PROLOG

Željko Vrba

veljača 2000.

Sadržaj

Predgovor

Ovaj dokument se može smatrati vrlo kratkim uvodom u Prolog. Iako kratak, ipak pokriva i neke napredne stvari, poput reza. Tekst je baziran na auditornim vježbama kolegija "Inteligentni sustavi" koje je u akademskoj godini 1999./2000. održao Siniša Šegvić na Fakultetu elektrotehnike i računarstva u Zagrebu. Neke stvari sam i sam dodao, poput metoda pretraživanja ili pregleda ugrađenih predikata (vrlo često sam bio svjedokom kako ljudi otkrivaju toplu vodu, umjesto da upotrijebe ugrađeni predikat). Slijedi **upozorenje** (samo za FER-ovce):

> **Profesor dr. sc. Slobodan Ribarić, asistent Siniša Šegvić, Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave (ZEMRIS) kao niti Fakultet elektrotehnike i računarstva (FER) nisu ni na kakav način sudjelovali u izradi ovog dokumenta. Ovaj dokument je rezultat isključivo moje vlastite inicijative i želje da budućim generacijama FER-ovaca (kao i drugim zainteresiranima) olakšam svladavanje Prologa. Drugim riječima, ako je iz ovog dokumenta izostavljeno nešto što je obrađeno na predavanjima, auditornim vježbama ili labosima, ne znači da se to ne mora znati. Nedostatak tog materijala u ovom dokumentu NIJE i NE MOŽE biti izgovor za neznanje na ispitu ili labosu.**

Gotovo nijedan primjer koda nije isproban, pa ako ne radi...žao mi je, tough luck! Javite se mailom, ali *jedino* u slučaju da ste popravili kod tako da radi! Moja e-mail adresa je

`mordor@fly.srk.fer.hr`

Zašto bi netko htio naučiti Prolog? Nemam pojma. Kolegij na FER-u je uključivao i vježbe na kojima su se razni zadaci trebali riješiti u Prologu. Međutim, svaki od tih zadataka je puno kompliciranije i teže riješiti u Prologu nego u C-u. Da ne govorim da se kod sporije izvršava, troši više memorije itd. U svakom slučaju, nakon napisanih cca. 600 linija koda i razno raznih programa, nisam se uvjerio u prednost Prologa nad C-om ili nekim drugim "klasičnim" jezikom. Štoviše, nakon mukotrpnog rješavanja u Prologu, odmah mi je pala na pamet vrlo jednostavna implementacija u C-u koju sam vrlo često realizirao u 2-5 puta kraćem vremenu nego što mi je trebalo da natjeram Prolog program da radi. Jedina prednost Prologa je što su programi *kratki*. Ali isto tako i vrlo nerazumljivi.

Odjeljak 3.4 nije zanimljiv za širu javnost koja možda želi naučiti Prolog zato što je to samo generalna diskusija ekspertnih sustava. Pripadajući kod je predugačak da bi bio uključen u ovaj dokument. Taj odlomak je ipak uključen radi kompletnosti za buduće generacije FERovaca koje će se mučiti Prologom.

Svi primjeri u ovom dokumentu rađeni su za SWI Prolog. Relativno dobar (besplatan) Prolog uz vrlo malo bugova ali jedan koji jako živcira: kad ga se na Unixu prvi put prekine iz beskonačne petlje sa `^C`, na slijedeće prekide uopće ne reagira. Drugi put iz beskonačne petlje možete izaći samo sa `^\` ili `kill` naredbom shella. Ovaj Prolog preporučujem FER-ovcima jer se labosi rade upravo na SWI Prologu, a programi koje je asistent napisao koriste upravo one predikate koji se razlikuju u GNU i SWI Prologu (vidi dolje), pa bi bilo potrebno mijenjati source programa.

Odlična besplatna implementacija Prologa je GNU Prolog. Odličan Prolog, nema gore spomenuti bug, kompajlira u native-code, dakle i vrlo brzo izvršava programe. Jedina je mana što neki predikati iz SWI

Prologa fale, a neki se jednako zovu, ali imaju različitu semantiku. Osim toga ima ugrađen i finite-domain solver. Ne znam puno o tome, ali primjeri su vrlo impresivni (vrlo malo pisanja za rješavanje prilično složenih problema). Ovu implementaciju Prologa **preporučam svima**.

Za kraj, ako *zaista* želite naučiti Prolog, nabavite puno živaca, odreknite se života za slijedeća dva mjeseca i počnite rješavati probleme. Dobar izvor problema koji prate ovaj tekst je slijedeći URL:

`http://www.zemris.fer.hr/~ssegvic/prolog`

Na kraju, **copyright**:

Popis predikata

Ovaj popis predikata sadrži sve predikate koji se koriste ili je dan njihov kod u ovom dokumentu. Ovdje su uključeni i ugrađeni predikati. Ukoliko je broj stranice **boldan**, tada se na toj stranici nalazi definicija predikata. Na ostalim stranicama se nalaze programi u kojima se navedeni predikat koristi.

1 Uvod u Prolog

Dva osnovna pristupa pri projektiranju programske podrške za inteligentne sustave su proceduralno i deklarativno programiranje. Pri deklarativnom programiranju se koriste jezici LISP[1] i Prolog.[2] Američki istraživači i industrija tradicionalno koriste LISP, a u Europi se koristi Prolog. U novije vrijeme Prolog je sve popularniji i u Americi pa se čini da se radi o jeziku izbora za područje AI.[3]

Odnos deklarativnog i proceduralnog programiranja može se slikovito izreći na slijedeći način: Neka je problem koji treba riješiti predstavljen labirintom, a izlaz iz labirinta je rješenje problema. *Proceduralni program* bi tada odgovarao *nizu uputa* za kretanje po labirintu, a *deklarativni* program bi bio *mapa* labirinta.

Deklarativni programski jezici u sebi sadržavaju mehanizme za obradu pa se programom specificira *što* se želi postići, a ne *kako* do tog rezultata doći. Rezultat oba programa (proceduralnog i deklarativnog) je isti – put od ulaska u labirint do izlaska.

1.1 Jednostavan program

Prolog je programski jezik pogodan za simboličku (nenumeričku) obradu, a posebno za rješavanje problema koji se mogu opisati objektima i relacijama među njima (što vodi na opis problema grafom), kao na primjer činjenica da je Krešimir Lucijin roditelj. Ta činjenica se u Prologu može predstaviti slijedećim *stavkom*:

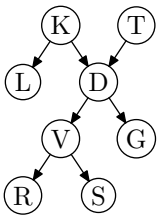
[1] LISP = LISt Processing
[2] Prolog = PROgramming in LOGic
[3] AI = Artificial Intelligence; umjetna inteligencija

```
roditelj(kresimir, lucija).
```

Ovdje je `roditelj` *ime relacije*, a `kresimir` i `lucija` su *atomi*. Atomi počinju malim slovom.

Porodično stablo (vidi **sliku 1.1**) neke obitelji može se opisati slijedećim programom (Program u Prologu se sastoji od *činjenica* i *pravila*):

```
roditelj(kresimir, darko).
roditelj(tereza, darko).
roditelj(darko, vigor).
roditelj(darko, goran).
roditelj(vigor, ruzica).
roditelj(vigor, suncana).
```



Slika 1.1 Porodično stablo

Nakon unosa programa mogu se postavljati upiti,[4] na primjer:

```
?- roditelj(X, ruzica).
   X = vigor

?- roditelj(kresimir, Y).
?- roditelj(kresimir, X), roditelj(X, Y).
?- roditelj(X, darko), roditelj(X, lucija).
```

Prvi upit pita tko su Krešimirova djeca, drugi upit daje odgovor tko je Krešimirov unuk, a treći upit pita da li su Darko i Lucija brat i sestra. U ovom primjeru su korištene *varijable*: imena koja počinju velikim slovom su varijable.

U Prologu se relacije mogu definirati eksplicitnim navođenjem n-torki koje relaciju zadovoljavaju. Programiranje se odvija u fazama opisa problema i upita. U fazi upita postavljaju se pitanja o problemu opisanom u fazi opisa.

Opis problema u Prologu ("program") sastoji se od *stavaka*; *svaki stavak se zaključuje točkom* (`.`). Argumenti relacija mogu biti atomi i varijable. Upiti se sastoje od više ciljeva; ako su ciljevi odvojeni zarezom (`,`), upit je njihova konjunkcija (logičko I). Ako ima više mogućih odgovora na upit, Prolog ih može potražiti.

1.1.1 Proširivanje programa s pravilima

Pravilima se na temelju postojećih relacija modeliraju nove. Svako pravilo pri tome definira jednu uzročno-posljedičnu vezu. Na primjer, porodično stablo (od prije) proširujemo informacijom o spolu ljudi:

[4] ?- je *prompt* Prologa

```
spol(kresimir, muski).
spol(tereza, zenski).
spol(lucija, zenski).
    ...
```

Na temelju relacije `spol/2`[5] možemo definirati nove relacije `musko/1` i `zensko/1` slijedećim stavcima:

```
musko(X) :- spol(X, muski).
zensko(X) :- spol(X, zenski).
```

Ovdje `:-` označava *grananje* (može se čitati kao "ako"). `musko` je *posljedica* (glava) stavke, a cilj desno od `:-` je *uzrok* (tijelo) stavke. Svaki stavak ima *glavu* i *tijelo*. **Glava** stavka je modelirana relacija (nova). **Tijelo** se sastoji od niza *ciljeva* (sintaksa im je ista kao i kod upita). **Stavak** može biti *činjenica* ili *pravilo*. Činjenice imaju samo glavu. Još neki primjeri pravila:

```
majka(X, Y) :- zensko(X), roditelj(X, Y).
djed(X, Z) :- musko(X), roditelj(X, Y), roditelj(Y, Z).
```

Stavci u izvornom programu mogu se proizvoljno formatirati, međutim, obično se glava stavka počinje pisati od 0. kolone, a pojedini dijelovi tijela se uvlače za nekoliko znakovnih mjesta:

```
sestra(X, Y) :-
    zensko(X),
    roditelj(Z, X), roditelj(Z, Y),
    \=(X, Y).
```

`\=` je relacija ugrađena u Prolog; dvije varijable su u relaciji `\=` ako nisu vezane za isti objekt.

1.1.2 Rekurzivno definirana pravila

Rekurzija je temeljni pristup programiranju u Prologu. Kao primjer, modelirat ćemo relaciju `predak/2` na temelju relacije `roditelj/2`:

```
predak(X, Z) :-
    roditelj(X, Z).

predak(X, Z) :-
    roditelj(X, Y),
    predak(Y, Z).
```

Relacija `predak/2` je definirana sa dva stavka i vrijedi ako je zadovoljen *bilo koji od njih*. **Proceduru** čini više stavaka koji definiraju jednu relaciju.

Apstrakcija relacija `roditelj/2` i `predak/2` je model općeg algoritma za pretraživanje acikličkog usmjerenog grafa.

Tokom izvođenja programa Prolog istražuje da li korisnikov upit sljedi iz činjenica i pravila zadanih programom. Ako se u upitu javljaju varijable, Prolog će potražiti takve objekte za koje bi upit slijedio iz programa.

[5] Oznaka /2 govori da relacija prihvaća dva argumenta

1.1.3 Znanje programa u Prologu

Sasvim općenito, programi sadržavaju *deklarativno* i *proceduralno* znanje. Primjer deklarativnog znanja kod Prologa su definicije relacija, dok je proceduralno znanje sadržano u algoritmima pretraživanja podatkovnih struktura kojima se relacije interno predstavljaju.

Osnovna karakteristika Prologa kao deklarativnog jezika je da se često može programirati i bez specifičnog proceduralnog znanja. Dakle, za neke primjene je jednostavnije programirati u Prologu nego u nekom proceduralnom jeziku. Na žalost, samo deklarativno znanje nije dovoljno za rješavanje nekih problema jer kod tih problema malo proceduralnog znanja puno utječe na efikasnost programa.

1.2 Formalna definicija Prologa

1.2.1 Tipovi podataka

Prolog od tipova podataka razlikuje **strukture** i **jednostavne objekte**. Jednostavni objekti se pak dijele na **varijable** i **konstante**. Konstante mogu biti **atomi** i **brojevi**. Prolog tip podatka raspoznaje *po sintaksi* (nema deklaracija).

1.2.1.1 Atomi i brojevi

Atomi mogu imati jedan od slijedeća tri oblika:

- Niz slova, brojeva i `_` (underscore) *koji počinje malim slovom*
- Niz specijalnih znakova, npr. `<--->`, `...`, `=`, `;`
- Niz znakova unutar jednostrukih navodnika (`'`), npr: `'Slovenija'` (ako je unutar literala, smije započeti i velikim slovom ili `_`)

1.2.1.2 Varijable

Imena varijabli se sastoje od slova, brojeva i `_`, a *obavezno započinju sa* `_` ili *velikim slovom*. Ukoliko se varijabla u stavku javlja *točno jednom*, može se označiti specijalnom neimenovanom varijablom `_`. Ukoliko se neimenovana varijabla pojavi u upitu, Prolog pri odgovoru ne navodi njenu vrijednost. **Leksički doseg** varijable je *stavak*.

Primjer upotrebe neimenovane varijable:

```
ima_dijete(X) :-
    roditelj(X, _).
```

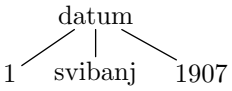
1.2.1.3 Strukture

Strukture su objekti s više komponenata, na primjer:

```
datum(1, svibanj, 1907)
```

Ovdje je `datum` ime strukture,[6] a elementi u zagradi su **komponente**.

[6] functor



Slika 1.2 Struktura kao stablo

Općenito, struktura je definirana **imenom** (ima sintaksu atoma) i **brojem argumenata**.[7] Strukture se mogu dobro predstaviti stablom (vidi **sliku 1.2**).

1.2.2 Unifikacija

Unifikacija (prilagođenje) je temeljna operacija nad objektima. Dva izraza se mogu međusobno **unificirati** ako su:

- identični
- ako se varijablama koje se u izrazima javljaju mogu pridijeliti vrijednosti tako da izrazi postanu identični

Primjer:

```
?- datum(D, M, 1983) = datum(D1, svibanj, Y).
   D = D1 = _G283
   M = svibanj
   Y = 1983
```

Druga linija pokazuje da su `D` i `D1` svedene na neku istu internu varijablu. Vidimo da operator `=` ne odgovara samo da li se dva izraza mogu unificirati, već tu unifikaciju *i provede*.

Postupak prilagođavanja dva objekta `S` i `T` se precizno može opisati na slijedeći način:

- Ako su `S` i `T` konstante, prilagođavanje je moguće ukoliko su `S` i `T` identični
- Ako je `S` varijabla, a `T` proizvoljan objekt, tada je prilagođavanje moguće, a `S` postaje **vezan** s vrijednošću `T`
- Ako su `S` i `T` strukture, unifikacija je moguća ako obje imaju isti funktor te se pojedine komponente daju unificirati

1.2.3 Deklarativno znanje programa

Program

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

ima slijedeću deklarativnu interpretaciju: "P vrijedi ako vrijede Q i R" , ili alternativno, "iz Q i R slijedi P". Formalno, deklarativno znanje za proizvoljnu tvrdnju govori da li iz programa slijedi njena istinitost.

Deklarativno znanje danog programa o nekom cilju `G` definirano je na slijedeći način: Cilj `G` **logički slijedi** iz programa ako i samo ako u programu postoji stavak `C` za čiji *primjer* `I` vrijedi da ima istu glavu kao i `G` i svi ciljevi tijela od `C` su istiniti.

[7] arity

Primjer stavka[8] `C` je stavak `C'` u kojem je svaka varijabla zamijenjena s nekim izrazom.

Unifikacija je funkcija koja svakoj varijabli iz `C` pridružuje izraz iz `C'`.

Niz ciljeva (upit) logički slijedi iz programa ako postoji neki skup vrijednosti varijabli iz upita za koji su svi ciljevi upita izračunljivi. Pri tome "`,`" označava **konjunkciju**. **Disjunkcija** se može opisati s *više stavaka* ili s "`;`" između dva dijela. Pri tome "`,`" ima veći prioritet od "`;`".

1.2.4 Proceduralno znanje programa

Proceduralno znanje definira kako dolazi do odgovora na upit. Ono je postupak za izvođenje danog upita definiran programom. Imenujmo taj postupak **execute**. Slijedi *formalni opis izvođenja* programa.

Neka upit čini niz ciljeva `G1`, ... `Gm`.

1. [execute]: Ako je upit prazan, završi s uspjehom
2. [pretraživanje]: Odozgo nadolje traži stavak koji se može unificirati s `G1`; neka je to stavak `Sx`
3. Neka `Sx` ima oblik `Sx` ≡ `H:- B1,...,Bn.`, `S` neka je unifikacija `H` i `G1`, a `Sx'` primjer stavka `Sx` dobiven unifikacijom `S`. Neka `Sx'` ima oblik `Sx'` ≡ `H' :- B1',...,Bn'`.
4. Formira se novi upit `B1',...,Bn',G2,...,Gm`
5. Rekurzivno pozovi [execute] s novim upitom
6. Ako je izvršavanje uspjelo, završi uspjehom
7. Inače, vrati se na [pretraživanje], s tim da se pretražuju stavci od `Sx` nadalje.

Napomena: povratak na [pretraživanje] može inicirati korisnik nakon uspješnog završetka programa, ako želi znati ima li još koje rješenje.

1.3 Liste

Lista je rekurzivno definirana struktura za pohranjivanje podataka koja je prirođena Prologu kao što je polje prirođeno C-u, a često se koristi i za modeliranje skupova i operacija nad njima.

Lista općenito može biti *prazna* (oznaka: `[]`) ili *par* `[glava|rep]` gdje je `glava` *prvi element liste*, a svi ostali elementi su *sadržani u listi* `rep`. Radi jednostavnosti, lista se može zapisati tako da joj se pobroje elementi. Tako su slijedeća tri zapisa ekvivalentna:

```
[l1 | [l2 | [l3 | [l4 | []]]]]
[l1, l2, l3, l4]
[l2, l2 | [l3, l4]]
```

Formalno, u Prologu se lista predstavlja kako *dvočlana struktura* sa funktorom "`.`" (točka), pa vrijedi `[H|T]` ≡ `.(H, T)`. Prazna lista *nema zapisa pomoću funktora*, a `[]` ima sintaksu *atoma*.

1.3.1 Operacije nad listama

1.3.1.1 Pripadnost

Želimo definirati predikat (relaciju) oblika `element(X, L)` koja vrijedi ako je objekt `X` član liste `L`. Tako želimo da od slijedećih upita:

[8] instance

```
element(b, [a, b, c]).
element([b, c], [a, [b, c]]).
element(b, [a, [b, c]]).
```

prva dva uspiju, a treći ne. Rješenje:

```
element(X, [X|_]).          % uspije ako je X prvi u listi
element(X, [_|Ostali) :-
    element(X, Ostali).
```

Alternativno, mogli smo (duže) pisati i:

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

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

1.3.1.2 Povezivanje

Neka predikat ima oblik `povezi(L1, L2, L3)` i neka vrijedi ako se lista `L3` može dobiti povezivanjem listi `L1` i `L2`. Tako želimo da od slijedećih upita:

```
povezi([a, b], [c, d], [a, b, c, d]).
povezi([a, b], [c, d], [a, a, b, c, d]).
```

prvi uspije, a drugi ne.

Rješenje gradimo tako da najprije ispitamo jednostavan slučaj koji završava rekurziju (u 4. redu `+` treba shvatiti kao povezivanje listi).

```
% L1 = [] ==> L2 = L3 = L
povezi([], L, L).

% L1 = [H1|T1] ==> L3 = [H1|T1+L2]
povezi([H1|T1], L2, [H1|T3]) :-
    povezi(T1, L2, T3).
```

Primjeri upotrebe predikata `povezi/3`:

```
?- 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]
   ...
```

```
?- povezi(Prije, [svibanj|Poslije], [sijecanj, ..., prosinac]).
   Prije = [sijecanj, ..., travanj]
   Poslije = [lipanj, ..., prosinac]

?- povezi(_, [X, svibanj, Y], [sijecanj, ..., prosinac]).
   X = travanj
   Y = lipanj

?- povezi(L, [z, z, z|_], [a, b, z, z, c, z, z, z, d, e]).
   L = [a, b, z, z, c]
```

Rezultat drugog upita su sve moguće liste `L1` i `L2` koje zadovoljavaju relaciju (tj. svojim povezivanjem daju listu `[a, b, c]`). Pretposljednji upit pokazuje određivanje *prethodnika* i *sljedbenika*. Posljednji upit briše sve elemente liste koji se pojavljuju nakon trostrukog pojavljivanja elementa `z`.

1.3.1.3 Traženje podliste

Želimo definirati predikat `podlista(P, L)` koji vrijedi ako je lista `P` sadržana u listi `L`. Tako prvi upit vrijedi, a drugi ne:

```
podlista([c, d, e], [a, b, c, d, e, f]).
podlista([c, e], [a, b, c, d, e, f]).
```

Program se ostvaruje jednostavno s dva poziva predikata `povezi/3`:

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

1.3.1.4 Dodavanje, brisanje i umetanje

Novi element `X` se najlakše dodaje na početak liste `L`; novodobivena lista tada ima oblik `[X|L]`.

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

Za brisanje elementa `X` želimo predikat oblika `obirisi(X, L, Nova)`; taj predikat uspije ako se brisanjem elementa `X` iz liste `L` dobije lista `Nova`.

```
obrisi(X, [X|Ostali], Ostali).

obrisi(X, [Y|Ostali], [Y|OstaliBezX]) :-
    obrisi(X, Ostali, OstaliBezX).
```

Primjeri upotrebe:

```
?- 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]
```

U prvom primjeru, drugi i treći odgovor su jednaki: prvi je rezultat brisanja drugog `a`, a treći je rezultat brisanja trećeg `a`.

Drugi primjer je inspiracija za definiciju predikata `umetni(X, L, Nova)` koji umeće element `X` na proizvoljno mjesto liste `L`, a rezultat unificira s listom `Nova`:

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

1.3.1.5 Permutacije

U nekim programima potrebno je generirati sve permutacije zadane liste. Definirajmo zato predikat `permutacija/2` koji vrijedi ako je lista iz drugog argumenta (`arg2`) permutacija liste iz prvog argumenta (`arg1`); na primjer (drugi upit daje ukupno 6 rezultata, koliko ima permutacija liste od 3 elementa):

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

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

Kod implementacije razlikujemo dva slučaja:

- `arg1` je `[]`: permutacija prazne liste je prazna lista
- `arg1` ima oblik `[Prvi|Ostali]`: tada sve permutacije od `arg1` dobijemo umetanjem elementa `Prvi` na sve pozicije svih permutacija liste `Ostali`

```
% prvi slucaj
permutacija([], []).

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

Napomena: treba razlikovati slijedeća dva upita:

```
?- permutacija([r, g, b], P).
?- permutacija(P, [r, g, b]).
```

Prvi će dati sve permutacije. Međutim, drugi će se, nakon što izlista sve permutacije, *zaglaviti*. Kažemo da predikat `permutacija/2` ne podržava *obrazac instancijacije* (engl. instantiation pattern) u kojem je prvi argument neinstancirana varijabla (ona koja nije vezana za neku konkretnu vrijednost).

Zato naš predikat `permutacija/2` dokumentiramo na slijedeći način:

```
permutacija(+Lista, ?Permutirana)
```

Ovdje "`+`" znači da je argument **ulazni**: mora biti objekt ili vezana varijabla. "`?`" označava da argument može biti i **ulazni i izlazni** (tj. nevezana varijabla). U ovoj notaciji se **izlazni** argumenti (oni koji moraju biti nevezane varijable) označavaju sa "`-`", kao npr. `-izlazni`.

1.4 Operatori

Operatorski zapis izraza je uobičajen u matematici:

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

u ovom izrazu su `+` i `*` **operatori**, a 2, `a`, `b` i `c` su **argumenti**. U Prologu se ovaj izraz može prikazati i u standardnom obliku sa zagradama (formula u drugom redu: ovo je oblik strukture gdje su `+` `*` funktori).

Zbog bolje čitljivosti Prolog dozvoljava i operatorski zapis strukturiranih objekata i predikata. Prologu se može reći da operatorski zapis pojedinačnih funktora interpretira kao da su oni zadani standardnim oblikom. Neki operatori su unaprijed definirani pa su prvi i drugi izraz za Prolog ekvivalentni (ako programer ne odredi drukčije).

Operatorski zapis funktora se omogućuje pozivom ugrađenog predikata `op/3`:

```
:- op(600, xfx, ima).
```

Ovdje smo uveli i *stavak bez glave*: on se izvršava prilikom učitavanja programa (`consult`). 600 je *prioritet operatora*, xfx je *tip* operatora, a "ima" je funktor. Nakon definicije operatora možemo postavljati upite poput:

```
ima(marko, zir).

?- X ima Y.
   X = marko
   Y = zir

?- ima(mirko, X) = Y ima loptu.
   X = loptu
   Y = mirko
```

Omogućavanje operatorskog zapisa funktora utječe samo na dozvoljenu sintaksu pisanja programa i nije povetano ni sa kakvom operacijom na podacima. Zato je omogućavanje operatorskog zapisa slično pretprocesorskoj direktivi `#define` u C-u.[9]

Prioritet operatora je broj 1...2000: veći broj označava veći prioritet u smislu da *operator s najvećim prioritetom postaje glavni funktor izraza*. Alternativno, može se reći da operator s *manjim prioritetom veže jače*. Tako se npr. `a+b*c` interpretira kao `+(a, *(b, c))` jer `+` ima veći prioritet od `*`.[10]

[9] engl. "syntactic sugar"

[10] Dakle, ovo je obrnuto nego matematički prioritet operatora: veći broj znači manji prioritet u matematičkom smislu.

Tip operatora može biti xfx, xfy, yfx, fx, fy, xf, yf. Slovo `f` u oznaci tipa označava *položaj operatora* u odnosu na argumente. Slovo `x` se interpretira tako da prioritet odgovarajućeg argumenta mora biti *manji od prioriteta operatora*, inače će biti prijavljena sintaksna greška. Slovo `y` znači da je prioritet odgovarajućeg operatora proizvoljan. *Prioritet argumenta* je 0, osim ako je argument operatorski izraz (tada je prioritet jednak prioritetu glavnog operatora). Izraz u zagradama također ima prioritet 0. Posljednji primjer ovog poglavlja neformalno i jasno objašnjava razliku tipova operatora.

Primjer: xf i yf su *postfiksni operatori*, a fx i fy su *prefiksni operatori*.

```
:- op(600, xf, ^^).      :- op(600, fx, ^^).
[H|_]^^ :- H = b.        ^^([H|_]) :- H = b.

?- [b, s]^^              ?- ^^[b, s].
   Yes.                  Yes.
```

Primjer: Izraz `a-b-c` obično interpretiramo kao `(a-b)-c`, *a ne kao* `a-(b-c)`. To Prologu možemo reći tako da tip operatora `-` definiramo kao yfx. Time zabranjujemo interpretacije u kojima je desni argument operatorski izraz istog prioriteta. Tako je jedino ispravno tumačenje: `-(-(a, b), c)` (prvi argument je `y`, a drugi argument je `x`).

Upotrebom operatora čitljivost programa se može povećati: pretpostavimo da u Prologu želimo zapiasti deMorganovo pravilo: $\overline{AB} \Longleftrightarrow \overline{A} + \overline{B}$. Standardno, to možemo učiniti na slijedeći način:

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

Upotrebom operatorskog zapisa izraz izgleda

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

uz slijedeće definicije operatora:

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

Definicija ekvivalencije ne dozvoljava izraz poput: `A <==> B <==> C`. Dakle, xfx je definicija *neasocijativnog operatora*: u izrazima poput ovog, *obavezna je upotreba zagrada*.

Definicija za `v` i `&` ima tip xfy. To znači da će izraz poput `A v B v C` biti interpretiran kao `A v (B v C)`. Dakle, tip xfy je definicija *desno-asocijativnog operatora*. Analogno, tip yfx označava *lijevo-asocijativan operator*.

1.5 Aritmetičke operacije

Prolog *nije* namijenjen za opisivanje matematičkih postupaka. Zato je programska podrška za aritmetičke operacije u Prologu skromna. Ključni predikat za aritmetičke operacije je `is/2`. Predikat `is/2` vraća istinu ako mu se prvi argument može unificirati s aritmetičkom vrijednošću drugog. Aritmetička vrijednost izraza se pronalazi ugrađenim procedurama koje su pridružene aritmetičkim funktorima (`+-*/` i ostali). Ugrađeni operatori su definirani kao *lijevo asocijativni* (yfx). Ugrađeni predikat `current_arithmetic_function/1` vraća sve funktore za koje su aritmetičke funkcije definirane. Primjer:

```
?- X = 1+2.
   X = 1+2

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

Skup funktora koje predikat `is/2` može izračunati se može proširiti ugrađenim predikatom `arithmetic\_function/1`, npr:

```
:- arithmetic_function(kvadrat/1).
kvadrat(X, Rez) :- Rez is X*X.

?- X is kvadrat(12).
   X = 144
```

1.5.1 Predikati usporedbe

Pored predikata `is/2` aritmetičku vrijednost izraza određuju i predikati usporedbe: `>`, `<`, `>=`, `=<`, `=\=` (različito) i `=:=` (jednako).[11] To su dvomjesni predikati čije je vrijednost istinita ako su aritmetičke vrijednosti operanada u odgovarajućoj relaciji, npr: Programom je definirana realacija `godiste(Covjek, Godina)`, a željeli bismo pronaći sve ljude koji su rođeni između 1950. i 1960. godine.

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

Računanje najvećeg zajedničkog djelitelja (modificirani Euklidov algoritam; ne računaju se ostaci direktno već uzastopno oduzimanje):

```
nzd(X, X, X).
nzd(X, Y, D) :-
    X < Y,
    Y1 is Y-X,
    nzd(X, Y1, D).
nzd(X, Y, D) :-
    X > Y,
    nzd(Y, X, D).
```

Računanje duljine liste:

```
duljina([], 0).
duljina([_|Rep], D) :-
    duljina(Rep, D1),
    D is D1+1.
```

[11] Operator `=` je operator unifikacije, `==` testira *jednakost atoma*, `is` je aritmetički jednako (pri računanju vrijednosti), a `=:=` je aritmetička jednakost.

2 Neke tehnike programiranja

2.1 Prenošenje akumulatora

U konvencionalnim jezicima (C, Pascal) procedure mogu imati parametre čije promjene se vide i u glavnom programu. U Prologu se takav poziv postiže tehnikom koja se zove *prenošenje pomoću akumulatora*.

U toj tehnici predikat prima ulaznu vrijednost varijable u jednom argumentu (`Pobrojac`), a novu vrijednost veže s drugim argumentom (`Ukupno`), na primjeru za duljinu liste:

```
duljina(Lista, Duljina) :-
    d(Lista, 0, Duljina).

d([], Ukupno, Ukupno).
d([_|T], Pobrojac, Ukupno) :-
    Pobrojac1 is Pobrojac+1,
    d(T, Pobrojac1, Ukupno).
```

2.2 Optimizacija posljednjeg poziva

U toj tehnici optimizacije posljednji cilj u tijelu predikata se ne izvodi kao poziv procedure (`call`) već kao obično grananje (`jump`). Time se smanjuje potrebna veličina stoga za izvođenje programa i povećava brzina izvođenja programa.

Drugi oblik predikata `duljina/2` ima rekurzivni poziv kao posljednji cilj drugog stavka, pa se optimizacijom posljednjeg poziva procedura može provesti iteracijom umjesto rekurzijom.

2.3 Upravljanje postupkom vraćanja

Kad Prolog prilikom izvođenja programa ne uspije razriješiti upit, automatski se vraća na mjesto posljednjeg izbora i upit pokušava riješiti na neki drugi način. Taj ugrađeni mehanizam vraćanja je jedno od osnovnih svojstava Prologa. No, povremeno je zgodno imati mogućnost da izbjegnemo vraćanje preko nekih grana stabla izvođenja programa u Prologu.

Na primjer, definirajmo relaciju koja je zadana funkcijom:

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

Ovu relaciju ostvaruje slijedeći program:

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

Stablo izvođenja za upit `f(1,Y), Y > 2` prikazano je na slici `??`. Prvo i treće pravilo se međusobno isključuju pa bismo htjeli da se, nakon što je jednom razriješen prvi stavak, stavci 2 i 3 uopće ne pokušaju razriješiti jer je to gubljenje vremena. Za takve namjene u Prologu postoji naredba `!` (**rez**). Nova definicija programa:

```
(S1)    f(X, 0) :- X < 3, !.
(S2)    f(X, 2) :- X >= 3, X < 6, !.
(S3)    f(X, 4) :- X >= 6.
```

Primijetimo da tokom izvođenja S2 i S3 provjeravamo jedan nepotreban cilj. Ako je $X < 3$, rez u pravilu će onemogućiti upravljanje u slijedeći stavak. Zato pišemo najefikasniju verziju programa:

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

2.3.1 Formalna definicija reza

Neka polazni cilj bude cilj koji je u toku izvođenja prilagođen glavi stavka u kojem se nalazi rez. Ako cilj `!` dođe na red za izvođenje, on će uspjeti, a pored toga će se *zabraniti vraćanje u sve do tada pozvane ciljeve stavka* (tj. one ciljeve koji se u definiciji stavka nalaze lijevo od reza).

Zadan je slijedeći program:

```
(P1)    c :- p, q, r, !, s, t, u.
(P2)    c :- v.
(P3)    a :- b, c, d.
```

Prilikom izvođenja stavka P1 vraćanje na početak je moguće samo iz ciljeva `p`, `q` i `r`. Jednom kad se niz predikata `p`, `q`, `r` razriješi, neće se razmatrati nijedno razrješavanje na drugi način. Korištenje pravila P2 će u tom slučaju također biti onemogućeno, dok će vraćanje unutar ciljeva `s`, `t`, `u` biti dozvoljeno (npr. ako `t` ne uspije). Rez utiče samo na izvođenje predikata *u čijoj se definiciji nalazi.* U zadanom primjeru, rez u pravilu P1 ne utječe na pravilo P3, pa se u tom pravilu vraćanje normalno odvija.

Rezom se često može poboljšati efikasnost programa. Prologu se tako može izričito reći da ne pokušava razriješiti alternative za koje je poznato da ne mogu uspjeti, ili još gore, vode na beskonačnu petlju.

Upotrebom reza mogu se izreći pravila koja se međusobno isključuju, poput klasičnog `if P then Q else R`, na primjer:

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

U tom smislu rez povećava izražajnu moć Prologa. Međutim, unošenjem rezova mijenjaju se odnosi deklarativnog i proceduralnog znanja programa. U programu bez rezova deklarativno znanje ovisi isključivo o *sadržaju* stavaka, a ne i o njihovom redoslijedu. U programu sa rezovima, *redoslijed stavaka bitno utječe na deklarativno znanje.* Na, primjer:

```
p :- a, b.
p :- c.

p :- c.
p :- a,b.
```

Ova dva programa su ekvivalentna i izražavaju logičku formulu $(a \wedge b) \vee c \Rightarrow p$.

Međutim, uvođenjem reza, dva programa se nakon promjene redoslijeda bitno razlikuju:

```
p :- a, !, b.
p :- c.

p :- c.
p :- a, !, b.
```

Prvi program ekvivalentan je formuli $(a \wedge b) \vee (a \wedge c) \Rightarrow p$. Međutim, drugi program ekvivalentan je formuli $c \vee (a \wedge b) \Rightarrow p$. Zato rez valja koristiti samo kad je nužno potreban.

2.3.2 Primjeri

2.3.2.1 Negacija

Do sada je predikat `not` (`\+`) korišten u standardnom i operatorskom zapisu, ali bez razmatranja njegove izvedbe. Predikat `not` se izvodi preko reza te specijalnih predikata `fail` i `true`. Vrijednost predikata `true` je uvijek istina, dok predikat `fail` uvijek vraća neuspjeh.

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

2.3.2.2 Razvrstavanje u klase

Zadatak je razvrstati igrače nekog teniskog kluba u kategorije na temelju susreta održanih jedne nedjelje po slijedećem principu:

- kategorija borac – igrači koji su i dobili i izgubili bar po jedan susret
- kategorija pobjednik – igrači koji su pobijedili u svim susretima
- kategorija sportaš – igrači koji su izgubili u svim susretima

Rezultati neka su opisani relacijom `pobijedio(X, Y)` koja označava da je X pobijedio Y.

```
pobijedio(ivo, tin). pobijedio(ana, ivo). pobijedio(eva, tin).

kategorija(X, borac) :-
    pobijedio(X, _),
    pobijedio(_, X), !.
kategorija(X, pobjednik) :-
```

```
    pobijedio(X, _), !.
kategorija(X, sportas) :-
    pobijedio(_, X).
```

Rez u prvom stavku je potreban jer bi tada svaki borac bio proglašen i pobjednikom zbog drugog stavka. Rez u drugom stavku je samo zbog efikasnosti kako se ne bi ispitivao treći stavak.

2.4 Nepotpuno instanciranje objekata

Lijepo svojstvo Prologa je i da podržava objekte koji su samo djelomično instancirani. Takvi objekti u svojoj strukturi sadrže varijable koje naknadno tokom izvršavanja programa mogu biti instancirane, ili, što je češći slučaj, unificirane s novim, često instanciranim objektom. Da bi se to omogućilo, potrebno je uz ovakav djelomično instancirani objekt pamtiti i sve neinstancirane varijable koje se u njegovoj strukturi javljaju.

2.4.1 Pojednostavljivanje izraza

Na primjer, razmotrimo strukturu za pojednostavljivanje aritmetičkih izraza. U ovoj jednostavnoj verziji preocedura radi samo za izraze u kojima je jedina dozvoljena operacija zbrajanje, npr: `3+a+b+5` daje `8+a+b`. Aritmetički izraz se u toj proceduri predstavlja stablom na čijem je jednom kraju neinstancirana varijabla zvana `Rupa`. Varijabla `0` ulazna suma, a `N` je izračunata suma brojeva u danom izrazu.

```
pojednostavi_sumu(Suma, Jednostavna) :-
    napravi_stablo(Suma, Stablo:Rupa, 0, N),
    N =\= 0, !,                 % if N != 0 then
    Rupa = N,
    Jednostavna = Stablo
    ;                           % else
    var(Stablo), !,             % if var(Stablo) then
    Jednostavna = 0
    ;                           % else
    Stablo = Jednostavna + Rupa.

napravi_stablo(A+B, Stablo:Rupa, N0, N) :-
    !,
    napravi_stablo(B, Stablo:Rupa1, N0, N1),
    napravi_stablo(A, Rupa1:Rupa, N1, N).
napravi_stablo(C, Rupa:Rupa, N0, N) :-
    number(C), !,
    N is N0 + C.
napravi_stablo(X, Rupa+X:Rupa, N, N).
```

2.4.2 Povezivanje liste

Dosada smo za povezivanje nizova koristili proceduru `povezi/3` (vidi **odjeljak 1.3.1.2**). Razmotrimo izvođenje te procedure:

```
?-  povezi([a, b, c], [d, e], L).
    povezi([a, b, c], [d, e], L).
     povezi([b, c], [d, e], L1)         % L  = [a|L1]
      povezi([c], [d, e], L2)           % L1 = [b|L2]
       povezi([], [d, e], L3)           % L2 = [c|L3]
        L3 = [d, e]                     % L3 = [d, e]
```

Vidimo da je složenost $O(n)$, dakle *linearna*.

Procedura se može bitno poboljšati ako primijenimo alternativan prikaz listi. U tom prikazu lista se prikazuje parom `Lista` i `Rupa` gdje je `Lista` lista koja završava sa rupom koja je varijabla `Rupa`. Na, primjer, alternativni prikaz liste `[1, 2, 3]` bio bi

```
[1, 2, 3 | Rupa] : Rupa
```

Sada možemo definirati povezivanje na mnogo efikasniji način:

```
povezi_a(Al:Ar, Bl:Br, Cl:Cr) :-
    Ar = Bl, Cl = Al, Cr = Br.
```

Očito je da ova procedura ima *konstantnu* složenost $O(1)$.

2.5 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 moraju izvesti kao argumenti rekurzivnog predikata jer je leksički doseg varijable u Prologu jedan stavak.

Kao primjer razmatramo proceduru koja vektor iz prvog argumenta množi skalarom iz drugog, a rezultat vraća u trećem argumentu. Rješenje u C++:

```
#include <vector>

void mnozi(const vector<double>& x, double m, vector<double>& r)
{
    int n = x.size();

    r.resize(n);
    while(--n >= 0) {
        r[n] = x[n]*m;
    }
}
```

Rješenje u Prologu:

```
mnozi([], _, []).
mnozi([X|X1], M, [R|R1]) :-
    R is X*M,
    mnozi(X1, M, R1).
```

Ovakav zapis algoritma u Prologu se čini neefikasnim jer se parametar `M` beskorisno prenosi u svaki novi rekurzivni poziv. Međutim, postoji tehnika optimizacije koja može utvrditi da je drugi

argument predikata `mnozi` jednak u svim rekurzivnim pozivima, pa se u izvršnom kodu `M` može izvesti kao statička lokalna varijabla na razini predikata.

Konačno, ako prevodilac primijeni optimizaciju posljednjeg poziva, teoretski nema razloga da izvršni kod procedure u Prologu bude sporiji od izvršnog koda procedure u nekom konvencionalnom jeziku.

Općenito, vrijedi slijedeće pravilo: odsječak konvencionalnog programa koji referencira N različitih varijabli koje nisu lokalne za taj odsječak može se prevesti u predikat u Prologu sa N argumenata koji se onda nazivaju **argumenti konteksta.** To pravilo je zgodno znati prilikom čitanja programa u Prologu: ako u nekoj proceduri postoje varijable koje se nepromijenjene javljaju kao argumenti u pozivima raznih predikata, njih valja smatrati varijablama koje nisu lokalne za tu proceduru.

Primjer: iz zadane liste brojeva treba odabrati one koji su veći od zadane granice:

```
veliki([], _, []).
veliki([X|X1], Granica, Veliki) :-
    X <= Granica, !,
    veliki(X1, Granica, Veliki).
veliki([X|X1], Granica, [X|JosVeliki]) :-
    veliki(X1, Granica, JosVeliki).
```

U standardnim jezicima predikat `veliki` bio bi izražen petljom, dok bi varijabla `Granica` bila deklarirana prije te petlje.

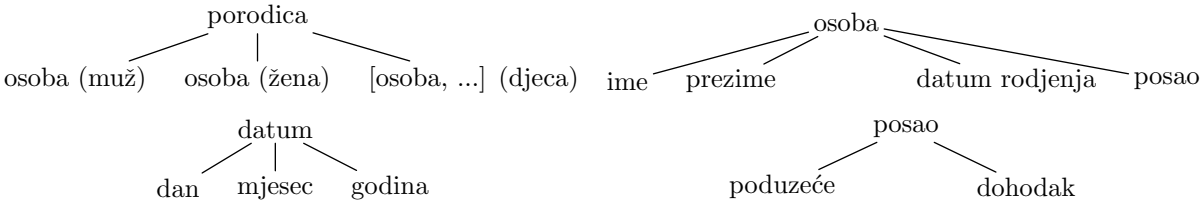
Prilikom korištenja argumenata konteksta ponekad se javlja problem da ih je potrebno puno. U takvom slučaju standardno se koristi idiom u kojem su svi argumenti konteksta objedinjeni zajedničkom strukturom. Pojedinačnim argumentima se tada pristupa pomoćnim predikatom koji iz strukture vadi željenu vrijednost.

3 Primjena Prologa

U prvom dijelu vježbi opisane su temeljne tehnike programiranja u Prologu: unifikacija, vraćanje (backtracking), strukturiranje podataka i aritmetika. Primjenom tih tehnika u drugom dijelu vježbi će biti pokazana primjena Prologa na rješavanju (nešto) složenijih problema.

3.1 Pretraživanje baze podataka

Programom u Prologu želimo pristupati skupu objekata sa strukturom prikazanoj na **slici 3.1**.



Slika 3.1 Elementi strukture porodica

Prirodni način predstavljanja baze podataka u Prologu je skup činjenica. Svaku porodicu iz željenog skupa možemo opisati jednim stavkom, na primjer:

```
porodica(
    osoba(jerko, duplencic, datum (1, svibanj, 1950),
          posao(cementara, 4000)) -
    osoba(vesna, krstic, datum(23, rujan, 1947),
          posao(kolodvor, 4000)) -
    [ osoba(suncana, duplencic, datum(31, listopad, 1971),
            nezaposlen),
      osoba(darko, duplencic, datum(6, 11, 1975), nezaposlen)]
).
```

Prolog je posebno prikladan za asocijativni dohvat podataka, tj. kada želimo dohvatiti objekt iz skupa ne na temelju njegovog fizičkog smještaja u memoriji (pointer ili indeks) nego na temelju njegovog djelomično specificiranog sadržaja.

Na primjer, želimo dohvatiti sve obitelji u kojima se majka i otac prezivaju Zoranić. Korisno je definirati pomoćni predikat `prezime/2`:

```
prezime(osoba(_, Prezime, _, _), Prezime).
?- porodica(P), P=(O-M-_),
   prezime(O, zoranic), prezime(M, zoranic).
```

Imena i prezimena svih žena koje imaju troje ili više djece:

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

U većim programima stabilnost sustava se postiže *skrivanjem informacija.* Jedan od vidova skrivanja informacije je i **apstrakcija podataka**. Podatkovnom apstrakcijom osiguravamo da korisnik može (još bolje — *mora*) pristupati podacima bez poznavanja detalja fizičke organizacije podataka u memoriji računala. Time se postiže mogućnost nezavisnog razvijanja struktura podataka i procedura za njihovu obradu.

Na primjeru već opisanih `ime/2` i `prezime/2`, a u cilju apstrakcije podataka bilo bi korisno dodati i slijedeće predikate:

```
otac(P, O) :- porodica(P), P=O-_-_.
otac(O) :- otac(_, O).
majka(P, M) :- porodica(P), P=_-M-_.
majka(M) :- majka(_, M).
djeca(P, Djeca) :- porodica(P), P=_-_-Djeca.
djeca(Djeca) :- djeca(_, Djeca).
dijete(P, D) :- djeca(P, Djeca), clan(D, Djeca).
dijete(D) :- dijete(_, D).
postoji(X) :- otac(X) ; majka(X) ; dijete(X).
roden(osoba(_, _, D, _), D).
placa(osoba(_, _, _, posao(_, P)), P).
placa(osoba(_, _, _, nezaposlen), 0).
dohodak([], 0).
dohodak([Prvi|Ostali], Ukupno) :-
    placa(Prvi, P),
    dohodak(Ostali, Po),
    Ukupno is P+Po.
```

Predikat `dohodak/2` računa dohodak liste osoba.

Koristeći pomoćne procedure možemo zadati slijedeće upite: djeca mlađa od 14 godina (prvi upit), ljudi stariji od 30 godina koji zarađuju manje od 3000 (drugi upit) ili porodični dohodak za porodicu Jerka Duplenčića (treći upit):

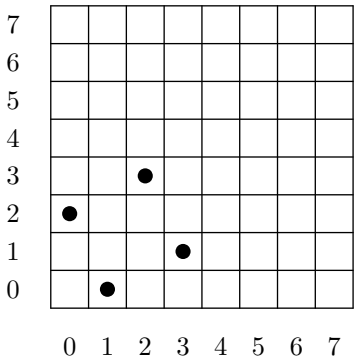
```
?- dijete(X), roden(X, D), D=datum(_, _, G), G >= 1985.

?- postoji(X), roden(X, datum(_, _, G)), G =< 1969,
   placa(X, P), P < 3000.

?- porodica(P), otac(P, O), prezime(O, duplencic), ime(O, jerko),
   majka(P, M), djeca(P, D),
   dohodak([O, M | D], Dohodak).
```

3.2 Problem osam kraljica

Općenito, n kraljica u šahovskom polju dimenzija $n \times n$ treba rasporediti tako da se nijedne dvije kraljice međusobno ne napadaju. Skup svih razmještaja je ogroman. Ima $\binom{n^2}{n}$ kombinacija koje nije praktično sve ispitati. Različiti pristupi rješavanju problema provjeravaju različite podskupove svih rješenja i zato imaju različite složenosti.

3.2.1 Prvi pristup

Za svaki dozvoljeni položaj vrijedi da se u svakom stupcu ploče nalazi *točno jedna* kraljica. Rješenje možemo prikazati listom koordinata redaka kraljica uz konvenciju da koordinata stupca odgovara položaju kraljice u listi. Za primjer sa **slike 3.2** lista je `[2, 0, 3, 1]`.



Slika 3.2 Početak raspoređivanja kraljica

Skup svih rješenja je podskup skupa svih permutacija liste s elementima od 0 do $n-1$; ima ih ukupno $n!$.

```
rjesenje(R) :-
    permutacija([0, 1, 2, 3, 4, 5, 6, 7], R),
    rjesenje1(R).

rjesenje1([]).
rjesenje1([Prva|Ostale]) :-
    rjesenje1(Ostale),
    ne_napada(Prva, Ostale, 1).
```

Prostor svih razmještaja je smanjen na $n!$ elemenata koje ispitujemo sekvencijalno predikatom `rjesenje1`. Predikat `ne_napada(Kraljica, Lista, Udaljenost)` vrijedi ako kraljica s koordinatama (Kraljica, Stupac) ne napada neku od kraljica čiji su retci predstavljeni listom, a stupci su im: Stupac+Udaljenost, Stupac+Udaljenost+1,... Ovime smo osigurali ne napadanje po stupcima i retcima; ostaje još dijagonalno:

```
ne_napada(_, [], _).
ne_napada(Kraljica, [Prva|Ostale], Udaljenost) :-
    Kraljica-Prva =\= Udaljenost,   % napadanje po silaznoj dijagonali
    Prva-Kraljica =\= Udaljenost,   % uzlazna dijagonala
```

```
    Udaljenost1 is Udaljenost+1,
    ne_napada(Kraljica, Ostale, Udaljenost+1).
```

3.2.2 Drugi pristup

Položaj ćemo predstaviti listom *n parova koordinata.* Zadatak ćemo riješiti na klasičan način: izdvojit ćemo prvu kraljicu iz niza, provjeriti da se kraljice iz repa niza međusobno ne napadaju i na kraju provjeriti da prva kraljica ne napada niti jednu kraljicu iz repa niza. Ponovo ćemo se okoristiti činjenicom da se u rješenju problema nikoje dvije kraljice ne nalaze u istom retku. U rješenju `/` koristimo za odvajanje koordinata.

```
sredi(R) :- R = [1/s1, 2/s2, 3/s3, 4/s4, 5/s5, 6/s6, 7/s7, 8/s8].

rjesenje([Red/Stupac|Ostale]) :-
    rjesenje(Ostale),
    element(Stupac, [1, 2, 3, 4, 5, 6, 7, 8]),
    ne_napada(Red/Stupac, Ostale).

ne_napada(R/S, [R1/S1|Ostale]) :-
    S =\= S1,                          % stupac
    S1-S =\= R1-R,                     % silazna dijagonala
    S1-S =\= R-R1,                     % uzlazna dijagonala
    ne_napada(R/S, Ostale).
```

Odgovor ćemo dobiti upitom

```
?- sredi(R), rjesenje(R).
```

Treba naći sva rješenja.

Drugi pristup je za male n puno bolji jer rješenje gradi tako da novu kraljicu stavlja samo na polja ploče koja ne napadaju već postavljene kraljice. Pri tome, na žalost, nema garancije da nađeno parcijalno rješenje vodi do cilja. Zato se složenost algoritma za traženje rješenja ne može ograničiti polinomom, iako se provjera rješenja može provesti u polinomnom vremenu.

3.2.3 Treći pristup

Ubrzana varijanta drugog pristupa: osnovna ideja je ubrzavanje zauzimanja polja pomoću tablice. Prilikom rješavanja koristit će se liste nepotrošenih redaka, stupaca, 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 spomenutim listama. Ako to nije slučaj, nova kraljica se nalazi na položaju međusobnog napadanja s nekom od već postavljenih kraljica pa razmještaj nije valjan.

Pri rješavanju se može uzeti u obzir i činjenica da je svaka rotacija ispravnog razmještaja za 90^o također ispravan razmještaj.

3.2.4 Formalni opis problema

Problem se može formalno opisati na slijedeći način: treba naći osam legalnih četvorki (i, j, u, v) tako da se ni za jedan član ne upotrijebi dva puta ista vrijednost. Izvedba će se temeljiti na proceduri `rijesi`:

```
rijesi(Razmjestaj, Retci, Uzlazni, Silazni).
```

`Retci` je lista slobodnih redaka, `Uzlazni` i `Silazni` su liste slobodnih dijagonala.

```
rijesi([Redak/Stupac | JosKraljica], R, Du, Dv) :-
    obrisi(Redak, R, R1), U is Redak+Stupac,
    obrisi(U, Du, Dv, Du1), V is Redak+Stupac,
    obrisi(V, Du, Dv, Dv1),
    rijesi(JosKraljica, R1, Du1, Dv1).

rjesenje3(Razmjestaj) :-
    Razmjestaj = [_/1, _/2, _/3, _/4, _/5, _/6, _/7, _/8],
    Retci = [1, 2, 3, 4, 5, 6, 7, 8],
    Du = [-7, -6, ... , 6, 7],
    Dv = [2, 3, ... , 16],
    rijesi(Razmjestaj, Retci, Du, Dv).
```

3.3 Problem slagalice

Cilj ove igre (vidi **sliku 3.3**) je pomicanjem pločica dovesti slagalicu u završno stanje. Postoji zgodan *teorem o rješivosti* ovog problema; ali prvo se moramo upoznati s jednim svojstvom permutacija.

8	3	4
5		6
2	1	7

1	2	3
4	5	6
7	8	

Slika 3.3 Početna pozicija i složena slagalica

Neka je $a_1 a_2 \ldots a_n$ permutacija skupa $1, 2, \ldots, n$. Ako je $i < j$ i $a_i > a_j$, tada se par (a_i, a_j) naziva **inverzija** permutacije; na primjer, permutacija (3142) ima tri inverzije: $(3, 1)$, $(3, 2)$ i $(4, 2)$. Jedina permutacija bez inverzija je "sortirana" permutacija $(12 \ldots n)$. Ukoliko je broj inverzija u permutaciji *paran*, permutacija se naziva parna. Analogno definiramo *neparnu* permutaciju.

Sada se može iskazati **teorem o rješivosti** igre: Igra je rješiva (općenitije, jedna pozicija se može prevesti u drugu poziciju) ukoliko su obje pozicije, promatrane kao permutacije, *iste parnosti.* Permutaciju pozicije dobijemo tako da jednostavno ispišemo elemente po retcima, izostavljajući prazno polje. Tako početnoj poziciji sa **slike 3.3** odgovara permutacija (83456217), a složenoj igri odgovara permutacija (12345678). Treba primijetiti da su obje permutacije iste parnosti (parne) te je stoga moguće složiti početnu poziciju.

3.3.1 Metode pretraživanja

Za dani graf `G`, početni čvor `s` i predikat `cilj` (koji definira ciljne čvorove), problem pretraživanja se sastoji u određivanju puta od `s` do `t` u `G` tako da vrijedi `cilj(t).` Većina algoritama prilikom pretraživanja održava dvije liste: lista `OTVORENI` sadrži čvorove do kojih je postupak došao, dok lista `ZATVORENI` sadrži već ispitane čvorove. Algoritmi imaju slijedeću temeljnu strukturu:

```
trazi(graf G, cvor Pocetni, predikat Cilj)
    OTVORENI := {Pocetni}
    ZATVORENI := {}
    pronasao := 0
    dok (OTVORENI nije prazna) i (pronasao == 0) radi
        prebaci cvor N iz OTVORENI u ZATVORENI
        ako cilj(N) onda
            pronasao := 1
        inace
            nadji susjede od N u G koji nisu u listama OTVORENI
            i ZATVORENI i dodaj ih u listu OTVORENI
    ako pronasao == 1 onda
        vrati N
    inace
        neuspjeh
```

Često se još dodatno traži da nakon uspješnog pretraživanja algoritam vrati i put kojim je pronađen ciljni čvor – ovo se ostvaruje jednostavnom modifikacijom osnovnog algoritma. Postoje dvije osnovne strategije odabira slijedećeg čvora iz liste `OTVORENI` što vodi na *pretraživanje u širinu* i *pretraživanje u dubinu.* U nastavku su dana oba algoritma implementirana u Prologu. Karakteristika je pretrage u širinu da uvijek nalazi *najkraći* put do rješenja.

```
%%
% Pretraga u dubinu (dfs, Depth First Search)
% dfs(+Arc, +Start, +Goal, -Solution)
% Arc je predikat koji vraca susjede nekog cvora.
% Start je pocetni cvor, Goal je predikat koji ispituje da li je cvor ciljni.
% Sol je put do rjesenja.
%%
dfs(Arc, Start, Goal, Sol) :-
    dfs1(Arc, [[Start]], [], Goal, Sol).

%%
% Ovo je stvarna procedura. Drugi argument je
% stack OTVORENI, a treci je lista ZATVORENI.
%%
dfs1(_, [[Node|Path]|_], _, Goal, [Node|Path]) :-
    call(Goal, Node).

dfs1(Arc, [[Node|Path]|MoreOPEN], CLOSED, Goal, Sol) :-
    % nadji sve susjede od prvog OTVORENOG cvora i
    % dodaj trenutni put na svakog od njih.
    findall([Next,Node|Path],
        (call(Arc, Node, Next),
        \+ member([Next|_], [[Node|Path]|MoreOPEN]),
        \+ member(Next, CLOSED)),
        NewPaths),
    % stavi nove puteve na VRH STACKA
    append(NewPaths, MoreOPEN, NewOPEN),
```

```
    closing(Node),
    dfs1(Arc, NewOPEN, [Node|CLOSED], Goal, Sol).

%%
% Pretraga u sirinu (bfs, Breadth First Search)
% isto kao u dubinu, jedino sto se umjesto stacka koristi red (queue))
%%
bfs(Arc, Start, Goal, Sol) :-
    bfs1(Arc, [[Start]], [], Goal, Sol).

bfs1(_, [[Node|Path]|_], _, Goal, [Node|Path]) :-
    call(Goal, Node).

bfs1(Arc, [[Node|Path]|MoreOPEN], CLOSED, Goal, Sol) :-
    findall([Next, Node, Path],
        (call(Arc, Node, Next),
        \+ member([Next|_], [[Node|Path]|MoreOPEN]),
        \+ member(Next, CLOSED)),
        NewPaths),
    % stavi nove puteve na KRAJ REDA
    append(MoreOPEN, NewPaths, NewOPEN),
    closing(Node),
    bfs1(Arc, NewOPEN, [Node|CLOSED], Goal, Sol).
```

Predikat `closing` se poziva za svaki posjećeni čvor grafa i može biti iskorišten za ispis redoslijeda posjećenih čvorova.

Osim ova dva osnovna "slijepa" algoritma, postoje i **heuristički** algoritmi. Oni se temelje na računanju **heurističke funkcije** za svakog susjeda. Susjedi se stavljaju u listu `OTVORENI` tako da onaj koji ima *najmanju vrijednost* heurističke funkcije *bude prvi* u listi `OTVORENI`. Tako će se najprije posjećivati oni čvorovi sa najmanjom vrijednošću heurističke funkcije. Heuristička funkcija se može promatrati kao funkcija koja vraća približnu udaljenost nekog čvora do cilja. Jedan takav heuristički algoritam je best-first-search.

3.3.2 Rješenje problema

Problem slagalice se može svesti na problem pretraživanja: *čvor* u grafu je neka pozicija, a dva čvora su *povezana* ako se iz jedne pozicije može prijeći u drugu nekim potezom. Iz razmatranja u uvodu proizlazi da je prostor stanja (broj čvorova grafa) ogroman: iznosi $N!/2$ gdje je N *ukupan* broj polja na igri (npr. za 3×3 slagalicu, $N = 9$).

Rješenje u najmanjem broju poteza bi dala pretraga po širini. Međutim zbog veličine prostora stanja, to troši previše vremena i prostora: slijepa pretraga po širini već za 3×3 slagalicu ne može naći rješenje.[12] Stoga je potrebno upotrijebiti neki heuristički algoritam (koji gotovo sigurno neće dati najkraće rješenje, ali će bar doći do njega).

Pri upotrebi heurističkog algoritma, trebamo izabrati heurističku funkciju. Jedna jednostavna heuristička funkcija je ona koja vraća broj pločica koje nisu na svojem mjestu. Malo bolju heurističku

[12] Barem u Prolog implementaciji. Napisao sam program u C-u koji pretraživanjem po širini nalazi rješenje u najmanjem broju poteza u 1.5 sekundi. Zbog veličine prostora stanja više ni u C-u nije moguće riješiti 4×4 slagalicu slijepim pretraživanjem.

funkciju (ona koja daje rješenje u manjem broju poteza) možemo napraviti tako da broju elemenata koji nisu na svojim mjestima dodamo i broj inverzija u permutaciji pozicije. Upotrebom bilo koje od ove dvije heurističke funkcije moguće je riješiti 3×3 slagalicu, međutim nijedna nije dovoljno dobra za rješavanje 4×4 slagalice.

3.3.3 Drugi pristup

Zgodan put rješenju je slaganje po slojevima, koji je prikladan samo za kvadratne slagalice $n \times n$. Prvo slažemo tako da su svi elementi prvog retka i prvog stupca na svojim mjestima. Kad smo to napravili, sveli smo problem na rješavanje $(n-1) \times (n-1)$ slagalice. Pri rješavanju manje slagalice, ne moramo dirati elemente koji su već na svojim mjestima.

8	21	13	17	4
14	23		19	15
3	7	11	2	18
12	10	16	1	20
6	5	24	9	22

1	2	3	4	5
6				
11				
16				
21				

1	2	3	4	5
6	7	8	9	10
11	12			
16	17			
21	22			

Slika 3.4 Slaganje po slojevima

Slika 3.4 daje početno stanje 5×5 slagalice, te prvih par koraka rješavanja po slojevima. Sva neoznačena polja označavaju bilo koju konfiguraciju pločica koja ostaje nakon slaganja nekog sloja (među njima se negdje nalazi i stvarno prazno polje). Preostalu 3×3 slagalicu složimo npr. heuristikom.

3.4 Ekspertni sustavi

Temeljne operacije koje obavljaju ekspertni sustavi su rješavanje problema iz nekog uskog područja, obrazloženje rješenja (poput živih stručnjaka, ekspertni sustav mora moći pojasniti svoje odluke i zaključke), dijalog s korisnikom (postupak rješavanja se mora moći odvijati i interaktivno tako da korisnik može utjecati na tok procesa rješavanja problema) te rad s nepotpunim i nedeterminističkim znanjem.

3.4.1 Osnovna struktura ekspertnog sustava

Sasvim općenito, ekspertni sustav se može podijeliti na tri modula:

- baza znanja
  Sadrži specifična znanja o području primjene (**domeni**) ekspertnog sustava. Ona može sadržavati činjenice, pravila i metode za rješavanje problema iz domene.
- mehanizam zaključivanja
  Pronalazi rješenje problema koji je zadao korisnik na temelju znanja iz baze znanja.
- korisničko sučelje
  Omogućava korisniku uvid u postupak zaključivanja i proširivanje baze znanja. Često se mehanizam zaključivanja i korisničko sučelje naziva **ljuskom** ekspertnog sustava.

Tako se sugerira da se ekspertni sustav sastoji od znanja o domeni i algoritma za baratanje tim znanjem (mehanizam zaključivanja i korisničko sučelje). Prednost takve podjele je u tome što je ljuska načelno *neovisna o domeni.* Tako se jedna te ista ljuska može koristiti u različitim ekspertnim

sustavima u komunikaciji sa različitim bazama znanja. Naravno, tada sve baze znanja moraju biti izvedene u skladu s formalizmom kojeg opisuje ljuska (u praksi teško).

Opći plan razvoja ljuske ekspertnog sustava mogao bi biti slijedeći:

1. izbor formalizma za prikaz znanja
2. razvoj mehanizma zaključivanja sukladno s odabranim formalizmom
3. dodaci za dijalog s korisnikom
4. dodaci za zaključivanje na temelju deterministčkog znanja

3.4.2 Formalizam za prikaz znanja

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

- ako uvjet P, onda zaključak Z
- ako stanje A, onda akcija A
- ako vrijede uvjeti U1 i U2, onda vrijedi i uvjet U3

Pokazuje se da takav prikaz znanja ima slijedeća lijepa svojstva:

- modularnost: svako pravilo predstavlja mali, relativno nezavisni dio znanja
- proširivost: baza znanja sa takvom strukturom se lako proširuje novim pravilima
- jednostavno održavanje kao posljedica modularnosti: pojedina pravila baze znanja mogu se jednostavno mijenjati neovisno o ostalim pravilima
- transparentnost: mogućnost obrazloženja odluke, odnosno rješenja. Ekspertni sustav čije se znanje sastoji od if-then pravila može odgovarati na dvije vrste upita: "kako si došao do tih zaključaka?" te "zašto ti taj podatak treba?".

U nekim programskim domenama (npr. medicina) prevladavaju pojmovi stohastičke prirode (npr. povišena temperatura). Takav nedeterminizam se može izraziti tako da se if-then pravila opišu dodatnim vjerojatnosnim pravilima. Na primjer, jedno pravilo iz ekspertnog sustava MYCIN (1976):

- Ako:
  1. infekcija je primarna bakteriemija
  2. uzorak je uzet steriliziranom opremom
  3. ulazno mjesto infekcije je probavni trakt
- Onda: postoji utemeljena sumnja (vjerojatnosti 0.7) da je uzročnik infekcije bacil iz porodice bacteroides.

Iskustvo pokazuje da je za razvoj upotrebljivog ekspertnog sustava nužno sudjelovanje stručnjaka za to područje. Prikupljanje znanja konzultiranjem stručnjaka te njegovo oblikovanje u skladu sa zadanim formalizmom naziva se **inženjerstvom znanja**.[13] To je u pravilu vrlo složen, opsežan i zahtjevan zadatak, pa će se u okviru ovih vježbi razmatrati jednostavnja baza znjanja koja omogućuje raspoznavanje životinja na temelju njihovih karakteristika.

Opći oblik pravila u ovoj bazi će biti

```
ime_pravila :: if Uvjet then Zakljucak.
```

[13] knowledge engineering

pri čemu se `Uvjet` može sastojati od više jednostavnih uvjeta koji su povezani operatorima `and` i `or`. Prednost Prologa u modeliranju takvih pravila je u tome što definiranjem operatora `::`, `if`, `then`, `and` i `or` takva pravila postaju legalni stavci Prologa. Na primjer:

```
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.
```

Predložena sintaksa pravila u bazi znanja je vrlo slična sintaksi Prologa. Najjednostavniji način rada s tim pravilima bi stoga bio njihovo prepisivanje u sintaksu Prologa i naknadna primjena Prologovog mehanizma zaključivanja. Na primjer, pravilo 1 bi se u Prologu moglo zapisati:

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

Neka je Mirko tigar sa slijedećim osobinama: 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.
mirko eats meat.
```

Tada bi se mogli postaviti slijedeći upiti:

```
?- mirko isa leopard.
   No.
```

```
?- mirko isa tiger.
   Yes.
```

Iako Prolog pravilno odgovara na upite i pri tome koristi bazu znanja zadanu if-then pravilima, program iz primjera se ne može nazvati ekspertnim sustavom jer:

- Program ne možemo pitati *kako* je došao do zaključka da je Mirko tigar (od Prologa očekujemo odgovor da ili ne).
- Program ne možemo pitati *zašto* nije zaključio da je Mirko leopard.
- Sve informacije o domeni je u sustav potrebno unijeti prije postavljanja upita. Tako korisnik može zadati nepotrebne podatke (velik, lijen), a može zaboraviti bitne (npr. da Mirko jede meso).

Stoga izrađujemo ljusku ekspertnog sustava koja će moći na temelju baze o životinjama s korisnikom voditi slijedeći dijalog:

```
Question: mirko isa tiger.
Is it true: mirko has hair? yes.
Is it true: mirko eats meat? no.
Is it true: mirko has claws? why.
To investigate, by rule3, mirko is a carnivore.
...
```

Korisnik može odgovoriti na dva načina: tako da da traženu informaciju (yes/no), ili tako da pita ekspertni sustav za što mu je ta informacija potrebna (why). Korisnik odgovara pitanjem kada nije shvatio pitanje ili kada traženu informaciju nije jednostavno ili jeftino dobiti.

Pogled u mehanizam zaključivanja mogli bismo dobiti i praćenjem izvođenja programa u Prologu (trace). Praćenje je, međutim, projektirano prvenstveno za dijagnosticiranje pogrešaka u programu, pa bi njegova upotreba za uvid u mehanizam zaključivanja bila nepotrebno komplicirana jer bi korisnik dobivao višak informacija. Kada ekspertni sustav konačno odredi odgovor na početni upit korisnika, može zanemariti *kako* je sustav došao do rješenja. Na takvo pitanje ekspertni sustav ispisuje glavne korake zaključivanja.

3.4.3 Struktura programa

Glavni objekti koji se javljaju u predloženoj izvedbi ljuske ekspertnog sustava su:

- Put zaključivanja u obliku od početnog upita do tekućeg upita: `Razlog`.
- Od tekućeg upita do rješenja: `Cilj`.
- Na kraju dobivamo objekt `Odgovor`.

Glavne procedure ljuske ekspertnog sustava su:

- `istrazi(+Cilj, +Razlog, -Odgovor)` pronalazi rješenje `Odgovor` na upit `Cilj`.
- `korisnik(+Cilj, +Razlog, -Odgovor)` pita korisnika da li je cilj istinit. Po potrebi se odgovara na korisnikovo pitanje "why" tako da se pokaže dosadašnji tijek zaključivanja.
- `prikazi(Odgovor)` prikazuje konačni rezultat zaključivanja i po potrebi odgovara na korisničko pitanje *kako* na temelju objekta `Odgovor` (cijeli tijek zaključivanja).
- `ekspert` povezuje pojedine dijelove ekspertnog sustava.

4 Ugrađeni predikati

Ovdje ću opisati najkorisnije predikate u Prologu. Kod opisa predikata, svaki argument ima prije imena jedan od znakova `+-?`. `+` označava *ulazni* argument, `-` *izlazni*, a `?` bilo ulazni, bilo izlazni argument. Ima još dosta korisnih predikata (npr. ulaz i izlaz), ali ja sam ovdje opisao samo one koji su bili korisni za izradu labosa iz Prologa.

Predikati su opisani za SWI Prolog. Razlike između GNU i SWI Prologa su navedene. U manualu za GNU Prolog će se naći drugi tip parametara nego što je ovdje: tamo su većinom svi parametri navedeni kao ulazno-izlazni. Međutim, ja parametar ne smatram ulazno-izlaznim ako upotreba izlaznog parametra zablokira proceduru kod vraćanja. Stoga sam takve argumente dokumentirao kao ulazne.[14]

4.1 Ispitivanje tipa izraza

Prolog je u suštini slabo tipiziran jezik.[15] Međutim, provjera tipa izraza ponekad dobro dođe.

`var(+Term)`

Uspije ako je argument slobodna (nevezana) varijabla.

`nonvar(+Term)`

Uspije ako je argument nije slobodna varijabla.

`number(+Term)`

Uspije ako je argument vezan za broj (prirodni ili realni).

`atom(+Term)`

Uspije ako je argument vezan za atom.

4.2 Usporedba i unifikacija izraza

Izrazi[16] su poredani u tzv. "standardni poredak":

[14] GNU Prolog ih navodi kao ulazno-izlazne zato što upotreba bilo ulaznog, bilo izlaznog parametra ne napravi instantiation_error (kao što bi se desilo da se umjesto ulaznog parametra stavi izlazni ili obratno).
[15] weakly-typed
[16] Term

- varijable < atomi < stringovi
- stara varijabla < nova varijabla[17]
- Atomi se uspoređuju abecedno.
- Stringovi se uspoređuju abecedno.
- Brojevi se uspoređuju po vrijednosti. Cijeli i realni brojevi su ravnopravni.
- Izrazi se prvo provjeravaju po funktoru (abecedno), zatim po broju argumenata te na kraju rekurzivno po argumentima, počevši s lijevim.

`+Term1 == +Term2`

Uspije ako su argumenti ekvivalentni. Varijabla je identična jedino dijeljenoj varijabli.

`+Term1 \== Term2`

Isto kao `\+ Term1 == Term2`.

`+Term1 = +Term2`

Unificira `Term1` i `Term2`. Uspije ako unifikacija uspije. Ukoliko je jedan od argumenata varijabla, tada se ta varijabla veže za drugi argument.

4.3 Kontrolni predikati

`fail`

Nikad ne uspije.

`true`

Uvijek uspije.

`repeat`

Uvijek uspije. Stvara beskonačan broj točaka izbora.

`!`

Rez. Odbaci točke izbora od pozivajućeg predikata i svih ostalih predikata pozvanih nakon njega. Za detaljnije objašnjenje pogledati **odjeljak 2.3**.

`+Condition -> +Action1 ; +Action2`

Ovo je ekvivalent `if then` i `if then else` strukture u Prologu. Ukoliko nam ne treba `else` dio tada izostavimo `Action2`. Ovo je čitljivije nego da se ista struktura relizira rezom.

`\+ +Goal`

Uspije ako se `Goal` ne može dokazati. Treba paziti da `Goal` nije neinstancirana varijabla.

[17] Varijable se u stvari uspoređuju prema svojim adresama. Varijable na globalnom stacku su uvijek < od varijabli na lokalnom stacku. Programi se ne trebaju oslanjati na red kojim su varijable sortirane.

`call(+Goal, +ExtraArg1, ...)`

Pozove `Goal` uz dodatne argumente. Na, primjer `call(plus(1), 2, X)` će pozvati `plus/3` i unificirati `X` sa 3. Ovo je korisno ako ime predikata prosljeđujemo kao parametar drugom predikatu koji kasnije treba pozvati proslijeđeni predikat. Ovaj predikat se u GNU Prologu naziva `call_with_args`.

`call(+Goal)`

Ovaj predikat jednostavno izvrši `Goal`, te uspije ukoliko izvršeni cilj uspije. Ukoliko `Goal` sadrži rez, tada se taj rez *ne propagira* izvan `Goal`. Ovo je kontrolni predikat prema ISO standardu. Ako `Goal` ima neke parametre, tada se može sastaviti pomoću =.. operatora.

4.4 Ugrađeni operatori

Za detaljniji opis operatora pogledati **odjeljak 1.4**. **Tablica 4.1** daje pregled ugrađenih operatora zajedno sa tipom i prioritetom.

Prioritet	Tip	Operatori
1200	xfx	`--> :-`
1200	fx	`:- ?-`
1150	fx	`dynamic`
1100	xfy	`; |`
1050	xfy	`->`
1000	xfy	`,`
954	xfy	`/`
900	fy	`\+ not`
900	fx	`~`
700	xfx	`< = =:= =< == =\= > >= \= \== is`
600	xfy	`:`
500	yfx	`+ - /\ \/ xor`
500	fx	`+ - ? \`
400	yfx	`* / // << >> mod rem`
200	xfx	`**`
200	xfy	`^`

Tablica 4.1 Ugrađeni operatori

`+nonvar =.. ?list -nonvar =.. +list`

`Term =.. List` uspije ako je `List` lista čija je glava atom koji je glavni funktor izraza `Term`, a čiji je rep lista argumenata od `Term`. Ovaj predikat se naziva "univ". Na primjer:

```
Term = baz(foo(1))
```

4.5 Aritmetika

Pored aritmetičkih operatora, ima nekoliko korisnih predikata vezanih uz cijele brojeve.

`between(+Low, +High, ?Value)`

`Low` i `High` su cijeli brojevi uz `High >= Low`. Ako je `Value` cijeli broj, onda predikat uspije ako je `Low =< Value =< High`. Ako je `Value` varijabla, onda se `Value` uzastopno veže na sve cijele brojeve u zadanom intervalu. Ovaj predikat ne postoji u GNU Prologu.

`succ(?Int1, ?Int2)`

Uspije ako je `Int2 = Int1 + 1`. Bar jedan argument mora biti instanciran cijeli broj. Ovaj predikat ne postoji u GNU Prologu.

`plus(?Int1, ?Int2, ?Int3)`

Uspije ako je `Int3 = Int1 + Int2`. Bar dva od tri argumenta moraju biti instancirani cijeli brojevi. Ovaj predikat ne postoji u GNU Prologu.

4.6 Manipulacija listama

`append(?List1, ?List2, ?List3)`

Uspije ako se `List3` može unificirati sa rezultatom nadovezivanja prve dvije liste.

`member(?Elem, ?List)`

Uspije ako se `Elem` može unificirati s nekim članom liste.

`memberchk(?Elem, ?List)`

Ekvivalentan sa `member/2`, ali ne ostavlja točku grananja.

`delete(+List1, ?Elem, ?List2)`

Briše sve elemente iz `List1` koji se mogu unificirati sa `Elem` i unificira rezultat sa `List2`.

`select(?List1, ?Elem, ?List2)`

Bira element iz `List1` koji se može unificirati sa `Elem`. `List2` se unificira sa listom koja preostaje nakon brisanja izabranog elementa. Normalno se poziva kao `+List1, -Elem, -List2`, ali se može koristiti za umetanje elementa u listu sa `-List1, +Elem, +List2`. GNU Prolog predikat ima istu semantiku, ali različit raspored parametara: `select(?Elem, ?List, ?List)`.

`permutation(+List1, ?List2)`

Uspije ako je `List2` neka permutacija od `List1`. Ovaj predikat ne postoji u SWI Prologu.

`prefix(?Prefix, ?List)`

Uspije ako je `Prefix` neki prefiks liste `List`. Ovaj predikat ne postoji u SWI Prologu.

`suffix(?Suffix, ?List)`

Uspije ako je `Suffix` neki sufiks liste `List`. Ovaj predikat ne postoji u SWI Prologu.

`nth0(?Index, ?List, ?Elem)`

Uspije ako se element liste na koji pokazuje `Index` unificira sa `Elem`. Indeksi počinju od 0. Ovaj predikat ne postoji u GNU Prologu.

`nth1(?Index, ?List, ?Elem)`

Isto kao `nth0/3`, ali indeksi počinju od 1. Ovaj predikat se u GNU Prologu zove `nth`.

`last(?Elem, ?List)`

Uspije ako se `Elem` može unificirati sa zadnjim elementom liste. GNU Prolog ima obrnut redoslijed parametara: `last(?List, ?Elem)`.

`reverse(+List1, -List2)`

Obrne redoslijed elemenata u `List1` i unificira rezultat sa `List2`.

`flatten(+List1, -List2)`

Izravnava `List1` i unificira rezultat sa `List2`. Ovaj predikat ne postoji u GNU Prologu.

`length(?List, ?Int)`

Uspije ako `Int` predstavlja broj elemenata u listi. Može se koristiti za stvaranje liste koja sadrži samo varijable.

`merge(+List1, +List2, -List3)`

`List1` i `List2` su liste poredane u standardan poredak izraza. `List3` se unificira sa uređenom listom koja sadrži elemente iz obje liste. Duplikati se *ne* brišu. Ovaj predikat ne postoji u GNU Prologu.

4.7 Skupovi

Nijedan od ovih predikata ne postoji u GNU Prologu.

`is_set(+Set)`

Uspije ako je `Set` prava lista bez duplikata.

`list_to_set(+List, -Set)`

Uspije ako `Set` sadrži iste elemente kao `List`, ali bez duplikata.

`intersection(+Set1, +Set2, -Set3)`

Traži presjek dva skupa. Ulazne liste koje predstavljaju skupove ne moraju biti uređene, ali ne smiju sadržavati duplikate.

`subtract(+Set, +Delete, -Result)`

Briše sve elemente skupa `Delete` iz skupa `Set` i unificira rezultat sa `Result`.

`union(+Set1, +Set2, -Set3)`

Unija dva skupa. Ulazne liste koje predstavljaju skupove ne moraju biti uređene, ali ne smiju sadržavati duplikate.

`subset(+Subset, +Set)`

Uspije ako su svi elementi skupa `Subset` također i elementi skupa `Set`.

`merge_set(+Set1, +Set2, -Set3)`

`Set1` i `Set2` su sortirane liste bez duplikata. `Set3` se unificira sa uređenom listom bez duplikata koja sadrži uniju elemenata prva dva skupa.

4.8 Sortiranje listi

`sort(+List, -Sorted)`

Uspije ako se `Sorted` može unificirati sa listom koja sadrži elemente `List` koji su sortirani u standardan poredak izraza. Duplikati se brišu.

`msort(+List, -Sorted)`

Isto kao `sort/2`, ali ne briše duplikate. Ovaj predikat se u GNU Prologu naziva `sort0`.

`keysort(+List, -Sorted)`

`List` je lista parova `Key-Value`, tj. struktura kojima je funktor "`-`". Sortira listu kao `msort/2`, ali uspoređuje samo ključeve (`Key` dio strukture). Sortiranje je stabilno.

`predsort(+Pred, +List, -Sorted)`

Sortira kao `msort/2`, ali za usporedbu dva elementa koristi predikat `Pred`. Ovaj predikat ne postoji u GNU Prologu.

4.9 Sva rješenja

`findall(+Var, +Goal, -Bag)`

Stvara listu instanciranja koje varijabla `Var` poprima uzastopnim vraćanjem na `Goal` i unificira rezultat sa `Bag`. Uspije sa praznom listom ako `Goal` nema rješenja.

`bagof(+Var, +Goal, -Bag)`

Unificira `Bag` sa alternativama od `Var`. Ako `Goal` ima slobodne varijable osim onih koje dijeli sa `Var`, `bagof` će se vraćati preko svih izbora za te varijable, unificirajući `Bag` sa odgovarajućim izborima za `Var`. Konstrukcija `Var^Goal` govori da `bagof` ne veže varijablu `Var` u cilju `Goal`.

`setof(+Var, +Goal, -Bag)`

Kao `bagof/3`, ali koristi `sort/2` da se dobije sortirana lista bez duplikata.

4.10 Baza podataka

Da bi se na nekom predikatu mogli koristiti predikati iz ovog odjeljka, on mora biti deklariran kao `dynamic`, npr.

```
:- dynamic input/3
```

`abolish(Predicate/Arity)`

Briše sve stavke koji imaju funktor `Predicate` i broj argumenata `Arity`, npr: `abolish(input/3)`.

`retract(+Term)`

Kad je `Term` atom ili izraz, unificira se sa prvom činjenicom iz baze. Ta činjenica ili stavak se briše iz baze.

`asserta(+Term)`

Umeće činjenicu ili stavak u bazu. Stavak se umeće kao prvi stavak odgovarajućeg predikata.

`assertz(+Term)`

Kao `asserta/3`, ali se stavak umeće kao *zadnji*.

`assert(+Term)`

Kao `assertz/3`.