14],
  rješenje2(S, Redci, Dijag_uzl, Dijag_sil).

rješenje2([],_,_,_).
rješenje2([Redak/Stupac|Rep], R, Du, Ds):-
  obriši(Redak,R,R1),
  U is -Redak+Stupac,
  obriši(U, Du, Du1),
  S is Redak+Stupac,
  obriši(S, Ds, Ds1),
  rješenje2(Rep, R1, Du1, Ds1).
```

4.2.3 Otkrivanje bezperspektivnih djelomičnih razmještaja

```
rješenje3(S,N):-
  interval(0,N, L),
  napravi(N,L,Ls),
  rješenje3w(S, Ls).

rješenje3w([], []).
rješenje3w([X|Tx], [Hs|Ts]):-
  member(X, Hs),
  obnovi(X, 1, Ts, Ts1),
  rješenje3w(Tx, Ts1).

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

```
obnovi(X,N,[S|Ts],[S1|Ts1]):-
  brisi(X, S, Sh),
  Xu is X+N,
  brisi(Xu, Sh, Shu),
  Xs is X-N,
  brisi(Xs, Shu, S1),
  S1 \= [],
  N1 is N+1,
  obnovi(X,N1, Ts,Ts1).

napravi(0,_,[]).
napravi(N, X, [X|R]):-
  N>0,
  N1 is N-1,
  napravi(N1, X, R).
interval(M, M, []).
interval(M, N, [M|R]):-
  M<N,
  M1 is M+1,
  interval(M1, N, R).
brisi(X, L, L1):-
  select(X,L,L1), !.
brisi(_, L, L).
```

4.2.4 Usporedba efikasnosti rješenja problema 8 kraljica

Efikasnost opisanih pristupa je uspoređena upotrebom slijedećih predikata za testiranje:

```
for(0,_).
for(N,Cilj):-
  N>0,
  Cilj,
  N1 is N-1,
  for(N1,Cilj).

test0:-
  time(for(1000,rješenje([0,4,7,5,2,6,1,3],8)).
test1:-
  time(for(1000,rješenje(_,8))).
test2:-
  time(rješenje(_,20)).
```

Rezultati postignuti upotrebom SWI Prologa v5.2.13 na računalu sa performansom od 1000 SpecInt2000 sažeti su u slijedećoj tablici:

predikat	test0	test1	test2
`rješenje1`	1.9s	0.3s	–
`rješenje2`	0.1s	0.1s	15s
`rješenje3`	0.2s	0.2s	3.5s

4.3 Ekspertni sustavi

Ekspertni sustav je program koji može obavljati posao stručnog savjetodavca za neko usko određeno područje. Značajna područja primjene su medicinska dijagnostika te nadgledanje i upravljanje složenim sustavima. Temeljne operacije koje tipičan ekspertni sustav obavlja su:

- **rješavanje problema** — ES se sastoji od mehanizma za prikaz znanja iz područja primjene te od procedura za primjenu tog znanja pri rješavanju problema;
- **obrazloženje rješenja** — poput "živih" stručnjaka, ekspertni sustav mora moći pojasniti svoje odluke i zaključke;
- **dijalog s korisnikom** — postupak pronalaska rješenja se odvija interaktivno, korisnik može utjecati na tok pronalaženja rješenja;
- **rad s nepotpunim i neizrazitim podacima** — u stvarnim problemima, ulazni podaci su često (u pravilu) nepotpuni i nesigurni: potrebno je zato omogućiti baratanje pravilima kao npr. “tjelesna temperatura malo veća od 37° upućuje na upalu pluća”.

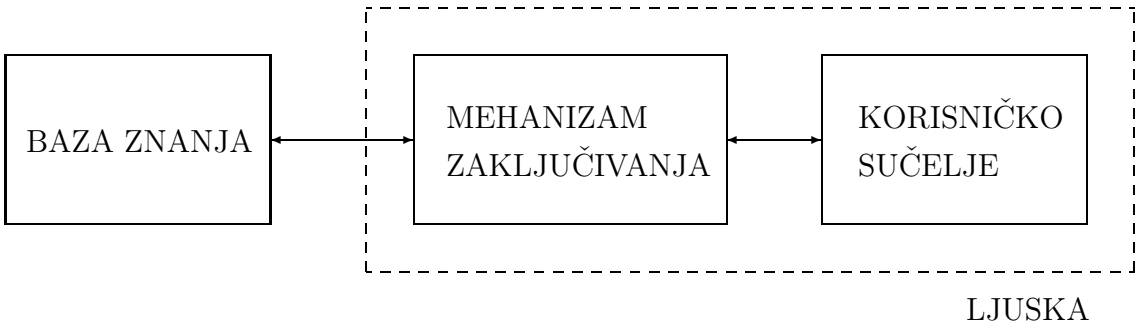
4.3.1 Struktura ekspertnog sustava

Na najvišoj razini, eksperni sustav se može podijeliti na slijedeća tri modula: bazu znanja, mehanizam zaključivanja i korisničko sučelje (sl.4.3.1). Baza znanja sadrži specifično znanje o području primjene: činjenice, pravila i metode za rješavanje problema iz domene ekspertnog sustava. Mehanizam zaključivanja definira postupak pronalaženja rješenja problema kojeg je zadao korisnik, na temelju elemenata baze znanja. Korisničko sučelje omogućava korisniku uvid u postupak zaključivanja i proširivanje baze znanja.

Često se mehanizam zaključivanja i korisničko sučelje zajednički nazivaju ljuskom ekspertnog sustava. Tako se sugerira da se ekspertni sustav sastoji od znanja o domeni (baza znanja) i generičkih postupaka za njegovu primjenu. Prednost takve podjele je u tome što je ljuska načelno neovisna o domeni ekpertnog sustava. Ista ljuska bi se u takvoj organizaciji mogla koristiti u različitim ekspertnim sustavima, u kombinaciji sa prikladnim bazama znanja. Naravno, sve baze znanja u tom slučaju moraju biti izvedene u skladu s formalizmom koji propisuje ljuska što u praksi može prouzročiti probleme.

Opći plan razvoja ljuske ekspertnog sustava se stoga sastoji od slijedećih koraka:

1. izbor formalizma za prikaz znanja;



Slika 4.1: Pojednostavljena struktura ekspertnog sustava

2. definiranje mehanizma zaključivanja, u skladu s odabranim formalizmom;

3. razvoj korisničkog sučelja;

4. dodavanje podrške za zaključivanje na temelju nepotpunog, nesigurnog ili neizrazitog znanja.

U okviru auditornih vježbi, razmatrat će se prvenstveno prvi korak i analize zahtjeva za drugi i treći korak, dok su odgovarajuće izvedbe predložene u okviru materijala za laboratorijske vježbe. Četvrti korak razvoja se u okviru ovih vježbi neće razmatrati zbog vremenskih ograničenja i složenosti područja.

4.3.2 Formalizam za prikaz znanja

Uvriježena je praksa da se znanje u ekspertnim sustavima prikazuje nizom produkcijskih pravila sa tzv. *if-then* strukturom. Primjeri takvih pravila su:

- `ako` vrijedi uvjet P, `onda` se može izvesti zaključak C;
- `ako` je trenutno stanje S, `onda` se izvodi akcija A;
- `ako` vrijede uvjeti U1 i U2, `onda` sud S nije valjan.

Pokazuje se da takav prikaz znanja ima slijedeća lijepa svojstva:

modularnost
svako pravilo predstavlja mali, relativno nezavisan dio znanja;

proširivost
baza znanja s takvom strukturom se lako proširuje novim pravilima;

jednostavno održavanje
pojedina pravila se u bazi znanja mogu jednostavno mijenjati, neovisno o ostalim pravilima (posljedica modularnosti);

transparentnost
ekspertni sustav može obrazložiti odluke odnosno rješenja; konkretno, mogu se ponuditi odgovori na dvije vrste korisničkih pitanja:

1. Kako si došao do tog zaključka?
   (takav podatak je potreban primjerice kod sustava koji imaju savjetodavnu ulogu pri upravljanju složenim postrojenjima npr. nuklearnom elektranom čiji pogrešan rad može imati nesagledive posljedice)
2. Zašto ti treba taj podatak?
   (ekspertni sustav može tražiti dodatne podatke koji su povezani sa značajnim izdacima, primjerice kod sustava za upravljanje geološkim istraživanjem — korisnik tada može odlučiti vrijedi li precizniji odgovor ekspertnog sustava dovoljno da opravda dodatna ulaganja)

U nekim problemskim domenama (npr. medicina), prevladavaju pojmovi stohastičke prirode (npr. "povišena temperatura"). Takav nedeterminizam se može izraziti tako da se pravila u ekspertnom sustavu opišu dodatnim vjerojatnosnim faktorom. Kao primjer, navodi se jedno pravilo iz ekspertnog sustava MYCIN (1976):

Ako:

1. infekcija je primarna bacteriemia
2. uzorak je uzet steriliziranom opremom
3. ulazno mjesto infekcije je probavni trakt

Onda:
Postoji utemeljena sumnja (0.7) da je uzročnik infekcije bacil iz porodice Bacteroides.

Iskustvo pokazuje da je za razvoj upotrebljivog ekspertnog jako bitno tzv."inženjerstvo znanja" (knowledge engineering), odnosno prikupljanje znanja konzultiranjem stručnjaka i njegovo oblikovanje u skladu sa odabranim formalizmom. To je u pravilu vrlo složen, zahtjevan i opsežan zadatak, pa će se u okviuu ovih vježbi razmatrati ekspertni sustav za područje koje je više manje svima poznato — razvrstavanje životinja na temelju njihovih karakteristika. Pokazuje se također da je engleski najpogodniji jezik za opisivanje pravila u bazi jer se njegovom upotrebom izbjegavaju problemi vezani uz složena pravila konjugacije glagola i deklinacije imenica.

Neka je opći oblik pravila u skromnoj bazi kojom će se ilustrirati izloženi koncepti zadan kako slijedi:

```
imePravila ::  if Uvjet then Zakljucak.
```

Pri tome se *Uvjet* može sastojati od više jednostavnih uvjeta povezanih operatorima and i or. Prednost Prologa pri modeliranju takvih pravila je u tome što ona mogu postati legalni stavci Prologa, uz odgovarajuću definiciju operatora `::`, `if`, `then`, `or`, `and`. Neka se stoga naša baza sastoji od slijedećih pravila, na kojima će se temeljiti daljnja diskusija:

```
rule1::
   if
      Animal has hair
      or
      Animal gives milk
   then
      Animal isa mammal.
```

```
rule3::
   if
      Animal isa mammal and
      (
         Animal eats meat
         or
         Animal has pointedTeeth and
         Animal has claws and
         Animal has forwardPointingEyes
      )
   then
      Animal isa carnivore.
```

```
rule5::
   if
      Animal isa carnivore and
      Animal has tawnyColour and
      Animal has blackStripes
   then
      Animal isa tiger.
```

4.3.3 Razvoj ljuske

Iako je odabrana sintaksa pravila iz baze donekle kompatibilna sa sintaksom Prologa, najjednostavniji način primjene tog znanja bi ipak bio korištenjem ugrađenog mehanizma zaključivanja. Tako bi se pravilo `rule1` uz odgovarajuću definiciju operatora `isa`, `has`, `gives` moglo izraziti kako slijedi:

```
Animal isa mammal :-
   Animal has hair,
   Animal gives milk.
```

Slično bi se mogla izraziti i pravila `rule3` i `rule5`. Neka je poznato da je Mirko tigar koji ima slijedeće osobine: ima dlaku, velik je i lijen te ima crne pruge. To bi se u Prologu moglo zapisati na slijedeći način:

```
mirko has hair.
mirko is lazy.
mirko is big.
mirko has blackStripes.
mirko has tawnyColour.
```

Sada se na temelju gornjeg programa i odgovarajućeg zapisa pravila `rule1`, `rule3` i `rule5` može s Prologom voditi slijedeći dijalog:

```
?- mirko isa tiger.
yes.

?- mirko isa leopard.
no.
```

Iako Prolog pravilno odgovara na zadane upite, i pri tome koristi bazu znanja zadanu pravilima sa *if-then* strukturom, predloženi program se ne može zvati ekspertnim sustavom iz slijedećih razloga:

1. Program ne možemo pitati *kako* je zaključio da je Mirko tigar.

2. Program ne možemo pitati ni *zašto* nije zaključio da je Mirko leopard.

3. Svu informaciju o domeni je potrebno unijeti u sustav prije postavljanja upita. Tako korisnik može zastraniti i zadavati nepotrebne podatke (npr. svojstva Mirka da je velik

i lijen koja su, sa stanovišta ove baze znanja, irelevantna), ne specificirajući pri tome neke bitne podatke (npr. da Mirko ponekad jede meso).

Zato postavljamo zahtjev da mehanizam zakjučivanja ljuske ekspertnog sustava omogućava slijedeći dijalog s korisnikom, na temelju opisane baze znanja za klasifikaciju životinja:

```
Question, please:
   mirko isa tiger.
Is it true:  mirko has hair?
   yes.
Is it true:  mirko eats meat?
   no.
Is it true:  mirko has pointedTeeth?
   yes.
Is it true:  mirko has claws?
   why.
To investigate, by rule3, mirko isa carnivore
To investigate, by rule5, mirko isa tiger
This was your question.

<...>
```

```
<...>

mirko isa tiger is true
Would you like to see how?
   yes.

mirko isa tiger
   was derivedBy rule5 from
      mirko isa carnivore
         was derivedBy rule3 from
            mirko isa mammal
               was derivedBy rule1 from
                  mirko has hair
                     was told
               and
            mirko has pointedTeeth
               was told
               and
            mirko has claws
               was told
               and
            mirko has forwardPointingEyes
               was told
         and
      mirko has tawnyColour
         was told
         and
      mirko has blackStripes
         was told
```

U prikazanom dijalogu, ekspertni sustav pita korisnika o jednostavnim činjenicama koje su mu potrebne u procesu zaključivanja. Korisnik može odgovoriti na dva načina:

1. tako da dâ traženu informaciju (*yes-no*);
2. tako da pita ekspertni sustav zašto mu je ta informacija potrebna (*why*).

Korisnik odgovara pitanjem (*why*) kada nije siguran da je shvatio pitanje ili kada traženu informaciju nije jednostavno (jeftino) dobiti. Kada ekspertni sustav konačno pronađe odgovor na početni upit, korisniku se nudi mogućnost ispisa lanca zaključivanja, odnosno odgovor na pitanje *kako* je sustav došao do rješenja.

Pogled u mehanizam zaključivanja bi se mogao dobiti i praćenjem izvođenja programa (`trace`), međutim, to praćenje je oblikovano prvenstveno za dijagnosticiranje pogrešaka u programima. Upotreba praćenja kao prozora u mehanizam zaključivanja bi stoga bila komplicirana jer bi korisniku pokazivala višak informacije.

4.3.4 Struktura programa

Izvedba ljuske ekspertnog sustava koja zadovoljava izložene zahtjeve je predložena u okviru materijala za laboratorijske vježbe. Slijedi kratak pregled najbitnijih izvedbenih detalja.

Glavni objekti koji se javljaju u okviru predložene izvedbe su:

- `Cilj` — upit koji se trenutno razmatra;
- `Odgovor` — lanac zaključivanja kojim je sustav pronasao odgovor (pozitivan ili negativan) na početni upit korisnika (odgovor na pitanje *kako?*);
- `Razlog` — dosadašnji tijek zaključivanja kojeg je potrebno pamtiti kako bi se moglo odgovoriti na pitanje korisnika *why* (zašto?).

Glavne procedure predložene ljuske su:

- `istrazi(Cilj,Razlog,Odgovor)` — zahtijeva se pronalaženje rješenja `Odgovor` na upit `Cilj`. Dosadašnji tijek zaključivanja opisan je objektom `Razlog`.
- `korisnik(Cilj,Razlog,Odgovor)` — sustav poziva tu proceduru kako bi od korisnika saznao nepoznati podatak `Cilj`. Sustav po potrebi odgovara na korisnikovo pitanje *zašto?* tako da pokaže dotadašnji tijek zaključivanja (`Razlog`). Relacija `upit` koja je definirana u sklopu baze znanja, određuje kakva sve pitanja ekspertni sustav smije uputiti korisniku.
- `prikazi(Odgovor)` — prikazuje pronađeno rješenje `Odgovor` u konciznom ili opsežnom obliku.
- `ekspert` — ta procedura objedinjuje pojedine dijelove ljuske ekspertnog sustava i koristi se u posljednja tri zadatka od kojih se sastoji laboratorijska vježba.

Poglavlje 5

Naprednije tehnike programiranja

5.1 Prenošenje akumulatorskog para

U konvencionalnim jezicima, parametri procedura mogu se prenositi na način da promjene tih parametara budu vidljive i u glavnom programu (tzv. call-by-reference). U Prologu se takva funkcionalnost postiže tehnikom u kojoj se jedan logički argument predikata opisuje sa dva fizička argumenta: ulaznim istanciranim te izlaznim neinstanciranim. Predikat na temelju ulaznog argumenta instancira izlazni sa novom vrijednošću logičkog argumenta, kao u sljedećem primjeru:

```
% ne ovako (jedan argument)...
uvećaj(X):-
  X is X+1. % podiže se iznimka, program se prekida!!
% nego ovako (dva argumenta):
uvećaj(X, Xr):-
  Xr is X+1.
```

Tehnika se posebno često koristi u rekurzivnim predikatima što se može ilustrirati na primjeru određivanja duljine liste. Sučeljni predikat `duljina(Lista, Duljina)` poziva radnu proceduru `d(Lista, VecPobrojano, Ukupno` tako da svoj drugi argument `Duljina` predstavi sa dva parametra, ulaznim `VecPobrojano` (koji se inicijalizira na nulu) i izlaznim `Ukupno`. Drugi parametar radne procedure `d`/3 odgovara broju do sada pobrojanih elemenata, dok se konačna vrijednost prepisuje iz drugog u treći argument kad se dođe do kraja liste. Može se reći da se tokom izvođenja procedure rezultat akumulira u ulaznom parametru da bi se na kraju konačno prepisao u izlazni, pa se tehnika naziva prenošenjem akumulatorskog para.

```
duljina(Lista,Duljina):-
  d(Lista, 0, Duljina).
d([], Pobrojao, Pobrojao).
d([_|Ostali], Pobrojao, Ukupno):-
  Pobrojao1 is Pobrojao+1,
  d(Ostali, Pobrojao1, Ukupno).
```

Pokazuje se da prenošenje akumulatorskog para često omogućava pisanje jednostavnijih i efikasnijih programa. Tako "klasična" izvedba predikata `okreni/2` ima složenost $O(n^2)$, za razliku od izvedbe uz pomoć prenošenja akumulatorskog para čija složenost je linearna.

```
okreni([],[]).
okreni([H|T],R):-
  okreni(T,To),
  povezi(To,[H],R).
```

```
okreni(Ulaz,Rezultat):-
  okreni(Ulaz, [], Rezultat).
okreni([], Rezultat, Rezultat).
okreni([Prvi|Ostali], VecOkrenuto, Rezultat):-
  okreni(Ostali,[Prvi|VecOkrenuto],Rezultat).
```

5.2 Argumenti konteksta

Problemi koji se u konvencionalnim jezicima rješavaju petljom, u Prologu se obično izražavaju rekurzivnim predikatom. Lokalne varijable čiji doseg obuhvaća petlju proceduralnog jezika, u Prologu se tada moraju izvesti kao argumenti rekurzivnog predikata (leksički doseg varijable je jedan stavak) koji se tada nazivaju argumentima konteksta. Za primjer, mogu se razmotriti dvije izvedbe procedure koja vektor iz prvog argumenta množi skalarom iz drugog, a rezultat sprema u vektor zadan trećim argumentom:

C++

```
#include<vector>
void mnozi(
  const std::vector<double>& X,
  double m,
  std::vector<double>& R)
{
  int n=X.size();
  R.resize(n);
  for (int i=0; i<n; ++i)
    R[i]=X[i]*m;
}
```

Prolog

```
mnozi([],_,[]).
mnozi([X|Xovi], M, [R|Rovi]):-
  R is X*M,
  mnozi(Xovi, M, Rovi).
```

Zapis algoritma u Prologu se čini neefikasnim jer se parametar M bespotrebno prenosi u svaki novi rekurzivni poziv. Međutim, postoji tzv. tehnika optimizacije posljednjeg poziva (last call optimization) kojom se rekurzivni postupak može prevesti u iterativan, ukoliko se rekurzivni poziv nalazi na kraju procedure. Drugim tehnikama optimizacije se može detektirati da je neki argument zapravo argument konteksta, pa konačni izvršni kôdovi za iterativnu petlju i rekurzivni predikat u Prologu u teoriji mogu biti identični.

Općenito, vrijedi sljedeće pravilo: odsječak konvencionalnog programa koji referencira N različitih varijabli koje nisu lokalne za taj odsječak, može se prevesti u predikat u Prologu sa N dodatnih argumenata konteksta. To pravilo je zgodno znati prilikom čitanja programa: ako u nekom predikatu postoje argumenti koji se nepromijenjeni javljaju u svim rekurzivnim pozivima, oni vjerojatno odgovaraju varijablama koje nisu lokalne za taj predikat.

Upotrebu argumenata konteksta možemo ilustrirati procedurom za selekciju elemenata ulazne liste koji su veći od zadane granice. U svim pozivima procedure `veliki`/3, argument `Granica` se prenosi nepromijenjen pa se odmah može pretpostaviti da se radi o argumentu konteksta, koji bi u iterativnoj petlji konvencionalnog programa bio deklariran izvan petlje.

```
veliki([],_,[]).
veliki([X|Xovi],Granica, Veliki):-
  X=<Granica, !,
  veliki(Xovi, Granica, Veliki).
veliki([X|Xovi],Granica, [X|JosVelikih]):-
  veliki(Xovi, Granica, JosVelikih).
```

Prilikom korištenja argumenata konteksta, može se javiti problem da ih je potrebno imati toliko puno da program zbog toga postaje nečitak i težak za održavanje. U takvom slučaju, standardno se koristi idiom u kojem se svi argumenti konteksta objedinjuju zajedničkim strukturiranim objektom. Pojedinim argumentima je prikladno pristupati pomoćnim predikatima, kako bi se olakšalo dodavanje novih argumenata u složeni objekt.

5.3 Nepotpuno instancirani objekti

Korisno svojstvo Prologa je da podržava objekte koji su samo djelomično instancirani. Takvi objekti u svojoj strukturi sadrže varijable koje tokom izvršavanja programa mogu biti potpuno instancirane, ili što je češći slučaj, unificirane sa novim djelomično instanciranim objektom. Da bi se to svojstvo moglo korisno primijeniti, potrebno je uz svaki djelomično instancirani objekt pamtiti i sve neinstancirane varijable koje se u njemu javljaju.

Za ilustraciju, može se razmotriti alternativni *razlikovni* zapis listi (*engl.* difference list), koji omogućava značajno brže obavljanje temeljnih operacija. Efikasniji prikaz liste se sastoji od dvočlane strukture `Lista-Rupa`, gdje je `Lista` "klasična" lista, čiji rep je neinstancirana varijabla `Rupa`. Tako se klasična lista `[1,2,3]` da zapisati kao `[1,2,3|X]-X`. Razlikovni zapis liste omogućava dopisanje elemenata na kraj liste u konstantnom vremenu, po cijenu mijenjanja izvorne liste:

```
povezi_rl(A1-Z1, A2-Z2, A3-Z3):-
  Z1=A2, % prvi argument predikata se mijenja!!
  A3=A1,
  Z3=Z2.

stavi_na_kraj_rl(X, A-Z, A1-Z1):-
  Z=[X|Z1], % drugi argument predikata se mijenja!!
  A1=A.
```

Kao druga ilustracija, može se razmotriti procedura za pojednostavljanje zbrojeva, koja bi trebala moći izraz `a+3+b+5+c+3+4+d` svesti na `15+a+b+c+d`. Aritmetički izrazi će se u toj proceduri predstaviti strukturom na čijem jednom kraju je neinstancirana varijabla `Rupa`. Sučeljni predikat `pojednostavni_sumu`/2 poziva radnu proceduru koja izdvaja simboličke pribrojnike od numeričkih. Simbolički pribrojnici se pri tome efikasno dodaju u izlaznu strukturu korištenjem varijable `Rupa` kao "pointera" na mjesto za proširenje strukture, dok se numerički pribrojnici jednostavno zbrajaju ugrađenim predikatom `is`/2.

```
pojednostavni_sumu(Suma, Jednostavna):-
  pojednostavni(Suma, Rezultat:Rupa, 0,N),
  (
    % imamo li numerički dio?
    N=\=0 ->
      Rupa=N, Jednostavna=Rezultat
    ;
      % imamo li simbolički dio?
      var(Rezultat) ->
        Jednostavna=0
      ;
        % Rupa se unificira s posljednjim simbolom!
        Rezultat=Jednostavna+Rupa
  ).
```

```
% radna procedura rekurzivno slijedi strukturirane podobjekte...
% a+b+c = +(+(a,b),c).
pojednostavni(A+B, Rezultat:Rupa, N0,N):-
  !,
  pojednostavni(B, Rezultat:Rupa1, N0,N1),
  pojednostavni(A, Rupa1:Rupa, N1,N).

% za numericki pribrojnik, rezultat je prazna struktura Rupa:Rupa
pojednostavni(C, Rupa:Rupa, N0,N):-
  number(C),
  !,
  N is N0 +C.

% novi simbolički pribrojnici se upisuju s lijeve strane,
% rupa ostaje na pocetku strukture
pojednostavni(X, Rupa+X:Rupa, N,N).
```

Bibliografija

[Bratko01] Ivan Bratko, *Prolog: programming for artificial intelligence*, Addison-Wesley Longman Publishing Co., Inc., 3 edn., 2001.

[O'Keefe90] Richard A. O'Keefe, *The craft of Prolog*, MIT Press, 1990.

[Russel95] Stuart J. Russel and Peter Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Upper Saddle River, New Jersey, 1995.

[Shoham94] Yoav Shoham, *Artificial Intelligence Techniques in Prolog*, Morgan Kaufmann Publishers, San Francisco, California, 1994.