Slaviša B. Prešić

Поштованом колеги
професору Др Жарку Мијајловићу
са
најбољим
жељама
1.3.96г.
Ђ Ђ

PROLOG

Relacijski jezik

Veliki broj primera i zadataka,
lakših i težih, rešenih i nerešenih

Podroban opis opšteg
algoritma Prologa

Matematički fakultet, Beograd

NAUKA

Beograd, 1996 g.

Profesor dr. Slaviša B. Prešić, Matematički fakultet, Beograd

P R O L O G
Relacijski jezik

Recenzenti

Profesor dr. Žarko Mijajlović, Matematički fakultet, Beograd
Profesor dr. Đura Paunić, Institut za matematiku, Novi Sad
Istraživač-saradnik mr. Miodrag Kapetanović, Matematički institut, Beograd

Izdavači
Matematički fakultet, Beograd
Studentski trg 16

IP, ‚Nauka', Beograd
Bulevar Revolucije 314/25
tel-faks (011) 421-834

Za izdavače
Profesor dr. Zoran Kadelburg, dekan
Nikola Dončev, direktor

Korice
Nenad Beočanin

CIP - Katalogizacija u publikaciji
Narodna biblioteka Srbije, Beograd

519.682 "Prolog"
PREŠIĆ, Slaviša B.
Prolog : relacijski jezik / Slaviša B. Prešić.- Beograd: Matematički fakultet: Nauka, 1996 (Beograd: Beopres).-316 str.; 24 cm.

Tiraž 500. -Bibliografija: str. 313.-
Registar.

a) Programski jezik "Prolog"
ID=44071948

Štampa: ‚Beopres', Beograd.

S A D R Ž A J

R e č - d v e o k n j i z i

Knjiga se odnosi na tri verzije Prologa: Micro, Arity i LPA- prolog. U njoj se postupno i podrobno izlaže opšti mehanizam (algoritam) Prologa. Izlaganje ide od prvih, jednostavnijih primera i odgovarajućih proloških pravila do potpunog opisa opšteg proloških algoritma. Taj opis je na jeziku tzv. i-ili drveta i sadržan je u tački 5. Pored toga, u knjizi ima podrobnijeg izlaganja o

algoritmu unifikacije (deo 5.1 tačke 5)
vezi Prologa i matematičke logike (tačka 7)
gramatikama na koje se uspešno primenjuje Prolog (deo 4.4 tačke 4)
bazama podataka (tačka 8)
pravljenju .exe, .com -fajli u Prologu (tačka 9)

U knjizi ima veliki broj rešenih zadataka, u kojima se često pojavljuju

fajle, prozori, (algoritamski) svetovi i dr.

U tački 12 je dat veoma iscrpan pregleg predikata ugrađenih u Prolog, odnosno u Micro, Arity, LPA- verziju.
Knjiga se može koristiti i za početno učenje Prologa, ali takođe i za produbljivanje i logičko sagledavanje tog jezika. Sledstveno, pored raznih neškolskih lica nju mogu koristiti djaci i studenti raznih uzrasta, uključujući i one na trećem stepenu studija.

1. RELACIJE, FORMULE

1) Za Prolog se može reći da je relacijski jezik, budući da u njemu pojam (matematičke) relacije ima osnovnu ulogu.
U vezi sa relacijama uopšte, u matematičkoj logici a i u mnogim verzijama Prologa, koriste se zapisi oblika, tzv. formule :

(1.1) $$rel(a1,\ldots,an)$$

(čitati : "*rel* od a1,...,an " ili objekti a1,...,an su u relaciji *rel*)

gde je *rel* tzv. ime relacije (reći ćemo i formule (1.1)), dok su a1,...,an objekti na koje se odnosi ta relacija. Odmah istaknimo da ćemo pored zapisa (1.1) koristiti i ovakav zapis (to je tzv. l i s t a) :

(1.2) $$(rel\ a1\ \ldots\ an)$$

koji je, u stvari, pozajmljen iz jezika LISP[1]. Inače, zapisi tipa[2] (1.2) koriste se u osnovnoj verziji Micro-prologa.

Napomenimo da je u literaturi već uobičajeno da se za zapise tipa (1.1)[3] kaže da pripadaju tzv. Edinburgškoj sintaksi. Većina današnjih Prologa koristi tu sintaksu. Međutim, osnovna verzija Micro-prologa kao što smo rekli koristi jezik lista, reći ćemo drukčije koristi: LISP-sintaksu.
Napomenimo da ćemo u ovoj knjizi koristiti ove verzije Prologa:

Micro-prolog, kao predstavnika Lispovske sintakse
Arity-prolog i LPA-prolog koji koriste Edinburgšku sintaksu

Istaknimo, da ćemo u daljim delovima knjige postupno upoznavati opšti proloski algoritam. Na ovom mestu pomenimo da će se tokom algoritma u svakom koraku pojavljivati računanje vrednosti neke formule, a sama ta vrednost će biti tačno, netačno[4] , znači potpuno slično kao što je to slučaju matematičkoj logici (i matematici uopšte). Znači, kratko rečeno u Prologu će osnovni sastavci, "atomi" biti formule[5].

---

[1] Lisp je jedan od najstarijih viših programskih jezika. Nastao je krajem pedesetih godina. Izumeo ga je John Mc Carthy. Lisp se veoma često koristi u oblasti Veštačke inteligencije. Za razliku od Prologa on je pravi funkcijski jezik. U Lispu osnovnu ulogu imaju liste oblika (1.2), ali tada je *rel* oznaka neke funkcije. Recimo, (+ 2 3) iznosi 5, dok (+ 2 (* 3 4)) iznosi 14. Inače, reč "lisp" dolazi od engleskih reči List Processing.

[2] U stvari, osnovna verzija Micro-prologa koristi još opštije formule. Bliže to su liste oblika (rel! _X). Više o tome kasnije (vid. (4.1.4)).

[3] U naše vreme već ima priličan broj verzija prologa, kao:
DEC-10 Prolog, CProlog, Quintus, Arity, LPA, Micro-prolog, Prolog-1, Prolog-2, Prolog-3, Sigma Prolog, Turbo Prolog , i dr.
Međutim, uz neke nevelike razlike svi oni u osnovi imaju isti mehanizam, isti algoritam (kaže se i Warren-ovu "mašinu", jer David Warren je 1977 u potpunosti opisao taj algoritam).

[4] Ponekad ćemo umesto reči tačno, netačno koristiti redom reči da, ne (na engleskom yes, no).

Primer 1.1. Sa LESS se u Micro-prologu označava relacija biti manji. Tako, tačne su ove formule (LESS 1 2), (LESS 5 6), (LESS 2 3.4). U slučaju LPA, kao i Arity -prologa umesto tih imamo redom ove zapise 1<2, 5<6, 2<3.4. To u stvari, nije zapis tipa (1.1), već sada ćemo reći njegovo proširenje. Naime, kao i uopšte u matematici, ako je n=2 obično se umesto rel(a,b) piše: a rel b. Primera radi, umesto =(x,y) pišemo x=y i sl.

Primer 1.2. Sa EQ se u Micro-prologu označava relacija jednakost[6]. Tako, tačne su formule (EQ 2 2), (EQ 6 6). U LPA, kao i u Arity-prologu te se formule pišu u obliku 2=2, 6=6.

Primer 1.3. Kao što znamo sabiranje(brojeva) je operacija, a ne relacija. Međutim, uvek se operaciji može pridružiti odgovarajuća relacija. Recimo, u vezi sa sabiranjem možemo uočiti ovu relaciju zbir:

$$zbir(x,y,z) \Leftrightarrow x + y = z$$

U svakoj verziji Prologa postupa se u duhu te definicije . Tako u Micro-prologu imamo ovu definiciju (umesto reči zbir koristi se reč SUM) :

(1.3) $$(SUM\ \_X\ \_Y\ \_Z) \Leftrightarrow \_X + \_Y = \_Z$$

Prema toj definiciji imamo primere: (SUM 1 2 3), (SUM 2 -3 -1) i sl. U (1.3) simboli _X, _Y, _Z mogu biti zamenjeni ma kojim vrednostima (brojevima) - to su tzv. promenljive. Od sintakse Prologa zavise oznake promenljivih. Tako:

(1.4) U Edinburgškoj sintaksi promenljive su ma koje reči koje počinju nenekim velikim slovom, kao: A, X, Y23, Beograd, a takođe kao u (1.3) počinju znakom donje podcrte. Takve, sa podcrtom počinjuće, promenljive su jedine vrste promenljivih u slučaju Micro-prologa.

U Micro-prologu se množenju pristupa slično sabiranju. Naime, u tu svrhu koristi se relacija TIMES :

$$(TIMES\ \_X\ \_Y\ \_Z) \Leftrightarrow \_X * \_Y = \_Z.$$

Recimo, tačna je formula (TIMES 2 3 6). U vezi sa SUM i TIMES dodajemo i sledeće: kod njihovih formula je dozvoljeno da jedan od argumenata bude promenljiva, koja još nije dobila vrednost. Tako, "smislene" su formule:

(SUM _x 2 3), (SUM 3 2 _y), (SUM 2 _z 15)

i ukoliko _x, _y, _z još nemaju vrednosti, onda sve tri su tačne, i dodatno (kao tzv. bočni efekat) te promenljive dobiju redom vrednosti 1, 5, 13. . Međutim, ako recimo _a, _b
nemaju nikakve vrednosti, formula (SUM _a _b 7) je besmislena, i pri pošaju njenog računanja Prolog prekida algoritam porukom: CONTROL ERROR.

---

[5] Istaknimo da se u mnogim proloskim knjigama sreće malo drukčije izražavanje. Tako, umesto formula obično se kaže cilj, pa dalje shodno tome se umesto recimo računamo formulu ϕ kaže postavljamo cilj ϕ, i sl. Mi inače namerno koristimo "formulsko" izražavanje, jer tako smo manje udaljeni od pravog, odnosno strogog opisa opšteg proloskog algoritma.

[6] Ubrzo će o njoj biti više reči.

[7] U stvari, takvu promenljivu je bolje zvati nepoznata(vid. Napomenu 2.1.1)

Primer 1.4. Uočimo skup S={1,2,3,4,5,6} i jednu njegovu relaciju p određenu rečima biti činilac broja 6.U duhu Prologa relacija se zadaje navođenjem svih slučajeva njenog važenja, odnosno ovim formulama:

(p 1) , (p 2) , (p 3) , (p 6)

U zapisu **(1.1)**, odnosno **(1.2)** se za n kaže da je dužina relacije rel.Relacija LESS (odnosno <), EQ (odnosno =) su binarne, SUM i TIMES su ternarne dok poslednja relacija p je unarna.

Primer 1.5. Odrediti sve unarne relacije skupa {1,2,3}.
Ima osam unarnih relacija skupa {1,2,3} i to su :

1) Prazna relacija p; ne važi ni jedna od formula (p 1), (p 2), (p 3).
2) (p 1) (Znači samo 1 je u relaciji p)
3) (p 2)
4) (p 3) 5) (p 1),(p 2) 6) (p 1),(p 3)
7) (p 2),(p 3) 8) (p 1),(p 2),(p 3) ,tj.puna relacija.

2) Rekli smo za Prolog da je relacijski jezik, i da su (1.1),odnosno (1.2) formule u kojima je *rel* ime neke relacije. Međutim, kao što ćemo odmah videti, u njemu se pojam relacije uzima dosta opštije nego uopšte u matematici ili u matematičkoj logici.

Tako, *prvo,* u Prologu se mogu koristiti i relacije "mešovite dužine". Recimo, u vezi sa brojevima 1,2,3 uočimo ove formule:

(q 1), (q 2), (q 1 3), (q 2 1)

kojim je definisana jedna relacija q. Za nju imamo:

- brojevi 1 i 2 ponaosob su u relaciji q,
- broj 1 je u toj relaciji sa 3, i konačno
- broj 2 je u istoj relaciji sa 1.

Drugo, u **(1.1)** i **(1.2)** se dopušta čak da bude n=0.Tada se koriste zapisi :(p), (q),..u LISP- ,odnosno p. ,q., ...u Edinburgškoj sintaksi. Tada dogovorno kažemo da su p,q,...."nularne" relacije. Malo kasnije ćemo upoznati primere iz kojih će biti jasan smisao i korisnost takvih relacija.

Treće, u formuli (1.1), odnosno (1.2) za a1,a2,....,an smo malo neodređeno rekli "objekti na koje se odnosi relacija *rel*". Naravno budući da je reč o Prologu,dakle programskom jeziku, objekti mogu biti oni sa kojima on radi. Evo ukratko njihovog opisa.

Objekte delimo na proste i složene. U proste dolaze:

promenljive, brojevi (celi i realni) i konstantske reči, tj.reči koje nisu ni promenljive ni zapisi brojeva. U Micro-prologu se za njih koristi naziv konstante,a u Edinburgškoj sintaksi se nazivaju atomi.Primeri konstantskih reči:milan, k23, Jovan (u Micro-prologu), 'Milan'. Blagodareći navodima poslednja reč se u u obe sintakse prihvata kao konstantska.

Složeni objekti se grade pomoću prostih. Osnovna vrsta složenih objekata su liste. Micro-prolog i nema drugih. U Edinburgškoj sintaksi pored lista se dopuštene i tzv. strukture.

Primer: f(x,g(Y,23)). U tom primeru f je tzv. ime strukture. To inače

može biti ma koja konstantska reč(atom)[8].Tu se f odnosi na dva sastavka(dve komponente) x i g(Y,23). Drukčije se kaze da je f dužine 2. Dalji primer struktura je s(X,23,h(a,f(Y),f(X,Y))), gde s i h imaju dužinu 3, a f se pojavljuje dvaput.U delu f(Y) f ima duzinu 1,a u delu f(X,Y) ima dužinu 2)

U stvari u obema sintaksama se govori i o tzv. niskama (stringovima), kao "Milan" i sl. Medjutim, u Prologu se takve niske slova po pravilu shvataju kao liste, pa elem u suštini to i nije nova vrsta objekata.

Evo sada primera formula uz učešće opisanih objekata

(i) Edinburgška sintaksa:

f(X,g(Y,Z))
f(X,f(X)) Pazite, samo prvi f se tumači relacijski,a drugi u stvari, stvari ima "funkcijsku" prirodu. Eto, to je jedna od posebnosti Prologa. Kasnije ćemo videti primere u koji se to koristi.

(ii) LISP-sintaksa:

(f _X (g _Y _Z))
(f _X (f _X)) To su prevodi formula (i) na LISP-sintaksu.

Kao što se odatle već vidi, ma kojoj strukturi oblika f(Ob1,Ob2,...,Obn) se u LISP-sintaksi može pridružiti lista (f Ob1 Ob2 ... Obn) što u stvari, znači da Edinburgška sintaksa nije sintaksno jača od Lispovske. Može se reći i da je Lisp-sintaksa osnovna,a da je Edinburgška njeno tehničko proširenje, zbog koga je, tako neki misle, ona podesnija za upotrebu.
Međutim, u vezi sa tim dodajmo i sledeće veoma važno. U Edinburgškoj sintaksi postoji osnovni predikat, u oznaci =.. koji ,slobodnije rečeno, služi kao most između lista i struktura. Tako, uopšte imamo jednakost oblika

f(Ob1,Ob2,...,Obn)=..[Ob1,Ob2,...,Obn]

gde sa desne strane stoji (u Edinburgškoj sintaksi) lista čiji članovi su Ob1,Ob2,...,Obn. Inače o relaciji =.. više govorimo u tački Zadaci,I (Zadatak 3.28).

3) A sada, ukratko o nekim relacijama(predikatima) ugrađenim u Prolog. Do sada smo pominjali SUM,EQ i dr.

I)U vezi sa računskim operacijama već smo imali SUM i TIMES.U Edinburgškoj sintaksi se koriste uobičajene oznake osnovnih računskih operacija:
+, *, -, / ,ali uz to se koriste i dodatne oznake:

// za količnik pri delenju celih brojeva.Recimo,8//3 je 2, 14//4 je 3.
mod za ostatak delenja celih brojeva.Tako, 17 mod 4 je 1.
^ za stepenovanje. Recimo, 2^3 je 8.

U skladu sa rečenim u Edinburgškoj sintaksi je dopuštena upotreba "aritmetičkih" izraza, kao 4+3*X-Y^2 i sl.

II) Predikat EQ, u Edinburgškoj sintaksi je to =, je u osnovi predikat unifikacije (ujednačavanja) o kome će već u idućoj tački biti više reči. Tako, ako je _x promenljiva koja još nije dobila vrednost onda računanjem formule (EQ _x 4) dobija se vrednost tačno, i uz to se _x -u dodeli vrednost 4. Ali, ako sticajem okolnosti se traži vrednost gornje formule, a _x

[8] Ali ne i neka od "službenih" reči Prologa, kao not (smisao: ne) i dr.

je prethodno već dobilo neku vrednost, onda

prvo, tom _x se neće promeniti vrednost, a drugo formula (EQ _x 4) će biti tačna, netačna u zavisnosti od toga da li je vrednost _x-a jednaka, nejednaka 4.

U Edinburgškoj sintaksi pored rečenog ima i dodataka. Recimo, ako je _x promenljiva bez vrednosti, onda pri računanju formule _x=2+5 njoj će se kao vrednost pridružiti izraz 2+5, ali ne broj 7. Prosto, = znači unificirati, što ovde znači zameniti _x izrazom 2+5.

Evo još jednog malog primera sa EQ, tj. =. Računanjem formule

(EQ (_x 2) ((f _y) _y)), tj. formule [_x,2]=[f(_y),_y]
(_x, _y su po pretpostavci promenljive bez vrednosti)

će se dobiti rezultat tačno, i uz to prihvatiti ove zamene

_x --> (f 2) tj. _x -->f(2)
_y --> 2 tj. _y -->2

koje definišu vrednosti za _x, _y.

III) Kako u Edinburgškoj sintaksi, budući da ona dopušta korišćenje aritmetičkih izraza, nekoj promenljivoj dati vrednost jednaku vrednost nekog izraza ? Toj svrsi služi predikat is. Naime, ako je izraz ma koji aritmetički izraz i X neka promenljiva koja trenutno nema vrednost, onda pri računanju formule: X is izraz toj promenljivoj se kao vrednost dodeljuje vrednost tog izraza, ako ta vrednost postoji, a inače se prijavljuje greška i algoritam zaustavlja. Recimo, u slučaju ovih formula

X is 2+5, X is 6-3^2 , X is 2/0

promenljivoj X se dodeljuje 7, odnosno -3, u vezi sa poslednjom se prijavljuje greška. U vezi sa is-formulama dodajmo i da je dopušteno da im "leva" strana bude konstanta. Tako, smislene su formule

7 is 4+3 , 7 is 4*(2+3)

i njihove vrednosti su tačno, odnosno netačno.
Međutim, u vezi sa is predikatom dodajmo još da ovakva is-formula

3+2 is 2+3

"ne radi", kako bi se moglo pomisliti, tj. da se računaju leva i desna strana i proveri važenje jednakosti. U tu svrhu služi nov predikat =:=. Naime, kratko rečeno, ako su izraz1, izraz2 neka dva aritmetička izraza, onda vrednost formule

izraz1 =:= izraz2

je tačno, netačno već prema tome da li ti izrazi imaju, nemaju istu numeričku vrednost[9].

IV) Po malo je suptilan predikat ==, koji je približno opisiv rečima " bukvalno jednaki" . Recimo, formule X==X, 2==2, f(2)==f(2), [X,g(3)]==[X,g(3)] su tačne, jer im se leva i desna strana "bukvalno poklapaju". Ali, formula X==Y, gde su X, Y promenljive trenutno bez vrednosti, je netačna, jer

[9] Ako neki od njih nema vrednost algoritam se prekida sa porukom o grešci.

se ne dopusta[10] "naknadna unifikacija".

Primetimo da Micro-prolog nema takav predikat ==, ali u zamenu za tim ima predikat GRNHOL, o čemu će kasnije biti više reči.

V) Evo još nekoliko predikata sa kratkim opisom:

(INT _x),	odnosno integer(_x)	smisao :	_x je ceo broj, što znači da x ima vrednost koja je ceo broj[11]
(NUM _x),	odnosno number(_x)	smisao :	vrednost _x-a je broj
	float(_x)	smisao :	vrednost _x-a je realan broj
(VAR _x),	odnosno var(_x)	smisao :	_x je promenljiva koja još nije dobila vrednost ili je dobila vrednost, koja je promenljiva.
(CON _x),	odnosno atom(_x)	smisao:	_x je konstantska reč
	atomic(_x)	smisao:	_x je broj ili konstantska reč

Pomenimo da, u Micro-prologu nema nekih predikata, kao recimo float, atomic.

VI) Uz sve opisane predikate navedimo da u Edinburgškoj sintaksi za negacije neke od njih postoje posebni predikati. Tako, recimo, imamo:

negacija za	<	je >=
negacija za	>	je =<
negacija za	=	je \=
negacija za	==	je \==
negacija za	=:=	je =\=

Međutim, istaknimo da to nije bitno i sledstveno se ne mora koristiti jer, kao što ćemo već u narednoj tački videti, Prolog ima opšti predikat negacije, u oznaci NOT, odnosno not.

VII) Na kraju napomenimo da do sada naveden spisak u Prolog ugrađenih predikata nikako nije potpun. U daljem izlaganju ćemo ga dopunjavati. Inače, potpuni spisak takvih predikata se po pravilu može naći u Priručniku odgovarajuće verzije Prologa.

[10] Međutim, ako se tokom algoritma prvo pojavi X=Y, što znači da su X, Y "vezani" za istu adresu, a onda iza toga se računa formula X==Y, onda njena vrednost je tačno.

[11] U Micro-prologu INT predikat ima još jedan oblik: (INT _x _y), sa smislom _y je celi deo broja _x. Recimo, tačna je formula (INT 3.2 3). Drugi argument sme da bude promenljiva. Recimo, pri računu formula (INT 4.5 _x) promenljivoj _x se dodeljuje 4.

2. PRVO O PROLOŠKOM ALGORITMU

2.1 P o č e t a k

U ovom izlaganju, pri definisanju proloških programa, najpre ćemo koristiti lispovske zapise (oblik (1.2)) , tj. LISP-sintaksu, odnosno definisati programe u osnovnoj verziji Micro-prologa. Međutim, ubrzo ćemo upoznati pisanje programa i u Edinburgškoj sintaksi, gde osnovnu ulogu imaju zapisi oblika (1.1). Tako, neka su for1, for2, ..., fork neke formule oblika (1.2). Tada zapise oblika :

(2.1.1) (for1 for2 ... fork)

nazivamo *članak*. Ukoliko k>1 čitamo ga ovako:

for1 a k o for2 i for3 i ... i fork

Spoljne zagrade u (2.1.1) zvaćemo *poveznice*. Znači, ukratko, dopisivanjem nekoliko formula i dodavanjem poveznica nastaje članak. Može se desiti k=1. Tada (2.1.1) glasi (for1) i čitamo je: *Važi for1*. Primeri članaka:

((a 1)(b 1 2)(c 3)) (Znači, 1 je u relaciji a, ukoliko 1 i 2 su u relaciji b, i još 3 je u relaciji c).

((a 1)) (Znači, važi formula[1] (a 1), tj. 1 je u relaciji a)

U vezi sa člancima uopšte dodajemo sledeće: u Prologu se u članku oblika (2.1.1), dopušta da neki od for2, for3, ..., fork bude jedan od ovih posebnih znakova:

/ (tzv. "rez", engleski cut)

FAIL (čitati feil, ili NE, jer se koristi u značenju bliskom sa ne)

koje ćemo, da bismo objedinili izlaganje, takodje smatrati formulama. Imajući pojam članka neposredno se uvodi pojam prološkog programa. To je ma koji konačan niz: Članak1, Članak2, ..., Članakn članaka, pri čemu je njihov redosled bitan.

Pre navođenja primera opisujemo izgled članaka u Edinburgškoj sintaksi, odnosno u LPA- i ARITY-prologu. Tada, prvo jednočlani članci se umesto u obliku ((rel a1 a2 . . . an)) pišu ovako rel(a1,a2,. . .,an). sa tačkom na kraju[2]. Dalje, duži članci se pišu ovako

rel1(...) :- rel2(...),rel3(...), . . . ,relk(...).

gde u zagradama umesto ... stoje odgovarajući argumenti. Primetimo da sa desne strane između pojedinih formula stoji zarez i da je na kraju tačka. I sada je dopuštena upotreba gore navedenih posebnih znakova, s tim što

umesto / se piše ! , a umesto FAIL se piše fail

Primer 2.1.1. U vezi sa relacijama p,q,r,s,u uočimo ovaj program

---

1 Pazite, zapis oblika (a 1) nije članak, jer mu nedostaju poveznice.

2 Dopunimo još da se članci poput ((a)) ((b)) sa nularnim relacijama a, b, u Edinburgškoj sintaksi pišu ovako: a. b.

(u LISP-notaciji, tj. u Micro-prologu)

```
(2.1.2)     ((p 1))    ((p 2))    ((p 3))
            ((q 3))    ((q 44))
                 ((r _X)(p _X))
                 ((r _X)(q _X))
            ((s _X)(p _X)(q _X))
                 ((u 2))
```

Inače, krajnje jednostavno se taj program prevodi na LPA, odnosno ARITY-verziju. Prevod glasi

```
p(1).  p(2). p(3).
q(3).  q(44).
r(X):-p(X).
r(X):-q(X).
s(X):-p(X),q(X).
u(2).
```

Naravno, ti programi su međusobno ekvivalentni, samo se prividno razlikuju po formi.

Šta je logički smisao programa (2.1.2)? Kratko rečeno, (2.1.2) treba shvatiti kao data tvrdjenja (date pretpostavke, možemo reći i date aksiome). Tako, prema (2.1.2) 1,2 i 3 su u relaciji p, dok su 3 i 44 u relaciji q. Dalje, članci ((r _X)(p _X)),((r _X)(q _X)) poručuju da je neki predmet _X u relaciji r ukoliko je on u relaciji p ili je on u relaciji q. Slično, članak ((s _X)(p _X)(q _X)) poručuje da je _X u relaciji s ukoliko je taj _X i u relaciji p i u relaciji q. Recimo, u skladu sa rečenim važi (r 2), jer važi (p 2), slično važi (r 44) jer važi (q 44). Dalje, važi (s 3) jer tačne su obe formule (p 3), (q 3).

Već na tom mestu istaknimo da uopšte *ma koji proloski program P je logički shvatljiv kao konačan niz datih tvrdjenja (datih aksioma)*, koje su doduše veoma posebnog oblika[3]. Dalje, skladno tome, osnovna stvar za koju je namenjen Prolog kao programski jezik jeste da uposli svoj mehanizam (svoj algoritam) u cilju[4] da odgovori na pitanje oblika

(2.1.3) Da li neko dato ϕ sledi iz programa P ?

tj. da raspravi da li ϕ jeste teorema datih aksioma P. Recimo, u vezi sa (2.1.2), zamišljajući da smo "ušli" u Micro-prolog i "učitali" program (2.1.2), možemo postaviti razna pitanja oblika (2.1.3). Tako, pitanje oblika da li važi (p 1), tj. da li to sledi iz (2.1.2) se bukvalno ovako postavlja

&?((p 1))

gde & je z n a k (prompt) samog Micro-prologa, tj. on je sam već prisutan, a mi kucamo ?((p 1)). Odgovor naravno treba da bude da (po engleskom yes). Međutim, Micro-prolog, odnosno njegova osnovna verzija je "siromašna" u komentarima, pa se kao znak odobravanja, umesto yes, na ekranu pojavi znak &, tj. znak jezika. Međutim, ako pitamo &?((p 5)) odgovor treba da bude ne (po engleskom no). Sada se na ekranu pojavi ova slika

---

3 To su tzv. Hornovski oblici, o čemu izlažemo u tački 8.

4 U stvari, Prolog nije u stanju da odgovori na svako pitanje oblika (2.1.3). O tome više u Primeru 2.2.2.

&?

tj. pojavi se ? kao znak negacije, tj. nije tačno (p 5).

Pomenimo da su LPA i ARITY "izdašniji" u komentarima. Naime kod njih se na ekranu pojavljuje yes, no u zavisnosti da li je odgovor da, odnosno ne. Da bismo završili sa upoznavanjem takvih, u osnovi, tehničkih detalja postavimo ovo pitanje &?((p _X)), čiji smisao u Micro-prologu je: Da li važi (p _X) za neko _X. Budući da je to tačno, na ekranu će se pojaviti znak &. Međutim, u LPA- i ARITY-prologu smisao pitanja ?-p(X). je da li ima X za koje važi p(X) i uz to, ako ga ima štampaj ga! Sledstveno na ekranu nu se pojavi[5] 1 jer, konačno, 1 je "jedno rešenje" relacije p(X), u stvari prvo po redu. Da bismo slično postigli u Micro-prologu postavljamo pitanje

?((p _X)(PP _X))

gde (PP _X) znači štampaj _X. Njegov smisao je

Da li ima _X takvo da (p _X), ako ga ima, prvo koje nađeš, štampaj.

Napomena 2.1.1. U Prologu, kao i uopšte u matematici, postoji jasna i važna razlika između pojmova promenljiva i nepoznata. Naime, u Prologu se dogovorno[6] reči kao[7] _X, _Y, _a23 smatraju kao promenljive. To su, možemo tako reći, sintaksne[8] promenljive. Da bi stvar bila jasnija, podsetimo da se neki objekat smatra (stvarnom) promenljivom ako ima odliku zamenljivosti nekim datim objektima, tzv. vrednostima (te promenljive). Recimo, kad kažemo da jednakost

(*) x+y=y+x

važi za sve realne brojeve, onda to znači da su x, y stvarne promenljive i da sledstveno imaju rečeno svojstvo zamenljivosti. To praktično znači da važe sve jednakosti kao 2+3 = 3+2, 4.5 + 8 = 8 +4.5 koje nastaju iz (*) za posebne vrednosti od x i y. Inače, budući da su x, y stvarne promenljive u (*), za (*) obično kažemo da je identitet. Vratimo se Prologu. Kratko rečeno, u njemu svaki članak je svojevrstan identitet, tj. objekti _X, _y, .. koji recimo, učestvuju u članku su stvarne promenljive. Primera radi, u članku s(X):-p(X), q(X). X sme biti zamenjeno ma kojim proloških objektom. Recimo, iz tog članka, tog "identiteta" slede ove posebne "implikacije"

```
s(1):-p(1),q(1).          s(555):-p(555),q(555).
s(f(4)):-p(f(4)),q(f(4)). s(3):-p(3),q(3).
```

Inače jasno je da od svih njih od koristi je samo poslednja.

Pogledajmo sada gornje pitanje ?-p(X). Kakav je "status" tog X-a? Naravno, sintaksno gledano to X je promenljiva. Ali, očigledno je da X ne smemo zamenjivati ma čime, tj. to X nije stvarna promenljiva . Odgovor je kratak:

Sintaksna promenljiva X u pitanju ?-p(X) ima status nepoznate.

Na primer, sličan je slučaj i sa x u jednačini 5x+8 = x-7 . To je školski

---

[5] U LPA i Arity se nudi mogućnost da se pored prvog redom ispisuju i ostala rešenja.

[6] O tome smo govorili u prošloj tački.

[7] U Edinburgškoj sintaksi takođe reči kao A, J, K9 i sl.

[8] Hoćemo da istaknemo da su uvedene po dogovoru, po pravilu.

primer nepoznate. A sada izrecimo sledeću važnu opštu činjenicu za Prolog:

(2.1.4) Sintaksne promenljive koje se pojavljuju u datom proloskom pitanju imaju status n e p o z n a t i h, odnosno nisu stvarne promenljive.

Kraj Napomene 2.1.1.

2.2 N e k a o p š t a p r a v i l a

U daljem rasuđivanju pokušaćemo da korak-za-korakom prodremo u proloski mehanizam , odnosno način na koji Prolog raspravlja pitanje teoremnosti za dato pitanje ϕ. Da bismo lakše izlagali dogovorno kod članka oblika (2.1.1) prvu njegovu formulu tj. for1 nazivamo glava tog članka. Tako, recimo, kod programa (2.1.2) glave pojedinih članaka, sastavaka programa, su

(p 1), (p 2), (p 3), (q 3), (q 44)
(r _X), (r _X), (s _X), (u 2)

Dalje, članke koji imaju samo jednu formulu nazivamo elementarne aksiome , a ostale nazivamo[9] pravila. Sada navodimo nekoliko pitanja i pored njih u duhu Prologa rasuđivanja kojim se ta pitanja raspravljaju. Takođe ispisujemo opšte pravilo kad god se ukaže takva prilika.

P i t a nj e ?((p 2))

Rasudjivanje: Gledamo redom, idući po p-člancima, da li je formula (p 2) ujednačiva[10] sa glavom nekog takvog članka. Zastanemo kad se to prvi put dogodi. Ovde se to dešava sa glavom drugog članka. Budući da je taj članak elementarna aksioma, dokaz se završava sa da. Primećuje se ovo opšte pravilo[11]

> Pravilo 1: Neka formula ϕ je prološki dokaziva ukoliko je ϕ ujednačiva sa glavom nekog članka programa i uz to prvi takav članak je elementarna aksioma.

P i t a nj e ?((p 5))

Rasudjivanje: Formula (p 5) nije ujednačiva sa glavom nijednog članka programa (2.1.2). Sledstveno (p 5) nije tačno. Primećuje se ovo opšte pravilo[12]:

---

[9] U knjigama se često umesto elementarne aksiome kaže fakti

[10] To je drugi put da pomenemo veoma značajan pojam ujednačiti(poklopiti, unificirati) bez koga se Prolog ne može zamisliti. Inače, o ujednačavanju(unifikaciji) u nastavku će stalno biti reči. Posebno jedan algoritam unifikacije će biti izložen u delu 5.1.

[11] Tu se podrazumeva izlaganje u odnosu na ma koji program P.

[12] Razlog je prirodan. Naime, ako je (p 5) dokazivo onda to mora da znači da je članak ((p 5)) slučaj neke elementarne aksiome ili da (p 5) kao zaključak sledi iz nekog drugog članka. Recimo, iz članka ((r _X)(p _X)) je zaključivo (r 3) jer važi (p 3). Međutim, (p 5) ne može se dobiti na takav na-

Pravilo 2: Formula ϕ je proloski nedokaziva ukoliko ϕ nije ujednačiva sa glavom nijednog članka programa.

Istaknimo da se u Prologu u takvom slučaju kaže da je ϕ netačna formula. Dakle u Prologu se koristi ova definicija

ϕ je netačno ukoliko *ϕ je proloski nedokazivo*

Logičku ispravnost te definicije ćemo raspraviti u tački Hornovske formule, deduktivni modeli. Nju je inače u Prolog uveo K.L.Clark, 1978.godine.

P i t a nj e ?((s 3))

Rasudjivanje: Ako je (s 3) dokazivo, onda to mora slediti iz "pravila" ((s _X)(p _X)(q _X)) jer (s 3) je ujednačivo jedino sa glavom tog članka i to ujednačavanje traži _X=3. Dalje, pitanje (s 3) se svodi na ova dva podpitanja ?((p 3)), ?((q 3)).Lako se vidi da su za oba odgovori da. Sledstveno (s 3) je dokazivo. Uopšte imamo ovo pravilo

Pravilo 3: Neka je formula ϕ ujednačiva sa glavom nekog članka programa i neka prvi takav , nakon obavljene unifikacije glasi (ϕ for2 ... fork). Tada ϕ je proloski dokaziva ako su proloski redom dokazive formule for2,...,fork.

P i t a nj e ?((r 44))

Rasudjivanje: Formula (r 44) je prvo ujednačiva sa glavom r-članka

(*) ((r _X)(p _X))

pri _X=44,pa se sledstveno zapućujemo da dokažemo (p 44) ali, lako se vidi, to ne uspeva.U vezi sa r imamo još jedan članak

(**) ((r _X)(q _X))

Sada ćemo pokušati dokazivanje pomoću njega. To daje _X=44, i novo podpitanje je da li važi (q 44).Pošto je to tačno, dakle,konačno zaključujemo da važi (r 44).U ovom rasuđivanju se pojavila ideja "pregranjavanja". Naime, ne uspevši sa (*),tj.prvom r-granom ,prešli smo na drugu granu (**). Može se reći da je prisutno ovo opšte pravilo

Pravilo 4: Pretpostavimo da smo pri dokazivanju formule ϕ koristili članak[13] ("granu") (Φ for2 ... fork) i da smo doživeli neuspeh ,tj. pomoću njega zaključili da je ϕ netačna formula. Tada ako možemo vršimo pregranjavanje, tj. pokušavamo da ϕ dokažemo preko narednog članka ("grane") sa čijom glavom ϕ je ujednačiva.

čin jer (p 5) nije ujednačivo sa glavom nijednog članka.

[13] Kad tako kažemo podrazumevamo da je taj navedeni članak nastao iz stvarnog članka tekućeg programa nakon obavljenja unifikacije i odgovarajućih zamena.

P i t a nj e ?((p _X)(PP _X)(u 1))

Istaknimo da to pitanje za razliku od dosadašnjih nije[14] jednostruko, odnosno ono je oblika ? (ϕ1 ϕ2 . . . .ϕk) sa ovim smislom :

Da li važi konjunkcija ϕ1 i ϕ2 i . . . i ϕk ?

Rasudjivanje: Prvo se pitamo da li je proloski dokazivo[15] (p _X), sa nekim _X. Imamo tri p-članka (tri p-grane) : ((p 1)) ((p 2)) ((p 3)). Zapućujemo se prvom granom. Imamo _X=1. Znači, dokazali smo (p _X) pri _X=1. Sada "dokazujemo" (PP _X), tj. (PP 1),što se svodi na pojavu 1 na ekranu. Primetimo da se tako pojavilo jedno rešenje relacije (p _X). Međutim, dalje nas čeka dokazivanje (u 1). Pošto sa u-glavom je jedini članak ((u 2)), a formule (u 1),(u 2) su neujednačive (različite su, a nemaju neke promenljive koje bi dopustile mogućnost ujednačavanja za neke njihove pogodne vrednosti). I tako, propada nam pokušaj da dokažemo kraj postavljenog pitanja. Prolog tada postupa ovako:

Pošto ne važi (u 1), vraćamo se u datom pitanju korak ulevo tj.dolazimo na (PP _X), tj.(PP 1) i probamo da to dokažemo nekom drugom granom. Ali to je očigledno besmislica, jer u vezi sa (PP 1) nema nikakvih grana. Sledstveno, idemo još jedan korak levo, tj.idemo na (p _X) i sada ćemo probati da (p _X) dokažemo na neki drugi način (nekom drugom granom).Već smo imali upotrebu grane ((p 1)).Nju kao iskorišćenu "krešemo" i sada preostaju ove dve grane ((p 2)) ((p 3)).Zapućujemo se po grani ((p 2)),i dobijamo[16] _X=2. Dalje, zbog (PP _X),na ekranu se pojavi 2 , tj.drugo rešenje relacije (p _X).Ali, opet nas sačeka (u 1), kao nedokazivo.Opet idemo ulevo,korak pa još jedan korak i tražimo nov dokaz za (p _X), s tim što prethodno "skrešemo" već korišćenu granu ((p 2)). Tako ostaje nam grana ((p 3)). Zapućujemo se po njoj. To daje _X=3. Zbog (PP _X) na ekranu se pojavi 3, tj.treće rešenje relacije (p _X). Ali opet nas čeka (u 1), kao nedokazano. Opet idemo ulevo i hoćemo da nađemo novi dokaz za (p _X), ali za (p _X) više nema grana. Znači,(p _X) više nije dokazivo. Algoritam sada staje sa porukom

?

na ekranu, jer nije dokazano (p _X), pa je celo pitanje netačno.

Kao što vidite bilo je postavljeno pitanje ?((p _X)(PP _X)(u 1)), tj.u stvari da li je teorema ova konjukcija

(p _X) i (PP _X) i (u 1)

i konačni odgovor je ne. Međutim, a to je bitno, uspelo nam je da uz put dobijemo sva rešenja relacije (p _X) .Istaknimo da je pri svemu tome uloga dela (u 1) bila pomoćna,naime umesto njega je mogla da stoji ma koja nedokaziva formula, kao (p 4),(q 1),r(7) i sl. Međutim, ukoliko nam je namera

---

[14] U stvari, mada je to malo suptilnije, i takvo složeno pitanje je na određen način shvatljivo kao jednostruko, što ćemo bolje objasniti u tački 5. Drugo o prološkom algoritmu.

[15] Možemo i ovako reći: Da li ima neke teoreme oblika (p _X), gde je _X nepoznata.

[16] Primetimo, da nismo "kresali" granu ((p 1)) sada bismo se opet njome uputili i, u stvari, dalje bismo se vrteli u krug.

da takvim pitanjem, dosetkom "prevarimo" Prolog i nateramo ga da nam ispiše sva rešenja relacije (p _X) ne moramo sami izmišljati takvu netačnu formulu, jer u Prologu u tu svrhu upravo služi FAIL, za koji se može reći da je nularna relacija koja je po definiciji netačna. U skladu sa rečenim, ako želimo da nađemo sva rešenja za (p _X), onda možemo umesto gornjeg postaviti pitanje ?((p _X)(PP _X)FAIL) Zamisao je opšta. Naime:

> **Pravilo 5:** Ako želimo da nađemo sva rešenja po X1,X2,...,Xs date konjunkcije formula $\phi 1$ i $\phi 2$ i ... i ϕk onda u tu svrhu možemo postaviti ovo pitanje
> ? ($\phi 1$ $\phi 2$ ... ϕk (PP X1 X2 ... Xs) FAIL)

Istaknimo da se, s tim u skladu, primećuje da je i p i t a n j e p r a v l j e n j a s v i h r e š e n j a n e k e f o r m u l e prevodivo na pitanje da li je izvesna formula prološka teorema.
Pored te značajne činjenice rasuđivanje u prethodnom primeru je koristilo još neka opšta pravila:

> **Pravilo 6:** Elementarnu, već korišćenu granu, treba "kresati"

> **Pravilo 7:** Ako smo pri dokazivanju formule ϕ uz upotrebu članka (ϕ for2 for3 ... fork) stigli do dokaza za fori, gde i>1, i to nam nije pošlo za rukom tada se vraćamo ulevo na fori-1 i tražimo njoj nov dokaz, upošljavajući za to njenu prvu još nekorišćenu granu. Ako takve grane nema, idemo još korak ulevo na fori-2 i za nju tražimo nov dokaz, itd. Konačno, ako se dogodi da se ulevo pomerili sve do ϕ, tj. početka onda tu ϕ-granu napuštamo i tražimo za ϕ novu granu ne bi li smo preko nje dokazali ϕ. Ali, ako ni takve grane nema, onda formula ϕ je prološki nedokaziva.
> Istaknimo da smo u navedenom opisu pretpostavili da se tokom vraćanja ulevo nije naišlo na znak reza /, što je inače poseban slučaj (vid. Pravilo 8).

Primetimo da je to pravilo, uz navedeno ograničenje, do sada prvi opis čuvene procedure backtracking (zvaćemo je i procedura v r a ć a n j e). Ovo je prilika da istaknemo još neke značajne stvari u vezi sa načinom kako Prolog "dodeljuje vrednosti svojim promenljivim".

Napomena 2.2.1. U jednom koraku smo tražili _X tako da (p _X) bude ujednačena sa (p 1), jer tako smo našli jedan dokaz za (p _X). Tačno rečeno, u duhu Napomene 2.1.1 to _ X ima status nepoznate, pa smo za _X tražili vrednost tako da se ta formula (p _X) bukvalno prevede na formulu (p 1), koja je elementarna aksioma. Drugim rečima, imali smo ovu "jednačinu"

(p _X)=(p 1)

čije rešenje je određeno sa _X=1. U stvari, u Prologu se tokom njegovog algoritma često pojavljuju slične "jednačine". Naime, to je uopšte slučaj kad se tokom algoritma pojavi potreba ujednačavanja neke formule ϕ sa glavom ψ izvesnog članka. Tada se, tačno rečeno, pojavljuje rešavanje ove "jednačine" $\phi=\psi$. Sintaksne promenljive formula ϕ i ψ tada imaju status

nepoznatih. Inače za rešavanje takve "jednačine" zadužen je algoritam unifikacije. Kraj Napomene 2.2.1.

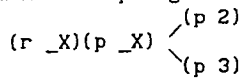
Primer 2.2.1. (nastavak prethodnog). Kako naći sve _X za koje važi (r _X) ?

Odgovor. U skladu sa Pravilom 5 postavljamo ovo pitanje

(*) ?((r _X)(PP _X)FAIL)

Rasuđivanje: Prvo upošljavamo prvu r-granu, tj granu (r _X)(p _X). U vezi sa (p _X) u glavi ćemo držati ovakvu sliku

Tražimo prvo _X za koje je (p _X). Dobijamo _X=1 (koristili smo prvu p-granu). Tako je dokazano (r _X), pri _X=1. Po datom pitanju idemo dalje, tj. "udesno". Dođemo na "dokazivanje" (PP _X), što se svodi na štampanje 1 na ekranu. Opet idemo udesno i dođemo na FAIL, što znači kao da smo stigli do formule koja nije dokazana. U skladu sa Pravilom 7 prvo idemo levo na (PP _X), ali pošto ta formula nema granu, opet idemo levo na (r _X), sledstveno, tražićemo drugi dokaz za (r _X). Značajno je, radi toga, u vezi sa dokazivanjem (r _X) znati dokle smo bili stigli, odnosno gde smo stigli. U tu svrhu može da pomogne slika



Vidi se da smo granu (p 1) "skresali" jer je iskorišćena (Pravilo 6). U skladu sa tim da bismo našli nov dokaz za (r _X), tražimo nov dokaz za (p _X) i dobijamo _X=2. U pitanju (*) idemo udesno na (PP _X) i na ekranu se štampa 2, a dalje opet idemo udesno i naidemo ponovo na FAIL (koje nije tačno). Koristimo Pravilo 7, tj. tražimo nov dokaz za (r _X), odnosno za (p _X). Odgovarajuća pomoćna slika je

(r _X)(p _X) —— (p 3)

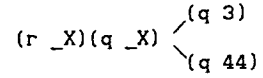
jer granu (p 2), kao korišćenu, smo sklonili (Pravilo 6). Sada vidimo da je (r _X) opet dokazivo pri _X=3. U (*) idemo udesno, tj. prvo se štampa 3 i dalje opet idemo udesno i dođemo na FAIL (koje nije tačno). Sledstveno, u skladu sa Pravilom 7, prvo stignemo do (r _X), tj. do traženja novog dokaza za njega, ali pošto je grana

(r _X)(p _X)

potpuno iskorišćena (jer sve p-grane su potrošene) vršimo pregranjavanje po r, odnosno upošljavamo ovu granu

(r _X)(q _X)

Sada bi odgovarajuća pomoćna slika bila



i nije teško videti, rasuđivanjem sasvim sličnim dosadašnjem, da će se

dalje kao nova rešenja relacije (r _X) pojaviti[17] 3, 44. Sledstveno potrošiće se i poslednja grana za r. I sada kad u pitanju (*) ponovo dođemo na FAIL u vraćanju natrag, kako je to opisano Pravilom 7, vratićemo se do samog kraja sa završnim odgovorom ne , odnosno ? .

Kao što se iz dosadašnjeg izlaganja neposredno vidi jedna od osnovnih crta prološkog mehanizma je da se uopšte pri dokazivanju neke formule ϕ najpre upošljava prva ϕ-grana, ako tako ne uspe, onda naredna itd., naravno pod uslovom da te grane postoje. Potpuno je očigledno da je to insistiranje na redosledu upošljenih članaka korisno iz "vremenskih razloga" -odnosno mnogo duže bi trajalo neko Prologu slično rasuđivanje i pretraživanje , u kome se ne bi tražio utvrđen redosled upotrebe članaka. Ali, s druge strane, prološki mehanizam upravo iz tog razloga ponekad ne može da dokaže neku formulu ϕ, iako je ona logička posledica datog programa (vid. Primer 2.2.2).

Primer 2.2.2. Uočimo ova dva prološka programa

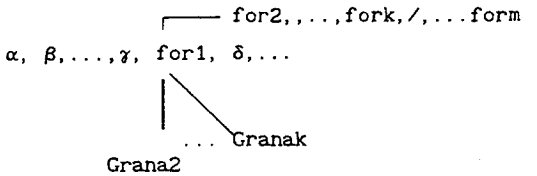
```
(a) ((p _X)(p _X))        (b) ((p 1))
    ((p 1))                   ((p _X)(p _X))
```

Za koje _X je dokazivo (p _X) u svakom od tih programa?

Program (a). Sada (p _X) nije dokazivo ni za jedan _X, jer recimo pitanje ?((p 5)) se, budući da najpre koristimo članak ((p _X)(p _X)) opet svodi na pitanje ?((p 5)), a ovo opet na pitanje ?((p 5)), i tako bez kraja. Znači, iako je (p 1) logička posledica programa (a), prološkim mehanizmom nikako se ne može dokazati (p 1).
Program (b). Ovde je dokazivo (p 1), jer se pri dokazu prvo upošljava članak ((p 1)) koji je elementarna aksioma. Međutim za ostale _X pokušaj dokazivanja (p _X) se produžuje bez kraja.

2.3 P r a v i l o o R E Z U

Do sada nismo govorili o prološkoj "formuli" rez / (odnosno ! u slučaju Edinburgške sintakse). Ona tokom prološkog algoritma ima veoma posebnu ulogu. Da bismo lakše opisali ponašanje formule /, najpre uočimo ovu skicu, odnosno drvo



U skladu sa tom slikom, pretpostavimo da smo pri dokazivanju formule for1 upotrebili navedenu granu for2,..,fork,/,.. ,form i da smo redom dokazali for2,..,fork. Naišavši na znak / preskačemo ga, tj. idemo udesno dalje na na dokaz fork+1,.....Ali, zamislimo da nismo uspeli da dokažemo jednu od njih, što nas nagoni na proceduru vraćanje. Dodatno pretpostavimo, da nas

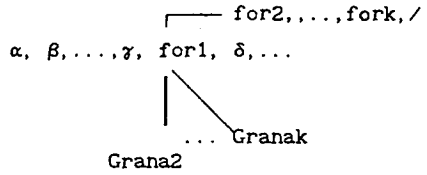
---

17 U stvari, 3 nije novo rešenje, ali, opšti prološki mehanizam ne vrši odgovarajuću proveru "novosti".

je vraćanje dovelo do znaka rez. E sada formula / "ne spava", odnosno evo kako se dalje nastavlja prološki algoritam. Radi lakšeg izražavanja za formulu for1 ćemo reći da je ona pod(/). Tada:

> Formula pod(/) se proglašava netačnom[18], pa sledstveno od tog mesta nastaje nova procedura vraćanje.

Znači, kao što vidite ne dozvoljava se nikakvo pregranjavanje ni po fork, ,...,ni po for2, i konačno ni po for1.
Istaknimo da ima još jedan slučaj pravila reza, koji je prikazan na donjoj slici:



Naime, zamislimo da smo tokom algoritma, pri proceduri vraćanje stigli do traženja novog dokaza[19] za for1, što povlači traženje novog dokaza za /, pošto vraćanjem po drvetu se dolazi na to mesto. E, onda rez / više ne "spava", odnosno kao i do sada skoči se na pod(/), tj. na formulu for1, ona se proglasi za netačnu i od nje nastaje novo vraćanje. Evo sada opšteg opisa pravila reza, ukratko izloženog

> **Pravilo 8:** Formula / je po definiciji tačna, ali ako se desi da se procedurom vraćanje(backtracking) dođe do nje, onda se skoči na mesto pod(/) i odatle se počinje sa novom procedurom vraćanje

Primetimo da se pravilo reza može iskazati i ovim rečima:

(2.3.1) Formula / nema nikakav drugi dokaz, pa ako se pojavi potreba traženja drugog dokaza za /, onda počinje dejstvo pravilo reza, odnosno nastavi se sa vraćanjem od mesta pod(/).

Navodimo neke primere sa upotrebom reza /.

Primer 2.3.1. Dati su programi

```
(a)   ((p 1))              (b)  ((p 0)/)
      ((p 2)/)                   ((p _x)(PP _x)(SUM 1 _xx _x)(p _xx))
      ((p 3))
```

koji sadrže formulu / , tj. rez (cut).

Tada u vezi sa programom (a) imamo ovakve zaključke

> Na svako od pitanja ?((p 1)), ?((p 2)), ?((p 3)) odgovor je da, tj. znak / je bez dejstva. Ali ako postavimo pitanje ?((p _x)(PP _x)FAIL)

---

18 Recimo, ako je podesnije za razmišljanje pretpostavite da smo pod(/), tj. for1 izbacili i zamenili formulom FAIL.

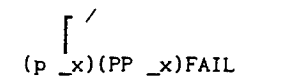
19 Tako, to se moglo desiti jer nam recimo nije uspelo da dokažemo δ.

tj. potražimo sva _x -rešenja relacije (p _x) tada se na ekranu pojave

1 i 2

ali ne i 3. Naime, kad nam prođe dokaz za (p 2) -- u tom koraku znak / ne smeta -- i posle toga dođemo do FAIL-a, on nas tera na vraćanje (backtracking) i kad dođemo do formule (p _x) pomišljamo da se pregranimo. Ali, budući da se na tom mestu nalazi rez / to pregranjavanje je zabranjeno .

Podrobnije rečeno, kad smo iskoristili članak ((p 2)/) imali smo ovo drvo



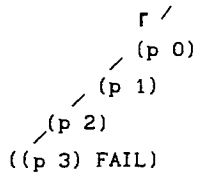
i "preteklo" je da dokažemo formulu / koja je po definiciji tačna. I tada smo se spustili i "dokazali" (PP _x), tj. na ekranu je štampano 2, ali posle toga nas je FAIL poterao nazad i došli smo do (p _x). Tu je sada važno da znamo gde se bio završio poslednji dokaz te formule. Jasno, to je bilo na mestu formule /. Možemo reći da nas čeka nov dokaz za /, što je po Pravilo o rezu nemoguće(vid.(2.3.1)). Elem, sada treba naći pod(/). Sadašnja skica je nešto skraćenija od gornje na kojoj smo objasnili pravilo reza. Formula pod(/) je formula (p _x). Dakle, ta formula je sada proglašena za netačnu i od (p _x) treba da nastane vraćanje. Ali, (p _x) nema ni neku formulu levo ni neku formulu pod sobom. Stoga se algoritam završava sa ne.

Sada u vezi sa programom (b) postavimo ova dva pitanja

?((p 3)) , ?((p 3) FAIL)

Zbog formule (PP _x), pri raspravljanju prvog pitanja na ekranu će se pojaviti 3 2 1 jer kraće rečeno:

(p 3) se preko drugog članka povezuje sa (p 2), ovaj sa (p 1), ovaj sa (p 0) i taj (p 0) se dokazuje pomoću prvog članka. Tu je kraj algoritma i rez / nema dejstva. Medjutim, drugo pitanje ima FAIL, koji će, kako ćemo videti, aktivirati rez. Naime, kraće rečeno imamo ovakvu skicu



koja, ignorišući deo (PP _x), poručuje da se (p 3) dokazuje pomoću (p 2), ovaj pomoću (p 1), ovaj pomoću (p 0) i ovaj pomoću formule /, koja je po definiciji tačna. Ali, kad se nakon tog dokaza "spustimo" dole na (p 3) i dođemo do FAIL, on nas potera nazad i vrati na mesto / . Sada pod(/) je formula (p 0), koja je dakle proglašena netačnom. Pošto za (p 1) nema drugog dokaza, to i ona je netačna. Dalje, isto važi i za (p 2) i (p 3), pa se algoritam završava porukom ? , tj. ne. U trenutku zamizamislimo da prvi članak programa (b) nema znak /. Tada bi se pri odgovaranju na drugo pitanje na ekranu pojavio neograničen niz

3 2 1 0 -1 -2 -3 ....

jer, sada usled odsustva reza kad se stigne na (p 0), pri traženju drugog dokaza, pregranimo se na drugi članak, a onda se dalje redom pojave svi negativni celi brojevi -1, -2, ... i algoritam se ne zaustavlja.

Primer 2.3.2. Uočimo program

```
((a 1))
((a 2))
((a 3))
((b 11))
((b 22))
((b _x)(a _x)/)
((b 33))
((c 111))  ((c 222))
((d _x _y)(a _x)(c _y)/)   ((d 7 77))
((e _x _y)(a _x)/(c _y))   ((e 8 88))
((f _x _y)/(a _x)(c _y))   ((f 9 99))  ((f 100))
```

Tada, pri razmatranju pitanja ?((b _x)(PP _x)FAIL) na ekranu se prvo pojave 11, 22 a zatim 1, koje je prvo rešenje za (a _x) ali zbog / , kad ga uključi FAIL, na ekranu se još samo pojavi ?, tj. drugim rečima zabranjuju se a- i b- pregranjavanja.
Pri razmatranju pitanja ?((d _x _y)(PP _x _y) FAIL) na ekranu se pojavi 1 111 , tj. "kombinacija" prvog a-rešenja sa prvim c-rešenjem, i još zbog FAIL-a pojavi se "neumitni" znak ? . Primetite da je znak / kod članka

((d _x _y)(a _x)(c _y)/)

na njegovom kraju. Elem, kad ga je "uključio" FAIL on je od njega ulevo zabranio pregranjavanje i po c, i po a, i po d. Pri razmatranju pitanja

?((e _x _y)(PP _x _y)FAIL)

na ekranu se redom pojavljuju

1 111 , pa 1 222 i najzad ?

Znači, u članku ((e _x _y)(a _x)/(c _y)) znak rez zabranjuje pregranjavanje po a i po e, ali ne i po c (to i ne može, jer (c _y) je desno od njega), pa su se sledstveno na ekranu pojavila oba c- rešenja 111 i 222.
Pri razmatranju pitanja ?((f _x _y)(PP _x _y)FAIL) na ekranu će se redom pojaviti

1 111, pa 1 222, pa 2 111, pa 2 222, pa 3 111, pa 3 222 i najzad ?

Znači, sada zbog reza nismo smeli da se pregranimo po f, pa elementarnu aksiomu (f 9 99) nismo "posetili". Medjutim, zanimljivo je da je na pitanje ?((f 9 99)) odgovor ?, tj. ne. Razlog je sledeći. Pri raspravljanju tog pitanja prvo se, u skladu sa redosledom i pošto može, upošljava prvi članak prema kome imamo

((f 9 99) / (a 9) (c 99))

Idemo udesno i prvo dođemo do /, koji se sada "preskače", i stižemo do formule (a 9), kojoj treba da proverimo važenje. Ona je netačna, pa sada treba da od njenog mesta upotrebimo proceduru vraćanje. Ali prisustvo reza zabranjuje f- pregranjavanje i konačno zbog FAIL-a algoritam se završava sa odgvorom ? , tj. ne.

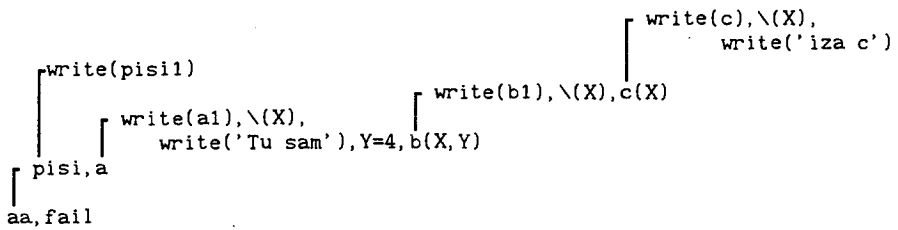
Na kraju postavimo sledeće pitanje ?((f 100)). Sada, to se lako vidi, u jednom koraku se stiže do odgovora da. Naime, sada je moguće korišćenje je-

dino poslednje f- grane, odnosno elementarne aksiome.

Dosadašnjem izlaganju o rezu, dodajemo još nešto, što je inače jedna od posebnosti LPA-prologa. Naime, u njemu pored "običnog" reza postoji i tzv. duboki rez(deep cut). Za njega se koristi oznaka oblika \(Prom), gde je Prom promenljiva, koja istina tokom algoritma može i da promeni vrednost, ali to je van kontrole korisnika. Najpre ga upoznajemo na jednom malom primeru. Uočimo ovaj program

```
(2.3.2)   a:-write(a1),\(X), write('Tu sam '),Y=4,b(X,Y).
          a:-write(a2).
             b(X,Y):-write(b1),\(X),c(X).
             b(X,Y):-write(b2).
          c(X):-write(c),\(X),write('iza c').
          aa:-pisi,a.
             pisi:-write(pisi1).
             pisi:-write(pisi2).
          aa:-write('drugo aa').
```

i postavimo ovo pitanje ?-aa,fail. Da bismo lakše uvideli smisao dubokog reza \(X) napravimo sledeće drvo koje odgovara traženju odgovora na to pitanje



Na drvetu je korak za korakom prikazano kako se računa formula aa. Prvo se koristi prva pisi-grana i na ekranu se ispisuje reč pisi1. Tada se prelazi na a, odnosno na prvu a-granu. Na ekranu se piše a1, znak dubokog reza se preskače, ide se na write('Tu sam') i na ekranu se štampa: Tu sam. Dalje Y dobije vrednost 4 i idemo na računanje formule b(X,Y). Primetimo da je Y "naša" promenljiva, dok X je promenljiva dubokog reza, i nju prenosimo da bi se ti razni \(X) povezali. Dalje se piše b1, opet preskače znak dubog reza i ide na c(X). Taj X slično maločašnjem jedino služi da poveže dva susedna znaka dubog reza. Dokazivanje(računanje) formule c(X) se završava štampanjem slova c i reči 'iza c', jer se duboki rez opet ignoriše. Dokazivanjem c(X) dokazano je i formula aa. U narednom koraku dođemo na fail. On nas potera nazad sve do formule write('iza c'), tražeći da joj nađemo nov dokaz što nije moguće. Otuda odatle opet vraćanje, ali

sada dođemo do znaka dubokog reza, koji deluje ovako. Prvo od jednog znaka dubog reza skočimo na onaj prvi ispod njega i tako radimo dok kog je moguce. Konačno se stiže do znaka \(X) u grani nad a. I sada od tog mesta deluje obično pravilo reza, gore već izloženo. To znači da se traži nov dokaz za pisi, tj. na ekranu se ispiše reč pisi2, posle opet "izpočetka" dokazujemo a, i konačno kad nas fail natera na vraćanje duboki rez će nas doterati do formule pisi, pa pošto za nju više nema drugog dokaza to će se dalje preći na drugu aa-granu, tj. štampaće se 'drugo aa', itd.

2.4 Još neki primeri

Navodimo još nekoliko primera proloških programa napominjući da se razni primeri nalaze u tački Zadaci, I . Takođe kratko govorimo o listama, i navodimo nekoliko primera programa sa njima.

Primer 2.4.1. Ovaj primer je u vezi sa funkcijom faktorijel. Tako

0!=1, 1!=1, 2!=1*2=2, 3!=1*2*3=6, itd.

Dogovorno umesto n! pisaćemo fak(n). Tada jedna matematička definicija faktorijela glasi

```
(*)       fak(0) = 1
          fak(n) = n * fak(n-1)      gde n = 1,2,3,... .
```

Da bismo napravili odgovarajući prološki program najpre ćemo od funkcije fak preći na odgovarajuću relaciju, u oznaci fakt. Definicija te relacije glasi:

(fakt X Y) $\longleftrightarrow$ Y = fak(X).

Imajući to na umu lako se prevođenjem (*) dobija ovaj prološki program[20]

```
(2.4.1)   ((fakt 0 1))
          ((fakt _x _y) (SUM 1 _x1 _x)
                        (fakt _x1 _y1) (TIMES _x _y1 _y))
```

Lako je napraviti LPA i Arity-verziju tog programa. Ona glasi

```
fakt(0,1).
fakt(X,Y):-X1 is X-1, fakt(X1,Y1), Y is X*Y1.
```

Nije teško videti da je dobijeni program sposoban da za dati n=0,1,2,..... izračuna n!, doduše uz praktično ograničenje da n nije neki veliki broj, jer n! brzo raste pa se uskoro pojavljuju veoma veliki brojevi. Recimo,

14! = 87 178 291 200

Na primer, izračunaćemo 3!. U tu svrhu postavljamo pitanje

?((fakt 3 _x))

sa ciljem da dokažemo formulu[21] (fakt 3 _x) sa nekim _x i da, za sebe, kao rezultat uzmemo to _x, tj. njegovu vrednost. Tokom algoritma koristićemo "jednačinsko" pisanje, sa smislom koji objašnjavamo

```
(fakt 3 _x) = (fakt 2 _y),x = 3 *_y   Koristili smo drugi članak; deo sa
                SUM formulom nismo pisali, jer smisao tog dela je prelaz od
                3 na 2, što smo konačno i učinili. Dalje, TIMES formulu smo
                napisali kao uobičajenu algebarsku jednakost. Smisao dobije-
                nog je da nas sada čeka dokaz formule (fakt 2 _y) sa nekim y
            = (fakt 1 _z),_y= 2* _z,_x=3*_y  Koristili smo drugi članak i
```

[20] Drugi članak poručuje: fakt _x je _y, ukoliko fakt _x1, tj. fakt od _x-1 iznosi _y1, i uz to _y je _x*_y1. To je prevod druge jednakosti u (*).

[21] Drugim rečima na tom mestu sintaksna promenljiva _x ima status nepoznate, čiju vrednost tražimo (vid. **Napomene 2.1.1 i 2.2.1**).

i obavili smo umetanje. Znači sada nas čeka dokaz te konjunkcije od tri člana.

= (fakt 0 _u), _z=1*_u, _y=2*_z, _x=3*_y

= _u=1, _z=1*_u, _y=2*_z, _x=3*_y Korišćen je prvi članak, koji je elementarna aksioma.

Stigli smo do konjunkcije "malih" jednačina[22], koje se lako rešavaju obavljanjem nekoliko zamena. Tako se konačno dobija:

_x=6, što je traženi rezultat za 3!

Kao što se vidi, problem se sveo na rešavanje više "malih" jednačina. Sada pomenimo, da je Prolog kao jezik sposoban da zapamti i reši takve skupove ve jednačina (naravno njihov broj ne može biti neograničen).

Istaknimo da ,recimo indukcijom po n, nije teško dokazati da je program (2.4.1) sposoban da za dati n=0,1,2,... izračuna n!.

Postavimo sada ovo pitanje

?((fakt 0 2))

na koje bi trebalo očekivati odgovor ne. Opet koristeći se jednačinskim pisanjem imamo

(fakt 0 2) = (fakt -1 _x),2=0*_x Prvi članak nismo mogli koristiti, pa smo prešli na drugi. SUM- deo je u "igru" uveo -1. Smisao bi bio: prvo da nadjemo _x, tj. (-1)!, pa onda da proverimo da li važi jednakost 2=0*_x. Naravno, mi odmah vidimo da takva jednakost ne može važiti, pa bi "po našem" završetak algoritma bio ne. Ali, Prolog se drži strategije prvog levog člana, pa stoga to ne može videti sve dok prvo ne "izračuna" formulu (fakt -1 _x).

= (fakt -2 _y),_x=(-1)*y,2=0*_x. Opet smo koristili drugi članak i u "igru" je ušao -2. U stvari, dalje će se redom uključivati -3,-4,-5,... i algoritam uopšte neće stati.

I tako, program (2.4.1) nije u stanju da proveri tačnost formule (fakt 0 2) Uopšte, program (2.4.1) nije u stanju da proveri tačnost ma koje formule oblika (fakt p q) ukoliko su brojevi p! i q različiti.

Medutim, taj se nedostatak programa (2.4.1) može otkloniti. Naime, dosta je da se drugi članak u (2.4.1) zameni ovim člankom

```
((fakt _x _y)
        (LESS 0 _x)(SUM 1 _x1 _x)(fakt _x1 _y1)(TIMES _x _y1 _y))
```

Pomenimo, da u slučaju Edinburgške sintakse umesto dela (LESS 0 _x) treba da stoji deo X > 0.

Primer 2.4.2. Prološki iskazati ovaj algoritam:

Napisati poruku: Dajte dva broja x,y.
Uneti x,y
Štampati njihov zbir

Rešenje. Prvo, može se dati rešenje, koje se sastoji iz direktnog pitanja.

[22] To su jednačine sa nepoznatim _x,_y,_z,_u.

Evo Micro-prolog verzije:

```
?((PP Dajte dva broja x,y)(R _x)(R _y)
        (SUM _x _y _z)(PP Njihov zbir je _z))
```

pri čemu sve, uključujući i ?, kucamo mi.

Inače, tu je korišćena jedna od osnovnih "input" relacija Micro-prologa, odnosno R (skraćenica od read). U LPA i Arity se umesto (R _X) piše read(X).

LPA i Arity verzija glasi:

```
?-  write('Dajte dva broja x,y '),nl,
        read(X),read(Y),Z is X+Y,write('Njihov zbir je '),write(Z).
```

pri čemu znak pitanja ? ne kucamo mi, on je prisutan kao znak jezika. Tu je, inače, korišćena osnovna "output" relacija write, kao i nl -za štampanje novog reda.

Medutim, nije teško da se umesto direktnih pitanja naprave i programi. Recimo u prvom slučaju napravimo ovaj članak

```
((upaljac)(PP Dajte dva broja x,y)(R _x)(R _y)
                        (SUM _x _y _z)(PP Njihov zbir je _z))
```

gde je upaljac pomoćna nularna relacija. Namerno smo je nazvali upaljac jer taj program se "uključuje","pali" na ovaj način ?((upaljac)). Naravno, ista ideja je moguća i u drugoj verziji. Program je odredjen člankom

```
upaljac:-write('Dajte dva broja x,y '),nl,read(X),read(Y),
                Z is X+Y,write('Njihov zbir je '),write(Z).
```

Primer 2.4.3. (nastavak prethodnog). Jedna mana navedenog rešenja je što nije predvidena objava greške ako neka od input- promenljivih nije broj. Otkloniti taj nedostatak.

Rešenje. Zamisao je ova:

Prvo, štampa se početna poruka
Drugo, "učitaju se" x,y
Treće, idemo na odvojeno mesto i tamo proverimo "brojnost" od x,y javimo grešku ako treba, a inače izračunamo im zbir i štampamo ga.

U tom opisu smo rekli "idemo na odvojeno mesto .....", da bismo izazvali pomisao neke procedure, a upravo to nam je i bio cilj. Medutim, pri odlaženju na proceduru moramo "sa sobom" poneti učitane x,y. Jedan program (Micro-prolog verzija) glasi

```
((upaljac)(PP Daj dva broja )
                (R _x)(R _y)(uradi _x _y))
((uradi _x _y)(NOT NUM _x)(PP Greska))
((uradi _x _y)(NOT NUM _y)(PP Greska))
((uradi _x _y)(SUM _x _y _z)(PP Njihov zbir je _z))
```

Istaknimo da je uvedena relacija uradi i ona upravo ima ulogu pomenute procedure. Primetimo da ćemo i kasnije koristiti sličnu ideju -zvaćemo je "pamti-ideja", i umesto imena 'uradi' često ćemo koristiti imena kao 'pamti','Zap' i sl.

Dalje, u tom programu, NUM je, u prošloj tački već opisana, osnovna relacija Micro-prologa, za koju kratko rečeno imamo: (NUM _x) znači vrednost od _x je broj.
I najzad NOT je opšta relacija negacije. Neka je (rel a1 a2 .....an) ma

koja formula. Tada, to sada uzimamo kao pravilo, njena negacija se ovako piše (NOT rel a1 a2 ...an).
Sličan je program i u LPA, odnosno ARITY prologu:

```
upaljac:-write('Daj dva broja '),
                              nl,read(X),read(Y),uradi(X,Y).
uradi(X,Y):-not(number(X)),write('Greska').
uradi(X,Y):-not(number(Y)),write('Greska').
uradi(X,Y):-Z is X+Y,write('Njihov zbir je '),write(Z).
```

a sada umesto NOT i NUM redom stoje not i number.
Evo kako se relacija not definiše u LPA i Arity prolozima:

```
(2.4.2)      not(X):-X,!,fail.
             not(X).
```

Da bismo uvideli smisao uočimo ovaj jednočlani program:

```
f(1).
```

Tada na pitanje ?- not(f(1)). imamo ovo rasuđivanje:

not(f(1)) je ujednačivo sa glavom prvog članka u (2.4.2), što daje zamenu X-->f(1). Dalje, idemo udesno na formulu X, tj. u skladu sa tom zamenom, na f(1).Prema datom programu ta formula je tačna. Idemo udesno, dođemo do znaka reza !, preskočimo ga i dođemo do fail, koja je netačna. Elem,nastaje vraćanje. Ali, znak ! to zabranjuje. Znači drugi članak iz (2.4.2) je sada nedostiživ. Krajnji odgovor je: not(f(1)) je netačna.

Međutim, na pitanje ?-not(f(2)). imamo rasuđivanje:

Prvo primenimo prvi članak , i kad dođemo do X to je onda f(2). Pošto f(2) je netačna formula, to u odnosu na to mesto X nastaje vraćanje, kojim stižemo do drugog članka u (2.4.2).Na osnovu njega,kao elementarne aksiome zaključujemo da je tačna formula not(f(2)).

Primetimo, da u definiciji (2.4.2) se pojavljuje kombinacija !, fail koja kad stupi u dejstvo, slobodnije rečeno, proizvodi "potpunu" negaciju, ne dozvoljavajući traženje nekog drugog rešenja za deo ispred njih.

Primer 2.4.4. Prološki izraziti ovaj algoritam

Prvo, na ekranu se pojavi poruka: Ovo je program za ilustraciju ideje repeat
Drugo, na ekranu se pojavi poruka Daj
Treće, učita se X
Četvrto, ispiše se X
Peto, ide se na korak označen sa Drugo.

Rešenje.Evo prvo rešenja upotrebom osnovne relacije repeat (u LPA i Arity-prologu):

```
upal:- write('Ovo je program za ilustracijuideje repeat '),nl,
       repeat,write('Daj '),read(X),write(X),nl,fail.
```

Taj se program uključuje pitanjem ?- upal. Tada, prvo se pojavi poruka Ovo je program za ilustraciju ideje repeat. Dalje, formula (što ćemo ubrzo objasniti) repeat se preskoči, pa se štampa reč Daj , učita se X, štampa se, i konačno dođe se do fail, koje tera nazad. Nijedna od formula sve do repeat nema drugog dokaza, pa tako vraćanjem dođemo do formule repeat. Ali ta formula ,u skladu sa svojom definicijom, vraća nas nazad, pa sledstveno opet ponavljamo deo od repeat-a do fail-a ,i tako u nedogled. To zanimljivo svojstvo formule repeat sledi iz ove njene definicije (koja je inače ugrađena u prologe LPA i Arity)

```
repeat.
repeat:-repeat.
```

Upravo kad smo fail-om bili doterani sve do repeat-a, onda smo se po njemu pregranili (prešli na drugi repeat -članak), zatim prešli na prvi i tako ponovo "dokazali" repeat. Iza toga smo se opet zaputili udesno, itd.

Micro-prolog nema ugrađen repeat, ali taj se nedostatak lako otklanja pozajmljivanjem prethodne definicije repeat-a.Tako,jedan Micro-prološki program navedenog zadatka glasi

```
((upal)(PP Ovo je program za ilustraciju ideje repeat)
                    (repeat)(PP Daj )(R _x)(PP_x) FAIL)
((repeat))
((repeat)(repeat))
```

U vezi sa izloženim primerom istaknimo da se za takvo dejstvo repeat formule, mora uvek, kao i u tom primeru, imati repeat-fail kombinacija,i uz to na putu od fail-a do repeat-a sve formule moraju biti "jednogranske". U vezi sa idejom repeat dodajmo i ovo:

(2.4.3) Da bi se repeat uključio fail ne mora bukvalno učestvovati već umesto njega može biti neka netačna formula.

Evo jednog lepog primera sa tom idejom. Prološki program:

```
ajde:-
   write('Daj redom reči za ispisivanje; za prekid kucati dosta'),
   repeat,
   read(X),
   write(X),
   X=dosta,
   write('Kraj').
```

Na pitanje ?-ajde pruža mogućnost upisa i ispisa željenih reči. Prekid se ostvaruje kucanjem reči dosta, i tada se na ekranu štampa reč Kraj.Kao što vidite iz njegovog sklopa, kad god učitana reč X nije jednaka reči 'dosta' nastaje teranje nazad do repeata i on stupa u dejstvo.

A sada pre navođenja daljih primera, dodajemo nekoliko[23] opštih reči o listama, koje smo u stvari i do sada stalno koristili. Kratko rečeno, lista može biti

ili prazna ,oznaka () u Lisp-sintaksi,odnosno [] u Edinburgškoj
ili neprazna koja je oblika (glava|rep) (u Lisp- sintaksi)
odnosno [glava|rep] u Edinburgškoj sintaksi.

Tu,znak | je tzv."list constructor", tj. znak za građenje lista.

Recimo,kod liste (1 2 3), odnosno [1,2,3] glava je 1,a rep je lista (2 3), odnosno lista [2,3]. Uopšte, po definiciji

Lista oblika (a1 a2 ... an), tj. oblika [a1,a2,...,an]
ima glavu a1, a rep je lista (a2 a3 ... an), odnosno [a2,a3,...,an].

Drugim rečima važi "identitet"

---

[23] Jer, recimo, svaki program u Micro-prologu je napisan na jeziku lista.

(a1 a2 ... an) = (a1|(a2 ... n)),
tj. [a1,a2,...,an] = [a1|[a2,...,an]]

U vezi sa listama iskažimo ovo opšte zapažanje. U Lisp-u, a posle toga i u Prologu osnovni koordinatni sistem mišljenja nije brojevan, kao za mnoge druge jezike, već sistem lista, odnosno drveta. Sledstveno, Lisp, a u novije vreme i Prolog su glavni jezici oblasti tzv. Veštačke inteligencije.

Primer 2.4.5. Napraviti proloski program koji je sposoban da odgovori na pitanje oblika : da li x jeste, ili nije element od y.

Rešenje. Recimo, 1 je element svake od lista (1 2 3), (2 Pera (3 4) 1), ali nije element nijedne od ovih (), (2 3),((2 3) (1 4) 5). Neka dogovorno formula (elem _x _L) odgovara relaciji: _x je element od _L. Tada, možemo, prvo ovako razmišljati

_x je element od _L ,ukoliko je on njen prvi element, njena glava, tj. ukoliko je _L oblika (_x|_y) sa nekim _y.

To zapažanje je zapisivo ovim člankom

(Δ 1) ((elem _x (_x|_y)))

Međutim, pored opisanog slučaja, može se dogoditi da je _x član od _L, ali nije njen prvi član. To logički znači da je _x član njenog repa. Znači, _x je član od _L, ukoliko je _x član njenog repa. To je zapisivo ovim člankom

(Δ 2) ((elem _x (_u|_v)) (elem_x _v))

Dva članka (Δ 1) i (Δ 2) definišu jedan proloski program P. Naravno, neposredno se program P iskazuje u Edinburgškoj sintaksi:

```
elem(X,[X|_]).
elem(X,[_|V]):-elem(X,V).
```

Primetite da smo umesto elem(X,[X|Y]). pisali elem(X,[X|_]). odnosno umesto _Y, budući da se u članku javlja samo jedanput, upotrebili smo crtici _; to je tzv. nema promenljiva. Slično smo uradili i kod drugog članka, gde nema promenljiva stoji umesto U. To je jedna posebnost Edinburgške sintakse.

Nije teško videti da taj program P rešava pitanje postavljeno u zadatku. Primera radi, razmotrimo pitanje

?((elem 1 (2 3 1 5)))

I ovde ćemo u raspravljanju to pitanja koristiti "jednačinsko" pisanje. Tako imamo:

(elem 1 (2 3 1 5))=(elem 1 (3 1 5)), budući da formula (elem 1 (2 3 1 5)) nije ujednačiva sa glavom prvom članka, tj. sa (elem _x (_x|_y)), jer ako je zapišemo u obliku (elem 1 (2|(3 1 5))), onda pokušaj unifikacije dovodi do protivrečnih zamena: _x -->1, _x -->2. Međutim, formula (elem 1 (2 3 1 5)) jeste ujednačiva sa formulom (elem _x (_u|_v)) i tada se dolazi do zamena _x -->1, _u -->2, _v -->(3 1 5). Znači, drugi članak programa je primenljiv, što smo i iskoristili.

=(elem 1 (1 5)) Opet, jer smo morali, koristili smo drugi članak.

= da, što odmah sledi po prvom članku -elementarnoj aksiomi.

U stvari, već pomoću tog primera se uvida da je program P sposoban da odgovori na pitanje oblika: Da li je x član od y, ukoliko je x zaista član. Razlog je što se tokom algoritma, pitanje *da li je x član od y* uvek prevona novo pitanje *da li je x član repa od y*, i slično dalje, dok se ne dode do repa čiji prvi član jeste x, i tada se algoritam završava sa da. Međutim, nije teško videti da P , kad treba, može dati i odgovor ne. Primera radi postavimo pitanje ?((elem 1 (2 3))). Tada imamo ovaj "proloski račun"

(elem 1 (2 3)) = (elem 1 (3)) ,jer moramo koristiti drugi članak

= (elem 1 ())

= ne, jer sada ne možemo koristiti nijedan članak iz P.

Znači, u slučaju kad x nije član od y, to se pitanje korak za korakom prevede na pitanje da li je x član prazne liste, i odgovor je naravno ne.

Međutim, zanimljivo je da je program P sposoban i da nađe sve članove zadate liste. Primera radi, potražimo sve članove liste (1 2). U tu svrhu, postavljamo ovo pitanje

?((elem _x (1 2))(PP _x)FAIL)

Imamo ovo prološko rasuđivanje[24] :

Prvo dokazujemo (elem _x (1 2)), sa nekim _x. Primenom prvog članka to odmah uspe i dobije se zamena _x -->1. Dalje, dokazujemo (PP _x) tj. na ekranu se štampa 1 i zatim dodemo do formule FAIL, koja je netačna. Stoga, nastaje vraćanje, i stigne se do formule (elem _x (1 2)) tražeći joj nov dokaz. Setimo se da smo bili koristili prvi članak. Zato sad koristimo drugi članak, prema kome (elem _x (1 2)) se zamenjuje ovom formulom (elem _x (2)), koju dalje treba dokazati (jer to će naravno onda biti dokaz i za (elem _x (1 2))). Jedan dokaz formule (elem _x (2)) se dobija pomoću prvog članka, što daje zamenu _x -->2. I tako, ponovo dokazavši (elem _x (1 2)) u postavljenom pitanju idemo na deo (PP _x), tj. sada se štampa 2, i opet dođemo do FAIL, koje kao netačno opet tera nazad. Ali, sada je situacija malo složenija. Naime, moramo se setiti šta smo poslednje bili dokazali pri dokazivanju formule (elem _x (1 2)). To je bio dokaz formule (elem _x (2)), za koji smo bili upotrebili prvi članak. Elem, tu se sada vratimo i tražimo nov dokaz te formule. U tu svrhu primenimo drugi članak i tako se sada pojavi traženje dokaza ove formule (elem _x ()). Međutim, ta formula je očigledno netačna, i to znači da će se čitav algoritam završiti sa ne.

U vezi sa izloženim rešenjem postavljenog zadatka navodimo još sledeće. U Micro-prologu postoji ugradjena relacija elem, u oznaci ON, a slično važi za LPA -prolog, gde se umesto ON koristi oznaka on.
Čitaoca upućujemo i na razne druge programe sa listama, izložene u tački Zadaci, I. Pomenimo, da je za mnoge od njih karakteristična rekurzija (poneje bolje reći spust) po nekom od argumenata. Recimo, u problemu traženja

[24] Primetite da sada ne pribegavamo "jednačinskom pisanju". Razlog: zbog procedure vraćanje koja će se usput pojaviti, takvo pisanje nije zgodno.

preseka[25] dve liste možemo rekurzivno misliti po prvoj listi. Zaista, neka zapis (pres _x _y _z) znači: _z je presek _x i _y. Tada, odmah imamo:

((pres () _y ()))

Dalje, ako je _x oblika (_a|_b) onda imamo dva slučaja:

(i) _a je element od _y.
(ii) _a nije element od _y.

Slučaju (i) odgovara članak

((pres (_a|_b)_y (_a|_z))(ON _a _y)(pres _b _y _z))
(Tj. ako _a je član od _y onda treba tražiti presek repa prve liste sa drugom i na taj presek dodati _a)

a slučaju (ii) članak

((pres (_a|_b) _y _z)(pres _b _y _z))
(Slično kao prethodno, ali sada na presek repa prve liste sa drugom se ne dodaje glava prve liste)

Napomena 2.4.1. Može se reći da smo u prethodnom programu jednu funkciju, odnosno presek dve liste $X \cap Y$, definisali na proloski način, tj. preko odgovarajuće relacije pres. Recimo, shodno tome, na pitanje

?((pres (1 2 3) (5 1 2 4) _x)(PP _x))

na ekranu će se ispravno pojaviti lista (1 2). Međutim, ako postavimo pitanje

?((pres (1 2 3) (5 1 2 4) _x)(PP _x)FAIL)

što bi odgovaralo traženju "drugih rešenja" pojaviće se još neke liste koje sa problemom nemaju nikakve veze. Kratko rečeno, navedeni program u stvari nije dobar, jer ne definiše jednu funkciju. Strogo rečeno on uopšte ne uzima u obzir svojstvo jednoznačnosti funkcije:

Za date liste _x, _y treba da postoji tačno jedna lista _z, tako da je tačna relacija (pres _x _y _z)

Međutim, taj nedostatak se može otkloniti kako sledi. Naime, uz prethodne pres- članke dodajmo još ovaj

(Δ) ((presek _x _y _z)(pres _x _y _z)/)

kojim se uvodi dodatna relacija presek. Sada nije teško videti da je upravo ta relacija rešenje postavljenog problema, jer:

prvo, za date _x, _y na pitanje ?((presek _x _y _z)(PP _z)) će se, kao i u prehodnom programu, pojaviti _z, tj. presek $_x \cap _y$. Ali,
drugo, za date _x, _y na pitanje ?((presek _x _y _z)(PP _z)FAIL) blagodareći rezu / neće se pojaviti nikakva "druga rešenja".

Značajno je istaći da je izložena ideja veoma opšta. Naime, ako nam je namera da proloski definišemo neku "funkcijsku" relaciju

(rel X1 X2 ...Xn Y)

onda se možemo držati ovog plana:

[25] Presek dve liste je lista svih njihovih zajedničkih članova.

Prvo napravimo proloske članke kojima se definiše relacija *relac* koja odgovara toj traženoj relaciji *rel* u smislu da je za date X1, X2,...,Xn dokaziva formula (relac X1 X2 ... Xn Y) sa nekim Y i da su ti članci tako napisani da smo sigurni da važi i formula

(rel X1 X2 ...Xn Y).

Tada, tražena relacija *rel* se , slično sa (Δ), može uvesti ovim člankom[26]

(ΔΔ) ((rel X1 X2 ... Xn Y)(relac X1 X2 ...Xn Y)/)

koji bi se mogao ovako čitati

X1, X2,..., Xn, Y su u relaciji *rel* upravo ako oni jesu u relaciji *relac* i uz to za date X1, X2,..., Xn taj Y je prvi takav da važi *relac* od X1, X2 ..., Xn, Y.

Moglo bi se pomisliti, a zašto uopšte obezbeđivati svojstvo jednoznačnosti? Razlog je jednostavan i prirodan. Zamislimo da je for neka funkcijska relacija koja tokom nekog proloskog algoritma se pojavi na jednoj i grani

$\alpha, \ldots, \beta, \text{for}, \gamma, \ldots, \delta$

i da nam se dokazavši for desilo kasnije da neki od njenih "desnjaka" $\gamma, \ldots, \delta$ "otkaže" , pa smo primorani na proceduru vraćanje. I sada zamislimo da smo tako stigli i do formule for. Znači, sada bi nas čekalo traženje njenog drugog dokaza, traženje drugih rešenja. I tada da nismo obezbedili jedinstvenost[27], mogle bi da nastanu razne greške. Naravno, nije tačno da se funkcijska jedinstvenost postiže jedino upotrebom formule oblika (ΔΔ). Recimo, u programu za faktorijel (vid. Primer 2.4.1) taj cilj je postignut dopunskim LESS-ograničenjem[28].

Međutim, u vezi sa funkcijskim relacijama ima još problema. Naime, dodavanjem članka oblika (ΔΔ) obezbedili smo se da se za date X1, X2,..., Xn pomoću njega izračunava tačno jedna vrednost funkcije, tj. tačno jedno Y. Ali, može se desiti da su svi X1,..,Xn,Y zadani i da je onda pitanje da li važi (rel X1 X2 ... Xn Y). U tom slučaju treba računati konjunkciju

(rel X1 X2 ...Xn Z)(EQ Z Y)

čiji je smisao : prvo izračunati vrednost funkcije u tački X1,...,Xn, tj. naći Z, a onda njega porediti sa Y. **Kraj Napomene 2.4.2.**

Sada prelazimo na naredni zadatak u čijem konačnom rešenju će se prirodno upotrebiti liste.

Primer 2.4.6. Pretpostavimo da je *rel* izvesna data binarna relacija, recimo

[26] Primetimo da Micro-prolog ima ugrađenu relaciju ! (ne treba brkati sa znakom reza u Edinburgškoj sintaksi), koja upravo služi istoj svrsi, odnosno relacija *rel* pomoće nje se uvodi ovako
((rel X1 X2 ... Xn Y)(! relac X1 X2 ...Xn Y))

[27] Umesto "jedinstvenost" može se prirodno reći i "jednogranost".

[28] Pomenimo da se u nekim knjigama nalaze programi u kojima se zahtev jedinstvenosti postiže što se ponekad znak reza , slobodnije rečeno, na volseban način koristi u raznim, i to u nekim, *nedovoljno objašnjeno i kojim*, člancima relacije *relac*. Od svakog takvog načina efikasniji je način pomoću definicije tipa (ΔΔ).

zadana ovim člancima

```
(Δ 1)   ((rel 1 2))    ((rel 1 3))
        ((rel 2 4))    ((rel 2 5))
        ((rel 3 5))    ((rel 3 7))    ((rel 3 8))
        ((rel 5 6))    ((rel 5 3))
        ((rel 6 13))   ((rel 6 14))
```

Označimo sa r najmanju tranzitivnu relaciju koja sadrži relaciju rel. Kako proloski raspraviti da li su ma koja dva data x,y u toj relaciji r;tj.drugim rečima, da li su x,y korišćenjem relacije rel tranzitivno povezivi. Recimo, 1 je povezivo sa 13 ,na primer ovako:

od 1 na 2; od 2 na 5;
od 5 na 6; od 6 na 13.

Istaknimo,da je taj mali zadatak pripadnik jedne veoma značajne klase problema : "utranzitivljenja " neke date relacije rel.

Rasudjivanje:Odmah je jasno, da u skladu sa vezom izmedju relacija rel i r moraju da važe ove dve implikacije

```
(rel x y)   ==> (r x y)
(r x y) i (r y z) ==> (r x z)
```

gde su x,y,z promenljive[29]. U skladu sa tim prirodno je napraviti ova dva članka

```
(Δ 2)      ((r _x _y)(rel _x _y))
           ((r _x _z) (r _x _y) (r _y _z))
```

Članci (Δ 1) (Δ 2) zajedno definišu jedan proloski program P1. Da li je taj program dobar za rešavanje postavljenog pitanja ? Nije teško videti da je odgovor odrečan. U tu svrhu postavimo ova dva pitanja:

```
?((r 1 4)),        ?((r 4 6))
```

Evo za prvo pitanje nešto skraćenijeg proloskog rasudivanja (uz korišćenje jednačinskog pisanja, koje ovde pomaže samo unekoliko,odnosno ne previše):

```
(r 1 4)=(r 1 _y)(r _y 4)    Koristili smo drugi članak iz (Δ 2).Tu je _y
                            neki _y, tj. nepoznata.
       = (r 2 4)            Formulu (r 1 _y) smo dokazali pomoću prvog
                            članka i _y je 2.
       = da,                na osnovu prvog članka iz (Δ 2) i trećeg iz (Δ 1).
```

Znači,program P1 je sposoban da raspravi prvo pitanje.

Sada razmotrimo drugo. U vezi sa njim imamo ovo rasudivanje:

```
(r 4 6)=(r 4 _y)(r _y 6)   jer moramo koristiti drugi članak iz (Δ2).Znači računanje formule (r 4 6), traži računanje
                           formule (r 4 _y), sa nekim nepoznatim _y
       =(r 4 _y1)(r _y1 _y)(r _y 6)
                           Kao što vidite, opet na samom početku nas čeka
                           računanje vrednosti formule oblika (r 4 ...).
```

[29] Upravo kad je rečeno da su x,y,z promenljive smisao je da ih smemo zamenjivati ma kojim vrednostima.

U stvari,dalje će se kao prve stalno pojavljivati takve formule sa 4 kao prvim argumentom, pa sledstveno algoritam nikad neće stati.

Znači,program P1 nije u stanju da odgovori na drugo pitanje.Nije teško videti da smo mu postavili "nezgodno" pitanje: da traži (r 4 6), a u stvari, nema čak nijednog _y tako da važi (r 4 _y) - što naravno Prolog "ne vidi", jer on "rasuđuje" na svoj način na osnovu programa P1.

Sada ćemo se potruditi da poboljšamo program P1. U tu svrhu drugi članak zamenjujemo ovim

```
((r _x _z)(rel _x _y)(r _y _z))
```

koji poručuje

Tačka _x je tranzitivno poveziva sa tačkom _z ukoliko je tačka _x "susedna",tj. u relaciji rel, sa nekom tačkom _y, a ta tačka je tranzitivno poveziva sa tačkom _z.

Očevidno je da je taj članak mnogo bolji od starog, jer sada se otklanja mogućnost bezuspešnog traženja vrednosti formule oblika (r _x _y) pri čemu _x nema nijednog "suseda", kao što je prethodno bio slučaj sa formulom (r 4 6). Označimo sada sa P2 program koji nastaje kad se u programu P1 deo (Δ 2) zameni ovim delom

```
(Δ 3)      ((r _x _y)(rel _x _y))
           ((r _x _z)(rel _x _y)(r _y _z))
```

Već na prvi pogled se vidi da je program P2 znatno poboljšanje programa P1.Primera radi,postavimo pitanje ?((r 2 8)). Tada jednačinskim pisanjem koje je sada u odnosu na prethodne primere nešto poboljšano, jer ćemo sa + označavati logički veznik ili, odnosno mogućnost pregranjavanja, imamo rasuđivanje

```
(r 2 8)=(r 4 8) + (r 5 8)  Naime,za traženje (r 2 8) mora se uposliti drugi članak. Tako, dođemo prvo do (rel 2 _y), i jasno je da ima dva
                           takva _y : 4 i 5. Znači, prvo nas čeka rasprava (r 4 8), a ako to
                           ne uspe, tj. dobijemo rezultat ne, onda nam još preostaje rasprava (r 5 8). S tim u vezi (r 2 8) je u "ili" vezi sa (r 4 8) i
                           (r 5 8). Gornja jednakost upravo to iskazuje uz rečeni dogovor da
                           + zamenjuje veznik ili.
       = (r 5 8), jer (r 4 8) netačno. Preostala nam je tačno jedna grana.
       = (r 6 8) + (r 3 8), jer 5 ima dva suseda: 6 i 3.
       = (r 13 8) + (r 14 8) + (r 3 8)
         6 ima dva suseda 13,14, pa su se u igru uključila još dva "sabirka".Dalje, u duhu Prologa "računamo" najpre prvi sabirak.
       = (r 14 8) + (r 3 8), jer (r 13 8) je netačno.
       = (r 3 8), jer (r 14 8) je netačno.
       = da, jer važi (rel 3 8).
```

Znači, završni odgovor je da. Moglo bi se pomisliti da je program P2 potpuno dobar za raspravljanje opšteg pitanja opisanog u razmatranom primeru. Da bismo uvideli da to ipak tako nije zamislimo da se (Δ 1) dopuni ovim novim člancima

```
(Δ 4)      ((rel a b))    ((rel b c))    ((rel c a))
```

(a,b,c su konstante)

i postavimo pitanje ?((r a d)). Tada imamo:

(r a d)=(r b d), jer primenljiv je samo drugi članak iz (Δ 3), i b je jedini sused za a.

=(r c d)

=(r a d) Buduci da opet imamo računanje formule (r a d), jasno je da algoritam nikad nece stati.

Znači, program P2 u slučaju prisustva članaka (Δ4) nije u stanju da odgovori na pitanjr ?((r a d)). Glavni razlog je što se tokom algoritma pojavi ponovno računanje iste formule[30], što vodi u beskonačnu petlju.

I sada se prirodno rađa misao:

> Kad bismo nekako zapamtili koje smo sve formule računali tako da odustanemo kad se desi ponavljanje ?

Istaknimo da je ta ideja proloski ostvariva na sledeci način, odnosno sledecim programom P3 :

```
((r  _A  _x  _y)(rel  _x  _y))
((r  _A  _x  _z)
                (NOT  ON  _x   _A) (rel _x _y)(r (_x |_A) _y _z))
((r _x _y)(r () _x _y))
```

U tom programu su navedeni samo r-članci, jer rel-članci, kao (Δ 1), zavise od konkretno zadane relacije *rel* koja se naravno može po volji zadati.

U programu, tako ćemo reći, je prisutna ideja pomoćne liste za pamćenje, ideja "torbe". U navedenim člancima torba je označena sa _A. Evo kratkog opisa rada tog programa. Pretpostavimo, da uz prisustvo (Δ 1) članaka želimo saznati da li je, na primer, 1 tranzitivno povezivo sa 8. U tu svrhu postavimo pitanje

?((r 1 8))

Na osnovu programa P3 to se pitanje preobrati u pitanje da li važi formula (r () 1 8). Znači, "u igru" uvodimo torbu, koja je na početku prazna. Dalje, program radi slično kao P2, ali u usputno pojavljenim formulama koje su oblika (r _A _x _z) uvek proverava da li je _x član torbe _A i ako nije, u stvari radi kao program P2, i pride taj _x stavlja u novu torbu koja je onda (_x|_A); a ako _x jeste član torbe _A odstupa od delanja kao program P2. To praktično znači da je izbegnuta mogućnost beskonačne petlje, odnosno nije teško zaključiti da je za razliku od programa P1, P2 program P3 uopšte potpuno dobar za raspravljanje tranzitivne povezivosti neka data dva elementa x, y uz unapred datu osnovnu relaciju *rel*.

Zanimljivo je da uprkos navedenoj manjkavosti program P2, koji ne koristi ideju torbe, je dobar u slučaju mnogih relacija *rel*, odnosno upravo onih

---

30 Osnovni razlog tome je što u vezi sa relacijom rel imamo kružnu petlju a--b--c--a. Ta petlja ima 3 člana, broj 3 nije važan, mogao je biti i ma koji prirodan broj. Recimo, isti problemi bi se pojavili i kada bismo umesto (Δ4) imali samo ovaj članak ((rel a a)) i uz to postavili pitanje ?((r a b)).

koji "ne sadrze kružne petlje". Takve su, recimo, linearne(totalne) relacije, kod kojih za ma koja dva objekta x, y (iz odgovarajuceg osnovnog skupa) važi (rel x y) ili (rel y x). U mislima takvoj relaciji odgovara "lanac" elemenata. Recimo, takva je ova relacija *rel*

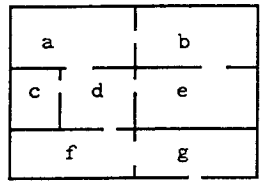
((rel 1 2)), ((rel 2 3)), ((rel 3 4)), ((rel 4 5))

kojoj odgovara ovaj lanac

1--2--3--4--5

Navedimo još da od linearnih relacija su opštije relacije *rel* čija "slika" (graf) podseća na drvo. I na njih se program P2 može uspešno upotrebiti.

Na kraju ćemo upoznati jedan veoma prirodan slučaj koji je pokriven idejama izloženog Primera 2.4.6. Reč je o raznim problemima u vezi sa lavirintima. Primera radi uočimo lavirint prikazan sledećim crtežom



Na tom crtežu su sa a, b, c, d, e, e, f, g označene pojedine prostorije lavirinta dok sa h je dogovorno označena spoljašnost. Kao što je dobro poznato u vezi sa lavirintima uopšte često se postavljaju pitanja poput: 'Da li se od neke njegove prostorije A može po njemu stici do druge prostorije B', dopuštajući da A ili B bude "spoljna" prostorija, tj. spoljašnost lavirinta. Za rešavanje takvih problema naravno nije ni od kakvog značaja oblik i veličina pojedinih prostorija, već je jedino važno znati kako su prostorije povezane, tj. iz koje u koju prostoriju se može preći. Ta povezanost je matematički shvatljiva kao binarna relacija imena, recimo, rel. Sledstveno je proloski iskaziva nizom rel- članaka. Tako, u vezi sa gornjim lavirintom uočimo sledece članke

(Lavir) ((rel h g)) ((rel g f)) ((rel f d)) ((rel d c))
((rel d a)) ((rel a b)) ((rel e b)) ((rel d e))

Primetite da smo svaku medju-vezu uzeli "jednosmerno". Recimo, imamo članak ((rel h g)) koji poručuje da se iz h može direktno stici do g, ali kako nemamo (jer dogovorno ga nismo stavili) članak ((rel g h)) to znači da ne pretpostavljamo postojanje direktnog vraćanja iz g u h. I sada ma koje pitanje gore opisanog oblika, kao što je recimo: 'Da li se iz prostorije h može stici u prostoriju b', je u stvari pitanje "utranzitivljenja" date relacije rel, pa se, u skladu sa dosadašnjim izlaganjem, rešava proloskim programom (Lavir) ┼ P3.

Napominjemo da ćemo se pitanjem "utranzitivljenja" relacijija takođe baviti u Zadacima 6.10-6.15.

3. Z A D A C I, I

Zadatak 3.1 Dati su programi

```
(1) ((p 1))                (2) ((p 1))
    ((p 2))                    ((p 1 2))
    ((p 3) FAIL)               ((p 3)(p 3))
```

Šta su odgovori na pitanje ?((p _x)(PP _x)FAIL)

Odgovor. U slučaju programa (1) konačan odgovor je ne, a na ekranu će se ispisati 1,2 kao _x-rešenja za (p _x). Naravno, 3 nije takvo rešenje (zbog FAIL-a).
Međutim, u slučaju drugog programa , pri raspravljanju datog pitanja algoritam se neće nikad zaustaviti. Naime , zbog trećeg članka algoritam će se zaplesti u beskrajnu petlju ("vrćenje u krug ").
Uprkos tome, drugi program je, recimo, u stanju da odgovori na ova pitanja

?((p 4)), ?((p _x _y)(PP _x _y)FAIL)

Napomenimo da se oba programa lako prevode na Edinburgšku sintaksu:

```
Prvi:     p(1).   p(2).   p(3):-fail.
Drugi:    p(1).   p(1,2).   p(3):-p(3).
```

Zadatak 3.2 Dat je program

((p 1)) ((p 2)) ((p 3 4)) ((p 5)/) ((p 5 6)) ((p 7))

Po Prologu, šta su _x -rešenja formule (p _x), odnosno _x,_y -rešenja formule (p _x _y) ?

Odgovor. Postavimo najpre pitanje traženja svih _x -rešenja za (p _x). Nije teško videti da su proloski ponaosob dokazive sve ove formule

(p 1), (p 2), (p 5),(p 7)

Recimo , već u jednom koraku se zaključuje da je da odgovor na pitanje ?((p 5)), jer / sada ne deluje. Znači,logički bi bilo da skup svih _x- rešenja formule (p _x) bude {1, 2, 5,7}. Ali, nažalost prološkim algoritmom to ne možemo postići, jer

> pod svim _x -rešenjima formule (p _x) smatramo sve _x koji će se pojaviti, tj. na ekranu štampati, tokom raspravljanja pitanja
>
> ?((p _x)(PP _x)FAIL)

Međutim, sada zbog / se kao rešenje neće pojaviti 7.
U slučaju traženja svih _x,_y -rešenja za (p _x _y) neće biti sličnih /-problema, i na ekranu će se pojaviti (3 4) i (5 6).

Zadatak 3.3. Dat je program ((p 1)) ((p 2)) ((p 3 4)). Može se reći da je relacija p "mešovite" dužine. Kako se uprkos tome jednovremeno mogu naći sva njena rešenja ?

Odgovor. Najpre primetimo da se članci programa mogu ovako napisati

((p|(1))) ((p|(2))) ((p|(3 4)))

tj. svi imaju oblik ((p|_x)). Sledstveno, postavimo pitanje

?((p|_x)(PP _x)FAIL)

i na ekranu će se pojaviti

(1) (2) (3 4)

Istaknimo, da je takva mogućnost, odnosno ideja tipična za Micro-prolog, u njemu i formule i članci su u nekom smislu "ravnopravni", odnosno svi su oni samo izvesne liste. Međutim , Edinburgška sintaksa je unela još i niz prividnih dodataka,kojima se doduše u pisanju formula i članaka "beži" od sveta lista,odnosno mnoštva zagrada, ali sledstveno ne dopušta rešenja poput prethodnog. Nažalost ima i mnogo drugih primera,u kojima se uviđaju razni nedostaci Edinburgške sintakse.

Zadatak 3.4. Neka je (p _x _y) neka data relacija, a (q _x) neka je njena "prva projekcija":

(q _x) važi <---> Postoji najmanje jedan _y tako da važi (p _x _y)

Da li se ta projekcija može definisati prološki ?

Odgovor. Može pomoću članka

((q _x)(p _x _y) (odnosno: q(X):-p(X,Y))

Zadatak 3.5. Uvedimo relaciju min ovako

(min _x _y _z) <---> _z≤_x, _z≤_y; i uz to: _z=_x ili _z=_y

Da li njoj odgovara program

```
((min _x _x _x))                    |   odnosno   min(X,X,X).
((min _x _y _x)(LESS _x _y))        |             min(X,Y,X):-X<Y.
((min _x _y _y)(LESS _y _x))        |             min(X,Y,Y):-Y<X.
```

u smislu da ta relacija zadovoljava navedene članke. Da li se program može koristiti

a) za proveru važenja (min _x _y _z), za date _x,_y,_z.
b) za generisanje _z tako da važi (min _x _y _z), gde su _x,_y dati.

Odgovor. Na sva pitanja odgovor je da.

Zadatak 3.6. Napisati prološki program koji "oponaša" rad sledećeg BASIC-programa

```
 5  INPUT A,B
10  X=A+B
20  PRINT X
```

Odgovor. Rešenje upotrebom pomoćne nularne relacije upal (skraćenica od reči upaljač):

((upal)(R _A)(R _B)(SUM _A _B _X)(PP _X))

odnosno

```
upal:-read(A),read(B),X is A+B,write(X),nl.
```

Zadatak 3.7. Prološki definisati relaciju r:

(r x y) važi <---> x*x + y*y < 25.

Odgovor. U LPA i Arity prolozima je kraće rešenje

```
r(X,Y):- X*X+Y*Y<25.
```

dok u Micro-prologu[1], pošto on koristi samo liste članak je nešto duži:

```
((r _X _Y)(TIMES _X _X _A)(TIMES _Y _Y _B)(SUM _A _B _C)(LESS _C 25))
```

Zadatak 3.8. Neka formula nzd(X,Y,Z) znači Z je najveći zajednički delilac prirodnih brojeva X,Y. Da li ta relacija zadovoljava članke ovog prološkog programa

```
(Δ)     nzd(X,X,X).
        nzd(X,Y,Z):-X<Y, YY is Y-X, nzd(X,YY,Z).
        nzd(X,Y,Z):-nzd(Y,X,Z).
```

Koja od ovih svojstava ima taj program[2]:

(i) Proveran po X,Y,Z relacije nzd(X,Y,Z).
(ii) Generatoran po Z relacije nzd(X,Y,Z).

Odgovor. Neka NZD(X,Y) označava najveći zajednički delilac brojeva X,Y. Tada uopšte važe ove jednakosti

```
NZD(X,Y)=X            ,ako X=Y
NZD(X,Y)=NZD(X,Y-X)   ,ako X<Y
NZD(X,Y)=NZD(Y,X)     ,ako X>Y.
```

Nije teško videti da je program (Δ) upravo relacijski, tj. prološki prevod tih jednakosti; stoga relacija nzd zaista zadovoljava članke programa (Δ). Navedene jednakosti,usled prisustva svojevrsne rekurzije, odnosno "spusta" mogu poslužiti za traženje NZD(X,Y), gde X,Y dati prirodni brojevi.Recimo, za NZD(25,10) prema gornjim jednakostima imamo ovaj račun

```
NZD(25,10)=NZD(10,25)   Po trećoj jednakosti
          =NZD(10,15)   Po drugoj
          =NZD(10,5)    Po drugoj
          =NZD(5,10)    Po trećoj
          =NZD(5,5)     Po drugoj
          =5            Po prvoj
```

pa je 5 rezultat. Međutim, lako je videti da program (Δ) radi skoro na podudaran način. I tako, lako se zaključuje da uopšte program (Δ) ima svojstvo (ii). U stvari, ima i svojstvo (i) , jer kad su dati X,Y,Z programom (Δ) se pitanje oblika ?- nzd(X,Y,Z). posle konačno koraka prevede na neko pitanje oblika ?- nzd(U,U,Z) , gde je U neki prirodan broj na koje se prema prvom članku odmah dobije odgovor.

Napomena 3.1. Navedeni NZD- program u osnovi skoro jedino koristi operaciju oduzimanje.Međutim, obično se NZD datih brojeva nalazi uz pomoć delenja, odnosno koristi se ovakvo tvrđenje

Ako važi jednakost a=b*q+r ,gde a,b,q,r celi brojevi
onda važi jednakost NZD(a,b)=NZD(b,r).

Evo u Edinburgškoj sintaksi jednog programa napravljenog sa tom idejom

```
nzd(A,0,A).
nzd(A,B,Rez):- A>B,R is A mod M, nzd(B,R,Rez).
nzd(A,B,Rez):- nzd(B,A,Rez).
```

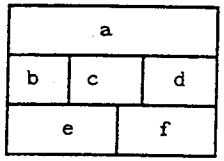
gde nzd(A,B,C) je zamena za: C je najveći zajednički delilac za A,B. Dalje korišćena je celobrojna operacija mod. Uopšte,[3] a mod b, gde su a,b dati celi brojevi znači ostatak delenja a sa b.Recimo, 17 mod 5 je 2. Takođe,iz praktičnih razloga je prvim člankom dopušteno i prisustvo 0.

Zadatak 3.9.
(i) Date su dve grupe brojeva

2,3,5 i 4,7

Napisati program za odvajanje svih dvojki (x,y), x je u prvoj a y u drugoj grupi, pri čemu x,y zadovoljavaju uslov : x+y je manji od 10.

(ii) (Problem četiri boje) Data je mapa



gde a,b,c,d,e,f odgovaraju izvesnim zemljama koje se graniče kao što je navedeno. Treba celu kartu obojiti sa četiri boje , tako da nikojim susednim zemljama ne odgovara ista boja.

Rešenje.(i) U rešenju koje navodimo upoznaćemo jednu od važnih metodoloških ideja -ideju "generiši i proveri" . Naime, zamisao je da nekako "generišemo" sve dvojke (x,y) i proverimo koje zadovoljavaju dati uslov.
U tu svrhu uvodimo najpre dva predikata dat1, dat2 koja će biti "zadužena" da zapamte date grupe brojeva. To činimo pomoću ovih članaka

```
dat1(2).   dat1(3).   dat1(5).
dat2(4).   dat2(7).
```

To je ceo program.Da bismo obavili traženi posao postavimo ovo pitanje

```
?- dat1(X),dat2(Y),Z is X+Y,Z < 10, write(X),tab(3),write(Y),fail.
```

Šta će se desiti? Prvo , gledano u pitanje sleva nadesno , vidimo da zbog dat1(X) X će dobiti prvu vrednost 2,a dalje zbog dat2(Y) Y će dobiti vrednost 4, dalje Z će biti 6, pošto 6<10 , na ekranu će se štampati 2,4, tj. vrednosti za X,Y sa međurastojanjem 3 mesta (zbog tab(3)).Iza toga stižemo do fail-a. On tera nazad, slobodnije rečeno da za njemu prethodeće formule tražimo nove dokaze.To ujedno znači da će i neke nepoznate[4] pretrpeti izmene, tj.dobiti nove vrednosti.Odmah napominjemo da je to jedno od zamršenijih pitanja u Prologu,i da prološke knjige po pravilu "ćute" o tome.O tome ćemo podrobnije pričati u delu 5.2, a ovom prilikom slobodnijim rečima iskazujemo ovo značajno pravilo:

> Neka nepoznata sme, pri proceduri vraćanje , da promeni svoju vrednost upravo na onom mestu gde je tu vrednost ranije bila dobila.

Vratimo se pitanju.Za sve levake fail-a do formule dat2(Y) nema drugih do-

---

[1] Tu mislimo na osnovnu verziju Micro-prologa.

[2] O pojmovima proveran, generatoran više u Napomeni 3.2

[3] Već smo je spomenuli u prethodnoj tački.

[4] Videti Napomenu 2.1.1.

kaza[5] pa sada tražimo drugi dokaz za dat2(Y). Prema iznetom pravilu Y sme da dobije novu vrednost koja će sada biti 7, i dalje će se na ekranu štampati 2 i 7, jer važi nejednakost 2+7<10. Opet zbog fail-a idemo nazad i dođemo čak do dat1(X), jer za dat2(Y) smo potrošili sve grane. Pošto je nepoznata X upravo tu bila dobila vrednost to tu sme i da je promeni. Nova vrednost za X će biti 3, i sa njom u vezi na ekranu će se pojaviti 3 i 4, jer jer važi nejednakost 3+4<10. Opet fail tera nazad i sada se dođe do dvojke 3 i 7, ali uslov 3+7<10 nije ispunjen pa vraćanje nastaje od mesta Z < 10. Pri tom vraćanju X će dobiti novu vrednost 5, a iza toga Y će najpre dobiti vrednost 4, i zbog 5+4<10 na ekranu će se štampati 5 i 4. U stvari, to je poslednja dvojka, jer pri novom vraćanju (zbog fail-a) za Y će se dobiti 7, ali dvojka (5 ,7) ne zadovoljava naloženi uslov.

(ii) Taj primer je u vezi sa čuvenim problemom četiri boje , koji glasi: Da li se svaka ravanska mapa može obojiti sa 4 različite boje tako da susednim zemljama uvek odgovaraju različite boje ?

Odgovor je da i nađen je tek 1976 godine.
Navešćemo dva rešenja postavljenog zadataka. Evo najpre prvog:

U ovom rešenju dogovorno koristimo žutu, zelenu, crvenu i plavu boju. Dalje predikatom imena sused ćemo zapisati da susedne zemlje moraju imati različite boje. U tu svrhu uvodimo ove sused-članke:

```
sused(zuta,zelena).
sused(zuta,crvena).
sused(zuta,plava).
    sused(zelena,zuta).
    sused(zelena,crvena).
    sused(zelena,plava).
sused(crvena,zuta).
sused(crvena,zelena).
sused(crvena,plava).
    sused(plava,zuta).
    sused(plava,zelena).
    sused(plava,crvena).
```

Dalje, u vezi sa konkretnom mapom uvodimo ovaj predikat ajde:

```
ajde(A,B,C,D,E,F):-sused(A,B),sused(A,C),sused(A,D),sused(B,C),sused(B,E),
                   sused(C,D),sused(C,E),sused(C,F),sused(D,F).
```

U ajde-članku su prisutne promenljive A,B,C,D,E,F kao predstavnici pomenuih zemalja a,b,c,d,e,f, odnosno tačnije rečeno, treba shvatiti da te promenljive predstavljaju tražene boje pojedinih mapa. Drugim rečima:

Te promenljive su nepoznati članovi skupa {zuta,zelena,crvena,plava}

Program se uključuje pitanjem ?-ajde(A,B,C,D,E,F). i Prolog[6] u stvari, upošljavajući svoj mehanizam, traži vrednosti tih promenljivih tako da bude

[5] Primetimo da je Z dobilo vrednost na mestu formule
Z is X+Y
pa bi u skladu sa iznetim pravilom tu smelo da promeni vrednost. Ali, to nije moguće jer za tu formulu nema drugih mogućnosti ("grana") za dokaz.

[6] Tj. nepoznatih

ispunjena konjunkcija sused(A,B).....,sused(D,F), odnosno rep ajde-članka. Naravno tako se upravo rešava postavljeno pitanje[7].
Inače nije teško uvideti da i u tom traganju za A,B,....,E se koristi metodološka ideja "generiši i proveri", ali malo složenije. Evo nekoliko reči o tečenju algoritma:

> Prvo , da bi se dokazala formula sused(A,B) za A,B se daju vrednosti žuta,zelena i idući udesno A,B smatramo konstantama , odnosno ako se bude ukazala potreba da im menjamo vrednost onda to može biti samo na mestu formule sused(A,B). Drugim rečima A,B se generišu na tom mestu tu. Dalje, javi se potreba dokaza formule sused(A,C). Na tom mestu se generiše C,dok A već ima vrednost žuta. Dalje, na mestu sused(A,D) se generiše D , i posle toga se dođe do mesta sused(B,C) gde nema novog generisanja već samo provere. Ako ta formula ne bude ispunjena vraćamo se ulevo do "prvog generatornog mesta"da bismo promenili vrednost neke od nepoznatih. To će ovde biti mesto sused(A,D) gde ćemo "dopustiti da D dobije novu vrednost, itd, itd. Jedno rešenje zadatka nastaje kad u toj "generatorno-provernoj igri" uspemo da se probijemo na krajnju formulu ajde-članka, tj, na formulu sused(D,F) i pritom se dogodi da je ona ispunjena.

Evo sada drugog rešenja. Ono je opštije jer se odnosi na ma koji broj datih zemalja i broj boja , tj. i zemlje i boje se zadaju pri postavljanju pitanja. Sam program glasi

```
:-write('Pitati sa: boji_mapu(A,B),gde A treba da budu date zemlje,a B '),
      nl,write('lista boja kao na primer:  [bela,zuta,plava,zelena]'),nl,
      write('Zemlje se zadaju kao liste čiji svaki član je oblika'),nl,
      write(' zem(ime,Boja,Lista_suseda)'),nl,
  write('Evo primera zadavanja liste zemalja, upravo prema postavci'), nl,
  write('ovog zadatka: '),nl,
  write('[zem(a,A,[B,C,D]),zem(b,B,[A,C,E]),zem(c,C,[A,B,D,E,F]),'),nl,
  write(' zem(d,D,[A,C,F]),zem(e,E,[B,C,F]),zem(f,F,[C,D,E])]').

boji_mapu([Zem|Zem1],Boje):-boji_zemlju(Zem,Boje),boji_mapu(Zem1,Boje).
boji_mapu([],Boje).

boji_zemlju(zem(Ime,Boja,Susedi),Boje)
   :-izvadi(Boja,Boje,Boje1),podskup(Susedi,Boje1).

izvadi(X,[X|Xs],Xs).
izvadi(X,[Y|Ys],[Y|Zs]):-izvadi(X,Ys,Zs).

podskup([X|Xs],Ys):-clan(X,Ys),podskup(Xs,Ys).
podskup([],Ys).

clan(X,[X|Y]).
clan(X,[U|V]):-clan(X,V).
```

Pošto program ima članak koji počinje sa znakom "grla" :- ,to znači da će taj članak biti odmah (pri učitavanju) izvršen, što ovde znači da će ispisati navedeno uputstvo , u kom je već prilično objašnjeno. Pre objašnjenja kako radi program dajemo reč-dve o usluznim predikatima: izvadi, podskup i clan. Clan je na uobičajen način definisan predikat "biti član". Predikat

[7] Naravno pitanjem:
?-ajde(A,B,C,D,E,F),fail.
se dolazi i do svih drugih rešenja.

izvadi po osnovnoj zamisli služi da "izvadi neki član iz date liste". Recimo, pri računu formule: izvadi(2,[3,2,4],X) X će dobiti vrednost [3,4]. Ali, pored toga, predikat izvadi može da ima i sposobnost "generisanja". Naime, ako računamo formulu, kao

```
izvadi(X,[3,2,4],Y)
```

gde su X, Y trenutno "nevrednovane" promenljive, onda one dobiju ove vrednosti:

```
X=3, Y=[2,4]
```

I više, ako bi se sticajem okolnosti, pri proceduri vraćanje , tražilo ponovno računanje te formule, onda bismo za X,Y imali: X=2, Y=[3,4] i slično dalje.

Predikat podskup u osnovi odgovara odnosu "članovi jedne liste su i članovi druge", ali takođe ima i "generativnu" sposobnost. Tako, pri računu formule, kao

```
podskup([A,B],[1,2,3])     (A,B su "nevrednovane")
```

za A,B će se dobiti: A=1, B=1, a pri eventualnom ponovnom računanju te formule redom će se pojaviti sve dvočlane liste koje su "podskupovi" liste [1,2,3].

Evo sada kako radi dati program. U vezi sa datom mapom postavimo ovo pitanje

```
?-boji_mapu([zem(a,A,[B,C,D]),zem(b,B,[A,C,E]),zem(c,C,[A,B,D,E,F]),
             zem(d,D,[A,C,F]),zem(e,E,[B,C,F]),zem(f,F,[C,D,E])],
                                          [žuta,zelena,crvena,plava]).
```

Tada prema datom programu, nešto kraće rečeno, treba računati ovu konjunkciju

```
(*) boji_zemlju(zem(a,A,[B,C,D]),Boje),
    boji_zemlju(zem(b,B,[A,C,E]),Boje),
      boji_zemlju(zem(c,C,[A,B,D,E,F],Boje),
        boji_zemlju(zem(d,D,[A,C,F],Boje),
          boji_zemlju(zem(e,E,[B,C,F],Boje),
            boji_zemlju(zem(f,F,[C,D,E],Boje)
```

gde Boje=[žuta,zelena,crvena,plava]. Računamo sada prvi "sastavak", što se svodi na račun ove konjunkcije

```
izvadi(A,[žuta,zelena,crvena,plava],Boje1),podskup([B,C,D],Boje1)
```

Tada se ovo dešava:

A dobija vrednost žuta, a Boje1 vrednost [zelena, crvena, plava]. Dalje, pri računu podskup-formule, imamo ove jednakosti

B=zelena, C=zelena, D=zelena

Tu istaknimo da je promenljiva A dobila vrednost u izvadi-formuli, pa ako pri backtrackingu ustreba tu može da je promeni. Sa druge strane promenljive B, C, D su dobile vrednosti u podskup-formuli, pa ih tu smeju menjati. Drugo, vidi se da je A, blagodareći izvadi-predikatu, dobilo "boju" koja je različita od "boja" za susede B,C,D. U stvari, to svojstvo: "zemlja je različito obojena od svojih suseda" je "udahnuto" u svaku od boji_zemlju- formulu konjunkcije (*). Sledstveno, ako nam

uspe da "stignemo na kraj te konjunkcije", sve će date zemlje biti obojene po postavljenom zahtevu.

Dalji tok algoritma približno teče ovako. Pri računu pojednih članova konjunkcije (*) stalno se pojavljuju računanja formula imena

izvadi, podskup

I tada ili neke promenljive dobijaju vrednosti, pa onda "po konjunkciji (*) idemo udesno", ili usled trenutne netačnosti se moramo vraćati nazad i pokušati "prodor" sa nekim novim vrednostima promenljivih. To tako slobodnije rečeno, cik-cak kretanje se dešava dok nam konačno ne uspe da promenljivim A,B,C,D,E,F damo dobre vrednosti i dokažemo (*). Kao što se vidi i taj program ima u osnovi ideju "generiši i proveri".

Zadatak 3.10. Prevesti na Prolog ovaj algoritam

```
Od i=1 do 10  štampaj  i
```

Rešenje. Taj algoritam je tipičan "navišan" algoritam , a Prologu su "najmiliji" rekurzivni, ili bolje reći "naniżni" algoritmi. Rekli smo, "najmiliji", ne želeći da tvrdimo da Prolog kao jezik nije u stanju da ostvari nikakav navišni algoritam. Može da ostvari mnoge od njih, ali često je potrebna određena dovitljivost. Jedno rešenje je izraženo ovim člankom

```
f(N):-N<11,write(N),N1 is N+1,f(N1).
```

kojim je implicitno definisana relacija f. Tada na pitanje ?-f(1). na ekranu će se pojaviti red-za-redom brojevi

```
1  2  3  4  5  6  7  8  9  10
```

Kao što vidite, prividno smo pitali Prolog za vrednost formule f(1), a u stvari, tako smo ga naterali da za nas uradi naloženi zadatak. Takva "prevarna" misao je, kao što će se dalje iz raznih primera videti, tipična kad kad god želimo da "uprologimo" neki navišni algoritam.

Pored prethodnog rešenja navodimo i ovo koje se sastoji iz dva članka , i i prvi od njih je uslov zaustavljanja (u prethodnom rešenju tu ulogu je imao deo N<11) :

```
f(11).
f(N):-write(N),nl,N1 is N+1,f(N1).
```

I sada se traženi posao postiže postavljanjem pitanja ?-f(1). Naravno, mogli bismo uvesti i članak

```
upaljac:-f(1).
```

i onda bi se pitanje postavljalo sa ?- upaljac.

Sada navodimo nekoliko zadataka sa listama. Koristimo LISP- sintaksu, jer je sada tehnički podesnija.

Zadatak 3.11. Prološki definisati relaciju: x nije član od y.

Rešenje. Navodimo nekoliko rešenja

```
(i)     ((Neclan1 _x ()))
        ((Neclan1 _x (_y|_z))(dif_x _y)(Neclan1 _x _z))
        ((dif _x _y)(NOT EQ _x _y))
```

Tu smo kao pomoćnu uveli relaciju dif, tj, x≠y. Nju smo mogli i ovako uvesti bez korišćenja NOT (u stvari se "oponaša" definicija NOT-a)

```
((dif _x _y)(EQ _x _y) / FAIL)
((dif _x _y))
```

Naravno, sasvim bi slično izgledao program u Edinburgškoj sintaksi, uz osnovnu razliku da se tada

umesto (EQ _X _Y) piše X=Y, a umesto
(NOT EQ _X _Y) se piše not(X=Y), ili još kraće X\=Y.

(ii) Pretpostavljajući ovaj program za x ε y ("x element od y")

```
((elem _x (_x|_y)))
((elem _x (_u|_v))(elem _x _v))
```

"neclan"-relaciju možemo definisati ovako

```
((Neclan2 _x _y)(NOT elem _x _y))
```

Naravno ako Prolog sa kojim radimo ima već ugrađenu relaciju elem, kao što je slučaj sa Micro- i LPA- prolozima, onda relaciju Neclan2 možemo definisati direktno pomoću nje. Recimo , u Micro-prologu možemo relaciju Neclan ovako definisati

```
((Neclan3_x _y)(NOT ON _x _y))
```

(iii) (LISP-sintaksa) Upotrebom osnovne prološke relacije FORALL imamo ovaj program

```
((Neclan4 _x _y)
        (FORALL ((ON _z _y))
                ((NOT EQ _z _x))
        )
)
```

Inače, FORALL ima ovakav opšti zapis

```
(FORALL  Sastav1 Sastav2)
```

gde su Sastav1, Sastav2 dve liste čiji članovi su formule[1]. Smisao:

Za sve slučajeve važenja svih članova Sastav1, tj. "sva njihova rešenja", "uradi", izračunaj redom članove sastava Sastav2

što se kraće opisuje ovako :

Za sva rešenja od Sastav1 "uradi" Sastav2.

Mali primeri

?((FORALL ((a _x)) ((PP _x)))), štampaće sva rešenja neke date relacije a.

?((FORALL ((ON _x (1 2 3))(ON _x (2 3 5))) ((PP _x)))), štampaće 2,3.

Primetimo , da u Arity-prolog nije ugrađena relacija poput FORALL. Ali, u LPA jeste i njen zapis je

```
forall(sastav1,sastav2)
```

sa smislom kao za FORALL. Recimo, program sa Neclan4 relacijom sada izgleda

---

[1] Recimo, takve su ove liste

```
((PP Pera)(EQ _x 3)(a _x _y))
((d _x _y)(s _x)(R _y))
```

Primetite, da su članci takođe takve liste. Ističemo još da o FORALL predikatu dublje izlažemo u Primeru 4.1.4


```
neclan4(X,Y):-forall(on(Z,Y),Z\=X).
```

Napomena 3.2. Neka je P neki proloski program i (rel x1 x2 ... xn) izvesna relacija (tj. tačnije rečeno formula). Tada kažemo da je program P za nju za nju proveran ukoliko kad x1, x2, .., xn imaju konstantne vrednosti program je u stanju da sračuna vrednost te relacije.
Dalje, generatoran je po nekim od tih promenljivih, recimo, po xi1, xi2, .., xik ako je pod pretpostavkom "datosti" preostalih promenljivih u stanju da nađe vrednosti za xi1, xi2, ..., xik tako da važi (rel x1 x2 ... xn).

Zadatak 3.12. Uz relaciju opisanu rečima priložen je proloski program. Za svaku relaciju definisanu programom proveriti da li je za nju program

(a) proveran
(b) generatoran po nekim argumentima

Dodatno, ako je relacija funkcijska proveriti da li joj je obezbeđena jedinstvenost (vid. Napomenu 2.4.1).

(i) x je broj članova od y (formulski zapis (broj y x))

```
((broj1 () 0))
((broj1 (_x|_y) _z)(broj1 _y _z1)(SUM 1 _z1 _z))
((broj _x _y)(broj1 _x _y)/)
```

Recimo, 3 je broj članova liste (2 (a b) 77).

(ii) x je zbir članova od y (formulski zapis (zbir y x))

```
((zbir1 () 0))
((zbir1 (_x|_y) _z)(zbir1 _y _z1)(SUM _x _z1 _z))
((zbir _x _y)(zbir1 _x _y)/)
```

Recimo, 7 je zbir članova liste (2 3 2).[2]

(iii) x je prvi član od y (formulski zapis (prvi y x))

```
((prvi1 (_x|_y) _x))
((prvi1|_x)(PP Nema))
((prvi _x _y)(prvi1 _x _y)/)
```

Recimo, (1 3) je prvi član liste ((1 3) 7 8). Međutim, u slučaju, recimo, prazne liste je predviđeno štampanje poruke Nema.

(iv) x je poslednji član od y (formulski zapis (posl y x))

```
((posl1 (_x) _x))
((posl1 (_x|_y) _Rez)(posl1 _y _Rez))
((posl1|_x)(PP Nema))
((posl _x _y)(posl1 _x _y)/)
```

(v) x,y su poslednja dva člana od z (formulski zapis (dva x y z))

```
((dva1 _x _y (_x _y)))
((dva1 _x _y (_u|_v))(dva1 _x _y _v))
((dva _x _y)(dva1 _x _y)/)
```

---

[2] Ako postavimo pitanje zbira članova liste ((1 2) 4) ili liste (a b), Micro će dati odgovor ?, tj. ne, Razlog je u definiciji relacije SUM. Recimo, svaka od formula (SUM a 2 6), (SUM (1 2) 6 7) ima vrednost ne. Sličan je pristup u LPA-prologu. Ali, u Arity bi se u takvim slučajevima kao rezultat dao err (skraćenica od error = greška). Tako, ako pitamo X is a+5, dobićemo X=err.

```
(vi) x,y su susedni članovi u z          (formulski zapis (sus x y z))
      ((sus _x _y (_x|(_y|_z))))
      ((sus _x _y (_u|_v))(sus _x _y _v))
      ((sus|_x)(PP Nema))

(vii) x je k-ti član od y                (formulski zapis (kti x k y))
      ((kti1 _x 1 (_x|_y)))
      ((kti1 _x _k (_u|_v))(kti1 _x _m _v)(SUM _m 1 _k))
      ((kti1|_x)(PP Nema))
      ((kti _x _k _y)(kti1 _x _k _y)/)

(viii) x je početni komad od y           (formulski zapis (pockomad x y))
      ((pockomad () _x))
      ((pockomad (_x|_A) (_x|_B))(pockomad _A _B))

(ix)  x je,dužine k,početni komad od y   (formulski zapis (kkomad k y x))
      Program sastaviti tako da za date y,k se može naći x(ako ga ima),tj.
      da bude funkcijski.
        ((kkomad1 1 (_x|_y) (_x)))
        ((kkomad1 _n (_a|_B) (_a|_C)) (SUM 1 _k1 _k)(kkomad1 _k1 _B _C))
        ((kkomad1|_x)(PP Nema))
        ((kkomad _k _x _y)(kkomad1 _k _x _y)/)

(x)   x je podlista od y, u smislu: svaki član od x je član od y
                                          (formulski zapis (podlis x y))
        ((podlis () _x))
        ((podlis (_a|_B) _x)(ON _a _x)(podlis _B _x))

(xi)  x je skupovno jednako sa y, tj. svaki član od x je član od y i
                                  obratno.   (formulski zapis  (jed x y))
         ((jed _x _y)(podlis _x _y)(podlis _y _x))
                                 gde se podrazumevaju podlis-članci iz (x).

(xii) z nastaje kad se na listu x redom dodaju članovi liste y,
      recimo,ako x=(1 2 3),y=(a b), onda z=(1 2 3 a b).
         Uz formulski zapis  (dopis x y z) imamo ovaj program
            ((dopis1 () _x _x))
            ((dopis1 (_a|_b) _x (_a|_c))(dopis1 _b _x _c))
            ((dopis _x _y _z)(dopis1 _x _y _z)/)

Istaknimo , da relacija dopis1 ima ovo lepo svojstvo -generativnost po
njena prva dva argumenta. Naime,recimo, na pitanje

          ?((dopis1 _x _y (1 2 3))(PP _x _y)FAIL)

na ekranu će se pojaviti ove dvojke lista

      ()  (1 2 3);  (1) (2 3);  (1 2) (3);  (1 2 3) ()

(xiii) (a) z nastaje kad se na početak liste y doda x;
                                    formulski zapis:      (pocdod x y z)
       (b) z nastaje kad se na kraj liste y doda x;
                                    formulski zapis:      (krajdod x y z)

          ((pocdod _x _y (_x|_y)))
           ((krajdod _x () (_x)))
           ((krajdod _x (_a|_b) (_a|_c))(krajdod _x _b _c))
Primetite da dodavanje na kraj iziskuje rekurziju.

(xiv)     z nastaje kad se u listi y obriše prva pojava x-a.
                                    formulski zapis (prvobrisi x y z)
```



```
          ((prvobrisi1 _x () ()))
          ((prvobrisi1 _x (_x|_a) _a))
          ((prvobrisi1 _x (_y|_a) (_y|_b)) (prvobrisi1 _x _a _b))
          ((prvobrisi _x _y _z)(prvobrisi1 _x _y _z)/)

(xv) x je lista nastala iz liste y izbacivanjem međusobno jednakih , odno-
     sno ostavljanjem po jednog takvog.       formulski zapis (oskup y x)
        ((oskup1 () ()))
        ((oskup1 (_a|_b) _c)(ON _a _b)(oskup1 _b _c))
        ((oskup1 (_a|_b) (_a|_c))(oskup1 _b _c))
        ((oskup _a _b)(oskup1 _a _b)/)

(xvi)  (a) z je presek "skupova" x,y;      formulski zapis  (presek x y z)
       (b) z je unija "skupova" x,y;       formulski zapis   (unija x y z)
       (c) z je razlika x sa y;            formulski zapis (razlika x y z)
              ((presek1 () _a ()))
              ((presek1 (_a|_b) _c (_a|_d))(ON _a _c)(presek1 _b _c _d))
              ((presek1 (_a|_b) _c _d)(presek1 _b _c _d))
              ((presek _x _y _z)(presek1 _x _y _z)/)
         ((unija1 () _a _a))
         ((unija1 (_x|_a) _b _c)(ON _x _b)(unija1 _a _b _c))
         ((unija1 (_x|_a) _b (_x|_c))(unija1 _a _b _c))
         ((unija _x _y _z)(unija1 _x _y _z)/)
                ((razlika1  () _a ()))
                ((razlika1 (_a|_b)_c  _d )(ON _a _c)(razlika1 _b _c _d))
                ((razlika1 (_a|_b) _c (_a|_d))(razlika1 _b _c _d))
                ((razlika _x _y _z)(razlika1 _x _y _z)/)

(xvii)  x se u listi y pojavljuje ukupno k-puta
                                          formulski zapis (kolkoputa x k y)
   ((kolkoputa1 _x 0 _y)(NOT ON _x _y))
   ((kolkoputa1 _x 1  (_x|_y))(NOT ON _x _y))
   ((kolkoputa1 _x _k (_x|_y))(kolkoputa1 _x _m  _y)(SUM  1  _m _k))
   ((kolkoputa1 _x _k (_u|_v))(kolkoputa1 _x _k _v))
   ((kolkoputa   _x   _k   _y)(kolkoputa1   _x   _k   _y)/)

Recimo,na pitanje
    ?((kolkoputa 2 _k (3 2 4 2 2))(PP _k))
na ekranu će se pojaviti 3.

Zadatak 3.13. Na  Prolog prevesti ovaj niz naloga:
            Daj _x
            Ako _x manje od 3 štampaj 'Manji od 3' a
                                          inače štampaj 'Nije manji od 3'

Rešenje. Jedno rešenje je određeno ovom program (pokreće se sa ?((upal)))
       ((upal)(PP Daj)(R _x)(pamti _x))
          ((pamti _x)(LESS _x 3)(PP Manji od 3))
          ((pamti _x)(PP Nije manji od 3))
Međutim, Micro-prolog ima ugrađen IF-predikat,koji se piše u obliku
          (IF _A _B _C)
o kome podrobije govorimo u Primeru 4.1.2, a ovom prilikom ovoliko:

       Ta formula bi se mogla ovako čitati:
       Ako važi _A onda "uradi" _B, a inače "uradi" _C.

Tu su _B,_C neki sastavi (već smo ih pominjali u Primeru 3.11).
```

Upotrebom IF-predikata imamo ovo drugo rešenje

```
((upal)(PP Daj)(R _x)
        (IF (LESS _x 3) ((PP Manji od 3)) ((PP Nije manji od 3))))
```

Zadatak 3.14. Na prolog prevesti ovaj niz naloga:

Daj koeficijente _a _b linearne jednačine _a*_x=_b_
Odredi x-rešenje te jednačine (ako ga ima, a inače upozori da ga nema)

Rešenje. Evo jednog programa, pokreće se sa ?((ajde))

```
((ajde)(PP Daj koeficijente)(P a= )(R _a)(P b= )(R _b)
    (IF (NOT EQ _a 0) ((TIMES _a _x _b)(PP Rešenje je _x))
                ((IF (NOT EQ _b 0) ((PP Nemoguća))
                                   ((PP Svaki broj je rešenje))
                 )
                )
    )
)
```

U tom rešenju se koristi predikat P , čijom "uslugom" za razliku od PP se vrši ispis ali bez ispisa novog reda. Dalje, u rešenju imamo "dvojni" IF, što je malo zamršenije. Naravno, rešenje se može osloboditi svega toga upotrebom pamti-ideje (videti Primer 2.4.3, kao i prethodni Zadatak 3.13)

Zadatak 3.15. U vezi sa brojevima uočimo relaciju 'izmedju' sa formulskim zapisom (izmedju _x _y _z), što bi se čitalo _x je između[3] _y i _z. Da li ta relacija uvedena člancima

```
((izmedju _x _x _y)(LE _x _y))
((izmedju _x _y _z)(SUM _y 1 _y1)(LE _y1 _z)(izmedju _x _y1 _z))
      ((LE _x _x))
      ((LE _x _y)(LESS _x _y))
```

ima svojstvo provernosti za formule oblika (izmedju p q r), gde su p,q,r _x p q), gde p,q dati celi brojevi ?

Zadatak 3.16. Pomoću odgovarajućeg prološkog algoritma:

Na ekranu štampati sve cele brojeve između dva data cela broja.

Rešenje. Jedno rešenje uz pomoć relacije 'izmedju' iz Zadatka 3.15 glasi

```
((pisi _x _y)(izmedju _z _x _y)(PP _z)FAIL)
((pisi _x _y))
```

gde je drugi članak dodat da se na kraju (zbog FAIL-a) ne bi na ekranu pojavio znak pitanja ? .

Zadatak 3.17. Da li je dati broj n oblika m!, gde m=0,1,2,...,tj. da li je dati broj faktorijel nekog broja m ?

Uputstvo. Jedna zamisao je sledeća
Redjamo brojeve 0,1,...,n i za svaki od njih izračunamo faktorijel i njega poredimo sa n, itd.
Bukvalno prevodjenje toga je izvedivo uz pomoć relacije 'izmedju' iz Zadatka 3.15, kao i uobičajenog prološkog algoritma za pravljenje faktorijela. Ali, takav program ima "dve rekurzije", jednu za redjanje brojeva i drugu za traženje faktorijela, pa je stoga prilično spor. Bolji algoritam

[3] Recimo, tačne su formule (izmedju 5 2 7), (izmedju 3 3 5), (izmedju 2 2 2)

ćemo upoznati kasnije (videti Zadatak 3.27).

Zadatak 3.18. Napraviti program za ispisivanje na ekranu brojeva

1!, 2!, ..., n! (n je željeni broj)

pri čemu treba izbeći računanje svakog faktorijela "od početka", već recimo nakon nalaženja 5! množenjem sa 6 dobiti 6!.

Rešenje. Izložićemo jedno rešenje upotrebom u Prolog ugrađenih predikata[4] ADDCL, DELCL koji redom služe da se tekućem programu doda, odnosno "oduzme", obriše neki članak[5]. Recimo, ako se tokom prološkog algoritma pojavi "izračunavanje" formule

(ADDCL ((a 1)(b 2)))

onda na tom koraku će se spisku a-članaka, ali NA KRAJ dodati nov članak

((a 1)(b 2))

Međutim u slučaju izračunavanja formule[6]

(ADDCL ((a 1)(b 2)) k) (k je 1,2,3,...)

isti članak će biti dodat, ali kao k-ti po redu.
Sada navodimo jedno "ADDCL-DELCL" rešenje uočenog problema:

```
((fakt 0 1))
((fakt _x _y)(LESS 0 _x)
             (SUM _x1 1 _x)(fakt _x1 _y1)(TIMES _x _y1 _y))
((faktor _x _y)(fakt _x _y)(DELCL ((fakt _p _q)))
                                  (ADDCL ((fakt _x _y)) 1))
((LE _x _x))
((LE _x _y)(LESS _x _y))
((uradi _i _p) (LE _i _p)
        (faktor _i _x)(PP _x)(SUM 1 _i _i1)(uradi _i1 _p))
((uradi _x _y)(DELCL ((fakt _p _q)))(ADDCL ((fakt 0 1)) 1))
```

U njemu je deo sa fakt- i faktor-člancima zadužen za postupno računanje faktorijela. Tako, ako recimo potražimo _x tako da važi (fakt 5 _x) onda će se na uobičajen rekurzivan način naći 5!. Ali, ako uposlimo faktor-članak, pa tražimo _x tako da važi (fakt 5 _x), onda:

Prvo će se na rečeni način pronaći 5!, a dalje obrisati članak oblika ((fakt _p _q)) za neke p,q, odnosno obrisati članak ((fakt 0 1)). Pride, na sam početak fakt-članaka će biti dodat članak

((fakt 5 120))

Drugim, rečima fakt-deo programa se promenio. Kakva korist od toga? Pa zamislimo da nakon traženja 5! hoćemo da tražimo 8!. Tada prema novom fakt-delu to traženje će se korak-za-korakom prevesti na traženje 7!, 6!, 5!, ali tu će sada biti kraj "spustu", jer imamo članak ((fakt 5 120)) na početku fakt-članaka.

[4] O njima podrobije govorimo u delu 4.1.

[5] U Edinburgskoj sintaksi ti predikati se redom nazivaju assert, retract.

[6] Pazite u njoj se pored članka pojavljuje i broj k.

Evo kako se pomocu datog programa na ekranu stampa 1!, 2!,. ..,n!, gde je n dat broj. Prosto postavimo pitanje

```
?((uradi 1 n))
```

jer uradi-članci su zaduženi da uslugom faktor-članaka redom prave i ispisuju faktorijele od 1! do n!.Na kraju, drugim uradi-člankom se fakt-članci "vrate na polazno stanje".

Primetimo,da se u mnogim problemima donekle na sličan način kao u tom primeru veoma uspešno može koristiti ADDCL-DELCL ideja. Njome se,može se tako reći, dopušta da se tokom proloških algoritma ponešto i definiše , zapamti što liči na mogućnost koje pruža upotreba "globalnih"promenljivih u raznim funkcijskim jezicima (Lisp,Pascal,C).A inače,bukvalno rečeno u Prologu nema pravih globalnih promenljivih,i to je jedan od velikih nedostataka proloških algoritma.

Zadatak 3.19. Među svim dvojkama (i,j), gde i=1,2,...,50, j=3,...,20 štampati sve one sa svojstvom i+j<20.Koliko ih ima ?
Rešenje. Koristićemo relaciju 'izmedju' iz Zadatka 3.15. Jedan deo postavljenog zadatka 'ispis takvih dvojaka' se može rešiti recimo ovako, pomoću relacije 'ispis'

```
((ispis)(izmedju _i 1 50)
        (izmedju _j 3 20)(SUM _i _j _k)
        (IF (LESS _k 20) ((PP Evo _i, _j)) ((EQ 1 1))) FAIL)
((ispis))
```

Program se pobuđuje sa ?((ispis)), a drugi ispis-članak je stavljen da bi se izbeglo završno ? .
Primetite da u IF-formuli ako _k<20 onda treba štampati _i,_j ,a u protivnom slučaju ta IF-formula zbog tačnosti formule (EQ 1 1) se izračuna na da i algoritam teče dalje.Ali FAIL tera nazad,itd.Istaknimo,da je IF tako definisan da sada FAIL neće tražiti neka nova rešenja za (LESS _k 20).
Međutim, ako želimo i da brojimo takve dvojke (i,j), zadatak je nešto teži Tako moguće je napraviti jedno rešenje primenom ADDCL-DELCL ideje.Tako,neka dogovorno br bude predikat "zadužen" za to brojanje. Onda evo odgovarajućeg programa

```
((ispis)
  (izmedju _i 1 50)(izmedju _j 3 20)(SUM _i _j _k)
  (IF (LESS _k 20)
       ((PP Evo _i , _j)(br _X)(SUM 1 _X _XX)(DELCL ((br _X)))
                                             (ADDCL ((br _XX))))
       ((EQ 1 1))
  )

     FAIL)

((ispis))
((br 0))
((ispis1)(ispis)(br _X)(PP Ima ih _X)(KILL br)(ADDCL ((br 0))))
```

Taj se program pobuđuje sa ?((ispis1)).Inače, (KILL br) "briše" sve br-članke.

Zadatak 3.20. Napraviti proceduru kojom se upisuje matrica imena mat formata p x q, gde p,q dati prirodni brojevi.
Uputstvo. Koristi se zamisao prethodnog zadatka, odnosno uslugom relacije

izmedu se u toku algoritma najpre redaju dvojke (i,j), gde i<= p, j <=q i u svakom koraku se stavi (ADDCL ((mat _i _j _vred))) , gde je _vred vrednost koja se prethodno predikatom R učita.
Dodajmo da se i dalje može prološki "prodirati" u algebru matrica, odnosno napraviti programi za razne njihove operacije.

Zadatak 3.21. Da li je dat prirodan broj prost?
Rešenje.Koristimo relaciju 'izmedju' iz Zadatka 3.15.Nju malo "prepravljamo" uvodeći novu relaciju 'medju'.Recimo, važi (izmedju 7 2 7), ali ne važi (medju 7 2 7).Kratko, kod (izmedju a b c) se traži i a<c.Inače, algoritam uz uslugu FORALL-predikata (videti Zadatak 3.11), ima osnovnu ovu misao:

Broj _X je prost upravo ako nije deljiv ni sa jednim od brojeva 2,3,...,_X-1.

To nije najbolji algoritam , ali se lepo ostvaruje u Prologu. Jedan takav program glasi

```
((medju  _X _Y _Z) (izmedju _X _Y _Z) (LESS _X _Z))

((prime 1) / FAIL)
((prime 2))
((prime _X) (FORALL ((medju _Y 2 _X)) ((NOT deljiv _Y _X))))

((deljiv _X _Y) (TIMES _X _Z _Y) (INT _Z))
```

i poziva se sa ?((prime p)), gde p neki dat prirodan broj[7].

Zadatak 3.22. Naći zbir kvadrata redom brojeva 1,2,...,n, gde n dat prirodan broj.

Rešenje. Navešćemo rekurzivno rešenje sa relacijom (zbir kvad a b),što se može čitati ovako: b je zbir kvadrata brojeva 1,2,...,a. Jedno takvo rešenje glasi

```
((zbir _fun 1 _x)(_fun 1 _x))
((zbir _fun _x _y)(SUM 1 _x1 _x)(zbir _fun _x1 _y1)
                                 (_fun _x _xx)(SUM _xx _y1 _y))
((kvad _x _y)(TIMES _x _x _y))
```

i pobuđuje se recimo ovako ?((zbir kvad 4 _x)(PP _x)) da bi se našao zbir kvadrata brojeva 1,2,3,4.Primetite da je rešenje znatno opštije od postavljenog zadatka. Naime, datim rešenjem se može računati

```
fun(1)+fun(2)+...+fun(n)
```

gde je fun data funkcija, određena svojim proloških članicima.

Zadatak 3.23. U datoj listi svaki od znakova a,b,c zameniti redom sa A,B,C a ostale znake ne menjati.

Rešenje.Jedan program glasi

---

[7] Primetimo da je odvojeno 1 proglašen za neprostog, a 2 za prostog.Razlog: FORALL predikat u formuli oblika
(FORALL A B)
ukoliko je A netačno izračunava se sa vrednošću tačno.To je čudno, ali valja ga pamtiti.To se sasvim dobro vidi uz korišćenje stroge definicije predikata FORALL (tačka 4,Primer 4.1.4).

```
((Smena a A))
((Smena b B))
((Smena c C))
((Smena _X _X))
 ((Zam () ()))
 ((Zam (_A|_X) (_B|_Y)) (Smena  _A _B) (Zam _X _Y))
```

i pobuđuje se recimo ovako ?((Zam (a 2 b c) _x)(PP _x)) i na ekranu će se štampati lista (A 2 B C). Primetimo da se izloženi zadatak može uopštavati na razne strane. Tako, recimo osnovna relacija zamene 'Smena' može ili biti zadana pomoću neke liste dvojaka ili se može postaviti zadatak da se osnovne zamene unose pomoću R (read) predikata.

Zadatak 3.24. Kako ispitati da li je data relacija rel
a) simetrična; b) tranzitivna

Rešenje. To su lepi primeri korišćenja FORALL-predikata. Naime, za a) je dosta pitati ?((FORALL ((rel _x _y)) ((rel _y _x)))) ,tj. za sve _x, _y za koje važi (rel _x _y) videti da li važi (rel _y _x).
Slično, za b) se može ovako pitati

```
?((FORALL ((rel _x _y)(rel _y _z)) ((rel _x  _z))))
```

Istaknimo da i LPA-prolog ima potpuno sličan 'FORALL' predikat u oznaci: forall.
Zadatak 3.25. Pođimo od liste [1,2,3,4] i obavimo ovaj niz koraka, u svakom od njih se pojavljuju po dve liste

Korak 1	[1,2,3,4]	[]	Znači počeli smo od date i prazne liste.
Korak 2	[2,3,4]	[1]	(Δ) Zadržali smo rep prve liste prethodnog koraka, a njenu glavu smo dodali na početak druge.
Korak 3	[3,4]	[2,1]	Uradili smo (Δ)
Korak 4	[4]	[3,2,1]	Uradili smo (Δ)
Korak 5	[]	[4,3,2,1]	Uradili smo (Δ)

Kao što se vidi na kraju kao druga lista je postala "obrat" početne liste. Pretpostavljajući da umesto [1,2,3,4] stoji ma koja neprazna lista na Prolog prevesti sličan niz koraka, odnosno, u stvari, algoritam.

Rešenje. Pitanje "obrta" date liste je važno, a prethodno opisan algoritam je veoma efikasan (on je navišan i ima K+1- koraka, gde je K dužina liste). Postupno ćemo opisati kako se naznačeni algoritam obrtanja date liste može "uprologiti". U stvari, čitavo izlaganje će ujedno biti ilustracija kako se izvesni navišni algoritmi mogu "uprologiti", tj. izraziti odgovarajućim proloških programom.

P r v i pokušaj:

Neka ob(X,Y) znači: Y je obrat od X. Članak ob([A|B],C):-ob(B,[A|C]). je tada očigledno sposoban da vrši uzastopne korake opisanog algoritma. Recimo, ako postavimo pitanje ?-ob([1,2,3,4],[]). onda, u skladu sa tim člankom tokom proloških mehanizma će se dešavati "koračanje" opisano u formulaciji zadatka. Ali, to se sve dešava "unutra" i ostaje pitanje kako "izneti" poslednju vrednost drugog argumenta, tj. obrat liste [1,2,3,4].

D r u g i pokušaj:

Sada ćemo malo preraditi gornji članak tako da na kraju dobijemo želje-

ni ispis:

```
obr(X,Y):-(X=[],write(Y));(X=[B|C],obr(C,[B|Y])).
```

Taj članak je ili-oblika, jer ; odgovara vezniku ili. Znači, sve dok prva lista nije prazna, radimo isto kao u prethodnom programu, ali kad se desi X=[], tj. prva lista postane prazna, onda se štampa Y, tj. traženi obrat. Inače o vezniku ili ima više reči u Primeru 4.1.1 .
Međutim, i tom se rešenju može staviti prigovor. Njime se dobije rešenje na ekranu, to je rešenje "za jednokratnu upotrebu", odnosno nemoguće je na takav način tokom nekog proloških algoritma usput po potrebi praviti obrate više lista. Kratko i strože rečeno, još nismo definisali funkciju funkciju : Y je obrat od X.

K o n a č a n pokušaj:

Da bismo napravili tu funkciju u relaciji obrata uposlićemo još jedan argument , koji će tokom algoritma biti "statista", ali na kraju ćemo preko njega zapamtiti rezultat finkcije. Program sa tom idejom ovako izgleda:

```
obrat([X|Y],Z,Rez):-obrat(Y,[X|Z],Rez).
obrat([],Rez,Rez).
obrat(X,Y):-obrat(X,[],Y).
```

Primetite, da je obrat relacija "mešane" dužine: i 2 i 3. Na pitanje oblika ?-obrat([1,2,3,4],Y). se najpre upošljava treći članak, prema kome se dato pitanje svodi na računanje formule obrat([1,2,3,4],[],Y). A sada u igru stupa prvi članak dokle god je prvi argument neprazan. A kad on bude prazan onda u igru ulazi drugi članak, kojim se algoritam završava.
U zadnjem koraku, pre susreta sa drugim člankom, imali smo formulu

```
obrat([],[4,3,2,1],Y)
```

i pri susretu sa drugim člankom Y je dobilo traženu vrednost [4,3,2,1], tj. obrat polazne liste.

Napomena 3.3. Ideja izložena u prethodnom zadatku, reći ćemo ideja navišnosti, je veoma značajna i može se ovako kratko opisati:

Da bi se rešio postavljeni zadatak najpre se sroči navišni algoritam i dalje se taj algoritam prevede na Prolog.

Naravno prevođenje na Prolog ne mora biti sasvim jednostavno, može iziskivati određenu dosetljivost.

Zadatak 3.26. Dati element a izbrisati svuda iz date liste b; (formulski zapis: (svebrisi a b rez), gde rez znači rezultujuću listu.)

Rešenje. Prvo, navodimo rešenje u kome se koristi prethodno opisana ideja "navišnosti" (Napomena 3.3). To je veoma kratak algoritam. Jedino što se u u rezultatu dobije okrenuta lista. Ali, korišćenjem algoritma iz prethodnog zadatka po želji može se otkloniti taj nedostatak.
Evo najpre algoritma u jednom primeru:

Izbaciti 3 svuda iz liste (1 5 3 4 3 2). Usput koristimo jednu listu Priv koja počinje sa () , i postupno postaje traženi rezultat Rez. I ta lista Rez se javlja u algoritmu, kao "statista", ali na kraju preko nje se dobije završni rezultat:

	a	Lista	Priv	Rez	
Korak 1	3	(1 5 3 4 3 2)	()	Rez;	Početno Priv je ().
Korak 2	3	(5 3 4 3 2)	(1)	Rez;	Glava Liste, tj. ovde 1 je različita od a, tj. 3, pa je dodajemo na Priv, a nova Lista je rep prethodne.
Korak 3	3	(3 4 3 2)	(5 1)	Rez;	Radimo kao u Koraku2
Korak 4	3	(4 3 2)	(5 1)	Rez;	Opet Listi sklanjamo glavu, ali pošto je jednaka sa a, tj. 3, ne dodajemo je na Priv.
Korak 4	3	(3 2)	(4 5 1)	Rez;	Radimo kao u Koraku 2
Korak 5	3	(2)	(4 5 1)	Rez;	Kao Korak 4
Korak 6	3	()	(2 4 5 1)	Rez;	Kao Korak 2

Tu je kraj, i Rez je (2 4 5 1). Evo sada opšteg programa

```
((svebri _a (_a|_b) _Priv _Rez) (svebri _a _b _Priv  _Rez))
((svebri  _a (_p|_q) _Priv _Rez)  (svebri _a _q (_p|_Priv) _Rez))
((svebri _a  () _Rez _Rez))
((svebri _a _b _Rez)(svebri _a _b () _Rez))
```

Primer korišćenja: Na pitanje ?((svebri 3 (1 5 3 4 3 2) _Rez) (PP _Rez)) se na ekranu pojavi lista (2 4 5 1).

Navodimo i jedno tipično rekurzivno rešenje. U njemu se koristi funkcijska relacija (prvobrisi x y z), rečima: z nastaje kad se u y skloni prva pojava od x. I dalje, slobodnije rečeno, za dato x i datu listu y ta se funkcija upotrebi nekoliko puta (u stvari, dovoljan broj puta) dok se ne uklone sve pojave od x. U tu svrhu služi relacija *svebrisi*:

```
((prvobris _x () ()))
((prvobris _x (_x|_a)  _a))
((prvobris  _x  (_y|_a) (_y|_b)) (prvobris _x _a _b))
((prvobrisi _x _y  _z)(prvobris _x _y _z)/)
   ((svebrisi _a _x _x) (prvobrisi _a _x _y)(EQ _y _x))
   ((svebrisi  _a  _x  _y)(prvobrisi  _a  _x  _z)(svebrisi  _a  _z  _y))
```

U pretposlednjem članku je osetljivo mesto u kome stoji EQ. Naime, tu se u stvari javlja potreba proveravanja da li važi (prvobrisi _a _x _x), i u duhu u Napomene **2.4.1** to je ostvareno pomoću jedne pomoćne promenljive _y i formule (EQ _x _y).

Zadatak 3. 27. Na navišni način definisati funkciju n!.
Rešenje. Ideja je kao u prethodna dva zadatka. Recimo, ako tražimo 3! postupićemo ovako (navodimo korak-za-korakom tok):

n	i	Priv	Rez;	
				n je 3, i je "tekućnik", Prvi služi za postupno pravljenje rezultata. Rez "čeka" krajnji rezultat.
3	1	1	Rez	n je stalno = 3, dok tekućnik i ide od 1 do 3. Algoritam se završava kad i bude n. Tada Priv
3	2	1*2	Rez	postane rezultat.
3	3	1*2*3	Rez	Znači, Rez je 1*2*3, tj. 6.

Sada navodimo opšti takav navišan program:

```
((fakt _n _i _Priv _Rez) (LESS _i _n)
     (SUM  1 _i _i1) (TIMES _i1 _Priv _Priv1)(fakt _n _i1 _Priv1 _Rez))
((fakt _n _n _Rez  _Rez))
((fakt _n _Rez)(fakt _n 1 1  _Rez))
```

Recimo, na pitanje ?((fakt 3 _x)(PP _x)) na ekranu se pojavi 6. Važno da je program i proveran za formule oblika (fakt p q), gde p, q dati brojevi. Recimo, na pitanje ?((fakt 3 7)) dobije se odgovor ?, tj. ne. Provernost je prisutna, dobrim delom, blagodareći formuli (LESS _i _n).

Zadatak 3.28. (nastavak prethodnog) Neka je funk(n), n=1,2,3,..... ma koji niz definisan rekuzivno ovako

```
funk(1)=Poc
funk(i+1)=A(i,funk(i))  (i=1,2,...)
```

gde je Poc data početna vrednost i A zadana funkcija. Napraviti navišni program za računanje (i proveravanje) funk(n).

Rešenje. Zamisao je kao kod faktorijela u prethodnom primeru uz dva proširenja:
    Poc je ma koji zadan broj, i A ma koja funkcija.
Program glasi:

```
(Δ)
  ((fun _A _n _i _Priv _Rez) (LESS _i _n)(SUM _i 1 _i1)
                          (_A _i _Priv _Priv1)(fun _A _n _i1 _Priv1 _Rez))
  ((fun _A _n _n _Rez _Rez))
  ((fun _A _Poc _n  _Rez)(fun _A _n  1 _Poc _Rez))
```

Neka uz njega imamo i ova dva članka, koji definišu dve relacije a i b:

```
((a _x _y _z)(TIMES _x _y _z))
((b _x _y _z)(SUM _x _y _z))
```

Tada jedan primer korišćenja (Δ) je postavljanje pitanja

```
?((fun a 1 6 _x)(PP _x))
```

što praktično znači da hoćemo da se navedeni navišni algoritam obavi kad je _A upravo relacija a, i kad je Poc =1.
Nije teško videti, da je program (Δ) mnogo opštiji od svih dosadašnjih, jer eto u njemu je dozvoljeno da ulazna promenljiva bude relacija. Napomenimo da je takva mogućnost tipična za LISP, ali kao što vidite i u Prologu se se sme koristiti.

Međutim, istaknimo da je prethodni program pisan u Micro-prologu a da u drugim verzijama prologa ima nekih novih pojedinosti. Recimo, program(Δ) se se skoro bukvalno prenosi na LPA-prolog. I tada formula

```
(_A _n _i _Priv _Rez)
```

koja je, inače, najosetljiviji deo programa (Δ), jer njeno ime je promenljiva, se u LPA -prologu prosto bukvalno prevodi sa

```
(*)          A(N,I,Priv,Rez)
```

Međutim, u Arity-prologu to nije dozvoljeno. Bliže, pri prevođenju na Arity-verziju umesto (*) se stave ove dve formule

```
X=..[A,N,I,Priv,Rez],
call(X),
```

U tim formulama se koristi relacija =.. jedna od osnovnih relacija Edin-

burgske sintakse. Recimo, u prethodnom primeru

```
..[A,N,I,Priv,Rez]
```

je u stvari A(N,I,Priv,Rez), odnosno strogo rečeno promenljiva X se vezuje za tu uobičajenu relacijsku formulu. Evo još nekih primera sa tom relacijom

Y=..[b,c,d],	Ako je Y promenljiva[8] koja nije još dobila vrednost, onda se njoj dodeljuje vrednost b(c,d). A ako je Y već imala vrednost, onda vrednost te formule da ili ne.
p(q,r,s)=..Y,	Ako je Y promenljiva[9] bez vrednosti, onda se tom formulom njoj dodeljuje vrednost lista [p,q,r,s], a ukoliko Y već ima vrednost, vrednost te formule je da ili ne.
a(b,c)=..[a,b,c]	Vrednost te formule je da.

Zadatak 3.29. Fibonacci-ev niz fibo(n) se definiše ovako

```
(φ)     fibo(1)=1, fibo(2)=1,
        fibo(n+2)=fibo(n+1)+fibo(n) , n=1,2,3,...
```

Napraviti odgovarajući prološki program, za računanje i proveru ma kog člana niza.

Rešenje. Skoro se neposredno napravi prevod datih jednakosti i tako dođe do ovog programa (osnovna relacija je *fib*):

```
fib(1,1).
fib(2,1).
fib(N,K):-N>2,N1 is N-1,fib(N1,K1),
                N2 is N1-1,fib(N2,K2),
                        K is K1+K2.
```

Taj program je -u načelu- sposoban i da za dati n nađe odgovarajući član niza, a takođe i da proveri da li je neki broj član tog niza uz pretpostavku datosti njegovog indeksa. Recimo, takva su ova pitanja

```
?- fib(10,X).          ?-fib(23,67).
```

Rekli smo "u načelu"[10] jer navedeni program zbog dvojne rekurzije u drugom članku je veoma spor.

Međutim, moguće je napraviti i navišan fib-program. Evo jednog takvog

```
((fib _n _i _Priv1 _Priv2 _Rez) (LESS _i _n)
                        (SUM _Priv1 _Priv2 _Priv3) (SUM _i 1 _i1)
                                (fib _n _i1 _Priv2  _Priv3 _Rez))
((fib _n _n _Priv _Rez _Rez))
((fib 1 1))
```

[8] Tj. nepoznata.

[9] Tj. nepoznata.

[10] Naime, kad se recimo tim programom računa fib(10), onda se to svodi na dve odvojene računice fib(9) i fib(8). Tokom prve računice se naravno pojavi traženje fib(8), ali to se ne pamti, pa kad se nakon prve računice prede na traženje fib(8), onda se opet ponavljaju razni "stari" koraci. I tako slično se postupa i za sve druge podslučajeve.
Međutim, pomenimo da istu takvu manu imaju i svi drugi rekurzivni programi za jednakosti (φ) pisani u jednom od jezika Lisp, Pascal, C.


```
((fib _n _Rez)(fib _n 1 0 1 _Rez))
```

Kako se on može "doraditi" tako da bude sposoban i da generiše prvi argument relacije fib; recimo, da ume da nađe n --ako ga ima-- za koje važi (fib n 567)?

Zadatak 3.30. Neka je n dat prirodan broj i neka je koren celi deo njegovog kvadratnog korena. Recimo, ako n=5, onda koren=2. Jedan navišan algoritam za ispitivanje "prostosti" broja n je:
Redom za brojeve 2,3,...., koren proveriti da li su činioci broja n. Ako se to desi sa nekim od tih brojeva, onda algoritam se završava sa odgovorom Nije prost, a u suprotnom odgovor je Jeste prost.
Taj algoritam prevesti na Prolog.

Zadatak 3.31. Napisati prološki program za računanje vrednosti brojevnih +,*,- izraza datih kao liste. Primeri takvih izraza:

```
(2 + 3)     (4 - (6 + (7 * 8)))
```

Rešenje.

```
((vred _x _x)(NUM _x))
((vred (_x + _y) _z)(vred _x _x1)(vred _y _y1)(SUM _x1 _y1 _z))
((vred (_x - _y) _z)(vred _x _x1)(vred _y _y1)(SUM _y1 _z __x1))
((vred (_x * _y) _z)(vred _x _x1)(vred _y _y1)(TIMES _x1 _y1 _z))
```

Pomenimo da se u Edinburgškoj sintaksi taj program jednostavnije iskazuje, a dalje možemo ga lako proširiti u program za računanje vrednosti izraza koji takođe smeju da sadrže razne elementarne funkcije kao sin, cos i dr. Međutim, ima još jedna zanimljiva stvar. Kako da se dopusti mogućnost da izrazi imaju promenljive, tj. nepoznate kojima se dogovorno daje vrednost ? Tu su sada moguća dva slučaja:

Slučaj 1: Hoćemo da izrazi imaju svoje vlastite promenljive, različite od proloških, i recimo te "promenljive" su x,y,z (Pazite to nisu prološke promenljive, jer napred nemaju znak _)

Slučaj 2: U izrazima se pojavljuju prološke promenljive kao _x , _X, itd. i svakoj od njih se tokom algoritma po jedanput daje vrednost.

Evo najpre rešenja u Slučaju 2. Gornji program dopunimo ovim člankom

```
((vred _x _x)(VAR _x)(PP Daj vrednost za _x)(R _x))
```

Tada recimo na pitanje

```
?((vred (_x + (_y - _x)) _Rez)(PP _Rez))
```

ćemo usput biti pitani za vrednost od _x i za vrednost od _y, a na ekranu će se pojaviti poruka

```
Daj vrednost za ....
```

gde će se umesto.... pojaviti adresa promenljive (nije prejaka informacija ali bolja od nikoje).

Sada navodimo rešenje u Slučaju 1. Recimo, da hoćemo da za opisane izraze promenljiva bude x. Sada gornji program dopunjujemo ovim članicima, sa pomoćnim predikatom pamti:

```
                        Tako smo zapisali da x na samom početku još nema
((pamti x nema))        vrednost. Ali, ako recimo želimo da već ima vrednost
```

```
                    onda umesto članka ((pamti x nema))  uzmemo članak
                    ((pamti x 55)).

    ((vred x _Rez)(pamti x _X)(vidi _X _Rez))
    ((vidi nema _Rez)(PP Daj vrednost za x)(R _Rez)
                     (DELCL ((pamti x nema))) (ADDCL ((pamti x _Rez)))
                                                                    )
    ((vidi _X _Rez)(pamti x _Rez))
    ((pamti x nema))
```

Tu je malo složeniji članak sa glavom (vidi nema _rez) koji odgovara slučaju kad x još nije dobilo vrednost. Tada se prvo učita željena vrednost Rez i onda, uslugom u Prolog ugrađene relacije DELCL se "obriše" , skloni članak ((pamti nema)) a zatim se pomoću ADDCL doda nov članak ((pamti _Rez)), kojim se pamti pridružena vrednost za x.

Pomenimo još,da ako želimo da računamo vrednost više izraza i da pritom po želji menjamo vrednost za x onda se moramo pobrinuti da nakon svakog računa sklonimo članak oblika ((pamti x _Rez)) i umesto njega stavimo članak ((pamti nema)). U tu svrhu možemo uvesti ovaj dopunski članak za traženje vrednosti izraza

```
((vred1 _X _Y)(vred _X _Y)(KILL pamti)(ADDCL  ((pamti  x  nema))))
```

Tu se uslogom KILL-predikata uklanja pamti-članak, a posle se dodaje članak ((pamti x nema)).

Zadatak 3.32. Napisati proloski program za diferenciranje po x datog +,-, *,/,sin,cos,exp izraza, pisanih u obliku liste. Primeri takvih izraza su

```
45,    (x + (sin (x * 5))),    (exp (cos (4 - (1 / x))))
```

Rešenje.

```
((dif x 1))
((dif _X 0)(NUM _X))
((dif (_A + _B) (_P + _Q))(dif _A _P)(dif _B _Q))
((dif (_A - _B) (_P - _Q))(dif _A _P)(dif _B _Q))
((dif (- _A) (- _P))(dif _A _P))
((dif (_A * _B) ((_P * _B) + (_A * _Q)))(dif _A _P)(dif _B _Q))
((dif (_A / _B) (((_P * _B) - (_A * _Q)) / (_B * _B)))
                                          (dif _A _P)(dif _B _Q))
((dif (sin _A) (_P * (cos _A)))(dif _A _P))
((dif (cos _A)  (- _P * (sin _A)))(dif _A _P))
((dif (exp _A) (_P * (exp _A)))(dif _A _P))
```

Bez spora u Prologu, nezavisno od verzije, je nakraći program za diferenciranje.

Zadatak 3.33. Uočimo ovu opštu definiciju *-termova, građenih od a,b i binarnog operacijskog znaka:

(i) a i b su *-termovi
(ii) Ako su X i Y *-termovi, onda i reč (X*Y) je *-term.
(iii) *-term je samo ona reč koja se može dobiti konačnom primenom pravila (i) i (ii).

Napraviti proloski "prevod" te definicije.

Rešenje. U stvari, Prolog je skoro idealan za taj i takve zadatke, u osnovi rekurzivne. Koristeći relaciju imena *term* uočimo ove članke


```
(Δ)   ((term a))
      ((term b))
      ((term (_X * _Y))(term _X)(term _Y))
```

koji su skoro bukvalan prevod gornje definicije. Nije teško videti da je taj program sposoban da za ma koju listu L utvrdi da li jeste ili nije *-term. Recimo,na pitanje ?((term ((a * b) * a))) Prolog "rasuduje" ovako:

Računanje proloske vrednosti formule (term ((a * b) * a)) se svodi redom na računanje takve vrednosti redom ovih formula

(1) (term (a * b)) (2) (term b)

Dalje , vrednost prve od njih je konjunkcija vrednosti redom formula (term a), (term b). Obe te formule su tačne, pa je dakle tačna formula (1). Sada je na redu formula (2) koju smo u stvari već računali ali Prolog to ne pamti. I formula (2) je tačna , pa znači konačno odgovor na postavljeno pitanje je da.

U vezi sa programo (Δ) pomenimo i sledeće. Ako mu postavimo pitanje

```
?((term _X)(PP _X)FAIL)
```

on će stalno praviti sve nove i nove *-termove, ali u duhu Prologa, slobodnije rečeno, stalno će hvatati "što leviju granu", što ćemo bolje objasniti. Naime,*-termi se mogu ovako redati prema složenosti, odnosno broju znakova *:

```
                                a      b                    (Sa 0 zvezdica)
                (a*a) (a*b) (b*a) (b*b)                     (Sa 1 zvezdicom)
      (a*(a*a)) (a*(a*b)) (a*(b*a)) (a*(b*b))
        (b*(a*a)) (b*(a*b)) (b*(b*a)) (b*(b*b))
          ((a*a)*a) ((a*b)*a) ((b*a)*a) ((b*b)*a)
           ((a*a)*b) ((a*b)*b) ((b*a)*b) ((b*b)*b)           (Sa 2 zvezdice)
```

i tako dalje

I nije teško zaključiti da ako je Term ma koji zadan *-term,on će se tokom takvog redanja pojaviti u nekom koraku. Ali, Prolog odgovarajući na gornje gornje pitanje redom daje terme:

```
a
b
(a * a)
(a * b)
(a * (a * a))
(a * (a * b))
(a * (a * (a * a)))
(a * (a * (a * b)))
```

i slično dalje.

To praktično znači da "mimoilazi"mnoge terme. Recimo,nikada se neće pojaviti termovi (b * a), (b * b) i dr.
Napominjemo da se u Zadatku 9.1 upoznaje otklanjanje tog nedostatka.

Zadatak 3.34. Uočimo ovaj program

```
(β)        brisi(X,[X|Y],Y).
           brisi(X,[Y|Z],[Y|U]):-brisi(X,Z,U)
```

kojim se definise relacija *brisi*[1].

1. Da li se relacija elem(X,Y) (X je element od Y) može ovako definisati pomoću te relacije *brisi*: elem(X,Y):-brisi(X,Y,Z).
2. Da li relacija dodaj(X,L,L1) (L1 nastaje od liste L kad joj se X doda na početak) može ovako definisati: dodaj(X,L,L1):-brisi(X,L1,L).

Odgovor je da. Recimo, na pitanje ?-elem(2,[4,2,3]) odgovor će biti da, jer brisi (2,[4,2,3],Z) će se uspešno završiti (sa Z=[4,3]). Međutim, na pitanje ?-elem(2,[4,3,5]) odgovor će biti ne. Razlog: brisi(2,[4,3,5],Z) će se završiti sa ? jer relacija *brisi* nema članak oblika brisi(X,[],[]). Inače, da je taj članak uključen u definiciju relacije *brisi* relaciju *elem* ne bismo mogli definisati na navedeni način.

U vezi sa relacijom *dodaj* navodimo ovaj primer. Postavimo pitanje

```
?- dodaj(3,[5,7],X)
```

Ono se svodi na računanje vrednosti ove formule brisi(3,X,[5,7]). Sada primenom prvog članka iz (β), nakon unifikacije se dobije X=[3,5,7].

Zadatak 3.35. (Nastavak prethodnog) Definišimo ovu relaciju *redjaj*:

```
redjaj(L):-brisi(X,L,L1),write(X),write(' : '),write(L1),nl,fail.
```

Koji je odgovor na pitanje ?-redjaj([a,b,c,d]).

Odgovor. Dobiće se ovaj niz zapisa

```
a : [b,c,d]
b : [a,c,d]
c : [a,b,d]
d : [a,b,c]
```

Kao što se vidi, možemo tako reći, iz polazne liste se najpre vadi njen prvi član a i odvaja preostatak liste tj. podlista [b,c,d], pa se na ekranu pojavljuje gornji prvi red. Dalje, budući da u *redjaj*-definiciji se nalazi fail opet se dođe na formulu brisi(X,L,L1) i sada se desi "vađenje" drugog člana, tj. b, iz polazne liste, a L1 postane [a,c,d]. Sledstveno na ekranu se štampa

```
b : [a,c,d]
```

i slično se dešava dalje.
Istaknimo, da bi se uopšte , slobodnije pisano, na pitanje oblika

```
?-redjaj([a1,a2,...,an]).   (Naravno to nije konkretno pitanje)
```

redom pojavili zapisi

```
a1 : [a2,a3,...,an]
a2 : [a1,a3,...,an]
....
an : [a1,a2,...,an-1]
```

Zadatak 3.36. Napisati program koji pravi sve permutacije date liste.
Rešenje. Recimo, u slučaju liste [a,b,c] sve permutacije su sledeće

```
[a,b,c]   [a,c,b]   One sa a na prvom mestu
[b,a,c]   [b,c,a]   One sa b na prvom mestu
[c,a,b]   [c,b,a]   One sa c na prvom mestu
```

Primećuje se svojevrsna "rekurzija", koja bi se u slučaju ma koje liste [a1,a2,a3,...,an] slobodnije mogla ovako zapisati

```
Perm([a1,a2,a3,...,an])
        = [a1|Perm([a2,a3,...,an])]
          + [a2|Perm([a1,a3,a4,...,an])]
          + ...
          + [an|Perm([a1,a2,...,an-1])]
```

gde Perm(L) znači skup svih permutacija liste L, a znak + "glumi" skupovnu uniju.

Kao što se vidi u toj jednakosti sa desne strane je, moglo bi se tako reći, upletena relacija redjaj iz prethodnog zadatka.
Program koji sledi je u stvari proloski prevod gornje jednačine

```
brisi(X,[X|Y],Y).
brisi(X,[Y|Z],[Y|U]):-brisi(X,Z,U)
```

Prepisali smo definiciju relaciju *brisi* iz Zadatka 3.34.

```
perm([],[]).
perm(L,[A|B]):-brisi(A,L,L1),perm(L1,B)
```

Formula brisi(A,L,L1) odgovara korišćenju relacije *redjaj* prethodnog Zadatka.

Taj članak proloski zapisuje gornju rekurzivnu jednakost.

```
permutacija(L):-perm(L,Y),write(Y),nl,fail.
permutacija(L).
```

Dati program se uključuje pitanjem oblika ?-permutacija(L) , gde je L data lista. Prolog onda prvo računa formulu perm(L,Y) i upošljavajući perm-članke nalazi jednu permutaciju od L. Nakon štampanja dolazi se do fail-a, koji iziskuje vraćanje nazad odnosno traži se novo Y tako da važi perm(L,Y), tj. traži se nova permutacija od L, itd.
Recimo, na pitanje ?-permutacija([1,2,3]) dobiće se ovaj "ispis"

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

Pošto imamo i drugi permutacija- članak izbegli smo da se zbog fail-a na kraju pojavi i no (tj. ne).

Zadatak 3.37. Među brojevima 2,7,9,1,4 naći sve trojke x,y,z sa svojstvom x+y+z>13.

Rešenje. Koristimo metodološku ideju "generiši i proveri", izloženu u Zadatku 3.9. Tako, prvo ćemo na određen način, recimo pomoću predikata *dat* zapisati pretpostavku o datosti navedenih brojeva. Shodno tome uvodimo ovih pet *dat*- članaka

```
dat(2).
dat(7).
dat(9).
```

[1] Taj program je deo programa relacije *prvobris* u Zadatku 3.12

```
dat(1).
dat(4).
```

Dalje, program se pored njih sastoji iz ova dva članka

```
kreni:-dat(X),dat(Y),dat(Z),X+Y+Z>13,
                write(X),write(Y),write(Z),nl,fail.
kreni.
```

Program se uključuje pitanjem ?-kreni.

Napomena 3.4. Neka su Obj1,Obj2,....,Objn neki dati objekti, gde je n neki konkretan prirodan broj i neka je

Uslov(X1,X2,...,Xm) (m je neki prirodan broj)

neki zadan uslov (koji je smislen u vezi sa datim objektima). Tada, program oblika

```
dat(Obj1). dat(Obj2). ... dat(Objn).
kreni:-dat(X1),dat(X2),...,dat(Xm), Uslov(X1,X2,...,Xm),
                write(X1),write(X2),...,write(Xn),nl,fail.
kreni.
```

na pitanje ?-kreni. će ispisati sve m-torke (X1,X2,...,Xm) objekata Obj1, Obj2,...,Objn koji zadovoljavaju dati uslov U. Recimo, neka su a,b,c dati objekti. Tada, ako želimo sve njihove permutacije, za uslov U(X,Y,Z) možemo uzeti X\=Y,X\=Z,Y\=Z. To bi bilo direktna primena opisane ideje. Međutim, u svom tečenju algoritam će proći kroz mnoga "neplodna i nekorisna mesta", jer konačno treba da se naredaju sve varijacije

aaa, aab, ..., ccc

tih elemenata i onda za svaku od njih proveri gornji uslov. Podesnije je da se sada napravi ovakav članak[2]:

```
perm:-dat(X),dat(Y),X=\=Y,dat(Z),Z=\=Y,Z=\=X,
        write(X),write(Y),write(Z),fail.
```

jer onda pitanjem ?-perm. će se kraće i brže ispisati sve tražene permutacije. U stvari i uopšte pri primeni ideje "generiši i proveri" bolje je da ona pored generatornosti sadrži i pravovremenu provernost, blagodareći čemu će se svet mogućnosti bitno smanjiti.

Zadatak 3.38 Dati su lista brojeva L i jedan broj X. Listu L razdvojiti na dve podliste L1,L2 - u prvu ulaze oni članovi iz L koji su manji od X, a u drugu preostali.

Rešenje. Neka zapis razdvoji(X,L,L1,L2) znači L1,L2 su liste nastale iz L opisanim razdvajanjem po X. Tada imamo ovaj program

```
razdv(X,[],[],[]).
razdv(X,[Y|Z],[Y|U],V):-X>Y,razdv(X,Z,U,V).
razdv(X,[Y|Z],U,[Y|V]):-razdv(X,Z,U,V).
razdvoji(X,Y,Z,U):-razdv(X,Y,Z,U),!.
```

Zadatak 3.39. Koristeći se relacijom *razdvoji* iz prethodnog zadatka napisati proloski program koji odgovara poznatom *duick-sort* algoritmu za sre-

[2] Predikat =\= pri zapisu A=\=B ima smisao:
A,B imaju različite vrednosti.

divanje po veličini (sortiranje) date liste brojeva.

Rešenje. Quick-sort algoritam koristi ovu zamisao, kao polugu za rekurziju:

Neka je dat neki niz brojeva a(1),a(2),...,a(N) koji treba da sredimo dimo (rastući po veličini). Međutim, zamislimo takođe da taj niz ima ovo svojstvo:

U nizu ima jedan njegov član jednak M, a indeksa recimo i , tako da da članovi a(1),a(2),...,a(i-1) su manji M, a članovi a(i+1),...., su veći ili jednaki M. Tada imamo ,slobodnije pisano, ovakvu rekurzivnu jednakost

Sred(a(1),a(2),...,a(n))=
Sred(a(1),..,a(i)) + Sred(a(i+1),...,a(N))

gde Sred(a(1),...) znači "sred" (tj. rezultat sređivanja) niza a(1),... , a znak + "glumi" neku vrstu dopisa niza na niz.

Program koji navodimo upravo koristi tu osnovnu zamisao *duick-sort*-a, s tim da se niz zadaje listom, a da se za M uzima prvi član liste. Na početku se u odnosu na taj M lista razdvaja na podliste L1, L2, i to bukvalno primenom relacije *razdvoji* iz prethonog zadatka, a onda rekurzivno koristi navedena zamisao.

Program glasi

```
quick([],[]).
quick([X|Y],Z):-razdvoji(X,Y,Levi,Desni), quick(Levi,Levi1),
                        quick(Desni,Desni1),dopis(Levi1,[X|Desni1],Z).
```

gde *dopis* je relacija dopisivanje liste na listu:

```
dop([],A,A).
dop([A|B],C,[A|D]):-dop(A,C,D).
dopis(A,B,C):-dop(A,B,C),!.
```

Primer :

Na pitanje ?-quick([4,7,2,1,3,9],X). na ekranu će se pojaviti lista

[1,2,3,4,7,9].

Zadatak 3.40. Datu listu brojeva srediti praveći redom njene permutacije i odvajajući onu koja je sređena (*permutacijski način sredjivanja)*

Rešenje.

```
permsred(L,Rez):-perm(L,Rez), uredjena(Rez). Tu se pretpostavlja
                                korišćenje relacije perm iz Zad.3.36.
uredjena([]).
uredjena([X]).
uredjena([X|[Y|Z]]):-X=<Y,uredjena([Y|Z])Sa ta tri poslednja čla-
                        nka se definiše relacija uredjena(lista)
```

Zadatak 3.41. Na Prolog prevesti poznate algoritme sredjivanja: *bubble* i *umetački*

Rešenje. *b u bb l e* (tj. *mehurić)* algoritam. Najpre uočimo ove članke relacije *mena*:

```
mena([X,Y|Z],[Y,X|Z]) :-  X > Y.
mena([X|R],[X|R1])  :-   mena(R,R1).
mena([],[]).
```

Relacija *mena* je "zadužena" da u svom prvom argumentu, listi L zameni a sa

b ukoliko su a,b dva susedna člana liste L, sa svojstvom a>b, i uz to a,b su prvi takvi članovi liste L. Recimo, ako L=[2,4,3,1,5] onda formula mena(L,X) je tačna za X=[2,3,4,1,5]. Drugim rečima, *mena* je zadužena da u listi L ukloni prvu "menu"[3], odnosno deo oblika

...,a,b,...

gde a>b, i sa L prede na listu

...,b,a,...

gde tačkice označavaju nenavedene članove liste. Međutim ,ako lista L nema nijednu menu, tj. nikoja dva susedna člana a, b gde a>b, tada formula mena(L,X) je tačna za X=L. Imajući mena-članke proloski možemo ovako iskazati bubble-algoritam:

```
bub(L,S)  :-mena(L,L1), (L1=L,S=L;bub(L1,S)).
```

Smisao: Ako je L data lista tada bub(L,S) se računa ovako: pravi se mena(L,L1) i dobije se L1.Ako L1=L, posao je završen i rezultat je L, a u suprotnom treba nastaviti sa računanjem bub(L1,S).

```
buble(L,S):-bub(L,S),!
```

Recimo, na pitanje ?-buble([6,4,2,3],X) za X ćemo dobiti [2,3,4,6].

u m e t a č k i algoritam. On koristi ovakvu rekurzivnu ideju. Pretpostavimo da treba da sredimo niz

a(1),a(2),a(3),...,a(N).

Možemo ovako rasuđivati

Zamislimo da je podniz a(2),a(3),...,a(N) već sređen i da smo dobili niz b(2),b(3),...,b(N).Da bismo polazni niz sredili dosta je da član a(1) pravilno umetnemo u taj b-niz, tj. umetnemo ga tako da levo od njega bude manji a desno naredni veći ili jednak član tog b-niza.

U donjem programu za to umetanje je zadužena relacija *umetni*.

```
umetni(X,[Y|S],[Y|S1]) :- X > Y, umetni(X,S,S1).
umetni(X,S,[X|S]).
umetsort([],[]).
umetsort([X|R],S) :- umetsort(R,SR),  umetni(X,SR,S).
sredi(X,Y):-umetsort(X,Y),!.
```

Napomena 3.5. - U rešavanju mnogih problema, posebno u oblasti tzv. Veštačke inteligencije, često je podesno u mislima držati predstavu raznih drveta. I Prolog može da izađe u susret takvom načinu rasuđivanja, odnosno u njemu se može ostvariti ideja drveta. Na slici

je navedeno jedno (binarno) drvo kome

a je Srce
b je Leva grana
c je Desna grana

Da bismo u Prologu predstavili drveta uvodimo predikat drvo(X,Y,Z), gde će Y biti srce, X leva grana, a Z desna grana U nekim problemima je podesno

[3] Slobodnije rečeno, "mena" je shvaćena kao slučaj u kome prvo dođe veći,pa manji broj. Recimo, dvojka 7,6 je primerak "mene".

korišćenje drveceta (tj. najkraćih drveta),kakvi su b i c na gornjoj slici. Za njih ćemo dogovorno reći da su im i leva i desna grana konstanta nil.

I tako, recimo, gornje drvo se može proloski zadati ovim člancima

```
drvo(nil,b,nil).
drvo(nil,c,nil).
drvo(b,a,c).
```

Primetimo da se umesto ta tri članka može koristiti samo jedan:

```
drvo(drvo(nil,b,nil),a,drvo(nil,c,nil))
```

Sada navodimo neke zadatke sa drvetima.Kraj Napomene 3.5

Zadatak 3.42. (Uz Napomenu 3.5) Za dato drvo definisati njegovo srce, levi i desni deo.Takođe definisati relaciju : "neko drvo je drvce".

Rešenje.

```
srce(drvo(X,Y,Z),Y).
levi(drvo(X,Y,Z),X).
desni(drvo(X,Y,Z),Z).
jedrvce(X):-levi(X,nil),desni(X,nil).
```

Tu imamo prećutan dogovor da se konstanta *nil* koristi samo za pravljenje drveceta. Primera radi na navedena pitanja imamo navedene odgovore

```
Pitanje   ?-levi(drvo(p,g,h),X).                          Odgovor X=p
Pitanje   ?-srce(drvo(drvo(nil,g,nil),hh,j),X).           Odgovor X=hh
```

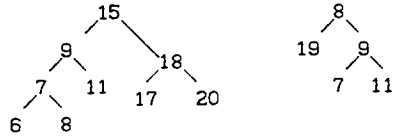
Zadatak 3.43 (Nastavak prethodnog) Napraviti neku prološku "proceduru" ispisivanja datog drveta

Rešenje.

```
drvoispis(X):-jedrvce(X),srce(X,Srce),write(Srce).
drvoispis(X):-write('('),levi(X,Levi),drvoispis(Levi),
              write(' '),srce(X,Srce),write(Srce),write(' '),
               desni(X,Desni),drvoispis(Desni),write(')').
```

Tim ispisom će se drveta pisati u obliku lista. Recimo, u Napomemi 3.5 navedeno drvo imaće ovaj ispis (b a c).

Napomena 3.6. Neka je D neko binarno drvo čiji svi "sastavci" su brojevi, recimo u ukupnosti to su a1,a2,...,aN. Dogovorno skup tih brojeva zvaćemo *skup*(D).Recimo kod prvog drveta



taj skup iznosi {15,9,18,7,11,17,20,6,8}. Uvodimo ovu definiciju *sredjenog* drveta:

Drvo D je sređeno ukoliko za svako njegovo poddrvo D1 važi ovakva nejednakost

(Δ) skup(Levi(D1)) <= Srce(D1) <= skup(Desni(D1))

gde zapis skup(Levi(D1))<= Srce(D1) znači da je svaki član skupa levog

dela od D1 manji ili jednak od srca od D1, a slično izražavanje važi i za drugu nejednakost u (Δ).

Recimo, prvo navedeno drvo je sređeno, a drugo nije. Pomenimo još da se u literaturi za sređena drveta obično koristi naziv B-drveta, tj. balansirana, uravnotežena. Videti i **Napomenu 3.7.** Kraj **Napomene 3.6.**

Zadatak 3.44. (Uz Napomenu 3.6) Od date liste brojeva napraviti bar jedno sređeno drvo.

Rešenje. Koristimo ovu rekurzivnu zamisao

> Ako je data lista L brojeva [a1,a2,.....,an] onda uzmemo joj prvi član a1, dalje pomoću njega *razdvojimo* članove liste na dve podliste L1,L2 gde u L1 dolaze članovi manji od a1, a u L2 preostali. Tada se drvo od L može zamisliti kao drvo srca a1, levog dela =drvo(L1),a desnog dela =drvo(L2).

Tu se koristi relacija *razdvoji*, koju smo prvi put sreli u Zadatku 3.38. Jedan program sa navedenom idejom glasi:

```
odrv([A|B],drvo(D1,A,D2)):-razdvoji(A,B,L1,L2),
                           odrv(L1,D1),odrv(L2,D2).
odrv([],[]).
odrvi(X,Y):-odrv(X,Y),!.
razdvoji(A,[G1|R1],[G1|T1],T2):-G1<A,razdvoji(A,R1,T1,T2).
razdvoji(A,[G1|R1],T1,[G1|T2]):-razdvoji(A,R1,T1,T2).
razdvoji(A,[],[],[]).
```

Znači glavna relacija je *odrvi*, koja je zadužena da od date liste brojeva napravi sređeno drvo.

Zadatak 3.45. (Nastavak prethodnog) Koristeći ideju sređenih drveta napraviti prološki program za sređivanje liste brojeva.

Rešenje. Zamisao je ova. Data je lista L. Prvo, koristeći relaciju *odrvi* iz prethodnog zadatka napravimo sređeno drvo X. Dalje, relacijom *ulisti* koju definišemo dole od drveta X se pravi lista Y, i ona je traženi rezultat. I tako, uz korišćenje članaka iz prethodnog zadatka imamo još ove članke

```
sredi(L,Y):-odrvi(L,X),ulisti(X,Y).
ulist(drvo(D1,A,D2),L):-
     ulist(D1,L1), dodaj(L1,[A],Pomoc), ulist(D2,L2), dodaj(Pomoc,L2,L).
ulist([],[]).
ulisti(X,Y):-ulist(X,Y),!.
    dodaj([],A,A).
    dodaj([A|B],C,[A|D]):-dodaj(B,C,D).
```

Tu se koristi dobro poznata relacija *dodaj* kojom se jedna lista dopisuje drugoj. Recimo, na pitanje ?-sredi([5,7,3,2,4],X) dobiće se X=[2,3,4,5,7].

Napomena 3.7. Ističemo da Arity-prolog ima ugrađena sređena drveta, tzv. B-drveta, o čemu podrobnije govorimo u delu 9.3.

4. JOŠ O FORMULAMA, ČLANCIMA

4.1 Slučaj Lisp-sintakse

U ovom izlaganju će biti nekih suptilnijih pojedinosti o formulama i člancima u slučaju Micro-prologa, odnosno Lisp-sintakse u kojoj se sve izražava jedinstveno na jeziku drveta i korisnik, u stvari, može koristiti taj jezik -sa svim slobodama koje su pritom prisutne. S druge strane i ako "unutra" i Prolozi Edinburgške sintakse rade sa sličnim drvetima korisnik ih direktno ne može upotrebiti već je sintaksno veoma sputan. Inače, o člancima te sintakse biće reči u delu 4.2 ove tačke.

Ranije smo za znak | rekli da je "list constructor", tj. da se pomoću njega postupno grade liste. Znak | shvatamo kao znak binarne operacije, i pišemo ga infiksno (recimo, ne pišemo |(a,b) što bi inače bio prefiksan način već (a|b)). Liste su potpuno strogo rečeno jedan deo, jedan podskup svih |-termova. Evo stroge definicije tih termova:

(4.1.1)
- (i) Term-jedinke su |-termovi.
- (ii) Ako su A,B |-termovi, onda i reč (A|B) je |-term.
- (iii) |-termovi su samo one reči koje se mogu dobiti konačnom primenom pravila (i),(ii) ove definicije.

U toj definiciji "term-jedinka" znači:

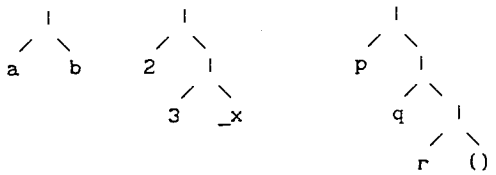
> ma koji prost proloski objekat :
> brojevi, promenljive i konstantske reči,
> dalje, ime ma kog u Prolog ugrađenog predikata, kao :
> PP, R, LOAD, SAVE, INT, CON, SUM, ...
> a takodje i jednu posebnu konstantu, tzv. praznu listu () .

Umesto da kažemo "ime u Prolog ugrađenog predikata" često ćemo kraće reći: "sistemska reč".

Kao što vidite koristili smo Lisp-sintaksu, a krajnje je jednostavan prelaz na Edinburgšku sintaksu. Naime, tada se umesto malih zgrada),(koriste uglaste],[i još kod lista umesto "belina" za razdvajanje članova liste koristi se znak zapete. Tako, recimo lista (a b c) se piše u obliku [a,b,c].

Evo primera |-termova (a|b), (2|(3|_x)), (p|(q|(r|()))). Njih sasvim prirodno možemo prikazati sledećim binarnim drvetima

(4.1.2)



Takva drveta kako već rekosmo u Prologu imaju osnovnu ulogu. Istaknimo odmah da se ideja takvih drveta prvo koristila u jeziku Lisp, a da je Prolog, jezik oko 10 godina mladi od njega, prihvatio istu ideju. U vezi sa drvetima za sada pomenimo dva važna pojma car, cdr -inače preuzeta iz Lispa. Naime, ako je D neko drvo (tj. |-term) oblika (A|B), onda njegovi car, cdr

su redom delovi[1] A, B. Dakle:

car((A|B)) = A, cdr((A|B)) = B

Za |-term ćemo reći da je složen(prost) već prema tome da li ima, nema neki znak |. Prost |-term je dakle term-jedinka, a složen term ima oblika (A|B).

Liste, kako rekosmo su posebni |-termi. Tako, jedino poslednji naveden |-term u (4.1.2) je lista. Slobodnije rečeno, liste su |-termovi sa posebnim rasporedom zagrada i koji "na kraju" imaju i praznu listu (). Podrobnije rečeno liste su ma koji |-termovi jednog od oblika

```
()
(A1|())
(A1|(A2|()))
(A1|(A2|(A3|())))
    ...i slično dalje.
```

Tu su A1, A2, A3, ... ma koji |-termovi. Navedene liste, od druge nadalje, se dogovorno ovako redom drukčije zapisuju

```
(A1)
(A1 A2)
(A1 A2 A3)
... i slično dalje.
```

Znači, drugim rečima, po definiciji imamo ovakve jednakosti :

```
(A1)      = (A1|())
(A1 A2)   = (A1|(A2|()))
(A1 A2 A3)= (A1|(A2|(A3|())))
            ...i slično dalje.
```

Ako je (A1 A2 .... An) uopšte lista od n-članova A1, A2, ..., An onda imamo ove opšte jednakosti (istićane ranije)

```
(A1)=(A1|())
(A1 A2 ... An)=(A1|(A2 ... An))
```

Pomenimo, da se u Prologu obično pored takvog lista-zapisa

(A1 A2 ... An)

koriste i malo opštiji u kojima umesto An stoji An|X, odnosno zapisi oblika

(A1 A2 ...An|X)

Oni se uvode sledećim dogovorima

```
(A1|X)       = (A1|X)
(A1 A2|X)    = (A1|(A2|X))
(A1 A2 A3|X) = (A1|(A2|(A3|X)))
 i slično dalje.
```

Neka je (A |X1) neki takav zapis. Ukoliko je X1 neki složen |-term oblika (B|X2), onda se taj zapis može ovako "produžiti" (A B|X2). Naravno, ako je X2 složen |-term, recimo oblika (C|X3), onda imamo ovo novo produženje (A B C|X3), i slično dalje. Uopšte imamo ovo tvrđenje:

[1] Ponekad se u radu sa opisanim (binarnim) drvetima umesto CAR, CDR kaže i levi, odnosno desni rep.

(4.1.3) Svaki složen |-term je jedinstveno izraziv u obliku

(A1 A2 ... An|X)

za neko n=1, 2, ..., neke |-termove A1, ..., An, i neko X, ali koje je prost |-term.

Dogovorno ćemo zapise oblika (4.1.3), uz navedene uslove, zvati uopštene liste. Za A1, A2, ..., An ćemo reći da su njeni članovi, a za X da joj je kraj Primetimo da se prethodno tvrđenje može i ovako prirodno iskazati:

Svaki složen |-term je jedinstveno izraziv kao uopštena lista.

U skladu sa tim kad god nam bude podesno neki poznat složen |-term ćemo posmatrati u obliku (4.1.3).
Inače, ako u (4.1.3) ne postavimo zahtev "prostosti" X, tada jednom datom |-termu može odgovarati više takvih oblika (4.1.3), sa raznim n.
Odmah istaknimo, da se u skladu sa navedenim tvrđenjima u Prologu uopšte često koriste zapisi oblika (4.1.3), sa prostim ili složenim X.

U narednom koraku na jeziku |-termova, tj. u Lisp-sintaksi upoznajemo najopštiji oblik formule i članka, koji predstavljaju osnovne sastavke(delove) te sintakse. Naime, prvo smo na samom početku tačke 1 (vid. (1.2)) zapise oblika

(*rel* a1 a2 ... an) (rel je neka konstantska reč)

nazvali formulama. Može se reći da su to liste ovog oblika

(*rel*|X)

gde je X=(a1 a2 ... an). S tim u vezi u Micro-prologu

(4.1.4) Pod formulom smatramo ma koji |-term oblika (A|B), gde je A tzv. ime formule, neka konstantska ili sistemska[2] reč. Takodje dogovorno i znake / i FAIL uključujemo u takve formule.

To je naravno mnogo opštije od (1.2), ali Micro-prolog upravo koristi takav širi pojam formule. Znači, drugim rečima neki |-term je formula upravo ukoliko njegov car je konstantska ili sistemska reč.

Uz korišćenje definicije (4.1.4) sa (2.1.1) je određen opšti pojam članka. Znači, kratko rečeno:

Svaki članak ima oblik (for1 for2 ... fork), tj. on je lista izvesnih formula

Nije teško videti da u takvoj listi car(car(članak)) je ime formule for1, dakle to je neka konstantska ili sistemska reč. To i posebno ističemo:

(4.1.5) Neophodan uslov da neki |-term Term bude članak je da njegov car(car(Term)) bude konstantska ili sistemska reč.

Ta je činjenica od velikog značaja za Micro-prolog, jer u njemu pri zadavanju članaka, tj. pri pisanju programa za članke se ne proverava da li su liste formula već jedino se proverava ispunjenje tog neophodnog uslova. Recimo, Micro-prolog će prihvatiti i ovakve zapise kao članke

(4.1.6) ((a|2)|3), ((radi _x)|_x) i dr.

[2] To se odnosi na formule ugrađene u Prolog.

Zasto je tako nesto dopusteno u Micro-prologu objasnicemo ubrzo. Sada pomenimo, da smo drugom članku namerno dali ime radi. Naime, osnovni predikat svakog Prologa je ? i njegova definicija je slična tom radi-članku, odnosno bukvalno ovako glasi

(4.1.7) ((? _x)|_x)

što se lako saznaje iz Micro-prologa, jer kad udemo u taj jezik i postavimo pitanje &LIST ? pojaviće se upravo navedena definicija (jedino će se možda umesto _x pojaviti koja druga promenljiva).

Može se odmah pomisliti kako Micro-prolog može biti "pouzdan", kad dopušta toliku slobodu pri prihvatanju članaka? Upravo u skladu sa tom "slobodom" tokom prološkog algoritma se može pojaviti poruka o nekoj grešci koja ukazuje da se pojavio slučaj u kome je tekući članak postao nemoguć za dalji prološki "račun", i tada se, naravno, algoritam prekida.

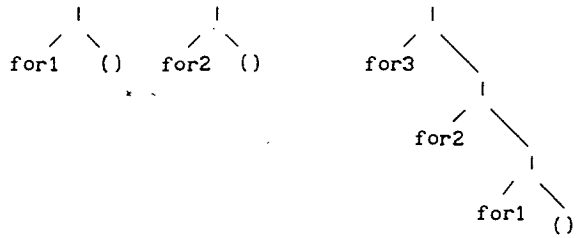
Naime , u svakom koraku prološkog algoritma, može se tako reći, Prolog na svoj način računa "parčiće", "delove" pojedinih članaka i, kraće rečeno, pobuni se, odnosno stane ukoliko takvo "parče", odnosno njegova trenutna vrednost nije formula u smislu (4.1.4) ili jeste, ali kojoj ne odgovara nikoji članak programa. Da bismo to bolje razumeli uočimo sledeći mali shema-program

```
(Δ)          (for1)
             (for2)
             (for3 for2 for1)
```

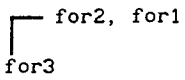
gde su for1,....,for3 neke formule. Rekli smo da je to shema-program, jer for1,...,for3 nisu neke određene formule. Tako, jedna mogućnost je da su to redom ove formule (a 2),(b 4 5),(c 7). Naravno, tada gornji shema-program prelazi u ovaj pravi program

((a 2)) ((b 4 5)) ((c 7)(b 4 5)(a 2))

I tako uočimo program (Δ) i , da ne bismo mnogo pričali o unifikaciji (jer to trenutno nije bitno), pretpostavimo da formule for1,..,for3 nemaju nikakvih promenljivih. Naravno tada se unifikacija svodi na običnu jednakost. Članke programa (Δ) prikažimo sledećim shema-drvetima (jer for1, for2, for3 su neka drveta za sebe)



Poslednje drvo označimo sa D3 i nacrtajmo ga i ovako



(ΔΔ)

Postavimo sada pitanje ?(for3), tj. potražimo prološku vrednost za formulu for3. Naravno, to je trivijalno, ta vrednost je da. Međutim, propratimo odgovarajuce prolosko rasudivanje na jeziku drveta:

> Zadatak je izračunati for3. Tražimo članak čija glava je ujednačiva sa for3. To je naravno treći članak, i pazite na drvetu tog članka, tj. na D3, formula for3 je njegov car. Sada se računanje vrednosti za for3 može ovako ispričati:
>
> > Od D3 pređemo na njegov cdr, tj. na cdr(D3), pa onda treba da izračunamo car od tog novog kraćeg drveta, tj. da izračunamo for2. Pošto imamo članak (for2), tj. elementarnu aksiomu to je for2 tačna. Idemo dalje, odnosno od cdr(D3) pređemo na cdr(cdr(D3)) i u narednom koraku treba da izračunamo car od tog drveta, tj. da izračunamo for1. Zbog članka (for1) ta formula je tačna. Da li sada opet treba još jednom primeniti cdr, pa posle računati car od novog drveta , itd? U stvari, pošto smo u prethodnom koraku došli do praznog drveta (), to ne nastavljamo na naveden način. U stvari, algoritam se završava i formula for3 je tačna.
>
> Primetite, što je bitno, "hodanje udesno" po for3-članku se završava kad se pojavi prazno drvo. Sada se jasno može uvideti da se prazno drvo pojavilo blagodareći okolnosti da članak ima zagrade "poveznice".

Naravno, prethodno rasudivanje se može podesno pratiti i na drvetu (ΔΔ), koje valja shvatiti kao drvo D3, doduše malo tehnički prerađeno.

Zamislimo sada da se u programu (*) poslednji članak zameni ovim

(for3 for2 for1 888)

Tada pri odgovaranju na pitanje ?((for3)), slično prethodnom, prvo će se računati for2, pa for1 i onda će na pozornicu doći 888. Međutim, tada će algoritam stati sa ovom porukom: CONTROL ERROR. Razlog je što 888 nije nikakva formula.

Možda ste već pomislili kakva je uopšte korist od dopuštanja da članci budu "toliko slobodni". Glavna korist je

> U "člancima" mogu pojedini delovi da budu i promenljive, ali koje kad na njih dođe red imaju ispravne[3] vrednosti. Takve "članke" nazivaćemo meta-članci. Ubrzo ćemo upoznati razne značajne takve članke.

Evo jednog takvog primera. Uočimo program

```
((a 1)(PP 1)) ((a 2)(PP 2))
((b 11)(PP 11)) ((b 33)(PP 33))
((c)(PP Sta da uradim) (R _x) _x)
```

U njemu jedino je treći članak meta-članak, jer njegovi sastavci su formule (c), (PP Sta da radim), (R _x), ali takodje i neformula _x, odnosno promenljiva. Pretpostavimo da smo postavili pitanje ?((c)). Evo šta se dešava

Na ekranu se prvo pojavi poruka Sta da radim, a dalje zbog R (tj. read) predikata se od nas očekuje da _x-u zadamo vrednost. Ako zadamo 88 pojaviće se poruka CONTROL ERROR, jer ta vrednost 88 nije prološka formula. Ali ako zadamo vrednost: (b 11) , onda Prolog pristupa računanju te

[3] Tj. prološki izračunljive.

formule, pa se sledstveno na ekranu pojavi 11.

Nešto kasnije ćemo još više i dublje govoriti o meta-članicima. A sada najpre navodimo jednu bitnu činjenicu koja ukazuje kad se sve Micro-prolog stavlja u pogon da odgovori na neko pitanje, jer kao što ćemo videti ne mora svako pitanje početi sa ?. Naime, u Micro-prologu se u nekim slučajevima pitanje može postaviti i bez usluge osnovnog predikata ? . Tako, pretpostavimo da smo nekim člankom oblika

((uno A) ....)
(A je neki |-term, a tačkice znače neke izostavljene delove)

definisali unarnu[4] relaciju uno. Tada umesto pitanja oblika ?((uno B)) možemo pitati ovako: uno B. Evo konkretnih primera:

(j1) Ako imamo članak ((a 1)(PP Pera)(PP Mile)) onda na pitanje
&a 1
dakle bez upitnika, a & je već sam prisutan kao znak Micro-prologa na ekranu će se štampati reči Pera, Mile u dva reda.
Ako u vezi sa istim člankom pitamo
&a 2
dobićemo odgovor ?, tj. ne.

(j2) Relacije LOAD, SAVE i druge su unarne. Zato, recimo, ako hoćete da učitate program imena PERA, onda to možete učiniti ili ovako

&?((LOAD "PERA.LOG")) ili kraće ovako &LOAD PERA

Doduše tu ima nekih malih sintaksnih dogovora, po kojima u prvom obliku pitanja se ime fajle daje celo i stavlja pod navodnike (kao reč, kao string), a u drugom se ime navodi bez navodnika i bez produžetka .LOG tipičnog za fajle Micro-prologa.

Sada je prilika da uvidimo pravu snagu meta-članaka. U tu svrhu uočimo definiciju ? predikata, tj. (4.1.7). U skladu sa okolnošću da je ? unaran smemo ga "uključivati" sa ? Nesto, ali šta mora biti to Nesto ? Neka recimo, imamo ovaj mali program ((a 1)(PP Jedan)). Da li kao Nesto smemo uzeti (a 1), tj. da li je ispravno ovo "uključivanje":

? (a 1)

Da bismo to raspravili setimo se definicije (4.1.7) i u nju umesto _x zamenimo (a 1). Dobićemo ovaj |-term ((? (a 1))|(a 1)), čije drvo D se može ovako prikazati

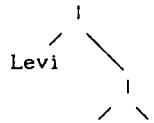
(Levi je (? (a 1)), a Desni je (a 1))

Znači, pošto (? (a 1)) je car tog drveta D, kao naredno treba da računamo car(cdr D), tj. car(Desni), što iznosi a. Ali, budući da taj car nije nikakva formula algoritam će stati sa porukom CONTROL ERROR.
Kao iduće pokušajmo da kao Nesto stavimo ((a 1)), tj. da "uključivanje" obavimo ovako

?((a 1))

što očigledno odgovara uobičajenoj upotrebi predikata ?. Tada ćemo dobiti ovaj |-term ((? ((a 1)))|((a 1))), čije drvo D se može ovako prikazati

[4] Tj. dužine 1.



(Levi je formula (? ((a 1))), a Desni formula (a 1))

Sada je već sve drukčije. Naime, kako (? ((a 1))) je car(D), to treba da računamo car(cdr(D)), tj. da računamo (a 1), što je proloski smisleno i što će uzrokovati da se na ekranu pojavi reč Jedan. Iduci korak je da od drveta D predemo na cdr(cdr D)) i ukoliko nije prazno drvo (prazna lista) da mu računamo car. Međutim cdr(cdr(D)) je upravo (), pa se čitav algoritam završava.

Pominjemo da ćemo slično kao i do sada radi lakšeg izražavanja umesto "rep nekog članka" govoriti i sastav, jer se za rep članka može slobodnije reći da predstavlja sastav (konjunkciju) formula koje treba redom sračunati.

Sada u Primerima 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5 navodimo značajne meta-članke pomoću kojih su u Micro-prologu definisane neke njegove "sistemske" relacije.

Primer 4.1.1. Predikat OR (tj. ili) se u Micro-prologu definiše sa ova dva proloska meta-članka

((OR _A _B)|_A)
((OR _A _B)|_B)

Dosta je važno steći iskustvo u korišćenju takvih Micro-proloških meta-članaka u smislu:

odmah znati koja od promenljivih mora dobiti kakvu vrednost, koja je uglavnom ili formula ili rep članka.

Recimo, prvi se može ovako shvatiti (For|Rep), gde For je upravo formula (OR _A _B) i taj deo nije sporan, a Rep je promenljiva _A. Prvo je odmah jasno da za _A kao vrednost ne smemo uzeti neku formulu, kao (for|...), jer u protivnom slučaju kad od prvog članka, shvaćenog kao drvo predemo na njegov cdr dođemo do (for|...) čiji car iznosi for, što je proloski neizračunljivo. Druga mogućnost je da za _A uzmemo neki sastav kao

(*) (for1 for2 ... fork)

gde su for1, for2, ..., fork izvesne formule. Tada prvi meta-članak postaje

(For|(for1 for2 ... fork))

tj. (For for1 for2 ...fork) što je očigledno proloski smisleno, jer razni car-ovi su sada for1, for2, ..., fork dakle neke formule. U skladu sa rečenim smisleno je recimo ovo OR-pitanje:

?((OR ((PP Jedan)(PP Dva)) ((PP Nije)(PP Jok))))

jer imamo ova davanja vrednosti:

_A--> ((PP Jedan)(PP Dva)) _B--> ((PP Nije)(PP Jok))

Pri odgovaranju na pitanje prvo upošljavamo prvi članak i sledstveno čeka nas računanje _A, tj. računanje njegovih "car-ova" dok se stigne do (). To praktično znači da će se u dva reda ispisati reči Jedan, Dva i time će biti

završeno odgovaranje na pitanje. Formalno, konačni odgovor je jeste. Primetite da drugi članak nije ni korišćen jer, možemo tako reći, formulu pitanja smo uspeli da dokažemo pomoću prvog članka[5]. Pretpostavimo sada da program sadrži članke

```
((a 1)(PP Pera))
((a 1)(PP Dara))
((b 2))
```

i da onda postavimo ovo OR-pitanje:

```
?((OR ((a 1)(b 4)) ((PP Ovde sam))))
```

Sada će se opet prvo koristiti prvi OR-članak što sledstveno znači računanje ovog repa ((a 1)(b 4)). Prvo će se štampati reč Pera, a posle kad se stigne do (b 4), budući da je to netačno, ponovo će se dokazivati (a 1), tj. štampaće se reč Dara. Posle toga opet se dolazi na (b 4), i pošto više nema nema nikakvih novih mogućnosti za dokaz (b 4) zaključujemo da do sada formula (OR _A _B) još nije dokazana. Budući da smo koristili samo prvi OR-granu sada prelazimo na drugu, tj. sada računamo ovaj rep ((PP Ovde sam)). To praktično znači da sada se još štampaju reči Ovde sam i čitav algoritam se završava potvrdno.
Već na osnovu tih primera vidimo kako uopšte deluju OR-članci i kakvu vezu imaju sa logičkim veznikom ILI. Naime :

> Ako se tokom prološkog algoritma dođe na računanje formule oblika (OR _A _B), onda --slobodnije rečeno-- uradi se sastav _A, tj. izračunaju mu se redom car-ovi. Tada ako se konačno dobije vrednost da, formula (OR _A _B) je izračunata sa vrednošću da , a ako se pri računanju _A konačno dobije ne, onda se radi traženja (OR _A _B) upošljava druga OR-grana, tj. onda se računa sastav _B.

Na osnovu izloženog primera izvlačimo ovu pouku:

(4.1.8) Ako meta-članak izgleda (For|_V), gde je For izvesna formula, onda razni sastavi su ispravne vrednosti za promenljivu _V.

Primer 4.1.2. U Micro-prologu u vezi sa čuvenim logičkim sklopom IF ... THEN ... ELSE se sledećim meta-člancima uvodi predikat IF

```
((IF _A _B _C) _A / |_B)
((IF _A _B _C)|_C)
```

Odmah recimo, da ti članci u velikoj meri pokrivaju ovu opštu IF-THEN-ELSE zamisao: Ako važi uslov _A onda uradi _B, a ako ne važi _A onda uradi _C. Podrobnije rečeno ti članci ovako "delaju":

> Pretpostavimo da se tokom prološkog algoritma pojavi potreba računanja neke formule oblika (IF _A _B _C). Tada, da bismo izračunali (IF _A _B _C) prvo treba da izračunamo deo _A, (koji očigledno mora biti formula)a dalje, ako je _A tačno, treba računati sastav _B. Znači drugim rečima, ako važi _A onda uradi _B. Ali, šta raditi ako je _B netačno.

[5] S tim u vezi pomenimo ovu zanimljivost. Da smo umesto postavljenog pitanja dali ovo ?((OR ((PP Jedan)(PP Dva)) 888) Micro-prolog bi dao odgovor isti kao u polaznom pitanju, jer:
Budući da neće biti primoran da koristi drugi članak, i "ne primećuje" da _B ima neprikladnu vrednost.

> E tada, u prvom članku u dejstvo stupa rez /, koji zabranjuje da pređemo i na drugi IF-članak.
> Međutim, ako _A nije tačno tada da bismo izračunali (IF _A _B _C) moramo koristiti drugi IF-članak, tj. uraditi sastav _C.

Kao što se vidi smisao formule (IF _A _B _C) je stvarno u velikoj meri u vezi sa IF-THEN-ELSE logičkim sklopom. Neka program P sadrži ove članke

```
((a 1)(PP Jedan))
((a 1.5)) ((a 2))((a 3))
((b 2)) ((b 3)) ((b 4))
```

Tada na pitanje ?((IF (a _x) ((PP Da)) ((PP Ne)))) imamo ovakvo dešavanje: Prvo se proverava uslov (a _x) -- sa nekim _x. Kao prva vrednost za _x se dobije 1. Formula (a 1) je tačna, jedino "platimo" štampanjem poruke Jedan na ekranu. I tako, budući da je uslovni deo IF-sklopa ispunjen treba izvršiti njegov deo _B, tj. ovde deo: ((PP Da)). I to se ubrzo završava štampanjem reči Da. To je kraj odgovaranja na pitanje; sam formalan rezultat je da (tj. tačno). A sada prepostavimo da hoćemo da pitamo

> Ako za neki _x važe (a _x) i (b _x) onda štampaj takvo _x, a inače štampaj reč Nema .

Tu je sada problem što proverni uslov treba da bude sastav ((a _x)(b _x)), a tako što nije dozvoljeno jer u IF-sklopu njegov _A-deo mora biti f o r m u l a. Odmah ćemo videti da postoje dva moguća izlaza. Jedan je da programu P dodamo još i članak

```
((c _x)(a _x)(b _x))
```

i onda pitamo na dozvoljen način

```
?((IF (c _x) ((PP _x)) ((PP Nema))))
```

Međutim, ima načina i bez unošenja nove relacije. To je veoma opšti način, koji je principijelno važan. Taj način koristi uslugu predikata ? (ili radi -članka (4.1.6) ako je uključen u program). Naime, pitanje sa opisanim sadržajem možemo ovako postaviti

```
?((IF  (? ((a _x)(b _x))) ((PP _x)) ((PP Nema))))
```

Šta će se desiti? Prvo, _A-deo je

```
(? ((a _x)(b _x)))
```

je formula i Prolog se neće "pobuniti". Dalje, zaputiće se da izračuna tu formulu. U tu svrhu upošljava ?-članak: ((? _X)|_X), koji u sadašnjim okolnostima postaje

```
((? _X)|_X)= ((? Sastav)|((a _x)(b _x)))   Tu , iz razloga preglednosti, reč
                                           Sastav stoji umesto ((a _x)(b _x))
           = ((? Sastav)|((a _x)|((b _x)|()))) Zamenili smo ((a _x)(b _x))
                                               sa ((a _x)|((b _x)|()))
           = ((? Sastav) (a _x)(b _x))
```

To praktično znači da radi izračunavanja gornjeg _A-dela treba da izračunamo (a _x), a potom (b _x). Naravno, to se ubrzo završi i _x dobije vrednost 2 , jer to je prvo _x koje zadovoljava i (a _x) i (b _x). Izračunavši _A-deo vidimo da je on tačan uz prihvatanje zamene _x-->2. Dalje, nas čeka

_B-deo tj. sastav ((PP _x)) pa sledstveno na ekranu se štampa 2 i time se završava algoritam. U tom primeru upotrebe predikata ? se krije sledeca opšta zamisao:

(4.1.9) Ako je S ma koji sastav(rep nekog članka) tada preslikavanje ϕ određeno sa

$$\phi(S)=(?\ S)$$

jeste preslikavanje "sveta" sastava u "svet" formula.
Dalje, to preslikavanje ima ovo opšte svojstvo:

Uraditi sastav S, tj. redom izračunati odgovarajuće car-ove, je proloski ekvivalentno sa Izračunati formulu $\phi(S)$.

To praktično, pored ostalog, znači ako se po prirodi stvari nekoj promenljivoj kao vrednost mora dati formula, a nama opet je potrebno da se uradi niz stvari, tj. formalno rečeno izračuna neki sastav S, onda možemo toj promenljivoj kao vrednost dati (? S).

Primer 4.1.3. Relacija NOT , tj. ne se u Micro-prologu definiše sa ova dva meta-članka

```
((NOT|_X) _X / FAIL)
((NOT|_X))
```

Da bismo ih lakše "skrozirali" uočimo program P koji sadrži ova četiri članka

```
((a 1))    ((a 5 6))
((b 3))    ((b 6 8))
```

i zamislimo da hoćemo da postavimo pitanje oblika ?((NOT|Nesto)), gde će umesto Nesto stajati "ono što sme". Budući da u prvom članku iza glave stoji _X zaključujemo da to Nesto mora biti formula. Sledstveno, postavimo ovo pitanje

```
?((NOT|(a 1))
```

i pratimo šta ce se dogadati. U skladu sa prvim člankom čeka nas računanje ovog sastava: (a 1) / FAIL. Pošto je (a 1) tačno idemo udesno na /, njega preskačemo i dođemo do FAIL, po definiciji netačnog. Dakle moramo nazad (procedura vracanje, backtracking). Ali, / nas "ometa", odnosno u skladu sa pravilom o rezu (vid. deo 2.3) algoritam se završava sa konačnim odgovorom ne. Već sada primetimo dve stvari. Prvo, u skladu sa definicijom lista umesto (NOT|(a 1)) smo mogli pisati (NOT a 1). Uopšte, ako je (ime a1 a2 ...an) neka formula, onda umesto (NOT|(ime a1 a2 ... an)) se može pisati ovako (NOT ime a1 a2 ... an). Drugo, formula (a 1) --koja je bila vrednost za _X- je tačna, a njen NOT, tj. formula (NOT a 1) je netačna. Glavni krivac je splet / FAIL. U stvari, uopšte ako je for neka tačna formula tada je formula (NOT|for) netačna.
Postavimo, sada, ovo pitanje ?((NOT a 7 6)), tj. pitanje ?((NOT|(a 7 6))). Tada, upošljavajući prvi NOT članak dolazimo do računanja ovog sastava (a 7 6) / FAIL. Rezultat je ne. Stoga upošljavamo drugi NOT-članak i zaključujemo da je da odgovor na postavljeno pitanje. U stvari, već i preko tog malog primera se vidi da ako je for neka netačna formula onda formula (NOT|for) je tačna. I tako kratko:

> vrednost formule oblika (NOT|_X) je upravo negacija vrednosti formule _X.

Evo još jednog malog primera koriscenja NOT predikata. Pretpostavimo, da hocemo da na ekranu ispisemo sva rešenja relacije a, kao i sva rešenja relacije b. Kao što znamo na pitanje oblika ?((a|_X)(PP _X)FAIL) ispisace se sva rešenja relacije a, odnosno ispisace se ove dve liste (1),(5 6). Ali, pored toga na ekranu cemo imati i ? kao znak da je završni odgovor ne. I tu je sada problem: kako se ratosiljati tog ? . Odgovor je jasan: uslugom NOT predikata. Naime, prethodna namera se može ostvariti postavljanjem ovog jednog pitanja

```
?((NOT ? ((a|_x)(PP _x)FAIL))(NOT ? ((b|_y)(PP _y)FAIL)))
```

Zanimljiva je i duboka struktura tog pitanja, jer sada "sarađuju" predikati ? i NOT. Naime, pitanje je gledljivo ovako ?(for1 for2) gde for1, for2 su redom formule: (NOT ? ((a|_x)(PP _x)FAIL)), (NOT ? ((b|_y)(PP _y)FAIL)).

Znači, prvo treba izračunati for1, pa ako je rezultat da nastaviti dalje i izračunati for2. Da bismo izračunali for1 moramo prethodno izračunati vrednost ove formule (? ((a|_x)(PP _x)FAIL)), tj. u skladu sa definicijom ? predikata treba računati sastav: ((a|_x)(PP _x)FAIL). To računanje se zbog FAIL-a konačno završi sa ne, ali usput se ispišu sva a-rešenja[6] tj. liste (1),(5 6). Kako je formula (? ((a|_x)(PP _x)FAIL) netačna, formula for1 je tačna, pa u narednom koraku treba da računamo for2. To je slično prethodnom odnosno ispisaće se sva b-rešenja, tj. liste (3),(6 8), a formula for2 će imati vrednost da. Znači, konačan odgovor na pitanje ?(for1 for2) je da. U zi sa prethodnim primerom primetimo da ga , blisko izloženom, možemo rešiti i ovako -uvođenjem jednog dodatnog članka

```
((ispis_resenja _X)(NOT ? ((_X|_x)(PP _x)FAIL)))
```

Tada se gornje pitanje može kratko postaviti ovako

```
?((ispis_resenja a)(ispis_resenja b))
```

Primer 4.1.4. U Micro-prologu se predikat FORALL uvodi ovim člankom

```
((FORALL _A _B)(NOT ? ((? _A)(NOT ? _B))))
```

To na prvi pogled izgleda dosta složeno. Zamislimo da uopšte treba da izračunamo vrednost formule (FORALL _A _B), gde _A, _B imaju neke određene vrednosti. Već prvim pogledom na dati FORALL članak budući da u njemu imamo "parčiće" ? _A, ? _B zaključujemo da te vrednosti moraju biti sastavi. Da bismo lakše objasnili kako uopšte "radi" taj FORALL-članak najpre razmatramo primer u kome _A=((a _x)) , _B=((PP _x), a u kome tekući program ima ova dva članka ((a 1)), ((a 2)). U vezi sa računanjem vrednosti formule (FORALL _A _B) imamo ovako rasuđivati:

> Da izračunamo (FORALL _A _B) treba da izračunamo (NOT ? ψ), gde je ψ ovaj sastav ((? _A)(NOT ? _B)). To praktično znači da treba da uradimo to ψ i na kraju da negiramo dobijeni ψ-rezultat. Sledstveno, zaputimo se da proloski izračunamo sastav ψ. Kao prvo, treba izračunati (? _A), tj. treba izračunati sastav _A, odnosno izračunati ((a _x)). Odmah se jedan račun završava prihvatanjem zamene _x-->1. Znači, (? _A) ima trenutnu vrednost da , pa sledstveno prelazimo na računanje NOT- formule (NOT ? _B). U tu svrhu prvo računamo formulu (? _B) što se u uočenom primeru svodi na štampanje _x tj. 1. Formula (? _B) je dakle tačna. Stoga formula (NOT ? _B) je netačna. To iziskuje vraćanje nazad. I tako,

[6] Jer FAIL nas je stalno terao nazad da iznova dokažemo (a|_x) sa nekim _x.

opet dodemo na dokazivanje formule (? _A), što uzrokuje da sada _x dobije novu vrednost _x=2. Iza toga, opet računamo (NOT ? _B) , što se svodi na štampanje 2 na ekranu i zbog NOT opet vraćanju nazad. Međutim, pošto (a _x) nema više rešenja to znači da je završeno računanje vrednosti formule (? ψ) i da ta vrednost iznosi ne. Sledstveno, konačna vrednost tj. (NOT ? ψ) iznosi da.

Evo kako se može opisati ponašanje FORALL -članka u opštem slučaju, odnosno kako teče računanje formule oblika (FORALL _A _B) :

(i) Računa se _A. Ako je dobijen rezultat da, onda se ide na deo (ii), a ako je taj rezultat ne, onda se završava računanje vrednosti formule (FORALL _A _B) sa konačnim rezultatom da.
(ii) Uradi se _B i ako je rezultat ne algoritam se završava sa konačnim odgovorom ne , a ako je rezultat da, onda idemo na (i).

Rekli smo da ako se desi da je _B netačno, da onda računanje vrednosti formule (FORALL _A _B) se završava sa rezultatom ne. Zaista, na osnovu FORALL-članka vidimo da deo (NOT ? _B) ima vrednost tačno, što dalje znači da je izračunata vrednost formule

(? (? _A) (NOT ? _B))

sa rezultatom tačno. Sledstveno, formula (FORALL _A _B) kao njena negacija ima konačno vrednost netačno.

Primer 4.1.5. Neka program P ima ove članke ((a 1)) ((a 2)). Napraviti listu svih _x za koje važi (a _x).
Rešenje. Kao što je dobro poznato veoma je jednostavan proloski posao da se redom na ekranu ispišu sva _x-rešenja za (a _x). Dosta je postaviti ovo pitanje

(Δ) ?((a _x)(PP _x)FAIL)

ili, ako nećemo da se zbog FAIL na kraju ne pojavi i ?, postaviti pitanje

?((NOT ? ((a _x)(PP _x)FAIL))

Međutim, druga je "priča" ako hoćemo da se na ekranu jedanput, odnosno na kraju algoritma pojavi lista rešenja, tj. lista (1 2). Tu se sada pojavljuju principijelne teškoće uzrokovane prirodom proloskog algoritma. Naime, recimo u slučaju pitanja (Δ) sa rešenjem _x se dešava sledeće:

Prvo, prvi prvom dokazivanju (a _x), _x dobije vrednost 1 i pošto je formula (PP _x) "desnjak" od (a _x) ta se vrednost štampa na ekranu. Dalje, FAIL nas potera nazad i traži nam drugi dokaz za (a _x). I sada: Drugi dokaz se nadje davanjem _x-u vrednosti 2, što ujedno znači da se stara vrednost _x-a zaboravlja. Naravno, sada kad se dode na formulu (PP _x) štampa se ta nova vrednost za _x, itd.

Kao što vidite, kad bi nam nekako bilo dozvoljeno da "zapisujemo" usputne vrednosti promenljive problem bismo mogli rešiti. Sada, upoznajemo jedno takvo rešenje, u kome se koristi ovaj osnovni proloski predikat: ADDCL, čitati "dodaj članak". Evo kako on deluje:

Vrednost formule oblika (ADDCL Članak), gde je Članak neki proloski članak je tačno, a kao bočni efekat tekućem programu se na kraj dodaje taj Članak.

Recimo, odgovor na pitanje ?((ADDCL ((a _x)(b _x)))) formalno je da(tačno), a programu se dodaje članak ((a _x)(b _x)). Međutim, ako pitamo

?((ADDCL (a 6)))

dobicemo odgovor ?, tj. ne. Pored ADDCL u Prolog je ugraden i predikat DELCL sa smislom skloni, "brisi" neki članak. Recimo, pri računanju vrednosti formule (DELCL ((a _x)(b _x))), kao formalni rezultat će se dobiti da, a u bočnom efektu će se iz programa skloniti članak ((a _x)(b _x)). U vezi sa DELCL pomenimo još da ako želimo da obrišemo sve članke čije glave imaju izvesno ime Ime, onda u tu svrhu direktno služi osnovni proloski predikat KILL, koji u zamišljenom slučaju se upotrebi u obliku: (KILL Ime).

Sada navodimo pomenuto rešenje i to u slučaju traženja liste svih rešenja ma koje formule oblika (Ime|Cdr) , gde je Ime- ime formule. Jedan takav program glasi

```
((listares _A _L)(ADDCL ((lista ()))(NOT ? ((_A|_X)(radi _X)FAIL))
                                          (lista _L)(KILL lista))

 ((radi _X)(lista _S)(EQ _S1 (_X|_S))(DELCL ((lista _S)))
                                          (ADDCL ((lista _S1))))
```

Evo opisa kako radi. Glavna relacija je (listares _A _L) sa smislom: _L je lista svih rešenja formule čije je ime _A. Znači, ako recimo, hoćemo sva rešenja po _x za (a _x), onda pitamo ovako ?((listares a _L)(PP _L).
Pretpostavimo sada da tekući program P sadrži ta dva članka sa imenima listares i radi i da pored toga ima neke druge članke. Tada ako je Ime ime neke formule- glave bar jednog članka, onda da bismo dobili sva rešenja te formule postavimo pitanje ?((listares Ime _L)(PP _L). Evo šta se dešava tokom algoritma:

Prvo se programu pridružuje članak ((lista ())), što bi se moglo shvatiti kao da smo na samom početku listu svih rešenja postavili na praznu. Dalje, kao što će se videti, lista-članak će se menjati i u stvari će postupno uključivati razna rešenja, a na samom kraju ćemo imati članak ((lista _L)) gde je _L tražena lista svih rešenja.
I tako, kao što rekosmo na samom početku dodaje se članak ((lista ())))
Sad na pozornicu nastupa formula
(not) (NOT ? ((Ime|_X)(radi _X)FAIL)
sa smislom
Uradi sledeći sastav:
(Σ) ((Ime|_X)(radi _X)FAIL)
Čim se dode iza (Ime|_X), tj. čim se nade jedno _X za koje važi (Ime|_X) onda treba "izračunati" (radi _X). U stvari, tada se od lista-članka oblika ((lista _S)) --brišući ga-- prelazi na novi lista-članak ((lista (_X|_S))), tj. praktično lista rešenja se dopuni tim _X. U idućem koraku se stiže na FAIL koji nas tera nazad i dotera nas do traženja novog rešenja za (Ime|_X). Iza toga se i to rešenje slično prethodnom uključuje u listu rešenja. Kao što se vidi, izvršavanje sastava (Σ), blagodareći i FAIL-u, na kraju proizvede lista-članak oblika ((lista _L)), gde je _L lista svih rešenja. Ali, zbog FAIL-a sastav (Σ) ima vrednost ne. Međutim, formula (not) je (NOT ? (Σ)), pa znači ona je tačna. Sledstveno idemo na njenog "desnjaka" , tj. formulu (lista _L). Pošto je u tom trenutku već napravljena lista svih rešenja to će _L dobiti željenu vrednost. To je važno jer na početku smo tražili vrednost formule (listares Ime _L), pa znači da je sada _L dobilo tu vre-

dnost. Inače, iza formule (lista _L) nalazi se deo (KILL lista) u kome brisemo lista-članak jer nam više ne treba.

U vezi sa navedenim rešenjem istaknimo sledeće. U njemu je implicitno pretpostavljeno da tekući program P ne sadrži nijedan lista-članak, jer se ti članci uvode i gase tokom rešavanja postavljenog pitanja. Dalje, navedeno rešenje usled povećanog korišćenja usluga predikata ADDCL, DELCL i KILL, reći ćemo slobodnije korišćenjem "dodajne tehike", nije previše brzo.
S tim u vezi istaknimo da, s obzirom na važnost navedenog pitanja u Micro-prologu postoji predikat ISALL kojim se rešava takvo pitanje. Taj važan predikat objašnjavamo postupno. Tako, smisao formule

(ISALL _L _x for1 for2 ... fork)

je _L je lista svih _x za koje važe sve formule for1, for2, ..., fork i gde k može biti 1, 2, ... Značajno je istaći da je upotreba tog predikata veoma efikasna, jer -kako piše u priručniku za Micro-prolog- on nije u potpunosti napisan na Prologu, pa ne pati od sporosti tipične za gore navedeno rešenje "dodajnom" tehnikom. Evo malog primera za ilustraciju korišćenja ISALL predikata. Uočimo program od ovih članaka

```
((c 2))  ((c 3))  ((c 4))
((b 1))  ((b 2))  ((b 3))
((a) (ISALL _L _y (b _y) (c _y))(PP Evo _L))
```

Tada na pitanje ? ((a)) će se dobiti (2 3), tj. lista _L svih _y koji zadovoljavaju uslove (b _y), (c _y). Uočimo i ovaj primer u kome program sadrži sledeće članke

```
((a 1)) ((a 2)) ((a 3))
((zbir () 0))
((zbir (_x|_y) _Rez)(zbir _y _Rez1)(SUM _x _Rez1 _Rez))
((saberi) (ISALL _L _y (a _y)) (zbir _L _Rez)(PP Evo _Rez))
```

U tom programu se na pitanje ?((saberi)) najpre napravi _L -lista svih _y -rešenja od (a _y), tj. napravi lista (1 2 3), a onda se pomoću zbir-članaka nađe zbir članova te liste. To je _Rez i on se konačno štampa.

ISALL predikat je moćniji nego što se iz dosadašnjeg izlaganja zaključuje. Naime, kao prvo pored toga što ima svojstvo generatornosti, tj. njime se može napraviti lista rešenja date formule, on ima i svojstvo provernosti, tj. i sa datom listom L možemo pitati da li je ona lista rešenja date relacije. Međutim, drugo, ISALL predikat nije samo zadužen za pravljenje i proveravanje liste rešenja date formule, već može "uraditi" mnogo širi posao. Da bismo to lakše videli najpre zamislimo da tekući program P ima ove članke

```
((a 1))  ((a 2))  ((a 3))
((b 2)) ((b 3)) ((b 4))
```

Postavimo ovo pitanje ?((ISALL _L (b _x) (a _x))(PP _L)) sa smislom:

_L je (nepoznata) lista svih članova oblika (b _x), a pritom to _x prođe preko svih rešenja formule (a _x)

Znači, zbog (PP _L), na ekranu treba da se pojavi sledeća lista

((b 1) (b 2) (b 3))

Primetite to b nema nikakve veze sa b u b-člancima uočenog programa P. Naime u gornjem ISALL-pitanju deo (b _x) je samo UZORAK OBLIKA ČLANOVA TRA-

ZENE LISTE _L. Da smo umesto (b _x) stavili

(c _x 888) ili (b|_x) ili 777

onda za _L bismo dobili redom

```
_L:    ((c 1 888) (c 2 888) (c 3 888))
_L:    ((b|1) (b|2) (b|3))
_L:    (777 777 777)
```

Opet u vezi sa programom P postavimo sledeće pitanje

?((ISALL _L (c _x _x) (a _x) (b _x))

Znači traži se lista svih članova oblika (c _x _x), pri čemu _x prođe preko svih rešenja obe formula (a _x), (b _x). Za _L će se dobiti ova lista ((c 2 2) (c 3 3)). Završavajući primere, pomenimo još da smo recimo postavili pitanje

?((ISALL (3 4 7) (c _x) (b _x))

odgovor bi bio ne, jer lista (3 4 7) nije jednaka listi svih članova oblika (c _x), gde _x ide redom preko svih rešenja za (b _x). I sada se uputimo opštem slučaju. Neka je _L promenljiva. Tada pri računanju vrednosti formule

(ISALL _L Uzorak for1 ... fork)

promenljivoj _L se kao vrednost daje lista svih članova oblika Uzorak za sve slučajeve tačnosti redom svih formula for1, ..., fork. Da bismo bolje objasnili pretpostavimo da su X1, ..., Xm sve promenljive iz tih formula i pomenuti uzorak označimo ovako Uzorak(X1, ..., Xm) -što ne mora da znači da naznačene promenljive stvarno učestvuju u njemu. Tada zamislimo da se proloski računa sastav

for1 ...fork (PP Tu smo) FAIL

i da kad god se dođe na mesto (PP Tu smo) u listu _L, koja je na samom početku prazna, nekako se doda član oblika Uzorak(X1, ..., Xm), gde X1, ... Xm imaju zatečene vrednosti. Naravno, zbog FAIL-a će se tako, u mislima, napraviti čitava lista _L. Naravno taj opis nije čist proloski, odnosno hipotetički je.
Međutim, pored tog svojstva generativnosti ISALL predikat ima i svojstvo provernosti. Tako ako imamo formulu oblika

(ISALL Lista Uzorak for1 ... fork)

gde je Lista neka određena lista, onda formula je tačna, netačna prema tome da li je ta Lista jednaka, nejednaka pretodno opisanoj "listi svih uzoraka Uzorak pri važenju redom formula for1 ... fork".

Na kraju ovog dela o formulama i člancima Lisp-sintakse još nekoliko reči o u Prolog ugrađenim predikatima (relacijama). O nekim od njih kao ADDCL, DELCL, KILL je već bilo reči. U vezi sa njima sada dodajemo i ovo:

ADDCL ima dva vida:

(ADDCL X), kojim se tekućem programu dodaje članak[7] X, i to ako već

[7] Može X biti i promenljiva ali u trenutku njenog računanja mora postati

program ima neke članke imena istog kao ime tog članka X, onda taj članak X se stavlja iza svih njih.
(ADDCL X m), gde je X članak, a m=1,2,3,... neki prirodan broj. Njime se članak X dodaje tekućem programu i pritom se smešta na m-to mesto među člancima imena istog kao X[8].

DELCL takođe ima dva vida

(DELCL X) X je ili određen članak ili je promenljiva koja u trenutku računanja mora biti oblika članka(u smislu (4.1.6), tj. biti oblika ((ime|A)|B) gde je ime neka konstantska reč.

(DELCL ime m) gde je ime neka konstantska reč, tj. ime članka. Smisao je:
"Obrisati" m-ti po redu članak imena ime.

Kao što smo već, istakli i kod ADDCL i DELCL X sme biti promenljiva, ali u trenutku kad te formule dođu na pozornicu (na račun) X mora već steći određenu vrednost. Evo malog primera za objašnjenje. Recimo, možemo relaciju upis_članka uvesti ovako,

```
((upis_clanka)(R _X)(ADDCL _X))
```

Znači tu se (ADDCL _X) pojavljuje nakon (R _X) kada smo dužni da _X-u damo prikladnu vrednost--u stvari, Micro-prolog jedino proverava da li važi uslov (*4), tj. da li je car(car(X)) konstantska reč.

KILL se koristi u tri vida:

```
(KILL ime)  --briše sve članke imena ime.
(KILL (ime1 ime2 ... imek)) --briše sve članke imena ime1,.....,imek.
(KILL ALL)  --briše sve članke tekućeg programa.
```

Pored ADDCL, DELCL i KILL predikata u radu sa člancima u Micro-prologu je dostupan još jedan značajan predikat o člancima. To je CL, koji se pojavljuje u dva vida

```
(CL X)                                   (X je članak programa)
(CL X Y Z)                               ( O b j a š nj e nj e  dole)
```

Evo nekoliko reči o prvom. X recimo može biti neki potpuno određen članak, i onda vrednost od (CL X) je da, ne prema tome da li X je, ili nije članak programa. Međutim, pored toga (CL X) je sposoban da slično uradi i kada X nije potpuno određen članak, ali uz uslov da je poznato makar ime tog članka. Naime, ako nas recimo zanima da li uopšte ima neki članak sa imenom ime, to možemo saznati postavljanjem pitanja

```
?((CL ((ime|_X)|_Y))(PP ((ime|_X)|_Y)))
```

gde su _X, _Y su promenljive. Primetite da smo u pisanju članka koristili samo najopštiji uslov (*4), koji traži da car(car(članak)) bude konstantska reč.

---

neki članak (u smislu (4.1.6)).

[8] Recimo, ako program sadrži članke (navedene redom)
((a 3)) ((a 44)) ((a 55))
tada nakon odgovora na pitanje ? ((ADDCL ((a 7)) 3)) prethodni spisak a-članaka se preobraća u ovaj
- ((a 3)) ((a 44)) ((a 7)) ((a 55))

Formula (CL X) se , recimo, koristiti da napravimo svoj vlastiti LIST svih članaka datog imena. Jedan takav "list" predikat se može definisati ovako[9]

```
((list _X)(CL ((_X|_Y)|_Z))(PP ((_X|_Y)|_Z))FAIL)
((list|_X))
```

Recimo, na pitanje ?((list a)) imaćemo ispis svih a-članaka. Recimo, ako u programu imamo članke

```
((a 1)(c 6 7)))
((a _X _Y)(b _X _6)(c _Y))
```

onda će se gornjim pitanjem ispisati oba ta članka--uprkos okolnosti da je relacija glave prvog članka dužine 1, a relacija glave drugog je dužine 2. To je opet jedna sjajna osobina Micro-prologa. Slično nije moguće u prolozima Edinburgške sintakse.
Malo je složenija formula

```
(CL X Y Z)
```

Naime, u njoj X je članak, naveden makar u obliku ((ime|_X)|_Y), a Y i Z su 1,2,3,.... pri čemu Y mora biti zadan, a za Z se pretpostavlja uslov Z>=Y. Da bismo lakše objasnili uočimo ovaj program

```
((a 2)(s 4))    ((b 55))    ((a 4)(b 7))    ((a 6))
((a 8)(s 5))    ((s 4))     ((a 10))        ((a 12))
```

Tada, prvo, na pitanje oblika ?((CL ((a|_X)|_Y) p q)(PP ((a|_X)|_Y))), gde su p, q dati

u slučaju q<p na ekranu će se pojaviti ?, jer kako smo gore rekli traži se q>=p;
a u slučaju q>=p na ekranu će se pojaviti p+(q-p), tj. q-ti po redu a-članak. Recimo, za p=2, q=5 na ekranu će se pojaviti ((a 10)).

Međutim, drugo, ukoliko je q promenljiva ,što je inače dopušteno, "dešavaju" se nove i zanimljive stvari. Naime, tada promenljiva q najpre dobije vrednost p a potom, ako se desi "povraćaj" na to mesto (backtracking) ta promenljiva se uvećava za 1, i tako pri povraćaju se nastavlja dokle može; najveća vrednost za q je upravo broj svih članaka dotičnog imena. Evo konkretnih primera:

```
?((CL ((a|_X)|_Y) 2 _q)(PP _q))       Na ekranu će se pojaviti  2
?((CL ((a|_X)|_Y) 2 _q)(PP _q)FAIL)   Na ekranu će se pojaviti  2,3,4,5,6
                                      kao i završni ? (zbog FAIL-a).
```

Kao što se primećuje, tro-parametarski CL predikat dopušta razne korisne mogućnosti u radu sa člancima nekog imena.

---

[9] Da bismo ispisali sve članke datog imena _X pored ostalog koristimo i uslugu FAIL-a. Ali, da na kraju ne bismo imali formalan rezultat ne, uveli smo i drugi list-članak. Međutim, navedimo da umesto navedena dva možemo koristiti i ovaj jedan:
((list _X)(NOT ? ((CL ((_X|_Y)|_Z))(PP ((_X|_Y)|_Z))FAIL)))

4.2 Slučaj Edinburgške sintakse

Za razliku od Lispovske sintakse u kojoj je sve,tj.formule i članci, pravljeno samo iz lista, u Edinburgškoj se pored njih koriste i strukture kao

```
write(A),NOT(f(3)),f(x),g(X,23,h(4)),s(6,[X|Y])
```

op-izrazi[10] kao što su 4-8, X=Y, X is 2+3, A=..B

i tzv. gramatički izrazi,kao što su

```
rečenica -->podmet,prirok.
podmet -->[dete];[kuća].
prirok-->glagol.
prirok-->podmet,glagol.
glagol-->[voli].
```

(Smisao: rečenica nastaje dopisivanjem podmeta i priroka. Dete i kuća su primerci podmeta,voli primerak glagola i dr.)

U vezi sa gramatičkim izrazima za sada recimo samo toliko da se oni na odreden, ali veoma lep način, koji detaljnije upoznajemo u delu 4.4 ove tačke, prevode na obične prološke članke,tj.članke bez takvih izraza. Može se reći da su ti izrazi "nakalemljeni" na Prolog,tj. Prolog može bez njih, ali s druge strane, pomenuto prevođenje daje jedan značajan metod rešavanja raznih pitanja u vezi sa njima,odnosno -kaže se u vezi sa gramatikama.

Takode i o op-izrazima će biti šire izlaganje.Ono se nalazi u delu 4.3 ove tačke.A na ovom mestu pomenimo samo da slično gore navedenim primerima op-izraza -ugradenim u Prolog- korisnik sam može uvesti svoje op-izraze. Primeri:

```
fi X,  X psi,  X medu Y
```

gde su fi,psi,medu tzv. imena tih izraza, i ona su -tako se kaže- redom pisana

```
prefiksno, kod  fi X
sufiksno, kod   X psi
infiksno, kod   X medu Y.
```

U Edinburgškoj sintaksi od najvećeg značaja su pomenute strukture, odnosno podesnije je reći termi. Inače nije teško po ugledu na definiciju [4.1.1] napraviti njihovu potpuno strogu definiciju[11]:

(i) Term-jedinke su strukture
(ii) Ako je f konstantska reč ,onda reč

```
f(s1,s2,...,sk)
```

je takode struktura,ukoliko svaki od s1,..,sk je ili struktura ili lista.

Kao što vidite u toj definiciji broj k, tj. broj argumenata na koje se f odnosi,tzv. dužina(f), se ne precizira. Recimo, zapis f(2,[3,4],f(X)) je

[10] Rekli smo op-izrazi umesto "izrazi sa operatorima"

[11] Tu se podrazumeva i deo tipa (iii) iz definicije (4.1.1).

ispravan primer strukture , premda f u njemu nema fiksiranu duzinu. Ali, u vezi sa tom definicijom dodajmo i ovo: ukoliko je f ime u Prolog ugraden predikata,recimo write,read,not i dr.onda dužina(f) je fiksirana,tj. k je je ili 1 ili 2 ili 3 ,itd. Tako,od zapisa write(4),write(6,7) samo prvi je je struktura.

Odlažući za kratko strogu definiciju op-izraza sada prelazimo na objašnjenje pojma formula i članka Edinburgške sintakse. Formula je uopšte

(j) neka složena struktura,tj. struktura koja nije term-jedinka;ili
(jj) neka term jedinka,koja nije oznaka broja; ili
(jjj) neki op-izraz.

Primeri formula su f(2), g(7,[3,X]), X is 3+7, kon, X. Pretposlednji primer je konstatska reč, a poslednji je promenljiva. Ako je formula oblika f(...),onda f zovemo ime te formule.Ako je formula oblika nekog op-izraza Članak može biti

ili jedno-formulski (tzv. fakt, ili elementarna aksioma);
ili više-formulski.

Jedno-formulski članak je oblika

```
for1.        (Znak tačke je obavezan)
```

gde for1 (glava članka) je

(*) Formula koja nije promenljiva i čije ime nije ime nekog osnovnog prološkog predikata.

Tako,recimo ispravan je članak f(2,f(X,4)). ali ne i write(4).Više-formulski članak ima oblik

```
for1:-for2,...,fork      (k>1)
```

gde for1 (glava članka) ispunjava uslov (*) i svaki od for2,...,fork je formula ili jedan od znakova !,fail....Medutim, na tom mestu istaknimo da LPA-prolog dopušta da neki od tih for2,.....,fork bude oblika Var(...), gde je Var promenljiva.Podsetimo da smo tu odliku LPA-prologa koristili u Zadatku 3.16. U vezi sa člancima dodajemo jednu važnu dopunu. Kao što već na početku ovog izlaganja videli u Micro-prologu je osnovni "pokretački" predikat ?, čija definicija je ((? _X)|_X). Budući da _X mora biti sastav, njegov osnovni smisao se može opisati rečima "uradi" _X,tj. redom proloski izračunaj pojedine CAR-ove tog _X. U Edinburgškoj sintaksi korisniku nije dostupan nijedan predikat potpuno sličan predikatu ?, ali zato u njoj se ideja tog predikata ostvaruje na način koji opisujemo. Da bismo to lakše uvideli, zamislimo,da smo uveli sledeći članak

```
uradi(X):-X.
```

Tada je očigledno, da računanje vrednosti formule uradi(write('Pera')) se svodi na računanje vrednosti formule write('Pera'), tj.svodi se na štampanje reči Pera. Ali, kako naložiti Prologu da ,recimo, izračuna dve formule zaredom kao : write('Pera'),write('Jova') ? Edinburgška sintaksa to dopušta na sledeći način. Treba za X u gornjem članku uzeti

(4.2.1) (write('Pera'),write('Jova'))

tj. pojedine sastavke write('Pera'), write('Jova') staviti u male zagrade (). U opšte, ako se umesto X zameni (for1,for2,.....,fors) onda uradi(X) se

svodi na traženje vrednosti konjunkcije for1,for2,...,fors. Može se uopšte reći, da upotrebom malih zagrada () -u značenju konjunkcije sastavaka- i Edinburgška sintaksa postiže dejstvo prethodno isticanog predikata ?. Može se slobodnije reći da se upotrebom malih zagrada od više sastavaka pravi jedna celina. Iz tog razloga ćemo govoriti da je zagradama () definisana jedna vrsta struktura imena ujednač[12].

U Primeru 4.1.1 smo u Micro-prologu upoznali OR-predikat. Po analogiji sa tim OR- člancima napravimo ova dva

(4.2.2)
```
ili(X,Y):-X.
ili(X,Y):-Y.
```

Nije teško videti da se tako uveden ili-predikat, isto ponaša kao rečeni OR-predikat. Ali, naravno, pri korišćenju tih članaka ako X ili Y treba da dobije "složenu" vrednost kao for1,for2,....,fors onda moramo koristi male zagrade kao "poveznice",kao "ujednač".Međutim, u skladu sa okolnošću da je logički veznik ili veoma značajan u Edinburgškoj sintaksi ima "jedan ugrađen dodatak" koji ostvaruje ulogu tog veznika.Naime, zapis oblika:

(a1,a2,...,ak ; b1,b2,...,bl)

koji sadrži znak ; ima upravo ovaj smisao: ili(X,Y) ,pri čemu

X je (a1,...,ak), a Y je (b1,...,bl).

Evo primera članaka sa upotrebom znaka ;

```
p:-q,r,(a,b;c,d),u.
f:-a,b;c,d.
```
Pazite tu su dogovorno izbrisane spoljne zagrade (). Znači, taj članak je zamena za ovaj f:-(a,b;c,d).

Sada upoznajemo nekoliko u Prolog ugrađenih predikata u vezi sa strukturama,listama i članicima. Među prvim je predikat functor,koji se koristi u obliku

(4.2.3) functor(Struct,Ime,Duzina)

gde Struct je izvesna struktura,Ime je ime te strukture, a Duzina je njena dužina,tj. broj argumenata na koji se odnosi. Evo primera upotrebe:

1) Na pitanje ?-functor(f(a,b,g(c,d)),f,3) se dobija odgovor yes, jer za datu strukturu f(a,b.g(c,d)) ime je zaista f, a duzina je 3. Uopšte, ako su zadani Struct,Ime,Duzina onda predikat functor je "sposoban" da proveri tačnost od (Δ).

2) Na pitanje ?-functor (f(a,b,c),X,Y). će se dobiti X=f,Y=3. Uopšte, za datu Struct u (Δ) predikat functor je sposoban da nađe Ime i Duzinu.

3) Može se slobodno reći da su prethodna dva svojstva predikata functor trivijalna i da bi čovek mogao sam da napravi predikat sa te dve sposobnosti[13]. Međutim, postavimo sada ovakvo pitanje

```
?-functor(S,f,2).
```

Kao odgovor za S ćemo dobiti f(Var1,Var2), gde su Var1, Var2 neke dve promenljive. Uz to istaknimo da je predikat functor tako sagrađen da čak dopušta da mu Ime, Duzina budu promenljive, ali koje kad po prološkom računu na red dođe (Δ) imaju određene ("zatečene") vrednosti. Recimo,na pitanje ?-read(N),functor(S,f,N). odgovori će zavisiti od toga koliko N zadamo. Ubrzo ćemo upoznati jedan dublji primer u kome se koristi opisani slučaj 3).

Naredni predikat je clause(članak) koji se koristi u obliku

(4.2.4) clause(Glava,Rep)

gde Glava,Rep redom označavaju glavu,rep nekog članka. Da bismo objasnili primere korišćenja uočimo ovaj prološki program

```
p(1).          p(2).
p(3):-q(77).   p(3,4):-p(7),p(8).
```

Tada na pitanje ?-clause(p(3),q(77)). je odgovor yes, jer tekući program ima članak sa glavom p(3) i repom q(77). Primetimo, da za prva dva članka, koji su oblika "elementarne" aksiome ("fakta") ,u Prologu se smatra da su im repovi konstantska reč true.Međutim,predikat clause pored svojstva provernosti,u smislu koji objašnjavamo,ima i svojstvo generatornosti. Recimo, na pitanja oblika ?-clause(p(1),X). ?-clause(p(3,4),Y). će se redom dobiti X=true, Y=p(7),p(8).

Uopšte,na pitanje oblika ?-clause(Glava,X). Prolog ovako postupa.U tekućem programu traži (prvi) članak sa glavom Glava, i ako ga nađe onda X -u daje vrednost rep tog članka;a ako ne nađe nijedan Glava-članak onda je odgovor no, tj. ne. Međutim, uz sve to, clause-predikat je još moćniji. Naime, u prethodnom pitanju nije neophodno da se da "čitava" glava, dovoljno je dati ime glave,a njeni argumenti se mogu ostaviti kao promenljive. Recimo,na pitanje oblika ?-clause(p(X),Y),write(Y),fail. preko Y će se dobiti repovi svih članaka čija glava je oblika p(X),sa nekim X, odnosno Y će imati redom ove vrednosti true,true,q(77).Da bismo upotpunili opis clause-predikata pretpostavimo da želimo da prološki definišemo predikat

clanak_ispis(Ime,Duzina)

koji je u stanju da za dato Ime i datu Duzinu ispiše sve članke čija glava ima to Ime,i koja kao struktura ima tu datu Duzinu[14] Tu je, u odnosu na prethodno, dodata teškoća što Duzina nije unapred data[14].Međutim, tu će nam u pomoć priskočiti predikat functor. Evo jedne moguće definicije predikata clanak_ispis:

```
clanak_ispis(Ime,Duzina):- functor(Glava,Ime,Duzina), clause(Glava,Rep),
                           write(':-'(Glava,Rep)),nl,fail.

clanak_ispis(Ime,Duzina).
```

---

[12] U stvari,malim zagradama i bukvalno odgovara određena struktura. Naime,zapisu (a,b) odgovara ova struktura (imena ','): ','(a,b),a zapisu (a,b,c) odgovara struktura ','(a,','(b,c)) i slično dalje.

[13] Zaista,neka je, recimo, f(a,b,c) data struktura. Evo kako joj možemo naći ime: prvo primenom predikata =.. u obliku f(a,b,c)=..X nađemo X tj.listu [f,a,b,c], a dalje je lako naći njen prvi član- što je traženo ime. Naravno, ta se zamisao lako uopštava na slučaj ma koje strukture. Takođe je uz to jednostavno pronaći dužinu strukture.

[14] Jer,ako bi to bio slučaj i recimo Duzina=3, a Ime=f, onda bismo mogli kao i gore tražiti članke sa glavom f(X,Y,Z).

Imamo dva članka, drugi je dodat da se zbog fail-a u prvom "spasimo" od završnog no- odgovora. Primetimo još da pri ispisu članka on se iskazuje u obliku strukture

(4.2.5) ':-'(Glava,Rep).

Uz predikate functor i clause dolazi i predikat arg, koji se koristi u obliku arg(Broj,Struct,Clan), što se približno može opisati rečima:
U strukturi Struct Clan je upravo argument koji je rednog broja Broj. Recimo, na pitanje oblika

(4.2.6) ?-arg(2,f(a,b,c,d),X).

za X će se dobiti a. Međutim, arg-predikat je takođe u stanju da delom nepoznatoj strukturi Struct na željeno mesto postavi željeni argument. Tako, na pitanje oblika ?-arg(2,f(a,X,g(c)),pera). će se dobiti: X=pera, tj. u strukturi f(a,X,g(c)), drugi član koji je inače nepoznat je postao pera.

Sada ukratko izlažemo o predikatima čijim dejstvom se može menjati tekući program; neki članak dodati, neki obrisati i sl. Tako, za dodavanje članaka imamo tri slična predikata assert, assertz, asserta sa ovim opisom:

(4.2.7) assert(Clanak), assertz(Clanak)

dodaju članak Clanak na kraj člancima čiji predikat(ime) je isto sa tim člankom. Tako, računanjem formula assert(s(2));assert(':-'(p(X),(q(X),r(X))) tekućem programu će se dodati

članak s(2). na kraj svih s-članaka[15]
članak p(X):-q(X),r(X) na kraj svih p-članaka.

Primetimo da smo drugu assert-formulu mogli i ovako pisati

assert((p(X):-q(X),r(X)))

Istaknimo da oba predikata assert, assertz deluju na isti način. Međutim, za razliku od njih predikat asserta, korišćen inače u obliku

(4.2.8) asserta(Clanak)

dotični članak Clanak umesto na poslednje stavlja na prvo mesto (odgovarajućih članaka).

Za brisanje, uklanjanje nekog članka koristi se predikat retract. Naime, računanjem formule oblika

(4.2.9) retract(Clanak)

iz tekućeg programa , odnosno spiska njegovih članaka izostavlja se prvi članak ujednačiv sa Clanak. Recimo, ako tekući program sadrži članke

a(1). a(2). a(3). a(4).

tada računanjem formule retract(a(2)) će se ukloniti članak a(2). i onda će spisak a-članaka postati

a(1). a(3). a(4).

Ali, ako sada dođe prilika da se računa formula retract(a(X)) onda će se

[15] Ako ih ima, a inače taj članak će biti jedini s-članak.

obrisati prvi članak ujednačiv sa a(X)., tj. članak a(1)., pa će novi spisak a-članaka biti

a(3). a(4).

Uz navedene predikate dolazi i predikat[16] abolish . Naime, sa

(4.2.10) abolish(ime/arnost)

se brišu svi članci sa predikatom(imenom) ime i u kojima ta relacija ima dužinu arnost. Recimo, sa abolish(a/1) bismo u prethodno zamišljenom programu obrisali sve a-članke: a(3). a(4). Međutim, da je program sadržao i ovakav a-članak a(X,Y):-b(X),c(Y). on ne bi bio obrisan[17].

Sada nekoliko reči o Edinburgškim predikatima u vezi sa "listama rešenja". Prvi takav je findall, korišćen u obliku

(4.2.11) findall(Uzorak,Relacija,Lista)

sa smislom

Lista je lista svih izraza oblika Uzorak tako da važi Relacija

Recimo, ako program sadrži ove a-članke: a(1,2). a(2,3). a(1,4). onda na pitanje ?-findall(X,a(1,X),L). za L će se dobiti [2,4], jer to je lista svih X za koje važi a(1,X). Dalje, na pitanje oblika

?-findall(b(X,777),a(1,X),L).

za L će se dobiti lista [b(2,777),b(4,777)] jer sada je uzorak oblika b(X,777). Uzorak, uopšte može biti ma koji izraz. Tako, na pitanje

?-findall(5,a(1,X),L).

dobiće se L=[5,5]. Dodajmo još da predikat findall ima i svojstvo provernosti, tj. u slučaju formule findall(Uz,Rel,Lis) sme Lis već imati neku vrednost i tada vrednost te formule je tačno ili netačno. Recimo, vrednost formule findall(4,aa(X,7),9) je netačno. Pored findall postoje još dva donekle slična, ali složenija, predikata: bagof, setof. Najpre uočimo ovakav primer, odnosno ove članke ("bazu podataka"):

b(22). b(3). b(4). b(4). b(3).
a(5,3). a(1,2). a(1,7). a(1,2). a(2,3).

Tada pri računu formule

bagof(X,b(X),L) čitati: L je lista svih X za koje važi b(X)

za L će se dobiti lista [22,3,4,4,3] pa se u tom primeru bagof vlada kao findall. Međutim, pri računu formule setof(X,b(X),L) za L će se dobiti lista svih različitih X , navedenih rastući, za koje važi b(X), tj. L će biti [3,4,22]. U stvari, ta uređenost i različnost je jedina razlika između bagof("torbe rešenja") i setof ("sređenog skupa rešenja"). Međutim, dalja priča oko bagof i setof je nešto složenija. Naime, u vezi sa tim formulama, na poseban način neke promenljive se označavaju kao "vezane", a preostale

[16] Sličan KILL-predikatu u Micro-prologu.

[17] Ali, recimo, računanjem formule abolish(a/2) on bi bio obrisan.

su onda "slobodne". Recimo, pri pisanju X^ je naznačeno da je X vezano[18], slično sa A^B^ je označeno da su i A i B vezane. A sada evo kako uopšte izgleda neka bagof, odnosno setof formula

(4.2.12) bagof(Uzorak,V1^V2^...Vk^Rel,Lista)
setof(Uzorak,V1^V2^...Vk^Rel,Lista) , gde k≥0

što se može ovako pročitati:

> Lista je torba, odnosno uređen skup svih uzoraka Uzorak, tako da važi Rel, za neke V1,V2,...,Vk .

Recimo, u vezi sa prethodnim primerom računanjem formule

bagof(X,Y^a(X,Y),L)

za L će se dobiti [5,1,1,2], a da je umesto bagof bilo setof dobili bismo listu [1,2,5]. Navedenu formulu bismo mogli ovako čitati

L je lista svih X za koje važi a(X,Y) sa po nekim Y.

Međutim, pri računu formule bagof(X,a(X,Y),L) nastaju nove stvari u skladu sa okolnošću da je sada Y nevezana, tj. slobodna promenljiva u a(X,Y). Ta se formula računa ovako:

> Prvo se traži prvo X,Y-rešenje i to je 5,3. Sada se Y "ukoči" sa tom tom vrednošću 3 i onda traže preostali X za koje važi a(X,3). Dobije se lista(torba) [5,2]. To je prva vrednost za L. Rekli smo prva, jer ako se kojim slučajem zbog vraćanja(backtrackinga) moramo tražiti drugu mogućnost dokazivanja gornje formule, onda se dalje pristupa kako sledi. Traži se novo X,Y-rešenje kod koga Y nije 3. Dobije se Y=2, a odgovarajuća torba je sada L=[1,1] i konačno ako se javi ponovo ponovo potreba za novim rešenjem (zbog vraćanja), onda se Y daje preostala vrednost 7, a L postaje [1].

Završna napomena:
U vezi sa formulama i članicima Edinburghške sintakse na kraju istaknimo i ovo. Ona je tako građena da korisniku na usluzi budu razna sintaksna sredstva. Ali, da li je moguće sve te njene sintaksne raznolikosti misaono prevesti na njihov manji broj, pa čak i sve njih svesti na samo jednu vrstu, kao što je slučaj sa Lisp-sintaksom? Odgovor je potvrdan. Naime, prvo svaki članak koji je inače ili oblika elementarne aksiome Glava. ili oblika Glava:-Rep se može prevesti na odgovarajuću strukturu: ':-'(Glava,true), odnosno[19] ':-'(Glava,Rep). A dalje, blagodareći predikatu =.. svaku strukturu možemo prebaciti na listu. To je tačno i za op-izraze, pa konačno zaključujemo da je i u Edinburgškoj sintaksi lista osnovni sintaksni materijal. To nam, pored ostalog, u daljem izlaganju dopušta mogućnost da u opštem opisu proloških algoritma isključivo koristimo jezik lista.

---

[18] Podesno je zamišljati kao da to označava: postoji X, tj. znak ^ treba tumačiti kao logički kvantor postoji.

[19] Ima nekih dodataka ako Rep nije jednočlan, kao recimo kod ovih članaka
p:-q,r. a:-b,c,d.
Njima odgovaraju redom ove strukture
':-'(p,(q,r)) ':-'(a,(b,c,d))
u kojima učestvuju male zagrade, tj. strukture imena ' , ' (ujednači).

4.3 Još o op-izrazima

Sada prelazimo na opis op-izraza[1] (zovu se i operatori). Odmah napomenimo da se ti izrazi koriste da bi se korisniku dopustilo da --pod određenim uslovima-- u pisanju struktura ponegde ne piše zagrade, tj. da koristi pisanje kao što je uobičajeno u matematici. Recimo, piše se a+b*c, što bi inače u "strukturnom" pisanju postalo +(a,*(b,c)). Kratko rečeno op-izrazi se pojavjuju u jednom od ovih oblika

(i) Unaran slučaj:
(a) pre Deo1
(b) Deo1 posle
(ii) Binaran slučaj: Deo1 medju Deo2

gde su Deo1, Deo2 neke formule, dok reči

pre, posle, medju

su imena, ili glavni znaci, tih op-izraza. To uopšte mogu biti neke konstantske reči, za one op-izraze koje mi sami, tj. korisnik uvodi, odnosno neke od reči koje ne odgovaraju op-izrazima ugrađenim u Prolog, u koje dolaze:

+, -, *, =, =.., <, is

i drugi. Inače, u navedenom opisu kao op-imena korišćene su reči pre, posle, medju da bi se ukazalo na kom mestu se takvo ime nalazi. S tim u skladu se kaže da su izrazi (i) (a), (i)(b), (ii) redom u prefiksnoj, sufiksnoj odnosno infiksnoj notaciji. Dalje, kad korisnik uvede neki od op-izraza tada je na njemu da se opredeli na jednu od ovih mogućnosti :

da li hoće da se op-ime shvata kao relacija ili kao operacija.

Recimo, pretpostavimo da želimo da uvedemo op-izraz sa imenom pozit, koje hoćemo da pišemo prefiksno i takođe da smisao tog pozit bude 'pozitivan'. U tu svrhu možemo uvesti ovaj članak

pozit X:-X>0.

Ali, tu ima jedna bitna novost. Naime, Prolog ne dopušta tako neposredno, "bez najave" uvođenje op-izraza, odnosno najpre na određen način mu moramo saopštiti neke pojedinosti o op-izrazu, pored ostalog da li je pozit relacijske ili operacijske prirode. Toj svrsi služi, u Prolog ugrađen, predikat op, koji se koristi kako opisujemo. Da bismo saopštili Prologu da želimo da pozit bude ime jedne prefiksno pisane relacije najpre, tj. pre gornjeg članka, stavljamo ovakav poseban "najavni" članak

(Oper) :-op(100,fx,pozit).

gde treći argument, tj pozit, saopštava ime op-izraza; drugi argument je fx, on saopštava da je pozit ime relacije pisane prefikno; i najzad 100 je tzv. prednost (na engleskom precedence)- to je neki prirodan broj, čiji smisao ćemo ubrzo objasniti.

Inače, za ma koji novouveden tip op-izraz, recimo imena ime članak oblika (Oper) izgleda

---

[1] U proloških knjigama se po pravilu kaže operator. Međutim, jasno je da su ta kao i prethodno navedena reč struktura prejake, neprimerene. To je tim pre tačno ako se na umu ima kako se te reči koriste u matematici.

```
:-op(predn[2],tip,ime).
```

i,kako rekosmo,pre upotrebe ime-izraza moramo staviti takav članak. Tu nas sada očekuje objašnjenje o tipu i o prednosti.Najpre o tipu.Imamo:

U slučaju unarnih izraza tip može biti

```
fx ili xf     -u slučaju relacija;
fy ili yf     -u slučaju operacija.
```

U slučaju binarnih izraza tip može biti

```
xfx           -u slučaju relacija
xfy ili yfx   -u slučaju operacija.
```

Ignorišući smisao prvog argumenta koji ukazuje na prednost navodimo nekoliko primera op-izraza --navodeći njegovu op-najavu, poput (Oper).Primeri:

:-op(200,xf,iza). Tako se najavljuje unarni relacijski op-izraz u kome je iza ime relacije.Recimo,X iza možemo definisati kao:X je nedativan.Formalno,u tu svrhu bismo mogli uvesti članak

```
, X iza:-X<0.
```

Relacijski znak pisan sufiksno jer je tip xf. Pošto je iza relacijski znak to je besmisleno koristiti zapis poput: X iza iza.

:-op(250,fy,dodaj) .Tako je najavljena unarna operacija imena dodaj.Recimo,dodaj X možemo zamišljati kao: X+1.Evo jednog primera u kome je to iskorišćeno. Reč je o predikatu vred definisanom ovim člancima

```
vred(X,X):-integer(X).
vred(dodaj X,Y):-vred(X,X1),Y is X1+1.
```

Vidi se da je vred u stvari relacija vrednost (izraza), i da se pritom dodaj X uzima kao X+1. Shodno tome,recimo,u pitanjima

```
?-vred(dodaj 7,X).
?-vred(dodaj dodaj 7,Y).
```

X,Y će redom biti 8,9. Primetite da budući da je dodaj najavljeno kao (prefiksni unarni) operacijski znak, a dakle ne relacijski znak, to smo smeli da pišemo: dodaj dodaj 7. Slično, da je dodaj bilo definisano kao sufiksni unarni operacijski znak, što znači da mu je onda tip jednak yf,smeli bismo da pišemo: 7 dodaj dodaj.

:-op(210,xfx,rel). Tako je najavljena binarna relacija sa imenom rel. Iza tog najavnog članka smemo recimo dodati ovakav članak

```
X rel Y :-X<3 ,3<Y.
```

i tako uvesti jednu binarnu relaciju rel. Kako je rel relacijskog tipa to su besmisleni zapisi kao: 5 rel 6 rel 8, 7 rel (7 rel 9).

:-op(400,yfx,fo). Tako je najavljena binarna operacija sa imenom fo. Znači, sada za razliku od prethodnog relacijskog slučaja smislena je upotreba zapisa poput X fo Y fo Z, A fo B fo C fo D, (A fo B) fo C ali, nije odmah jasno šta je njihov smisao.Recimo, da li li prvom fo-izrazu odgovara "zagradski" izraz ((X fo Y) fo Z) ili ovaj drugi (X fo (Y fo Z)) ? Budući da znak fo ima tip yfx njemu odgovara prvi zagradski izraz. Medjutim da mu je tip bio xfy,tada bi mu odgovarao drugi. Uopšte

Ako je X1 fo X2 fo X3 ... fo Xn ma koji "čisti" fo-izraz, tj. koji pored fo nema i neki drugi operacijski znak tada

[2] Reč prednost smo skratili na "predn".

njemu se pridruzuje zagradski izraz

(...(X1 fo X2) fo X3)... fo Xn), kad mu je tip yfx

odnosno izraz

(X1 fo (X2 ...(Xn-1 fo Xn)...))), kad mu je tip xfy.

Pretpostavimo sada da su operacijski izrazi fo, go uvedeni ovim najavama:

```
:-op(400,yfx,fo).
:-op(500,yfx,go).
```

Vidi se da je fo "prednosniji" od go, jer su kao prednosti redom dodeljeni brojevi 400,500. Evo kakvog smisla imaju te prednosti:

Neka je recimo, A fo B go C op-izraz koji pored fo,go ne sadrži i neki drugi operacijski znak. Tada u skladu sa tim da predn(fo) < predn(go) prvo "obavljamo radnju fo pa go", tj. taj izraz zagradjujemo ovako

(A fo B) go C.

Ali, ako imamo "čisti" fo-go izraz: A go B fo C, onda ga zagradjujemo ovako A go (B fo C). Možemo i ovako slobodnije reći, odnosno primetiti fo bi moglo da podseća na "množenje", a go na "sabiranje".

Uz vezi sa op-izrazima dodajemo i sledeće. Prvo, uglavnom se za prednost uzima broj u rasponu[3] 1,2,3,...,1200. Drugo, svaki od osnovnih prološkiih op-znakova kao što su +,*,-,is, not,=,i dr.ima svoj broj prednosti.Recimo, predn('*')=400.

4.4 G r a m a t i k e i P r o l o g

Na samom početku ove tačke pomenuli smo gramatičke izraze. Tamo smo naveli jedan jednostavan,ali po malo i "nategnut" primer na našem jeziku.Naime u skladu sa okolnošću da naš jezik ,pored ostalog, ima više padeža,ima razne oblike za vremena,za rodove i dr. dosta je teško napraviti jednostavne a prave primere na njemu.Sada navodimo primer gramatike na engleskom jeziku. Tu gramatiku čine ova, kaže se, pravila zamene (gramatički izrazi)

(4.4.1)

```
sentence-->noun_phrase,verb_phrase.
noun_phrase-->determiner,common_noun.
noun_phrase-->proper_name.
verb_phrase-->verb;verb[4],noun_phrase.
determiner-->[the];[a].
verb-->[likes];[looks].
common_noun-->[man];[woman].
proper_name-->['John'];['Vera'].
```

Najpre istaknimo da tako prološki zapisana gramatika se nebitno razlikuje

[3] U stvari,takvu pojedinost treba proveriti u zvaničnom Priručniku verzije Prologa sa kojim se radi.

[4] Ako se na tom mestu umesto verb stavi verb_phrase nastaje još složenija gramatika.

od gramatika pisanih na uobičajen način. U toj gramatici reči[5]:

(4.4.2) sentence, noun_phrase, determiner, common_noun, proper_name, verb, verb_phrase

su nezavršni (neterminalni) znaci, dok reči[6]

man, woman, 'John', 'Vera'

su završni(terminalni) znaci. Za njih se može reći da predstavljaju izvesne posebne vrednosti (primerke) nezavršnim znacima. Tako, reči man i woman su dva primerka za common_noun, tj. to su dva zajednička imena, dok John i Vera su primerci vlastitih imena. Dalje, the i a su članovi, a likes i looks su su glagoli. Sada navodimo jednu rečenicu (sentence) napravljenu tom gramatikom:

(*1) John likes a woman

Da je ta rečenica, taj zapis zaista "plod" gramatike (4.4.1) može se zaključiti ovako:

Rečenica (sentence) kao plod može da nastane jedino primenom prvog pravila iz (4.4.1). Elem, neki početni komad iz (*1) treba da bude noun_phrase (imenski izraz), a preostatak onda treba da bude verb_phrase (glagolski izraz). Za noun_phrase postoje dve mogućnosti: može da počinje nekim članom(determiner), ili da bude proper_name (vlastito ime). Ovde nastupa druga mogućnost i početak rečenice je vlastito ime John.
Dalje treba da vidimo da li preostatak rečenice (*1), tj. zapis

(*2) likes a woman

jeste verb_phrase. Prema pravilima gramatike (4.4.1) tu sada postoje dve mogućnosti: to je neki glagol -što očigledno nije- ili to je zapis oblika

verb, noun_phrase

Zaista, (*2) počinje sa likes i to jeste verb(glagol)[7] pa nam preostaje da raspravimo da li je

(*3) a woman

noun_phrase. To sada sledi na osnovu prvog noun_phrase--pravila, reč a je determiner(član), a reč woman je common_noun.
I tako konačno zaključujemo da je (*1) zaista rečenica uočene gramatike.

Neka je trenutno G ,poput (4.4.1), ma koja gramatika i Zap dati zapis poput (*1). Da bismo ubrzo rasuđivali "na prološki način", za taj Zap ćemo pretpostaviti da je zadat kao lista[8]. Tada jedno od uopšte osnovnih pitanja

---

[5] One po smislu redom odgovaraju ovim našim rečima:
rečenica, imenički izraz, član , zajednička imenica, vlastita imenica glagol, glagolski izraz.

[6] Reči John, Vera su stavljene između navodnica, da ne bi inače bile shvaćene kao proloske promenljive.

[7] Jer verb-pravilo kaže da su likes, runs glagoli.

[8] Recimo, (*1) u tom izdanju glasi: [John, likes, a, woman]

tzv. pitanje analize je

Raspravljanje da li Zap jeste ili nije rečenica gramatike G.

Drugo opšte pitanje, tzv. pitanje generativnosti :

Po nekom redu, nizanje svih rečenica gramatike G.

Zanimljivo je, da se, na način koji ćemo izložiti, može napraviti proloski program Prolog(G), koji je "nameren" rešavanju oba ta problema. Rekli smo "nameren", jer se može dogoditi da u slučaju izvesne gramatike G za rešavanje prvog od tih pitanja uopšte ne postoji algoritam, kada naravno program Prolog(G) će jedino bezuspešno pokušati da nadje odgovor. To naravno možemo i "oprostiti" tom programu. Ali, ako je u pitanju drugi problem i pritom gramatika G ima beskonačno mnogo "plodova", rečenica, onda -u duhu Prologa- nizanje rečenica po pravilu nije "iscrpljujuće", jer opšti proloski algoritam "uvek hvata najleviju granu" (videti s tim u vezi Zadatak 3.33 u kome se "nižu" razni termi.).

Da bismo lakše izložili pravljenje takvog Prolog(G), najpre ga pravimo u slučaju gramatike (4.4.1). Uz to zamislimo da nam je neko zadao listu Zap i da nam je zadatak da raspravimo da li je rečenica. Drugim rečima, hoćemo proloski da raspravimo pitanje analize te gramatike. Međutim, videćemo da je taj isti program i generativni, tj. da je sposoban i za rešavanje drugog pitanja.

Prva pomisao je kako nekim proloskim člancima opisati gramatiku (4.4.1). Čini se prirodnim da se svaki od nezavršnih znakova shvati kao svojevrsna relacija. Tako, recimo, sentence(X) bi značilo: X je rečenica. Odmah ističemo da se u duhu te misli može napraviti dobar proloski program. Tako, recimo, prvo pravilo gramatike (4.4.1) valja pretvoriti u ovaj proloski članak

```
sentence(X):-dopis1(A,B,X),noun_phrase(A),verb_phrase(B).
```

gde je dopis1 relacija definisana u Zadatku 3.12 (xii), kojom se data lista X razdvaja na početni komad A i preostatak B. Smisao tog članaka je:

X je rečenica ukoliko X je razdvojivo na A i B, i uz to
A je noun_phrase, a B je verb_phrase

Istaknimo da se ta zamisao, sa bitnim korišćenjem relacije dopis1, može do kraja ostvariti, i tako napraviti odgovarajući proloski program gramatike (4.4.1). Da bismo istakli ulogu relacije dopis1 taj program (i uopšte takav za ma koju gramatiku) ćemo zvati razdvojni.
Nije teško videti da se tim programom, recimo sa uspehom može potvrditi da je (*1) zaista rečenica gramatike (4.4.1). U većoj meri to je blagodareći okolnosti da program kojim se uvodi relacija dopis1(A,B,X) je generativan po sastavcima A,B, pa otuda u "proloSku igru" mogu stupiti svemogući "parčići" polaznog zapisa (*1). Ali, upravo u toj moći pretrage po svim sastavcima se krije i osnovni nedostatak zamišljenog programa. Da bismo to bolje uvideli zamislimo da uz gramatiku (4.4.1) dodamo još ovo pravilo

```
sentence-->[Pera,ide,u,skolu].
```

kojim se propisuje da i lista [Pera,ide,u,skolu] je rečenica i onda postavimo pitanje da li je to rečenica. Tada, budući da zamišljamo da je članak tog pravila dodat na sam kraj, Prolog će najpre upošljavajući prvi članak, tu listu bezuspešno razlagati na sve moguće načine --ne bi li ustanovio da je ona rečenica-- i tek posle dugog "proloSkog putešestvija" nakon pregranjavanja na tu drugu sentence-granu na kraju će sa uspehom utvrditi

da ta lista stvarno jeste rečenica.
Drukčije, kraće rečeno, osnovni nedostatak "razdvojnog" programa je što pri upošljavanju relacije dopisi, tj. pri razdvajanju, po potrebi redom ,dakle bez ikakvog kriterijuma, se prave sva moguća razdvajanja. Uz to u slučaju njegovog korišćenja za generisanje rečenica stalno će se pojavljivati potreba 'spajanja', dopisivanja dve liste. Svako takvo spajanje za sebe traži izvesne korake, odnosno vreme. Da li se svi takvi nedostaci mogu ukloniti ? Odgovor je potvrdan i u tu svrhu se koristi ideja tzv. "različnih" lista (differnce-lists). Da bismo dobili neku prvu predstavu o njima, zamislimo da razmatramo izvestan problem o listi A, i da se po prirodi stvari taj problem svodi na neke podprobleme za njene "komade" P,Q, gde je P njen početni, a Q njemu dopunski komad liste A. Tada, može biti mudro da se lista A eksplicitno izbaci iz igre, i da se kao njena zamena uvede "različna" lista od P,Q, koju inače možemo ostvariti na više načina. Evo nekih:

To je, po definiciji, dvočlana lista (P Q)
To je, u slučaju Edinburgške sintakse, neka struktura kao f(P,Q) ili neki op-izraz kao P\Q, što bukvalno podseća na razliku[9]

Sada ćemo izložiti, kako se ideja različnih lista koristi u pitanju gramatika. Naime, u slučaju ma koje gramatike G , na način koji opisujemo, napravićemo takav proloski program Prolog (G). Dajemo podrobniji opis u slučaju gramatike **(4.4.1)**, ali pritom ćemo iznositi razne opšte zamisli.
Pri opisu "razdvojnog" programa smo rekli da sve nezavršne znake shvatamo kao relacije. Ostajemo pri tom pristupu, ali sada su to binarne relacije ! Može se reći da su one po smislu malo "dublje". Neka dogovorno rel bude zajedničko ime za sve relacije(**4.4.2**). Tada nam je osnovna namera da rel kao relacija nad listama bude ovako definisana:

(4.4.3) rel(X,Y) važi ukoliko lista X je oblika [a1,a2,...,ak|Y] sa nekim k i uz to početni komad [a1,a2,...,ak] je u "staroj", unarnoj relaciji rel.

Recimo:

determiner([the,man,runs],[man,runs])
važi jer u listi [the,man,runs] početni komad [the] jeste "determiner" a preostatak je lista [man,runs], što je upravo drugi argument te binarne relacije determiner.

noun_phrase([a,man,likes,a,woman],X)
važi ako X=[likes,a,woman], jer početni komad [a,man] prvog argumenta jeste u "staroj" unarnoj relaciji noun_phrase, a X je onda preostatak prvog argumenta.

sentence(['John',likes,a,woman],[])
važi jer lista ['John',likes,a,woman] jeste početni komad prvog argumenta, sa preostatkom [] i taj početni komad jeste u "staroj" unarnoj relaciji sentence, što smo već bili utvrdili.

---

[9] Evo jednog malog problema u kome se pojavljuje misao slična "različnim" listama. Zamislite, da vam neko traži da za cele brojeve nađete mudar zapis blagodareći kome se sabiranje dva cela broja može neposredno izvesti. Jedan način, je da cele brojeve shvatimo kao razlike po dva prirodna broja kao 3-2, 9-7, 5-45 itd. Tada (!) imamo ovu "zlatnu" jednakost
(a-b) + (b-c) = (a-c)
U stvari, to je u osnovi zamisao različnih lista.

determiner([the|X],X) važi, čak za ma koje X.

Kao što vidite gornja definicija (4.4.3) prirodno "upliće" zamisao različnih lista. Naime, da smo recimo u definiciji (4.4.3) umesto rel(X,Y) pisali rel([X,Y]) imali bismo skoro bukvalnu pojavu takvih lista.
U pisanju programa Prolog(G), uopšte od osnovnog značaja je definicija oblika **(4.4.3)**. Shodno tome, recimo, pravilo

sentence-->noun_phrase,verb_phrase.

prevodimo u sledeći članak

sentence(X,Y):-noun_phrase(X,Z),verb_phrase(Z,Y).

što je upravo u duhu definicije **(4.4.3)**. Da bismo odmah uvežbali pravljenje takvog prevoda uopšte zamislimo za trenutak da treba da prevedemo ova tri pravila

a-->b,c.
a-->b,c,d.
a-->b,c,d,e.

gde su a,b,c,d,e nezavršni znaci. Tada prevodi redom glase

a(X,Y):-b(X,Z),c(Z,Y).
a(X,Y):-b(X,Z1),c(Z1,Z2),d(Z2,Y).
a(X,Y):-b(X,Z1),c(Z1,Z2),d(Z2,Z3),e(Z3,Y).

Znači, slobodnije rečeno sa leve strane su promenljive X, Y a sa desne se pojavljuje "putanja" oblika

X,Z1; Z1,Z2 ; i sl. a na samom kraju recimo Zk,Y
gde k govori o broju članova u putanji.

Da bismo dopunili opis prevođenja, zamislimo još, da treba, da prevedemo i ova dva "pravila"

a-->b.
a-->[the].

Njihov prevod glasi

a(X,Y):-b(X,Y).
a([the|X],X).

Primetite da je drugi članak bukvalno napravljen pomoću definicije (4.4.3)
Posle tog opšteg opisa dajemo program Prolog(G), gde je G gramatika(4.4.1):

(4.4.4)
```
sentence(X,Y):-noun_phrase(X,Z),verb_phrase(Z,Y).
noun_phrase(X,Y):-determiner(X,Z),common_noun(Z,Y).
noun_phrase(X,Y):-proper_name(X,Y).
verb_phrase(X,Y):-verb(X,Y).
verb_phrase(X,Y):-verb(X,Z),noun_phrase(Z,Y).
determiner([the|X],X).
determiner([a|X],X).
verb([likes|X],X).
verb([looks|X],X).
common_noun([man|X],X).
common_noun([woman|X],X).
proper_name(['John'|X],X).
proper_name(['Vera'|X],X).
```

Da bismo makar delom videli kako radi taj program, najpre pomocu njega će-

mo raspraviti da li lista ['John',likes,a,woman] jeste rečenica, što smo u stvari već prethodno bili učinili. Koristićemo "jednačinsko" pisanje[10]; zapeta će glumiti veznik i, a + veznik ili.Tako imamo:

```
sentence(['John',likes,a,woman],[])
    =noun_phrase(['John',likes,a,woman],Z),verb_phrase(Z,[])
        Za predikat noun_phrase imamo dve grane, obe uvodimo u "račun"
    =((determiner(['John',likes,a,woman],Z1),common_noun(Z1,Z))
                +
      (proper_name(['John',likes,a,woman],Z) ),verb_phrase(Z,[])
    =proper_name(['John',likes,a,woman],Z),verb_phrase(Z,[])
        Jer prva grana determiner(...) je netačna,što se brzo vidi.
    =verb_phrase([likes,a,woman],[])
        Jer na osnovu proper_name članka prvi sastavak je tačan pri
        Z=[likes,a,woman].
    =verb([likes,a,woman],[])
        +
      verb([likes,a,woman],Z),noun_phrase(Z,[])
        Jer za verb_phrase relaciju imamo dve grane
    =verb([likes,a,woman],Z),noun_phrase(Z,[])
        Jer netačna je prva grana
    =noun_phrase([a,woman],[])
        Jer verb([likes,a,woman],Z) je tačno upravo pri Z=[a,woman]
    =determiner([a,woman],Z),common_noun(Z,[])
                +
      proper_name([a,woman],[])
        Druga grana je netačna, ali to Prolog sada ne "vidi", jer najpre
        računa prvu granu
    =common_noun([woman],[]) + proper_name([a,woman],[])
        Jer na osnovu determiner-članaka dobije se Z=[woman].
    =Tačno
        Jer tačno je common_noun ([woman],[]). Kao što se vidi, pošto je
        prva grana tačna, račun je završen.
```

Da li se već i po tom primeru može zaključiti da je program (4.4.4) u stanju da pozitivno odgovori na pitanje oblika

(4.4.5) ?-sentence([...],[]).

gde je [...] izvesna lista koja je rečenica polazne gramatike (4.4.1) ? Ali,uz to da li je isti program sposoban da redom ispisuje rečenice gramatike (4.4.1) ?
Da bismo to lakše uvideli u vezi sa programom (4.4.4) istaknimo sledeće. Neka je rel ma koja relacija programa (4.4.4),tj. rel je jedna od relacija

[10] I pored nekih očigledno lepih svojstava "jednačinski" način pisanja ima i mana. Naime,"osetljiv" je na proceduru vraćanje(backtrackig), odnosno tim pisanjem ako se desi vraćanje mora se domišljati " šta je u nekom levom delu izraza" prethodno bilo, koje grane su korišćene i dr. Kratko, ako se tokom alogoritma ne pojavljuje vraćanje jednačinsko pisanje je izvrsno.

(4.4.2). Može se reći da program (4.4.4) svaku od tih relacija definiše sa po jednim ili dva članka (dakle,disjunkcijom),a u tim člancima se opet pojavljuju po neke od tih relacija. Međutim, bitno je:

> U tim definicijama, tj. člancima programa (4.4.4) drugi argument ma koje relacije je prava podlista prvog, a posle taj drugi je prvi argument naredne relacije članka i slično nadalje.Shodno tome, za program (4.4.4) se može reći da on relacije rel definiše rekurzivno po njihovim prvim argumentima.Međutim,baza,osnova tim rekurzijama nisu, odnosno ne moraju biti prazne liste[11], već su to nekolike liste koje počinju jednom od reči
>
> the,a,man,woman,'John','Vera'

Kratko rečeno, upravo blagodareći tim osnovnim svojstvima, program (4.4.4) je sposoban da "obavi opisane zadatke".Pomenimo još da program (4.4.4) detaljno razmatramo u Zadatku 7.1.

Kao što smo već rekli, u slučaju ma koje gramatike G odgovarajući proloski program se označava sa Prolog(G).Važno je istaći da je pravljenje programa Prolog(G) "ugrađeno" u Prolog u sledećem smislu:

> kad mi napravimo neku fajlu[12] ime.ext sadržaja jednakom nekoj gramatici G i ako posle toga u okviru Prologa pozovemo[13] tu fajlu onda se ne "uči tava" gramatika G, već prološki program Prolog (G), već napravljen "od strane Prologa".

To praktično znači da onda možemo postavljati razna pitanja u vezi sa tim Prolog(G), tj. praktično sa gramatikom G. Tako, ako je G već uočena gramatika (4.4.1), onda možemo ,primera radi, postaviti ova pitanja

```
?-sentence(['John',likes,the,woman],[]).
?-sentence(X,[]).
```

U slučaju drugog -ako nastavimo sa stalnim traženjem novog rešenja[14]- redom će se pojaviti razne rečenice gramatike (4.4.1).
Do sada pominjani gramatički izrazi (pravila) bi se mogli nazvati osnovni. Naime, Prolog dopušta još neke proširene mogućnosti u vezi sa gramatičkim izrazima. Najpre, uočimo ovu malu gramatiku

```
a-->b,"ABC",c.
b-->[pred].
c-->[iza].
```

Pre objašnjenja smisla prvog pravila ističemo da je " ABC " primer stringa (reči) i da se u Prologu on shvata kao lista $[\alpha,\beta,\gamma]$, gde su α,β,γ ASCII-

[11] Što je inače čest slučaj(recimo, u programu traženja dužine liste,spajanju dve liste i dr.).

[12] Tu je ime njen odgovarajući naziv(ime),a ext je zavisno od verzije prologa.Recimo, ari za ARITY-prolog, dec za LPA-prolog.

[13] To se po pravilu čini kucanjem : [ime.ext], naravno za određene vrednosti ime i ext.

[14] Znači kucamo ; u slučaju ARITY-prologa, odnosno kucamo znak " beline " u slučaju LPA-prologa.

kodovi redom znakova A,B,C, odnosno "ABC" je upravo[15] [65,66,67]. Sledstveno smisao prvog pravila je:

Neka lista je a-ovska, ako jedan njen početni komad je b-ovski, zatim dolazi podlista jednaka [65,66,67] i iza nje preostala lista c-ovska[16].

Shodno tome prvo pravilo se prevodi u ovaj prološki članak

```
a(X,Y):-b(X,[65,66,67|Z]),c(Z,Y).
```

Sada navodimo nekoliko pravila sa učešćem stringova i odgovarajuće prevode

Pravilo:	Prevod:
`a-->b,"A","B",c.`	`a(X,Y):-b(X,[65,66|Z]),c(Z,Y).` Znači, "A","B" isto je kao "AB".
`a-->b,"AB",c,"CDE".`	`a(X,Y):-b(X,[65,66|Z1]), c(Z1,[67,68,69|Y]).`
`a-->"AB",b.`	`a([65,66|X],Y):-b(X,Y).` Pazite, reč AB treba da bude početak od a.
`a-->"AB",b,"C".`	`a([65,66|X],Y):-b(X,[67|Y]).`

Druga proširenje u vezi sa gramatičkim izrazima je dopuštenje da se kao deo izraza pojavi zapis oblika { ... },oivičen vitičastim zagradama i gde umesto tačkica mogu doći neke obične prološke formule. Evo takvog malog primera: a-->b,{write('Pera'),nl,write('Dara')},c. Naravno, njegov prevog glasi a(X,Y):-b(X,Z),write('Pera'),nl,write('Dara'),c(Z,Y).

I najzad, poslednje proširenje se odnosi na dopuštenje da, nezavršni znaci sadrže argumente koji su neke promenljive. Mali primer je sadržan u ovoj gramatici

```
a(V)-->b(V),c(V).
b(1)-->[dobar].
b(2)-->[dobra].
c(1)-->[covek].
c(2)-->[zena].
```

Tada, a-plodovi te gramatike su [dobar,covek] i [dobra,zena]. Inače prološki prevod te gramatike se pravi neposredno. Naime, među argumente se uključuje i to V, što znači da se a,b,c shvataju kao ternarne relacije. Prevod glasi

```
a(V,X,Y):-b(V,X,Z),c(V,Z,Y).
b(1,[dobar|X],X).
b(2,[dobra|X],X).
c(1,[covek|X],X).
c(2,[zena|X],X).
```

Na kraju izlaganja o gramatikama navodimo jedan malo složeniji primer. On se odnosi na *,+ izraze bez zagrada, kao što su

```
2+3,  7*4+9,  5*2*2+3*4*3+6+8
```

čiji osnovni sastavci su cifre 0,1,2,..,9. Gramatika tog primera je sposobna da raspozna pravilan takav izraz i pride da mu izračuna vrednost. Ta gramatika glasi

```
(4.4.6) izraz(Z)-->proiz(X),"+",izraz(Y),{Z is X+Y}.
        izraz(Z)-->proiz(Z).
        proiz(Z)-->cif(X),"*",proiz(Y),{Z is X*Y}.
        proiz(Z)-->cif(Z).
        cif(X)-->[C],{"0"=<C,C=<"9",X is C-"0" }.
```

Njoj odgovarajući prološki program je

```
(4.4.7)  izraz(Z,A,B):-proiz(X,A,[43|E]), izraz(Y,E,B),Z is X+Y.
         izraz(Z,A,B):-proiz(Z,A,B).
         proiz(Z,A,B):-cif(X,A,[42|E]),proiz(Y,E,B),Z is X*Y.
         proiz(Z,A,B):-cif(Z,A,B).
         cif(X,[A|B],B):-[48]=<A,A=<[57],X is A-[48].
```

Najpre navodimo šta taj program može da uradi. Ako recimo postavimo pitanje

```
?-izraz(X,"2+5",[])
```

dobićemo X=7, što je vrednost izraza "2+5". Slično rade i ova pitanja

```
?-izraz(X,"2*5+7+4",[]), ?-izraz(X,"7+1*3*5+4",[])
```

Kratko rečeno program je sposoban da računa vrednosti pravilnih izraza. Uz to, ako je izraz nepravilan kao odgovor se pojavljuje no. Recimo, to će biti odgovor na ovo pitanje

```
?-izraz(X,"4++4",[])
```

Međutim, taj program nije generativan po izrazima. Glavni razlog je što je cif -pravilo samo proverno.

A sada nekoliko reči objašnjenja kako je taj primer sastavljen, odnosno koje zamisli su u pozadini.

Kao prvo ističemo, da relacija cif je u vezi sa "biti cifra". To se direktno vidi iz poslednjeg pravila u (4.4.6), jer ono poručuje da je jednočlana lista [C] cifra ukoliko je C između[17] reči "0" i "9", što u stvari znači da C treba da bude između 48 i 57, jer to su ASCII-kodovi sa znake 0 i 9. U tom pravilu se još kaže da je u formuli cif(X), to X, tj. vrednost jednaka C-"0", što recimo u slučaju formule cif("5") će dati

```
X="5"-"0"= ASCII("5")-ASCII("0")= 53-48= 5
```

Relacija proiz na određen način odgovara operaciji * , odnosno proizvodu. Slobodnije rečeno, ako kod gramatike (4.4.6) u trenutku ne uzimamo u obzir argument koji govori o vrednosti izraza, onda se ona približno može ovako opisati

Izraz je ili cifra ili proizvod ili proizvod iza koga dolazi izraz.

U duhu te zamisli je i sklopljena gornja gramatika. To se lakše vidi ako se za trenutak koristi funkcijski pristup ka pitanju vrednosti. Naime, tada nije teško videti da važe ove rekurzivne jednakosti

```
vred([A,+,P1,P2,...,Pk])=A+vred([P1,P2,...,Pk])
vred([a, *, b, P1,...,Pk])=vred([c,P1,...,Pk]), gde c=a*b.
```

---

[15] Čak ako se postavi pitanje ?-write("ABC") na ekranu se štampa [65,66,67].

[16] Budući da je navedena gramatika jednostavna lako je videti da u skladu sa rečenim njen jedini a-plod je lista [pred,65,66,67,iza].

[17] Podsećamo da je lista shvatljiva kao string. Tako, [48] i "0" su iste stvari.

5. DRUGO O PROLOŠKOM ALGORITMU

U ovom izlaganju ćemo u potpunosti opisati opšti proloski algoritam, doduše uz jedno malo ograničenje: sve je iskazano na jeziku lista. Shodno tome opis bukvalno odgovara Micro-prologu sa Lisp-sintaksom. Međutim to ograničenje uopšte nije bitno jer od ma koje strukture, poput f(a,b,....) možemo preći na odgovarajući listu [f,a,b,...] i dalje u prološkom algoritmu rasuđivati sa njom.

U nastavku prvo dolazi izlaganje o algoritmu unifikacije, zatim razna produbljivanja u vezi sa pitanjem promenljivih i nepoznatih, dalje razne procedure koje su sastavci opšteg prološkog algoritma i na kraju se izlaže sam taj algoritam.

5.1 Jedan algoritam ujednačavanja (unifikacije)

Kao što je dobro poznato u funkcijskim programskim jezicima[1] izvesna funkcija f(x,y,....) se po pravilu traži u nekoj potpuno odredjenoj tački, kao (2,3,...). Međutim, u vezi sa takvom funkcijom opet po pravilu je nemoguće postaviti pitanje traženja njene vrednosti u tačkama koje nisu konkretno zadane, već je samo zadan njihov o b l i k. Evo primera za objašnjenje. Recimo, u Pascalu možemo definisati funkciju f(x,y), gde su x,y,f(x,y) celi brojevi, na ovakav način (navodimo samo glavni deo definicije)

If x=y then f:=111 else f:=222.

Tada očigledno imamo ove jednakosti

f(3,3)= f(4,4)= f(5,5)= ...= f(123,123)= ... = 111,

odnosno uopšte f(p,p)= 111, gde p ma koji ceo broj, tj, drugim rečima funkcija ima vrednost 111 u svim tačkama o b l i k a (p,p). Ali, uprkos svemu tome ne sme se postaviti pitanje traženja f(p,p), ukoliko p nije konkretno zadano. I odmah istaknimo, Prolog se upravo u takvoj stvari razlikuje od svih funkcijskih jezika i jedna od njegovih suštinskih osobenosti je da

on dopušta rad sa "tačkama" izvesnih oblika.

Upravo u tu svrhu je u njega ugrađen algoritam ujednačavanja (unifikacije). Kratko rečeno, taj algoritam je "zadužen za pojam O B L I K A", odnosno sme se tvrditi da ga on na svoj način formalizuje, izražava . Algoritam unifikacije, koji ćemo dalje navesti, nije najkraći, ali s druge strane je veoma jednostavan i prirodan, podesan za "ručno" korišćenje pri praćenju rada konkretnih prološkh programa.

Opis koji dajemo se odnosi na unifikaciju ma kojih formula kao:

[1] Glavni predstavnik je Lisp, a uz neka odstupanja u takve jezike spadaju Pascal, C i dr.

fo(27,X56) a([X|Y],4) g(f(X),pera)

Tokom algoritma svaku od formula posmatramo i kao nisku (tj. konačan niz) njenih osnovnih sastavaka ,koji mogu biti:

znaci zagrada (,),[,]; znak zareza; znak | tzv. "list constructor", konstantske reči[2], znaci brojeva i promenljive

Recimo, prvonavedena formula shvaćena kao niska ima redom ove članove(sastavke)

fo (27 , X56)

Inače, reč niska u ovom izlaganju uopšte koristimo za ma koji konačan niz takvih sastavaka. Iz praktičnih razloga niske grafički prikazujemo rečima, ali pri tom se uvek mora voditi računa koji su pojedini njeni sastavci. Na početku algoritma se po pravilu zadaju dve formule, dve niske N1,N2, koje, kraće rečeno, treba unificirati, ako može. Tokom algoritma će se ta dvojka niski postupno menjati i u svakom koraku će se pojaviti nova dvojka N1,N2. Važno je istaći da te, tokom algoritma, pojavljene niske uglavnom neće biti formule, što i nije bitno, jer pravo rečeno algoritam koji izlažemo uopšte radi sa niskama.
Pre podrobnijeg prodora u algoritam navodimo neke primere. Recimo, formule

for(a,for,X) for(a,for,X)

su bukvalno (tj. sastavak po sastavak) jednake; elem ujednačive su. Promenljiva X može imati ma koju "vrednost". Međutim, formule

(*1) a(b,[X,sada,Y) a(b,[f(a),sada,Y])

nisu bukvalno jednake, jer 6-ti sastavci im se razlikuju (odnosno X i f), ali ako se obave ove zamene njihovih promenljivih X,Y

(*2) X-->f(a), Y-->f(a)

od tih formula će nastati bukvalno jednake, elem za te formule kažemo da su su ujednačive.
Naredno pitanje je strogo definisati šta znači ujednačiti(unificirati) dve date formule N1,N2.

Neka su N1,N2 dve bilo koje formule i neka je x1 ,..., xk spisak svih u njima učestvujućih promenljivih(može taj spisak biti i prazan). Sve te sintaksne promenljive imaju status nepoznatih. Tada, kažemo da su te dve formule ujednačive, ukoliko postoje |-termovi[3] ili formule term1, term2,..., termk tako da nakon obavljanja zamena (substitucija)

(*3) x1-->term1,x2-->term2,...,xk-->termk

[2] U konstantske reči, bez mogućnosti nastajanja zbrke, u delu 4.1 uključujemo i razne sistemske predikate Prologa, kao SUM,LOAD,KILL,.... u slučaju Lispovske sintakse, odnosno =, is, write, read itd. u slučaju Edinburgške.

[3] Tu podrazumevamo definiciju |-termova (4.1.1) u tački 4. Naravno, pretpostavljaju se nebitne promene, prouzrokovane okolnošću da je 2.4 pisano u Lisp-sintaksi, a ovaj tekst o unifikacijskom algoritmu je na Edinburgškoj sintaksi. Ovde smo izabrali tu drugu sintaksu, jer je onda algoritam malo složeniji i uz to --uz tehičke razlike-- on pokriva algoritam unifikacije u Lisp-sintaksi.

od formula N1,N2 nastaju dve bukvalno jednake formule[4]. I osnovni problem koji rešavamo je:

Pronaći algoritam kojim se za ma koje dve date formule N1,N2 može raspraviti da li su ujednačive, i ukoliko jesu odrediti i najopštiji spisak zamena oblika (*3), tj. vrednosti nepoznatih x1,....,xk.

Inače, najopštiji znači da iz njega kao poseban slučaj može da nastane svaki drugi spisak zamena, kojim se takođe ujednačuju formule N1,N2, što ćemo odmah objasniti. Recimo, formule (*1) su ujednačive pri zamenama (*2), ali te zamene nisu najopštije, odnosno najopštije su ove:

(*4) X-->f(a), Y-->Term (Term je ma koji |-term ili formula)

i zamene (*2) su očigledno njihov poseban slučaj.

Napomena 5.1.1. Pojam unifikacije(ujednačivosti) datih formula može se potpuno prirodno gledati jezikom matematičkih jednačina. Naime, ujednačiti dve formule N1,N2, čije sve promenljive su x1,...,xk, prosto znači rešiti termovsku[5] jednačinu

(*5) N1=N2

po nepoznatim x1,x2,...,xk ,koje mogu biti |-termi ili formule. Dalje, zamene oblika (*3) definišu, može se tako reći, formule rešenja jednačine(*5). Izraz "najopštiji" inače potpuno odgovara jednačinskom izrazu "opšte rešenje". Recimo, pitanje ujednačavanja formula (*1) se jezikom jednačina prevodi na pitanje rešavanja ove X,Y-jednačine

(*6) a(b,[X,sada,Y])=a(b,[f(a),sada,Y])

Tada zamenama (*2) odgovaraju ove formule rešenja

X=f(a), Y=f(a)

i one određuju jedno partikularno (posebno) rešenje jednačine (*6). S druge strane zamenama (*4) odgovara ova formula opšteg rešenja jednačine (*6)

X=f(a), Y=Term (Tu je Term ma koji |-term ili formula)

Primetimo da se ta formula može i ovako kraće zapisati

(*7) X=f(a)

uz prećutno podrazumevanje da Y, budući da za njega nemamo nikakvu jednakost, može biti proizvoljan. Da bismo takođe i dalje izlaganje imali logički povezanije ukratko ćemo tehnikom tipičnom za jednačine rešiti jednačinu (*6) i stići do formule (*7) opšteg rešenja. Pritom usput ćemo raditi sa odgovarajućim niskama, jer one su pogodne i za izražavanje |-termova, formula kao i ma kojih njihovih delova. Evo tog puta rešavanja

a(b,[X,sada,Y])=a(b,[f(a),sada,Y])
<-->(b,[X,sada,Y])=(b,[f(a),sada,Y]) Jezikom niski imamo ovo opšte tvrđenje
(ψ) Ako a1=b1 onda:

[4] Kako smo istakli x1,...,xk imaju status nepoznatih, pa (*3) određuje njihove vrednosti. Inače, u skladu sa tim, u (*3) smo umesto --> mogli staviti znak jednakosti.

[5] Tu smo radi lakšeg izražavanja kratko rekli termovska umesto termovsko-formulska.

Niska a1 a2 ... an = Niska b1 b2 ...bn
ako i samo ako
Niska a2 ...an=Niska b2 ...bn

<--> b,[X,sada,Y])=b,[f(a),sada,Y]) Po (ψ)

<--> ,[X,sada,Y])=,[f(a),sada,Y]) Po (ψ)

<--> [X,sada,Y])= [f(a),sada,Y]) Po (ψ)

<--> X,sada,Y])= f(a),sada,Y]) Po (ψ)

<--> ,sada,Y])= ,sada,Y])
X=f(a) To je malo suptilnije mesto. Naime, X kao promenljiva sme da ima vrednost koja je |-term ili formula, odnosno odgovarajuća niska, budući da radimo na jeziku niski. Shodno tome u niski f(a),sada,Y]) "otkidamo" početni komad, odnosno podnisku f(a), koja je formula i X-u dodeljujemo tu vrednost[6].

<--> sada,Y])=sada,Y])
X=f(a) Po (ψ)

<--> ,Y])=,Y])
X=f(a) Po (ψ)

<--> Y])=Y])
X=f(a) Po (ψ)

<-->])=])
X=f(a) Po (ψ)

<-->)=)
X=f(a) Po (ψ)

<--> X=f(a) Po (ψ)

Kraj **Napomene 5.1.1.**

Algoritam unifikacije izlažemo na nekoliko primera, ističući opšte crte. Kako rekosmo, formule shvatamo kao nizove njihovih osnovnih sastavaka. Inače u algoritmu se kao deo često pojavljuje ovaj podalgoritam (procedura):

Brisanje (jednakih početnih komada):
Njime se ,uz pretpostavku[7] a1=b1, a2=b2, ,ak=bk, ak+1≠ bk+1, od dve niske

[6] Jezik niski, koji ćemo koristiti u opštem opisu algoritma unifikacije, ima određenih dobrih strana, kao što su kraće pisanje "brisanjem" raznih već ujednačenih početnih komada i sl. Međutim, kad kao na tom mestu dođemo u položaj da promenljivoj damo vrednost, moramo voditi računa o raznim svojstvima |-termova i formula. Evo recimo jednog strogog objašnjenja zašto smo X pridružili tu vrednost. Dve formule su jednake ako i samo ako imaju ista imena i uz to redom su im jednaki argumenti. Shodno tome polazna jednačina je ekvivalenta sa konjunkcijom ovih jednačina
a=a, b=b, [X,sada,Y]=[f(a),sada,Y]
I sada koristimo činjenicu: dve liste su jednake ako i samo ako su im članovi po redu jednaki. I to odmah dovodi do X=f(a).

[7] Tu znak = predstavlja bukvalnu jednakost.

N1: a1 a2 ... ak ak+1 ... an
N2: b1 b2 ... bk bk+1 ... bn

prelazi na ove dve

N1: ak+1 ... an
N2: bk+1 ... bn

U algoritmu se polazi od dve , za unifikaciju, date formule N1,N2 (shvaćene kao niske). Dalje, tokom algoritma se te niske menjaju, a ukoliko je moguća unifikacija kao završni plod se,poput (*3), dobije spisak zamena, u kraćoj oznaci Zamen. Na samom početku algoritma skup Zamen je prazan. U svakom koraku algoritma se pojavljuje trojka oblika

N1
N2
Zamen

Primer 5.1.1. Unificirati formule N1,N2 navedene niže

N1: a(X)
N2: a(55)
Zamen: prazan

Formule N1,N2 su navedene kao reči, ali naravno očigledno je šta su im osnovni sastavci.
Prvo upošljavamo proceduru Brisanje jednakih početnih komada.Dobije se ova nova trojka

N1: X)
N2: 55)
Zamen: prazan

Došli smo do dve niske N1,N2 čiji prvi članovi su različiti, odnosno došli smo , tako ćemo reći, do mesta nesklada. Odmah recimo da uopšte

nesklad može biti ili otklonjiv ili neotklonjiv.

U ovom primeru je otklonjiv, jer prvi član od N1 je X, a to je promenljiva (nepoznata),a prvi član druge niske je 55, pa je radi otklanjanja nesklada dosta da se nepoznatoj X da vrednost(zamena) 55. Upravo tako i činimo.
Sledstveno skup Zamen sada će imati jedan član,jednu zamenu:

X--->55

Našavši vrednost za X sada u obe niske N1, N2 zamenjujemo X sa 55. I sada imamo ovu trojku

N1: 55)
N2: 55)
Zamen: X-->55

Sada opet primenjujemo proceduru Brisanje .Obe niske N1,N2 "iščeznu".Sledstveno,zaključak je

Polazne formule su ujednačive pri zamenama Zamen.

Već na tom mestu možemo iskazati veliki deo algoritma unifikacije. Naime, ako su date dve formule za unifikaciju N1, N2 onda prvo upošljavamo proceduru Brisanje jednakih početnih komada. I tada se može dogoditi jedan od ovih slučajeva:

(*8) Sl1 Stigli smo do nekih niski N1,N2 čiji početni članovi, u oznaci,

prvi, drugi su medusobno različiti, odnosno stigli smo do mesta nesklada. Kao što ćemo u daljem izlaganju više objasniti, tada su moguća dva podslučaja.

(i) Nesklad prvi,drugi je neotklonjiv. Tada se čitav algoritam unifikacije završava bezuspešno, odnosno polazne formule nisu ujednačive.

(ii) Nesklad prvi,drugi je otklonjiv. Tada kao što ćemo videti, izuzev u jednom slučaju,neka nova promenljiva dobije zamenu(vrednost), što se na odreden način iskoristi; a posle se sa novim niskama N1, N2 opet ide na Brisanje jednakih početnih komada.

Sl2 Došli smo do kraja obe niske[8], tj. obe niske su "izčezle". Tada su polazne formule bukvalno jednake, elem i ujednačive.

Primer 5.1.2. Unificirati formule N1,N2 navedene niže

N1: f(pera,g([a,b],X,9),Y)
N2: f(pera,g([a,b],X,9),Y)
Zamen: prazan

Koristimo proceduru Brisanje. Nastaje Sl2 iz (*8), odnosno N1, N2 su bukvalno jednaki.To praktično znači da će od njih uvek nastati jednake formule ako im se promenljive X,Y po želji zamene nekim l-termima,ili formulama.

Primer 5.1.3. Unificirati formule N1,N2:

N1: gf23(X,[a,b,pera],....)
N2: gf23(X,[a,b,jova],....)
Zamen: prazan

gde tačkice ... označavaju nenavedene delove.
Upošljavamo proceduru Brisanje .Nastaje Sl1 iz (*8),odnosno nove niske N1, N2 glase:

N1: pera],...)
N2: jova],...)
Zamen: prazan

Nesklad čine dve različite konstantske reči pera,jova. Taj nesklad je neotklonjiv[9] i algoritam se završava porukom: polazne formule nisu ujednačive.

Primer 5.1.4. Unificirati formule N1,N2:

N1: f(23,[pera,g(jova,Y),[23,45],67],X)
N2: f(23,X,[pera,g(jova,[b|Z]),[23,45],67])
Zamen: prazan

Upošljavamo proceduru Brisanje. Nova trojka je

(*9)
N1: [pera,g(jova,Y),[23,45],67],X)
N2: X,[pera,g(jova,[b|Z]),[23,45],67])
Zamen: prazan

---

[8] Algoritam unifikacije podrazumeva da su mu na početku zadane prave, tj. sintaksno ispravne formule. Sledstveno se može dokazati da nije moguće da se dođe do kraja samo jedne od niski.

[9] Jer kako inače postići da važi jednakost pera=jova ?

Nesklad čine znak leve uglaste zagrade [i promenljiva X. Pošto je jedan od njih, upravo drugi, promenljiva to nesklad je možda otklonjiv. Naime, kako je prvi leva uglasta zagrada, onda pomišljamo da je to početak nekog I-terma koji bi mogao da bude pridružena zamena za X. Odgovarajući I-term odredićemo jednom opštom idejom-- idejom brojanja zagrada.
Naime, [je prvi član prve niske. Elem, u toj niski treba da "otkinemo" traženi I-term. U tu svrhu:

> Od leve zagrade [idemo udesno po članovima prve niske, brojimo znake leve i desne zagrade, stanemo kad dođemo do desne zagrade] i uz to su se ujednačili brojevi levih i desnih zagrada[10].

Na takav način se od niske N1 otkine ovaj podniz

[pera, g(jova, Y), [23, 45], 67]

kome pridružimo odgovarajući I-term (oni se grafički i ne razlikuju). Tako dobijeni I-term zvaćemo kandidatska zamena promenljive, ovde upravo od X. Za njega smo rekli da je kandidatska zamena jer pitanje je da li je logički moguća zamena:

X-->[pera, g(jova, Y), [23, 45], 67]

Naravno, verovatno ste se već začudili u čemu bi mogla da bude nekakva nelogičnost. U stvari, ta zamena jeste logički "čista", ali pretpostavimo da se kod polaznih formula N1, N2 ne pojavljuje Y, već da umesto njega stoji X. Tada bismo umesto gornje zamene imali ovu:

X-->[pera, g(jova, X), [23, 45], 67]

A ta zamena je očigledno logički nemoguća, jer X učestvuje u svojoj "kandidatskoj zameni", odnosno termu

[pera, g(jova, X), [23, 45], 67]

Uopšte,

(*10) Kad se tokom algoritma nekoj promenljivoj Var dodeli kandidatska zamena Zam, promenljiva Var ne sme biti član od Zam.

Odnosno, logički je jasno da ako taj uslov nije ispunjen, algoritam unifikacije mora da stane sa zaključkom da polazne liste nisu ujednačive.
Međutim, tu sada dolazi jedno veliko ALI. Naime, iako je (*10) potpuno logičan zahtev u Prologu se po pravilu taj uslov ne postavlja, odnosno algoritam unifikacije ugradjen u Prolog (iz vremenskih razloga) ne vrši proveru da li je neka promenljiva član svoje kandidatske zamene. Recimo, ako Prologu postavite pitanje

?((EQ _X (2|_X)) odnosno ?-X=[2|X]

dobićete odgovor da. A ako čak tražite da se štampa takvo X dobićete nezaustaviv zapis [2|[2|[2|[, itd.
Nastavimo sada dalje algoritam. Bili smo stigli do trojke (*9) i da bismo otklonili nesklad, promenljivoj X pridružili smo navedenu zamenu. Da bismo odmah dobili i opšte pravilo, promenljivu ćemo zvati Var, a dobijenu zamenu sa Zam. I tada u narednom koraku uradimo sledeću proceduru:

(*11) Zamena promenljive

[10] Koristi se činjenica da svaki I-term mora imati isti broj levih i desnih zagrada [,].

(1) Niski, čiji početni komad je Zam, "otkinemo" taj čitav Zam(tj. sve njegove sastavke), a drugoj niski "otkinemo" njen prvi član, koji je promenljiva.
(2) Kod dobijenih niski svuda promenljivu Var zamenimo sa Zam.
(3) U skupu Zamen svim njegovim članovima promenljivu Var zamenimo sa Zam i još zamenu Var-->Zam priključimo novom skupu Zamen.

Primenom (*11) u razmatranom primeru dobijemo ovu trojku:

N1: ,[pera, g(jova, Y), [23, 45], 67])
N2: ,[pera, g(jova, [b|Z]), [23, 45], 67])
Zamen: X--> [pera, g(jova, Y), [23, 45], 67]

Opet upošljavamo proceduru Brisanje. Dolazimo do ove trojke

N1: Y), [23, 45], 67])
N2: [b|Z]), [23, 45], 67])
Zamen: X-->[pera, g(jova, Y), [23, 45], 67])

Sada nam se pojavio sličan nesklad kao u (*9) : promenljiva i znak [. Evo opšteg opisa šta raditi tada:

(*12) Ako nesklad čine neka promenljiva i znak [onda u niski koja počinje sa [idejom brojanja zagrada "otkinemo" odgovarajuću kandidatsku zamenu Zam.

Idejom brojanja zagrada lako se dolazi do ove kandidatske zamene [b|Z]. Uslov (*10) je ispunjen pa činimo (*11). Tako dolazimo do ove nove trojke

N1: ,[23, 45], 67])
N2: ,[23, 45], 67])
Zamen: X-->[pera, g(jova, [b|Z]), [23, 45], 67])
Y-->[b|Z] (Primetite da je u staroj zameni za X, Y zamenjeno novo-dobijenom vrednošću)

Opet upošljavamo proceduru Brisanje. Sada se dođe do kraja obe niske, tj. one "iščeznu". Znači, algoritam je završen sa uspehom, a skup Zamen je "zapamtio" tražene zamene (ujednačavanja polaznih formula).
Kao što se primećuje u ovom izlaganju algoritma unifikacije kao osnovno preostaje potpuno raspravljanje pitanja nesklada. Pretpostavimo, s tim u vezi da su članovi nesklada označeni sa prvi, drugi. Postoje dve mogućnosti:

Bezpromenljivski slučaj, tj. nijedan od prvi, drugi nije promenljiva
Promenljivski slučaj, bar jedan od prvi, drugi je promenljiva.

Razmatramo prvi slučaj. Tada:
Nesklad je neotklonjiv izuzev u slučaju kad su članovi nesklada upravo znak | i znak zapete.

Slučaj zapete i znaka | je u direktnoj vezi sa ovakvom opštom jednakošću

(*13) [a1, a2, ..., ak, ..., an] = [a1, a2, ..., ak|[ak+1, ...]],

Recimo, konkretno imamo ove jednakosti:

[a, b, c, d] = [a, b|[c, d]]; [a, b, c] = [a|[b, c]],
[a, b, [p, [q, r]], d] = [a, b|[[p, [q, r]], d]]

U otklanjanju nesklada jednakost (*13) se ustvari tako koristi što se lista sa leve strane te jednakosti zamenjuje njenom desnom stranom. To grafički gledano znači da se zarez koji stoji iza ak zamenjuje znakom | i iza njega se stavi i znak leve zagrade [. Ali, ta leva zagrada ima i svoju des-

nu, to je ona na kraju druge strane jednakosti (*13). Kako je prepoznati u okviru neke formule ? Opet brojanjem zagrada [,]. Naime, ako pođemo od zapete iza ak onda pošto razni članovi ak+1,..,an za sebe mogu sami sadržati razne zagrade[11] [,] to traženo mesto desne zagrade] možemo ovako naći:

(*14) Idemo od zareza udesno i brojimo leve i desne uglaste zagrade. Stanemo na mestu gde je broj desnih zagrada za 1 veći od broja levih.

E upravo, iza tog mesta treba postaviti znak desne zagrade. Evo primera u kome će se dvaput pojaviti slučaj (i).

Primer 5.1.5. Unificirati formule

N1: f(a,[p,q,[r,s],t])
N2: f(a,[p,q|[X|Y]])
Zamen : prazan

Nakon brisanja jednakih početnih komada dođe se do ove trojke

N1: ,[r,s],t])
N2: |[X|Y]])
Zamen : prazan

Pojavljuje se nesklad tipa |,.U skladu sa gore rečenim, prvo umesto zareza stavimo |[, a onda za prvu nisku radimo (*14). Tako se dođe do desne zagrade, one baš iza t. Tu stavimo znak].Tako od gornje trojke imamo ovu novu

N1: |[[r,s],t]])
N2: |[X|Y]])
Zamen : prazan

Primetite da obe niske počinju sa |,što je u skladu sa našim delanjem. Naravno da smo malo drukčije postupili to se ne bi desilo. Ali to nije tako bitno, u opštem opisu otklanjanja nesklada tipa | , to ćemo izbeći. Nastavljamo dalje algoritam opet brisanjem jednakih početnih komada ubrzo dolazimo do ove trojke

N1: [r,s],t]])
N2: X|Y]])
Zamen : prazan

Sada imamo nesklad tipa: promenljiva i leva zagrada [, tj. slučaj (*12). Lako se zaključuje da se taj nesklad otklanja prihvatanjem zamene

X-->[r,s]

Uradimo (*11) i dolazimo do ove trojke

N1: ,t]])
N2: |Y]])
Zamen : X-->[r,s]

Sada opet nastaje slučaj | ,.Postupajući kao i maločas kad smo takođe imali taj slučaj prelazimo na ovu trojku (znak | kao prvi nismo pisali)

N1: [t]]])
N2: Y]])

[11] Usled pretpostavljene sintaksne ispravnosti formula sa kojima u algoritmu radimo, svaki od tih članova mora da sadrži paran broj tih zagrada.

Zamen : X-->[r,s]

Opet imamo slučaj (*12). Nesklad se otklanja prihvatanjem zamene Y-->[t], a nova trojka glasi

N1:]])
N2:]])
Zamen : X-->[r,s]; Y-->[t]

Brisanjem jednakih početnih komada obe niske iščeznu, pa se algoritam završava uspešno, a skup Zamen sadrži tražene zamene (kojima se ujednačuju zadane formule).

Evo sada opšteg opisa otklanjanja nesklada tipa | , :

(*15) Ako nam se tokom algoritma pojavi ovakva trojka[12]

N': , a1 a2 ... ak] b1 b2 ...
N'': | c1 c2 ...
Zamen : ...

gde redosled N',N'' nije bitan, tada polazeći od zapete, tj. prvog člana niske N', idemo udesno, brojimo leve i desne uglaste zagrade i stanemo na mestu one desne zagrade] gde je broj desnih postao za 1 veći od broja levih. Neka je to mesto koje se nalazi upravo iza ak na gornjoj skici. Tada gornju trojku zamenjujemo ovom

N' : [a1 a2 ak]] b1 b2 ...
N'': c1 c2 ...
Zamen : ...

Sada razmatramo slučaj promenljivskog nesklada. Taj slučaj smo prethodno već sretali. U tom slučaju se , kao što ćemo videti, jednoj promenljivoj u oznaci Var najpre odredi kandidatska zamena Zam. Taj Zam je inače izvestan term ili formula. Niske se pritom mogu zamisliti u obliku

N' : Var ...
N'': Zam ...

gde redosled nije bitan, i gde je Zam neki početni komad (neki podniz)niske N''. I tada se prvo proverava uslov (*10). Ako nije ispunjen, čitav algoritam se prekida sa zaključkom da polazne formule nisu ujednačive. U protivnom, se uradi procedura Zamena promenljive (*11).

Kao što vidite, preostaje da se objasni kako se promenljivoj određuje kandidatska zamena. Razlikujemo ove podslučajeve

(*16) (j) Oba člana nesklada su promenljive
(jj) Iza promenljive stoji znak (
(jjj) Iza promenljive ne stoji znak (.

Razmatramo slučaj (j). Dogovorno sa Var1, Var2 označimo promenljive, članove nesklada. Neka su one, recimo, redom _x, _y. Jasno je da sada postoje tačno dve mogućnosti za zamenu

_x-->_y ili _y-->_x

Na tom mestu, da bismo obezbedili jedinstvenost rezultata, dogovorno ćemo prihvatiti zamenu Var1 --> Var2. Da budemo precizniji:

[12] Tačkice označavaju nenavedene članove. Inače, skup Zamen se ne menja tokom opisane transformacije.

Ako se postavlja pitanje unificiranja N1 sa N2, tj. insistira se na tom tom redosledu, onda se prihvata baš takva zamena[13]

I tako, Var2 proglašavamo kanditatskom zamenom. Drugim rečima Var1 uzimamo za Var, a Var2 za Zam (i posle sa tim Var i Zam se ide na (*11)).
Da bismo razmotrili slučaj (jj) najpre uočavamo sledeći primer.

Primer 5.1.6. Treba da unificiramo ove dve formule

N1: a(55,X(7,8))
N2: a(55,b(7,8))

Tada nakon brisanja jednakih početnih komada dolazimo do ove trojke

N1: X(7,8))
N2: b(7,8))

Očigledno nesklad se otklanja prihvatanjem zamene X-->b. Novodobijene niske su bukvalno jednake. Tako:

Polazne formule su ujednačive pri zameni X-->b.

Međutim, tu moramo upozoriti da prethodno zaključivanje ne prihvataju svi Prolozi. Naime, u jednoj od gornjih formula, upravo u N1, ime jedne podformule je X, tj. promenljiva. To mnoge verzije Prologa ne dopuštaju, recimo Arity spada u takve. Sledstveno kod tih verzija Prologa podslučaj (jj) ne može da nastane. Međutim, recimo, LPA-prolog dopušta da ime neke formule bude promenljiva pa prihvata i mogućnost slučaja (jj). Evo sada opšteg opisa slučaja (jj):

Ako tokom algoritma dodemo do trojke (redosled niski nije bitan)

N' : Var (b1 b2 ...
N'': a1 a2 ...
Zamen: ... (Tačkice znače neke nenavedene članove)

gde je Var promenljiva, onda za kandidatsku zamenu Zam uzimamo a1.

Sada razmatramo podslučaj(jjj). Označimo sa Var dotičnu promenljivu a njenu nisku sa N', a onu drugu sa N''. Označimo sa Drugi početni član te druge niske. Znači niske zamišljamo u obliku

N' : Var ...
N'': Drugi ...
Zamen : .... (Tačkice znače neke nenavedene članove)

I sada treba u toj drugoj listi "otkinuti" kandidatsku zamenu Zam. Razlikujemo ove podslučajeve:

(ψ1) Drugi je [.
Tada idejom brojanja zagrada " otkinemo " odgovarajući |- term, (vid. (*12)) kao kandidatsku zamenu Zam.

(ψ2) Drugi je broj.
Tada taj broj je kandidatska zamena Zam.

(ψ3) Drugi je konstatska reč, recimo u oznaci, Kon. Tada su moguća dva podslučaja:

(ψ31) Iza Kon naredni sastavak nije znak leve zagrade (
Tada taj Kon je kandidatska zamena Zam.

(ψ32) Iza Kon se nalazi (.Niska N'' je onda oblika

[13] A jasno je ,ako bi se tražilo da se N2 unificira sa N1, onda bi se na tom mestu prihvatila zamena Var2-->Var1.

N'': Kon (b1 b2 ... bk) ...
Tada polazeći od znaka leve zagrade (idemo "udesno" po članovima N'', brojimo zagrade (,) i stanemo kad dodemo do one zagrade) na čijem mestu su se ujednačili brojevi zagrada[14]. Recimo, da je to slučaj sa) iza bk. Tada, se za Zam uzima ova formula
Kon(b1,b2,...,bk)

Navodimo primer za ilustraciju podslučaja (ψ32).

Primer 5.1.7. Unificirati date formule N1,N2

N1: a(X,b([Z|g(b,23)]))
N2: a(b([Y|Z]),X)
Zamen: prazan

Brisanjem jednakih početnih komada dode se do ove trojke

N1: X,b([Z|g(b,23)]))
N2: b([Y|Z]),X)
Zamen: prazan

Sada imamo slučaj (ψ32). Nesklad se otklanja prihvatanjem zamene

X-->b([Y|Z])

jer uslov tipa (*10) je ispunjen. Nakon procedure Zamena promenljive dode se do ove trojke

N1: ,b([Z|g(b,23)]))
N2: ,b([Y|Z]))
Zamen: X-->b([Y|Z])

Sada opet nastaje procedura Brisanje. Dode se do ove trojke

N1: Z|g(b,23)]))
N2: Y|Z]))
Zamen: X-->b([Y|Z])

Nesklad čine dve promenljive. Po rečenom prihvata se zamena

Z-->Y

Uslov (*10) je trivijalno ispunjen. Nakon procedure Zamene promenljive dolazimo do ove trojke

N1: |g(b,23)]))
N2: |Y]))
Zamen: X-->b([Y|Y])
Z-->Y

Opet nastaje Brisanje. I posle toga opet nastane nesklad tipa (ψ32). Na samom kraju će se zaključiti da su date formule ujednačive pri zamenama:

X-->b([g(b,23)|g(b,23)])
Y-->g(b,23)
Z-->g(b,23)

Sada možemo dati opšti opis algoritma unifikacije

[14] Tako delamo, jer u slučaju (ψ32) zamišljamo da se u niski N'' može otkinuti početni komad oblika Kon(...),koji je neka formula.

(S.1.1) Algoritam unifikacije formule N1 sa formulom N2:

Skup Zamen je prazan.

(i) Upošljavamo proceduru Brisanje jednakih početnih komada.
Ako se dolazi do kraja obe niske N1,N2 algoritam se završava uspešno, a skup Zamen sadrži tražene zamene ujednačavanja. Posebno, ako je Zamen prazan skup polazne formule su bukvalno jednake.
Ako se ne dolazi do kraja niski N1,N2 onda se pojavljuje nesklad,koji je ili bezpromenljivski ili promenljivski. Ako je nesklad neotklonjiv, algoritam se završava bezuspešno, tj. sa porukom da polazne formule nisu ujednačive.
Bezpromenljivski nesklad koji nije sastavljen od znaka | i znaka zapete je neotklonjiv.
Ako je bezpromenljivski nesklad sastavljen od znaka | i znaka zapete, onda uradimo (*15) i sa novom trojkom N1, N2, Zamen idemo na (i). U slučaju promenljivskog nesklada, tada se držimo (*16) tj. slučajeva tamo opisanih. Sve u svemu jedna promenljiva Var dobije svoju kandidatsku zamenu Zam. Tada ako uslov tipa (*10) nije ispunjen nesklad je neotklonjiv (algoritam se završava bezuspešno, tj.sa porukom da polazne formule nisu ujednačive). U protivnom, uradimo proceduru Zamenjivanje promenljive i nakon toga sa novom trojkom N1,N2, Zamen idemo na (i).

Uz taj opšti opis dodajmo i sledeće. Može se dogoditi da tokom unifikacije neka promenljiva Var ne dobije nikakvu zamenu. Jasno to je znak da će ujednačavanje nastupiti ukoliko takva promenljiva ima ma koju vrednost. Takvu promenljivu ćemo zvati slobodna.

Napomena 5.1.2. Očigledno je da opisani algoritam 5.1.1 unifikacije formule N1 sa formulom N2 ima jedinstven rezultat. Naime, gledajući algoritam korak za korakom vidi se da je prisutna jedinstvenost sve dok se ne dođe do slučaja nesklada oblika

Var1,Var2 gde su Var1,Var2 promenljive,

kada su logički moguće dve zamene. Ali, prihvatanjem zamene Var1-->Var2 je takva dvojba otklonjena,pa sledstveno imamo jedinstvenost toka algoritma i rezultata. Međutim,da nismo precizirali redosled onda bismo na kraju i to u slučaju pojave slobodnih promenljivih mogli dobiti nebitno različite skupove Zamen. Evo jedan mali primer. Recimo,da smo pri nekom problemu unificiranja na kraju dobili ovaj skup zamena

```
Zamen:   X-->g(a,Y)
         Z-->Y
```

gde je očigledno Y slobodna promenljiva. Tada jedina druga mogućnost za taj skup je:

```
Zamen1:  X-->g(a,Z)
         Y-->Z
```

gde je sada Z slobodna promenljiva. Možemo slobodnije reći,da budući da smo u polaznom skupu Zamen imali zamenu oblika

Z-->Y

gde su Z, Y promenljive i uz to Y je slobodna, onda taj skup Zamen možemo "prepraviti" u nov skup Zamen1 zamenom Y sa Z, što ujedno znači da Y prestaje da bude slobodna promenljiva već to svojstvo prepušta promenljivoj Z. Ne opisujući opšti slučaj,samo pomenimo da su takve razmene neke slobodne promenljive nekom drugom promenljivom (koja onda postaje slobodna) jedini načini kojim se od jednog skupa Zamen dobiju svi drugi.
Kraj Napomene 5.1.2.

5.2 Pitanje promenljivih i nepoznatih; Dodelnik; Procedura spajanje

U vezi sa promenljivim u Prologu i do sada smo često govorili[15]. Želeći da najpre istaknemo jednu bitnu činjenicu uočavamo sledeći mali primer.

Primer 5.2.1. Dat je program ((p _x _y)(SUM _x _y 5)). Dati odgovor na pitanje: ?((p 3 _x)(PP _x))

Rešenje. To je skoro trivijalan primer, ali krenimo redom. Naime, ako prvo pokušamo unificiranje (p 3 _x) sa (p _x _y) doći ćemo do zamena

_x-->3, _y-->3

pa pošto zbir brojeva 3 i 3 nije 5 odgovor na pitanje će biti ne i neće se doći do dela (PP _x), tj. na ekranu se neće ništa pojaviti. Međutim, s druge strane budući da važi(p 3 2) vidi se da nešto nije u redu sa navedenim tokom algoritma, odnosno verovatno je napravljena neka greška. Objašnjenje je veoma prirodno:

U datom članku

(*1) ((p _x _y)(SUM _x _y 5))

sintaksne[16] promenljive _x, _y su u stvari stvarne promenljive, odnosno one smeju biti zamenjene[17] ma kojim vrednostima, ma kojim drugim objetima. Taj članak je po svojoj prirodi svojevrsni "identitet". U skladu sa tim taj članak je mogao da bude dat tako da se umesto _x, _y koriste neke druge dve promenljive kao _u, _v. Tada bismo imali članak

(*2) ((p _u _v)(SUM _u _v 5))

Za njega ćemo reći da je nastao iz (*1) preznačavanjem promenljivih. Ako sada sa tako preznačenim člankom pristupimo datom pitanju, što znači da ćemo unificirati (p 3 _x) sa (p _u _v) onda će se ubrzo za _x dobiti 2, a na ekranu će se štampati 2.

Iako je taj primer jedostavan on sadrži ovakvu opštu poruku, opšte pravilo

(5.2.1) Pre upotrebe ma kog članka nekog programa obavlja se preznačavanje njegovih promenljivih novim, odnosno promenljivim nekorišćenim do tog koraka algoritma.

Pored toga, i ako smo to i ranije isticali, ponovo podvlačimo:

(5.2.2) Prološke promenljive koje učestvuju u nekom članku su stvarne pro-

---

[15] Posebno videti Napomene 2.1.1 i 2.2.1.

[16] Tu reč 'sintaksne' ukazuju da je reč o promenljivim uvedenim definicijom (1.4) - kao svojevrsne reči, dakle kao sintaksne tvorevine.

[17] U stvari, svojstvo 'zamenljivosti' je toliko bitno za promenljive, da bi se, s tim u vezi, bez velike misaone štete reč promenljiva mogla izbaciti i zameniti reč 'zamenljiva'.

menljive, shodno tome svaki članak je svojevrsan identitet.

Međutim, za razliku od rečenog, kao što smo i do sada isticali[18], prološke promenljive upotrebljene tokom algoritma uopšte nisu promenljive, već su u stvari nepoznate. Recimo, u gornjem pitanju ?((p 3 _x)(PP _x)) _x je nepoznata[19]. Tako je i uopšte sa prološkim promenljivim u ma kom pitanju. Shodno rečenom imamo ovo zapažanje:

(5.2.3) Tokom ma kog prološkog algoritma korak-za-korakom se javlja relacijsko računanje vrednosti neke formule for(X1,...,Xk), pri čemu svako od tih Xi, koji je promenljiva u smislu definicije (1.4), uopšte nije stvarna promenljiva već jedino nepoznata.

Iduće pitanje u vezi sa prološkim promenljivim, i to onda kad su one u stvari nepoznate, je da li je tokom prološkog algoritma dozvoljeno da se nepoznata uklanja, zamenjujući je vrednošću koju je usput dobila ? Da to ne mora biti ispravno pokazuje sledeći primer.

Primer 5.2.2. U vezi sa programom

(*3) ((a 1)) ((a 2)) ((a 3))

postavimo pitanje

(*4) ?((a _x)(PP _x)FAIL)

koje, u stvari, iziskuje "putovanje" po svim _x -rešenjima formule (a _x). Šta se dešava ako se usput _x zamenjuje dobijenom vrednošću?

Odgovor. U prvom koraku formula (a _x) se ujednači sa (a 1), što daje zamenu _x->1. Obavljanjem te zamene pitanje (*3) se prevodi u ovo

?((a 1)(PP 1)FAIL)

i zbog FAIL moramo se vratiti na (a 1) i za njega tražiti nov dokaz-što je očigledno besmislica, znači kratko rečeno promenljiva, tj. nepoznata _x se nesme zamenjivati i tako izgubiti iz pitanja. Kraj Primera 5.2.2.

Kao što vidimo u postavljenom pitanju promenljiva, odnosno nepoznata _x se ne sme uklanjati zamenjujući je njenom trenutnom vrednošću. Inače nije teško uvideti, što je inače veoma bitno, da uopšte glavni razlog ne dozvoljavanju zamene nepoznatih jeste potreba korišćenja procedure vraćanje (backtracking).

Sada ćemo izložiti jedan način na koji se može rešiti to pitanje "nedozvoljenog zamenjivanja". Taj način koristi zamisao jednog konačnog niza tzv. d o d e l n i k a, u kraćoj oznaci Dodel, čiji članovi su uređene trojke oblika *(Var, Zam, Svoj)*, gde je *Var* nepoznata, *Zam* njena tekuća zamena (vrednost), *Svoj* je broj 0,1,2,... čija uloga je da bliže zapamti kakva je priroda(svojstvo) te nepoznate, što ćemo ubrzo bolje objasniti. Primera radi, evo kako se upotrebom dodelnika raspravlja pitanje (*4):

Najpre recimo da je na samom početku Dodel prazan niz, odnosno bez članova. Drugo, na samom početku algoritma on "prihvata" sve nepoznate učestvujuće u datom pitanju, a ovde je to samo _x. U vezi sa tim _x u dodelnik stavljamo ovu trojku[20] (_x,_,0) sa ovim smislom:

Treća komponenta je 0, i tako ističemo da nepoznata _x još nije dobila nikoju vrednost. Druga komponenta je znak[21] _ što ukazuje da nam vrednost još nije poznata.

Sasvim je prirodno što je _x preko navedene trojke stavljeno u dodelnik, jer konačno deo našeg zadatka upravo je traženje vrednosti tog _x. U ovom koraku dodelnik glasi

Dodel= (_x,_,0)

Radi računanja, tj. dokazivanja formule (a _x) nju ujednačujemo sa glavom prvog a-članka, tj. formulom (a 1), i tako dolazimo do zamene _x ->1. Dokaz se završava, jer prvi članak je elementarna aksioma. Budući da je u tom koraku nepoznata _x dobila vrednost to joj u dodelniku menjamo drugu i treću komponentu, odnosno umesto _ stavljamo vrednost 1, a treću tj. *Svoj* postavljamo na 1. Tako sada dodelnik izgleda

Dodel= (_x,1,1)

Do sada smo sreli sledeće dve opšte činjenice:

(5.2.4) Prvo, na samom početku prološkog algoritma sve nepoznate sadržane u pitanju stavljaju se u dodelnik preko svoje trojke, koja izgleda (Var,_,0), gde je Var dotična nepoznata. Slično se čini i nadalje tokom algoritma kad god nam se pojavi nova nepoznata, koju moramo zapamtiti, ali koja u tom koraku algoritma ne dobija neku vrednost. U protivnom, tj. kad nova nepoznata pri svojoj prvoj pojavi odmah i dobije neku vrednost onda u dodelnik stavljamo trojku (Var,Zam,1), gde Var nepoznata, a Zam njena vrednost.
Drugo, kad se u nekom koraku dogodi da izvesna nepoznata Var, koja nije nova, tj. ona je već bila uključena u dodelnik, dobije neku vrednost Zam onda iz dodelnika izbacimo njenu trojku (Var,...,...) i umesto nje stavimo novu trojku (Var,Zam,1).

U vezi sa tim pravilom podvucimo da odrednica *Svoj* iznosi 0 upravo u onom koraku algoritma u kome smo nepoznatu Var stavili[22] u dodelnik, ali tada Var nije još dobila nikakvu vrednost; dok *Svoj* iznosi 1, možemo tako reći, upravo na onom mestu gde je Var dobila vrednost.

Vraćamo se zadatku. Nakon dokazivanja (a _x) idemo na njenog "dešnjaka", tj. na (PP _x), što je naredna formula za dokazivanje. Šta je sada sa tim _x u formuli (PP _x)? Na osnovu dodelnika zaključujemo da _x ima vrednost 1, tj. x je već "vrednovano". Stoga, odmah u dodelniku broj *Svoj* povećamo za 1 i dodelnik sada glasi

Dodel= (_x,1,2)

Budući da *Svoj* na ovom mestu iznosi 2, tj. nije 1, to kad nam bude zatrebalo znaćemo da _x nije vrednovano upravo u formuli (PP _x). Odmah istaknimo

---

[18] Vid. Napomene 2.1.1 i 2.2.1.

[19] Recimo, to je slično sa : Štampaj ono x koje je rešenje jednačine 3+x=5. Jasno je da to x je nepoznata, dakle nikakva promenljiva.

[20] Znači, trenutno dodelnik je jednočlani niz.

[21] U stvari, u prolozima je umesto takve crtice na tom mestu upravo adresa promenljive _x. Zato, ako na primer Prologu postavite pitanje kao
?-write(_x) odnosno ?((PP _x))
na ekranu će se u stvari štampati adresa od _x.

[22] To je malo uprošćenije rečeno, jer u stvari, nismo nju stavili već odgovarajuću trojku.

opšte pravilo:

(5.2.5) Ako se neki korak algoritma sastoji od prelaza od formule F1 na F2 koji je njen "dešnjak", onda uradimo sledeće:
Svakoj onoj nepoznatoj formule F2, koja je nepoznata i formule F1 parametar *Svoj* povećamo za 1.

Kao naredno u zadatku nas čeka da "dokažemo" (PP 1). Ta formula je tačna, a kao bočni efekat na ekranu se pojavi 1. U narednom koraku dođemo do FAIl. Medjutim, FAIL je netačna formula, pa nastaje backtracking (vraćanje). Prvo dođemo do (PP _x). Šta sada raditi ? Budući da tu _x ima vrednost 1, ali njegov *Svoj* iznosi 2 zaključujemo da ta vrednost nije stečena na tom mestu, dakle stoga _x-u ne smemo promeniti vrednost, pa je isto kao da smo došli do (PP 1). Jasno je da za tu formulu nema nikakvog drugog dokaza[23], pa sledstveno od (PP _x) idemo jedan korak ulevo i dolazimo do (a _x). Kao, neophodno pri tom hodu ulevo broj *Svoj* smanjimo za 1. Trenutno dodelnik postaje

Dodel= (_x,1,1)

Opet ističemo opšte pravilo za hod zdesna-nalevo (pri backtracking-u):

(5.2.6) Ako se neki korak algoritma sastoji od prelaza od formule F1 na F2 koja je njen "levak", onda uradimo sledeće:
Svakoj onoj nepoznatoj formule F2 ,koja je nepoznata i formule F1 parametar *Svoj* smanjimo za 1, izuzev ako je taj *Svoj* jednak 0 kada ga ne menjamo.

Vratimo se zadatku. Pošto *Svoj*=1 to znači da je _x tu dobilo vrednost i to je sada bitno, imamo opšte pravilo[24]:

(5.2.7) Nepoznata sme da promeni vrednost upravo na onom mestu gde je staru vrednost bila stekla.

I tako, sada tražimo nov dokaz za (a _x), dopuštajući mogućnost da _x promeni vrednost. Imajući na umu činjenicu da je prva a-grana već iskorišćena upošljavamo drugu ,tj. drugi članak ((a 2)). Novi dokaz se završava prihvatanjem zamene _x -->2, a dodelnik sada postaje

Dodel= (_x,2,1)

Dokazavši (a _x) idemo udesno na (PP _x), i uradimo (5.2.5), tj. broj *Svoj* povećamo za 1. Inače, formula (PP _x) je tačna, a kao bočni efekat na ekranu se štampa 2, jer prema dodelniku _x ima vrednost 2. Ponovo dođemo na FAIL. On tera nazad, prvo na (PP _x). Iz dodelnika saznajemo da *Svoj* od tog _x iznosi 2, pa zaključujemo da u stvari treba na neki drugi način da dokažemo (PP 2). To je nemoguće, pa idemo ulevo na formulu (a _x), ali pri tom uradimo (5.2.6), tj. broj *Svoj* smanjimo za 1. Sada nas čeka neki drugi dokaz za (a _x), i kako *Svoj*=1, to _x sme uzeti neku novu vrednost. Nov dokaz za (a _x) upošljava treći a-članak i taj se dokaz "plaća" zamenom _x -->3.

---

[23] Zamislite za trenutak da u pitanju (*4) umesto (PP _x) stoji (p _x) i da u vezi sa tom formulom u programu imamo ova tri članka

((p _x)(PP _x)) ((p 1)(PP Jedan)) ((p 2)(PP Dva))

Tada bi se na tom mestu algoritma pojavio i nov dokaz za (p 1) koji bi se sastojao od štampanja reči Jedan.

[24] O tome smo ,u stvari, već govorili u Zadatku 3.9.

Dodelnik postaje

Dodel= (_x,3,1)

Dalje, ponovo dođemo do (PP _x), a *Svoj* postane 2. Na ekranu se štampa 3. Iza toga stignemo do FAIL, koji nas tera nazad i još jednom treba da nađemo nov dokaz za (a _x), pri čemu *Svoj* opet ima vrednost 1. Međutim, sve a-grane su potrošene, pa više nema takvog dokaza, odnosno algoritam se formalno završava znakom pitanja ?.

U vezi sa pitanjem nepoznatih u Prologu sada činimo korak dalje, upoznavajući slučaj kad se one, za razliku od dosadašnjih slučajeva, čak moraju zameniti stečenim vrednostima, pa stoga one i neće ući u dodelnik. Taj korak objašnjenja će biti u okviru izlaganja jedne značajne proloske procedure, tzv. spajanja. Odmah ističemo da u Prologu uopšte neka nepoznata postaje vrednovana, tj. dobija vrednost upravo u okviru te procedure. Za trenutak zamislimo da u odnosu na izvestan program P i neko postavljeno pitanje se tokom algoritma upravo stigne do dokazivanja jedne formule ϕ koja se nalazi na izvesnoj i-grani oblika

... ϕ Des1 Des2 ...

gde tačkice označavaju izvesne nenavedene delove, ako ih ima. One ispred ϕ predstavljaju "levake" od ϕ, dok Des1, Des2, ... su "dešnjaci" od ϕ. Radi dokaza formule[25] Φ upošljava se (sa prethodno "obnovljenim" promenljivim) neki članak[26]

(*5) (Ψ for1 for2 ...)

pri čemu ϕ je ,naravno, ujednačivo sa glavom Ψ tog članka. Tu sada nastaje jedna značajna procedura koju ćemo zvati s p a j a nj e ϕ sa tim članak[27].

(5.2.8) Procedura spajanja formule ϕ sa člankom[28] (ψ for1 for2 ...) uz pretpostavku da je ϕ ujednačivo sa ψ :

Upotrebimo jedan od algoritama unifikacije da bismo unificirali ψ sa[29] ϕ, ali prethodno formulu ϕ pripremimo ovako: one njene nepoznate koje su ranije već dobile izvesne vrednosti zamenimo tim vrednostima uzetim iz dodelnika, i uz to njihove parametre *Svoj* povećamo za po 1.

---

[25] Ako je ϕ sistemska formula, tj. formula čije ime je ime jednog od u Prolog ugrađenih predikata, onda u osnovi se slično dešava kao u opisu koji sledi. Podrobnije o nekim pojedinostima oko dokaza sistemskih formula govorimo u Napomeni 5.2.1.

[26] U stvari, prvi takav nam raspoloživ, ali takvi detalji sada nisu važni.

[27] Do sada smo u takvom slučaju često govorili "upošljavanje tog članka"

[28] Pretpostavljamo da su promenljive članka "osvežene"--u skladu sa pravilom (5.2.1).

[29] Pazite, redosled je bitan. Dakle, unificira se ψ sa ϕ. To praktično znači da ako se tokom algoritma pojavljuje slučaj unificiranja neke dve promenljive Var1, Var2 onda se prihvata zamena Var1-->Var2 pri čemu je Var1 iz ψ. Primera radi, ako je ϕ formula (a _x), a ψ formula (a _y), onda plod unifikacije je zamena _y-->_x, ali ne i obratna zamena.

Algoritmom unifikacije se nekim od promenljivih formula ϕ, Ψ dodeljuju odgovarajuće zamene. Neka je Var ma koja od takvih promenljivih i neka je Zam njena zamena. Tada:

Prvo, sve nepoznate koje su u nekom od Zam-ova, i koje su takođe nove, tj. nisu već stavljene u dodelnik, stavimo u dodelnik, u smislu da dodelniku dodajemo trojke oblika (Ime,_,0), gde sa Ime je označena ma koja od tih nepoznatih.
Drugo, ako je Var promenljiva iz ψ tada to Var ne stavljamo u dodelnik već u repu korišćenog članka, tj. u formulama for1, for2 ... to Var zamenimo njegovim Zam.
Treće, ako je Var promenljiva iz ϕ tada iz dodelnika uklonimo trojku oblika (Var,...,...) i umesto nje stavimo (Var,Zam,1).

U vezi sa tom procedurom istaknimo da u Prologu jedno od najosetljivijih pitanja je proširenje dodelnika novim članovima, jer tada se uvek iziskuje nov memorijski prostor. Prema proceduri (5.2.8) dodelnik moramo proširiti svim nepoznatim (tj. odgovarajućim trojkama), koje pri unifikaciji ostanu slobodne (tj. ne dobiju neku zamenu) i koje su nove, odnosno nisu članovi dodelnika. U stvari, ima još samo jedan slučaj kad takođe moramo proširiti dodelnik. To se odnosi na sam početak algoritma, što smo već istakli sa (5.2.4). Istićemo da ćemo iza naredne Napomene 5.2.1 u Primeru 5.2.3 videti proceduru 5.2.8 na izvesnim slučajevima.

Napomena 5.2.1. Ako je ϕ sistemska formula, tj. formula sa imenom jednakim imenu nekog u Prolog ugrađenog predikata, onda i u takvom slučaju, kao što smo već rekli, procedura (5.2.8) u osnovi slično teče, ali tada korisnik programa po pravilu ne može saznati razne pojedinosti, budući da su mu najčešće nedostupni odgovarajući ϕ-članci[30]. Pomenimo još dve stvari. Većina sistemskih predikata su "jednograni", tj. definisani sa jednim člankom, pa stoga recimo backtracking "ne deluje na njih". Primera radi, takvi su write, read, razni predikati sa fajlama (učitavanje, snimanje i dr.). Drugo, ako sistemska formula ϕ nije ujednačiva ni sa jednim od svojih članaka tada često umesto da se proglasi netačnom objavljuje se pojava greške kao: syntax error, CONTROL ERROR i sl. Kraj Napomene 5.2.1.
Sada prelazimo na već pominjani primer.

Primer 5.2.3. Odgovoriti na dato pitanje u odnosu na pripadni program:

(a) Program: p(X,X):-q(X). q(f(77)). Pitanje: ?-p(f(X),Y),write(X).
(b) Program: p(f(X)):-q(X). q(77). Pitanje: ?-p(X),write(X).

Rešenje. (a)
Korak 1. Dodel= (X,_,0),(Y,_,0) Nepoznate iz datog pitanja smo stavili u dodelnik.
Korak 2. Radi računanja p(f(X),Y) hoćemo da obavimo spajanje sa člankom p(X,X):-q(X). Najpre tom članku preznačimo promenljive. Recimo, uzmemo ovo njegovo "izdanje" p(Z,Z):-q(Z). Ujednačujemo p(Z,Z) sa sa p(f(X),Y). Dobijaju se zamene

Z->f(X)
Y->f(X)

Promenljiva Z je iz ψ formule (izražavanje kao u Proceduri 5.2.8)

[30] U stvari, jedino u Micro-prologu nije takav slučaj, jer u njemu za skoro sve sistemske predikate možemo saznati izgled odgovarajućih članaka. Recimo, na pitanje LIST IF na ekranu se pojave oba IF-članka.

Budući da je ona dobila vrednost svuda u članku p(Z,Z):-q(Z). zamenjujemo Z sa f(X). Drukčije rečeno, spoj formule p(f(X),Y) se obavlja sa ovim "primerkom", podslučajem

p(f(X),f(X)):- q(f(X)).

tog članka. Odatle je jasno da Z uopšte ne moramo pamtiti, tj. stavljati u dodelnik. U Zam-ovima učestvuje X. Da je to X nova nepoznata (što nije), sada bismo ga stavili u dodelnik.
U ovom koraku dodelnik glasi:

Dodel= (X,_,0),(Y,f(X),1)

X još nije dobilo vrednost, pa njegov Svoj je ostao 0, dok Y jeste dobilo vrednost, stoga njegov Svoj je postao 1.
Korak 3. Sada nas čeka dokaz formule q(f(X)). Obavljamo spoj sa člankom q(f(77)). Dokaz se završava prihvatanjem zamene X->77. Trenutno dodel glasi

Dodel= (X,77,1),(Y,f(X),1)

Korak 4. Prelazimo na "dokaz" write(X). Iako za ovu raspravu to nije bitno, navedimo da Svoj od X sada postaje 2. Inače, dokaz za write(X) se završava štampanjem na ekranu 77, jer to je vrednost od X. Tu je kraj algoritma. Formalno, odgovor na dato pitanje je da("yes").

(b)
Korak 1. Dodel= (X,_,0). Nepoznate iz pitanja stavimo u dodelnik.
Korak 2. Radi računanja formule p(X) hoćemo da obavimo spoj sa člankom p(f(X)):-q(X), pa zato uzimamo ovo njegovo "izdanje"

p(f(Y)):-q(Y)

Ujednačujući p(f(Y)) sa p(X) dolazimo do zamene

X->f(Y)

Prema Proceduri 5.2.8 to Y moramo staviti u dodelnik. U stvari, to je prirodno jer za sada smo X jedino izrazili pomoću Y, i preostaje da nađemo Y (slobodnije rečeno "hteli-ne hteli taj Y moramo zapamtiti). Dodelnik, trenutno glasi

Dodel= (X,f(Y),1), (Y,_,0)

Svoj od X je postao 1, jer X je dobilo vrednost.
Korak 3. Sada nas čeka dokaz formule q(Y). Dokaz se završava spajanjem sa elementarnom aksiomom q(77). Y dobije vrednost 77. Trenutno dodelnik glasi

Dodel= (X,f(Y),1),(Y,77,1)

Korak 4. Čeka nas "dokaz" za write(X), što se ovde svodi na štampanje f(77) na ekranu, jer Prolog je "pametan" da iz X->f(Y), Y->77 zaključi X->f(77). Inače u tom koraku dodelnik glasi

Dodel= (X,f(Y),2),(Y,77,1).

Algoritam se završava sa da("yes").

Na kraju dodajemo i sledeću napomenu.
Napomena 5.2.2. U vezi sa pitanjem promenljivih na kraju ovog dela dodajemo i sledeće o prološkoj osnovnoj relaciji (predikatu) R (odnosno read) Uočimo ovaj mali program

```
((a _x)(R _y) _y (c _x))
((b 0)(PP Evo B0))
((b 1)(PP B1))
((c 1)(PP C1))
```

i zadajmo ovo pitanje ?((a 1)). U prvom koraku algoritma se naide na relaciju R. Razmotrimo ove tri mogućnosti vrednosti za _y:

1) _y je 77 2) _y je (b 1) 3) _y je (b _x)

U a-članku "desnjak" od (R _y) je _y. Sledstveno, vrednost tog _y mora biti formula. Stoga slučaj 1) je nemoguć; pojaviće se poruka o grešci (CONTROL ERROR). U slučaju 2) na ekranu će se prvo javiti B1, a potom C1, što iziskuje (c _x), jer _x=1, dakle ništa neobično. Ali slučaj 3) ima svojih posebnosti. Naime, budući da smo _y zamenili sa (b _x) očekujuće je da se rep a-članka ovako zamisli (sklonili smo (R _y))

(b _x)(c _x)

a dalje, pošto _x=1, trebalo bi očekivati da se na ekranu pojave reči B1 i C1. Medutim, proverite, Prolog "ne misli" tako. On će na ekranu štampati reči Evo BO i C1. Zašto? Proloski algoritam je tako napravljen da ako se pri upisu ukjuče ma koje promenljive, onda ih on -za razliku od korisnika - gleda kao potpuno nove. Znači, kad smo _y dali vrednost (b _x) to je isto kao da smo mu dali ma koju od ovih: (b _x55), (b _Y) i dr.

5.3 I - ILI d r v e t a
P r o c e d u r e prikljućivanja i pregranjavanja

Uočimo najpre sledeći mali zadatak:
Primer 5.3.1. Izračunati vrednost izraza p, ako je dato

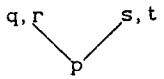
```
p=q*r + s*t
q=a + b*c
r=a + b
a=0
b=1
c=1
```

gde +, * dogovorno zamenjuju logičke veznike ili, i a 1,0 odgovaraju logičkim istinotosnim vrednostima TAČNO, NETAČNO.
Primetitimo da se tom zadatku - čini se potpuno prirodno - može pridružiti ovaj proloski zadatak: Ako je dat program

```
p:-q,r.    p:-s,t.    q:-a.     q:-b,c.
r:-a.      r:-b.      a:-fail.  b.    c.
```

šta je vrednost formule p ? Vrednost za p tražićemo načinom bliskim do sada upoznatom proloskom algoritmu i uz to koristićemo i-ili drveta.

Korak 1. Koristeći datu p-jednakost (možemo slobodnije reći i p-članke) napravimo ovaj crtež[1], reći ćemo i: i-ili drvo

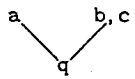


Drvo 1

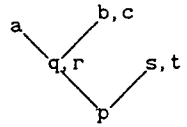
gde zapete odgovaraju vezniku i, a raklje vezniku ili.

Korak 2. Da bismo izračunali p najpre ćemo upotrebiti prvu granu, i potražićemo njenu vrednost. U tu svrhu prvo ćemo izračunati q, a potom r. I tako sada nas čeka računanje q. Radi toga najpre q-jednakosti pridružimo ovo i-ili drvo

[1] Rekli smo "crtež" jer još nemamo strogu definiciju i-ili drveta



Sada to drvo "priključimo"[2] na Drvo 1 na mestu q. Dobijamo ovo i-ili drvo



Drvo 2

Korak 3. Opet radimo po zamisli "prva grana i na njoj prvi član". Znači treba da izračunamo a. Medutim, budući da u vezi sa a je data samo jednakost a=0, zaključujemo da a ima vrednost 0 (tj. a je netačno).

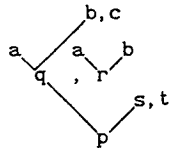
Korak 4. Izračunavši a silazimo na q. Naravno ne sme se zaključiti da je i q izračunato sa vrednošću 0. U stvari, q-jednakost sada postaje

q=a+b*c=0+b*c, tj. q=b*c

Znači sada nas čeka traženje redom vrednosti za b,c.

Korak 5. U skladu sa jednakostima b=1, c=1 i-grana b*c je izračunata sa vrednošću 1. Od te grane[3] silazimo dole i dolazimo do q. Ono je znači izračunato sa vrednošću 1. Kako je q član jedne i-grane i uz to q ima "desnjaka" to sada tražimo vrednost tog desnjaka, tj. vrednost od r. Za r imamo ovo i-ili drvo

Sada njega "priključimo" na Drvo 2. Nastaje ovo novo drvo



Drvo 3

Korak 6. Da bismo izračunali r opet po ohom "prva grana i prvi član" idemo na a. Budući da za a imamo samo jednakost a=0 to zaključujemo da je 0 tražena vrednost od a. Silazimo dole na r i mislimo ovako: propao nam je pokušaj da prvom granom izračunamo r (jer iz a=0 ne sledi r=0). I sada nastaje bitan trenutak:

Naime, Prolog "misli" ovako:
Trenutno r nije dokazano, pa stoga mora da nastupi procedura vraćanje (backtracking). I pošto r ima levaka q, sada bi trebalo za njega naći neki drugi dokaz i nakon toga iznova probati

[2] U stvari je po sredi primena procedure spajanja 5.2.7, ali iz razloga boljeg zamišljanja slike drveta podesnije reći "priključiti". Slično se izražavamo i u nastavku ovog dela.

[3] Mogli smo za q imati i još neku granu, što bi se recimo desilo kada bi q-jednakost glasila q=a+b*c+u*v*w. I tada bismo zaključili q=1, jer bi bilo: q=0+1+u*v*w=1+u*v*w=1.

dokazivanje r-a.

Medutim, ovaj primer koji razmatramo nema nikojih promenljivih[4], pa i kad bismo se vratili nazad i nekako ponovo dokazali q, opet bismo dolazeći na r imali iste okolnosti kao i ranije. Znači nekorisno je "predokazivnje" levaka, pa stoga sada vršimo pregranjavanje, odnosno upošljavamo drugu r-granu. To traži da računamo b. Zbog jednakosti b=1 zaključujemo da b ima vrednost 1.

Korak 7. Pošto b=1 to silazeći na r zaključujemo r=1. Vidimo da je r na kraju jedne i-grane. Elem, cela ta grana je izračunata, odnosno ima vrednost 1. Idemo na početak te grane i spuštamo se dole. Dolazimo do p. Znači i p je izračunato, ima vrednost 1. U stvari, tu je kraj ovog "prološkog" računanja i završni zaključak je p=1.

Kao što ste već primetili jezik i-ili drveta je bio veoma prikladan u rešavanju datog zadatka, i uz to se svakako rada pomisao da bi takav jezik mogao biti prikladan pri izlaganju prološkog algoritma uopšte. Odgovor je potvrdan, što će se videti iz daljeg izlaganja.

Prološki gledano, Primer 5.4.1 sadrži samo nularne relacije p, q, ... pa smo shodno tome svaku relaciju definisali sa po jednom jednakošću, čija desna strana je i-ili logički izraz. Tako, za p smo imali jednakost

p=q*r + s*t

Medutim, bukvalno takav pristup kod ne-nularnih relacija ne možemo koristiti. Primera radi zamislite da u okviru nekog prološkog programa imamo ovakve članke koji definišu relaciju imena *rel*

((*rel* _x _x) for2 ...*fork*)
((*rel* _x _y) For2 .. For*l*)

gde su for2, ..., *fork*, For2, ..., For*l* izvesne formule[5]. Već u tom primeru je jasno da relaciju *rel* ne možemo direktno definisati nekom jednostavnom i-ili jednakošću[6].
Da bismo opisali opšti slučaj, pretpostavimo da imamo ma koji program kojim se definišu izvesne relacije. Neka je ϕ ime jedne od tih relacija i neka njoj redom odgovaraju članci Čl1, Čl2, ..., Čls . Tada postupamo ovako: Od članaka pravimo LISTU[7], koju ćemo ovako označiti

(5.3.1) Čl1 *ili* Čl2 *ili* ... *ili* Čls

gde je znak *ili* očigledno u vezi sa logičkim veznikom ili, odnosno on ukazuje na postojanje raznih članaka (grana) relacije ϕ. Naravno sada je jasno da je ma koji proloski program shvatljiv kao konačan spisak članova

[4] Tj. nikojih nepoznatih.

[5] koje sadrže izvesne relacije, za svaku od kojih , recimo, u programu postoje odgovarajući članci.

[6] U stvari, pomalo "nategnuto" može ovako: Te članke zamenimo sa ova dva
((rel _t)(EQ _t (_x _x)) for2 ...fork)
((rel _t)(EQ _t (_x _y)) For2 ... For*l*)
Označimo ih trenutno redom sa ((rel _t) Rep1) ((rel _t) Rep2)
Tada (rel _t) je , slobodnije pisano: ILI(Rep1, Rep2)

[7] To može , recimo, biti dinamička lista (sa upotrebom pointera, tj. pokazivača) ili statička lista, kao konačan niz njenih članova.

koji su dvojke oblika

(Ime, Lista)

gde je Ime neka reč -služi kao ime relacije, a Lista je neka lista poput (5.3.1) --njome su redom zadani članci koji odgovaraju imenu Ime[8].

Primetite da su u (5.3.1) članci zadani "izvorno", tj. kako glase. A u primeni će se naravno redom priključivati pojedini od njih. Istaknimo da same članke uzimamo u duhu Micro-prologa, tj. shvatamo kao liste[9], odnosno još opštije kao I-termove, podrazumevajući sve ono što je o člancima rečeno u delu 4.1.
Do sada smo u raznim primerima proloških programa upotrebljavali nazive "levak", "dešnjak". Ti se pojmovi uvode sasvim prirodno:

Kod I-terma oblika (A|B) za B kažemo da je dešnjak od A, a za A da je levak od B[10].

S tim u vezi dodajmo sledeće. Znači, ako je S neki I-term oblika (A|B), tada imamo ovakve jednakosti:

car(S)=A, cdr(S)=B , dešnjak(A)=B, levak(B)=A

Kao što smo već videli, jedan od učestalijih koraka u prološkom algoritmu je , kao što smo u Primeru 5.3.1 govorili, procedura "priključivanje", koja je u osnovi primena procedure spajanja 5.2.8. Upravo na Primeru 5.3.1 smo videli da "priključivanje" iziskuje, odnosno "uvodi u igru" i-ili drveta. Tako, kao pri objašnjavanju procedure 5.2.8 zamislimo da smo tokom nekog prološkog algoritma stigli do dokazivanja formule ϕ koja se nalazi na izvesnoj i-grani[11] ...ϕ, Des1, Des2, ... i da smo u tu svrhu, kao prvi moguć, upotrebili članak

(5.3.2) (ψ for1 for2 ...)

odnosno da smo obavili proceduru spajanje ϕ sa tim člankom. Označimo sa Rep rep tako nastalog članka. U skladu sa izlaganjem u 4.1 jasno je da se račun za ϕ prevodi na računanje tog Rep (naravno "parče po parče", tj. računanje redom njegovih CAR-ova). U duhu do sada pravljenih raznih crteža, slika sada bismo mogli napraviti ovu sliku

(δ) ...,ϕ, Des1, Des2, ... ← Rep

U stvari, slika bi bila još "plastičnija" da smo umesto Rep stavili for1',

[8] Možemo reći i : odgovaraju relaciji tog imema.

[9] Već smo u delu 4.2 videli da se i u Edinburgškoj sintaksi liste mogu uzeti kao osnovni "gradbeni" sastavak.

[10] Znači, ne radimo sa "običnim" binarnim drvetima već eto sada uspostavljamo dodatne među-veze njihovih sastavaka.

[11] To je rečeno na uobičajen slikovit ali nedovoljno precizan način. U duhu maločas rečenog tu i-granu shvatamo kao ovakav I-term
(...|(ϕ|(Des1|(Des2|...))))
i računajući redom njegove CAR-ove (videti mali primer (Δ) iz 4.1), stigli smo znač do "računanja" sastavka ϕ.

for2',..,gde znak ' hoće da ukaže da je rečeni članak "pretrpeo"[12] spajanje sa formulom ϕ. Iako je navedena slika dosta podesna za praćenje prološkog toka, ona ima i ovaj bitan nedostatak:

Zamislimo da smo u daljem računanju za Rep dobili vrednost ne. Onda naravno pogrešno bi bilo da odmah zaključimo da ϕ ima takođe vrednost ne, jer možda za ϕ postoje i neki drugi članci,"grane": Čl1,Čl2,...,Člr koji još nisu korišćeni.

Elem, jasno je da sliku (δ) treba poboljšati tako da pamti i tu informaciju. Radi lakšeg izražavanja uvodimo naziv gornjak. Slika (δ) nas prirodno vodi na pomisao da je Rep gornjak formule ϕ. Međutim, jasno je da on sam nije sposoban da zapamti njemu naredne grane (za moguće pregranjavanje). U cilju poboljšanja možemo formuli ϕ pridružiti ovakav informativniji "gornjak":

To je ili lista: Rep ili Čl1 ili ... ili Člr

Drugim rečima,tako bismo bukvalno uveli i-ili drveta.Ali je takođe očigledno da takvo, odnosno bukvalno ostvarenje tih drveta nije dovoljno "ekonomično". Da li se ta zamisao može malo promeniti i postati praktičnija? Jedna lepa takva mogućnost je sledeća:

Označimo sa duz broj svih članaka imena jednakog imenu formule ϕ, i označimo sa red redni broj upošljenog članka (5.3.2). Tada brojevi red, duz i pomenuti Rep zajedno "pamte" celu navedenu ili-listu.

S tim u vezi,od tih odrednica sklapamo uređenu trojku (red,duz,Rep) i nju nazivamo gornjak za ϕ. Napišimo to i u obliku jednakosti

gornjak(ϕ) = (red,duz,Rep)

Za trenutak priču potpuno uopštimo. Naime,ako je (A|B) ma koji |-term onda kao što smo videli s njim u vezi su pojmovi car, cdr, levak, dešnjak. Sada dodajemo još pojam gornjaka. Formalno rečeno takvom termu i njegovim sastavcima A,B se dodeljuje po jedan gornjak, svaki od njih je izvesna uređena trojka oblika (prvi,drugi,drvo) gde prvi, drugi su dva prirodna broja, uz uslov prvi <= drugi, i gde drvo je neki |-term.
Ističemo da smo priču o "gornjaku" namerno tako uopštili, da bismo za njih imali potpuno formalnu (strogu) definiciju. Međutim, tokom prološkog programa će nas zanimati samo neki tokom algoritma uvedeni i menjani gornjaci.

Jasno je da se potreba za gornjakom po prvi put pojavi pri proceduri spajanje.U skladu sa tim kao naredno izlažemo proceduru priključivanje. Grubo rečeno njen osnovni zadatak je da za neku datu formulu ϕ odredi njenog gornjaka, ako ikoji postoji, odnosno u suprotnom slučaju da nas preko svoje promenljive indik[13] obavesti o njegovom nepostojanju. Kad je gornjak određen onda se u njemu odvaja i jedan deo tzv. sledbenik (ϕ).U stvari,prološki algoritam -- kao što ćemo videti u delu 5.5 -- se dalje nastavlja "računanjem",odnosno dokazivanjem tog sledbenika.

(5.3.3) Procedura priključivanje formule ϕ

Ako je ϕ jednako FAIL parametru indik damo vrednost 0 i procedura se završava.
Inače, uočimo redom sve članke sa imenom jednakim imenu formule ϕ.

[12] Što pored ostalog znači da su neke nepoznate dobile vrednosti.

[13] Kraće od indikator ("ukaznik").

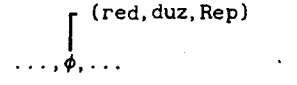
Neka je njihov broj duz. Ako duz=0 tada stavimo indik = -1 i procedura se završava[14]. Ako duz > 0 onda:
(*)

(i) Ako ϕ nije ujednačivo sa glavom nijednog članka onda parametru indik dajemo vrednost 0 i procedura se završava.

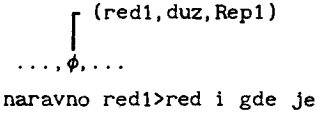
(ii) U suprotnom, uočimo prvi takav članak, koji je recimo red po redu, dalje obavimo proceduru spajanje formule ϕ sa tim člankom i označimo sa Rep rep novonastalog članka. Tada, trojku (red,duz,Rep) uzimamo kao gornjak za ϕ, a indik postavimo na 1 ukoliko[15] Rep=() odnosno u protivnom slučaju indik postavimo na 2.
U slučaju kad Rep<>() još za formulu ϕ određujemo sledbenika To je car(Rep).

Istaknimo još da ćemo u slučaju kad je, recimo, Gor gornjak formule ϕ, za to ϕ reći da je dolnjak za Gor. Međutim, u daljem izražavanju ćemo ponekad (jer to nam je kraće) takođe i za Rep, dakle deo pravog gornjaka govoriti da je gornjak.

Pored procedure priključivanje u prološkom algoritmu će se pojaviti njoj bliska procedura pregranjavanje. Naime zamislimo da u vezi sa formulom ϕ imamo ovakvu sliku (isečak drveta)



što znači da je na ϕ makar jednom bila upotrebljena procedura priključivanja.Tada procedurom pregranjavanje, tačnije rečeno ϕ-pregranjavanje najčešće se toj formuli ϕ pridružuje neka druga grana, prva moguća, i nakon toga gornja slika prelazi u ovakvu



gde je naravno red1>red i gde je Rep1 novi rep.Podrobnije rečeno, procedura pregranjavanje se opisuje bukvalno isto kao i procedura priključivanje s tim da umesto prvih šest redova u opisu (5.4.3), odnosno redova :

Ako je ϕ jednako FAIL ....
....
.....
Ako duz>0 onda:

dolazi ovaj tekst:

Uočimo sve članke (imena istog kao i formula ϕ), čiji je red red+1,red+2,...,duz

a iza tog uvodnog teksta se pretpostavlja nastavak teksta iz (5.4.3), odnosno deo od (*) nadalje..

[14] Kao što ćemo dalje u 5.5 videti tada se čitav proloski algoritam završava sa porukom o nedefinisanosti(nepoznatosti) formule ϕ.

[15] To praktično znači da korišćeni članak je neka elementarna aksioma. Recimo očigledna je jednakost ((a _x))=((a _x) |()).

Neka je njihov broj duz. Ako duz=0 tada stavimo indik = -1 i procedura se završava[14]. Ako duz > 0 onda:

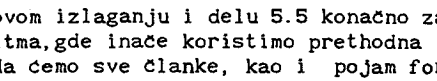
(*)

(i) Ako ϕ nije ujednačivo sa glavom nijednog članka onda parametru indik dajemo vrednost 0 i procedura se završava.

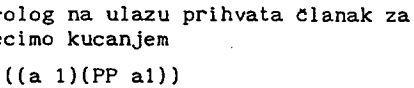
(ii) U suprotnom, uočimo prvi takav članak, koji je recimo red po redu, dalje obavimo proceduru spajanje formule ϕ sa tim člankom i označimo sa Rep rep novonastalog članka. Tada, trojku (red, duz, Rep) uzimamo kao gornjak za ϕ, a indik postavimo na 1 ukoliko[15] Rep=() odnosno u protivnom slučaju indik postavimo na 2. U slučaju kad Rep<>() još za formulu ϕ odredujemo sledbenika To je car(Rep).

Istaknimo još da ćemo u slučaju kad je, recimo, Gor gornjak formule ϕ, za to ϕ reći da je dolnjak za Gor. Medutim, u daljem izražavanju ćemo ponekad (jer to nam je kraće) takode i za Rep, dakle deo pravog gornjaka govoriti da je gornjak.

Pored procedure prikljućivanje u prološkom algoritmu će se pojaviti njoj bliska procedura pregranjavanje. Naime zamislimo da u vezi sa formulom ϕ imamo ovakvu sliku (isečak drveta)



što znači da je na ϕ makar jednom bila upotrebljena procedura prikljućivanja. Tada procedurom pregranjavanje, tačnije rečeno ϕ-pregranjavanje najčešće se toj formuli ϕ pridružuje neka druga grana, prva moguća, i nakon toga gornja slika prelazi u ovakvu



gde je naravno red1>red i gde je Rep1 novi rep. Podrobnije rečeno, procedura pregranjavanje se opisuje bukvalno isto kao i procedura prikljućivanje s tim da umesto prvih šest redova u opisu (5.4.3), odnosno redova :

Ako je ϕ jednako FAIL ....
....
.....
Ako duz>0 onda:

dolazi ovaj tekst:

Uočimo sve članke (imena istog kao i formula ϕ), čiji je red red+1, red+2, ..., duz

a iza tog uvodnog teksta se pretpostavlja nastavak teksta iz (5.4.3), odnosno deo od (*) nadalje.

[14] Kao što ćemo dalje u 5.5 videti tada se čitav proloski algoritam završava sa porukom o nedefinisanosti(nepoznatosti) formule ϕ.

[15] To praktično znači da korišćeni članak je neka elementarna aksioma. Recimo očigledna je jednakost ((a _x))=((a _x) |()).

Istaknimo, da u slučaju procedure pregranjavanje ulazni parametri su red, duz a izlazni su nove vrednosti za indik i red. Ukoliko indik je 1 ili 2 onda formuli ϕ se određuje i novi gornjak (kome je ona dolnjak).

Sada na kraju ovog izlaganja o prikljućivanju, odnosno pregranjavanju dodajemo nekoliko reči o slučaju sistemskih, tj. u Prolog ugrađenih relacija kao što su

PP, R, LOAD, SAVE, LESS, SUM, INT , itd.

One su uglavnom "jednograne", pa tokom algoritma ne moramo brinuti o njihovim gornjacima, odnosno oni su oblika (1,1,()). Međutim, neki od tih predikata, kao

ADDCL, DELCL

po pravilu iz osnova menjaju tekući program, što obično iziskuje dosta vremena, pa se u literaturi za njih sa razlogom kaže da su "vrlo skupi". Inače samo to menjanje je malo složenije i o njemu govorimo u delu 5.7.

5.4 P r o c e d u r e penjanje na vrh i plus_spust

Zajedno u ovom izlaganju i delu 5.5 konačno završavamo opis čitavog prološkog algoritma, gde inače koristimo prethodna izlaganja iz ovog poglavlja 5 Istićemo, da ćemo sve članke, kao i pojam formule uzimati u duhu LISP-sintakse. Od posebnog značaja je izlaganje u delu 4.1, čiji jedan deo ćemo na određen način ponoviti i dopuniti.
Tako, zamislimo da smo ušli u Micro-prolog[16] i da hoćemo da "ukucamo" neke članke, a takode i da nam Prolog odgovori na neko pitanje (nešto "izračuna"[17], nešto "uradi"[18]) .Tada:

Micro-prolog na ulazu prihvata članak za člankom kako ih korisnik "ukucava". Recimo kucanjem

((a 1)(PP a1))
((b _x)(a _x)(PP _x))

Prolog će prihvatiti dva članka, odnosno tačnije rečeno napraviće odgovarajuća im drveta. Kako Prolog zna kad se pri kucanju završava neki članak ? Kratko rečeno, glavni kriterijum za to je poklapanje brojeva levih i desnih zagrada. Inače, tokom kucanja nekog članka Prolog odmah gradi njegovo drvo. Medutim ako tokom tog građenja Prolog ustanovi da je

[16] Iz Dos-a kucamo reč Prolog, pretpostavljajući upotrebu IBM PC-računara i naravno da imamo fajlu imena prolog.exe, koja "izražava" Micro-prolog.

[17] Rečeno je "izračuna", da bi se podvuklo da se čitavo "delanje" u Prologu može svatiti kao svojevrsno relacijsko računanje.

[18] Recimo, možemo mu naložiti
LOAD MILE
i ako fajla imena MILE.LOG postoji u radnoj direktoriji, prolog će je "učitati", odnosno dodati je na tekući proloski program, ako smo neki već sami "ukucali". Ako fajle imena MILE.LOG nema pojaviće se poruka o grešci.

Istaknimo, da u slučaju procedure pregranjavanje ulazni parametri su red, duz a izlazni su nove vrednosti za indik i red. Ukoliko indik je 1 ili 2 onda formuli ϕ se odreduje i novi gornjak (kome je ona dolnjak).

Sada na kraju ovog izlaganja o prikljucivanju, odnosno pregranjavanju dodajemo nekoliko reči o slučaju sistemskih, tj. u Prolog ugrađenih relacija kao što su

PP, R, LOAD, SAVE, LESS, SUM, INT , itd.

One su uglavnom "jednograne", pa tokom algoritma ne moramo brinuti o njihovim gornjacima, odnosno oni su oblika (1,1,()). Međutim, neki od tih predikata, kao

ADDCL, DELCL

po pravilu iz osnova menjaju tekući program, što obično iziskuje dosta vremena, pa se u literaturi za njih sa razlogom kaže da su "vrlo skupi". Inače samo to menjanje je malo složenije i o njemu govorimo u delu 5.7.

5.4 P r o c e d u r e penjanje na vrh i plus_spust

Zajedno u ovom izlaganju i delu 5.5 konačno završavamo opis čitavog prološkog algoritma, gde inače koristimo prethodna izlaganja iz ovog poglavlja 5 Ističemo, da ćemo sve članke, kao i pojam formule uzimati u duhu LISP-sintakse. Od posebnog značaja je izlaganje u delu 4.1, čiji jedan deo ćemo na određen način ponoviti i dopuniti.
Tako, zamislimo da smo ušli u Micro-prolog[16] i da hoćemo da "ukucamo" neke članke, a takođe i da nam Prolog odgovori na neko pitanje (nešto "izračuna"[17], nešto "uradi"[18]) . Tada:

Micro-prolog na ulazu prihvata članak za člankom kako ih korisnik "ukucava". Recimo kucanjem

```
((a 1)(PP a1))
((b _x)(a _x)(PP _x))
```

Prolog će prihvatiti dva članka, odnosno tačnije rečeno napraviće odgovarajuca im drveta. Kako Prolog zna kad se pri kucanju završava neki članak ? Kratko rečeno, glavni kriterijum za to je poklapanje brojeva levih i desnih zagrada. Inače, tokom kucanja nekog članka Prolog odmah gradi njegovo drvo. Međutim ako tokom tog građenja Prolog ustanovi da je

---

[16] Iz Dos-a kucamo reč Prolog, pretpostavljajući upotrebu IBM PC-računara i naravno da imamo fajlu imena prolog.exe, koja "izražava" Micro-prolog.

[17] Rečeno je "izračuna", da bi se podvuklo da se čitavo "delanje" u Prologu može svatiti kao svojevrsno relacijsko računanje.

[18] Recimo, možemo mu naložiti
LOAD MILE
i ako fajla imena MILE.LOG postoji u radnoj direktoriji, prolog će je "učitati", odnosno dodati je na tekući prološki program, ako smo neki već sami "ukucali". Ako fajle imena MILE.LOG nema pojaviće se poruka o grešci.

ukucani članak neispravan[19] pojaviće se poruka o grešci i takav "članak" neće biti prihvaćen. Zamislite da ukucavanjem željenog programa, nakon toga želimo da Prolog nešto uradi za nas. Kako mu to reći? Prolog se rukovodi načelom koje smo u 4.1 već naveli, odnosno

Ako je korisnik ukucao
rel A
gde je rel ime r e l a c i j e duzine j e d a n[20] onda Prolog to "shvata" kao da treba da izračuna formulu (rel A).

Recimo, zamišljajući da imamo članak ((a 1)(PP a1)), onda na "unos" a 1 na ekranu će se ispisati a1, a na "unos" ? ((a 1)) na ekranu će se opet pojaviti reč a1. Zašto? To je u skladu sa definicijom predikata ?. Naime, njegova definicija, kao što smo ranije već rekli, glasi ((? _X)|_X) pa imamo prethodno opisan slučaj "reagovanja" Prologa na unos oblika rel A, s tim da je rel ovde ? .

Neka je sada P ma koji prološki program (u LISP-zapisu) i neka je njemu postavljeno ovakvo pitanje

(5.4.1) rel A

pretpostavljajući naravno da program P ima makar jedan članak sa glavom imena rel i uz to -po prethodno rečenom- u tom članku rel je relacija dužine 1. Jedan poseban slučaj je pitanje oblika

(5.4.2) ? (for1 for2 ...fork)

koje je u stvari oblika ? L , gde je L ovaj |-term: (for1.....fork). Tokom prološkog algoritma ćemo postupno graditi drvo algoritma, koje je po onom rečenom u 5.3 i-ili drvo , ali sa gornjacima definisanim kao uređene trojke oblika

(red, duz, Rep)

U praktičnom crtanju drveta u ulozi trojke pisaćemo ovako

(red, duz), Rep

Na samom početku prološkog algoritma za drvo se uzima[21] PRAZNO DRVO, u označi (). Slobodnije rečeno, algoritam počinje, ali kao što ćemo videti se i završava na tom praznom drvetu ("opštem korenu"). Iza toga odmah se formula pitanja proglašava za gornjaka tog drveta uzimanjem po dogovoru za red, duz brojeve[22] 1,1. Znači, drvo algoritma onda izgleda

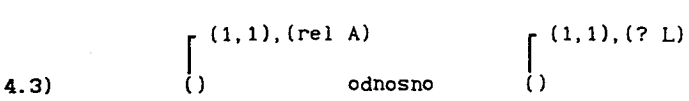
---

[19] Recimo, takav je ovaj
((a 1)))
jer smo u istom redu ukucali jednu desnu zagradu više.

[20] Znači među rel-člancima se nalazi makar jedan ovakav
((rel B)...)
gde B neki |-term.

[21] Ubrzo ćemo videti zašto je značajno na početku imati neko "opšte" , "konstantno" drvo.

[22] To smo uradili jer gornjak mora biti trojka. Međutim, kao što ćete dalje videti te uzete vrednosti 1,1 ne utiču na sam tok prološkog algoritma, jer -to je pravi razlog- procedura vraćanje(backtracking) me može da dosegne do tog mesta.



(5.4.3)

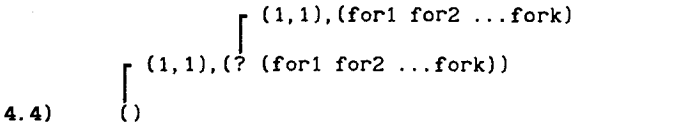
Tokom algoritma će se pojavljivati razni koraci . U svakom od njih će "na pozornicu" stupiti neka formula koja je stigla na red za "prološko računanje". Ta formula će biti na određenom mestu M na drvetu algoritma. Takvo mesto na kome se u dotičnom koraku --recimo tako-- usredsredila "žiža pažnje" zvaćemo žižnik. I sada , ukratko rečeno, tokom prološkog algoritma će se menjati tri glavne stvari:

drvo algoritma, koje će se "dogradivati","povećavati", "smanjivati", i sl.
žižnik, koji u svakom koraku iskazuje šta se upravo "računa", tj.na kom mestu drveta se nalazimo.
indik[23], broj koji će na svoj način "pamtiti" šta se dešava u algoritmu.Recimo, ako se pojavi neka sintaksna greška taj indik će dobiti vrednost -2, a ako se pojavi formula nepoznatog imena biće indik=-1.

U slučaju pitanja (5.4.1), odnosno (5.4.2) smo stigli do drveta (5.4.3) i sada se žižnik nalazi na mestu (rel A), odnosno (? L). U vezi sa drugim drvetom u (5.4.3) navedimo da će se, u skladu sa definicijom predikata ?

((? _X)!_X)

u idućem koraku pojaviti ovo drvo algoritama



(5.4.4)

Naravno sada upotreba brojeva 1,1 nije dogovorna, već

prvo, za predikat ? imamo tačno jedan članak (ugrađen u Prolog)
drugo, "u upotrebi" je taj prvi članak.

Međutim,pomenimo da ćemo na takvom drvetu najčešće deo (for1 ....fork) crtati ovako

for1,for2,...,fork

jer će nam tako biti lakše -kad ustreba- videti levaka, odnosno desnjaka neke od formula for1,...,fork[24].
I jasno je sada da u slučaju drveta (5.4.4) žižnik se nalazi na mestu for1 tj. na CAR(L). Znači, njega treba "računati" (dokazivati),a radi toga trebaće "računanje" neke druge formule ,itd. Naravno slično počinje "raspletanje" prološkog algoritma i u slučaju pitanja oblika (5.4.1).

[23] Skraćeno od indikator.

[24] U stvari,kao što smo u 4.1 videli !-term L ne mora bukvalno biti u obliku (for1 for2 ...fork) gde su for1,for2,...,fork formule. Naime,samo se traži da CAR-ovi od tog L (tj. CAR(L),CAR(CRD(L)),....) i to oni do kojih se tokom algoritma stigne postanu formule. Međutim, imajući na umu tu opštu misao čovek može iz razloga jednostavnosti praviti takve po malo uprošćenije slike.

Da bismo uvideli šta se sve dalje u algoritmu dešava, zamislićemo da smo tokom algoritma uopšte došli do neke formule ϕ koju treba dokazati, odnosno koja je došla na red za dokaz[25]. Znači,žižnik je trenutno na njenom mestu. Nastupa upravo prilika za prološku proceduru koju ćemo zvati penjanje na vrh (od mesta ϕ). Prvo ćemo je opisati slobodnije. Radi dokaza rečene formule ϕ upošljavamo prvi članak koji možemo, podrobnije rečeno,obavljamo proceduru priključivanje, kojom je , recimo, formuli ϕ određen izvestan gornjak. U tom gornjaku pronađemo sledbenika.Sada je žižnik na njemu. Opet nastaje kao maločas:priključivanje,gornjak,sledbenik.U stvari,može se desiti da se to "penjanje na vrh" završi, odnosno prestane. To će se desiti

ili ako indik=-1 ili indik=-2, kada će se ceo algoritam prekinuti sa porukom o grešci
ili ako indik=0, kada smo znači stigli do nekog netačnog "parčeta",formule i onda mora da nastane procedura vraćanje(backtracking)
ili ako indik=1, kada smo znači pomoću elementarne aksiome neko "parče" dokazali, pa iza toga procedurom plus spust tražimo novo mesto žižnika, tj. tražimo šta je naredno za "dokaz", za "račun".

Može se opisno reći da se procedurom penjanje na vrh drvo algoritma sve više usložnjava i da se često kao plod te procedure određuje novo mesto žižnika. I u toj proceduri, kao uostalom i u svim narednim prološkim procedurama, učestvuje indikator indik.

(5.4.5) Procedura penjanje na vrh - od mesta ϕ
Žižnik je na ϕ.
(*) Ako ϕ nije formula i nije joj trenutna vrednost formula, onda stavimo indik=-2 i procedura se završava.
U protivnom slučaju obavimo proceduru priključivanje primenjenu na žižnik. Ako je dobijeni indik jednak -1,0,1 indik ne menjamo i procedura se završava.
Ako indik=2 tada:
Ako Rep=(/) stavimo indik=1 i procedura se završava
Inače :
Za novi žižnik uzmemo sledbenika starog žižnika. Ako žižnik=/ tada žižnik=CAR(CDR(Rep)),indik=2;idemo na (*).

Kao što se vidi kad se završi procedura penjanja na vrh onda indik ima jednu od četiri vrednosti -2,-1,0,1. Tada:

Ako indik=-2 tada će se, kao što ćemo iz daljeg opisa prološkog algoritma videti, algoritam prekinuti sa porukom o sintaksnoj grešci.
Ako indik=-1 tada će se algoritam takođe prekinuti sa porukom o grešci sa smislom: formula ϕ nije definisana(nije poznata).
Ako indik=0 tada nastaje procedura vraćanje,koju opisujemo u delu 5.5.
Ako indik=1 tada nastaje procedura plus_spust,koju opisujemo u nastavku.

Istaknimo još da se žižnik tokom te procedure stalno menja i na kraju,izuzev ako indik=-2 ili indik=-1, žižnik se smesti na izvesno mesto. To je važno, jer ako iza penjanja na vrh treba da dođe neka druga procedura ona počinje od zatečenog drveta,zatečenog žižnika i zapamćenog indikatora.Pro-

[25] Recimo, u maločas pričanom slučaju pitanja (5.4.1), (5.4.2) ta mesta su (rel A) na prvom drvetu (5.4.3) ,odnosno for1 na drvetu (5.4.4).

cedurom plus_spust se takode odreduje novo mesto žižnika ili,što je takode moguće, indik se postavi na 3, što će biti znak da se čitav algoritam završava.

(5.4.6) Procedura plus _ spust od mesta m = žižnik
(*) Ako žiznik ima dešnjaka, recimo, D, tada
Ako D nije / najpre uradimo (5.2.5), tj. svakoj onoj nepoznatoj formule D koja je nepoznata i žižnika parametar *Svoj* povećamo za 1, zatim stavimo žižnik=D,indik=2 i procedura se završava.
Ako D je / stavimo žižnik=/ i idemo na (*)
Ako žižnik nema dešnjaka,ali ima levaka, tada:
idemo po njegovoj i-grani na njen početak Pocet. Iz Pocet silazimo na dolnjaka,recimo Dol.Tada:
Ako Dol je (),stavimo indik=3 i procedura se završava.
Ako Dol nije (),stavimo žižnik=Dol i idemo na (*)
Ako žižnik nema dešnjaka i nema levaka, tada:
silazimo na dolnjaka,recimo Dol.
Ako Dol,je (), stavimo indik=3 i procedura se završava.
Ako Dol nije (),stavimo žižnik=Dol i idemo na (*).

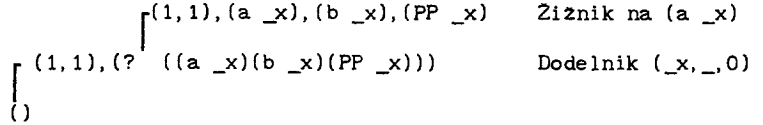
U vezi sa procedurom plus_spust dodajemo sledeće. Ako se procedura završi sa indik=2, onda to znači da smo odredili drugo mesto (vrednost) žižnika i dalji proloski algoritam se od tog mesta nastavlja procedurom penjanje na vrh. Medutim, ako indik=3 što znači da smo se spustili na prazno drvo () čitav algoritam se završava sa konačnim odgovorom da. Da bi se lakše uvideli razlozi takvom "ponašanju" proloskog algoritma navodimo sledeći mali primer,u kome će se na kraju desiti "spust do kraja", odnosno na ().

Primer 5.4.1. Dat je program P

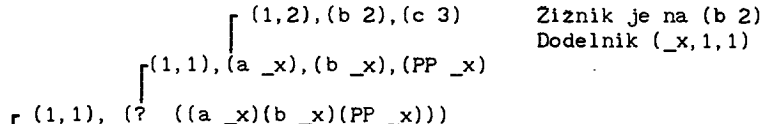
```
((a 1)(b 2)(c 3))    ((a 2))    ((b 1))
((c 1))    ((b 2))    ((b 3 4))    ((c 3))
```

Proloski "izračunati" formulu ?((a _x)(b _x)(PP _x)).

Rešenje.To je slučaj ?-formule,za koje smo u opštem slučaju bili stigli do drveta oblika **(5.4.4)**, odnosno ovde je to ovo drvo

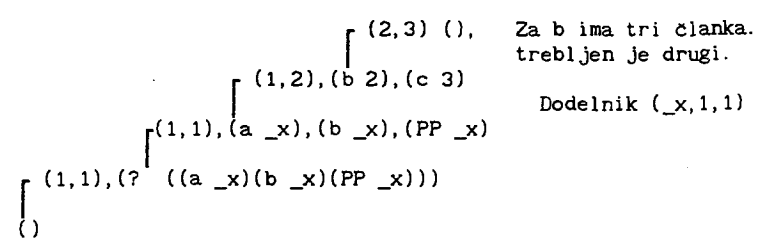


i žižnik je na naznačenom mestu. Čeka nas naravno računanje formule (a _x) sa nekim _x, pa nastaje procedura penjanje na vrh. U prvom koraku se pojavi ovo drvo

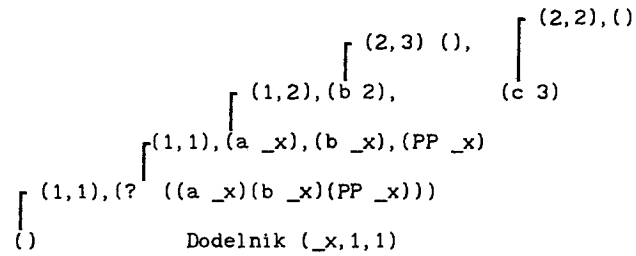


gde je žižnik na navedenom mestu. Primetimo da kod gornjaka formule (a _x)

red je 1, jer u proceduri priključivanje (a _x) je "spojeno" sa prvim a-člankom, što je takode dalo i navedeni dodelnik.Dalje, duz je 2, jer ukupno ima 2 a-članka. Znači, sada je u upotrebi prvi, ali nam je jasno zbog 1 < 2 da ima još preteklih a-članaka.Nastavljamo penjanje na vrh od mesta (b 2). U idućem koraku dobijemo ovo drvo

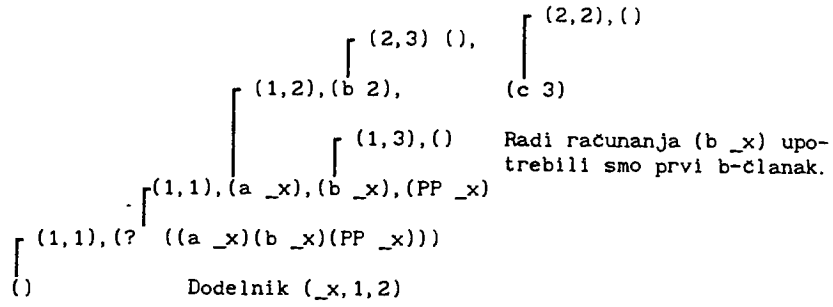


Sada je penjanje na vrh završeno, a imamo slučaj indik=1.U stvari,trenutno smo završili jedan dokaz, odnosno dokazana je formula (b 2) i u tom dokazu je iskorišćena elementarna aksioma (zato je indik i dobio vrednost 1).Žižnik je na (b 2).Odatle nastaje plus_spust.Ta procedura ovde traje samo jedan korak, jer (b 2) ima dešnjaka. To je (c 3). Žižnik je na njemu i sada sada odatle se penjemo na vrh (tj. tražimo dokaz za (c 3)).

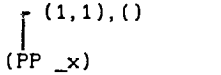


Penjanje na vrh je završeno u jednom koraku jer ((c 3)) je elementarna aksioma.Stoga žižnik je sada na (c 3) i čeka nas plus_spust. Sada imamo slučaj "nemanja dešnjaka i imanja levaka".Naime,od (c 3) krenemo po njegovoj i-grani i idemo na njen levi kraj. Tako dodemo do (b 2). Sada se spustimo na dolnjaka, tj. na (a _x) i sada odatle zamišljamo nov plus_spust. Ali sada nastaje slučaj u kome trenutni žižnik,tj. (a _x) ima dešnjaka koji je (b _x).Sledstveno najpre uradimo (5.2.5), što se ovde svodi na to da *Svoj* od _x od 1 postane[26] 2. Dalje idemo na (b _x), tu je kraj plus_spustu. Žižnik je trenutno na (b _x),i sada nas od tog mesta čeka penjanje na vrh, tj. dokazivanje (b _x). Pošto *Svoj* iznosi 2, to nas u stvari, čeka dokazivanje formule (b 1).Novo drvo izgleda

[26] To je dosta prirodan korak,jer pri dokazivanju (a _x) _x je steklo vrednost, i njegov *Svoj* je postao i ostao 1. Stoga, pošto iza dokaza (a _x) nas čeka dokaz za (b _x), onda je jasno da _x u toj formuli ima već odredenu vrednost. Elem, njegov *Svoj* postaje 2.



Opet "smo se popeli na vrh"- slučaj 2, pa nas od mesta (b _x) čeka procedura plus_spust. U jednom koraku dođemo do njegovog dešnjaka[27] (PP _x) i sada odatle treba da se "penjemo na vrh". Ali, to je elementarna aksioma (ugrađena u Prolog) i kao "bočni efekat" na ekranu se štampa vrednost od _x, tj. 1. Na drvetu bi sada trebalo deo (PP _x) dograditi ovako



Sada nas od mesta (PP _x) čeka plus_spust. Pošto je ta formula bez dešnjaka to po njenoj grani idemo na sam početak, tj. (a _x) i spustimo se na dolnjaka, tj. na formulu

(? ((a _x)(b _x)(PP _x)))

koja je oblika (? L) i nema dešnjaka. Stoga se od nje spustamo na dolnjaka Dolnjak je sada () i čitav algoritam se završava rezultatom da. Tu se vidi korisnost uvođenja praznog drveta () na samom početku algoritma, jer u protivnom slučaju ne bi bilo jasno šta znači da se plus_spustom "spustimo do kraja" i završimo algoritam.

5.5 P r o c e d u r a vraćanje (backtracking)

Može se slobodno reći da je to za Prolog najtipičnija procedura. O njoj smo -ali ne sasvim strogo- govorili u tački 2 (Pravilo 7). Budući da je potpuna priča o njoj veoma složena, najpre ćemo izložiti razne njene "osnovne sastavke", i nakon toga dati strog opis. Prvo, da vidimo makar ne potpuno strogo opisano kad sve ona "stupa u dejstvo". Postoje ukupno četiri takva slučaja:

P r v i s l u č a j :
Kao što smo u delu 5.3 videli, potreba za tom procedurom je nastala kad se pri proceduri penjanje na vrh došlo do mesta žižnik

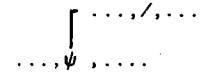
....,žižnik,...

i onda pri pokušaju priključivanja doživeo neuspeh, odnosno indik dobio vrednost 0. Kratko rečeno, na tom mestu smo ustanovili da je trenutna vrednost neke formule ne, pa odatle treba da nastane procedura vraćanje.

[27] Na tom mestu *Svoj* od _x postane 3.

Ostala tri slučaja se mogu javiti tokom obavljanja neke procedure vraćanje. Evo redom njihovih opisa.

D r u g i s l u č a j :
Ako smo vraćanjem stigli na znak reza / tada, prema Pravilu 8 iz tačke 2, treba da se spustimo na mesto pod(/) i od njega da započnemo novo vraćanje. Tako, na slici

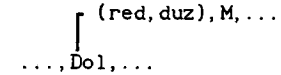


je to mesto ψ.

T r e ć i s l u č a j :
Zamislimo da smo na nekom mestu bili prinuđeni na vraćanje. Kao što će se videti, u toku vraćanja će se uvek postavljati pitanje novih dokaza formula na koje stignemo. Ali, može se dogoditi da tako dođemo do formule ψ čiji gornjak je ovakav

tj. drugim rečima formula ψ je "istrošena", jer sve su joj grane iskorišćene i uz to zadnji put, tj. u prethodnom dokazu za ψ je bila iskorišćena neka elementarna aksioma (o čemu svedoči deo ()). Tada se od mesta ψ vrši nova procedura vraćanje. To možemo i ovako prirodno pamtiti: gornjak nad ψ je "istrošen", pa možemo u tom koraku ψ proglasiti za netačnu. Elem, odatle valja krenuti sa novom procedurom vraćanje.

Č e t v r t i s l u č a j :
Opet zamislimo da smo tokom procedure vraćanje u jednom koraku bili prinuđeni da se od mesta M spustimo na dolnjaka Dol:



Tada ako je red=duz, što znači da je u prethodnim koracima algoritma upotrebljena i poslednja grana za Dol, onda nadalje od tog mesta Dol započinjemo novu proceduru vraćanje[1].

Ako dogovorno i slučaj reza shvatimo kao slučaj proglašavanja netačnom jedne formule, odnosno upravo formule pod(/) onda

Možemo kratko reći da procedura vraćanje se uključuje na mestu izvesne formule f čija netačnost : ili je trenutno dokazana ili proglašena.

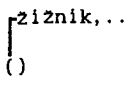
A sada evo podrobnijeg opisa procedure vraćanje od nekog mesta žižnik. Razlikujemo dva slučaja:

(A) žižnik nema levaka (B) žižnik ima levaka

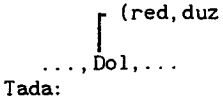
Razmatramo najpre slučaj A. Tada

[1] To je unekoliko slično sa trećim slučajem; odnosno i sada se čini prirodnim da se formula Dol smatra trenutno netačnom, jer čak, do tog koraka, upošljavanjem svih Dol-grana to nam nije pošlo za rukom.

Od žižnika se spustimo na njegovog dolnjaka Dol.
Ako Dol je[2] () onda se čitav proloski algoritam završava sa konačnim rezultatom ne. Znači imamo ovakvu sliku



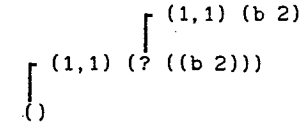
Ako Dol nije (), onda imamo ovakvu sliku



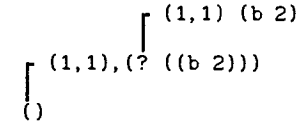
Tada:
Ako red=duz, što znači da su od Dol već upotrebljene sve grane stavimo žižnik = Dol, i od tog mesta započinjemo novu proceduru vraćanje.
Ako red<duz, onda se na mestu Dol završava vraćanje i sada sa žižnikom na Dol obavljamo pregranjavanje[3].

Da bismo objasnili slučaj A uočimo sledeći primer.

Primer 5.5.1. "Izračunati" formulu ?((b 2)) u odnosu na program ((b 1)).
Rešenje. Ubrzo se dođe do ovog drveta algoritma



i žižnik je trenutno na (b 2), odakle treba da se nastavi penjanje na vrh. Medutim, pri pokušaju priključivanja (b 2) se doživi neuspeh, odnosno indik dobije vrednost 0. Dakle, od (b 2) nastaje vraćanje, tj. žižnik je sada na toj formuli. Pošto žižnik nema levaka nastupa slučaj A, pa u skladu sa njim naredno je ovo drvo



a žižnik silazi na formulu (? ((b 2))). Sada odatle, budući da je u njenom gornjaku red=duz, nastaje nova procedura vraćanje. Opet nastupa slučaj (A), i uz to novi dolnjak je (). Opet je red=duz i sada se čitav algoritam se završava sa ne.

Sada razmatramo slučaj B. Sa Levi označimo levaka žižnika. Znači isečak drveta izgleda

...,Levi,žižnik,...

Sada je u duhu Prologa osnovna ova misao

[2] U stvari, na samom početku procedure vraćanje to se ne može desiti. Ali, namera nam je da se što više približimo njenom strogom opisu.

[3] Jer za Dol nisu sve grane upotrebljene.

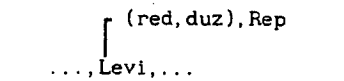
Budući da smo bili na mestu žižnik taj Levi, ako nije jednak rezu / , je već jednom dokazan. Sada bi, ako je moguće, trebalo naći neki nov, drugi dokaz za njega i nakon toga ponovo pokušati sa dokazivanjem žižnika.

Tu razlikujemo ova dva podslučaja

(B 1) Levi je znak reza /
(B 2) Levi nije /

O slučaju (B 1) smo u potpunosti govorili u tački 2 (Pravilo 8).

Razmatramo slučaj (B 2). Da bismo lakše videli mogućnosti koje postoje zamislimo da nad Levi imamo ovakvu sliku (koja "pamti" šta se "ranije dogodilo)

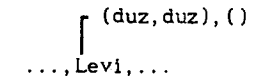


Tada tok daljeg algoritma zavisi od zamišljenog gornjaka. Razlikujemo ove podslučajeve

(B 2 1) Rep=(). Tada imamo dva podslučaja
(B 2 1 1) red=duz
(B 2 1 2) red<duz

(B 2 2) Rep<>().

U slučaju (B 2 1 1) nad levi imamo ovakvu sliku



što praktično znači u tom koraku nema druge mogućnosti za dokazivanje ("računanje") Levog. Stoga:

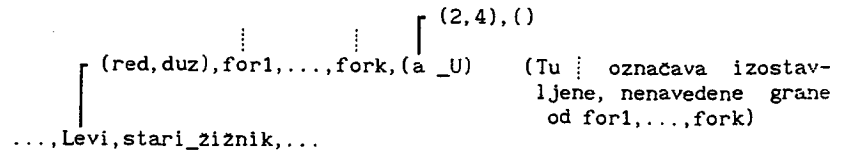
Stavimo žižnik na mesto Levi, i odatle počnemo novu proceduru vraćanje[4].

U slučaju (B 2 1 2) zbog red<duz pomišljamo da radi nalaženja novog dokaza za Levi treba uposliti neku novu Levi- granu čiji red je jedan od ovih brojeva

red+1,red+2,...,duz

Tačno rečeno nastaje procedura pregranjavanje, koju smo objasniti ranije (videti deo 5.3). A iza nje? Slobodnije rečeno, prvo penjanje na vrh, itd.

Sada razmatramo slučaj (B 2 2). Da bismo ga lakše shvatili zamislimo ovakvu sliku (deo drveta)



[4] Možemo slobodnije reći kao da smo u tom trenutku proglasili Levi za netačan pa sledstveno odatle započinjemo novo vraćanje.

tj. nad Levi "stoje" formule for1,....,fork. i na samom desnom kraju stoji (a _U), gde pretpostavljamo da je _U na tom mestu stekla vrednost. Dalje, zamislimo da tekući program ima ukupno ove a-članke

((a 11)) ((a 22)) ((a 33)) ((a 44))

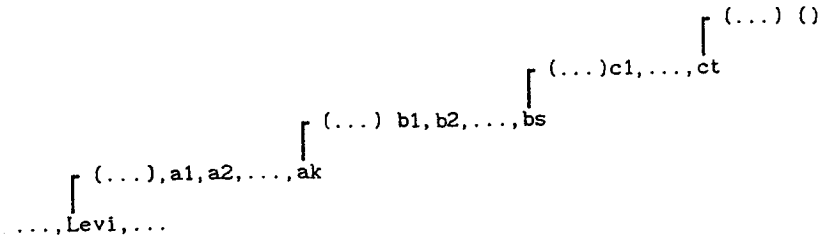
U skladu sa tom slikom, možemo videti kako je Levi dokazan poslednji put. Naime, nad Levi stoje

for1,..,fork -koje su znači bile dokazane (njihovi gornjaci nisu navedeni; označeni su sa ⋮)

i stoji (a _U), koja je znači bila poslednja dokazana.

Shodno tome, da bi se našao nov dokaz za Levi treba tražiti nov dokaz za (a _U). Pošto je poslednji put upotrebljen drugi članak, pa je _U dobilo vrednost 22, to sada upošljavamo treći članak[5] i _U postaje 33. Znači, opet je Levi dokazan pa nastaje plus_spust, kojim se na tom drvetu stigne do mesta stari_žižnik i sada za njega se traži dokaz "od početka", tj. od tog mesta nastaje penjanje na vrh, itd.

Značajno je da se izložene misli mogu uopštiti. Da bismo to lakše učinili uvešćemo i pojam "najdešnjaka" neke formule[6]. Naime, za formulu Levi odgovarajući najdešnjak je upravo (a _U). Zamislimo, trenutno ovakvu sliku



što znači da nad Levi se nalaze

izvesne formule[7] a1,a2,...,ak, tj. ne nalazi se (); nad poslednjom od njih, tj. ak, se nalaze formule b1,b2...,bs ,odnosno ne nalazi se (); nad poslednjom od njih, tj. bs, se nalaze formule c1,...,ct, odnosno ne nalazi se () i konačno nad ct se upravo nalazi prazno drvo ()

Taj ct, nad kojim je prazno drvo, je -po dogovoru- najdešnjak formule Levi.

Sada smo u prilici da uopšte prozborimo o slučaju (B 2 2). Tada se prvo nađe najdešnjak od Levi, i na to mesto postavi žižnik. Dalje, ako je taj žižnik upravo znak reza, onda postupamo po Pravilu 8, tačke 2, a ako nije jednak znaku reza onda nastupa procedura pregranjavanje, itd. Recimo, ako je pregranjavanje bezuspešno[8] onda od tog mesta nastaje procedura vraćanje.
To se slaže sa ovim razmišljanjem: ne uspeva nam pokušaj ikakvog drukčijeg dokazivanja ct, pa znači taj ct je postao netačan, elem odatle nastaje vraćanje, itd.

---

[5] Obavljeno je pregranjavanje.

[6] Reč nije idealna, ali bolje ikakva nego nikakva.

[7] Tačkice u (...) znači da odgovarajući red,duz nisu navedeni.

[8] Kratko, to znači da nismo mogli da nademo novu granu.

U dosadašnjem slobodnijem opisu procedure vraćanje korišćen je pojam najdešnjaka neke formule ϕ (bolje reći nekog mesta m) na izvesnom drvetu D. Evo kako se on strogo određuje:

(5.5.1) (i) Od mesta m idemo na njenog gornjaka i zatim na njegov desni kraj, recimo Des. Ako Des je znak reza ili ako gornjak za Des je (), onda stavimo najdešnjak =Des i postupak se završava, a inače stavimo m=Des i idemo na (i).

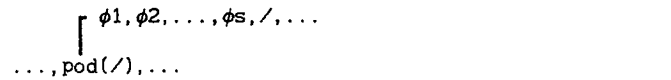
Podvucimo da dosadašnji opis procedure ima jedan nedostatak: ne pominje probleme u vezi sa prološkim promenljivim, odnosno nepoznatim, što je itekako važno. Taj nedostatak otklanjamo sada. S tim u vezi navodimo sledeće pravilo koga se treba držati:

(5.5.2) Ako se pri proceduri vraćanje dogodi da od nekog mesta M moramo ići na njegovog levaka Levi (vid. sliku),

...,Levi,M,...

onda svakoj onoj nepoznatoj formule M, koja je nepoznata i formule F1 parametar *Svoj* smanjimo za 1, izuzev ako je taj *Svoj* jednak 0 kada ga ne menjamo.
Potpuno slično pravilo važi i za slučaj kada sa nekog mesta M pri proceduri vraćanje moramo da se "spustimo" na dolnjaka.

Primetimo da se to pravilo mora čak više puta primeniti kad se pri proceduri vraćanje naide na znak / (vid. sliku i Pravilo 8 iz tačke 2)



Naime, tada (5.5.2) treba uraditi redom u formulama $\phi s,\ldots,\phi 2,\phi 1,pod(/)$.

Posle čitave prethodne pripreme dajemo strog opis procedure vraćanje. Znači, pretpostavljamo da imamo neko drvo D i na njemu uočeno mesto m, gde je žižnik i sada želimo da od tog m obavimo proceduru vraćanje. Evo njenog opisa:

(5.5.3) P r o c e d u r a vraćanje (backtracking)

(*) Ako žižnik nema levaka, onda se spustimo na njegovog dolnjaka Dol.
Ako Dol=(), stavimo indik=0, procedura se završava.
Ako Dol<>() i njegov gornjak je (red,duz,Rep) tada:
Ako red=duz stavimo žižnik=Dol i idemo na (*)
Ako red<duz stavimo žižnik=Dol,indik=1 i procedura se završava.
Ako žižnik ima levaka Levi, onda u žižniku uradimo (5.5.2). Dalje:
Ako Levi je znak reza / onda idemo na početak grane i u svakoj formuli ϕ na koju naidemo uradimo(5.5.2) spustimo se na dolnjaka Dol, u njemu takode uradimo (5.5.2), dalje stavimo žižnik=Dol, i idemo na (*)
Ako Levi nije / pretpostavimo da je njegov gornjak (red,duz,Rep)
Tada:
Ako Rep=() i red=duz stavimo žižnik=Levi,idemo na (*)
Ako Rep=() i red<duz onda stavimo
žižnik=Levi,indik=1 i procedura se završava.

Ako Rep<>() onda onda prvo pronademo[9] najdešnjak (Levi), u oznaci Najdes. Ako je Najdes jednak znaku / onda idemo na početak grane i u svakoj formuli ϕ na koju naidemo uradimo(5.5.2)spustimo se na dolnjaka Dol, u njemu takode uradimo (5.5.2), dalje stavimo žižnik=Dol, i idemo na (*).
Ako Najdes nije jednak / neka je njegov gornjak (red,duz,())
Tada:
Ako red=duz stavimo žižnik=Najdes i idemo na (*)
Ako red<duz onda stavimo žižnik=Najdes, indik=1 i procedura se završava.

Kao što se primećuje procedura vraćanje se završava

ili sa indik=0 ili sa indik=1

Ako indik=0, onda smo se "spustili" na sam "koren" (), tj.prološki algoritam je završen sa konačnim odgovorom ne.
Ako indik=1, onda je odredeno i novo mesto žižnika i dalje će se na tom mestu obaviti pregranjavanje,itd. U vezi sa pregranjavanjem dodajemo sledeće

(5.5.4) Ako se procedura vraćanje završi i onda pojavi potreba za procedurom pregranjavanje,tada čitavu staru granu smemo "ukloniti",u smislu da iz dodelnika izbacimo sve one nepoznate koje su u njega ušle preko te grane.

Pomenimo još jednu stvar koja može biti korisna u radu sa drvetom algoritma. Naime, da li se ponekad na drvetu ponešto može skloniti, "odseći" iz razloga što recimo znamo da je taj deo "zastareo, odigrao svoju ulogu", pa u tom obliku se i neće koristiti.Evo takvog pravila:

(5.5.5) Ako se tokom procedure vraćanje stigne na neko mesto M pa se po opisu te procedure kaže da od tog mesta treba početi novu proceduru vraćanje, onda toj formuli M sklonimo gornjak i stavimo joj ovaj (0,0,()).Pri tom sklanjanju starog gornjaka uradimo slično kao u (5.5.4),tj. iz dodelnika izbacimo sve one nepoznate koje su u njega ušle preko tog gornjaka.

Naime, za takvo M,buduci da se od njega moramo vraćati, možemo reći kao da je "proglašeno za trenutno netačno" pa njen gornjak, koji "pamti" šta se prethodno desilo, više nije zanimljiv[10], pa ga smemo promeniti.Elem,stavili smo jedan "provizoran" gornjak.

5.6 Još o p r e d i k a t i m a ADDCL, DELCL, odnosno assert,retract

Najpre ćemo na jednom primeru videti kako se "izračunavaju" ADDCL-formule

Primer 5.6.1. Dat je program

((a 1)) ((a 2)) ((a 3))

[9] Koristeci (5.5.1).

[10] Naime, ako se dogodi da ponovo dođemo na tu formulu onda će se na tom mestu obaviti procedura prikljucivanja.

Raspraviti pitanje ?((a _x)(PP _x)(ADDCL ((a 77)) 1) FAIL)

Rešenje.Kratko rečeno,prvo pri prvom računu (a _x) će se stici do _x=1, pa će se na ekranu štampati 1, dalje će se programu dodati članak ((a 77)) i to na prvo mesto,jer tako je navedeno.Znači,u tom trenutku program se preobraća u ovaj

((a 7)) ((a 1)) ((a 2)) ((a 3))

Dalje se dode do FAIL, koji tera na vraćanje. Ali, to je važno, pošto je prošli put potrošena grana ((a 1)),a nova grana je dodata pre nje, to novu novu granu ne možemo iskoristiti. Tako, na ekranu će se dalje štampati 2,3 i na kraju ? , što će označiti kraj algoritma.
Medutim, sasvim bi bilo drukčije dešavanje pri raspravljanju ovog pitanja

?((a _x)(PP _x)(ADDCL ((a 77)))FAIL)

gde bi se članak ((a 77)) , kad na njega dode red, stavio na kraj pa bi se u vezi sa tim pitanjem na ekranu pojavili:

1 2 3 7 ?

Evo sada kratkog opisa računanja formule oblika (ADDCL Clanak k) uopšte:

Kad se tokom algoritma, upravo penjanja na vrh, dode na takvu formulu onda:
Prvo, spisak (lista) članaka sa imenom Ime(Clanak) se proširi na k-tom mestu tim novim člankom.Ako k nije naveden, tj.imamo formulu oblika (ADDCL Clanak),taj Clanak se dodaje na kraj tog spiska.
Drugo[11], na celom drvetu algoritma se kod svakog gornjaka čiji doljnjak je imena Ime(Clanak) se poveca za 1 duz, a uz to se isto učini i sa odgovarajucim red-ovima, ukoliko su veci ili jednaki od k.

Sada prelazimo na slučaj DELCL-formula koje se javljaju u dva oblika

(DELCL ime k) -sa smislom : "Obrisati" k-ti članak medu člancima na ime.
(DELCL Clanak) -sa smislom: "Obrisati članak koji je baš jednak sa Clanak, ili opštije koji je ujednačiv sa Clanak.

Recimo, ako imamo program

((a 1)) ((a 2)) ((a 3)) ((a 4)) ((a 5))

onda pri računanju:

formule (DELCL a 1) će se brisati prvi a-članak
formule (DELCL ((a _X))) će se brisati prvi od trenutno postojecih članaka oblika ((a _X)), tj. od a-članaka
formule (DELCL ((a 5))) će biti obrisan baš članak ((a 5)).

Jasno je da je i DELCL-"operacija" veoma skupa.Jer,i tada slično gore opisanom moramo izvršiti odgovarajuće izmene u programu,i uz to,na drvetu algoritma na odreden način promeniti parametre red , duz , naravno tamo gde to treba[12].

[11] Što može biti vremenski "vrlo skupo".

[12] Kod gornjaka čiji doljnaci imaju ime jednako imenu naznačenom u DELCL-formuli.Ako je reč o formuli (DELCL ime k), to ime je upravo ime,a ako je

5.7 Završni opis prološkog algoritma

Imajući sve osnovne sastavke u rukama možemo napraviti celinu, čije puno ime je opšti proloski algoritam. Kao što smo već ranije rekli sve izražavamo na jeziku i-ili drveta. U vezi sa ovim -podsetimo se- pojavljuju se pojmovi

prazno drvo (), CAR, CDR, levak, dešnjak, dolnjak, gornjak, najdešnjak, red, duz, indikator, žižnik, dodelnik

Dalje, do sada smo opisali ove tri osnovne prološke procedure

penjanje na vrh, plus_spust, vraćanje,

pri čemu penjanje na vrh koristi proceduru priključivanje (a ova u osnovi koristi proceduru spajanje), a vraćanje koristi proceduru pregranjavanje[13]. I tako sada zamislimo da je zadan ma koji proloski program P iskazan preko odgovarajuće liste članova oblika

(ime,Lista)

gde ime je neka konstatska reč, a Lista je konačan spisak[14] izvesnih članaka sa imenom ime. Dalje, u vezi sa zadanim programom neka je postavljeno pitanje oblika

(5.7.1) ? Lista

gde je Lista ,strogo rečeno, neki dati !-term čiji CAR-ovi, tj.

CAR(Lista), CAR(CDR(Lista)), CAR(CDR(CDR(Lista)),...

su ili bukvalno formule[15] ili to usled unifikacije postanu kad na njih tokom algoritma dođe red. Najprostiji slučaj, ali i najčešći, je kada je ta Lista bukvalno lista nekih formula. Recimo, takav je slučaj sa ovakvim pitanjima

? ((a _x)(PP _x)FAIL)
? ((b _x _y)(c _y)(a _x)(SUM _x _y _z)(PP _z))

Umesto da kažemo dato je pitanje oblika možemo reći dosta opštije:

Treba proloski izračunati vrednost formule (? Lista)

Tokom prološkog algoritma, kao što smo ranije već rekli, postupno će se menjati drvo algoritma, u oznaci Drvo, i na tom drvetu će se u svakom koraku pažnja usredsređivati na određeno mesto, odnosno žižnik. Sledstveno, čitav algoritam je u osnovi priča o drvetu D i tekućem žižniku.
Na samom početku kao što smo već rekli u delu 5.4 polazimo od ovog drveta (vid. (5.4.3))

---

reč o formuli oblika (DELCL Clanak) ime je ime glave članka Clanak.

[13] Kao što smo videli ona je bliska proceduri priključivanje.

[14] Videti (5.3.1) i okolni tekst.

[15] Može neki od njih biti i poseban znak kao : rez / , i FAIL. Više o svemu tome smo imali u delu 4.1.

(5.7.2) Drvo — ⌜ (1,1),(? Lista)
⌞ ()

i žižnik je na (? Lista). Kako smo u delu 5.5 rekli, Micro-prolog pored pitanja oblika (5.7.1) može da razmatra i pitanja oblika

rel Telo

gde je rel bar u jednom svom članku unarna relacija, tj. odnosi se na jedan argument. U takvom slučaju radi se o prološkom računanju formule (rel Telo) i pojavi se drvo potpuno slično sa (5.7.2), odnosno umesto dela (? Lista) dolazi (rel Telo) i žižnik je onda na toj formuli. Međutim, bitno je da je jedinstven opis prološkog algoritma u oba slučaja.
Sada ćemo dati taj opis. On kao što rekosmo pretpostavlja zadani program P i neko pitanje (" za računanje"), tj. polazimo od drveta (5.7.2), smatrajući da je žižnik na mestu (? Lista). Opis glasi

(5.7.3) Opis prološkog algoritma

(A) Penjanje na vrh od mesta žižnik.

Ako se ta procedura završi[16] onda idemo na (Raskrsnica)

(Raskrsnica)

Ako indik=-2 proloski program se prekida sa porukom[17]
SYNTAX ERROR
Ako indik=-1 proloski program se prekida sa porukom
Predicate Not Defined
Ako indik=0 ,onda idemo na (B)
Ako indik=1 ,onda idemo na (C).

(B) Vraćanje od mesta žižnik.

Ako indik=0 program se završava sa ?, tj. ukupan rezultat je ne
Ako indik=1 onda neka ϕ bude trenutni žižnik. Tada
obavimo ϕ-pregranjavanje i idemo na (A).

(C) Plus_spust od mesta žižnik.

Ako indik=3 program se završava sa porukom[18] yes (da).
Ako indik=1 od mesta novog žižnika nastaje (A).

---

[16] Naime, može da se dogoditi
1) Upadanje u beskonačnu petlju, tj. pri traženju vrednosti neke formule ϕ tokom algoritma (odnosno tokom penjanja na vrh) se opet pojavi potreba traženja vrednosti iste formule; i l i
2) Stalno se uvode novi objekti, nove nepoznate.

[17] Bukvalno je ta poruka u Micro-prologu.

[18] U stvari, Micro-prolog "ćuti", tj. ne daje nikakvu poruku, odnosno na ekranu se jedino pojavi znak &, tj. prompt (znak) Micro-prologa.

6. Z A D A C I, II

U ovoj tački se bitno koristi izlaganje iz prethodne tačke, odnosno izlaganje opšteg prološkog algoritma. Pored ostalog veoma je važno imati na umu razne činjenice u vezi sa promenljivim, odnosno nepoznatim. Kao što znamo dodelnik je "zadužen" da zapamti razne podatke u vezi sa, u toku algoritma pojavljenim, nepoznatim (koje inače moramo da zapamtimo). Ali, s druge strane sam dodelnik je po malo složen, a i dosta je zamorno da se u skoro svakom koraku algoritma na njemu obave odgovarajuće promene. Čini se da je za korisnika podesnije da se, što ćemo odmah objasniti, dodelnik malo uprosti i da neke od činjenica u vezi sa nepoznatim, odnosno upravo one kako im se menja parametar *Svoj*, pamte uz pomoć drveta koje se gradi i razvija tokom algoritma. Određenije rečeno:

(6.1) Umesto dodelnika, kao skupa uređenih trojki, radićemo sa, reći ćemo, skraćenim dodelnikom kao skupom uređenih dvojaka oblika (*Var*,*Zam*) i uz to neku nepoznatu stavljaćemo u dodelnik samo na onom mestu, dakle ne pre, na kom dobija vrednost. Budući da smo parametar *Svoj* izbacili to će biti bitno da se na drvetu takvo mesto M zapampti, recimo podvlačenjem dotične nepoznate *Var*. Na takvom mestu nepoznata *Var* sme da promeni svoju vrednost, ali i samo na njemu, odnosno slobodnije rečeno, na drvetu "iza mesta M" nepoznata *Var* nikako ne sme da menja vrednost.

Zadatak 6.1. Ovaj zadatak se odnosi na primer gramatike **4.4.1**, odnosno na njen proloski prevod (to je **(4.4.4)**), koji glasi

```
sentence(X,Y):-noun_phrase(X,Z),verb_phrase(Z,Y).
    noun_phrase(X,Y):-determiner(X,Z),common_noun(Z,Y).
    noun_phrase(X,Y):-proper_name(X,Y).
  verb_phrase(X,Y):-verb(X,Y).
  verb_phrase(X,Y):-verb(X,Z),noun_phrase(Z,Y).
    determiner([the|X],X).
    determiner([a|X],X).
  verb([likes|X],X).
  verb([looks|X],X).
    common_noun([man|X],X).
    common_noun([woman|X],X).
  proper_name(['John'|X],X).
  proper_name(['Vera'|X],X).
```

Korak za korakom naći rečenice, "plodove" te gramatike.

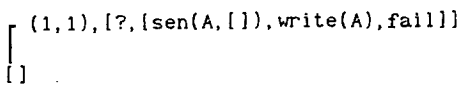
Rešenje. Jasno je da treba postaviti ovo pitanje

```
?-sentence(A,[]),write(A),fail.
```

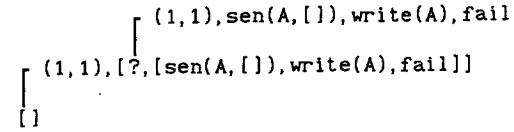
Iz tehničkih razloga koristićemo ove skraćenice:

```
sentence--> sen, noun_phrase-->nphrase, verb_phrase-->vphrase
common_noun-->cnoun,determiner-->deter, proper_name-->pname
```

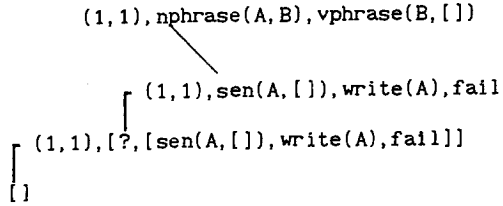
Sada ćemo korak za korakom izlagati algoritam, ali ne do samog kraja, jer inače ukupno ima 80 "plodova" te gramatike. Međutim, iz izlaganja će biti potpuno jasno kako se algoritam može sprovesti do samog kraja. Polazimo od ovog drveta tipa (5.4.3)



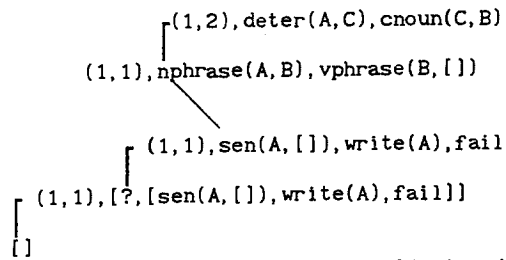
ali naravno koristimo Edinburgšku sintaksu, što inače nije presudno. U idućem koraku na osnovu definicije[1] ? imamo ovo drvo



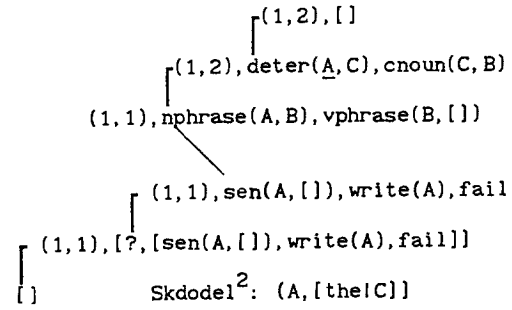
i žižnik je na sentence (A,[]). Sada na pozornicu stupa procedura penjanje na vrh. Ovde ćemo je prikazati postupno. Primenom procedure priključivanje na formulu sen(A,[]) dobije se ovo drvo



Sada je žižnik na nphrase(A,B). Njegovim priključivanjem nastaje ovo drvo



a žižnik je na deter (A,C). Primenom priključivanja na njega dobije se drvo

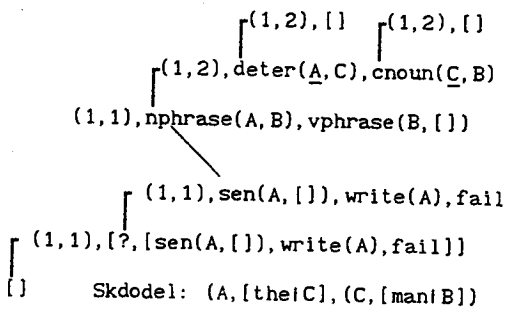


[1] Pozajmili smo tu definiciju iz Micro-prologa, a u Edinburgškim verzijama Prologa predikatu ? u stvari odgovara predikat call.

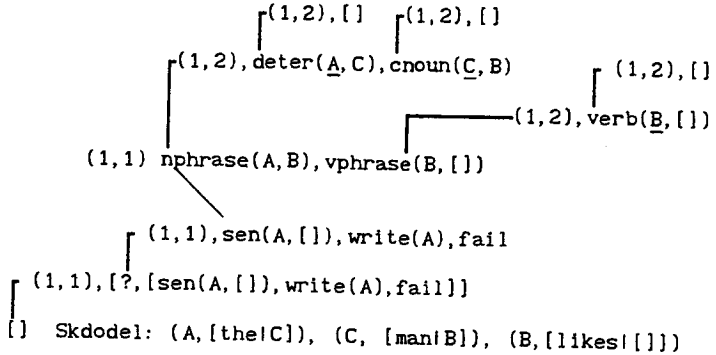
[2] To je skraćeni dodelnik u duhu (6.1)

Penjanje na vrh se završilo, jer pri priključivanju formule deter(A,C) je iskorišćen prvi deter-članak, koji je elementarna aksioma. Znači penjanje je obavljeno sa uspehom (formalno rečeno indikator je dobio vrednost 1). U Skdodel je uključena zamena (A,[the|C]). Na tom mestu je A dobilo vrednost.

Da bismo to lakše zapamtili na slici smo na tom mestu A podvukli[3]. Slično činimo i do kraja ovog primera.
Plus_spustom se odmah stiže na noun(C,B). Sada odatle nastaje novo penjanje na vrh. Nakon penjanja (u jednom koraku) dobije se se drvo



navedeno desno. Skdodel je "primio" i dvojku (C, [man:B]), pa smo C podvukli na mestu gde je dobilo vrednost ,tj. u formuli cnoun(C,B).
Sada nastaje plus_spust od mesta cnoun(C,B), pa pošto smo na kraju i-grane[4] to idemo na njen početak, spustimo se na nphrase(A,B) i pređemo na njenog dešnjaka tj. vphrase(B,[]), gde je kraj plus_spustu. Na redu je dokazivanje vphrase(B,[]), tj. penjanje na vrh. Na tom mestu B je prava nepoznata, jer još nije dobila neku vrednost, što se zaključuje iz toga što nikoja B dvojka se ne nalazi u Skdodel-u[5]. Nakon uspona na vrh drvo algoritma postaje



Dokaz je uspešno završen, u Skdodel je ušla i dvojka (B,[likes]|[]]), a B je podvučeno na mestu svog "vrednovanja".
Plus_spust počinje od mesta verb(B,[]). Završava se na mestu[6] write(A) i sada iz Skdodel-a se vidi da A=[the|[man|[likes|[]]]], tj. A=[the,man,likes] i na ekranu se štampa ta lista, prvi "plod" uočene gramatike.
Formalno rečeno dokazana je formula write (A), tj. popeli smo se na vrh.

---

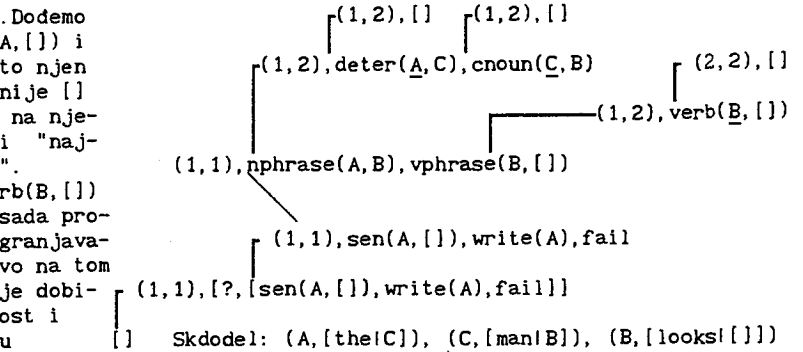
[3] U duhu (6.1)

[4] U stvari, smo na kraju liste [deter(A,C),cnoun(C,B)], jer i-grana je upravo ta lista. Međutim, često je slikovitije govoriti "i-grana".

[5] Jer Skdodel gradimo u duhu (6.1).

[6] Mislimo na mesto na "trećem spratu" drveta.

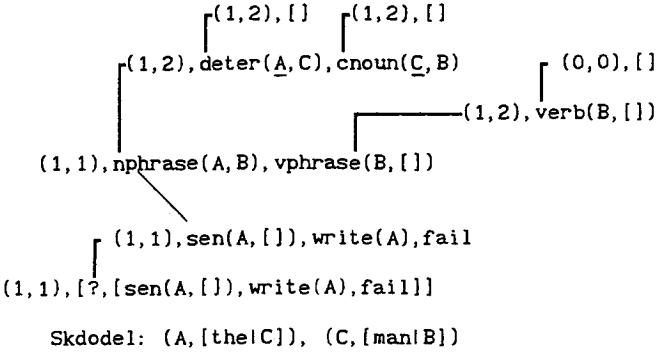
Plus_spustom dođemo do fail-a, od koga mora da nastane vraćanje. Prvo idemo na write(A), pa pošto je to jedno-grana formula[7], od nje nastavljamo vraćanje. Dođemo do sen(A,[]) i sada pošto njen gornjak nije [] ići ćemo na njega i naći "najdešnjaka". To je verb(B,[]) Tu ćemo sada probati pregranjavanje. Upravo na tom mestu B je dobila vrednost i stupila u



Skdodel[8]. Pregranjavanje je moguće i B dobije novu vrednost [looks|[]] i dobije se drvo algoritma prikazano na slici gore desno.
Sada od mesta verb(B,[]) nastaje plus_spust. Stignemo do write(A) i sada na osnovu Skdodel-a na ekranu se štampa lista [the,man,looks]. Ali, fail tera nazad, i brzo opet dođemo do najdešnjaka verb(B,[]), ali njegov gornjak je [], a još red=duz. Znači sada za verb(B,[]) ne možemo napraviti nikakav novi dokaz, jer on je "istrošen". Tu uradimo (5.5.5), tj. toj formuli za gornjaka stavimo (0,0,[]). Međutim, u skladu sa okolnošću da koristimo skraćeni dodelnik uradićemo i ovo:

(6.2) Promenljive koje su "vrednovane" upravo u toj formuli izbacićemo iz Skdodel-a.

tj. izbacujemo B. Drvo algoritma trenutno postaje



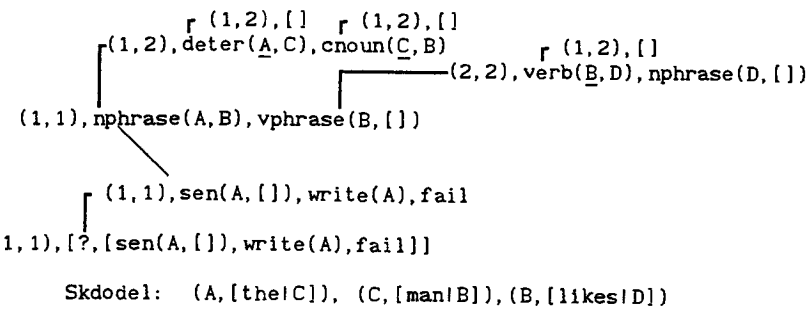
Od formule verb(B,[]) sada treba da počne nova procedura vraćanje. Pošto je ta formula na levom kraju to se spustimo na dolnjaka, odnosno na formulu vphrase(B,[]). Sada gledamo da li tog dolnjaka možemo nekako drukčije doka-

---

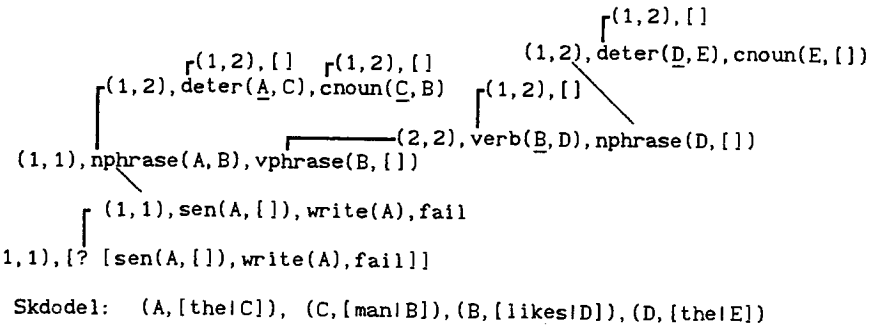
[7] Iz tog razloga joj nismo stavili nikakvog gornjaka.

[8] Zato smo je prethodno na tom mestu podvukli.

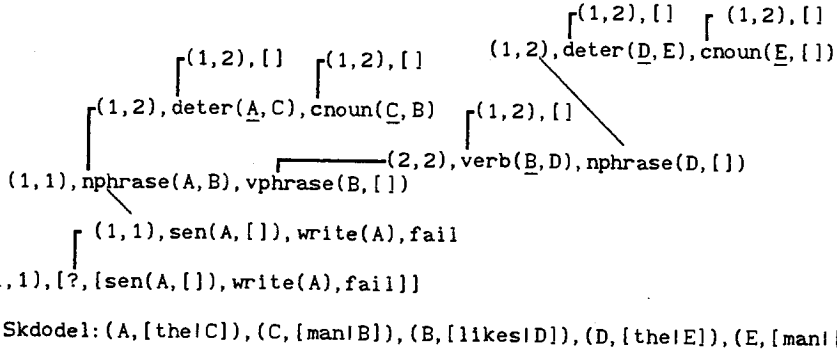
zati. Nad njim stoji red=1,duz=2, tj.red<duz pa obavljamo pregranjavanje[9]. Upošljavamo drugi članak tog dolnjaka. I penjemo se na vrh. Drvo postaje



Dokazavši verb(B,D) , što je " plaćeno " unošenjem u Skdodel dvojke[10] (B,[likes|D]) plus_spustom dodemo do nphrase(D,[]). Odatle nastaje penjanje na vrh. Za to koristimo prvu nphrase-granu kao i prvu deter-granu. Nepoznata D na tom mestu dobija vrednost [the|E], dvojka (D,[the|E]) se unosi u Skdodel ,i još se D podvuče na mestu deter(D,E). Drvo postaje



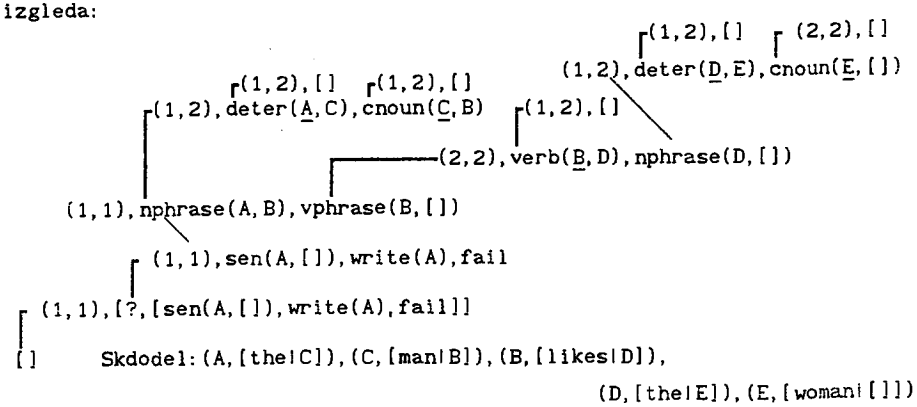
Nakon spusta žižnik je na cnoun(E,[]). Opet idemo "na vrh". Drvo izgleda:



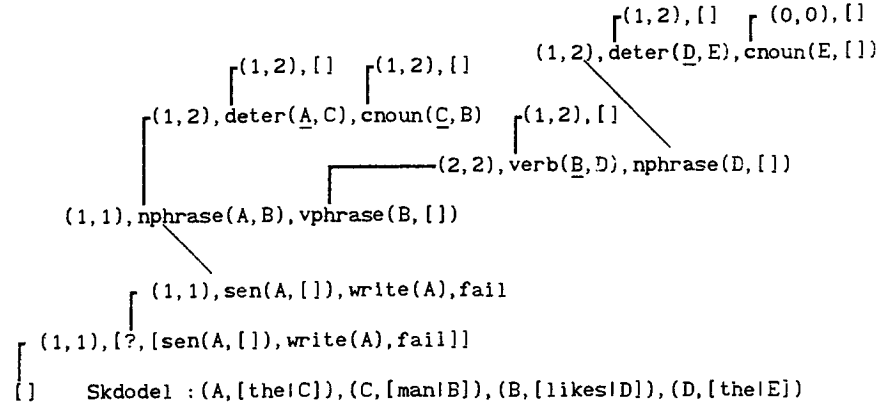
[9] Uradimo i (5.5.4)

[10] Na_tom mestu u formuli verb(B,D) je sada B podvučeno.

i Skdodel je dobio "prinovu" (E,[man|[]]), a E je podvučeno u cnoun(E,[]). Sada nastaje zanimljiv plus_spust, jer triput zaredom smo na kraju i-grane Plus_spustom se dode do write(A) i na ekranu, na osnovu Skdodel-a, se štampa lista [the,man,likes,the,man]. Ali, došavši na fail opet nastaje vraćanje. Brzo dodemo do najdešnjaka koji je ovde cnoun(E,[]) i tražimo mu drugi dokaz. To nam uspe upošljavanjem drugog cnoun-članka, jer E je na tom mestu dobila vrednost pa smo da je promeni. Opet smo "na vrhu". Drvo algoritma izgleda:



Pazite sada u Skdodelu stoji dvojka (E,[woman|[]]). Plus_spustom opet dodemo do write(A) i sada se na ekranu pojavi lista [the,man,likes,the,woman]. Opet nas fail tera nazad i dotera do najdešnjaka cnoun(E,[]). Ali, slobodnije rečeno "on je istrošen". Znači, sada od tog mesta nastaje vraćanje, s tim da se takođe uradi (5.5.5) i (6.2). To znači da iz Skdodel-a izbacujemo E. Novi gornjak za cnoun(E,[]) je (0,0,[]). Trenutno drvo izgleda



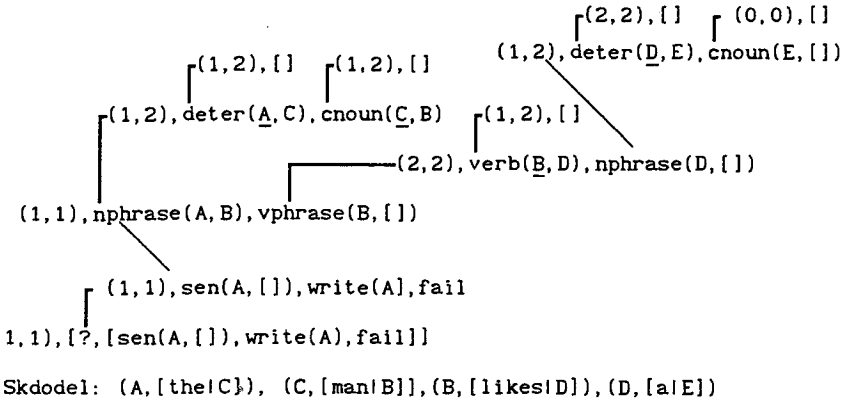
Vraćanjem se odmah dode do levaka deter(D,E) koji ne izgleda "istrošen", jer i ako mu je gornjak [] njegov red<duz. Obavimo pregranjavanje[11] dopuštajući da D,E dobiju nove vrednosti ,i to iz ovih razloga

D je na tom mestu vrednovana, a

[11] Uradimo i (5.5.4)

E nije član Skdodel-a (jer je prethodno bila izbačena)[12]

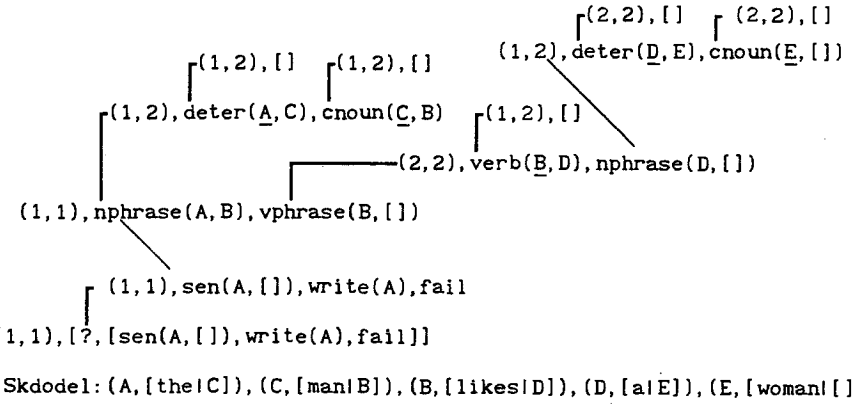
Pregranjavanje iskoristi drugi deter-članak. Opet smo na vrhu, a drvo je:



Plus_spustom odmah dođemo do cnoun(E,[]). I sada je veoma značajno:

Premda je maločas ta formula bila "istrošena", sada -budući da smo na nju došli plus_spustom --je posmatramo kao potpuno novu.

U stvari, samo dosadašnje delanje u algoritmu će nam to obezbediti. Naime, kratko rečeno tražimo dokaz za cnoun(E,[]) i zagledom u Skdodel vidimo da se E tamo ne nalazi, tj. ta nepoznata još[13] nije dobila vrednost. Tu formulu cnoun(E,[]) ćemo prvo dokazati uz zamenu E-->[man|[]] i na ekranu će kad se spustimo na write(A) biti štampano [the,man,likes,a,man]. Ali, zbog fail kad se vratimo do tog "čoška", tj. najdešnjaka ubrzo ćemo stići do novog A, i opet plus_spustom na write(A) na ekranu će se štampati lista [the,man, likes,a,woman]. U tom trenutku drvo algoritma će biti

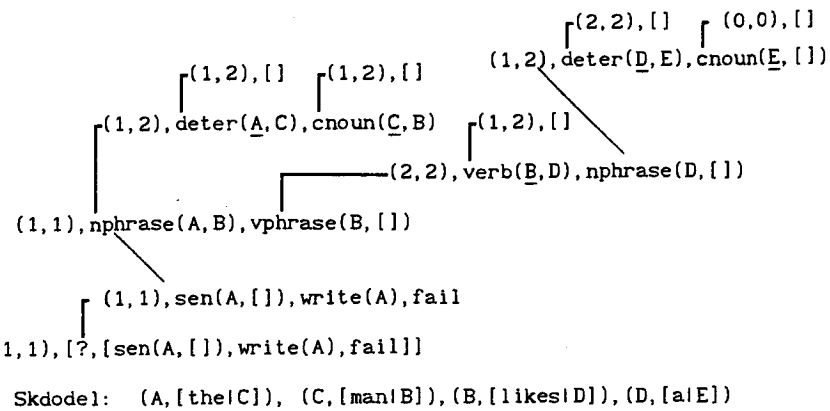


Pazite za deter(D,E) sada stoji (2,2), tj. sve grane su potrošene. Sada će

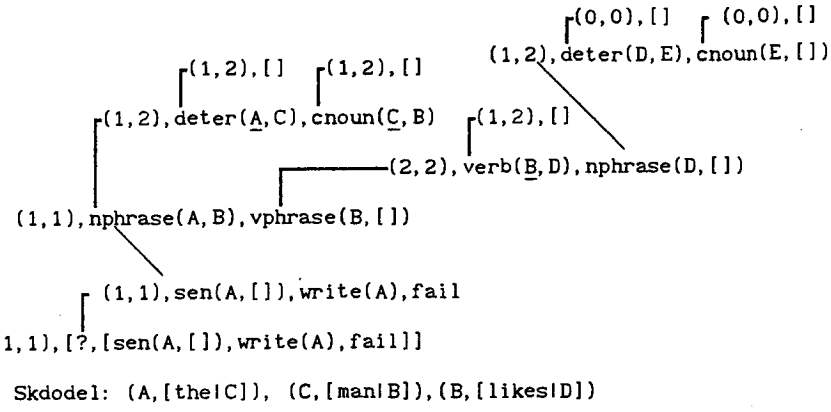
[12] Eto lepe koristi od izbacivanja E.

[13] Znači, opet E vracamo u Skdodel i podvlačimo ga u cnoun(E,[]).

nas fail prvo poterati do najdešnjaka cnoun(E,[]). Ali on je "istrošen". Stoga, opet E, jer je tu bila vrednovana, izbacimo iz Skdodel-a i još toj formuli stavimo gornjak (0,0,[])[14]. Drvo trenutno izgleda



Od mesta cnoun(E,[]) nastaje vraćanje. Dođemo do levaka deter(D,E), takođe istrošenog. Stoga i D izbacujemo iz Dodelnika (jer tu je bilo vrednovano) i još nad deter(D,E) stavimo (0,0,[]) kao gornjaka. Drvo trenutno glasi

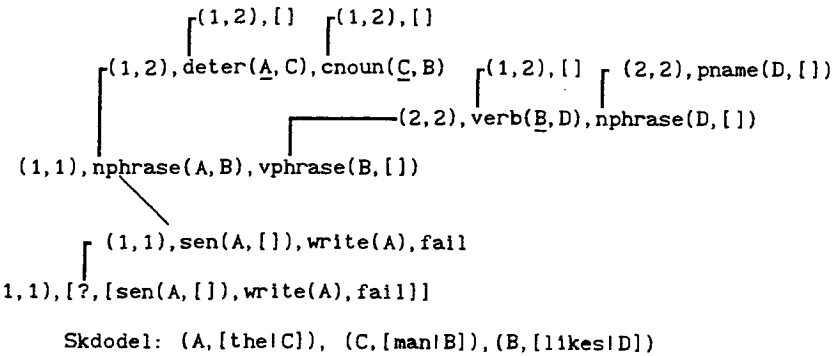


Znači sada vraćanje treba da počne od formule deter(D,E). Pošto ta formula nema levaka, spustimo sa na nphrase(D,[]). Pošto je red manji od duz obavimo pregranjavanje[15] po formuli nphrase(D,[]), gde D nepoznata koje nema u Skdodel-u. U tu svrhu iskoristimo drugu nphrase-granu[16]. Drvo trenutno izgleda

[14] To je u skladu sa (5.5.5).

[15] Naravno imamo na umu i (5.5.4).

[16] Naravno, usled pregranjavanja prethodna grana je izgubila svaki značaj za tok algoritma. To je važno pri eventualnom pisanju Prologa jer na takvom mestu se pruža prilika oslobađanja dela memorije.



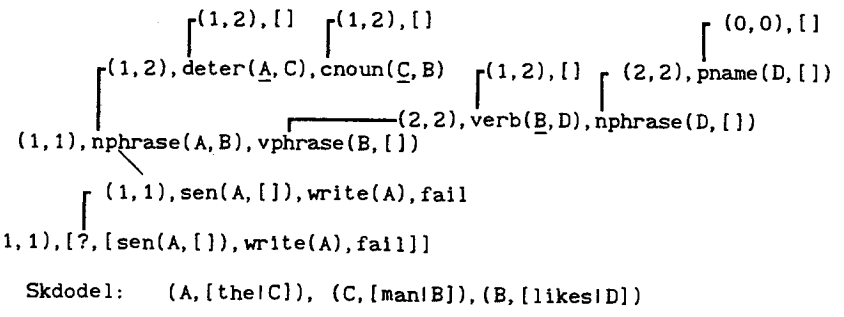
i žižnik je na pname(D,[]), odakle nastaje penjanje na vrh. To penjanje daje D-->['John'|[]] i plus_spustom se stigne do write(A), kada se štampa

[the,man,likes,John]

Ali, fail tera nazad i našavši nov dokaz za pname(D,[]) na ekranu se, opet nakon spusta na write(A), štampa lista

[the,man,likes,Vera]

Sada nas opet fail tera nazad i ponovo dođemo do pname(D,[]), koja je sada istrošena. Njeno D izbacimo iz Skdodel-a i njoj stavimo gornjak (0,0,[]). Drvo trenutno izgleda

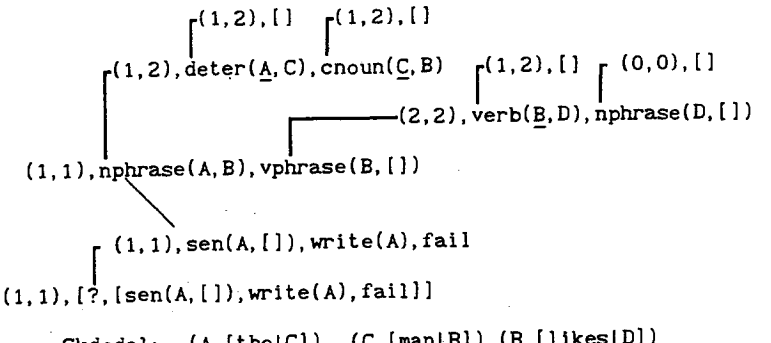


i spustimo se dole na nphrase(D,[]) ne bili smo njemu pregranjavanjem našli nov dokaz. To nije moguće jer u njemu red=duz, pa od tog mesta nastaje novo vraćanje, s tim da taj dolnjak ovako "preuredimo":

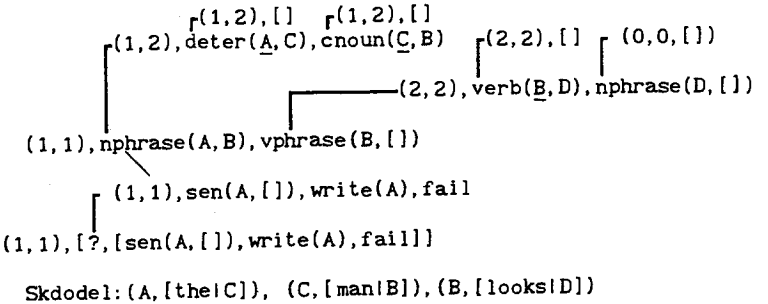
1) sve nepoznate koje su u njemu bile vrednovane izbacimo iz Skdodel-a, što je ovde bez dejstva jer D nije tu bilo vrednovano
2) sklonimo[17] mu gornjak i kao novi mu stavimo (0,0,[]).

Drvo trenutno izgleda

[17] U skladu sa (5.5.5).



i žižnik je na verb(B,D). Sada nastaje verb-pregranjavanje[18] korišćenjem drugog verb-članka, pri čemu B je prava nepoznata (jer na tom mestu je vrednovana). Penjanje na vrh se odmah završava, a "u igru sada ulazi glagol looks", jer sada se u Skdodel stavi dvojka (B,[looks|D]). Drvo postaje



a žižnik je na nphrase(D,[]). Tu smo stigli nakon plus_spusta pa tražimo dokaz za nphrase(D,[]) "kao od početka", pri čemu je D prava nepoznata. Odatle ćemo ukratko ispričati tok algoritma do kraja. Primetimo da do sada imamo određene A,C,B -sve pomoću D. Može se reći da

A=[the man looks|D]

Kako rekosmo treba da dokažemo nphrase(D,[]), gde D nepoznata. Ali, prema sklopu drveta, nakon dokaza ćemo se spustiti na write(A), štampati rešenje A, ali fail će nas vratiti nazad i svaki put, dok je to moguće, tražiti nov dokaz te formule. To je potpuno slična situacija kao na drvetu (δ1), na kome smo takođe bili pred dokazom iste formule, ali fail nas je primorao da "izpronađemo" sve dokaze, odnosno sve vrednosti za D i one su redom bile:

[the,man]
[the,woman]
[a,man]
[a,woman]
[John]
[Vera]

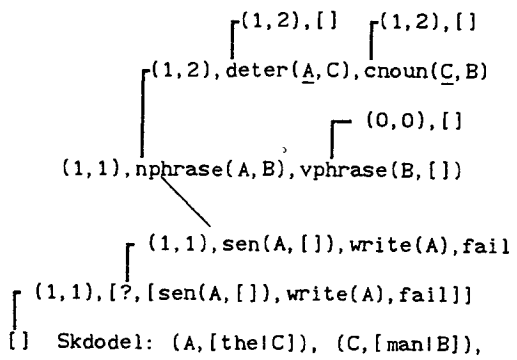
U stvari, javiće se D=[the|E],E=[man|[]]

[18] Imamo na umu (5.5.4).

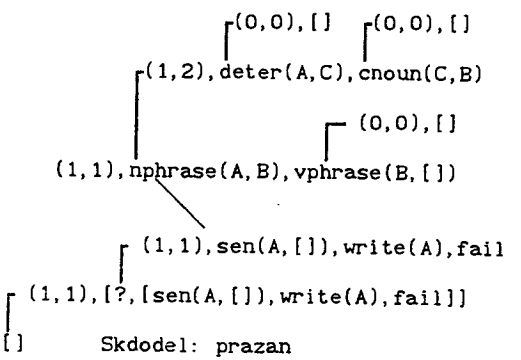
Sledstveno i sada u nastavku algoritma će se redom pojavljivati te vrednosti i za A ćemo dobiti 6 novih rešenja:

[the,man,looks,the,man]
[the,man,looks,the woman]
[the,man,looks,a,man]
[the,man,looks,a,woman]
[the,man,looks,John]
[the,man,looks,Vera]

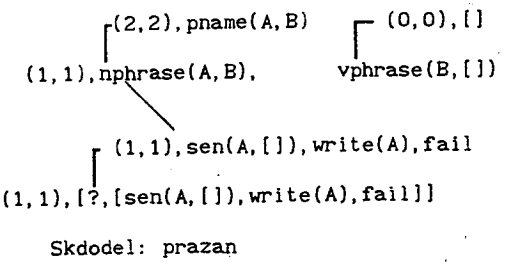
Ali, posle toga kada nas fail opet potera nazad prvo ćemo videti da je nphrase(D,[]) istrošena, takođe je istrošena i verb(B,D) jer smo iskoristili i njemu poslednju granu pa se moramo spustiti na vphrase(B,[]), i njega proglasiti za netačnog. Tako trenutno imamo drvo:



i treba da nastavimo vraćanje od vphrase(B,[]). Idemo levo do nphrase(A,B) i tražimo najdešnjaka. Tako se dođe do cnoun(C,B) i tu se potroši još jedna grana ovom vezom C=[woman|B]). A sada se otvara čitavo obilje novih rešenja. Naime, spustom stignemo do vphrase(B,[]) (zbog faila) tražeći mu sve moguće dokaze. Ta nova rešenja, može se tako reći, nastaju iz do sada navedenih kad se u njima reč man zameni rečju woman. A posle svega, opet nas fail potera nazad i onda se posle dođe do deter(A,C) tražeći mu novo rešenje što daje vezu A=[a|C]. Iza toga, budući da se slobodnije rečeno svi dešnjaci dokazuju iznova, pojave se sva dosadašnja rešenja uz zamenu reči the rečju a. I posle svih tih A-rešenja fail nas opet potera nazad i pošto je deter(A,C) postala potrošena suočimo se sa ovim drvetom



i treba od deter(A,C) da nastavimo vraćanje. Spustimo se na dolnjaka i pregranimo se po nphrase upošljavajući drugi nphrase-članak. Drvo izgleda



Žižnik se trenutno nalazi na pname(A,B). Znači, očekuje nas da tu formulu izdokazujemo na sve moguće načine. Tako, sve dosadašnje rečenice počinju sa

the man, the woman, a man, a woman

Novodobijena A-rešenja nastaju iz njih kad im se za početak umesto takvog uzme reč John, odnosno Vera. U stvari, tako sve u svemu se pojavi 80 rešenja. I posle svega zbog fail-a, formalni odgovor na postavljeno pitanje će biti no. Kraj algoritma.

Zadatak 6.2. Napraviti program za ovaj algoritam:
Članovi liste se daju jedan za drugim, sa među-prekidima u kojima se na ekranu štampa poruka Daj. Za završetak se daje znak @.

Rešenje. Jedan takav program glasi:

```
((upis _Rez) (PP Daj) (R _Y)
   (IF (EQ _Y @) ((PP To je lista _Rez))
         ((upis (_Y|_Rez)))
   )
)
```

Pobuđuje se sa ?((upis ()))

Zadatak 6.3. Ispisati član za članom datu listu, usput na postavljeno pitanje 'Brisati ga: Da/Ne ' neke članove po želji izbaciti iz liste.

Zadatak 6.4. Slično osnovnom predikatu ? uvesti predikat 'Koji', čije dejstvo je da pri računu formule oblika (Koji _A), gde je _A neka formula, daje sve vrednosti promenljivih formula A za koje je ona tačna i uz to na kraju štampa poruku 'Nema više'. Recimo, jedan oblik Koji-pitanja je

```
?((Koji (a _x)))
```

Rešenje. Jedan program glasi

```
((Koji|_X) (? _X)(EQ _X ((_Ime|_Y))) (PP _Y)FAIL)
((Koji|_X)(PP Nema vise))
```

Evo primera pitanja sa tim predikatom Koji

```
?((Koji (a _x _y)))
```

pretpostavljajući naravno da je relacija a definisana.

Zadatak 6.5. Kako za datu relaciju a:

```
((a 1))  ((a 2))   ((a 3))  ((a 4)) ((a 5))
((a 6))  ((a 7)) ((a 8))  ((a 9))   ((a 10))
```

naći upravo k prvih rešenja ?

Rešenje. Uz prisutne a-članke, jedan program glasi

```
((br 0))
((pit _z)(a _y) (PP Resenje _y)
                (br _x) (SUM 1 _x _xx)
                (DELCL ((br _x))) (ADDCL ((br _xx)))
                (IF (EQ _xx _z) ((KILL br)(ADDCL ((br 0))))
                   (FAIL)
                )
        )
```

On se pobuđuje recimo sa ?((pit 3)) i tako se dobiju 3 a-rešenja. Predikat br ima ulogu brojača.

Zadatak 6.6. Definisati predikat (Ako _A _B) koji se ovako računa:
Ako je _A tačno, onda se dalje računa _B , a
ako je _A netačno formula (Ako _A _B) je tačna

Rešenje. U Micro-prologu imamo ovo rešenje:

```
((Ako _A _B) _A /!_B)     ((Ako _A _B))
```

Tu naravno _A mora biti formula, a _B oblika "repa nekog članka". Recimo, nije dobro (Ako (EQ 1 1) (print Jeste)) dok (Ako (EQ 1 1) ((print Jeste))) jeste. Međutim, Arity i LPA imaju ugrađen takav predikat:

ifthen(A,B) -u Arity, odnosno A -->B u LPA.

S tim u vezi dodajmo da se slično tome u njima If-then-else predikat zapisuje redom ovako

ifthenelse(A,B,C), odnosno A --> B;C.

Evo jednog primera: a,b --> c,d;e,f sa smislom: (a,b) -->(c,d);(e,f)

Zadatak 6.7. Zamislimo da nam se pri pravljenju nekog programa pojavljuje potreba za drvetom oblika

```
┌ q,r,S,t,u
p
```

za neke p,q,... ali da uz to želimo da se program ovako "vlada"

Ako je S netačno da se izađe iz te grane i da se štampa poruka 'Kraj'.

Kako "dopraviti" navedeni članak ?

Rešenje. Prolog kao jezik nije najpodesniji za takve GO-TO ideje, ali uz određene dosetke i to se može ostvariti. Naime, problem je što je zamišljeno ljeno S u sredini, a ne na kraju grane. Stoga, ako S bude tačno, algoritam će nastaviti sa računanjem t, i iza toga (ako t tačno) sa računanjem u, pa ako i ono bude tačno, onda se možemo "odlepiti" od te grane. S druge strane, ako S netačno, onda nas čeka vraćanje nazad- što ne želimo. Ako umesto S stavimo !,S onda ćemo zaista postići "iskakanje iz grane", ali moramo voditi računa da će nam se onda dogoditi da (u duhu CUT-pravila) odemo do p i odatle započnemo vraćanje. Međutim uz to ako nam se dogodi da se pri računanju nekog desnjaka od S se pojavi potreba za vraćanjem i ako nas ona natera na taj znak !, pitanje je da li želimo takvo vladanje programa. Naravno imajući sve to na umu u nekim slučajevima ta CUT-ideja se može upotrebiti. Ali, što je važno, ima rešenja i bez upotrebe reza.

P r v i način: pomoću IF-THEN-ELSE predikata (dajemo Micro-prolog zapis)
Umesto gornjeg članka napravimo ova dva

```
(p q r (IF S (t u) ((aa))))
((aa)(print Kraj))
```

Primetite da smo u stvari uspeli da S, ako je netačno, "oteramo" na kraj grane.

D r u g i način: pomoću ILI-predikata (dajemo Edinburgški zapis)

```
p:-q,t,(not(S),aa;t,u).
aa:-write('Kraj').
```

Zadatak 6.8. U Arity-prologu postoji jedan zanimljiv predikat snip. Recimo, u članku oblika

```
p:-q,r,[! a,b,c !],s,t.
```

taj predikat odgovara delu [! ....!]. Njegov smisao je sličan i različan od smisla reza. Naime, ako se dogodi da je s netačno, pa stoga odatle treba da nastane vraćanje, onda snip-predikat poručuje: "Mene preskočite", tj. ide se na mesto r, itd. Da li se taj predikat može definisati preko ostalih osnovnih predikata ?

Rešenje. Predikat snip možemo kratko ovako definisati[19]

```
((snip!_X) _X) /)
```

pri čemu naravno _X se mora zamenjivati "repom članka". Evo konkretnog primera:

```
((a)(p Pera)(snip!((p Dara)(p Mara)))(b 7))
((p _X)(PP _X))
((p _X)(PP Jos _X))
((b 2))
```

U tom programu se pojavljuju relacije p ,b i a. Na pitanje ?((a)) kad se zbog netačnosti formule (b 7) pojavi potreba vraćanja, algoritmom ćemo preskočiti snip-formulu i doći do (p Pera), kada će se štampati reči Jos Pera. Razlog ? Sve se lepo vidi sa drveta

```
                                ┌ (1,1)(PP Dara)(PP Mara)), /
                                │
┌ (1,1)(p Pera),(snip ((p Dara)(p Mara))),(b 7)
│
(a)
```

nacrtanog sa početkom (a), tj. kome jedan pred-deo je izostavljen. Sa drveta se vidi kad dođemo do (b 7), i odatle budemo morali da se vraćamo, onda dolazimo do snip-formule i na njenom drvetu nađemo "najdešnjaka". to je rez /. Stoga, prema pravilu o rezu moramo se spustiti na snip i od njega se vraćati. Praktično znači bili smo primorani da snip preskočimo, što nam je i inače bio zadatak.
Primetimo da uvedeni snip-predikat u stvari traži prvo rešenje, tj. pri računanju (snip _X) se za _X uzima samo jedna mogućnost. Micro-prolog ima takav predikat, u oznaci ! ,što ne treba brkati sa znakom reza u Edinburgškoj verziji. Njegova definicija glasi ((!!_A) _A /) što se bukvalno slaže

[19] U Edinburgškoj verziji to bi moglo da glasi ovako: snip(X):-X,!.

sa snip-definicijom.

Zadatak 6.9. Napraviti predikat koji ostvaruje ovo delanje:

```
Ako važi A1 uradi B1,inače
Ako važi A2 uradi B2,inače
....
Ako važi Ak,uradi Bk
```

Rešenje. To je prilično "sistemski" zadatak. Slično, kao kad bi neko želeo da mu u Prolog ugradimo IF-predikat,pod pretpostavkom da ga taj Prolog nema.Bitno je pri pisanju "sistemskih" programa voditi računa o tome da Prolog dati članak D "shvata" kao drvo, odnosno listu

```
((for1|(for2|(for3| ...  |()...)
```

gde for1,for2,..koje smo zvali CAR-ovi od D^{20} moraju u trenutku kad na njih dođe red postati formule. Imajući to na umu evo prvo jedne definicije predikata sluc:

```
((sluc _for1 _for2|_X) _for1 _for2)
((sluc _for1 _for2|_X)(sluc|_X))
((sluc|()))
```

gde u skladu sa rečenim vrednosti promenljivih _for1, _for2 moraju biti formule, a _X mora biti rep nekog članka. U skladu sa tim,recimo pri pitanju

```
?((sluc (EQ 1 2) (PP Nisu) (EQ 4 4) (PP Jednaki)))
```

na ekranu će se štampati Jednaki.Međutim, ne smemo dati ovo pitanje

```
?((sluc (EQ 5 5) ((PP Prvi)(PP Drugi))))
```

jer deo ((PP Prvi)(PP Drugi)) nije formula[21], već oblika repa članka. Pa da li ima načina, da ga ipak nateramo da štampa reči Prvi,Drugi ? Jasno je, sporni deo moramo pretvoriti u formulu, što se po pravilu radi pomoću predikata ?.Tako, pitanjem

```
?((sluc (EQ 5 5) (? ((PP Prvi)(PP Drugi)))))
```

na ekranu će se štampati navedene reči. Međutim,može se potražiti popravka relacije sluc tako da ne moramo mi sami da stavljamo taj ?.Tu ćemo relaciju zvati slucaj i njena definicija glasi

```
((slucaj _for1 _Rep|_X) _for1 (? _Rep))
((slucaj _for1 _Rep|_X)(slucaj|_X))
((slucaj|()))
```

Sada se u pisanju slucaj-formule redom smenjuju formula i rep članka. Evo ispravnih primera uporebe relacije slucaj:

1)Na pitanje[22]

```
?((slucaj (EQ 2 3) ((PP 2)(PP 3))
```

[20] U stvari, for1=car(D), for2=car(cdr(D)),...

[21] Prolog će prekinuti algoritam porukom: CONTROL ERROR.

[22] Pomenimo da smo umesto dela ((PP 2)(PP 3)) smeli staviti skoro bilo šta kao 777, jer Prolog u svom "računanju" neće stići na taj deo. Razlog:formula (EQ 2 3) je netačna.


```
          (EQ 4 5) ((R _x)(PP Evo _x))
          (EQ 7 7) ((PP Kraj)(PP 77))
  ))
```

će se na ekranu štampati reči: Kraj, 77.

2) Evo malog programa uz korišćenje te relacije slucaj

```
((pit)(PP Daj)(R _x)
        (slucaj  (EQ _x 3) ((PP Tri)(PP Da tri))
             (EQ _x 4) ((PP Cet)(PP Da cetri))
             )
            )
```

U vezi sa izloženim istaknimo da Arity-prolog ima ugrađen case predikat koji je malo širi od maločas uvedenog slučaj predikata (u Micro-prologu). Evo primera:

```
case([X=0 -> write('Broj Nula'),
      X=1 -> write('Broj Jedan')|
      write('Kraj ').
```

Znači ako je X jednak 0 štampaće se reč Broj Nula, ako X jednak 1 reč Broj Jedan,a inače,tj. ako X nije ni 0, ni 1 štampaće se reč Kraj. Uopšte,case formula ima jedan od ova dva oblika:

```
case([A1->B1,...,An->Bn|Inače])
```
Smisao: Ako A1 izračunaj(uradi) B1,inače ...,Ako An izračunaj Bn,a inače,tj. kad nijedan od A1,...,An nije tačan, izračunaj deo Inače.

```
case([A1->B1,...,An->Bn]).
```
Smisao: Ako A1 izračunaj B1,inače ...,Ako An izračunaj Bn, a ako nijedan od A1,...,An nije tačan onda case-formula je, po definiciji, tačna.

Zadatak 6.10. Neka je data binarna relacija kao što je relacija a zadana ovim člancima (elementarnim aksiomama)

```
((a 1 2))  ((a 2 3))   ((a 2 4))   ((a 3 7))
((a 4 5))  ((a 7 88))  ((a 5 3))
```

Naći minimalnu tranzitivnu relaciju koja sadrži tu relaciju.

Rešenje.Osnovna zamisao je

```
Videti da li je (a _x _y), (a _y _z), ali NIJE (a _x _z).
Ako se to desi dodati članak ((a _x _z)), i ići nazad, da bi
se opet tako što uradilo (dokle god može).
A kad više ne može, recimo, odštampati dobijenu relaciju a
```

Jedan takav program je:

```
((utran _a)(uslov _a _x _y _z) FAIL)
((utran _a)(LIST _a))
((uslov _a _x _y _z)(_a _x _y)(a _y _z)
                    (NOT _a _x _z)(ADDCL ((_a _x _z))))
```

Program se pobuđuje sa ?((utran a)), naravno ako želimo da "utrazitivimo" baš relaciju a.
Kao što vidite program je začudujeće kratak i jednostavan,ali kako dokazati da je u opštem slučaju dobar.Da bismo to lakše videli uvešćemo određeno izražavanje.Naime,neka je poput a uopšte zadana izvesna relacija rel-zadana navođenjem slučajeva važenja, tj. člancima oblika ((rel a b)), za neke konstante a,b. Pretpostavimo, sada da se pomoću takvih "datosti" upravo u

k koraka zaključuje da je neko A tranzitivno povezivo sa nekim B. Tada ćemo reći da je taj zaključak (rel A B) izveden na "k-tom spratu". Traženi dokaz se izvodi indukcijom "po spratovima". Naime, prvo se lako može videti da će navedeni program prvo proizvesti sve prvospratne zaključke, i njih dodati na kraj "datosti". Blagodareći tome u narednom delu program će proizvesti sve drugospratne zaključke, i slično dalje.

Zadatak 6.11. Datu relaciju, određenu navođenjem slučajeva važenja, dopuniti do minimalne relacije ekvivalencije.

Uputstvo. U tu svrhu je dovoljno da se data relacija najpre "usimetri", a potom primenom programa iz prethodnoh primera "utranzitivi".

Zadatak 6.12. Data[23] je osnovna relacija susednost u oznaci sused. Za date A, B odrediti sve "tranzitivne" putanje od A do B.

Rešenje. U stvari, glavne zamisli (ideja "torbe", tj. putanje) su već izložene u Primeru 2.4.6. Njegovom malom "doradom" se dolazi do ovog programa

```
((Putanje _x _y) (r () _x _y _Put) (PP Putanja : _Put) FAIL)
((Putanje|_X))
((psi _A _x)(NOT VAR _x)(ON _x _A))
    ((r _A _x _y _B)  (NOT sused _x _z)  /  FAIL)
    ((r _A _x _y _B)  (NOT sused  _z _y)  /  FAIL)
    ((r _A _x _y (_y _x|_A)) (sused _x _y))
    ((r _A _x _z _B)(NOT psi _A _x)(sused _x _y)(r (_x|_A) _y _z _B))
```

Pretpostavimo da je sused data relacija, kao:

```
((sused 1 2))   ((sused 2 66))   ((sused 2 5))
((sused 5 10))  ((sused 2 7))    ((sused 7 9))   ((sused 9 10))
```

Tada pitanjem oblika ?((Putanje 1 10)) će se štampati sve putanje, odnosno

```
(10 5 2 1)    (10 9 7 2 1)
```

Primetite da je jedino redosled "preokrenut", jer odgovara redosledu punjenja "torbe", tj. putanje.

Zadatak 6.13. (Nastavak prethodnog). Opet je zadana relacija sused, ali sada su zadata i rastojanja među "tačkama". Jedan takav primer je određen članicima:

```
((sused 1 2 10)) ((sused 2 3 20))  ((sused  2 77 90))
((sused 2 80 78)) ((sused 80 7 88)) ((sused 88 4 45))
((sused 3 4 30)) ((sused  4 5 40)) ((sused 7 3 55))
```

Sada je zadatak da se odrede sve tranzitivne putanje od _x do _y, i uz to odredi i medju-rastojanje od _x do _y (po putanji).

Rešenje. Rešenje je slično prethodnom, ali sada imamo dve "torbe", jednu za putanju, a jednu za rastojanje. Jedan takav program glasi:

```
((r _A _x _y _B _Poc _Zbir)(NOT sused _x _z _Duz)/FAIL)
((r _A _x _y _B _Poc _Zbir)(NOT sused _z _y _Duz)/FAIL)
((r _A _x _y (_y|(_x|_A)) _Poc _Rez)(sused _x _y _Duz)(SUM_Duz _Poc _Rez))
((r _A _x _z _B  _Poc _Rez)
   (NOT psi _A _x) (sused _x _y _Duz)
        (r (_x|_A) _y _z _B _Poc _Zbir)
        (SUM _Duz _Zbir _Rez))
((psi _A _x)(NOT VAR _x)(ON _x _A))
((rr _x _y)(r () _x _y _Put 0 _Zbir )(PP Putanja:_Put,a Zbir = _Zbir)FAIL)
((rr|_X))
```

Recimo, na pitanje ?((rr 1 5)) se na ekranu štampa:

```
Putanja (5 4 3 2 1), a Zbir=  100
Putanja (5 4 3 7 80 2 1), a Zbir= 301.
```

Zadatak 6.14. (Nastavak prethodnog). To je poznat problem trgovačkog putnika (salesman problem). Naime, uz postavke prethodnog zadatka pronaći najkraću putanju od tačke _x do tačke _y.

Rešenje. Rešenje koje navodimo je mala dorada prethodnog. Tako bez FAIL-a, a uz uslugu ISALL predikata se najpre nađu sve putanje sa udaljenjima, a onda se pomoću minimum-predikata (uobičajeno traženje minimuma) nađe traženo rešenje. Program glasi:

```
((r _A _x _y _B _Poc _Zbir)(NOT sused _x _z _Duz)/FAIL)
((r _A _x _y _B _Poc _Zbir)(NOT sused _z _y _Duz)/FAIL)
((r _A _x _y (_y|(_x|_A)) _Poc _Rez)(sused _x _y _Duz)
                                    (SUM _Duz _Poc _Rez))
((r _A _x _z _B  _Poc _Rez) (NOT psi _A _x) (sused _x _y _Duz)
      (r (_x|_A) _y _z _B _Poc _Zbir) (SUM _Duz _Zbir _Rez))

((psi _A _x)(NOT VAR _x)(ON _x _A))

      ((minimum ((_Put _Zbir)) (_Put _Zbir)))
      ((minimum ((_Put _Zbir)|_X)  (_Put _Zbir))
                (minimum _X (_Put1 _Zbir1))(LESS _Zbir _Zbir1))
      ((minimum ((_Put _Zbir)|_X) (_Put1 _Zbir1))
                (minimum _X (_Put1 _Zbir1)))

((putnik _x _y)(ISALL _Lis (_Put _Zbir)  (r () _x _y _Put 0 _Zbir))
                (PP Imam _Lis)(minimum _Lis _Res)(PP Evo resenja _Res))
```

Osnovni predikat je putnik. Recimo, uz relaciju sused definisanu u prethodnom zadatku na pitanje ?((putnik 1 5)) na ekranu se pojavi:

```
Imam   (((5 4 3 2 1) 100)  ((5 4 3 7 80 2 1) 301))
Evo resenja  ((5 4 3 2 1) 100)
```

tj. pojavi se lista dvojaka (putanja zbir), a na kraju i dvojka kojoj ogovara minimum.

Zadatak 6.15. Na jednoj strani reke se nalaze otac i njegova dva sina. Oni žele da pređu na drugu stranu reke i na raspolaganju im je jedan čamac. Kako da pređu na drugu stranu reke uz ove uslove:

> Otac može sam da pređe reku, može on sa još jednim sinom. Dalje, mogu sinovi ponaosob, a takođe i dva sina zajedno.

Znači jedino nije dozvoljeno da čamcem zajedno pređu i otac i oba sina.

Rešenje. To je jedan od tzv. zadataka prelaza, i što je važno mnogi od njih se rešavaju na način koji opisujemo za ovaj problem. Da bismo problem "poduhvatili" matematički u igru uključujemo razne četvorke kao

```
(1 1 1 1)   (1 0 0 1)
```

čiji članovi su 0 ili 1. Uopšte kod četvorke (Cam Otac Sin1 Sin2) koordi-

[23] Navođenjem svih slučajeva ispunjenja.

nate su dogovorno redom u vezi sa položajem čamca, oca, prvog i drugog sina. Dalje, dogovorno sa (1 1 1 1) označavamo početno stanje, kad su svi sa jedne strane reke. Ako recimo otac pređe čamcem na drugu obalu tada na njoj ćemo imati ovo stanje, ovu četvorku: (0 0 1 1). Kao što se vidi u problemu se jasno pojavljuje "svet" svih stanja, kao i jedna relacija susednost među njima:

> Stanje (a b c d) je susedno sa stanjem (a1 b1 c1 d1) ukoliko se iz prvog jednim prelaskom reke pređe na drugo stanje.

U stvari, dalje je jasno da je glavni problem da se definiše ta relacija susednost, jer onda prosto treba tražiti ma koju tranzitivnu putanju od od stanja (1 1 1 1) do stanja (0 0 0 0). Evo opisa relacije sused:

```
((sused (_Cam _Otac _Sin1 _Sin2) (_Camm _Otacc _Sin1 _Sin2))
          (EQ _Cam _Otac)(zbir _Cam _Camm)(zbir _Otac _Otacc))
((sused (_Cam _Otac _Sin1 _Sin2) (_Camm _Otac _Sinn1 _Sin2))
          (EQ _Cam _Sin1)(zbir _Cam _Camm)(zbir _Sin1 _Sinn1))
((sused (_Cam _Otac _Sin1 _Sin2) (_Camm _Otac _Sin1 _Sinn2))
          (EQ _Cam _Sin2)(zbir _Cam _Camm)(zbir _Sin2 _Sinn2))
((sused (_Cam _Otac _Sin1 _Sin2) (_Camm _Otac _Sinn1 _Sinn2))
          (EQ _Cam _Sin1)(EQ _Cam _Sin2)
          (zbir _Cam _Camm)(zbir _Sin1 _Sinn1)(zbir _Sin2 _Sinn2))

((zbir 0 1))
((zbir 1 0))
```

Primera radi, potražimo susedno stanje za (1 1 1 1), tj. početno stanje. Primenom prvog sused-članka dobiće se (0 0 1 1), jer:

> Prvo, uslov (EQ _Cam _Otac) važi, a njegov smisao je da su otac i čamac na istoj strani reke. Dalje, _Camm i _Otacc su nove vrednosti nepoznatih _Cam, _Otac. Te nove vrednosti se traže pomoću predikata zbir, definisanog navedenim zbir-člancima. Kratko rečeno taj predikat preobraća 0 u 1, odnosno 1 u 0. Stoga će 0,0 biti nove vrednosti za _Camm, _Otacc. Dakle, po prvom sused-članku zaključujemo da su stanja (1 1 1 1), (0 0 1 1) susedna što odgovara činjenici da otac sme sam da pređe reku. Nije teško videti da, opet po prvom članku sledi i da važi obratno, tj. (0 0 1 1) je u relaciji sused sa (1 1 1 1)

Primetimo, da smo mogli relaciju sused zapisati i kraće, recimo izbacivanjem predikata EQ i sl, ali onda bi se smanjila "preglednost". Imajući tu relaciju sada treba pozajmiti program za pravljenje tranzitivnih putanja, odnosno program[24]:

```
((rr _x _y)  (r () _x _y _Put)  (PP Putanja : _Put))
((rr|_X))

((psi _A _x)  (NOT VAR _x)  (ON _x _A))
((r _A _x _y _B)(NOT sused _x _z) / FAIL)
((r _A _x _y _B)(NOT sused _z _y)  /  FAIL)
((r _A _x _y (_y _x|_A))  (sused _x _y))
((r _A _x _z _B)  (NOT psi _A _x)  (sused _x _y)  (r (_x|_A) _y _z _B))
```

i postaviti pitanje ?((rr (1 1 1 1) (0 0 0 0))). Na ekranu će se pojaviti

[24] Pazite, programom se traži samo jedna putanja.


```
((0 0 0 0) (1 0 1 1) (0 0 1 0) (1 1 1 0) (0 1 0 0) (1 1 1 1))
```

što opisuje jedan način rešavanja postavljenog zadatka.

Zadatak 6.16. Neka su, recimo, date ovakve dve liste brojeva

```
[a,b],   [p,q,r,s]
```

gde su a,b,...,r,s zadani brojevi. Kako proloski ostvariti ovaj algoritam množenja tih lista:

> Prvo a, tj. prvi član prve liste pomnožimo redom sa p,q,r,s i tako dobijemo proizvode a*p, a*q, a*r, a*s, a potom b takode redom pomnožimo sa p,q,r,s i dobijemo proizvode b*p, b*q, b*r, b*s. Kao plod algoritma treba da bude lista tih proizvoda.

Kao što se vidi traži se algoritam sa redosledom koji po svojoj prirodi liči na uobičajene "navišne" algoritme u BASICu, FORTRANu i sl. Inače, odgovarajući proloski program treba da se odnosi na ma koje dve liste brojeva.

Rešenje. Jedno rešenje je dato ovim programom

```
proiz(_a,[_b|_c],_Rez1,_Rez)
                          :- _cc is _a*_b, proiz(_a,_c,[_cc|_Rez1],_Rez).
proiz(_a,[],_Rez,_Rez).

pro([_a|_b],_p,_Stog,_Rez)
                    :- proiz(_a,_p,_Stog,_Stog1), pro(_b,_p,_Stog1,_Rez).
pro([],_a,_Rez,_Rez).
```

U tom programu se relacijom proiz obavlja množenje jednog člana redom sa članovima jedne liste. Recimo, pri računu formule proiz(2,[10,20,30],[],Rez) za Rez će se dobiti lista [60,40,20]. Rezultat se postupno pravi od Rez1, koji je na početku prazna lista. Usled toga redosled članova rezultata je kao što je naveden ("preokrenut"). Glavna relacija programa je pro, kojom se uz uslugu prethodne relacije obavlja naloženi zadatak. Recimo, pri računu formule pro([1,2,3],[10,20],[],Rez) Rez će biti [60,30,40,20,20,10].

Zadatak 6.17. Jedan od problema u vezi sa listama je njihovo "peglanje". Recimo, peglanjem liste (a b (c d) ((e f) g) 55) nastaje lista (a b c d e f g 55). Slobodnije rečeno unutar date liste "obrisali" smo sve znakove zagrada), (. Napraviti proloski program za peglanje date liste, datog |-terma.

Rešenje. Jasno je da ako treba da peglamo neki |-term oblika (_A|_B), onda treba prvo ispeglati _A, i recimo tako dobiti _A1, dalje ispeglati _B i tako dobiti _B1 i onda

```
(Δ)          na _A1 dopisati _B1
```

I tu je sada glavno gubljenje vremena, jer takav korak (Δ) se kod složenijeg |-izraza pojavljuje više puta. Da bi se to izbeglo, slično kao u slučaju gramatika (videti Zadatak 6.1), upotrebićemo zamisao diferencnih (različnih) lista. Grublje rečeno kao rezultat peglanja neke liste L uzimamo "razliku" P - Q, odnosno u stvari ma koji izraz poput f(P,Q) koji ima dva sastavka P,Q. Rešenje koje navodimo je u Micro-prologu pa sledstveno takav izraz ćemo pisati ovako (f P Q). Uz takvo pisanje diferencnih lista imamo ovaj program:

```
   ((peg _x _y)(peglaj _x (f _y ()) ))
(Δ1)    ((peglaj [_x|_y)  (f _A _C))
            (peglaj _x (f _A _B)) (peglaj _y (f _B _C)))
(Δ2)    ((peglaj _x (f (_x|_y) _y)) (konst _x))
```

```
(Δ3)     ((peglaj () (f _x _x)))
    ((konst _x)(CON _x))
    ((konst _x)(NUM _x))
```

Osnovni predikat je peg. Recimo, na pitanje

```
?((peg (a (b c) ((d))) _X)(PP _X))
```

za _X će se dobiti (a b c d). Inače, predikat konst je uveden sa smislom "biti konstanta", kao broj ili konstatska reč.
Buduci da je ideja diferncnih("različnih") lista veoma bitna, na jednom primerku ćemo postupno videti kako ona deluje. Pošto se neće pojaviti potreba za procedurom vraćanje to ,bez gubitka ikakvih detalja , upotrebićemo jednačinsko pisanje. Tako, "ispeglajmo" listu ((a b) c), odnosno potražimo _y za koje važi (peg ((a b) c) _y). Imamo:

```
(peg ((a b) c) _y) =(peglaj ((a b) c) (f _y ()))
                   =(peglaj ((a b)|(c)) (f _y ()))
                     Jer ((a b) c)=((a b)|(c))
                   =(peglaj (a b) (f _y _z)) (peglaj (c) (f _z ()))
                     Koristeći članak (Δ1). Smisao: da nađemo odvojeno
                     "peglaj" od (a b) i od (c). Sada će nastati putova-
                     nje dok ne nađemo "peglaj" od (a b).
                   =(peglaj (a|(b)) (f _y _z))(peglaj (c) (f _z ()))
                   =(peglaj a (f _y _z1))(peglaj (b) (f _z1 _z))
                                                (peglaj (c) (f _z ()))
                     Po (Δ1)
                   =(peglaj (b) (f _z1 _z))(peglaj (c) (f _z ()))
                    Koristeći (Δ2) prva gornja formula, tj. prvi sasta-
                    vak konjunkcije je dokazan i to je "plaćeno" zamenom
                    _y-->(a|_z1).
                    Možemo reći da je _y određeno do na _z1.
                   =(peglaj (b|()) (f _z1 _z))(peglaj (c) (f _z ()))
                   =(peglaj b (f _z1 _z2))(peglaj () (f _z2 _z))
                                             (peglaj (c) (f _z ()))
                     Po (Δ1)
                   =(peglaj () (f _z2 _z)) (peglaj (c) (f _z ()))
                     Prva gornja formula je dokazana primenom (Δ2) uz
                     zamenu
                          _z1-->(b|_z2)
                     Sada je _z1 određeno do na _z2.
                   =(peglaj (c) (f _z ()))
                     Prva formula je dokazana primenom (Δ3) uz zamenu
                         _z2 --> _z
                     pa nas čeka još traženje _z.
                   =(peglaj (c|()) (f _z ()))
                   =(peglaj c (f _z _z3))(peglaj () (f _z3 ()))
                   =(peglaj _z3 ())
                     Prva formula je dokazana primenom (Δ2) uz zamenu
                        _z-->(c|_z3)
                    pa nas sada čeka traženje _z3.
                   = DA
                   Dokaz je završen primenom (Δ3), što je dalo zamenu
                        _z3-->()
```

Kao što se vidi tokom algoritma su se pojavile ove zamene


```
_y-->(a|_z1),   _z1-->(b|_z2),   _z2-->_z, _z-->(c|_z3), _z3-->()
```

što odmah daje _y-->(a|(b|(c|()))), tj. _y-->(a b c).

Jedna od suštinskih stvari prisutnih u algoritmu da blagodareći upotrebi diferencnih listi je u potpunosti izbegnut ikakakav korak tipa (Δ), a umesto toga su došla navedena zamenjivanja.

Zadatak 6.18. Neka je data lista L= [X,Y,Z], čiji članovi su promenljive X,Y,Z. Za ispitivanje "članosti" uopšte dobro je poznat ovaj program

```
elem(P,[P|Q]).
elem(P,[Q|R]) :- elem(P,R).
```

Recimo, pomoću njega možemo raspraviti da li je 2 član liste [4,5,2,3]i sl. ali pomoću njega ne možemo saznati da su članovi liste L upravo promenljive X,Y,Z. Naime, na svako od pitanja ?-elem(U,[X,Y,Z]), ?-elem(X,[X,Y,Z]) će se dobiti odgovor da. Glavni "krivac" je unifikacija koju Prolog u osnovi koristi. Da li se ipak može relacija elem prepraviti i osposobiti i za takva pitanja sa bitnim učešćem promenljivih ?
Rešenje. Jedno rešenje se može napraviti uslugom predikata == i glasi

```
elem(X,[Y|Z]):-X==Y.
elem(X,[Y|Z]):-elem(X,Z).
```

U Micro-prologu nema predikata ==, ali se on može u njemu definisati na način koji navodimo:

```
((jed _x _y) (GRNHOL _Z (_x _y))(EQ _Z (_P _P)))
```

Tu se koristi Micro-prološki predikat GRNHOL, kojim se u tom slučaju od liste (_x _y) najpe napravi _Z, to je "grnhol" od te liste, i onda u narednom koraku se proverava da li je taj _Z oblika (_P _P) za neki _P. Evo i reč više:

GRNHOL se pojavljuje u obliku (GRNHOL Var Lista), gde Var je neka promenljiva, a Lista je lista, ili uopšte neki |-term, sa bar jednim učešćem znaka |. Tako ispravna je svaka od formula

```
(*)   (GRNHOL _X (_x _y)),  (GRNHOL _Y (a|u))
      (GRNHOL _Z ((a _x _y)(b _x)(c _y)))
```

i smisao je: vrednost za Var je |-term koji nastaje kad se u |-termu njegove promenljive redom zamene ovim "skrivenim" objektima[1] _A, _B[2] _C, ... Tako, u gornjim primerima (*) _X, _Y, _Z imaju redom ove vrednosti

```
(_A _B),   (a|_A)   ((a _A _B)(b _A)(c _B))
```

---

[1] Taj spisak objekata je ugrađen u Micro-prolog i korisnik ga može saznati. Recimo, ako želite spisak od prvih deset možete postaviti ovakvo pitanje

```
?((GRNHOL _Var (_x1 _x2 _x3 _x4 _x5 _x6 _x7 _x8 _x9 _x10))(PP _Var))
```

i na ekranu dobiti taj spisak.

[2] Pomenimo da se GRNHOL koristi i u obliku sa 3 argumenta, kao

```
(GRNHOL Var Lista Listadvojaka)
```

koja sadrži i treći argument Listadvojaka. Pretpostavimo, na primer, da data Lista ima upravo tri promenljive, koje su _x, _y, _z. Tada taj treći objekat izgleda

```
((_A|Adresa1) (_B|_Adresa2) (_C|Adresa3))
```

gde su Adresa1, Adresa2, Adresa3 adrese promenljivih _x, _y, _z.

U vezi sa tim objektima istaknimo sledece: Oni su u stvari konstante.

Recimo shodno tome, ako je, kao gore _X dobilo vrednost (_A _B) onda pri računu formule (PP _X) umesto adrese _X štampa se (_A _B). Dalje, ti _A _B su medusobno različite konstante, pa u tom trenutku formula (EQ _A _B) je netačna. Upravo to vladanje "skrivenih" proloških konstanata _A, _B, _C, ... je iskorišćeno u gornjoj definiciji relacije jed.
Evo kako se može utvrditi da su _A, _B, .. zaista konstante u Micro-prologu. Postavimo ovo pitanje ?((GRNHOL _S (_x))(EQ _S (_u))(VAR _u)) i dobićemo negativan odgovor. Evo obrazloženja:
Prvo će _S dobiti vrednost (_A), a zatim u drugom koraku će _u dobiti vrednost _A, i kako _A nije promenljiva[3] to vrednost od (VAR _u) biće ne

Zadatak 6.19. Pretpostavimo da u Micro-prologu nema predikata GRNHOL, kakav je recimo slučaj sa Arity prologom. Da li se naknadno može uvesti takav predikat ?

Rešenje. Evo jednog rešenja u Micro-prologu; uvedeni predikat je nazvan grnhol:

```
((lista (a b c d e f g h i j k l m n)))
((promkon _p _x)(VAR _x) (lista1 (_p|_q))
                         (EQ _x _p)(KILL lista1) (ADDCL ((lista1 _q))))
((promkon _x _x)(NUM _x))
((promkon _x _x)(CON _x))
((promkon (_xx|_yy) (_x|_y))(promkon _xx _x) (promkon _yy _y))
((promkon () ()))
((grnhol _x _y)(lista _Lis)(ADDCL ((lista1 _Lis)))
                                        (promkon _x _y)(KILL lista1))
```

Kao "skrivene" konstante tu su korišćene a, b, c, d, e, f, g, h, i, j, k, l, m, n; ukupno njih 15. Recimo, na pitanje ?((grnhol _Rez (_x _y _x))(PP _Rez)) na ekranu će se pojaviti lista (a b a).

Napomena 6.1. U vezi sa GRNHOL predikatom i u njemu korišćenih konstanti _A, _B, _C, ... ima još nekih donekle čudnih pojedinosti. Naime, pretpostavimo da se u Micro-prologu postavi ovo pitanje:

```
?((GRNHOL _Z ((a _x)(PP _x)))(ADDCL _Z)))
```

sa smislom:

_Z je ((a _A)(PP _A)); dodaj članak ((a _A)(PP _A))

Međutim, ako iza toga postavimo pitanje ?((a 45)) na ekranu će se pojaviti 45, tj. ispada da u dodatom članku _A je promenljiva ! Proverite to direktno. Razlog ? Prosto to je odlika Micro-prologa; tako je napravljen. Može se ovako reći:

Ako se u ADDCL-formuli kao argumenti koriste reči koje počinju podcrticom tada ih Micro-prolog algoritam na tom mestu smatra promenljivim.

Da je to i bukvalno tačno uverite se ovako:
Postavite pitanje ?((ADDCL ((f "_Pera")(PP "_Pera")))) i videćete da će se dodati članak oblika ((f _X)(PP _X))

---

[3] To je osetljivo mesto. Recimo, na pitanje oblika
?((EQ _x _y)(VAR _x))
odgovor je potvrdan jer _x je "vezano" za _y koja je promenljiva.

Kraj Napomene 6.1.

Zadatak 6.20. Da li se proloski može ostvariti ova zamisao:

Dodati članke: dat(_a1), dat(_a2), ..., dat(_ak), gde k može biti ma koji, ali zadat tek na početku programa, tj. k je input-promenljiva.

Glavni problem su tri tačkice ..., jer broj k nije unapred zadat.
Rešenje. Navodimo rešenje u LPA-prologu:

```
ajde:-write('Koliko '),read(K),K1 is X+1, dodaj(1,K1).
dodaj(N,N).
dodaj(I,N):- J is I+48,name(X,[95,97,J]),
                                assert(dat(X)), I1 is I+1, dodaj(I1,N).
```

Program se pokreće pitanjem ?-ajde i onda se da vrednost za K. Pomoću dodaj -članaka se "obavi čitav posao". Tako, prvo se koristi drugi dodaj-članak i I ima vrednost 1. J dobije vrednost 49, a X se računa preko formule

```
name(X,[95,97,49])
```

tj. X je reč od tri slova, čiji ASCII-kodovi su 95, 97, 49, tj. X je reč _a1. I slično dalje X postupno dobije vrednosti _a2, ... U svakom od tih trenutaka se pomoću assert-predikata dodaje članak dat(X). I to je sada bitno:
Pošto X kao vrednost ima reč počinjuću sa _ , to pri dodavanju tog članka X se smatra kao promenljiva.
To je potpuno u skladu sa onim rečenim na kraju prethodne Napomene 6.1. Ali, istaknimo to, takva stvar ne važi za Arity-prolog, pa u tom Prologu prethodni program ne bi dao zamišljeni "plod".

Napomena 6.2. U LPA- prologu uslugom name predikata se mogu graditi reči koje počinju podcrticom _ , a koje onda pri korišćenju assert-predikata se shvataju kao promenljive. Recimo, radi dodavanja članka dat(_a3) možemo umesto assert(dat(_a3)) koristiti ovo: name(X,[95,97,51]),assert(dat(X)). Takva zamisao nije ostvariva u Arity prologu, a takode ni u Micro-prologu, jer u njemu umesto name predikata imamo STRINGOF predikata, ali koji je od njega slabiji, pa ne dopušta reči počinjuće podcrticom.

Zadatak 6.21. (U vezi sa Zadatkom 3.37). Da li je proloski ostvariva ova zamisao:
Neka su date neke konstante kao a, b, c, koje zapamtimo uslugom jednog pomoćnog predikata dat, odnosno prihvatanjem ova tri članka:

```
dat(a).   dat(b).   dat(c).
```

Da bismo od a, b, c napravili sve reči dužine k, gde je k ulazna (input) promenljiva, hoćemo Prologu da postavimo ovo "pitanje"

```
?-dat(X1),dat(X2),...,dat(Xk),write(X1),write(X2),...,write(Xk),nl,fail.
```

Ali, problem je u tome što to pitanje zbog ne-datosti k sadrži tri tačkice ..., tj. uopšte nije pravo prološko pitanje.
Da li se to ipak nekako može ostvariti ?

Rešenje. Navodimo jedno rešenje u LPA- prologu, koji inače za rad sa promenljivim uopšte ima dva dodatna predikata tohollow, toground, njih bliže opisujemo u Napomeni 6.3. To rešenje glasi :

```
ajde:-write('Koliko '),read(X),X1 is X+1, pravi(1,X1,[nl,fail],Rez),
            razdvoj(Rez,Prvi,Drugi), dodaj(Prvi,Drugi,Plod),!,radi(Plod).
pravi(N,N,Lis,Lis).
pravi(I,N,Lis,Rez):-J is I+48,name(X,[97,J]),
```

```
  tohollow([dat(X),write(X)],XX,[X]), I1 is I+1, pravi(I1,N,[XX|Lis],Rez).
radi([X|Y]):-call(X),radi(Y).
radi([]).
razdvoj([[A1,B1],nl,fail],[A1],[B1,nl,fail]).
razdvoj([[A1,B1]|X],[A1|Y1],[B1|Y2]):-razdvoj(X,Y1,Y2).
dodaj([],X,X).
dodaj([A|B],C,[A|C1]):-dodaj(B,C,C1).
        dat(a).
        dat(b).
        dat(c).
```

Evo prvo kako taj program radi. Pokreće se pitanjem ?-ajde. i na "upit" 'Koliko' da se željena dužina reči. Dalje, program obavi naloženi zadatak, odnosno na ekranu štampa sve k-reči od slova a,b,c. Dajemo još neke pojedinosti iz toka algoritma:

Prvo ističemo da se uslugom predikata name najpre[4] pravi k reči a1,a2, ...,ak. One su redom vrednosti promenljive X koje se dobijaju pri računu ovih formula: name(X,[97,49]) (To X je a1), name(X,[97,50]) (To X je a2) i sl. dalje. Dalje nam je zamisao da te napravljene reči a1,...,ak u idućem koraku "upromenljivimo". Tu se sada koristi predikat tohollow. On se koristi u obliku:

tohollow(Dat_izraz,Rezultat,Dat_spisak_imena_promenljivih)

Recimo, pri računu formule: tohollow([a,f(b,c),d],Rez,[a,c]) promenljivoj Rez će se dodeliti vrednost [Var1,f(b,Var2),d], gde su Var1,Var2 dve nove različite promenljive. Drugim rečima: u datom izrazu smo "upromenljivili" željene konstante a,c, koje smo u tu svrhu stavili kao treći argument za tohollow.
U našem programu se pojavljuju ovakve tohollow-formule

tohollow([dat(a1),write(a1)],XX,[a1])
tohollow([dat(a2),write(a2)],XX,[a2]) i sl. dalje.

One se pojavljuju u okviru predikata pravi, koji u stvari "navišno" napravi ovakvu listu:

[[dat(X1),write(X1)],...,[dat(Xk),write(Xk)],[nl,fail]]

gde su X1,...,Xk naravno neke međusobno različite promenljive nastale uslugom tohollow-predikata. Dobijena lista podseća na traženi oblik pitanja, jedino redosled članova nije odgovarajući. U tu svrhu se upošljavaju dosta jednostavni i očigledni razdvoj i dodaj članci. Njihovom uslugom se od prethodne liste napravi ova lista Plod:

[dat(X1),...,dat(Xk),write(X1),...,write(Xk),nl,fail]

I sada jedino preostaje da Prologu "kažemo" : u r a d i tu listu kao da to je pitanje. To nam polazi za rukom uslugom predikata radi, koji, kao što se vidi, ima samo ova dva članka:

```
radi([X|Y]):-call(X),radi(Y).
radi([]).
```

Na kraju, pošto Plod ima na kraju fail u ajde članku ispred radi(Plod) stoji znak reza !, da bi sprečio nelogično vraćanje nazad.

[4] Taj deo je sličan sa onim prisutnim u prethodnom zadatku.

Napomena 6.3. LPA-prolog ima dva zanimljiva i značajna predikata tohollow i toground kojima se slobodnije rečeno mogu obaviti ovakvi poslovi:

U datom izrazu(listi,formuli, članku) date konstantske reči "upromenljiviti" -tako što radi tohollow , odnosno u datom izrazu date promenljive "ukonstantiti", za šta je "zadužen" toground pedikat. Bliže:

(i) Pri računu formule oblika

tohollow(Dat_izraz,Rez,Data_lista_imena_promenljivih)

promenljivoj Rez se daje vrednost koja je izraz nastao iz Dat-izraz[5], kad se u njemu konstantske reči iz spiska Data_lista_imena_promenljivih preobrate u nove promenljive.
Predikat tohollow se može pojaviti i u ovom obliku sa 4 argumenta:

tohollow(Dat,Rez,Imena,Nove_promenljive)

čiji je smisao sličan prethodnom, ali dodatno argument Nove_promenljive ima vrednost jednaku listi novo-uvedenih promenljivih.

(ii) Pri računu formule oblika

toground(Dat_izraz,Rez)

promenljivoj Rez se pridružuje bez-promenljivski izraz nastao iz Dat_izraz kad mu se promenljive zamene konstantskim imenima posebnog oblika kao[6] '_A','_B' i sl.
Recimo,pri računu

toground(f(X,Y,6),Rez)

Rez će dobiti ovu vrednost f('_A','_B',6).
Predikat toground može imati 3,4 i 5 argumenata. Tako,pri računu formule

toground(Dat_izraz,Rez,Imena_prom)

Rez dobije vrednost kao gore, dok promenljiva Imena_prom dobije vrednost jednaku listi novouvedenih imena promenljivih. Recimo,u slučaju formule toground(f(X,Y,X),Rez,Ime), imaćemo

Rez je f('_A','_B','_A'), a Ime je ['_A','_B'].

Dalje, oblik sa 4 argumenta izgleda

toground(Dat_izraz,Rez,Spisak_prom,Imena_prom)

To je slično gornjem s tim što je sada promenljiva Spisak_prom "zadužena" da zapamti sve promenljive polaznog izraza Dat_izraz. Recimo, pri računu formule

toground(f(X,Y,X),Rez,Spisak,Imena)

imamo ova "vrednovanja"

Rez-->f('_A','_B','_A'),

[5] Taj izraz ne sme da sadrži promenljive.

[6] Obavezno se javlja i podcrta.

```
Spisak[7]--> [X,Y],
Imena-->['_A','_B']
```

Znači, Spisak zapamćuje -i to po redu učestvovanja - sve promenljive polaznog izraza, što itekako može biti korisno.
Rekosmo da za toground predikat postoji i peto-argumentni oblik, kratko opisan u zvaničnom Priručniku za LPA-prolog[8]. Tu je navedeno i nekoliko primera (str. 140,141). Ali, nijedan od njih neće da radi. U vezi sa promenljivim u LPA-prologu ima još jedan značajan predikat varsin. On se koristi u obliku

varsin(Dat_izraz,Spisak_prom)

gde Dat_izraz je neki dat izraz, slično kao gore, a Spisak_prom je lista njegovih promenljivih po redu učestvovanja. Recimo, pri računanju formule

varsin([23,X,Y,Y],S)

S će dobiti vrednost listu [X,Y], s tim što na ekranu će se umesto X, Y štampati njihove adrese.

Završetak Napomene 6.3.

Zadatak 6.22. Da li se prethodni zadatak može rešiti i bez usluge tohollow predikata ?

Rešenje. Može, korišćenjem ideje Zadatka 6.19 i Napomene 6.2. Naime, dosta je da se u programu navedenom u rešenju prethodnog zadatka ajde- i pravi- članci zamene sledećim:

```
ajde:-write('Koliko '),read(X),X1 is X+1,
                    pravi(1,X1,[nl,fail],Rez),
                    razdvoj(Rez,Prvi,Drugi),
                    dodaj(Prvi,Drugi,Plod),
                    assert(rel(Plod)),
                    rel(XXX),retract(rel(Plod)),
                     !,radi(XXX).
pravi(N,N,Lis,Lis).
pravi(I,N,Lis,Rez):-
            J is I+48,name(X,[95,97,J]), XX=[dat(X),write(X)],
            I1 is I+1, pravi(I1,N,[XX|Lis],Rez).
```

Objašnjenje:
U pravi-člancima se pojavljuje formula name (X,[95,97,J]) pri čijem računanju se X-u dodeljuju reči _a1, _a2,....
U ajde-članku, odnosno[9] u delu assert(rel(Plod)) se postiže da te reči "postanu" promenljive. U stvari, to se upravo desilo nakon pozivanja formule rel(XXX).

Kao što vidite, sa tim malim dosetkama dobija se rešenje koje je prostije od onog sa upotrebom tohollow-predikata.

[7] Pri štampanju na ekran se umesto tih X,Y pojavljuju njihove (memorijske) adrese.

[8] Tačan naslov je LPA PROLOG Professional Compiler, Version 2.0, Logic Programming Associates Ltd. 1988.

[9] Tu je rel samo privremena, odnosno uslužna relacija.

7. HORNOVSKE FORMULE ; DEDUKTIVNI MODELI

7.1 U v o d i z i s k a z n e l o g i k e

Neka je P neki skup reči, koje su dogovorno nazvane iskazna slova. Dalje, neka su $\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg$ oznake uobičajenih osnovnih logičkih operacija. Označimo sa For(P) skup svih odgovarajućih iskaznih formula. Recimo, ako p, q, r∈P, onda reči(zapisi): p , q , r , (p∧q) , ¬r , ((p∧q)⇒ r) pripadaju skupu For(P). Stroga definicija članova skupa For(P) je po svom sklopu slična sa definicijom *-termova izloženoj u Zadatku 3.33 i glasi

Definicija 7.1.1
(i) Iskazna slova su iskazne formule[10].
(ii) Ako su A,B iskazne formule, onda i ove reči
(A∧B) (A∨B) (A⇒B) (A⇔B) ¬A
su iskazne formule.
(iii) Iskazne formule su samo one reči koje se mogu dobiti konačnom primenom pravila (i) (ii) ove definicije.

Pretpostavljamo da su sa znacima[11] ⊥ (ne-te), ⊤ (te) definisane istinitosne tablice, koje redom navodimo:

p	q	p ∧ q	p ∨ q	p ⇒ q	p ⇔ q
⊤	⊤	⊤	⊤	⊤	⊤
⊤	⊥	⊥	⊤	⊥	⊥
⊥	⊤	⊥	⊤	⊤	⊥
⊥	⊥	⊥	⊥	⊤	⊤

p	¬ p
⊤	⊥
⊥	⊤

Vrednost ma kog iskaznog slova je svako preslikavanje v:P⟶{⊤,⊥}. Recimo, ako P={p,q,r}, onda preslikavanje

$$v=\begin{pmatrix} p & q & r \\ \bot & \top & \top \end{pmatrix}$$

je primer vrednosti slova. Neka je f(p1,...,pn) iskazna formula čija sva iskazna slova su među p1,...,pn. Tada svakoj vrednosti v

$$v = \begin{pmatrix} p1 \ldots pn \\ \tau 1 \ldots \tau n \end{pmatrix} \qquad (\tau i \in \{\top, \bot\})$$

tih slova jednoznačno odgovara ⊤ ili ⊥, odnosno vrednost formule, u oznaci v(f(p1...,pn)). Ta vrednost, koja je pojmovno samo jedan slučaj vrednosti termova uopšte, može se ovako rekurzivno definisati:

(7.1.1) (i) v(pi) = τi (gde 1 ≤ i ≤n)
(ii) v(A∧B) = v(A)∧v(B),
v(A∨B) = v(A)∨v(B),
v(A⇒B) = v(A)⇒v(B),
v(A⇔B)= v(A)⇔v(B).

Može se za neku iskaznu formulu A dogoditi da ima vrednost ⊤ za sve moguće

[10] Umesto članovi skupa P, članovi skupa F(P) rekli smo iskazna slova, odnosno iskazne formule.

[11] Pretpostavljamo da znaci ⊤,⊥ nisu članovi skupa P.

vrednosti njenih slova. Takvu formulu nazivamo tautologija. Recimo, tautologije su ove formule[12]

$$p \vee \neg p \ , \quad p \wedge q \Leftrightarrow q \wedge p \ , \quad \neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q \qquad (p,q \text{ su iskazna slova})$$

Tautologijama se, pored ostalog, izražavaju razni logički zakoni rasuđivanja.

Neka je, sada, $F \subseteq For(P)$ skup nekih iskaznih formula i $v : P \to \{\top, \bot\}$ vrednost slova. Za v kažemo da je model skupa formula F ukoliko jednakost $v(A)=\top$ važi za sve $A \in F$, tj. sve formule iz F su *tačne* pri toj vrednosti v. Sa $mod(F)$ označimo skup svih modela za F. Može se dogoditi da je to prazan skup, kada kažemo da je F (semantički) protivrečan. Znači, umesto

F nema nijedan model rekli smo i semantički protivrečan[13].

Neka je $A \in For(P)$ ma koja iskazna formula. Oznake

(7.1.2) $\qquad F \models A \quad , \quad F \vdash A$

se redom čitaju ovako

A je semantička posledica iz F, A je sintaktička posledica iz F.

Pri tome, $F \models A$ znači da je formula A tačna kad god su sve formule iz F tačne, tj. ako $v \in mod(F)$, onda $v(A)=\top$. Recimo, očigledno su tačni ovi semantički sledovi[14] $p,q \models p$; $p, p \Rightarrow q \models q$; $p, \neg q \models \neg(p \Rightarrow q)$. Nešto je složenije objasniti značenje sleda $F \vdash A$, koji se čita i ovako:

A je dokazivo na osnovu(pretpostavki) F.

To znači da postoji konačan niz formula:

(7.1.3) $\qquad F1$

$F2$

$\vdots$

Fk

gde Fk je upravo A i, pri čemu, svaki član Fi ($1 \le i \le k$) tog niza zadovoljava uslov:

1) Fi je neki član iz F; ili
2) Fi je neka tautologija; ili
3) Fi sledi iz neke dve prethodne formule istog niza po pravilu modus ponens, opisivom ovako:

$$\frac{A, A \Rightarrow B}{B}$$ (rečima: B je dokazivo pomoću A i $A \Rightarrow B$)

Inače, niz oblika (7.1.3) se naziva dokazom, tačnije dokazom formule A na osnovu pretpostavki F. Navedimo neke primere. Recimo, važi sled

$$p, p \Rightarrow q, q \Rightarrow r \vdash r$$

Zaista, jedan dokaz oblika (7.1.3) glasi

(1) p (Pretpostavka)

[12] Iz izgleda tih formula vidi se da su neke zagrade izostavljene.

[13] Naravno, ako F ima bar jedan model onda kažemo da je on (semantički) neprotivurečan.

[14] Svaki od zapisa oblika $F \models A$, $F \vdash A$ zvaćemo i sled.

(2) $p \Rightarrow q$ (Pretpostavka)
(3) q (Iz (1) i (2) primenom modus ponens-a)
(4) $q \Rightarrow r$ (Pretpostavka)
(5) r (Iz (3),(4) po modus ponens-u)

Dalje, važi sled $p, \neg q \vdash \neg(p \Rightarrow q)$ što se vidi iz ovog niza, odnosno dokaza

(1) p (Pretpostavka)
(2) $\neg q$ (Pretpostavka)
(3) $p \Rightarrow (\neg q \Rightarrow \neg(p \Rightarrow q))$ (Tautologija)
(4) $\neg q \Rightarrow \neg(p \Rightarrow q)$ (Iz (1),(3) po modus ponens-u)
(5) $\neg(p \Rightarrow q)$ (Iz (2),(4) po modus ponens-u)

Ključni deo tog dokaza je korak (3) u kome se koristi činjenica da je formula $p \Rightarrow (\neg q \Rightarrow \neg(p \Rightarrow q))$ tautologija, što samo za sebe se neposredno proverava. Međutim, odakle ideja da se koristi baš ta formula ? Ideja se može dobiti ako se na umu ima sledeća opšta činjenica:

(7.1.4) $F1, F2, \ldots, Fn \vdash A \quad \longleftrightarrow \quad F1 \Rightarrow (F2 \Rightarrow \ldots \Rightarrow (Fn \Rightarrow A) \ldots)$ je tautologija

gde su $F1, F2, \ldots, Fn, A$ ma koje formule. Zaista, $\longleftarrow$ deo neposredno sledi ovako. Ako je tautologija formula $F1 \Rightarrow (F2 \Rightarrow \ldots \Rightarrow (Fn \Rightarrow A) \ldots)$ onda korišćenjem pretpostavki $F1, F2, \ldots, Fn$ i primenom modus ponens-a najpre sledi formula $(F2 \Rightarrow \ldots \Rightarrow (Fn \Rightarrow A) \ldots)$, a ponovim korišćenjem modus ponens-a sledi $(F3 \Rightarrow \ldots \Rightarrow (Fn \Rightarrow A) \ldots)$ i takom redom dođe se do $Fn \Rightarrow A$ i u poslednjem koraku iz formula $Fn, Fn \Rightarrow A$ izvodimo A. Treba još da dokažemo $\longrightarrow$ -deo tvrđenja, tj. ako $F1, F2, \ldots, Fn \vdash A$ da onda formula $F1 \Rightarrow (F2 \Rightarrow \ldots \Rightarrow (Fn \Rightarrow A) \ldots)$ mora biti tautologija. Zaista, neka je v ma koja vrednost iskaznih slova. Razlikujemo dva slučaja

(i) Bar jedna od formula $F1, F2, \ldots, Fn$ je netačna
(ii) Sve $F1, F2, \ldots, Fn$ su tačne.

U slučaju *(i)*, na osnovu $\Rightarrow$ - tablice, neposredno se vidi da važi jednakost

(*) $\qquad v(F1 \Rightarrow (F2 \Rightarrow \ldots \Rightarrow (Fn \Rightarrow A) \ldots)) = \top$

U slučaju *(ii)* v je model svih formula $F1, F2, \ldots, Fn$. Pošto važi $F1, F2, \ldots, Fn \vdash A$ to A mora biti tačno[15] u svakom modelu skupa formula $\{F1, F2, , Fn\}$. Sledstveno $v(A)=\top$. Na osnovu toga opet dolazimo do jednakosti (*) čime se završava dokaz ekvivalencije (7.1.4).

Na osnovu dokazane ekvivalencije (7.1.4) ispada da je pojam $F \vdash A$, bar u slučaju konačnih skupova, skoro trivijalan. Trivijalnost se pojavljuje upravo iz razloga što smo u definiciji (7.1.3) dokaza, u njegovom delu 2) dopustili da član dokaza može biti ma koja tautologija. S tim u vezi ističemo, a to je bitno, u Matematičkoj logici se po pravilu ne dopušta toliko "široki" uslov 2) već se umesto njega uzima da članovi dokaza mogu biti samo neke tautologije (koje se onda obično nazivaju logičke aksiome) . Naravno, tada se ne mogu na trivijalan način graditi dokazi sledova oblika $F1, F2, \ldots, Fn \vdash A$. Međutim, a to je bitno, navedena definicija (7.1.3) je logički ispravna i potpuno opšta ; u daljem izlaganju ćemo je često koristiti.

[15] To je veoma opšta činjenica i sledi neposredno iz definicije dokaza. Naime, lako se vidi da u zamišljenom dokazu sleda $F1, \ldots, Fn \vdash F$ svaki član dokaza mora biti tačan u pretpostavljenom modelu ν formula $F1, \ldots, Fn$.

7.2 Pojam formalne teorije

Na ovom mestu smo se približili jednom od najvažnijih pojmova Matematičke logike i savremene matematike uopšte- pojmu formalne teorije. Ovlašno opisano tu se radi o aksiomama i teoremama uz potpuno strogo definisan logički put od prvih ka drugim ,odnosno strogo definisan pojam dokaza,koji se inače uvodi slično (7.1.3). Ali, pođimo redom. Neka je A neki alfabet, tj. skup izvesnih znakova[16] i neka je $W(A)$ skup svih reči[17] nastalih iz slova alfabeta primenom operacije dopisivanja (konkatenacije). Tada uopšte neka formalna teorija $\mathcal{T}$ nad tim alfabetom se određuje sa dva skupa:

- skupom $Ax \subseteq W(A)$ nekih reči, tzv. aksioma.
- skupom R izvesnih pravila izvođenja oblika

$$\frac{w_1,\ldots,w_n}{w_{n+1}}$$ (Čitati: Reč w_{n+1} sledi iz reči $w_1,\ldots,w_n$)

Recimo, ako je $A=\{a,b\}$, onda jedna formalna teorija $\mathcal{T}$ je odredena ovako

(7.2.1) $Ax=\{a,b\}$
R ima dva pravila: $\frac{Xa}{Xab}$, $\frac{Xb}{Xba}$
gde X može biti ma koja reč, možda i prazna[18].

Neka je sada $\mathcal{T}(A,Ax,R)$ ma koja formalna teorija. Dokaz u toj teoriji je svaki konačan niz reči

(7.2.2) $w1$
$\vdots$
wn

tako da svaki član wi tog niza zadovoljava uslov:

wi je neka aksioma i l i wi sledi iz nekih prethodnih članova tog niza po izvesnom pravilu izvođenja $\alpha \in R$.

Ako je (7.2.2) dokaz, onda za wn kažemo da je teorema teorije $\mathcal{T}(A,Ax,R)$ i da je (7.2.2) njen dokaz. Obično se rečenica: w je teorema teorije $\mathcal{T}$ zapisuje ovako $\vdash_{\mathcal{T}} w$. Pomenimo još da se uvodi i pojam

(σ) $F \vdash_{\mathcal{T}} w$,

što se čita: w je u teoriji $\mathcal{T}$ posledica hipoteza F. Inače (σ) važi ako postoji niz tipa (7.2.2) čiji svaki član zadovoljava uslov 1°, uslov 2°, ili je član iz F (tj. jednak je nekoj hipotezi).

Uočimo maločas definisanu formalnu teoriju (videti(7.2.1)). U njoj, naprimer imamo $\vdash$ abab. Zaista, niz:

(1) a (Aksioma)
(2) ab (Iz (1) po pravilu $\frac{Xa}{Xab}$)

[16] O alfabetima smo već imali u delu 7.2.

[17] Za alfabet A pretpostavljamo da zadovoljava uslov (7.2.2), tj. uslov o jedinstvenosti slova ma koje reči (nad A).

[18] Tako je u stvari, kraće rečeno da u datim pravilima reči Xa, Xb smeju da budu i a, b redom; kada inače Xab, Xba glase ab, odnosno ba.

(3) aba (Iz (2) po pravilu $\frac{Xb}{Xba}$)

(4) abab (Iz (3) po pravilu $\frac{Xa}{Xab}$)

je jedan njen dokaz. Ta formalna teorija je veoma prosta i lako se dalje vidi da je teorema ma koja reč s1..sn čija svaka dva susedna slova si, si+1 međusobno različita. Takve reči ćemo privremeno zvati *naizmenične*. Primeri takvih reči su:

a, b, ab, ba, aba, bab, abab i sl.

Međutim, zanimljivo je da važi i obrat iskazanog tvrdenja, odnosno imamo ovu ekvivalenciju

(7.2.3) $\vdash_{\mathcal{T}} w \longleftrightarrow$ w je naizmenična reč.

Dokaz. Preostaje dokaz $\longrightarrow$ dela. Znači, polazimo od pretpostavke $\vdash$ w. Reč w može imati razne dokaze. Označimo sa m(w) minimum dužine svih mogućih dokaza za w. Znači, najkraći dokazi za w imaju upravo m(w) članova. Dokaz izvodimo indukcijom po m(w). Ako je m(w)=1, onda je w aksioma i w je ili a ili b, dakle naizmenična reč. Neka je m(w)>1 i uočimo jedan dokaz dužine m(w)

(*) $w1$
$\vdots$
wn (wn je w)

za w. Reč wn ne može biti aksioma jer one imaju jednočlane dokaze. Znači wn nastaje iz nekog prethodnog člana niza (*) po jednom od pravila teorije. Moguća su dva slučaja: 1. wn se završava sa a, 2. wn se završava sa b .

U prvom slučaju u krajnjem koraku, tj. pri izvodenju wn je korišćeno pravilo

(**) $\frac{Xb}{Xba}$

gde wn je Xba, dok Xb je neki wi, za $1 \le i < n$. Budući[19] da Xb, tj. wi je učesnik dokaza (*), to primećujemo da Xb ima za sebe dokaz koji je kraći od m(w), sledstveno m(wi)<m(w). Na osnovu indukcijske hipoteze wi je naizmenična reč završavajuća sa b. Budući da se primenom pravila (**) naizmeničnost ne kvari, zaključujemo da i wn mora biti naizmenična. Završen je dokaz u slučaju 1. Slično se postupa i u slučaju 2. Tako se završava ukupan dokaz.

Na osnovu ekvivalencije (7.2.3) možemo reći da formalna teorija $\mathcal{T}$ ima ovo važno svojstvo

Za svaku reč w se u konačno mnogo koraka može utvrditi da li jeste ili nije teorema teorije $\mathcal{T}$.

Uopšte formalne teorije sa takvim svojstvom se nazivaju (algoritamski) odlučive. Znači, kratko rečeno prethodna formalna teorija je odlučiva.

Na kraju istaknimo da se pri potpuno strogom aksiomatskom zasnivanju raznih pojedinačnih matematičkih teorija (teorija skupova, aritmetika, realni brojevi, itd) u naše vreme po pravilu koriste odgovarajuće formalne teorije. Međutim, veći deo tih teorija je neodlučiv, tj. za njih ne postoje algoritmi raspravljanja pitanja teoremnosti.

[19] Konačno početni komad dokaza (*) koji se završava rečju Xb je sigurno jedan dokaz za Xb.

7.3 Deduktivan model Hornovskih formula (slučaj iskaznih formula)

Neka je $F \subseteq For(P)$ skup nekih iskazanih formula. Deduktivna vrednost skupa F je vrednost δ iskaznih slova uvedena sledećom definicijom

(7.3.1) $\delta(p)=\tau$ ako i samo ako $F \vdash p$

gde je $p \in P$ ma koje slovo[20]. Recimo, ako $F=\{p, p\Rightarrow q\}$ onda δ je $\begin{pmatrix} p & q \\ \tau & \tau \end{pmatrix}$, jer važe sledovi $F \vdash p, F \vdash q$, a ako $F=\{p \vee q\}$, onda δ je $\begin{pmatrix} p & q \\ \bot & \bot \end{pmatrix}$. Naime, tada nije tačan nijedan od sledova $p \vee q \vdash p$, $p \vee q \vdash q$. Zaista, neka važi sled $p \vee q \vdash p$. Tada p mora biti tačno u svakom modelu formule $p \vee q$, pa i u ovom $\begin{pmatrix} p & q \\ \bot & \tau \end{pmatrix}$ što je, očigledno, netačno. Slično ne važi $p \vee q \vdash q$.

Neka opet $F \subseteq For(P)$ ma koji skup formula. Može se dogoditi da deduktivna vrednost δ od F jeste model za F. Tada za δ kažemo da je *deduktivan model* za F. I tako, neki skup F, ima deduktivni model upravo u slučaju ako njegova deduktivna vrednost jeste i njegov model. Prema prethodnom, skup $\{p \vee q\}$ nema deduktivan model, dok skup $\{p, p\Rightarrow q\}$ ga ima.

Sada upoznajemo jednu prilično veliku klasu skupova F formula, koji imaju deduktivne modele. Kao što će se ubrzo uvideti takvi skupovi formula čine osnovu, još bolje rečeno kostur, ma kog prološkog programa. Reč je o tzv. *Hornovskim* formulama. To su formule[21] jednog od oblika

(7.3.2) *(i)* p, *(ii)* $p_1 \wedge p_2 \wedge \ldots \wedge p_k \Rightarrow p_{k+1}$

gde su $p, p_1, \ldots, p_k, p_{k+1}$ ma koja iskazna slova.

Potpuno je očigledno da u ma kom prološkom programu svaki članak je gledljiv kao Hornovska formula. Sada dokazujemo sledeći osnovni stav

Stav 7.3.1. Neka je $H \subseteq For(P)$ skup nekih Hornovskih formula. Tada H ima deduktivan model.

Dokaz. Neka je δ deduktivna vrednost od H. Uočimo ma koju formulu $A \in H$ i dokažimo da $\delta(A)=\tau$. Razlikujemo dva slučaja

1° A je oblika p, p je slovo 2° A je oblika $p_1 \wedge p_2 \wedge \ldots \wedge p_k \Rightarrow p_{k+1}$.

U prvom slučaju zbog $p \in H$ odmah sledi $H \vdash p$, pa sledstveno $\delta(p)=\tau$ tj. $\delta(A)=\tau$. U drugom slučaju, ukoliko bar jedan od $\delta(p_1), \ldots, \delta(p_k)$ je $\bot$, jednakost $\delta(A)=\tau$ trivijalno važi. Dalje, ukoliko $\delta(p_1)=\tau, \ldots, \delta(p_k)=\tau$ prvo zaključujemo sledove $H \vdash p_1, \ldots, H \vdash p_k$, a dalje budući da $p_1 \wedge p_2 \wedge \ldots \wedge p_k \Rightarrow p_{k+1} \in H$, zaključujemo[22] $H \vdash p_{k+1}$, odnosno $\delta(p_{k+1})=\tau$. Stoga imamo: $\delta(A)=\tau$. Kraj dokaza.

U vezi sa deduktivnim modelom δ izvesnog Hornovskog skupa H podvucimo sledeće opšte činjenice

---

[20] Rečima: Slovo p ima vrednost τ ako i samo ako je to slovo dokazivo na osnovu hipoteza F.

[21] U stvari, to su tzv. pozitivne Hornovske formule.

[22] Formula $(p_1 \Rightarrow (p_2 \Rightarrow (\ldots \Rightarrow (p_k \Rightarrow p_{k+1}) \ldots)) \Rightarrow (p_1 \wedge \ldots \wedge p_k \Rightarrow p_{k+1})$ je tautologija.

1° Slovo p ima vrednost τ, upravo ako je p dokazivo u H.
2° Slovo p ima vrednost $\bot$, upravo ako p nije dokazivo iz H.

Logičkim zagledom u dokaz Stava 8.3.1 vidi se da on logički počiva samo na ovim svojstvima sledova:

1° Ako $p \in H$, onda $H \vdash p$ (p je slovo)
2° Ako $p_1 \wedge \ldots \wedge p_k \Rightarrow p_{k+1} \in H$ i ako $H \vdash p_1, \ldots, H \vdash p_k$, onda $H \vdash p_{k+1}$.

Sada ćemo, u suštini koristeći to zapažanje, "iscediti" dokaz Stava 7.3.1 i kao plod dobiti deduktivan model sa što manje "šumova", tj. što manje suvišnosti. Neka je H zadan Hornovski skup formula. Izborom aksioma i pravila, načinom koji opisujemo, skupu H se dodeljuje jedna formalna teorija koju označavamo sa $ft(H)$:

(7.3.3) Ako je $p \in H$ i p je slovo, onda p uzimamo za aksiomu teorije $ft(H)$.

Ako je $p_1 \wedge \ldots \wedge p_k \Rightarrow p_{k+1} \in H$, onda pravilo $\dfrac{p_1, \ldots, p_k}{p_{k+1}}$

prihvatamo kao pravilo teorije $ft(H)$.

Recimo, ako $H=\{s \Rightarrow p, p \wedge r \Rightarrow q, s \wedge t \Rightarrow r, s, s \Rightarrow t\}$ onda $ft(H)$ ima aksiomu s i pravila

$$\frac{p}{s}, \quad \frac{p, r}{q}, \quad \frac{s, t}{r}, \quad \frac{s}{t}$$

U opštem slučaju sledeća veza između H i $ft(H)$ je očigledna

(7.3.4) Ako smo u teoriji $ft(H)$ dokazali teoremu oblika $\vdash p$ onda mora važiti $H \vdash p$, tj. $\delta(p)=\tau$.

Razlog je što se svaki dokaz u $ft(H)$ može lako pretočiti u dokaz sleda tipa $H \vdash p$. Recimo, u prethodnom primeru $ft(H)$ ima teoremu r sa ovim dokazom

(1) s	(Aksioma)
(2) t	(Po pravilu $\frac{s}{t}$)
(3) r	(Po pravilu $\frac{s, t}{r}$)

Odgovarajući, "pretočeni", dokaz za $H \vdash r$ glasi

(1) s	(član iz H)
(2) $s \Rightarrow t$	(član iz H)
(3) t	(Iz (1), (2) po modus ponensu)
(4) $s \Rightarrow (t \Rightarrow (s \wedge t))$	(Tautologija)
(5) $t \Rightarrow (s \wedge t)$	(Iz (1), (4) po modus ponensu)
(6) $s \wedge t$	(Iz (3), (5) po modus ponensu)
(7) $s \wedge t \Rightarrow r$	(član iz H)
(8) r	

Međutim, zanimljivo je da li važi obrat tvrđenja (7.3.4). Da bismo to raspravili najpre uvedimo sledeću vrednost $\delta 1$: $\delta 1(p)=\tau \longleftrightarrow \vdash_{ft(H)} p$.

Tvrdimo da je $\delta 1$ jedan model skupa H. Zaista, neka je h ma koji elemenat skupa H. Razlikujemo dva slučaja:

1° h je slovo 2° h je oblika $p_1 \wedge \ldots \wedge p_k \Rightarrow p_{k+1}$

U slučaju 1°, zbog $h \in H$ reč h je aksioma teorije $ft(H)$, sledstveno važi $\vdash h$, tj. $\delta 1(h)=\tau$, pa je dokaz završen u slučaju 1°. U slučaju 2° razlikujemo dva podslučaja

(i) Bar jedan od $\delta 1(p_1),\ldots,\delta 1(p_k)$ je $\bot$, *(ii)* Svi $\delta 1(p_1),\ldots,\delta 1(p_k)$ su $\top$.

Ako *(i)*, onda na osnovu $\Rightarrow$ - tablice važi jednakost $\delta 1(h)=\top$. Ako *(ii)*, onda u ft(H) važe sledovi $\vdash p_1, \ldots, \vdash p_k$. Kako $h\in H$ to u ft(H) se nalazi i pravilo $\dfrac{p_1,\ldots,p_k}{p_{k+1}}$, što daje $\vdash p_{k+1}$, tj. $\delta 1(p_{k+1})=\top$. Odatle zaključujemo $\delta 1(h)=\top$, pa se dokaz završava i u slučaju 2°. Sada ćemo dokazati obrat tvrđenja **(7.3.4)** odnosno ovo tvrđenje:

(7.3.5) $$H \vdash p \longrightarrow \vdash_{ft(H)} p$$

gde je p ma koje slovo. Zaista, neka važi $H \vdash p$. Uočimo maločas napravljenu vrednost $\delta 1$. Pošto je to model za H, to onda mora biti model i za p, tj. mora važiti $\delta 1(p)=\top$, odnosno drugim rečima $\vdash p$, čime se završava dokaz implikacije **(7.3.5)**. I tako, smo u stvari dokazali ovaj stav:

Stav 7.3.2. Deduktivan model δ Hornovskog skupa H može se ovako definisati

$$\delta(p)=T \longleftrightarrow \vdash_{ft(H)} p$$

gde je p ma koje slovo, a ft(H) je formalna teorija pridružena skupu[23] H.

Primer 7.3.1. Dat je skup H Hornovskih formula

$$u, \quad v, \quad a \wedge b \Rightarrow q, \quad q \wedge r \wedge s \Rightarrow p,$$
$$u \wedge v \Rightarrow q, \quad u \Rightarrow r, \quad q \wedge r \Rightarrow s$$

(i) Da li je p posledica skupa H ?
(ii) Opisati sve posledice skupa H.

Rešenje. U skladu sa Stavom 7.3.2 sva pitanja posledičnosti iz H se prevode na takva pitanja u odnosu na ovu formalnu teoriju ft(H):

Aksiome: u, v

Pravila $\dfrac{a,b}{q}$ $\dfrac{q,r,s}{p}$ $\dfrac{u,v}{q}$ $\dfrac{u}{r}$ $\dfrac{q,r}{s}$

Tim prevođenjem utvrđivanje posledičnosti je postalo mnogo određenije jer konačno umesto korišćenja "cele logike" dosta je da "uposlimo" formalnu teoriju ft(H). Ali kako upotrebiti teoriju ft(H) ? U tu svrhu upoznaćemo dve opšte zamisli: <u>navišnu</u> i <u>naniżnu.</u> Najpre ukratko izlažemo navišnu zamisao, odnosno navišan algoritam. Tokom njega će se korak po korak postupno praviti skup svih teorema teorije ft(H). Sledstveno, taj algoritam će svoim "plodovima" dati odgovor na pitanje (ii). Ako se tokom algoritma među teoremama bude pojavila formula p, onda ćemo moći da odgovorimo i to potvrdno i na prvo pitanje. Navišni algoritam ćemo izložiti nešto skraćenije, logički ispravno, ali i "malo grublje", jer ne ćemo previše voditi računa da li se neke radnje, sastavci algoritma ponavljaju. Drugim rečima taj algoritam je matematički ispravan, ali vremenski nije dovoljno kratak. Evo kako ukratko teče algoritam u slu[aju teorije ft(H):

Korak 1: Teoreme su u, v. (Uzeli smo date aksiome)

Korak 2: Sada na sve prethodne teoreme na razne načine koristimo data pravila. Tako se dolazi do ovih novih teorema, reći ćemo "novaka": q, r. Novake pridružimo starim teoremama i tako dobijemo nov skup teorema: { u, v, q, r }

Korak 3: Sada na sve teoreme na razne načine primenjujemo pravila teorije i odvajamo novake. To će sada biti samo s

Znači nov skup teorema je { u, v, q, r, s }

Tu se već primećuju neke crte opšteg navišnog algoritma. Naime, tokom njega se korak za korakom javlja nešto poput Koraka 3:

(ϕ) Na sve teoreme na razne načine primenjujemo pravila teorije i odvajamo novake. Novi skup teorema je unija starog skupa teorema i skupa novaka.

U vezi sa (ϕ) istaknimo da se - iz razloga da algoritam kraće traje - on treba tako poboljšati da se pravila teorije ne koriste na one formule na koje je to već bilo učinjeno u nekom prethodnom koraku[24]. Dalje, u vezi sa tokom navišnog algoritma u opštem slučaju postoje dve mogućnosti:

> Stalno se pojavljuju <u>novaci</u>, i tada algoritam nikad ne staje; <u>ili</u> u nekom koraku skup novaka je prazan, i tada se algoritam završava.

Nastavljamo izlaganje u vezi sa teorijom ft(H), odnosno idemo na iduci korak:

Korak 4 Radimo slično kao u Koraku 3. Pojavi se novak p. Znači u tom koraku je utvrđen potvrdan odgovor na pitanje (i).
Nov skup teorema je { u, v, q, r, s, p }

Korak 5 Radimo slično kao u Koraku 3. Ali, sada više nema novaka, pa se algoritam završava.

Znači imamo ovaj zaključak:

(i) p jeste posledica skupa H
(ii) Skup svih teorema[25] skupa H je { p, q, r, s, u, v }

Tako smo navišnim algoritmom dali odgovore na oba pitanja (i), (ii).

A sada prelazimo na izlaganje naniżnog algoritma. Uz insistiranje na redosled to će u stvari, biti prološki algoritam. Tako, na pitanje (i) odgovaramo ovako:

> Pošto p nije član skupa H, tj. nije aksioma onda p je posledica od H upravo ako je p dokazivo primenom drugog ili četvrtog pravila teorije ft(H). Pokušajmo prvo dokaz sa drugim pravilom. Da bi p bilo teorema treba teoreme da budu q, r, s. Za dokazivanje q se može koristiti prvo, odnosno treće pravilo. Međutim, dokaz prvim pravilom odmah propada jer u vezi sa a, b nemamo nikakvih podataka. Ali, dokaz trećim pravilom uspeva jer u, v su aksiome. Dokazavši q sada je na redu dokaz za r. I to uspeva na osnovu četvrtog pravila, jer u je aksioma. Dokazavši r sada je na redu dokazivanje za s. Koristimo peto pravilo prema kome treba da dokažemo q i r. U stvari to smo već bili učinili i konačno zaključujemo da je p posledica skupa H.

Međutim, kako sada naniżnim algoritmom odgovoriti na pitanje (ii) ? Kratko rečeno to je nemoguće. Prolog bi nam sa razlogom "zamerio":

> Moj algoritam i nije nameren da odgovara na takva pitanja.

---

[23] Videti **(7.3.3)**

[24] Jer ponovno korišćenje je beskorisno, ne može dati novu teoremu.

[25] To je redak slučaj da smo kao u tom primeru u stanju da nađemo skup svih teorema.

7.4 Deduktivan model Hornovskih formula (Slučaj predikatskih formula)

U delu 7.3 smo govorili o Hornovskim iskaznim formulama. Definicija 7.3.2 bi se mogla običnim rečima ovako kratko izreći:

(7.4.1) Ma koje "slovo" je Hornovska formula, a takođe i konjunkcija konačno mnogo "slova" povlači[26] "slovo" je Hornovska formula.

Taj oblik izražavanja je podesan jer iz njega neposredno sledi definicija Hornovskih predikatskih formula. Naime, tada se u tom opisu reč "slovo" tumači kao tzv. elementarna predikatska formula, koja uopšte ima oblik:

rel(term1, term2, ..., termk)

gde je rel znak neke relacije dok term1, ..., termk su izvesni izrazi (termi). Kao što vidite Prolog je pozajmio taj pojam i u njemu se koristi naziv formula (vid. (1.1)). Evo primera Hornovskih predikatskih formula[27]:

(*1) $a(2,f(x,y)),\ b(x,y*z) \wedge c(y,g(z,x)) \Rightarrow a(x,z)$

U tim formulama prećutno x, y, z su promenljive[28], a, b, c su relacijski znaci dužine 2, f, g i * su operacijski znaci takođe dužine 2, a 2 je tzv. znak konstante. Obično se umesto znak konstante kaže[29] operacijski znak dužine 0. U vezi sa tim znacima operacija i relacija kaže se da je skup formula (*1) na ovom relacijsko-operacijskom jeziku { 2, f, g, *, a, b, c}.
Sada iz Matematičke logike pozajmljujemo pojam modela predikatskih formula. Neka je uopšte H skup nekih Hornovskih formula na izvesnom relacijsko-operacijskom jeziku L. Dalje, neka je S neki neprazan skup. Svakom relacijskom i svakom operacijskom znaku iz L pridružimo po jednu relaciju, odnosno operaciju skupa N; pri čemu moraju biti iste dužine znak i njemu pridružena relacija (operacija)[30]. Kaže se da smo takvim pridruživanjem[31] od skupa S napravili jednu relacijsko-operacijsku strukturu nad jezikom L. Za sam skup S se kaže da je nosilac (domen) te strukture. Recimo, neka je H ovaj

---

[26] Reč povlači odgovara implikaciji ⇒.

[27] U stvari, izložena definicija (7.5.1) pokriva samo pojam tzv. pozitivnih Hornovskih formula bez kvantora. Opštije, poput slučaja iskaznih formula kao Hornovske formule se prihvataju
Negacije "slova", a takođe i formule oblika
Konjunkcija konačno "slova" povlači negaciju "slova"
a dalje dopušta se da Hornovske formule po volji sadrže logičke kvantore. Recimo, Hornovske formule su
$a(x,y),\ \neg\ p(u,7,g(6))$
$(\forall\ x)(\exists\ y)\ (a(g(x),y) \wedge (\exists\ u)\ b(2,g(u))) \Rightarrow (\exists\ z)\ c(z)$
Kratko rečeno, "brisanjem" kvantora bi nastale "obične", tj. bezkvantorske Hornovske formule.

Naravno, mogli smo i unapred usvojiti neki dogovor kakve reči se smatraju promenljivim.

To je recimo, uobičajeno u Algebri.

[30] Funkcijskim znacima dužine 0, tj. znacima konstanata se pritom pridružuju izvesni po volji odabrani elementi iz S. Kratko rečeno, znaci konstanata se tumače kao konstantni elementi skupa S.

[31] Često se koristi i termin interpretacija.

skup formula

(*2) $r(x,f(x)),\ r(x,y) \Rightarrow r(y,x)$

na jeziku L= {f, r}. Tada jedna relacijsko-operacijska struktura nad tim jezikom je određena ovako:

(*3) Skup S, nosilac strukture, je skup svih prirodnih brojeva 1, 2, ... Dalje, znaku f se pridružuje funkcija definisana jednakošću f(x)=x+1, a relaciji r se pridružuje relacije "različitost".

Sada smo u prilici da definišemo pojam modela. Tako, neka je H skup datih Hornovskih formula nad izvesnim jezikom L i neka je $\mathcal{S}$ izvesna relacijsko-operacijska struktura nad jezikom L. Tada kažemo da je ta struktura model formula H, ukoliko za sve vrednosti promenljivih (u skupu S -nosiocu strukture) svaka od formula iz H je ispunjena, tj. tačna jel[32]. Recimo, sa (*3) je očigledno opisan jedan model formula (*2).

Nama je osnovna namera da uvedemo pojam deduktivnog modela. Međutim, na putu se "prepreČuje" pojam izrazovskog (termovskog) modela. To je model koji ima domen posebnog oblika - napravljen od termova, i uz to funkcijski znaci se tumače na poseban način, odnosno njima se pridružuju odgovarajuće izrazovske (termovske) operacije. Evo prvo objašnjenja tih operacija. Uočimo, dve konstante a, b i binaran operacijski znak *. Dalje, napravimo "svet" odgovarajućih *-termova[33]. U njega, označenog sa Term dolaze:

a, b, (a*a), (a*b), (b*a), (b*b), ..., ((a*b)*(b*b)), itd.

Tada izrazovska operacija, u oznaci, * se uvodi ovako:

x * y je, po definiciji, term (x*y)

Znači, recimo, imamo ove jednakosti:

a*a=(a*a), (a*b)*(b*b)= ((a*b)*(b*b))

Neka je trenutno f ma koji operacijski znak dužine m. Njemu se onda dodeljuje izrazovska operacija f određena ovako:

(7.4.2) f(O1, ..., Om) je, po definiciji, term f(O1, ..., Om)

gde O1, ..., Om su izvesni termi. Recimo, ako m=3 imamo ovakve jednakosti

f(a, b, c)=f(a, b, c), f(g(a, b), f(a, a, a), b)=f(g(a, b), f(a, a, a), b)

Sada ćemo navesti jedan primer term-modela. Upravo ćemo opisati jedan takav model skup formula (*2). Prvo treba da napravimo domen, odnosno da uzmemo neki svet "konstantskih" f-termova, gde je f operacijski znak iz (*2). U tu svrhu uočimo ovaj pomoćni skup (tzv. "generatorni" skup) Γ={ a, b}, gde su a, b dogovorno znaci konstanata. Dalje, nad skupom Γ napravimo sve f-terme. ako nastaje "svet" termova Term u čije članove recimo, dolaze ovi termovi

a, b, f(a), f(b), f(f(a)), f(f(b)), ...

U stvari, tako smo gotovi sa pripremanjem skupovnog dela term-modela. Znaku

---

[32] kao odgovarajuće relacije, operacije domena S.

[33] Pretpostavljamo infix-notaciju, tj. * uvek stoji "u sredini" izraza, kao recimo kod izraza (a*b), ali ne i kod *(a, b).

f ,kako rekosmo, odgovara već opisana termovska operacija f:

f(x) je term f(x)

U skladu sa rečenim, term-model za (*2) će dalje biti ma koji model čiji domen je Term, a operacija je f. Drugim rečima, u odrednici term-modela nije ništa rečeno o relacijama. To praktično znači da nas sada čeka definisanje relacije r. Tu će se sada pojaviti jedan značajan pojam: razliv. Naime, zamislimo da se u formulama (*2), promenljive x,y na sve moguće načine zamene članovima iz Term, tj. sveta termova. Tako nastaju razne ovakve ve formule:

(*4) r(a,f(a))
r(b,f(b))
r(f(a),f(f(a)))
r(f(b),f(f(b)))
r(f(f(a)),f(f(f(a))))
.... (To su razne koje nastaju iz formule r(x,f(x)))
r(a,a)⇒r(a,a)
r(a,b)⇒r(b,a)
r(a,f(a))⇒r(f(a),a)
r(b,a)⇒r(a,b)
... (To su razne nastale iz formule r(x,y)⇒r(y,x))

Skup svih takvih formula se naziva razliv skupa (*2) po svetu Term, kraća oznaka

Razliv(H,Γ)

gde smo namerno upotrebili opšte oznake H,Γ. Ovde H je (*2), a Γ={a,b}. Sada je bitno primetiti da je dobijeni Razliv shvatljiv kao skup iskaznih formula po, doduše, malo složenijim "slovima":

r(a,a), r(a,b), r(b,a), r(b,b), r(a,f(a)),...

U skladu sa tim određivanje relacije r se svodi na traženje modela Razliva kao skupa iskaznih formula. Drugim rečima, da bismo našli r treba da zadovoljimo iskazne uslove Razliv, tj. u konkretnom primeru uslove (*4). Recimo, jedno rešenje je određeno rečima:

Svako "slovo" r(t1,t2) ima vrednost τ, tj. tačno, što daje relaciju r u kojoj se nalaze ma koja dva terma[34] t1,t2.

Moglo bi se reći da je to model "u kome ima najviše τ-ova", odnosno čak sva slova imaju vrednost τ. To je ujedno i trivijalno rešenje.
Može se pomisliti: da li postoji potpuno drukčije rešenje: "sa što manje τ-ova" ? Drugim rečima: "da važi samo ono što mora". Pokušaćemo da nađemo takvo rešenje. Kao prvo, vidimo da sva slova oblika r(t,f(t)) ,gde je t ma koji term, moraju imati vrednost τ, što sledi iz prvog dela Razliva (*4), odnosno dela koji "izvire" iz formule r(x,f(x)). Preostali deo Razliva čine formule oblika r(t1,t2)⇒r(t2,t1) gde t1,t2 "trče" kroz svet Term. Među tim formulama nalaze se i ovakve r(t,f(t))⇒r(f(t),t), gde t ε Term. Odatle zaključujemo da i sva slova oblika r(f(t),t) moraju imati vrednost τ. Nije teško videti da smo već pronašli sva slova koja moraju da imaju vrednost τ. Dogovorno preostalim slovima dodelimo vrednost ⊥. Znači, dobili smo ovu vrednost v slova:

$$v(r(t1,t2)) = \begin{cases} \tau & \text{ako t2 oblika f(t1) ili ako t1 oblika f(t2)} \\ \bot & \text{inače} \end{cases}$$

a ona "se pretače" na ovu relaciju r:

r(t1,t2) važi upravo ako je t2 oblika f(t1) ili ako je t1 oblika f(t2)

Lako se proverava da smo tako određenom relacijom r dobili jedan model skupa (*2) i na taj način postigli cilj "građenja modela sa najmanje τ-ova". To je u stvari deduktivan model skupa (*2), što će biti jasno iz opštih definicija koje slede.

Neka je H ma koji skup Hornovskih formula nad izvesnim jezikom L. Dalje, neka je Γ neki skup konstanata. Označimo sa Term(Γ,L) skup svih termova građenih od članova skupa Γ, znakova konstanata iz L (ako ih ima) i operacijskih znakova iz L. U skupu Term (Γ,L) svakom operacijskom znaku pridružimo odgovarajuću izrazovsku operaciju tipa (7.4.2). Oni modeli skupa H kojima je domen oblika Term(Γ,L) a operacije su im izrazovske nazivaju se term-modeli. Među njima posebnu ulogu imaju tzv. deduktivni modeli, koji nastaju kako sledi. Prvo se svim mogućim zamenama promenljivih članovima iz skupa Term(Γ,L) od formula iz H napravi skup iskaznih formula koji nazivamo Razliv H po Term(Γ,L) i koji kraće označavamo sa Razliv(H,Γ). Najzad, svakom relacijskom znaku r ε L, dužine m se u skupu Term(Γ,L) dodeljuje relacija r definisana ovako

(7.4.3) r(t1,t2,...,tm) važi ako i samo ako Razliv(H,Γ)⊢r(t1,t2,...,tm)

tj. rečima: Termi t1,t2,...,tm su u relaciji r ako i samo ako "slovo" r(t1,t2,...,tm) sledi iz Razliv(H,Γ).

Pošto je Razliv(H,Γ) Hornovski skup formula to po Stavu 7.3.1 on ima deduktivan model. Definicija (7.4.3) upravo znači "pretakanje" tog modela iskaznog tipa na relacisko-operacijski model skupa H. Znači, sa (7.4.3) je definisan jedan model od H. Taj model nazivamo deduktivan model od H generisan sa Γ i označavamo ga sa Ded(H,Γ).
Pomenimo da ukoliko jezik L na kome su formule iz H sadrži neki znak konstante, onda skup Γ može biti i prazan, jer konačno svet termova se može sagraditi pomoću tih znakova konstanata. Recimo, ako je H ovaj skup formula

r(a),r(b), r(x)∧r(y)∧r(z) ⇒ r(((x*y)*z))

gde su a,b znaci konstanata, onda u skladu sa rečenim postoji Ded(H,ν), gde ν označava prazan skup. Sada svet termova je svet *-termova građenih od a, b, tj. u njega ulaze termi kao a,b, (a*a), (a*b), ((a*b)*(b*b)) , itd.
Primetimo da definicija (7.4.3) propisuje i kad relacija r u uočenoj "tački" ne važi. Naime, to je upravo ako r(t1,...,tm) nije dokazivo. Verovatno ste to već povezali sa definicijom negacije u Prologu[35].

U vezi sa deduktivnim modelima uopšte sada istaknimo da se Hornovski skupovi i posebno deduktivni modeli (doduše pod imenom slobodni modeli, slobodne algebre) već odavno koriste u Algebri. To je prosto tipično za tzv. Teoriju univerzalnih algebri.

---

[34] To je tzv. puna relacija skupa Term.

[35] Kratko rečeno, Prolog sme da ima takvu definiciju negacije jer u njegovoj osnovi je Hornovski skup formula, koji kao takav ima deduktivan model.

7.5 Deduktivan model i Prolog

U ovom izlaganju ćemo potpuno podrobno objasniti vezu koja postoji izmedu Prologa i pojma deduktivan model. Da bismo to učinili najpre obavljamo sledeće razmatranje.

Neka je P ma koji proloski program i neka je ϕ neka njegova teorema, tj. formula dokazana proloskim algoritmom u konačno koraka. Označimo sa Δ taj pretpostavljeni proloski dokaz formule ϕ. Tada budući da je proloski algoritam veoma složen u tom dokazu se može pojaviti, slobodnije rečeno, čitava šuma mogućnosti. Naime, tokom algoritma, kao što znamo, često se radi dokaza neke formule upošljava neki članak, neka grana i pokušava se dokazivanje pomoću nje. Ali, ako takav pokušaj ne uspe, onda na izvesnom mestu nastaje procedura vraćanje (backtracking) koja pored ostalog podrazumeva i promenu neke grane ("pregranjavanje"). S tim u vezi, kad se dokaz završi smemo reći da su u njemu neke grane bezuspešno učestvovale, tj. "propale", za razliku od onih pomoću kojih je, u stvari, sklopljen čitav dokaz. Shodno tome, prirodno se pojavljuje pomisao da pomenutu "šumu mogućnosti" nekako "pročistimo" odbacujući sve "propale grane". U stvari, upravo nam je to sledeća namera. Da bismo to uradili u vezi sa zamišljenim dokazom Δ uvešćemo dva važna pojma:

svedočku grana neke formule ψ i svedočko drvo dokaza Δ

Da bismo objasnili te pojmove zamislimo dokaz Δ od samog početka. Posao je, kako smo pretpostavili, dokazivanje formule ϕ. U tu svrhu na početku su korišćeni članci sa glavama čija imena su ista kao ime formule ϕ. Privremeno takve članke nazovimo ϕ-članci. Naravno, jasno je da je najpre korišćen jedan od njih. Može se dogoditi da taj pokušaj ne uspe. To onda znači da moramo uposliti neki drugi od ϕ-članaka. I slično se razmišljanje nastavlja dalje. Ali, kako je pretpostavka da je formula ϕ dokazana, to zaključujemo:

Jedinstveno postoji ϕ-članak pomoću koga je dokazana formula ϕ

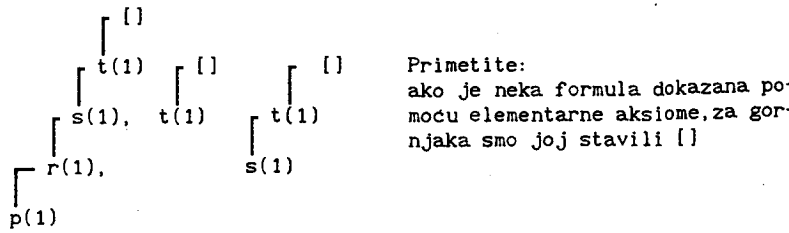
Pri korišćenju tog zamišljenog članka je upotrebljena procedura priključivanje. U vezi sa tim na drvetu celog proloskog algoritma će se pojaviti ovakav detalj:

$$\begin{array}{l} \ulcorner\ \phi 1,\ \phi 2,\ \ldots,\ \phi k \\ \phi \end{array} \qquad (7.5.1)$$

gde smo ispustili deo koji sadrži red i duz. Dakle, dokaz formule ϕ je sveden na dokaz gornjaka $\phi 1, \ldots, \phi k$, tj. na po redu dokaze formula $\phi 1, \ldots, \phi k$. Taj gornjak dogovorno nazivamo svedočka grana formule ϕ. U stvari, tokom dokaza Δ svaka usput pojavljena formula ψ na sličan način ima jedinstveni članak svog dokazivanja, kojim je jedinstveno određen odgovarajući gornjak. Taj gornjak, slično kao i maločas, se naziva svedočka grana formule ψ. I tako, kratko rečeno, u odnosu na zamišljeni dokaz Δ svakoj formuli, učestvujućoj u tom dokazu, odgovara svedočka grana. A sada zamislimo da se isključivo od takvih svedočkih grana[1] polazeći od ϕ postupno sklopi jedno drvo. Tako dobijeno drvo nazivamo svedočko drvo dokaza SD. Navešćemo jedan primer svedočkog drveta. Uočimo ovaj program

```
q(1):-r(2).        r(X):-s(X),t(X).
p(X):-q(X),r(X).   p(X):-r(X),s(X).
s(1):-q(1).        s(1):-t(1).      t(1).
```

i postavimo pitanje ?-p(1). Tada je odgovor da, a svedočko drvo izgleda



Primetite:
ako je neka formula dokazana pomoću elementarne aksiome, za gornjaka smo joj stavili []

U skladu sa njegovom definicijom to drvo pamti samo one grane preko kojih je obavljen dokaz, odnosno svedočke grane[2]. Blagodareći tom svojstvu na osnovu svedočkog drveta možemo napraviti ovakav, reći ćemo, "skraćen" proloski algoritam za dokaz formule p(1):

```
p(1):-r(1),s(1)
    r(1):-s(1),t(1)
        s(1):-t(1)
            t(1)
```

Inače rekli smo "skraćeni" proloski algoritam jer pravi proloski algoritam sadrži još mnoge detalje, koji se odnose na "propale medudokaze". Naravno, bilo bi idealno kad bismo u opštem slučaju umeli da pravimo takve skraćene proloske algoritme. Uprkos jasnim preprekama koje nam to nedopuštaju, ipak jasno je ovo:

(7.5.2) Ako je formula ϕ proloska teorema programa P čije svedočko drvo je SD, onda koristeći to drvo može se ta formula ϕ dokazati skraćenim proloskim algoritmom.

Zaista, slično gornjem primeru, iz drveta SD "izvadimo", izdvojimo sve proloske implikacije oblika

(7.5.3) ψ :- $\psi 1, \ldots, \psi s$ ($\psi 1, \ldots, \psi s$ je gornjak za ψ)

Tada je jasno da takve implikacije "podesno poredane" će predstavljati skraćeni proloski algoritam. U upravo rečenom možda nije odmah potpuno jasno šta znači "podesno poredane". To postaje jasno ako se primeti ova lepa i zanimljiva činjenica:

(7.5.4) Nije teško zaključiti da skraćeni algoritam, budući da je očišćen od svih koraka koji nisu bili uspešni, odnosno produživi u celokupni dokaz, od svih opštih proloskih procedura koristi samo ove dve:

Penjanje na vrh i Plus- spust.

To zapažanje može na prvi pogled izgledati kao veoma primamljivo za pravljenje skraćenih proloskih algoritma. Medutim, misleći o tim dvema procedurama lako se vidi da glavni problem je prva od njih. Naime, pri penjanju na vrh u svakom hodu bira se, možemo reći, "dobra", odnosno "uspešna" grana, a jasno je da to unapred ne možemo znati.

---

[1] Takva jedna je (7.5.1).

[2] Recimo, pri dokazivanju date formule p(1) Prolog je prvo bio uposlio granu p(X):-q(X),r(X) ali to se završilo neuspehom.

Sada smo blizu iskazivanja i dokazivanja osnovnog stava, odnosno niže izloženog Stava 7.5.1. Međutim, prvo ćemo uvesti jednu oznaku, a onda dokaz tog stava videti na nekoliko primera. I tek posle toga ćemo preći na sam stav. U vezi sa ma kojim proloških programom P sa H(P) označavamo skup odgovarajućih Hornovskih formula, pri čemu prvo znak reza ! izostavljamo i drugo znak fail zamenjujemo nekom dogovorno odabranom formulom koja je sigurno proloski nedokaziva[3]. Recimo, programu

```
a(1,2):-fail.
b(X):-a(X,Y),a(Y,X).
c(X):-b(X),!.
```

može da odgovara ovaj skup H(P)

```
a(777) ⇒ a(1,2).
a(X,Y),a(Y,X) ⇒ b(X)
b(X) ⇒ c(X)
```

Primetite da smo fail zamenili sa a(777), jer to očigledno nije dokazivo.

Sada na nekoliko primera potvrđujemo ono što tvrdi Stav 7.5.1, odnosno da je svaka proloska teorema ujedno teorema odgovarajućeg Hornovskog skupa

Primer 7.5.1. Uočimo proloski program

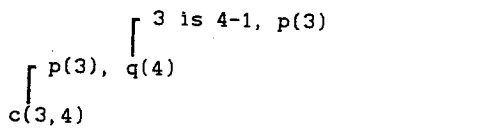
```
b(0,1):-!.  b(X,Y):-X>0,X1 is X-1,b(X1,Y1),Y is X*Y1.
a(X,Y):-b(X,Y).  a(X,Y):-c(X,Y).  a(X,Y):-p(X),q(Y).
c(3,4):-fail.
p(3).
q(X):-Y is X-1, p(Y).
```

i u vezi sa njim pitanje ?-a(3,4). Ako je ta formula teorema naći odgovarajući Hornovski skup H kome je ona takođe teorema.

Rešenje. Datom programu P odgovara ovaj skup Hornovskih formula H(P)

```
b(0,1)
X>0, X1 is X-1,b(X1,Y1), Y is X*Y1 ⇒ b(X,Y)
b(X,Y) ⇒ a(X,Y)
c(X,Y) ⇒ a(X,Y)
p(X),q(Y) ⇒ a(X,Y)
c(777) ⇒ c(3,4)
p(3)
Y is X-1, p(Y) ⇒ q(X)
```

Prolog prvo za a(3,4) traži dokaz pomoću grane: a(X,Y):-b(X,Y). ali to ne uspeva, jer b je u stvari faktorijelna relacija, a 3! nije 4. Posle se pređe na granu a(X,Y):-c(X,Y) ali i taj pokušaj propadne. I konačno, preostane dokaz preko grane a(X,Y):-p(X),q(Y). Odgovarajuće svedočko drvo izgleda



S tim u vezi skraćeni proloski dokaz glasi

```
c(3,4)   ukoliko p(3) i q(4)
```

[3] Cilj je da izbacimo fail, pa u skladu sa rečenim recimo možemo ga zameniti sa rel(a) uz pretpostavku da rel(a) nije proloski dokaziva.


```
Dalje, p(3) jeste.
Dalje, q(4) važi ukoliko 3 is 4-1 i p(3).
Tačno je : 3 is 4-1, kao i p(3).
Kraj dokaza.
```

Taj dokaz možemo i ovako kratko zapisati:

```
c(3,4) ⇐  p(3),q(4).
 p(3).
 q(4) ⇐ 3 is 4-1, p(3).
 3 is 4-1.
 p(3).
```

Označimo sa H Hornovski skup koji odgovara tim formulama, tj. ovaj skup

```
p(3),q(4) ⇒ c(3,4)
 p(3)
3 is 4-1, p(3) ⇒ q(4)
3 is 4-1
p(3)
```

Primetimo, što je bitno, da se taj skup se može dobiti iz H(P) ovako. On je deo razliva skupa H(P), kada se za Γ uzme skup ovih osnovnih sastavaka 3,4. To je očigledno jer, recimo, prva formula p(3),q(4) ⇒ c(3,4) nastaje iz pete formule H(P) pri zameni X-->3, Y-->4.
Međutim , potpuno je jasno da se na osnovu tog skupa H logički zaključuje teorema c(3,4). Naime, lako se vidi, da jedan takav dokaz nastaje "preokretanjem" navedenog skraćenog proloskog dokaza. I tako znači, za teoremu c(3,4) smo napravili Hornovski skup H tako da ta formula bude logička posledica od H. Primetite da je redosled koraka sledeći:

(7.5.5)

(i) Prvo je napravljen skraćen proloski dokaz formule ϕ

(ii) Drugo, za skup H je ovde uzet skup svih Hornovskih formula koje nastaju iz formula tog dokaza.

(iii) Nije upotrebljen dodelnik[4], a slučaj kad se on koristi ćemo objasniti malo kasnije.

(iv) Formula ϕ je logička posledica tog Hornovskog skupa H. Slobodnije rečeno, jedan takav dokaz nastaje preokretanjem skraćenog proloskog dokaza. Taj dokaz se može pratiti i na svedočkom drvetu i onda se može reći da on postupno teče od "vrhova" drveta, od "listova" ka njegovom početku, odnosno "korenu"..

(v) Skup H je deo razliva koji nastaje iz skupa H(P), kad se za Γ uzme

(*) skup svih znakova konstanata učestvujućih u H.

U tom primeru je tako, ali kao što ćemo dalje videti deo (*) se često znatno proširuje.

Opis je namerno dat i sa najavama uopštenja, da bismo pomoću njega lakše stigli do opšteg slučaja.

Primer 7.5.2. Dat je program

```
elem(X,[X|Y]).
elem(X,[U|V]):-elem(X,V).
```

[4] U ovom izlaganju rekavši Dodelnik podrazumevamo skraćeni dodelnik, tj. konačan niz članova oblika (Var,Vred) (rečima: promenljiva, vrednost).

```
dodaj([],A,A).
dodaj([A|B],C,[A|Rez]):-dodaj(B,C,Rez).
okret([],[]).
okret([A|B],Rez):-okret(B,Rez1),dodaj(Rez1,[A],Rez).
```

Za svaku od niže navedenih proloških teorema ϕ naći odgovarajući skup Hornovskih formula H, tako da važi $H \vdash \phi$

(a) elem(4,[3,2,4]). (b) elem(X,[1,2]). [c] dodaj([1,2,3],[4,5],X).
(d) elem(A,[A|B]) (e) okret([1,2,3],X).

Rešenje. Uočeni program je dobro poznat. Relacija elem je relacija "biti član od", dalje dodaj je relacija: "na listu A dopisivanjem liste B dobije se lista C" i najzad relacija okret je u vezi sa okretanjem date liste[5]. Datom programu P odgovara ovaj skup H(P) Hornovskih formula

(7.5.6)
```
elem(X,[X|Y])
elem(X,V) ⇒ elem(X,[U|V])
   dodaj([],A,A).
   dodaj(B,C,Rez) ⇒ dodaj([A|B],C,[A|Rez])
      okret([],[]).
      okret(B,Rez1),dodaj(Rez1,[A],Rez) ⇒ okret([A|B],Rez)
```

(a) Prološki dokaz, koji je ujedno i skraćen jer usput nijedna grana ne propada, glasi

```
elem(4,[3,2,4]) ⇐ elem(4,[2,4])      Na osnovu drugog elem-članka
   elem(4,[2,4]) ⇐ elem(4,[4])       Na osnovu drugog elem-članka
      elem(4,[4])                    Na osnovu prvog elem-članka
```

Odatle odmah vidimo ovaj Hornovski skup H

```
elem(4,[4])
elem(4,[4]) ⇒ elem(4,[2,4])
elem(4,[2,4]) ⇒ elem(4,[3,2,4])
```

I očigledno važi $H \vdash elem(4,[3,2,4])$. Inače taj skup H je deo razliva koji nastaje iz H(P) kad se za Γ uzme skup ovih osnovnih sastavaka

2, 3, 4 i prazna lista []

a svet termova se gradi pomoću tih sastavaka i operacijskog znaka | za građenje lista. U stvari, uvek u slučaju prisustva lista u skup Γ uključujemo i praznu listu, i podrazumeva se građenje termova i pomoću znaka |.

(b) Prološki dokaz, sada opet ujedno i skraćen, glasi
elem(X,[1,2]) je tačno po prvom elem-članku.
Dodelnik sadrži dvojku (X,1).
Taj dokaz je dužine 1. Sada se skup H odgovarajućih Hornovskih formula gradi ovako:

(Dod) Prvo se u skraćenom dokazu sve nepoznate, koje mogu, uklone zamenjujući ih njihovim vrednostima. To je inače detalj koji treba dodati u (7.5.5) (iii).

Opet smo namerno rekli nešto opštije i deo "koje mogu" ćemo objasniti malo

[5] Pomenimo, navedeni program nije najbolji za okretanje liste, jer ima dve rekurzije. Međutim, odabrali smo ga upravo iz tog razloga.

kasnije. I tako sada je H ovaj skup

elem(1,[1,2])

Uočena proloska teorema elem(X,[1,2]) pri X=1, je očigledno posledica tog skupa H. Skup Γ u ovom slučaju se sastoji od 1,2 i []. Skup H je deo razliva koji nastaje prve formule u (7.5.6), tj. prve formule u H(P).

(c) Najpre ćemo napraviti proloski dokaz teoreme dodaj([1,2,3],[4,5],X) pri čemu će X dobiti vrednost [1,2,3,4,5]. Dokaz će kao i u prethodnom primeru "sam po sebi" biti skraćen. U njemu znak ⇐ dogovorno zamenjuje reč "ukoliko". Evo prološkog dokaza:

```
dodaj([1,2,3],[4,5],X) ⇐ dodaj([2,3],[4,5],Y)     X=[1|Y]
 dodaj([2,3],[4,5],Y) ⇐ dodaj([3],[4,5],Z)        Y=[2|Z]
  dodaj([3],[4,5],Z) ⇐ dodaj([],[4,5],U)          Z=[3|U]
   dodaj([],[4,5],U)                              U=[4,5]
```

Kao što vidite nismo striktno naveli dodelnik već njegove pojedine članove izrazili jednakostima. Sada pravimo skup H. Sledimo (Dod). Naime, uklanjamo X, Y, Z, U zamenjujući ih njihovim vrednostima. Tako imamo X=[1,2,3,4,5], i sl. Tada navedenom prološkom dokazu odgovara ovaj Hornovski skup H:

```
dodaj([2,3],[4,5],[2,3,4,5]) ⇒ dodaj([1,2,3],[4,5],[1,2,3,4,5])
  dodaj([3],[4,5],[3,4,5]) ⇒ dodaj([2,3],[4,5],[2,3,4,5])
    dodaj([],[4,5],[4,5]) ⇒ dodaj([3],[4,5],[3,4,5])
      dodaj([],[4,5],[4,5])
```

Očigledno je formula dodaj ([1,2,3],[4,5],X) pri X=[1,2,3,4,5] logička posledica tog H; dosta je slediti put koji je suprotan onom koji je koristio Prolog, tj. "preokrenuti" skraćen proloski dokaz. Inače, skupu H odgovara Γ sastavljen od ovih članova 1,2,3,4,5 i []. Za građenje razliva dosta je da se koriste dodaj-formule iz H(P).

(d) Formula elem (A,[A|B]) je očigledno prološka teorema. Posebnost ovog slučaja je što sada nepoznate A,B ostaju "slobodne", tj. smeju imati proizvoljne vrednosti. Skup H sada glasi elem(A,[A|B]) i u njemu su preostale nepoznate A,B, odnosno preostale sve one nepoznate koje su slobodne. Sledstveno tome i te nepoznate ulaze u skup Γ. Naime, sada je Γ sastavljen od ovih članova A,B, []. Opet iskažimo opštu činenicu:

> Skup Γ kao članove ima i sve nepoznate koje su nakon završetka prološkog dokaza ostale slobodne.

(e) Formula okret([1,2,3],X) , kao što ćemo dokazati, je prološka teorema pri X=[3,2,1]. Taj primer smo uzeli jer on je "prepleten" od više sastavaka koji svaki za sebe ima svoj dokaz. Da bismo na kraju lakše "preokretanjem" napravili logički dokaz iz odnosnog skupa H čitav dokaz ćemo prikazati drvetom SD

```
                                  D e o 3
okret([],U),dodaj(U,[3],Z)                          U=[]
                                  D e o 2
okret([3],Z),  dodaj(Z,[2],Y)
                                  D e o 1
okret([2,3],Y),  dodaj(Y,[1],X)
okret([1,2,3],X)
```

u njemu su D e o 3, D e o 2, D e o 1 "poddrveta" koja cemo objasniti. Naime, prvo je pri dokazu okret([],U) pronadeno U i u dodelnik je ušla dvojka (U,[]). Na gornjem crtežu tome odgovara navedena jednakost U=[]. Tada smo prešli na dokaz formule dodaj (U,[3],Z). Tom dokazu odgovara pod drvo D e o 3:

dodaj(U,[3],Z) Z=[3]

koje se sastoji samo iz "korena". Nepoznata Z je dobila vrednost [3]. Dalje, nakon plus_spusta smo stigli na dokaz formule dodaj(Z,[2],Y), odnosno formule[6] dodaj([3],[2],Y). Tom dokazu odgovara ovo poddrvo D e o 2:

dodaj([],[2],Y1) Y1=[2]
dodaj([3],[2],Y) Y=[3|Y1]
D e o 2

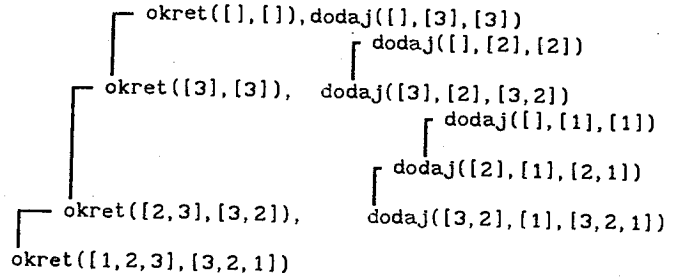
Nepoznate Y,Y1 su dobile navedene vrednosti. Nakon plus_spusta se dode na dokaz formule dodaj (Y,[1],X), odnosno formule[7] dodaj([3,2],[1],X). Taj dokaz je prikazan poddrvetom D e o 1:

dodaj([],[1],X2) X2=[1]
dodaj([2],[1],X1) X1=[2|X2]
dodaj([3,2],[1],X) X=[3|X1]
D e o 1

U tom dokazu nepoznate X,X1,X2 su dobile navedene vrednosti. Čitav dokaz je završen. Skup H se pravi na već opisan opšti način:

Od formula skraćenog dokaza se napravi odgovarajući Hornovski skup i u tim formulama sve se nepoznate zamene svojim vrednostima, izuzev slobodnih koje preostanu.

Medutim, da bismo pored toga lakše videli i da je formula okret([1,2,3],X) pri X=[3,2,1] logička teorema tog H na drvetu SD sve nepoznate zamenimo njihovim vrednostima. Tako nastaje ovo preradeno SD drvo



U stvari, drvo direktno pamti i skup H i dokaz da H⊢okret([1,2,3],[3,2,1].

---

[6] Zamenili smo Z da bismo se lakše izražavali.

[7] Na osnovu dodelnika se vidi da Y=[3,2]. Radi lakšeg izražavanja Y smo zamenili njegovom vrednošcu.

Postupno "od vrhova nanize " navodimo članove skupa H i usput ukazujemo kako se pravi dokaz tog "sleda" :

okret([],[])
dodaj([],[3],[3])
okret([],[]),dodaj([],[3],[3]) ⇒ okret([3],[3])
Dotle smo dokazali H⊢ okret([3],[3])
dodaj([],[2],[2])
dodaj([],[2],[2]) ⇒ dodaj([3],[2],[3,2])
Sa te dve formule smo dokazali H⊢ dodaj([3],[3],[3,2])
dodaj([],[1],[1])
dodaj([],[1],[1]) ⇒ dodaj([2],[1],[2,1])
dodaj([2],[1],[2,1]) ⇒ dodaj([3,2],[1],[3,2,1])
Sa tri poslednje formule smo dokazali
H⊢ dodaj([3,2],[1],[3,2,1])
okret([3],[3]), dodaj([3],[2],[3,2]) ⇒ okret([2,3],[3,2])
Pošto su pretpostavke te implikacije već H-dokazane to odatle sledi H⊢ okret([2,3],[3,2])
okret([2,3],[3,2]),dodaj([3,2],[1],[3,2,1) ⇒ okret([1,2,3],[3,2,1])
Njene pretpostavke su H-dokazane, pa konačno sledi:
H⊢okret([1,2,3],[3,2,1])

Inače, jasno je da je skup Γ u tom primeru sastavljen od ovih osnovnih članova 1,2,3 i [].Završetak Primera 7.5.1.

Sada iskazujemo sledeći osnovni stav o vezi proloških teorema sa teoremama odgovarajućih Hornovskih skupova formula:

Stav 7.5.1. Neka je P dat proloski program i neka je ϕ formula proloski dokazana iz P, čiji jedan dokaz je Δ. Tada se može efektivno naci skup Γ tako da važi:
Formula ϕ je logička posledica Hornovskog skupa Razliv(H(P),Γ)

Dokaz izvodimo korak po korak:
Prvo, na osnovu dokaza Δ napravimo svedočko drvo.
Drugo, na osnovu svedočkog drveta napravimo skraćeni proloski dokaz. Ma koji član tog dokaza ima oblik (7.5.2).
Trece, u vezi sa datim dokazom Δ odgovarajuci dodelnik pamti koje nepoznate su dobile svoje vrednosti. U ovom koraku, u svakom članu skraćenog dokaza sve nepoznate zamenimo njihovim vrednostima. Tako ce od proloske implikacije oblika (7.5.2) nastati nova implikacija označena ovako

(*) $\psi' :- \psi 1', \ldots, \psi s'.$

Ta implikacija je tačna, jer (7.5.2) važi uz neke veze nepoznatih, koje su zapamćene u dodelniku, a (*) je upravo napravljena koristeći te ci te veze. Naravno neke nepoznate, upravo one koje su slobodne, mogu preostati u formulama (*).
Skup H je , u vezi sa (*) , skup svih implikacija oblika
$j\psi 1', \ldots, \psi s' \Rightarrow \psi$
Očigledno je formula ϕ logička posledica tog skupa H. Neka je Γ skup svih znakova konstanata, preostalih nepoznatih i znaka [], ukoliko se u dokazu za ϕ i gde pojavljuju liste. Dokaz se završava buduci da je uočeni skup H deo skupa Razliv(H(P),Γ).

8. BAZE PODATAKA i PROLOG

8.1 Uvod

Jedna od savremenih oblasti Računarstva je tzv. Teorija baze podataka. U njoj postoje razni problemi i jedan od najvažnijih je , slobodnije rečeno, problem više-ključnih spiskova. Tako, za početak uočimo ovaj kratak spisak

(8.1.1)

red	ime	prezime	ocena
1	milan	perić	8
2	jovan	delić	7
3	zoran	arsić	9
4	dušan	ilić	8
5	milan	jović	7

studenata sa ocenama iz nekog predmeta. Reći ćemo da u tom spisku ima 3 "ključa": ime,prezime,ocena, pa je to tro-ključni spisak. Neki od problema koji se mogu postaviti su:

Koji student odgovara datom imenu, ili datom prezimenu, ili datoj oceni?

Recimo, sa imenom 'zoran' imamo , možemo reći, studenta broj 3,odnosno punim opisom, to je:

```
zoran  arsić  9
```

Na prvi pogled se čini da je Prolog, kao jezik,skoro idealan za takav problem. Prosto dati spisak prevedimo u ove prološke članke

```
stud(milan,perić,8).
stud(jovan,delić,7).
stud(zoran,arsić,9).
stud(dušan,ilić,8).
stud(milan,jović,7)
```

Tada se trivijalno rešava opisani problem. Tako, pitanjem

```
?-stud(zoran,X,Y),write(X),tab(3),write(Y),fail.
```

se iz spiska "vade" svi studenti imena 'zoran'. Iako je uočen mali primer lako je sklopiti opštiju sliku.Naime,slična ideja se može uopšte koristiti ali ako je spisak duži[8] ta ideja postaje skoro neupotrebljiva, jer Prolog pretragu vrši linearno, tj.redom i to može trajati veoma dugo.Međutim, zanimljivo je da u Arity- prologu ima izvestan broj ugrađenih "pomagala" pomoću kojih se opisani problem može savladati sa dosta uspeha. U nastavku ukratko opisujemo ta "pomagala", odnosno odgovarajuće predikate. O tome upravo "govore" delovi 8.2, 8.3 i 8.4 ove tačke.Posebno ističemo da u vezi sa tim pomagalima Arity-prolog je u velikoj prednosti nad LPA-prologom. Takođe podvlačimo da je način,metoda ugrađena u Arity-prologu, može se reći, potpuno profesionalna.

---

[8] Recimo, nekoliko hiljada članova ili i više.

8.2 Svetovi i record-predikati (Arity-prolog)

U Arity-prologu se za pamćenje članaka programa a i opštije pamćenje nekih podataka dopušta korišćenje posebne memorije[9] koja u načelu može biti 1 gigabajt,što je oko 1000 megabajta (oko milijardu bajtova).Rekli smo u načelu jer ta količina zavisi od raspoložive memorije računara na kome radimo. Cela ta memorija se dogovorno deli u manje delove, tzv. "svetove" (u originalu "world"). Svaki svet ima veličinu od 4096 kilobajta[10], a ukupno može da bude 256 svetova. Svaki novi svet se na poseban način pravi, što ćemo ubrzo videti.Ako ne napravimo nijedan drugi onda "pri ulaženju u Arity-prolog" se nalazimo u osnovnom svetu, imena main[11]. Ako recimo ušavši u Arity-prolog postavimo pitanje ?-assert(stud(milan,peric,8)) onda se dodaje članak stud(milan,peric,8).ali, to je bitno, on se smešta u svet main, odnosno u fajlu API.IDB, kao jedan od "podataka". Slično, ako u Arity-prolog učitamo neku fajlu sa izvesnim programom P sa kojim dalje radimo razne stvari, onda ukoliko želiimo možemo taj program ubaciti u tu fajlu API.IDB. U tu svrhu, prosto postavimo pitanje

```
?-save.
```

i čitav zamišljeni program P, tj. njegovi članci su ubačeni u API.IDB. Međutim, ako isključimo računar, pa ga kasnije opet uključimo i odemo u Arity-prolog, onda svi članci programa P su već tu, jer Arity-prolog pri svom uključivanju učitava API.IDB. To praktično znači da mi možemo napraviti raznorazne korisničke predikate i sve ih "ugraditi" u Arity-prolog.

Pretpostavimo da sada hoćemo da pored osnovnog sveta[12] napravimo jedan drugi imena 'spisi'. To se može učiniti ovom pitanjem

```
?-create_world(spisi).
```

u kome posao pravljenja tog sveta obavi predikat create_world.To je za sada prvi predikat sa svetovima. Ima ih nekoliko:

(8.2.1)

```
data_world(Stari,Novi),
code_world(Stari,Novi),
what_worlds(X),
save,
save(Ime),
restore,
restore(Ime),
delete_world(Ime_sveta)
```

Pre podrobijeg objašnjenja, navedimo da pri radu sa podacima za stvari poput upis,ispis,brisanje,pretraga i dr. u okviru nekog sveta postoje razni predikati, i sada navodimo ove

---

[9] Prološki algoritam se po pravilu "odvija" u delu memorije koja se obično slobodnije zove "workspace" ("radilište"). Taj deo je 64 kilobajta.

[10] Svaki svet je izdeljen na tzv. "stranice" ("pages"),a svaka od njih sadrži 16 kilobajtova.

[11] Tako je u verziji 6.0, dok u ranijim je bio api. Inače,API.IDB je ime osnovne Arity-fajle koja pored ostalog i to "pamti".

[12] Njegovo ime je 'main' (mislimo na verziju 6.0 Arity-a).

(8.2.2)
```
recorda(Ime[13],Podatak,Ref_broj),
recordz(Ime,Podatak,Ref_broj),
record_after(Ref1,Podatak,Nov_ref),
recorded(Ime,Podatak,Ref),
instance(Ref,Podatak),
key(Ime,Ref),
keys(Promenljiva),
nref(Ref,Sledeci),
pref(Ref,Prethodni),
ref(dati_broj),
nth_ref(Ime,N,Ref),
replace(Ref,Podatak),
erase(Ref),
eraseall(Ime).
```

Sada navodimo neke primere.

Primer 8.2.1 Napravi dva sveta imena svet1,svet2 u prvi od njih redom staviti podatke

```
a(3)   a(6,7)          pod ključem[14] a
b(5)   b(88,99,77)     pod ključem b
```

a u drugi podatke

```
c(44,89)   c(7)        pod ključem c
d(123)     d(6)        pod ključem d
```

Oba ta sveta snimi u fajlu imena 'podatak'.

Rešenje. Daćemo postupan opis, premda se rešenje može dobiti postavljanjem jednog jedinog pitanja. To pitanje nastaje "sojedinjavanjem" svih dole izloženih malih pitanja. Prvo, pitanjem

```
?-create_world(svet1).
```

napravimo svet svet1, a pitanjem ?-create_world(svet2) svet svet2. Da bismo išta radili u nekom svetu "moramo ući u njega". Pitanjem[15]

```
?-data_world(main,svet1).
```

iz osnovnog Arity-sveta, tj. iz main-a prelazimo u svet1. A sada ćemo u njega upisati date podatke. Postavimo ovo pitanje

```
?-recordz(a,a(3),_),recordz(a,a(6,7),_).
```

i tako se pod ključem ,odnosno imenom a u svet svet1 upisuju podaci a(3) i a (6,7). Pošto je korišćen recordz-predikat uvek se dodavanje obavlja na

[13] Prvi argument umesto Ime možemo zvati i Ključ.

[14] Tj. imenom a.

[15] Mogli smo i ovako pitati ?-data_world(X,svet1). Tada bismo opet prešli u svet1, dok X bi imalo vrednost sveta odakle smo krenuli, tj. ovde: main. U vezi sa predikatom data_world pomenimo da se pitanjem oblika

```
?-data(X,X).
```

promenljiva X vezuje sa ime "tekućeg" sveta, tj. onog u kome smo trenutno. Još istaknimo da se umesto data_world predikata, na sličan način može koristiti code_world predikat.

kraj tj. prvo se doda a(3), a zatim iza njega a(6,7). Međutim, da smo umesto recordz koristili recorda predikat dodavanje svakog novog bi bilo uvek na početak pa bi redosled bi bio drukčiji. Inače u gornjem pitanju sa _ je označena promenljiva koja kao vrednost dobija tzv. referentni broj. Pri upisu ma kog podatka Prolog mu uvek pridruži takav broj. Na sličan način pod ključem b upisujemo date podatke. Postavimo pitanje

```
?-recordz(b,b(5),_),recordz(b,b(88,99,77),_)
```

i tako u svet1 unesemo te b-podatke. Sada ćemo izaći iz sveta svet1 preći u svet2. Postavimo pitanje

```
?-data_world(svet1,svet2).
```

Iza toga upisujemo c- i d- podatke u taj svet, tj. postavimo pitanje

```
?-recordz(c,c(44,89),_),recordz(c,c(7),_),
                recordz(d,d(123),_),recordz(d,d(6),_).
```

Postavimo pitanje ?-data_world(svet2,main). i tako pređemo u osnovni svet. Konačno sa ?-save(podatak) se obavi traženo snimanje. Istanimo da puno ime fajle glasi podatak.idb, tj. ima nastavak .idb. Kraj Primera 8.2.1

Uopšte u vezi sa ključevima, imenima dodajmo dodajemo sledeće

(8.2.3) Ključ, ime može biti neki "atom", tj. konstantska reč kao a,b,h7, pera; može biti neki prirodan broj kao 3,0,78 , kao i neki izraz oblika ime(_,_,...,_), u kakve dolaze na primer f(_), g(_,_).

Takvi poslednji se po pravilu uzimaju kao ključevi ukoliko podaci su neki članci. Primer: pitanjem

```
?-recordz(r(_,_),(r(X,Y):-s(X)),t(Y),_)
```

se u odgovarajući svet unosi članak r(X,Y):-s(X),t(Y) pod ključem r(_,_). Primetite da je članak stavljen u zagrade.

Primer 8.2.2. (Nastavak prethodnog). Kako, obratno, ući u Arity-prolog, učitati fajlu 'podatak.idb', saznati koje svetove ima, u svakom od njih saznati ključeve i odnosne podatke ?

Rešenje. Prvo ,nakon ulaska u Arity-prolog, postavimo pitanje

```
?-restore(podatak).
```

i tako se obavi rečeno učitavanje[16]. Pitanjem

```
?-what_worlds(X),write(X),nl,fail.
```

se dobije spisak svih raspoloživih svetova, odnosno ovde:

main, svet1 i svet2.

Primetite da je korišćem predikat: what_worlds. Dalje, sa

[16] Predikat restore, korišćen za učitavanje, ima svojih posebnosti. Tako, ako smo recimo sa [ime_fajle] već na običan način učitali jednu fajlu, a onda postavimo pitanje kao ?-restore(podatak) desiće se sledeće. Ta učitana fajla će biti sklonjena i umesto svega "na radnom mestu" će biti prisutna samo fajla podatak.idb (obavljena "restauracija"). Međutim, nakon

```
?-restore(podatak)
```

možemo na običan način dodati, učitati nove fajle.

```
?-data_world(main,svet1).
```

se ulazi u svet1. Pitanjem

```
?-keys(X),write(X),nl,fail.
```

možemo saznati sve ključeve, tj. ovde će to biti a i b. Primetite da je korišćen predikat: keys. Dalje, da bismo recimo izlistali sve a-podatke koristimo predikat recorded, odnosno postavljamo pitanje

```
?-recorded(a,X,_),write(X),nl,fail.
```

i na ekranu će se pojaviti

```
a(3)
a(6,7)
```

Preostala pitanja se postavljamo slično navedenim pa ih izostavljamo. Kraj Primera 8.2.2.

Preostalo je da objasnimo još neke od predikata iz spiskova(8.2.1),(8.2.2) Tako smisao predikata delete_world(Ime_sveta) je da, uz pretpostavku da smo u svetu imena Ime_sveta, obrišemo, uklonimo taj svet.
Sada prepostavimo da smo ušavši u Arity-prolog napravili svet imena spisak i u njega pod ključem 'per' redom stavili ove brojeve

```
11  22  33  44  55
```

Takođe pretpostavimo da se nalazimo u tom svetu. Tada se, recimo, listanje svih tih per-podataka može ostvariti ovim pitanjem

```
?-recorded(per,X,_),write(X),nl,fail.
```

Dalje, recimo treći član per-spiska se može ovako naći

```
?-nth_ref(per,3,X),instance(X,Y),write('To je '),write(Y).
```

Tu se koristi predikat nth_ref koji uz zadan ključ, ovde je to per, i zadan red, ovde je to 3, u per-spisku određuje referentni broj trećeg člana. Znači X nije treći član već njegov referentni broj. Međutim, koristi se i usluga instance predikata, pomoću koga se od referentnog broja nalazi odgovarajući podatak. Kako bi se iz per-spiska obrisao treći član ? Dosta je postaviti ovo pitanje

```
?-nth_ref(per,3,X),erase(X).
```

Znači, sada na scenu stupa erase predikat, koji služi za brisanje podatka zadanog referentnog broja. Inače, brisanje svih članova nekog ime-spiska se može obaviti sa ?-eraseall(ime), tj. uslugom eraseall predikata. Opet se vratimo prethodnom per-spisku, koji sada nakon brisanja trećeg člana glasi

```
11  22  44 55
```

Kako da se recimo drugi član zameni sa 2222 ? Može ovim pitanjem

```
?-nth_ref(per,2,X),replace(X,2222).
```

gde se koristi replace predikat. Novi per-spisak glasi

```
11 2222 44 55
```

Kako bismo u tom per-spisku iza drugog člana kao nov treći član umetnuli recimo ovaj podatak: 3333 ? To se može uslugom record_after predikata, od-

nosno postavimo ovo pitanje

```
?-nth_ref(per,2,X),record_after(X,3333,_).
```

Odvojimo se od tog primera i uopšte zamislimo da u odnosu na neki svet imamo izvestan spisak čiji ključ se zove 'ime'. Taj ime-spisak možemo ovako predstaviti

(8.2.4)

ime	Ref
podatak1	Ref1
podatak2	Ref2
podatak3	Ref3
....	
podatakk	Refk

Tu se redom nižu ime ključa i podaci i uporedo odgovarajući referentni brojevi. U Arity-prologu spisak oblika (8.2.4) je dvostruko povezana lista. To, slobodnije rečeno, znači da se po njoj možemo korak po korak kretati u oba smera. S tim u vezi su predikati

```
key, nref, pref
```

Naime, pitanjem oblika

```
?-key(ime,X).
```

se u stvari dobija referentni broj ključa ime, tj. samog početka liste. Predikat nref služi za "putovanje napred", a pref za "vraćanje nazad". Tako sa

```
?-key(ime,X),nref(X,Y),instance(X,Y).
```

promenljiva Y će biti vezana za prvi podatak. Sada navodimo jedan zanimljiv primer u kome se koristi zamisao dvostruke liste (8.2.4) i odgovarajućih predikata.

Primer 8.2.3. Uočimo program

```
upis:- create_world(direk),
       data_world(main,direk),
       directory('*.*',X,_,_,_,_),
       recordz(dir,X,_),
       fail.
upis:-save(direk).

ispis1:- restore(direk),
         data_world(main,direk),
         key(dir,Kljuc),
         nref(Kljuc,X),
         napred(Kljuc,X).

         napred(X,X):-!.
         napred(Kljuc,Ref):-instance(Ref,Podatak),write(Podatak),nl,
                            nref(Ref,Nref),napred(Kljuc,Nref).

ispis2:-restore(direk),
         data_world(main,direk),
         key(dir,Kljuc),
         pref(Kljuc,X),
         nazad(Kljuc,X).
```

```
nazad(X,X):-!.
nazad(Kljuc,Ref):-instance(Ref,Podatak),write(Podatak),nl,
                  pref(Ref,Pref), nazad(Kljuc,Pref).
```

u kome pored ostalih se pojavljuje osnovni proloski predikat directory. Recimo, na pitanje

```
?-directory('c:\*.*',X,_,_,_,_),write(X),tab(2),fail.
```

se pojavljuje spisak svih fajli koje se nalaze na disku c, u njegovoj direktoriji na vrhu. Ako se u vezi sa tim programom postavi pitanje ?-upis. onda se najpre napravi svet imena direk, dalje pređe se u taj svet, a onda uslugom predikata directory se iz tekuće direktorije[17] "vade" imena fajli. Ta imena se pomoću recordz predikata pod ključem dir ubacuju u svet direk. Zbog fail-a to "traje dok može", a onda prelazimo na drugu upis-granu i tom prilikom snimimo svet direk u fajlu istog imena (u stvari, njeno ime je sa završetkom .idb). Pretpostavimo sada da se nakon pravljenja te direk .idb postavi pitanje

```
?-ispis1.
```

Tada se uslugom restore predikata najpre učita fajla direk.idb, a dalje pomoću data_world(main,direk) predemo u svet direk. Iza toga formulom key (dir,Kljuc) saznamo referentni broj ključa. Ako zamišljamo sliku kao(8.2.4) tako smo odredili Ref. Formulom nref(Kljuc,X)promenljiva X se veže za Ref1, tj. referentni broj prvog podatka, što će u našem slučaju biti ime prve fajle. Dobivši taj Ref1 upošljavano napred-članke. Pomoću drugog od njih redom "vadimo" i štampamo podatak po podatak. Predikat nref nas stalno vodi napred. Dokle? Kad u formuli nref(Ref,Nref) promenljiva Ref, prema slici (8.2.4), dobije vrednost Refk, onda

> Nref dobije vrednost "desnjaka" od Refk, a pošto je po sredi dvostruka lista(u stvari i kružna) to nova vrednost za Nref je upravo referentni broj polaznog ključa.

Sledstveno, upošljava se prvi ispis1-članak i algoritam se završava. Na kraju izloženog primera istaknimo da se i na pitanje ?-ispis2 dešava nešto slično, s tim što sada ispis teče od poslednjeg prema prvom podatku. Naravno sada se koristi pref predikat.

Napomena 8.2.1. Upoznali smo razne record-predikate. Svi primeri koje smo imali su u velikoj meri opterećeni semantikom, odnosno pitanje je šta uopšte smeju da budu argumenti tih predikata. Recimo, da li je ispravno postaviti ovo pitanje

```
?-recorda(pera,mika,_).
```

odnosno pitanje ?-recordz(pera,mika,_).

jer tu je dosta sporna semantika: pod ključem, imenom 'pera' stoji podatak 'mika'. Odgovor je DA. Naime, uopšte u pitanju oblika

```
?-recorda(Ime,Podatak,_).
```

odnosno ?-recordz(Ime,Podatak,_).

Ime i Podatak ne moraju da imaju "neku unutrašnju vezu"; kratko Ime mora da zadovolji uslov (8.2.3), a Podatak mora biti neki ispravan proloski izraz.

---

[17] One u kojoj je Arity-prolog trenutno kotišćen.

8.3 B-drveta (Arity-prolog)

Do sada opisani predikati, kao recorda, recordz i dr., nisu podesni za rad sa velikim spiskovima, jer oni u osnovi koriste "linearno hodanje", sekvencionalnu pretragu. Taj problem se u naše vreme po pravilu rešava korišćenjem uravnoteženih, sređenih odnosno tzv. B-drveta[18]. I veoma je značajno da su takva drveta ugrađena u Arity-prolog.

Najpre uočavamo spisak (8.1.1) i na njemu upoznajemo neke opšte činjenice. U vezi sa tim spiskom mogu se napraviti razna B-drveta, u zavisnosti od ključa(imena): po imenu, po prezimenu, po oceni. Evo kako se pomoću Arity-prologa pravi takvo drvo po ključu ime. Koristi se sistemski predikat

```
recordb(ime_drveta,ključ,podatak)
```

kod koga prvi argument je ime drveta, recimo ovde neka bude 'ime', drugi je ključ, ovde dogovorno na tom mestu će stajati reči milan, jovan, zoran, dušan. Treći argument je nazvan podatak, on će zapravo na svoj način ucelo zapisati pojedine članove spiska. Ovde će to na primer biti

```
stud(milan,perić,8)
```

Shodno rečenom to B-drvo se može uvesti postavljanjem ovih pitanja

```
?-recordb(ime,milan,stud(milan,perić,8)).
?-recordb(ime,jovan,stud(jovan,delić,7)).
?-recordb(ime,zoran,stud(zoran,arsić,9)).
?-recordb(ime,dušan,stud(dušan,ilić,8)).
```

Evo kako bismo mogli da zamislimo izgled tog B-drveta:

Obrazloženje:
Prvo je upisan milan,... gde tačkice zamenjuju podatak stud(milan,perić,8).
Drugo, zbog jovan<milan je na levu raklju stavljen jovan,....
Treće, pošto zoran>milan to na desnu raklju vrha je stavljen zoran,...
Četvrto, pri upisu dušana gledajući drvo ovako rasuđujemo. Pošto dušan< milan idemo na levu granu. Dalje, pošto dušan<jovan to idemo na levu granu jovana i tu stavljamo dušan,...

Kao što se vidi, B-drvo se pravi u odnosu na vrednosti ključa, koristeći leksikografski poredak. Kasnije po želji možemo na sličan način dodati nekog novog člana, tj. nov "podatak". Kratko rečeno za dodavanje je zadužen recordb-predikat. Predikat

```
retrieveb(Ime_drveta,ključ,Podatak)
```

je "zadužen" za traženje celog podatka na osnovu imena drveta i ključa. Tako, sa ?-retrieveb(ime,jovan,X). X će biti vezano sa stud(jovan,delić,7). Međutim sa ?-retrieveb(ime,X,Y), write(X),tab(3),write(Y),nl,fail. će se štampati čitav student-spisak i to leksikografski sređen po imenima. Za

---

[18] B dolazi od "balansiran", tj. uravnotežen, sređen. Takva drveta smo koristili i ranije, videti Zadatke 3.44 i 3.45.

brisanje nekog podatka se koristi predikat

```
removeb(ime_drveta,ključ,podatak)
```

Recimo sa ?-removeb(ime,jovan,X). će iz B-drveta biti izbačen jovan i odnosni podaci o njemu. Za brisanje celog B-drveta se koristi predikat

```
removeallb(ime_drveta).
```

Pre upoznavanja daljih dubljih činjenica o B-drvetima navedimo da su B-predikati "saglasni" sa raznim predikatima o svetovima, ali nisu saglasni sa predikatima recorda,recordz,nref,pref i dr. Naime, umesto tih predikata za rad sa B-drvetima moraju se koristiti B-predikati. S druge strane "sa svetovima" nema prepreka. Znači, na - u 8.2 opisan - način možemo graditi svetove, "šetati se" po njima, u okviru ma kog od njih definisati po neko B-drvo, i uz to pomoću save predikata obaviti zapamćivanje, snimanje ili u osnovnu API.IDB fajlu[19] ili u neku fajlu željenog imena.

Sada nastavljamo o B-drvetima sa više ključeva. Naime, u praksi se pojavljuje potreba za takvim drvetima. Recimo, prethodno smo u vezi sa spiskom (8.1.1) napravili B-drvo po ključu ime. Ali,u vezi sa istim spiskom možemo napraviti još dva drveta, imena 'prezime' i 'ocena' u kojima će ključevi biti prezimena, odnosno ocene. U tu svrhu je dosta postaviti pitanje:

```
?-recordb(prezime,peric,stud(milan,peric,8)),
  recordb(prezime,delic,stud(jovan,delic,7)),
  recordb(prezime,arsic,stud(zoran,arsic,9)),
  recordb(prezime,ilic,stud(dusan,ilic,8)),
      recordb(ocena,8,stud(milan,peric,8)),
      recordb(ocena,7,stud(jovan,delic,7)),
      recordb(ocena,9,stud(zoran,arsic,9)),
      recordb(ocena,8,stud(dusan,ilic,8)).
```

Imajući sva tri drveta možemo dosta lako raditi po svakom od ključeva.Primera radi, premda sa ocenom 8 imamo dva studenta, na pitanje

```
?-retrieveb(ocena,8,X),write(X),nl,fail.
```

će se ispravno pojaviti oba !Razlog: Arity prolog je tako dobro napravljen da dopušta da nekoj vrednosti ključa odgovara više, odnosno lista vrednosti.Međutim, očigledno je da u opštem slučaju opisana ideja može biti memorijski previše "skupa", jer uvek se u celosti pamti treći podatak. Kako se toga rešiti? Jedna mudra ideja je da se umesto pojedinih podataka,kao treći podatak uzme po jedan broj, zamišljajući uz to da je svakom od tih brojeva na određen način pridružen po jedan od podataka. Znači, takvi brojevi služe kao "posrednici". Značajno je da njih ne moramo sami izmišljati već uz korišćenje record-predikata upotrebiti odgovarajuće referentne brojeve. Ta ideja je ostvarena u programu koji sledi

```
(8.3.1)
radi:-
    write('Za dodati kucati:    dod. '),nl,
    write('Za naci po imenu kucati:    ime. '),nl,
    write('Za naci po broju kucati:    broj. '),nl,
    write('Za listati celo ime-drvo, kucati:  list_ime.'),nl,
    write('Za listati celo broj_drvo, kucati: list_broj'),nl,
    write('Za zavrsetak kucati:    kraj. '), nl,
    read(X), cini(X).

cini(dod):-write('Ime '),read(Ime),
          write('Broj '),read(Broj),
          recordz(stud,stud(Ime,Broj),Ref),
          recordb(ime,Ime,Ref),
          recordb(broj,Broj,Ref),
          radi.
cini(ime):-write('Daj ime   '),read(Ime),
          not
           (retrieveb(ime,Ime,Ref),recorded(stud,stud(X,Y),Ref),
           write('Evo '),write(X),tab(5),write(Y),nl,fail),
           radi.
cini(broj):-write('Daj broj  '),read(Broj),
          not
            (retrieveb(broj,Broj,Ref),recorded(stud,stud(X,Y),Ref),
            write('Evo '),write(Y),tab(5),write(X),nl,fail),
            radi.
cini(list_ime):-not
            (retrieveb(ime,_,Ref),recorded(stud,stud(X,Y),Ref),
            write('To su '),write(X),tab(5),write(Y),nl,fail),
            radi.
cini(list_broj):-not
             (retrieveb(broj,_,Ref),recorded(stud,stud(X,Y),Ref),
             write(' To su '),write(X),tab(5),write(Y),nl,fail),
             radi.
cini(kraj):-save(spisak).
```

Taj program se odnosi na građenje i korišćenje spiska studenata sa dva ključa: ime, broj.Program se pokreće pitanjem ?-radi.Tada se na ekranu pojavi ponuda od nekoliko mogućnosti. Recimo, da smo kucali

```
dod.
```

što znači da želimo spisku da dodamo novog člana.Tada, prema članku

```
cini(dod)
```

prvo kucamo Ime, zatim Broj i onda se redom računaju ove formule:

```
recordz(stud,stud(Ime,Broj),Ref)
```
znači pod ključem stud se upiše podatak stud (Ime,Broj) i sam Prolog nam da referentni broj Ref

```
recordb(ime,Ime,Ref),
recordb(broj,Broj,Ref)
```
gradimo ime-drvo i broj-drvo, kojima kao treći podatak damo "prispeli" Ref.

Nakon računanja tih formula opet se računa formula radi. I tako možemo po želji vršiti upis više studenata. Recimo, možemo usput zaželeti da nađemo sve studente datog imena.Tada, pri računu formule radi kucamo reč:

```
ime.
```

Tada na scenu stupa formula cini(ime). Na ekranu se postavi pitanje

```
Daj ime
```

[19] To se postiže pitanjem ?-save.

i tada ako recimo odgovorimo sa 'Pera'. u odgovoru cemo dobiti spisak svih studenata imena Pera, ako ih ima, sa odgovarajucim brojevima. Program je takode sposoban da da

listu svih studenata datog broja
listu svih članova ime-drvo ,kao i članova broj-drvo.

Ako pri radi-pitanju damo odgovor kraj. čitav se napravljeni spisak sa drvetima snima u fajlu imena: spisak.Ako zamislite da smo pre izvršenja programa (8.3.1) najpre napravili jedan svet, recimo student , dalje da smo u njega ušli , pa tek onda "ujključili" program (8.3.1), onda nakon pitanja ?-save(spisak) fajla spisak.idb će sadržati i taj podatak o svetu student. U skladu sa tim, ako nakon izlaska iz Arity-a i ponovnog vraćanja u njega postavimo pitanje

```
?-restore(spisak).
```

čitava fajla spisak.idb će se učitati. Dalje sa

```
?-data_world(main,student).
```

prelazimo u svet student i onda smo u prilici da koristimo već napravljeni spisak studenata i to korišćenje obavljamo postavljanjem pitanja

```
?-radi.
```

Kao što vidite pojavljuje se jedna "mala", ali važna stvar, koju do sada nismo dovoljno isticali. Naime, kad smo pitali

```
?-save(spisak).
```

onda i svi članci tekućeg programa su "ušli" u spisak.idb.

Završavajući izlaganje o B-predikatima navodimo da Arity-prolog ima još neke takve predikate.Prvo govorimo o predikatima

```
betweenb, betweenkeysb,
```

Pomoću njih, grubo rečeno, ako su u nekom B-drvetu data dva podatka lako možemo saznati sve "međupodatke",što više objašnjavamo na primeru:

Pretpostavimo da smo pitanjima

```
?-recordb(lice,pera,lice(pera,56)).
?-recordb(lice,aca,lice(aca,78)).
?-recordb(lice,mile,lice(mile,230)).
?_recordb(lice,jova,lice(jova,75)).
```

definisali B-drvo imena 'lice'. Postavimo ovo pitanje:
Koji su sve podaci između jove i pera ?
gde 'između' podrazumeva i: 'uključno i njih dva'. Za takvo pitanje je "nadležan" betweenb-predikat.Pitamo

```
?-betweenb(lice,jova,pera,=,=,Clan,Podatak),write(Clan),
                tab(3),write(Podatak),fail.
```

i dobićemo sve tražene međupodatke:

```
jova  lice(jova,75)
mile  lice(mile,230)
pera  lice(pera,56)
```

U opštem slučaju betweenb-formula izgleda

```
betweenb(ime_drveta,clan1,clan2,relacija1,relacija2,Clan,Podatak)
```

gde se određuju Clan i Podatak, a zadani su
ime_drveta,

clan1 i clan2
relacija1, relacija2. Taj par može biti jedan od ovih

```
=, =     ("uključno između")
=, <     (bez "desnog kraja",tj. Clan ne može dobiti i vred-
          nost clan2)
>, =
>, <
```

Evo sada primera za betweenkeys-predikat: Na pitanje

```
?-betweenkeysb(lice,a,m,Clan),write(Clan),tab(3),fail.
```

će se štampati svi Clan-ovi koji su uključno između reči 'a', 'm', tj. štampaće se

```
adam  jova mile
```

Uopšte betweenkeysb-formula izgleda

```
betweenkeysb(ime_drveta,leva_granica,desna_granica,Clan)
```

i njome se izražunava Clan (datog drveta, između datih granica).

Upoznati skup B-predikata još nije potpun, Arity prolog pruža još neke sjajne mogućnosti. Tako, on ima i ove B-predikate

```
replace
what_btrees
btree_count
defineb
```

Prva tri su dosta jednostavna:

replace služi za zamenu nekog podatka drugim i javlja se u obliku

```
replaceb(ime_drveta,clan,stari_podatak,Nov_podatak);
```

what_btrees korišćen u obliku ?-what_btress(X),write(X),fail. može dati spisak imena svih uvedenih B-drveta.
btree_count korišćen u obliku btree_count(ime_drveta,Broj_ključeva) može dati broj "ključeva" ("članova") u B-drvetu.Recimo, u prethodnom primeru podaci su oblika lice (ime,brojka), tj. lice se odnosi na broj argumenata("ključeva").Za drvo imena lice Broj_ključeva iznosi 1+broj_argumenata, tj. 3.

Međutim predikat defineb ("definiši B-drvo") je prilično suptilan i opisusujemo ga ukratko.Naime,u praksi kad radimo sa velikim brojem podataka čak i pri korišćenju B-drveta u načelu bi mogle da se pojave teškoće; recimo ukoliko B-drvo je po obliku blisko jednoj grani, tj.drugim rečima nije dovoljno "krošnjasto", kaže se i nije dobro uravnoteženo. I upravo predikat defineb je pored ostalog "zadužen" da otkloni taj nedostatak drveta.

Napomena 8.3.1. Ova napomena je na neki način nastavak Napomene 8.2.1. Naime, kao što smo videli neko B-drvo se uvodi pomoću recordb-predikata. Da li tada mora da bude prisutna cela semantika kakva se nameće u raznim primerima iz prakse. Odgovor je: NE MORA.Recimo, ispravna su pitanja

```
?-recordb(pera,f(7,8),g(9,7,8)).
?-recordb(d(_,_),mile,jova).
```

Uopšte pitanje oblika

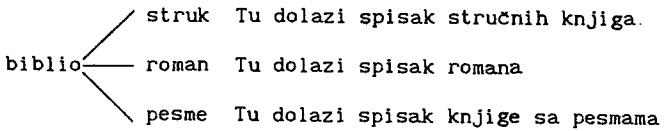
```
?-recordb(ime,izraz1,izraz2).
```

je ispravno ukoliko 'ime' je dobro ime (u smislu (8.2.3)), izraz1 i izraz2 su neki ispravni prološki izrazi. B-drvo imena 'ime' se gradi u odnosu na izraz1.

8.4 Hash tabele (Arity-prolog)

U vezi sa opštim pitanjem spiskova poznat je i način pomoću tzv. hash-tabela. Arity-prolog ima ugrađena sredstva za rad sa njima. Najpre uočimo jedan primer. Pretpostavimo da hoćemo da sredimo neku biblioteku tako da u posebne grupe dođu stručne knjige, romani, pesme. To nam stvara predstavu jednog ovakvog "vodoravnog" drveta

(8.4.1)



Ime tog drveta je 'biblio', na vrhu imamo 3-raklju, tj. 3 grane sa imenima 'struk', 'roman', 'pesme'. A dalje svaka od tih grana ima po izvestan broj raklji, prema broju odgovarajućih knjiga. Izloženi primer je upravo ilustracija zamisli "hash-tabela". Za definisanje i rad sa takvim tabelama u Arity-prologu postoje razni predikati kao

recordh, retrieveh, removeh, removeallh

Recimo pitanjem oblika

```
?-recordh(biblio,struk,k('Matematika 1','Milan Jović',567).
```

se Arity-prologu saopštava da hoćemo da imamo hash-tabelu sa imenom 'biblio', u kojoj je jedan od 'ključeva' struk i najzad sa

```
k('Matematika 1','Milan Jović', 567)
```

smo dogovorno zapisali

"knjigu Matematika 1 pisca Milana Jovića čija cena je 45 dinara "

Dalje na sličan način nastavimo upisivanje te tabele:

```
?-recordh(biblio,struk,k('Fizika','Zoran Antić',62).
?-recordh(biblio,roman,k('Rat i mir','Lav Tolstoj',75).
?-recordh(biblio,pesme,k('Tražim pomilovanje','Desanka Maksimović',47).
?-recordh(biblio,roman,k('Vreme smrti','Dobrica Ćosić',70).
```

itd.

Ako hoćemo sve te podatke da sačuvamo u nekoj fajli, onda kao i do sada koristimo save-predikat, odnosno restore za učitavanje. Evo sada nekoliko pitanja u vezi sa tako napravljeno hash-tabelom:

```
?-retriveh(biblio,struk,k('Fizika',X,Y)
```

Pitamo ko je pisac i koja je cena knjige 'Fizika'

```
?-retrieveh(biblio,struk,X),write(X),nl,fail.
```

Tražimo spisak svih stručnih knjiga

```
?-retriveh(biblio,X,Y),write(X),tab(2),write(Y),nl,fail.
```

Tražimo spisak svih knjiga biblioteke

```
?-retrieveh(biblio,X,k(Y,Z,62)).
```


Tražimo koja knjiga staje 62 dinara.

Primetite da retrieveh predikat ima tri argumenta i da prvi, ovde biblio, mora biti zadan. Naime, možemo jednovremeno praviti i više hash-tabela, svaka će imati svoje ime, i pri radu sa ma kojom od njih to ime mora biti navedeno. Znači besmisleno je pitanje oblika ?-retrieveh(X,Y,Z). Međutim, taj se problem može "ublažiti" uz pomoć keys-predikata. Naime:

(8.4.2) Pitanjem oblika ?-keys(X),write(X),nl,fail. se dobija spisak imena svih hash-tabela, B-drveta, i record-zapisa već uvedenih u dotičnom trenutku[20].

U programu (8.3.1) videli smo kako se B-drveta mogu mudro koristiti za rad sa više-ključnim podacima. Na sličan način se mogu koristiti i hash-tabele, iako to u odnosu na B-drveta može biti "vremenski skuplje". Zamisao je slična kao u rečenom programu, pa ćemo stoga izneti samo neke "isečke". Pretpostavimo da imamo ovaj spisak

```
ime prezime ocena
pera janić  8
mile tasić  9
ana  lukić  8
mila ilić   6
```

Tada pitanjima koja slede pravimo tri hash-tabele imena 'ime','prezime' i 'ocena' i kao u (8.3.1) bitno se pomažemo referentnim brojevima:

Prvo:
```
?-recordz(student,stud(pera,janic,8),Refnum),
      recordh(ime,pera,Refnum),recordh(prezime,janić,Refnum),
            recordh(ocena,8,Refnum).
```

Drugo:
```
?-recordz(student,stud(mile,tasić,9),Refnum),
      recordh(ime,mile,Refnum),recordh(prezime,tasić,Refnum),
            recordh(ocena,9,Refnum).
```

Treće:
```
?-recordz(student,stud(ana,lukić,8),Refnum),
      recordh(ime,ana,Refnum),recordh(prezime,lukić,Refnum),
            recordh(ocena,8,Refnum).
```

Četvrto:
```
?-recordz(student,stud(mila,ilić,6),Refnum),
      recordh(ime,mila,Refnum),recordh(prezime,ilić,Refnum),
            recordh(ocena,6,Refnum).
```

Recimo da onda želimo da saznamo sve podatke o studentu imena'ana'. Pitamo:

```
?-retrieveh(ime,ana,Refnum), instance(Refnum,X),write(X).
```

Kao što se vidi kao i u programu(8.3.1) koriste se razni record-predikati.

Napomena 8.4.1. Ta napomena je u stvari nastavak Napomene 8.3.1. Naime, kako da se "formalizuje" pojam hash-predikata recordh, tj. da se oslobodi od uobičajene prateće semantike ? Kratko rečeno, kao i u slučaju B-drveta ispravna je svaka formula oblika

recordh(ime,izraz1,izraz2)

gde 'ime' zadovoljava uslov (8.2.3) a izraz1,izraz2 su ispravni prološki izrazi. Za razliku od B-drveta u hash-tabelama se pojedini sastavci zapisuju po redu, "linijski", dakle ne pomoću nekog drveta.

---

[20] Ako smo u nekom svetu ,onda se to odnosi na one koji su u njemu.

9 IZVRŠNE FAJLE u PROLOGU

Zanimljivo je da, izuzev Micro-prologa, i u LPA- i u Arity-prologu se mogu praviti "samostalno izvršive" fajle, odnosno one koje se izvršavaju bez ulaženja u Prolog, jednostavno kucanjem imena fajle. Uopšte, postoje dve vrste takvih fajli: sa nastavkom .EXE odnosno nastavkom[1] .COM . U Arity se pojavljujuju .EXE dok u LPA .COM fajle. Slučaj svakog od tih jezika razmatramo posebno.

9.1 Slučaj Arity-prologa

Počinjemo jednim primerom. Pretpostavimo da fajlu 'pera. ari' čini ovaj program

```
f(1).
f(2).
ajde:-f(X),write(X),nl,fail.
ajde:-write('Kraj').
```

i da želimo da od nje napravimo izvršnu fajlu, koja je sposobna da odgovori na pitanje ?-ajde. Evo korak za korakom šta u tom primeru treba raditi:

Prvo, .EXE fajla mora da ima jedan glavni predikat imena 'main', i zadatak .EXE fajle je upravo da "proloski izračuna" taj main, tj. da da odgovor na pitanje ?-main. Shodno rečenom, prethodnu fajlu 'pera.ari' ovako promenimo, dopunimo

```
:-public main/0.
main:-ajde.
f(1).
f(2).
ajde:-f(X),write(X),nl,fail.
ajde:-write('Kraj').
```

Kao što vidite dodali smo dva nova reda, prvi i drugi. To je inače opšta zamisao. Naime, da smo imali ma koji program P sa nekim pitanjem ?-ϕ. onda bismo slično prethodnom fajli programa P dodali ova dva reda

(9.1.1)
```
:-public main/0.
main:-φ.
```

Drugo, od nove fajle 'pera.ari' treba da napravimo izvesne .obj fajle, videćemo koje, i zatim upotrebom ma kog od standardnih "linkera" (povezivača) da te objekt-fajle sa još nekim (videćemo kojim) povežemo i konačno kao plod dobijemo izvršnu fajlu 'pera.exe'. Ti koraci teku ovako:

Pravljenje odgovarajućih objekt fajli (tj. kompajliranje, prevođenje):

Iz[2] DOSa sa tastature kucamo

(9.1.2)
```
>apc pera, , /n
```

gde je > znak DOSa, koji mi ne kucamo.

Tu je korišćena fajla apc.exe ('Arity prolog kompajler'). Za sada ne više o tome. Linkovanje:

Pretpostavimo da računar ima neki od standardnih linkera, kao MS i sl. Tada iz DOSa sa tastature kucamo

```
>link
```

Pojaviće se nekoliko pitanja na koje treba da ukucamo odgovor. Navodimo red po red na levoj strani pitanje, a na desnoj ono što smo mi ukucali, tj. odgovorili

(9.1.3)
```
Object Modules[3]   :   code+pera
Run file : pera
Libraries:  arity
```

Znači, prvo smo kucali code+pera, a smisao toga je da se povežu fajle code.obj i pera.obj. Pomenimo da je code.obj osnovna objekt-fajla Arity-prologa ("njegovo srce"). U trećem redu smo kucali arity kao biblioteku ("library"). Smisao je: i ta bibliotečka fajla[4] treba da "uđe u proces povezivanja". A u drugom redu smo kucali pera, što znači da će se izvršna fajla upravo zvati 'pera.exe'. Time je završen posao pravljenja tražene .EXE fajle. Ako sada iz DOSa kucamo

```
>pera
```

na ekranu će se pojaviti odgovor na pitanje ?-main. tj.:

```
1
2
Kraj
```

Taj primer smo namerno objasnili vrlo kratko, prosto kao "mustru za rad". Nažalost, pomoću te mustre se ne može "uegziti" svaka .ari fajla, odnosno ne mogu one fajle koje sadrže bar jedan od ovih predikata

(9.1.4) call, not, setof, bagof, findall, stdin, stdout, stdinuot

što ćemo u nastavku objasniti podrobnije.

Pogledajmo sada koje su fajle oblika 'pera,*' napravljene. Tu su naravno 'pera.ari' i 'pera.exe'. Medutim, ako iz DOSa pitamo

```
>dir pera.*
```

videćemo da sa imenom 'pera' imamo i ove fajle pera.obj i pera.idb. Jasno je pojavljivanje prve , ali kakva je to pera.idb ? Možete proveriti da za rad fajle 'pera.exe' neophodno je prisustvo fajle 'pera.idb', odnosno eventualnim brisanjem poslednje fajle fajla 'pera.exe' prestaje da bude izvršna. U stvari slično uopšte važi za sam Arity prolog. On pored fajle api.exe ima i fajlu api.idb, neophodnu za rad Arity prologa. Ta .idb fajla je zadužena za pamćenje raznih "podataka" Prologa, kao što su imena, reči

---

[1] Takve fajle su kraće, do 64 kilobajta.

[2] Pretpostavka je da nam je MS-DOS operativni sistem na računaru.

[3] U stvari se pojavi nešto duži tekst, ali to nije bitno.

[4] To je osnovna bibliotečka fajla Arity prologa. Njeno puno ime je arity.lib gde .lib ukazuje da je reč o bibliotečkoj fajli.

koje odgovaraju u Arity prolog ugrađenim predikatima. Tako tu su reči

'write', 'nl', 'fail', 'not', itd.

Kad smo maločas napravili 'pera.exe', kao što videsmo, napravljena je i fajla 'pera.idb'. Nije teško videti da je fajla 'pera.idb' u stvari identična fajli 'api.idb'. To praktično znači da smemo umesto pera.idb uzeti api.idb. Recimo, ako prvo obrišemo 'pera.idb' a potom sa

```
>copy api.idb pera.idb
```

napravimo novu fajlu pera.idb videće se da će fajla 'pera.exe' opet biti izvršna. Međutim, u opštem slučaju fajla .idb ne mora biti jednaka sa fajlom api.idb. Naime, ako fajla koju želimo da "uegzimo" sadrži razne predikate o kojima smo govorili u tački 8, delovi 2,3,4 onda odgovarajuća .idb fajla će zapamtiti sve takve korišćene predikate. O .idb fajlama u takvom slučaju važi sve slično onom već opisanom u tački 8. Tako tamo,smo pomenuli da .idb fajla maksimalno može biti velika 1 gigabajt(oko milijardu bajtova). Sada se vratimo komandi (9.1.1), odnosno

```
>apc pera, , /n
```

Ona je u stvari skraćenica za ovu komandu

```
>apc pera,pera,pera/n
```

gde

prva reč pera, znači da je reč o fajli 'pera.ari',
druga reč pera da želimo da se napravljena objekt-fajla zove 'pera.obj',
a treća reč pera da želimo da fajla podataka se zove pera.idb. Uz to /n znači da je to potpuno nova fajla koju znači tek treba napraviti.

Recimo, nakon gornje komande možemo staviti ovu novu komandu

```
>apc pera,mile,pera
```

i onda će se napraviti fajla imena mile.obj (ona će u stvari biti ista kao već napravljena pera.obj), a kao .idb fajla koristiće se već postojeća[5] fajla pera.idb . Kao što se vidi u izboru imena .obj i .idb fajli imamo potpunu slobodu.

Vratimo se opštem pitanju pravljenja .exe fajle za datu .ari fajlu. Znači sada pretpostavljamo da data fajla sadrži neki od predikata (9.1.4).Recimo neka fajla 'mile.ari' ima ovaj sadržaj

```
:-public  main/0.
main:-ajde.
f(1).
f(2).
ajde:-setof(X,f(X),L),write(L).
```

Sada zbog predikata setof koji se odnosi na f-članke taj program dopunjujemo ovim dodatkom

(9.1.5) `:-visible f/1.`

Nova fajla 'mile.ari' sada glasi

[5] Budući da je postojeća, to na kraju komande ne stoji /n.

(9.1.6)
```
:-public main/0.
:-visible f/1.
:-main:-ajde.
f(1).
f(2).
ajde:-setof(X,f(X),L),write(L).
```

Pravljenje .obj fajle i linkovanje se obavlja kao i u već izloženom prvom primeru; kratko, u vezi sa tim delovima nema nikakvih promena. Kao što se vidi u fajli 'mile.ari' za razliku od fajle 'pera.ari' prisutan je dodatak (9.1.5). To je u stvari opšta zamisao:

Ako fajla koju želimo da "uegzimo" sadrži neki od predikata (9.1.4) i u njihovim formula u pojedinim člancima u ukupnosti učestvuju predikati[6]

p1 /ar1, p2/ ar2, ..., pk/ ark

onda tu fajlu treba dopuniti i ovim visible-formulama:[7]

```
:-visible(p1 /ar1).
....
:-visible(p2 /ark).
```

Tako smo za sada upoznali opšti način građenja .EXE fajle u Arity-prologu. Kratko rečeno:

Prvo uvedemo centralni predikat main, zatim proglasimo za visible one predikate koji su "pod nekim" od predikata (9.1.4), nakon toga pomoću apc.exe i link.exe na već opisani način završimo građenje odgovarajuće .EXE fajle.

Na kraju dodajmo još dve stvari:

Prvo, umesto korišćenja visible-predikata možemo , ali samo jednom, upotrebiti reconsult-predikat. Naime, u fajli sadržaja:

```
....
reconsult('jova.ari'),
...
```

gde tačkice znače nenavedene delove i gde se samo na jednom mestu pojavljuje reconsult svi predikati koji učestvuju u fajli 'jova.ari'su implicitno proglašeni kao v i s i b l e.

Primer:

Da bismo "uegzili" fajlu 'mile.ari' umesto korišćenja visible-predikata (vid. (9.1.6)) možemo i ovako postupiti. Napravimo jednu pomoćnu fajlu 'jova.ari' sa ovim sadržajem

```
f(1).
f(2).
```

i onda umesto (9.1.6) kao novu 'mile.ari' uzmemo fajlu sa ovim sadržajem

[6] Tu su navedena imena predikata, i njihove "arnosti".

[7] Recimo, ako se među člancima pojavljuje i ovaj
`p(X):-not(q(X)),call(r(X,Y)).`
onda među visible-dodacima će biti i ova dva:
`:-visible(q/1).` `:-visible(r/2).`

```
:-public main /0.
reconsult('jova.ari').
main:-ajde.
ajde:-setof(X,f(X),L),write(L).
```

Drugo, čitaoci koji makar delom poznaju asemblerski jezik znaju da u načelu možemo napraviti više .obj fajli, recimo faj1.obj,...,fajs.obj i onda ih sve zajedno povezati. To pored ostalog podrazumeva upotrebu 'public' i 'extrn' "odrednica". Ukratko rečeno

> Procedura (u Prologu je to "predikat") se naznačava 'public'u onoj fajli gde se ona definiše, odnosno naznačava se 'extrn' ukoliko se pojavi, odnosno koristi i u nekoj drugoj fajli. Neka procedura može biti i bez tih odrednica; to je ako se ona koristi kao pomoćna, kao "lokalna" u nekoj fajli, a u ostalim se nigde ne pominje.

Sada navodimo jedan primer. Neka faj1.ari ima ovaj sadržaj

```
:-public main/0.
:-extrn oskup /2.
main:-write(' Dajte listu '),read(X),oskup(X,Y),
      write(' Evo nje bez ponavljanja istih clanova '),write(Y).
```

a faj2.ari sadržaj

```
:-public oskup /2.
oskup([],[]).
oskup([A|B],C):-elem(A,B),oskup(B,C).
oskup([A|B],[A|C]):-oskup(B,C).
elem(X,[X|Y]).
elem(X,[U|V]):-elem(X,V).
```

Znači, faj2.ari sadrži dva predikata oskup i elem. Pošto smo namerni da oskup "izvezemo" to smo ga proglasili za 'public', a o elem nije ništa rečeno jer taj predikat je samo pomoćni, lokalni u toj fajli. Inače, jasno je da je elem uobičajen predikat za "biti član od", dok oskup uz njegovu uslugu iz date liste "izbacuje ponavljajuće članove". Recimo, u pitanju

```
?- oskup([2,3,2,4,1],X).
```

X se vezuje za listu [2,3,4,1].
U faj1.ari se koristi oskup predikat, pa je on tu naznačen kao 'extrn'. Evo ukratko kako od te dve fajle pravimo .exe fajlu[8] imena faj1.exe:

```
>apc faj1, , /n
>apc faj2, , /n
```

a onda obavimo linkovanje pri čemu, što je bitno, na prvo pitanje (OBJECT MODULES) damo odgovor

```
code+faj1+faj2
```

na drugo(RUN FILE) upišemo faj1, a na treće (LIBRAIRES) upišemo arity.

Pomenimo da na kraju od svih napravljenih fajli možemo zadržati faj1.exe i faj1.idb a ostale smemo "obrisati".

---

[8] Ne koristi se nijedan od predikata (9.1.4), pa nismo nigde upotrebili predikat 'visible'.

Na samom kraju dodajmo da pored izloženog u Arity prologu ima i raznih zanimljivih dodataka i proširenja. Tako, Arity na određen način može napraviti .EXE fajlu od raznih .OBJ fajli, a one ne moraju biti samo iz Arity prologa, već mogu biti poreklom iz asemblera, iz C-jezika, iz Pascala.

9.2 Slučaj LPA-prologa

U LPA-prologu se pitanju izvršnih fajli pristupa potpuno drukčije. Razlika nije samo u tome da su u njemu takve fajle tipa .COM. Počećemo jednim primerom. Pretpostavimo da imamo sledeće dve fajle

```
Ime:  mas1.dec,  sadržaj:
                 '<MAIN>':-ajde.
                 ajde:-write('Daj '),read(X), f(X).
                 '<ABORT>':-nl,write('Kraj').
Ime:  mas2.dec,  sadržaj:
                 f(X):-nl,write('Evo '),write(X),ajde.
```

i da nam je namera da za program P, koji čine obe fajle, napravimo jednu izvršnu fajlu imena, recimo, 'mas.com'. Prva fajla mas1.dec namerno sadrži dva predikata '<MAIN>' i '<ABORT>'. Naime, kad budemo napravili željenu izvršnu fajlu njeno izvršenje će početi r a č u n a n j e m f o r m u l e '<MAIN>', a ako se usput pojavi koja greška, onda LPA-prolog "računa" '<ABORT>'-formulu.
Već prema rečenom je jasno kako "teče" program P sačinjen iz te dve fajle:

> Računanje '<MAIN>'-formule traži računanje ajde-formule. Sledstveno na ekranu se pojavi poruka 'Daj '; nakon zadavanja X, prelazi se na f(X), tj. onda se u novom redu štampa poruka 'Evo' i iza toga samo X. Međutim, opet treba računati ajde, i tako slično do beskraja.
> Naravno mi možemo pomoću Ctrl-Break tipke prekinuti to"vrćenje u krug". I tada, shodno rečenom, LPA-prolog računa '<ABORT>'-formulu, tj. u novom redu štampa poruku 'Kraj'.

A sada pristupimo izradi fajle 'mas.com'. U prvom koraku se od datih fajli mas1.dec i mas2.dec prave odgovarajuće OBJECT-fajle mas1.pro i mas2.pro. U njima nastavak .pro je u vezi sa "proloska objekt fajla". Te se fajle mogu napraviti uslugom "prološkog kompajlera", odnosno fajle imena 'pc.com' jedne od osnovnih LPA- fajli. Evo kako se taj 'pc' koristi. Prvo napravimo jednu pomoćnu fajlu imena, recimo, gradi sa ovim sadržajem

```
mas1; +
mas2;
```

Primetite da su redom navedene date fajle sa znakom ; iza svake, i znakom + koji ukazuje da treba i iduci red uzeti u obzir[9]. Sada iz DOS-a kucamo

---

[9] Recimo, da smo imali tri fajle faj1.dec, faj2.dec, faj3.dec onda fajla gradi bi glasila

```
faj1; +
faj2; +
faj3;
```

Međutim, umesto znaka + se može koristiti i znak 'beline' (razmak), pa se ta fajla sme i ovako zadati

```
>pc @gradi
```

gde @ je znak posebne namene[10]. I tada uslugom pc se naprave dve fajle

```
mas1.pro,  mas2.pro
```

U narednom koraku treba da obavimo povezivanje ("linkovanje"). U tu svrhu se koristi prološka fajla imena 'genapp.com', jer napravljene objekt-fajle nisu standardne, pa ne možemo koristiti neki od dobro poznatih linkera. Evo kako se to čini. Iz DOSa kucamo

```
>genapp
```

i onda se korak za korakom pojave pitanja na koje dajemo odgovore(u zagradama je objašnjenje) :

```
Object Files to Combine:  mas1 +     (Kucali smo: mas1 +   i pritisli
                                                          Enter-tipku)
Object Files to Combine:  mas2       (Kucali smo: mas2     i pritisli
                                                          Enter-tipku)
Name of Application:   mas           (Kucali smo mas       i pritisli
                                                          Enter-tipku)
Use the LPA banner in the application [y/n] ?   n
                                  (Kucali smo n i pritisli Enter-tipku)
Command Line Switches:                    (Pritisli smo  Enter-tipku)
```

Tada se naprave dve fajle mas.com i mas.pro. Posao je obavljen, jer mas.com je tražena izvršna fajla. Kucanjem iz DOSa >mas treba da se izvrši opisani[11] program P. I onda se dešava upravo ono što smo prethodno već opisali. Evo sada nekoliko opštih reči:

Izvršna fajla se uopšte može napraviti od nekoliko unapred datih fajli, pri čemu jedna od njih mora da sadrži predikate '<MAIN>' i '<ABORT>'. Prvo se napravi jedna pomoćna fajla poput gradi, recimo istog imena, a onda se sa

```
>pc @gradi
```

naprave odgovarajuće .pro, tj. objekt-fajle. Dalje, za njihovo povezivanje se koristi fajla genapp.com, slično kao gore, pri čemu na pitanje

```
Use the LPA banner in the application [y/n]
```

smo mogli kucati y, što znači da želimo da se izvršenje odvija pod uobičajenim LPA-okvirom[12]. Taj okvir ima ovakav tekst

```
LPA PROLOG Professional Compiler  -v 2.5  28 Sep 1988
Copyright (c) 1988 - Logic Programming Associates Ltd
65052 Evl, 8180 Num, 49152 Txt, 211555 Prg Bytes Free
```

---

faj1 faj2 faj3;

[10] U stvari, gradi je primer tzv. "response" fajli, koje uvek na takav način koriste znak @.

[11] Izvršenje fajle mas.com može se obaviti i pomoću prološke fajle 'pk.com'. U tom slučaju iz DOSa kucamo >pk mas

[12] Ako uopšte "udemo" u LPA prolog onda se pojavi taj okvir.

Znači, u tom slučaju imali bismo utisak kao da smo ušli u LPA-prolog, i da u njemu se raspravlja pitanje ?-'<MAIN>'. U poslednjem redu navedenog okvira piše da:

Za odvijanje programa ("evaluation") imamo 65052 bajtova, za realne brojeve[13] ima dodatnih 8180 bajtova, za tekstovne podatke[14] 49122 bajtova i najzad ima još 211555 slobodnih bajtova.

Te brojke , odnosno "svičevi"[15] sa izuzetkom poslednje[16] se mogu u određenim granicama birati. Naime, da smo na pitanje Command Line Switches: recimo kucali[17] /T128 /N10 /E20 onda u okviru bi pisalo da za realne brojeve[18] (Num) imamo 10 kilobajtova[19], za "evaluation"[20] 20, a za tekstovne podatke 128. Pomenimo da pored navedenih u LPA postoje i još nekoliko drugih "svičeva", u vezi sa bojom ekrana i dr.

A sada nastaje nešto složenija priča o dvema vrstama predikata u LPA-prologu. Naime, u njemu postoje tzv.

statički (ili kompajlirani) i dinamički (ili interpretirani)

predikati. Recimo, u prethodnom primeru svi predikati prisutni u završnoj fajli mas.com su statički. U stvari, uvek kad se okonča pravljenje završne fajle svi predikati joj postaju statični. Najpre nekoliko reči o dinamičkim predikatima. Zamislimo da udemo u LPA-prolog i da učitamo neku fajlu imena recimo ime.dec koja sadrži izvesne članke. Tada svi predikati, tj. njihova imena, su dinamički. Recimo, neka je f jedan od tih predikata. Tada na pitanje

```
?-idef(f)
```

dobiće se odgovor da. Smisao predikata idef je opisiv ovako

Formula oblika idef<dato>
je tačna upravo ako je <dato> dinamički predikat

Pomenimo, da pored idef postoji dualan predikat cdef- koji se odnosi na statičnost predikata. Recimo, za svako od pitanja

```
?-cdef(write). ?-cdef(read). ?-cdef(assert).
```

odgovor je da. Kratko, svi u LPA-prolog ugradeni predikati su statički. Pri radu sa dinamičkim predikatima, a u skladu sa okolnošću da radimo sa interpreterom prološkim, imamo može se reći više slobode, više mogućnosti. Tako

---

[13] Tj. brojeve sa zapetom.

[14] U te podatke dolaze svi atomi, kao i "plodovi" record-predikata, o kojima je govoreno u 8.5.

[15] switches

[16] Ona govori o spoljnoj memoriji koju koristi Prolog (za smeštajnje programa, za prozore i dr).

[17] Primetite da se prvo piše znak /. Veličina slova nije bitna. Tako /T128 i /t128 isto vrede.

[18] Za N je dopušteno da je u granicama 1-16.

[19] Dopušteno je da e ide od 10 do 64. Kratko, program se mora odvijati u jednom segmentu.

[20] Za t je raspon od 4 do 128.

koristeći assert možemo tokom tečenja programa njemu dodavati članke, a sa retract sklanjati[21]. Dalje, koristeći trace (i uopšte razne debug-predikate) možemo korak za korakom pratiti tok programa, tok algoritma.
Nista od toga nije dozvoljeno u radu sa statičkim, ali zauzvrat izvršenje programa sa takvim predikatima je neuporedivo brže. A razlog je očigledan: u radi sa završnim fajlama nakon kompajliranja i linkovanja čitav kod je gotov i preostaje samo njegovo izvršavanje[22]. Naravno, jasno je da u složenijim primenama jedan od važnih zahteva je da program teče što brže. Shodno rečenom to znače da u takvim slučajevima treba graditi izvršne .com fajle tj. isključivo koristiti statičke predikate. Opet s druge strane jednostavnije je raditi sa dinamičkim predikatima. U skladu sa tim pri pravljenju složenije primene obično postupamo po ovakvom planu

> Čitav program P podelimo na izvesne podprograme P1,...,Ps a njih gradimo postupno, polazeći od .dec verzije nakon ispravljanja eventualnih grešaka pravimo odgovarajuće .pro verzije koje na kraju sve "povežemo" u traženu .com verziju programa P.

Međutim, zanimljivo je da pored već izloženih sredstava LPA-prolog ima još neka, blagodareći kojim se takav plan lakše sprovodi. Naime, fajle sa nastavkom .pro se mogu graditi i bez pomoći prološkog kompajlera pc.com. Drugi način je sledeći

> Neka je data fajla imena recimo 'pera.dec' koja sadrži neke prološke članke. Tada ulaženjem u LPA-prolog pitanjem ?-[pera].smo u mogućnosti da "dinamički" radimo sa tim članima. Ako zatim pitamo ?-csave[pera]. onda na takav način se direktno napravi fajla 'pera.pro'.
> Primetimo da se tu koristi predikat csave ("snimi kompajlirano"), i da ime ne mora biti 'pera.pro' (već recimo 'mile.pro' i sl).

Za tako napravljene .pro fajle ćemo reći da su napravljene "u samom Prologu". Pored predikata csave LPA-prolog ima i "dopunski" predikat cload. Pomoću njega se u Prolog može učitati ma koja fajla tipa .pro. Primera radi ako smo već u LPA-prologu onda pitanjem oblika

```
?-cload(ime)
```

se obavlja učitavanje fajle imena 'ime.pro'. Ali, tu sada imamo dva slučaja

> Ako je fajla 'ime.pro' napravljena pomoću prološkog kompajlera pc.com, onda predikati svih njenih članaka su statički, pa na njih ne možemo upotrebiti gore pomenute predikate (listing, assert i dr). Ali, ako je fajla 'ime.pro' napravljena "u samom Prologu", tj. na maločas opisan način, onda ti predikati nisu statički, već dinamički. To je veoma korisna činjenica jer nam može pomoći u građenju složenijeg programa.

Kao što se odatle već primećuje može se dogoditi da u programu koji pravimo neki predikati budu statički, a neki dinamički. Naime, ako smo recimo pomoću pc.com napravili jednu fajlu imena 'ime1.pro', učitali je u LPA-prolog sa ?-cload(ime1) , zatim dodali razne članke imena f1,f2,... iza toga ?-csave(ime2) napravili novu .pro fajlu, onda u njoj predikati f1,f2,... su dinamički. Međutim, ako na kraju krajeva od više .pro fajli (bez obzira kako napravljenih) napravimo "povezujuću" izvršnu fajlu imena recimo 'uradi.com', tada u njoj svi predikati su statički.

[21] Sa statičkim smemo koristiti predikate abolish, kill.

[22] Ako bi se ,za razliku od rečenog, tokom takvog izvršavanja mogla pojaviti formula oblika assert(Članak), gde je Članak neki usput zadan članak, onda je jasno da to zamišljeno izvršavanje ne može proći bez dodatnog menjanja programskog koda, što bi praktično značilo da bi izvšavanje delom bili "interpretatorsko".

10. SVETOVI ALGORITAMA

Jedna od najvažnijih savremenih ideja u Algoritmici[23] je ideja "objektskog programiranja". Premda je taj termin veoma slab i neodgovarajući sama ideja je veoma zanimljiva. Opisno rečeno, pretpostavimo da nas je problem koji smo rešavali doveo do jednog "sveta algoritama" A , i da uz to su se prirodno pojavili njegovi delovi, "podsvetovi" B,C,D,..... Svaki od podsvetova može imati svoje "lokalne" algoritme, tj. algoritme definisane u tom svetu i tu "dejstvujuće". Pored toga, podsvet može "uvesti" iz nekog drugog podsveta izvesne algoritme, i koristiti ih ; a dopunski može dopustiti da da neki od njegovih algoritama budu "izvozni", odnosno da se smeju koristiti i van njega. Takva zamisao je ostvarena u Micro- i u LPA-prologu, gde se umesto reči "svet" koristi reč modul. U delu 10.1, koji se odnosi na opis predikata Micro-prologa ukratko se izlaže i o modulima. Ovo izlaganje se u osnovi odnosi na LPA-prolog[24].

Pretpostavimo da smo upravo "ušli" u LPA-prolog. Tada se nalazimo u tzv. osnovnom modulu imena &. To možemo proloski proveriti postavljanjem pitanja

(10.1) ?-cumod(X).

kada će X biti vezano sa &. Tako smo delom upoznali jedan od modulskih predikata. Ima još jedan slučaj njegove upotrebe. Naime, ako iz osnovnog modula & hoćemo da pređemo u neki drugi ,već prethodno napravljen, modul imena recimo pera onda to možemo učiniti postavljanjem pitanja

```
?-cumod(pera).
```

Naravno, ako posle želimo da se vratimo u osnovni modul &, onda to postižemo pitanjem

```
?-cumod(&).
```

U LPA-prologu modul & je kako rekosmo osnovni, a svi ostali -uvedeni korisnikom- su njegovi podmoduli. Prema prethodnom, predikat cumod nam omogućuje da se krećemo od osnovnog modula & u neki podmodul, kao i obratno; ali nije dopušteno direknto kretanje iz jednog podmodula u neki drugi podmodul.
A sada da vidimo kako se prave podmoduli. Svaki podmodul se sastoji iz ovih "sastavaka"

> imena izvesnih u njemu definisanih članaka,
> i još dva spiska: "izvoznog" i "uvoznog".

Uvozni spisak je lista imena članaka koji taj podmodul želi da koristi sa strane iz nekog drugog modula, a izvozni je lista imena članaka tog podmo-

[23] Tu uključujemo razne programske jezike, i uopšte istraživanja u vezi sa algoritmima.

[24] U stvari, u pitanju modula skoro se ne razlikuju Micro- i LPA-prolog.

dula koji se smeju koristiti van njega. Recimo, neka pera bude ime podmodula koji želimo napraviti, dalje neka izvozna lista bude [c], a uvozna [a,b]. Tada pitanjem

(10.2) ?-crmod(pera,[c],[a,b])

se postiže taj cilj i uz to se odmah "selimo" u taj modul, što se uočava i tako što se umesto osnovnog znaka ?-, na ekranu pojavi znak

pera ?-

Prešavši u modul pera uslugom assert-predikata možemo u njega "uneti" razne članke kao

```
c(X):-X>0.
c(3,78).
d(X):-c(X).
d(44).
```

Kao što vidite definisali smo neke c-članke, i oni su "izvozni", tj. smeju da se koriste

prvo, u osnovnom modulu
drugo, u svakom podmodulu u kome je c član uvozne liste.

Naravno, u modulu pera ne smemo definisati nikakve a-, ni b-članke, jer svaki od njih mora biti "uvozan", tj. definisan

ili u osnovnom modulu
ili u nekom podmodulu gde je njegovo ime uključeno u "izvoznu listu".

Među gornjim člancima d-članci su "lokalni" za modul pera, to znači koristljivi samo u njemu.

Sada uočimo ovaj prološki program imena recimo svet.dec, gde sa strane iza znaka % komentara stoji objašnjenje

(10.3)

```
:- crmod(pera,[a,b],[c]).              % Tako se pravi modul imena pera
a(1).                                  % i ulazi u njega
a(2).                                  % Navedeni su a,b-članci koji su
d(55).                                 % izvozni
b(777).                                % bb-članak je lokalni
b(888).
bb(555).
:- cumod(&).                           %Nakon "punjenja" modula pera
                                       % prvo idemo u osnovni,
:- crmod(dara,[c],[a,f]).  % a onda pravimo nov modul dara
c(999).                                % i odlazimo u njega.
d(77).                                 % c-članak je izvozan, a a-članci
dd(X,Y):-c(X),d(Y).          %  su  uvozni  "od  negde".
:- cumod(&).               % Nakon punjenja modula dara vraćamo se u os-
                           % novni modul i u njega unosimo f,g-članke.
f(1,1).
f(X,Y):-X1 is X-1,Y1 is Y-2,f(X1,Y1).
g(X):-f(X,X).
```

Ako najpre uđemo u LPA-prolog i onda sa

?-[svet].

učitamo fajlu svet.dec onda će se desiti ono opisano u komentarima[25], tj. tekući program će sadržati module pera, dara i sve definisane članke koji se mogu koristiti u skladu sa onim što smo prethodno govorili. Tako, nalazeći se u modulu & na pitanje

?-a(X),write(X),fail.

će X biti redom vezivano za 1 i 2. Međutim, u modulu & ne smemo recimo pitati

?-bb(X).

jer bb-članci su lokalni u modulu pera.

Zamislimo sada da smo ušavši u LPA-prolog sami korak za korakom napravili prethodno pomenute module pera i dara i onda u te module, kao i u osnovni modul & uneli navedene članke. Tako, drugim rečima, na "radilištu" (workspace) imamo na raspolaganju gornji svet modula. Kako onda snimiti taj svet, taj sklop. U tu svrhu se može koristiti predikat

save(<ime>,<šta>)

gde <ime> je po želji odabrano ime, a <šta> je lista modula koje želimo da zajedno snimimo. Recimo, pitanjem

(10.4) ?-save(svet1,[pera,dara,&]).

čitav opisani sklop modula će biti sačuvan u fajli imena svet1.dec. Primetimo još da navodeći & smo naznačili da želimo da usnimimo sve članke osnovnog modula, tj. i f- i g- članke. Međutim, da smo želeli samo da usnimimo f-članke, onda bismo gornje pitanje zamenili ovim

?-save(svet1,[pera,dara,f]).

Znači, kratko ako ne želimo da usnimimo sve članke iz &, onda umesto & navedemo imena tih članaka. Slično, uopšte važi i za podmodule: usnimavaju se u jednu celinu svi oni čija imena se navedu. Primeri:
Pitanjem

?-save(svet2,[pera,&]). se usnimavaju & i modul pera
?-save(svet3,[pera,dara]). se usnimavaju pera, dara kao i &, ali bez ikojih njegovih članaka[26].

U nastavku upoznajemo još neke opšte predikate u vezi sa modulima. To su

(10.5) mdict i dict

Prvi se koristi u obliku mdict(<lista_modula>) a drugi u obliku

dict(<ime_modula>,<izvozna_lista>,<uvozna_lista>,<lokalna_lista>)

Recimo, zamislimo da smo nakon ulaženja u LPA-prolog učitali program (10.3), tj. fajlu svet.dec. Tada pored osnovnog su napravljena dva podmodula pera i dara. Tada na pitanje ?-mdict(X) X se vezuje za listu svih modula, koje

[25] Jer Edinburgški prolozi pri učitavanju reda oblika

:-a,b,...

to "shvataju" kao pitanje ?-a,b,... i odmah "odgovaraju", tj. urade redom a,b,....

[26] U svari, osnovni modul & se uvek usnimava, jedino je pitanje usnivanje njegovih članaka.

trenutno imamo, tj. [dara,pera, &][1]. Ta se lista može dobiti i ovim pitanjem ?-listing(mdict).
Evo primera koriscenja drugog predicata dict, uz pretpostavku da smo ušli u modul ,dara. Tada na pitanje ?-dict(M,A,B,C). M se vezuje za 'dara', A se vezuje za listu [c], B za listu [a,f], a C za listu [d,dd]. Istaknimo da smo te informacije mogli dobiti i pitanjem ?-listing(dict).

11. Z A D A C I, III

Zadatak 11.1. U Zadatku 3.33 smo imali program za građenje svih *-termova pomoću a,b, kao osnovih termova("term-jedinki"). U takve terme pored ostalih dolaze,recimo a,b,(a*a),(a*b),(b*a) i jedna od osnovnih mana tog programa je da on na pitanje

```
?((term _x)(PP _x)FAIL)
```

usled stalne prološke tendencije da upošljava prvu dostupnu granu, drukčije rečeno usled "strategije najlevije grane", recimo nije u stanju da dođe do terma (b * a). Da li se taj program može prepraviti tako da pri nizanju termova nijedan term neće biti "preskočen" ?

Rešenje. Uočimo ovaj program u Micro-prologu[2]

```
((term _x _Duz)(drvo 0 _Duz _x))

  ((drvo _N _Duz a))
  ((drvo _N _Duz b))
  ((drvo _N _Duz (_L * _D)) (LESS _N _Duz)
           (SUM 1 _N _N1)  (drvo _N1 _Duz _L) (drvo _N1 _Duz _D))
```

koji sadrži dve relacije term i drvo. Tada, naprimer, na pitanje

```
?((term _x 3)(PP _x)FAIL)
```

na ekranu će se prvo, pojaviti samo konačno mnogo termova, i drugo, među njima će se nalaziti i svi *-termi sa najviše 3 znaka *. Tu su recimo, a,b ..,(b * a),...((a * b) * a) ,..

Slično važi i uopšte za neko pitanje oblika ?((term _x p)(PP _x)FAIL) gde je p neki unapred dati prirodan broj. Šta je razlog svemu tome ?
Najpre pogledajmo kako za neke p radi taj program. Tako, pri raspravi pitanja ?((term _x 0)(PP _x)FAIL) evo šta se odigrava:

> Radi računa formule (term _x 0) se prvo traži (drvo 0 0 _x) i na osnovu prvog drvo-članka se dobije _x=a. Na ekranu se štampa a, ali kad FAIL "potera nazad" onda (drvo 0 0 _x) se dokaže pomoću drugog drvo-članka i tada _x=b. Na ekranu se pojavi b. FAIL opet tera nazad i sada bi trebalo koristiti treci drvo-članak. Ali, njegova "rampa" (LESS _N _Duz) je (LESS 0 0), dakle netačna pa pregranjavanje na treci članak nije moguce. Algoritam se završava. Znači, na ekranu su se pojavili samo termovi a,b, tj. termi sa 0-zvezdica.

Postavimo sada pitanje ?((term _x 1)(PP _x)FAIL). Tada imamo ovo dešavanje:

> U vezi sa formulom (term _x 1) sada se mogu koristiti sva tri drvo članka:
> ((drvo _N _Duz a))
> ((drvo _N _Duz b))
> ((drvo _N _Duz .... , gde tačkice zamenjuju nenavedene delove.
> Sledstveno, prvo će se pojaviti rešenja a,b. Onda, kad nas FAIL na to primora, prelazimo na korišćenje trećeg članka, koji uzimajući u obzir tekuće vrednosti promenljivih smemo ovako krace napisati:
>
> ((drvo 0 1 (_L * _D))
> (drvo 1 1 _L)(drvo 1 1 _D)) (_x je (_L * _D))
>
> Tada se za _L, prema prvom drvo-članku najpre dobije a. Slično za _D se dobije a. Stoga _x= (a * a) i to se štampa na ekranu. Kad FAIL potera nazad, onda se traži nov dokaz za (drvo 1 1 _D) i za _D se dobije b. Na ekranu se pojavi (a * b). Opet deluje FAIL. Ali sada za (drvo 1 1 _D) nema novog dokaza, jer pri pokušaju upotrebe trećeg drvo-članka "rampa" glasi (LESS 1 1) , što nije ispunjeno. Pošto za _D nema novog dokaza vraćamo se na formulu (drvo 1 1 _L) i njoj tražimo nov dokaz, što daje _L=b. Nakon toga opet dolazimo na, zbog FAIL-a, ponovljeno dokazivanje te formule i tako nastaju ova rešenja (b * a) i (b * b). Kad nakon toga opet FAIL potera nazad, onda budući da za (drvo 1 1_L), zbog "rampe", nema više novih dokaza algoritam se završava. Kao što se vidi pojavili su se svi termovi
>
> a b (a * a) (a * b) (b * a) (b * b)
>
> čiji broj zvezdica je 0 ili 1.

Već na osnovu tih primera je jasno da uopšte pri raspravi pitanja oblika

```
?((term _x p)(PP _x)FAIL)
```

gde je p zadat prirodan broj će se na ekranu pojaviti konačan broj "rešenja", odnosno vrednosti za _x. Grubo rečeno, osnovni razlog je što u formuli oblika (drvo _n _Duz _x), promenljiva _n prvo ima vrednost 0, i u daljem toku to _n se, možemo tako reći, stalno uvećava za 1, ali- to je bitno ne sme da bude veće od _Duz.

Zadatak 11.2 (Nastavak prethodnog). Da li se može napraviti prološki program koji će pomenute termove redati " po spratovima":

> Prvo, sa 0 zvezdica tj, a,b
> Drugo, sa 1 zvezdicom tj. (a * a), (a * b), (b * a), (b * b)
> itd. ?

Rešenje. Uočimo ovaj program

```
((term a 1))
((term b 1))
((term (_L * _D) _Rez) (LESS 1 _Rez) (razdvoj _Rez _pr _dr)
                                     (term _L _pr)(term _D _dr))
```

[1] U stvari, umesto znaka & pojaviće se reč $PROLOG.

[2] Lako se napravi prevod na ma koji Prolog Edinburgške sintakse.

```
((razdvoj _x 1 _p) (LESS 1 _x) (SUM 1 _p _x))
((razdvoj _x _p _q)
          (LESS 1 _x) (SUM _x1 1 _x)(razdvoj _x1 _p1 _q)(SUM 1 _p1 _p))

((termi _n)(SUM _n 1 _nn)(term_do 1 _nn))
((term_do _n _n)(PP Kraj))
((term_do _i _n) (NOT ?((term _x _i)(PP _x)FAIL))
                                    (SUM 1 _i _i1) (term_do _i1 _n))
```

Recimo, na pitanje kao ?((termi 5)) na ekranu se redom ispisuju termi sa 0, 1, 2, 3, 4 i na kraju sa 5 zvezdica. Tako na početku će biti termi

```
a
b
(a * a)
(a * b)
```

a na samom kraju term

```
(((b * b) * b) * b) * b)
```

U programu od najvećeg značaja je predikat term. Na pitanje oblika

```
?((term _x 5)(PP _x)FAIL)
```

se postupno ispisuju svi termi sa 4 zvezdice, odnosno uopšte na pitanje oblika

```
?((term _x p)(PP _x)FAIL)
```

gde je p =1,2,3,... zadat prirodan broj, se ispisuju svi termi sa p-1 zvezdicom. U term-člancima se nalazi relacija razdvoj, pomoću koje se dati prirodan broj na sve moguće načine može razdvojiti na dva sabirka.
Primera radi, na pitanje ?((razdvoj 5 _p _q)(PP _p _q)FAIL) na ekranu će se ispisati dvojke

```
1  4
2  3
3  2
4  1
```

Sada ćemo na jednom primeru videti kako "rade" term-članci. Postavimo ovo pitanje ?((term _x 3)(PP _x)FAIL). Tada, uz slobodnije označavanje članaka, imamo ovo rasuđivanje:

(*1) (term _x 3)= (razdvoj 3 _pr _dr)(term _L _pr)(term _D _dr)
gde _x=(_L * _D)

Upošljavanjem razdvoj-članaka prvo dobijemo _pr=1, _dr=2, pa tako imamo

(*2) (term _x 3)=(term _L 1)(term _D 2), _x=(_L * _D)
Za _L se odmah dobije a, a za _D se dobije ova "jednačina"
(term _D 2)=(razdvoj 2 _p _d)(term _L1 _p)(term _D1 _d),
gde _D=(_L1 * _D1)

odnosno ova

(*3) (term _D 2)=(term _L1 1)(term _D1 1) ,gde _D=(_L1 * _D1).
Najpre se za _L1 i _D1 dobiju vrednosti a, a , pa _x dobije vrednost (a * (a * a)) i to se štampa na ekranu.
Zbog FAIL-a se potraži drugi dokaz za (term _D1 1) što daje _D1=b, i na ekranu se štampa (a * (a * b)).
Kad opet FAIL potera nazad onda za (term _D1 1) više nema nove grane pa od mesta te formule mora da nastane novo vraćanje. Tako dolazimo do (term _L1 1) i za nju tražimo nov dokaz. Tako dobijemo _L1=b. Iza toga opet se zaputimo na (term _D1 1) i zbog FAIL-a ga "izdokazujemo" na sve načine. Tako će se na ekranu pojaviti ova rešenja

```
(a * (b *  a))    (a * (b * b))
```

Tako smo za sada na sve moguće načine "izdokazivali" (term _D 2). Prema jednakosti (*2) novi FAIL - "napad" će tražiti novi dokaz za (term _L 1). To će odmah dati jednakost _L=b. Iza toga nas čeka ponovno "izdokazivanje" formule (term _D 2). Tako se pojave još četiri rešenja

```
(b * (a * a))   (b * (a * b))   (b * (b * a))   (b * (b * b))
```

Novi FAIL- napad nas vrati do razdvoj-formule u(*1) i onda se pomoću razdvoj-članaka dolazi do ovih vrednosti: _pr=2, _dr=1. Sada se umesto (*2) pojavi ova "jednačina"

(*4) (term _x 3)=(term _L 2)(term _D 1), gde _x=(_L * _D)

I dalje, kratko rečeno sklapaju se sva _L-rešenja formule(term _L 2) sa svim _D-rešenjima formule(term _D 1). Tako se dolazi do ovih osam termova

```
((a * a) * a)  ((a * a) * b)   ((a * b) * a) ((a * b) * b)
((b * a) * a)  ((b * a) * b)   ((b * b) * a) ((b * b) * a)
```

čime se ujedno i završava algoritam.
U stvari, tako je objašnjena suština programa jer ostale stvari su tehničke prirode. Naime, predikat term_do na pitanje oblika

```
?((term_do 1 p))
```

redom "po spratovima" štampa sve terme sa 0,1,...,p-1 -om zvezdicom, dok predikat termi je završan uslužni predikat. Kao što smo već na početku rekli na pitanje oblika ?((termi p)), gde p je dat, se po spratovima redaju svi termi sa 0,1,2,...,p zvezdica.

U vezi sa prevodom datog programa na Edinburgšku sintaksu, istaknimo samo jednu pojedinost:

```
Deo   (NOT ? ((term _x _i)(PP _x)FAIL))   treba ovako prevesti
       not ((term(_x,_i),write(_x),nl,fail)
```

Zadatak 11.3. Neka su t1, t2 dva ma koja |-terma. Reći ćemo da su oni

do na promenljive jednaki

ukoliko se razlikuju samo u "oznakama promenljivih", tj. svaki od njih može nastati iz drugog zamenom njegovih promenljivih nekim drugim promenljivim, uz to različitim promenljivim treba da odgovaraju različite zamene. Recimo, do na promenljive su jednaki ovi termi:

```
(a _x _y)  i   (a _y _x)
((b _x)(c 3 _z))   i   ((b _A)(c 3 _B))
```

Napraviti prološki program za predikat: " do na promenljivsku jednakost ", dogovorno označen sa jeste.

Rešenje uslugom[3] GRNHOL:

```
((jeste _x _y)(GRNHOL _S1 _x) (GRNHOL _S2 _y) (EQ _S1 _S2))
```

Recimo, na pitanje

```
?((jeste (a _x _y) (a _y _z)))
```

odgovor će biti da. Razlog:

Prvo se traži _S1 tako da važi (GRNHOL _S1 (a _x _y)) i tada će se za _S1 dobiti vrednost (a _A _B) jer pri upošljavanju GRNHOL predikata se promenljive iz formule (a _x _y) redom zamenjuju članovima iz ovog spiska

```
_A  _B  _C  _D  _E  _F  _G ...
```

Upravo, u skladu sa tim, kad se dalje dođe do računanja

```
(GRNHOL  _S2 (a _y _z))
```

onda se promenljivim _y, _z terma (a _y _z) opet "od početka" redom daju zamene _A, _B i _S2 dobije vrednost (a _A _B). Sledstveno, _S1 i _S2 su zaista jednaki.

Zadatak 11.4. Neka je T zadani I-term. Napraviti njegovu kopiju T', tako da T' bude do na promenljive jednak sa T, ali ima svoje, nove promenljive.
Rešenje uslugom GRNHOL-predikata:

```
((kopija _X _Y)(GRNHOL _S _X)(ADDCL ((rrr _S)))(rrr _Z)(EQ _Y _Z))
```

Primetimo da je rrr uslužni predikat. Glavni koraci u algoritmu su:

Kad se ,za dati _X, napravi _S onda on kao rezultat GRNHOL-formule sadrži samo konstante, pored ostalih i ove prološke "sakrivene" _A, _B,.... Međutim, ako zatim dodamo članak ((rrr _S)) i potom sa (rrr _Z) potražimo _Z, to _Z će predstavljati traženi rezultat. Setimo se da smo sličnu dosetku koristili u Zadatku 6.21.

Zadatak 11.5. Uočimo program

```
(*)     ((a _x)(a _x))
        ((a 1))
```

Kao što dobro znamo na pitanje ?((a 1)) će se bez kraja uključivati prvi članak, elem pomoću Prologa nećemo saznati da je (a 1) tačna. Preraditi dati program tako da se otkloni taj nedostatak, odnosno strože rečeno:

Čim se dogodi upadanje u petlju da se obavi iskakanje iz nje i zatim prede na narednu granu.

Da li se uopšte ma koji zadan prološki program može preraditi u drugi koji prvome otklanja "upadanje u petlje" ?

Rešenje. Prvo navodimo rešenje za program (*). Taj program glasi:

```
(Δ1)
((a _x)
 (IF (CL ((rel a _x))) ((DELCL ((rel a _x))) FAIL) ((ADDCL ((rel a _x)))))
     (a _x)
 (DELCL ((rel a _x))))
((a 1))
((rel))
```

Vidi se da je prvi a-članak polaznog programa izmenjen tako da iza njegove "glave: (a _x) je dodata jedna IF-formula i na sam njegov "kraj" je dodata jedna DELCL-formula. Relacija rel je uslužna, tj. pomoćna. Evo kako radi taj program:

Recimo, da pitamo ?((a 1)). Tada se upošljava prvi a-članak i prvo proverava da li postoji članak ((rel a _x)), tj. članak ((rel a 1)). Kako njega nema to se prvo uslugom ADDCL-a on doda i nastavlja dalje "račun" po a-članku. Ali, opet se dođe na (a _x). Prema prološkom algoritmu upošljava se ponovo prvi a-članak. Međutim, sada kad se dođe na IF-formulu vidi se da članak ((rel a 1)) postoji. Stoga prema toj IF-formuli prvo "obrišemo" članak ((rel a 1)), jer on je svoju ulogu "odigrao", a zatim zbog FAIL-a moramo da predemo na drugi a-članak -što nam naravno baš odgovara. Tako saznajemo da je formula (a 1) tačna.

Istaknimo da je članak oblika((rel a _x)) samo "lokalan" za prvi a-članak, jer kad god se prvi a-članak napušta taj rel-"svedok" se briše. Doduše u ovom primeru budući da se nikako ne može stići na sam kraj prvog a-članka poslednja DELCL-formula je nepotrebna. Međutim, namerno smo je stavili da bismo lakše uvideli opšti slučaj, koji odmah izlažemo.
Neka je tako P neki zadat program. Od njega ćemo, na način koji opisujemo, napraviti nov program petlja(P), koji se vlada "skoro isto" kao P uz jedinu prednost da je obezbedjen od upadanja u neku petlju. Evo kako se pravi taj novi program:

Neka je Clanak ma koji članak iz P. Ako je to elementarna aksioma, tj. jednoformulski članak, onda ga zadržavamo, tj. ne menjamo. Ako je on oblika

```
(for1 for2 ... fork)
```

zamenjujemo ga ovim novim člankom

```
(for1
  (IF (CL (rel|for1) ((DELCL (rel|for1)) FAIL) ((ADDCL (rel|for1))))
        for2 ... fork (DELCL (rel|for1)))
```

Na kraju dodamo i jedan "provizoran" članak ((rel)), da se ne bi dogodilo da se Prolog "buni" da mu je rel nepoznat predikat.

Zadatak 11.6. Napraviti program za upis jedne reči, znak po znak tako da "završnik" bude znak "beline", tj. "znak" čiji ASCII-kod je 32.
Rešenje. Evo rešenja u Arity- i LPA-prologu. Uočimo ovaj program:

```
rec(C,[C|Cs]):-get0(C1),(slovo(C1),rec(C1,Cs);Cs=[]).

slovo(32):-!,fail.     /*  belina */
slovo(X).

rec1(X):-get0(C),rec(C,[C|Y]),name(X,[C|Y]).
```

Objašnjenje:

Program se pobuđuje sa ?-rec1(X). i onda korisnik kuca znak do znaka; reč se završava "kucanjem" znaka beline, tj. praznog mesta. Značajno je da u slučaju Arity-prologa čim se ukuca "završnik", tj. znak beline na ekranu, kao vrednost za X, se odmah pojavljuje ukucana reč, tj. Prolog prekida eventualno dalje ukucavanje;

[3] Videti Zadatak 6.18.

dok u slučaju LPA-prologa reč X se na isti način gradi, ali se ne štampa pri kucanju "beline" već tek kad korisnik "otkuca" nov red. Tako, ako ukucamo "struju znakova" :

pera ide u skolu

i onda pređemo u novi red za X će se dobiti reč pera.

Inače, kao što se vidi za "čitanje", tj. "read","input" znakova je upotrebljen prološki predikat get0. Pored njega u Prolozima Edinburgške sintakse postoji i predikat get. Između njih postoji ova razlika:

Za razliku od get0, predikat get ne može učitati znakove koji nisu "na ekranu štampivi". Recimo, to je slučaj sa znakom 'beline', sa Ctrl-A, Ctr-B i sl.

Sam algoritam približno ovako teče. Recimo, da smo na pitanje ?-rec1(X). kucali

pera#

gde dogovorno # stoji umesto znaka beline. Tada bi se odmah pojavilo računanje ove formule

rec(p,[p|Y])

a za nju imamo:

```
rec(p,[p|Y]) =rec(e,Y),      Drugi znak je 'e'
             =rec(r,Z)       Treci znak je 'r';  Y je [e|Z]
             =rec(a,U)       Četvrti znak je 'a'. Z je [r|U]
```

i to se dešava sve do kucanja beline. A onda, imamo

rec(a,U)=rec(a,[a|V]) jer U mora biti oblika [a|V]

i pošto je ukucan znak beline, "ukida" se prvi član disjunkcije i prelazi na drugi koji nalaže jednakost: V=[] i završava se račun formule rec(p,[p|Y]) . Dodelnik pamti ove jednakosti

Y=[e|Z] Z=[r|U] U=[a|V] V=[]

iz kojih se dobije Y=[e,r,a]. Stoga [p|Y]=[p,e,r,a].

Za polaznu formulu imamo konačno ovu jednakost

rec1(X)=name(X,[p,e,r,a])

iz koje se dobije X='pera'.

Na kraju istaknimo, da za razliku od navedenih Prologa, u Micro-prologu se ne može napraviti slično rešenje, jer u vezi sa upisom na ekran, tj. "čitanja fajle tastatura", nema predikate poput get, get0.

Zadatak 11.7. Pretpostavimo da korisnik kuca rečenicu prirodnog jezika kao struju znakova sa među belinama. Za završetak rečenice kuca se jedan od ovih znakova

'.' '?' '!'

a u rečenici može učestvovati i znak zareza. Napraviti prološki program koji će od takve struje znakova sastaviti listu "reči". Na primer, od struje znakova

idi tamo,a ne onamo

treba da se dobije lista

[idi,tamo,',',a,ne,onamo]

Rešenje koje navodimo je u stvari proširenje ideje izložene u prethodnom

zadatku. To rešenje je napisano prema rešenju Problema 80 iz knjige

How to solve it with Prolog , Lisabon 1982

pisaca: H.Coelho, J.C. Cotta i L.M.Pereira. Jedina izmena je obavljena u vezi sa "znacima" čiji ASCII kodovi su

```
10          (tzv. "line-feed", pravi smisao: "idi na početak reda")
13          (tzv. CR , kome odgovara tipka koja je obično naznačena
            kao Enter. Smisao: idi u nov red)
```

Naime, kad se u kucanju struje znakova kuca tipka Enter , onda prema izmeni koja je izvršena u programu se nalaže da se pređe i u novi red, što se inače bez te izmene ne bi desilo. Ta izmena je upravo u članku

slovo(13):-nl,!,fail.

Ceo program u Arity-prologu glasi:

```
recen(F):-get(C),reci(C,F).

reci(C,[P|Ps]):-slovo(C),rec(C,C1,L), name(P,L),reci(C1,Ps).
reci(44,[',|Ps]):-get(C1),reci(C1,Ps).
reci(63,[?]).
reci(46,[.]).
reci(33,['!']).
reci(U,P):-get(C),reci(C,P).

rec(C,C1,[C|Cs]):-get0(C2), (slovo(C2),rec(C2,C1,Cs); C1=C2,Cs=[]).

slovo(32):-!,fail.     /* belina  */
slovo(63):-!,fail.     /* znak ?  */
slovo(46):-!,fail.     /* znak .  */
slovo(33):-!,fail.     /* znak !  */
slovo(44):-!,fail.     /* znak ,  */
slovo(13):-nl,!,fail.
slovo(X).
```

Program se pobuđuje pitanjem ?-recen(X). nakon čega nastaje upis struje znakova, a Prolog na kraju ,kao X, štampa listu pojedinih reči, sastavaka.

Zadatak 11.8. Zamislimo da smo na samom početku na izvestan način uneli neke prološke članke i da hoćemo da ih snimimo kao odvojenu datoteku (fajlu) izvesnog imena. Kako to uraditi ?

Rešenje zavisi od vrste Prologa. Tako imamo:

Micro-prolog. Recimo da smo ušavši u jezik ukucali ove članke

```
((a 1 2))
((b _x _y)(a _y _x))
```

i da želimo da ih snimimo u fajlu 'pera.log'.

Tada prosto kucamo SAVE PERA ili postavimo pitanje[4] ?([SAVE PERA)).

LPA-prolog:

Ukucavanje programa se obavlja ovako. Prvo kucamo

?-[user].

naravno bez znaka pitanja. Iza toga možemo po želji ukucati članke, re-

[4] Može i ?((SAVE "PERA.LOG")).

cimo ove

```
a(1,2).
b(_x,_y):-a(_y,_x).
```

Za završetak se ukuca znak Ctrl-Z.

Nakon toga se opet pojavi prompt ? Prologa i to znači da su ti članci prihvaćeni. Pretpostavimo da sada hoćemo da ih snimimo u fajlu imena pera, odnosno pera.dec, jer u LPA se podrazumeva nastavak .dec. Postupamo slično kao u Micro-prologu:

```
?-save('pera.dec').
```

Arity-prolog:

Ukucavanje se obavlja slično kao u LPA-prologu, jedino za završetak se umesto Ctrl-Z kuca reč: end_of_file.
Međutim, snimanje je nešto drukčije. Prvo može ovako, odnosno postavljanjem pitanja

```
?-tell('pera.ari'),listing,told.
```

Objašnjenje:

Tu se koristi predikat tell, čijom uslugom se otvara fajla sa imenom pera.ari, i ta fajla je otvorena za upisivanje. Dalje,

```
listing
```

znači da se ispiše ("izlista") tekući program. Ali, kuda ? Pošto je pera.ari otvorena za upisivanje to taj listing ide u nju.
Konačno, nakon listing-a dolazi predikat told sa smislom da se zatvori "upisna" fajla. Inače, opisani način se može koristiti i u LPA-prologu. Drugi način je primenom predikata file_list, prosto postavljanjem pitanja

```
?-file_list('pera.ari').
```

To rešenje se odnosi na novije verzije Arity-prologa (od 4 naviše).

Zadatak 11.9. Kako da se fajla datog imena učita, tj. "unese" ("louduje")?
Rešenje zavisi od vrste Prologa, a uvek je pretpostavka da fajla datog imena postoji.

Micro-prolog:

Postavimo pitanje ?((LOAD PERA)) ili samo kucamo LOAD PERA i fajla imena pera.log će biti učitana.

LPA- i Arity prolozi:

Postavimo pitanje
?-consult('pera.dec'). odnosno ?-consult('pera.ari').
ili, što je kraći oblik toga: ?-[pera]. odnosno ?-[pera].

Napomena 11.1. Edinburgške verzije Prologa po pravilu pored predikata consult imaju i predikat reconsult. Evo kako on deluje:

Recimo da tekući program sadrži neke članke imena a,b,c,... i da onda računamo formulu reconsult(File), gde je ime File zadato, i to činimo ili postavljanjem direktnog pitanja ?-reconsult(File) ili tokom razvoja algoritma. Tada se fajla tog imena učita, ali dopunski

se uradi i ovo :

Neka ta fajla sadrži neke članke imena Ime i "arnosti" k[5]. Tada ukoliko tekući program već ima neke članke istog imena i iste arnosti svi se oni brišu, i umesto njih dolaze odgovarajući iz te fajle imena File.

Zadatak 11.10. Kako da se bez učitavanja neka fajla imena Ime otvori, redom pročitaju njeni sastavci, odnosno članci i ispišu na ekranu ?

Rešenje zavisno od verzije Prologa, ali kao i u prethodnom zadatku pretpostavka je da fajla navedenog imena postoji.

Micro-prolog:

Neka fajla ima ime: pera.log. Jedan program, koji se pokreće sa ?((ajde)) glasi:

```
((ajde)(OPEN "PERA.LOG")(citaj "PERA.LOG")(CLOSE "PERA.LOG"))

((citaj _X)(READ _X _Y) (PP _Y)(citaj _X))
((citaj _X))
```

Objašnjenje:

Na pitanje ?((ajde)) prvo se otvara fajla imena pera.log, a onda se ona "čita", odnosno dolazi predikat citaj koji može značiti što želimo, a ovde prema citaj-člancima imamo
prvo dođe (READ _X _Y), tj. (READ "PERA.LOG" _Y) i onda se iz fajle "PERA.LOG" učita jedan sastavak, odnosno upravo prvi članak. Zbog (PP _Y) se na ekranu štampa _Y, odnosno taj prvi članak. Iza toga opet dolazi (citaj _X) tj. (citaj "PERA.LOG") i ponovo se javi (READ "PERA.LOG" _Y), ali to je sada bitno, taj "READ" je tako pametan[6], da nastavlja sa čitanjem i sada čita drugi članak, itd. Kad se dođe na kraj fajle onda vrednost READ-formule je ne, pa da bismo što je prirodno pružili mogućnost da dejstvuje formula (CLOSE _X), tj. da se zatvori fajla "PERA.LOG" uz prvi citaj-članak je dodat i članak ((citaj _X))

U vezi sa navedenim Micro-prološkim programom dodajmo da je on sposoban ne samo da čita i ispiše fajlu sa prološkim člancima, već uopšte neku tekstovnu fajlu. Recimo, jedna takva fajla može biti sastavljena od ovih reči:

Danas je ponedeljak prvi dan u nedelji.

Tada će se štampati ovi sastavci (navedeni sa povećanim međurastojanjem)

Danas je ponedeljak prvi dan u nedelji.

Arity- i LPA-prolog (rešenje uslugom see predikata) :

```
ajde:-write('Daj ime '),read(Ime), see(Ime), uradi, seen.
uradi:-read(Term),pokazi(Term).
```

---

[5] Recimo, kod članka f(X,Y):-g(2),h(X,Y,Z). se kaže da je to članak sa imenom f, i da f ima "arnost" 2, jer : f se odnosi na dva argumenta. Kad bi umesto f(X,Y) stajala jedna od ovih formula
f,f(1),f(X,6,7,8) onda
onda bi arnost bila 0,1, odnosno 4.

[6] Odnosno algoritam je tako smišljen.

```
pokazi(end_of_file):-!.
pokazi(Term):-write(Term),nl,uradi.
```

Objašnjenje:

Na pitanje ?-ajde se prvo daje ime neke već postojeće fajle, recimo pera.ari[7]. Uslugom predikata see se za čitanje otvara ta fajla. Predikat uradi je u stvari "zadužen" da se njegovom pomoći prođe redom kroz fajlu i uradi željeno, uz uslugu predikata pokazi. To "prolaženje" je upravo čitanje iz fajle i tome služi read-predikat. Primetimo da za prepoznavanje kraja fajle služi konstantska reč: end_of_file.

Istaknimo, da pri ispisu članaka oni se tretiraju kao reči koje se završavaju tačkom, kao znakom kraja reči, pa se s tim u vezi svaki članak štampa u posebnom redu i uz to bez "svoje tačke". Takođe, promenljive se pri ispisu iskazuju preko svojih adresa, pa se recimo umesto X na ekranu pojavi nešto kao _4D3F.

Navodimo i ova rešenja bez upotrebe see predikata, u osnovi slična sa navedenim Micro-prološkim rešenjem. Rešenje u LPA-prologu je potpuno slično:

```
ajde:-open('pera.dec'),citaj.
citaj:-read('pera1.dec',Y), radi(Y).
radi(end_of_file):-write('Gotovo'),close('pera.dec').
radi(Y):-write(Y),nl,citaj.
```

Program se pokreće sa: ?-ajde. kada se ispisuje sadržaj fajle pera, dec. U Arity prologu algoritam je skoro isti, ali ostvarenje, odnosno program je drukčiji. Razlog je što Arity-prolog drukčije radi sa fajlama. Tako, pri otvaranju fajle 'pera.ari', već postojeće, njoj se dodeljuje tzv. ručka (handle) i u daljem se rad sa fajlom odvija preko njene ručke. O tome više videti niže u Zadatku 11.12. Inače, program glasi:

```
ajde:-open(Rucka,'pera.ari',r),      /* Slovo r poručuje da se fajla 'pera.
     citaj(Rucka),close(Rucka).         ari otvara za read, tj. čitanje  */

 citaj(Rucka):-read(Rucka,Y),
    ifthenelse(Y=end_of_file,(write('Kraj ')),(write(Y),nl,citaj(Rucka))).
```

Program se pokreće pitanjem ?-ajde. i onda se na ekranu ispisuje sadržaj fajle pera.ari.

Zadatak 11.11. Prološkim programom napraviti fajlu željenog imena, a zatim u nju upisati po volji: prološke članke ili opštije neke reči. Fajlu nakon toga zatvoriti.

Rešenje zavisi od vrste Prologa.

Micro-prolog:

```
((ajde)(PP Daj ime fajle)(R _Ime) (CREATE _Ime)(pisi _Ime) (CLOSE _Ime))

((pisi _Ime) (repeat)(PP Daj clanak, za kraj kucati Dosta)
        (R _Y)(IF (EQ _Y Dosta)((PP Kraj)) ((WRITE _Ime _Y) FAIL)))

((repeat))
```

[7] Odnosno pera.dec.

[8] To je predikat samog Prologa.


```
((repeat)(repeat))
```

Objašnjenje:

Pomoću repeat-članaka se "oponaša" repeat-predikat iz Edinburgške sintakse. Program se pokreće sa ?((ajde)). Najpre se zadaje ime fajle, recimo, "MILE.LOG". Nakon toga se uslugom predikata CREATE otvara za upis potpuno nova fajla datog imena "MILE.LOG". Iza toga nastaje

upis članaka ili reči

i završava se kucanjem reči Dosta. Fajla se onda zatvara. Kao što se vidi upis se obavlja uslugom predikata WRITE, koji se uopšte upotrebljava u obliku

```
(WRITE Kuda Sta),
```

gde Kuda je ime fajle najpre otvorene pomoću CREATE-predikata, a Sta je lista oblika

```
(*)      (Deo1 Deo2 ... Deok)
```

gde pojedini delovi mogu biti reči po volji, pa i članci. Evo primera takvih lista:

```
(pera jova 23 (a 3) )     U fajlu mile.log će se kao odvojeni sastavci upisati reči:   pera  jova  23  (a 3)
(((a 2)(b 3)) ((c 3) (d 4)))
                          Sada će se fajli dodati i ove reči, odnosno članci:  ((a 2)(b 3))  ((c 3)(d 4))
```

Naravno, ako želimo da napravimo fajlu samo od članaka, tada pri navedenom algoritmu te članke zadamo preko jedne ili nekoliko lista oblika(*), gde Deo1,...,Deok su članci.
Istaknimo da WRITE-predikat nakon upisa neke liste doda još naznaku za red (tj. fajli doda još "znake" čiji ASCI-kodovi su 10 i 13).

Arity-prolog: Navodimo najpre program koji je u osnovi sličan prethodnom napisanom na Micro-prologu, ali koristi predikate tell i see:

```
ajde:-write('Daj ime fajle'), read(Ime),tell(Ime), upis(Ime),told.

upis(Ime):-tell(user),write(' Daj clanke ,za kraj kucati  Dosta '),nl,
    repeat, tell(user),write('Daj '),nl, read(X),
    ifthenelse(X='Dosta',(tell(user),write('Kraj')),
                         (tell(Ime),pisi(X),fail)).

pisi(X):-write(X),write(.),write(' ').
```

Objašnjenje:

Prvo, nova fajla željenog imena se otvara predikatom tell. Shodno tome kad god iza tell(Ime) ma šta dođe za pisanje, kao write('Daj') i dr., to sve ide u tu fajlu, što naravno ne želimo. Da bismo to sprečili kad god je namera ispis na ekran prethodno je stavljena formula

```
tell(user)
```

koja nam "ekranske ispise" poput write('Daj ') usmerava upravo na ekran. I naravno, ako želimo da ispis ide u fajlu imena Ime onda prehodno je stavljena formula tell(Ime), da bi se to omogućilo.
I još jedna komplikacija. Zamislite da se u navedenom programu izbaci pisi-predikat i zameni sa write. Tada, ako tokom algoritma ukucamo

```
a(1).
b(2):-c(3).
Dosta.
```

fajla Ime će imati ovaj sadržaj[9]

```
a(1)b(2):-c(3)
```

kao niz znakova. To naravno nije niz članaka, jer iza svakog nedostaje tačka i bar jedna belina.

Predikat pisi je "zadužen" upravo da otkloni taj nedostatak. Međutim, ako program želimo da koristimo za upis nekih reči ali bez tačaka na kraju, onda u tu svrhu pisi-predikat možemo ovako uvesti

```
pisi(X):-write(X),write(' ').
```

Zadatak 11.12. Sadržaj postojeće fajle znak po znak, tj. kao "struju znakova" ispisati na ekran.

Rešenje zavisi od vrste Prologa.

Micro-prolog:

```
((pisi _Ime) (OPEN _Ime) (ispis _Ime))
((pisi _Ime)(CLOSE _Ime))

((ispis _Ime)(GETB _Ime _X)(PUTB "WND:" _X) (ispis _Ime))
```

Objašnjenje:

Recimo, na pitanje ?((pisi "PERA.LOG")) evo šta se dešava. Otvara se ta fajla i nastaje (ispis "PERA.LOG"), dok može, a kad ne može, tj. kad stignemo do kraja fajle, onda (ispis "PERA.LOG") trenutno dobije vrednost ne. To tera da se promeni pisi-grana, tj. pređe na drugu koja nalaže da se zatvori fajle "PERA.LOG". Sam ispis ovako teče. Pomoću predikata GETB se čita bajt po bajt, a onda preko PUTB se štampa na ekran, koji je označen sa "WND:".

Arity- prolog.

```
pisi(Ime):-open(Rucka,Ime,r), ispis(Rucka).
pisi(Ime):-close(Rucka).

ispis(Rucka):-get0(Rucka,X),put(X), ispis(Rucka).
```

Objašnjenje:

Sklop algoritma je sličan prethodnom uz prilagođavanje Arity- sintaksi i odgovarajućim predikatima. Program se recimo poziva sa

```
?-pisi('pera.ari').   odnosno ?-pisi('pera.dec')
```

i sledstveno se obavlja ispis fajle imena pera.
Na početku se koristi open-predikat. On se uopšte koristi u obliku

```
open(Rucka,Ime,Nacin)
```

gde je Ime ime željene fajle, dok za Način postoje ove mogućnosti

(ϕ)
- r za read, tj. "ispis"
- w za write, tj. "upis"
- a za append, tj. "dopisivanje na kraj fajle"

[9] To se recimo dobije ako se iz DOS-a koristi usluga procedure type.

- rw read/write. Dopusta se i ispis i upis.
- ra read/append. Dopusta se i ispis i dopisivanje na kraj.

Uz zadane Ime i Nacin putem open-predikata se kao plod daje Rucka, na engleskom Handle. I u daljem algoritmu se "saobraćaj sa fajlom" isključivo obavlja preko te Rucke. Tako preko

```
get0(Rucka,X)
```

se iz fajle sa imenom pera "čita" znak po znak X, a preko[10] put(X) se X štampa.

LPA- prolog:

```
pisi(Ime):-open(Ime),ispis(Ime).
pisi(Ime):-close(Ime).

ispis(Ime):-get0(Ime,X),put('WND:',X),ispis(Ime).
```

Objašnjenje:

To je skoro isti kao Micro-prološki program, s tim da su korišćeni predikati put, get0 (slično LPA-prolog ima i get-predikat) Edinburgške sintakse. Kao što vidite taj LPA-program, za razliku od Arity-programa ne koristi pojam Rucke, pa je jednostavniji. Ali, Arity za uzvrat u radu sa fajlama je "bogatiji". Naime, tablica (Φ) je moćna, a u LPA-prologu neke od njenih mogućnosti nisu dopuštene. To su upravo a i ra. Inače od istih slabosti pati i Micro-prolog.

Napomena 11.2. Navodimo u Micro-prologu program koji rešava nešto opštiji problem od razmatranog. Naime, tim programom se iz jedne fajle označene sa _Istok može njen sadržaj preneti u novo napravljenu fajlu imena _Utok:

```
((pisi _Istok _Utok) (OPEN _Istok)(CREATE _Utok) (ispis _Istok _Utok))
((pisi _Istok _Utok)(CLOSE _Istok)(CLOSE _Utok))

((ispis _Istok _Utok)(GETB _Istok _X)(PUTB _Utok _X)(ispis _Istok _Utok))

((ajde)(PP Daj ime istok- i utok-fajle)
                                    (R _Istok)(R _Utok)(pisi _Istok _Utok))
```

Program se pobuđuje sa ?((ajde)), nakon čega se zadaju imena za _Istok i _Utok fajlu.

Zadatak 11.13. Pretpostavimo da nam je data neka fajla čiji početak glasi

```
Danas je ponedeljak,mesec je jun, a godina je 1960.
```

Kako tu fajlu otvoriti i onda iz nje izvući podreč koja počinje od 10-og mesta a duga je 5 slova, što je očigledno podreč: poned ?

Rešenje. Da bismo lakše razumeli rešavanje podesno je da zamislimo kako od svog početka, u memoriji bajt po bajt je fajla ispunjena:
Početni bajt, reći ćemo 0-ti bajt je ispunjen sa ASCII kodom za 'D', tj. sa 68. Bukvalno bit po bit taj bajt izgleda

```
0 1 0 0 0 1 0 0        (To je 2-zapis za 68)
```

Prvi bajt je ispunjen sa ASCII kodom za 'a', a ostali redom sa ASCII kodovima za 'n', 'a', 's', ' ' (znak beline), itd.

U skladu sa rečenim bilo bi podesno da se po fajli možemo kretati bajt po

[10] Predikatima get, get0 se kao plod dobija ASCII-kod, pa stoga ako želimo da dobijeni plod štampamo na ekran kao običan znak koristimo predikat put.

bajt i takvom idejom "otkinuti" traženu podreč. Prolozi po pravilu omogucuju ostvarenje takve zamisli i tada osnovnu ulogu ima tzv. seek-predikat. Evo prvo rešenja u Micro-prologu, u slučaju ma koje fajle imena Ime, a traži se podreč od mesta _A do mesta _B. Pretpostavlja se opis SEEK predikata dat u delu 12.1, tačke 12. Program glasi

```
 ((ajde)(PP Daj ime fajle)
        (R _Ime)(OPEN _Ime)(PP Od kog bajta ?)(R _A)
        (PP Do kog bajta ?)(R _B) (pisi _Ime _A _B) (CLOSE _Ime))
((pisi _Ime _A _A))
((pisi _Ime _A _B)(SEEK _Ime 0 _A)
     (/* Kao sto se tu vidi pretpostavlja se da je podrec u pocet-)
     (/* nom bloku od 1024 bajtova, sto se lako moze promeniti)
     (GETB _Ime _X) (PUTB "WND:" _X)(SUM _A 1 _A1)(pisi _Ime _A1 _B))
```

Program se pokreće pitanjem ?((ajde)).

Arity-prolog je veoma podesan za takve zadatke. U stvari, smo tu zamisao izložili u delu 12.2 tačke 2 pri dosta podrobnom opisu seek-predikata. Na kraju, izlažemo i rešenje u LPA-prologu :

```
ajde:-write('Daj ime fajle '),read(Ime),open(Ime),write('Otvorena'),nl,
      write('  Od kog bajta podrec citati '), nl, read(Broj),
      write(' Kolika je duzina podreci ? '),nl, read(Duz),
      seek(Ime,Broj),     % Na tom mestu uslugom seek predikata
                          % fajlin pokazivač (pointer) se  upravlja na bajt
                          % čiji redni broj je Broj.
      fr(Ime,[c(Duz)],[Rec]),     % Tu se koristi  fr-predikat, o čemu više
                                  % reči dole
      write('To je rec '),nl, write(Rec),nl,close(Ime).
```

Program se pokreće pitanjem ?-ajde. na koje se zadaje ime fajle, početni i završni broj bajta. Tada se na ekranu štampa odgovarajuća podreč. Pored seek-predikata, u programu se koristi fr-predikat, odnosno "formated read"predikat[11]. On se javio u obliku

```
fr(Ime,[c(10)],[Rec]).
```

gde prvi argument Ime predstavlja ime fajle, drugi argument je tzv. "lista formata", ovde je to lista [c(10)], a treći je "lista podataka". Taj predikat, uz pretpostavku da je fajla Ime otvorena, i da je fajlin pokazivač dobio neku vrednost Broj, deluje ovako. Od mesta Broj u fajli Ime "otkida" 10 bajtova i "podatku" Rec kao promenljivoj pridružuje odgovarajuću reč i još fajlin pokazivač poveća za 10.
Inače, "otkidano" je 10 bajtova jer lista formata je [c(10)]. Da smo umesto c(10) stavili recimo c(5) otkidala bi se reč dužine 5 i fajlin pokazivač povećao za 5.

Napomena 11.3. Ova napomena je nastavak prethodnog teksta o fr predikatu. Iznosimo neka proširenja u vezi sa listama formata i podataka. Naime, lista podataka je uopšte neka lista promenljivih, a lista formata[12] je lista

[11] Formatirano čitanje.

[12] Uz korišćenje samo c-"opisnica".

oblika [c(v1),c(v2),...,c(vk)], gde su v1,...,vk neki celi brojevi[13]. Evo nekoliko određenih primera, sa odgovarajućim objašnjenjima

(i) Pri računu formule
fr(Ime,[c(10)],[A,B,C])
se otkidaju tri susedne 10-podreči[14]; A se vezuje za prvu od njih, B za drugu, a C za treću.

(ii) Pri računu formule
fr(Ime,[c(10),c(5)],[A,B])
A se vezuje za prvu 10-podreč ,a B za narednu 5-podreč.

(iii) Pri računu formule
fr(Ime,[c(10),c(5)],[A,B,C,D,E])
A se vezuje za prvu 10-podreč , B za narednu 5-podreč, C za narednu 10-podreč[15], D za narednu 5-podreč[16] i konačno E se vezuje za narednu 10-reč

Kraj Napomene 11.3

Zadatak 11.14. Napraviti program za pisanje fajle čiji osnovni sastavci treba da budu članovi datog spiska kao

Ime	Prezime	Ocena
Jovan	Mitić	6
Pera	Ilić	8
Goran	Janić	6
....		

gde se pretpostavlja da recimo, Ime može imati najviše 10 slova, a Prezime najviše 15 slova. Ocena je ceo broj, jedan od 5,6,7,8,9,10 . Takođe napraviti program za ispisivanje takve fajle.

Rešenje. Evo prvo takvog upis-ispis programa u Micro-prologu

```
((upis)(PP Daj ime)(R _Ime)(CREATE _Ime)
      (PP Pri upisi za kraj kucati rec Dosta) (upisi _Ime))
((upisi _Ime)(PP Daj ime, prezime, ocenu : ) (R _X)
   (IF (EQ _X Dosta)
      ((FWRITE _Ime ((CON 10)(CON 15)(NUM 2)) (Dosta Dosta 0))
                 (PUTB _Ime 13)(PUTB _Ime 10) (CLOSE _Ime))
      ((R _Y)(R _Z)(FWRITE _Ime ((CON 10)(CON 15)(NUM 2)) (_X _Y _Z))
                 (PUTB _Ime 13)(PUTB _Ime 10) (upisi _Ime))))

((ispis)(PP Daj ime)(R _Ime) (OPEN _Ime)(ispisi _Ime))
((ispisi _Ime)
    (FREAD _Ime ((CON 10)(CON 15)(NUM 2)(CON 2)) (_x _y _z _u))
    (IF (EQ _x Dosta) ((CLOSE _Ime))
    ((PP _x _y _z) (ispisi _Ime)))
 )
```

Željeno upisivanje spiska se postiže pitanjem ?((upis)), i tada se koriste članci imena 'upis' i 'upisi'. Posebnost tog dela programa programa je ko-

[13] Od -122 do +122.

[14] Umesto 'podreč dužine 10' rekli smo 10-podreč. Slično činimo i nadalje.

[15] Format c(10),c(5) je dotle jedanput "potrošen", i sada ga koristimo opet.

[16] Sada je format c(10),c(5) još jedanput "potrošen", pa ga koristimo opet.

riscenje FWRITE- predikata, čijom "uslugom" se podaci u fajlu mogu unositi u formatu koji hoćemo. Format je odreden listom

```
((CON 10) (CON 15) (NUM 2))
```

koja "poručuje" prvo dolazi reč na prostoru 10 polja, pa druga reč na prostoru 15 polja i na kraju broj zapisan na 2 polja[17]. FWRITE-predikat se uopšte koristi u obliku

```
(FWRITE ime lista-formata lista-podataka)
```

Primetite da iza FWRITE-formule u članku stoji

```
(PUTB _Ime 13)(PUTB _Ime 10)
```

sa smislom da se doda još po jedan novi red.
Upisivanje se završava kad se kao podatak otkuca reč Dosta. Tada iz razloga koje objašnjavamo fajli koju pravimo se dodaje ovaj red

```
(*)   Dosta     Dosta   0
```

Razlog je što ispisni-deo programa taj red koristi kao završni, kao "zaustavni". Ispisni-deo se sastoji iz 'ispis' i 'ispisi' članaka i pokreće se pitanjem ?((ispis)). Sada umesto FWRITE osnovnu ulogu ima dualan FREAD predikat. Primetite da je sada format sa dodatkom dela (CON 2), koji je "nameren" za dva bajta koji definišu novi red. To praktično znači da se pri ispisu kad god uztreba "vade" 2-bajta novog reda.
A inače na sam kraj je dodat navedeni zaustavni red (*), jer Micro-prolog pri korišćenju FREAD-predikata "nije sposoban" da prepozna kraj fajle.
Sada izlažemo rešenje u LPA-prologu, koje je u osnovi slično prethodnom. Umesto FWRITE, FREAD predikata se koriste LPA-proški predikati fr, fw. Međutim, program sada ne sadrži zaustavni red, jer LPA nema opisanu manu Micro-prologa. Podsećamo da smo o fr predikatu već govorili u prethodnom zadatku i Napomeni 11.3.

Program glasi:

```
upis:-write('Daj ime '),read(Ime),create(Ime),
      nl,write('Pri upisu za kraj kucati rec dosta '),upisi(Ime).

upisi(Ime):-write('Daj ime, prezime i ocenu'),nl,read(X),
           X\='dosta'->
              (read(Y),read(Z),fw(Ime,[c(10),c(15),c(2)],[X,Y,Z]),
               put(Ime,13),put(Ime,10)),upisi(Ime).
upisi(Ime):-put(Ime,13),put(Ime,10).

ispis:-write('Daj ime '),read(Ime), open(Ime),ispisi(Ime].

ispisi(Ime):-
     fr(Ime,[c(10),c(15),c(2),c(2)],[X,Y,Z,U]),
          write(X),tab(3),write(Y),tab(3),write(Z),ispisi(Ime).
ispisi(Ime):-close(Ime).
```

Za pravljenje upisne fajle zadženi su 'upis' i 'upisi' članci. Upis se obavlja postavljanjem pitanja ?-upis. Pri upisu osnovnu ulogu ima fw predikat. U navedenoj listi formata deo c(2) i promenljiva Z su mamenjeni oce-

[17] Neki drugi format je recimo određen listom ((NUM 3)(NUM 7)(CON 6)(NUM5)).

ni. Posto je u pitanju broj to pri zadavanju vrednosti moramo koristiti znake navoda ' ', pa recimo umesto 10, kucati[18] '10'. Dalje, istaknimo da premda su ime i prezime formatirani sa 10, odnosno 15 znakova pri upisu smemo zadavati i kraca (ali ne i duža !) imena, odnosno prezimena. Naravno, uprkos tome ime i prezime će u upisnoj fajli biti smeštena upravo na 10, odnosno 15 polja uz korišćenje određene "količine" znakova "beline". Recimo ako kao prvi podatak zadamo: pera jovic 2, onda u upisnoj fajli će on biti ovako prisutan[19]

```
pera       jovic          2
```

Međutim, ako želimo, možemo da postignemo da pri upisivanju prvo dođu beline pa onda stvarni podaci tj. da imamo ovakvu sliku

```
      pera           jovic 2
```

U tu svrhu u prvom upisi-članku umesto formata [c(10),c(15),c(2)] treba uzeti format [c(-10),c(-15),c(-2)].
Za ispisivanje zaduženi su 'ispis' i 'ispisi' članci. Ispis se obavlja postavljanjem pitanja ?-ispis. Primetite, da slično kao u slučaju gornjeg Micro-prološkog rešenja poslednji član u listi formata, tj. c(2) zajedno sa promenljivom U je zadužen da iz ispisne fajle "vadi" nove redove.

Napomena 11.4. U LPA-prologu pored "opisnica" tipa c, postoji još nekoliko drugih, kao q,u,g,v,d,s,b i f. Recimo, poslednja "opisnica" je u vezi sa realnim brojevima. Čitaoca upućujemo na priručnik LPA-proloqa u vezi sa raznim tim opisnicima. Nažalost, naše je iskustvo da rad sa nekim od njih nije pouzdan. Recimo, opisnica f, korišćena u obliku f(Duz,Dec), gde su Duz-dužina polja[20] a Dec broj decimala se vlada dosta nesigurno. Tako, ako postavimo pitanje

```
?-fr('BUF:',[f(7,2)],[A,B,C]).
```

onda -proverite to- smemo za A,B,C dati ove vrednosti[21]

```
  1234.67        (Broj je na 7 polja,i ima 2 decimale)
    12.10
  1000.00
```

dok recimo ako te vrednosti zadamo nekim drugim redom može se dogoditi neuspeh.
U stvari, među opisnicima je najvažnija c, kojoj smo i do sada poklonili najviše pažnje. Njeno ime c, dolazi od 'constant', u smislu konstantske reči, odnosno atoma. To znači ako je recimo, nekoj c-opisnici pridružena promenljiva X, onda vrednost tog X je neki atom, kao

```
      pera, jova, 'Mile', '1234'
```

Primetite da poslednje dve reči ne bi bile atom ako bi im se sklonili znaci ' ' navoda. Evo jednog malog primera za ilustraciju. U slučaju pitanja

[18] Još o nekim detaljima o c-opisnicama govorimo više u Napomeni 12.4 dole.

[19] Prvo reč pera pa onda 5 puta belina, i sl.

[20] Ako nam je "ulazna fajla" tastatura, onda je njeno ime 'BUF:'. Dodajemo i ovo. Ako je "izlazna fajla" ekran , onda se za nju koristi ime 'WND:' Kao što vidite to je slično kao u Micro-prologu.

[21] Pazite, tačka se ne kuca na kraju podatka !

```
?-read(X),fw('WND:',[c(6)],[X]).
    X-u sme dati nepromenljivska vrednost, kao
       pera.        'Mile'.
    ali ne sme promenljivska kao
        Pera.
    U prva dva slučaja X će imati vrednosti pera i Mile.Primetite:
       dužina zadane reči ne mora biti 6, sme biti i manja, ali ne
       sme biti veća.
    Tako, X-u ne smemo pri read(X) dodeliti vrednost kao:  petrovic.
```

Činjenicu da dužina reči sme da bude kraća od dužine propisane formatom primera radi možemo podesno koristiti u programima pravljenja "upisnih" fajli, što smo imali u prethodnom zadatku.
Kraj Napomene 11.4.

Zadatak 11.15. Kako sva rešenja neke formule smestiti

u jednu fajlu, koja
se najpre napravi ?

Rešenje. Navodimo dva rešenja u Micro-prologu, jedno korišćenjem običnih tekstovnih, tj. ASCII fajli, a drugo korišćenjem binarnih fajli. Pretpostavimo da je dat ovaj program

```
((a (11 22)))
((a 1))   ((a 77))  ((a 5 555))
```

i da hoćemo da sva _x-rešenja formule (a|_x) smestimo u jednu fajlu sa osnovnim imenom res .U tu svrhu uočimo ovaj program

```
((asci_upis)(CREATE "res.log")
     (NOT ? ((a|_x) (WRITE "res.log" (_x)) FAIL))(CLOSE "res.log"))

 ((asci_ispis)(OPEN "res.log")(asci_pisi "res.log"))

 ((asci_pisi _x)(INTERM _x _y)(PP Evo _y)(asci_pisi _x)(CLOSE _x))

((bin_upis)(CREATE "res.bin")
     (NOT ? ((a|_x) (BWRITE "res.bin" (_x)) FAIL))(CLOSE "res.bin"))

 ((bin_ispis)(OPEN "res.bin")(bin_pisi "res.bin"))

 ((bin_pisi _x](BINTERM _x _y](PP Evo _y))
 ((bin_pisi _x)(CLOSE _x))
```

Tada na pitanje ?((asci_upis)) se redom prave _x-rešenja za (a|_x) i stavljaju u fajlu "res.log", koja je inače najpre napravljena. Na kraju se ta fajla zatvara. Njen ispis se obavlja pitanjem ?[(asci_ispis)).
Slično, na pitanje ?((bin_upis)) se ista rešenja stavljaju u binarnu fajlu imena "res.bin", najpre napravljenu, a na kraju se ta fajla zatvara. Ispis se obavlja pitanjem ?((bin_ispis)). U vezi sa binarnim fajlama podvucimo:

> Njihova imena imaju ekstenziju bin. Dalje, za upis podatka u nju se koristi BWRITE-predikat, a za ispis, tj. iz nje uzimanje podatka se koristi BINTERM-predikat. BWRITE se koristi slično kao WRITE, tj. u obliku
>
> (BWRITE Kuda Lista_podataka)
>
> Recimo, (BWRITE "res.bin" (1 2 3 4)).

Zadatak 11.16. Kako u Prologu sa Edinburgškom sintaksom znati da li dati sintaksno ispravan objekat je

1) oblika prave liste, kao što su [a,23,X], [2,[a,b],c] ali, recimo nije [a,f(p),q] ,jer jedan sastavak je f(p), odnosno "struktura".
2) oblika "strukture" kao f(P,g,[a,b],X), k(h,s(u,V)) i dr.
3) oblika sastava (konjunkcije) kao (a,b,f(4,X)),([P,q],r,s).
4) oblika sastav-rastav kao
   (a,(B,c;p,f(q,s)),r)
5) oblika neelementarnog članka(tj. "pravila") kao
   a(8,9):-p(e),write(8).
   koji ima znak ":-" -reći ćemo grlo-znak.

Rešenje. Navodimo imena odgovarajućih predikata i definicione članke:

1) Ime predikata je jelista , a članci su :

```
jelista(X):-not var(X),X=[].
jelista([X|Y]):-(atomic(X);var(X);jelista(X)),jelista(Y).
```

Kao što se vidi :

> lista je ili prazna
> ili inače: njen rep je lista, dok "glava" joj je ili promenljiva ili "atomic" ili lista.

2) Ime predikata je jestrukt a članci su :

```
jestrukt(X):-not var(X),X=..[A|B],
             atomic(A), not number(A), A\=':-', A\='.', A\=',',A\=';'.
```

Kao što se vidi jestrukt(X) važi ukoliko pri prevodu X na listu [A|B], glava liste
je konstantska reč, koja nije jednaka jednoj od ovih
:- (ona se koristi kod neelementarnih članaka)
. ("Vezivni znak" za liste. Recimo, '.'(2,[]) je lista [2])
, ("Vezivni znak" za sastave kao (a,b,c) i dr.)
; ("Vezivni znak" za rastave(disjunkcije) kao a;b)

3) Ime je jesastav, a članak je :

```
jesastav(X):- not var(X),X=..[',',A,B].
```

4) Ime je jesastav-rastav , a članak je :

```
jesastav_rastav(X):-not var(X),(X=..[',',A,B]; X=..[';',A,B]).
```

5) Ime je jepravilo, a članak je:

```
jepravilo(X):-not var(X),X=..[':-',A,B],jestrukt(A),jesastavrastav(B).
```

Zadatak 11.17. (Problem 8 kraljica). Na šahovskoj tabli je raspoređeno 8 kraljica tako da se "ne tuku" nikoje dve. Odrediti sve takve rasporede.

Rešenje. Prvo navodimo jedno rešenje bukvalnom primenom ideje "generiši i proveri" iz Zadatka 3.37:

```
dat(1).
dat(2).
dat(3).
dat(4).
dat(5).
dat(6).
dat(7).
```

```
    dat(8).
ajde:-dat(X1),
      dat(X2),usl(X2,X1,1),
      dat(X3),usl(X3,X1,2),usl(X3,X2,1),
      dat(X4),usl(X4,X1,3),usl(X4,X2,2),usl(X4,X3,1),
      dat(X5),usl(X5,X1,4),usl(X5,X2,3),usl(X5,X3,2),usl(X5,X4,1),
      dat(X6),usl(X6,X1,5),usl(X6,X2,4),usl(X6,X3,3),usl(X6,X4,2),
                                                    usl(X6,X5,1),
      dat(X7),usl(X7,X1,6),usl(X7,X2,5),usl(X7,X3,4),usl(X7,X4,3),
                                                 usl(X7,X5,2),usl(X7,X6,1),
      dat(X8),usl(X8,X1,7),usl(X8,X2,6),usl(X8,X3,5),usl(X8,X4,4),
                                              usl(X8,X5,3),usl(X8,X6,2),usl(X8,X7,1),
      write(X1),tab(1),write(X2),tab(1),write(X3),tab(1),write(X4),tab(1),
      write(X5),tab(1),write(X6),tab(1),write(X7),tab(1),write(X8),nl,fail
ajde.

usl(X,Y,P):-X=\=Y,X-Y=\=P,Y-X=\=P.
```

Program se pokreće pitanjem ?-ajde.i onda se redaju svi traženi rasporedi. Recimo, prvo se pojavi ovaj

1 5 8 6 3 7 2

što znači da se, gledano sleva nadesno,prva kraljica nalazi u vrsti 1,druga u vrsti 5, treća u vrsti 8 ,itd.
Drugo rešenje , koje takođe koristi ideju "generiši i proveri" glasi:

```
kraljica(N,Rez):-lista(1,N,Lis),kraljica(Lis,[],Rez).
kraljica([],Rez,Rez).
kraljica(Lis,Priv,Rez):-izvuci(Q,Lis,Lis1),
                        not napada(Q,Priv), kraljica(Lis1,[Q|Priv],Rez).

izvuci(X,[X|Xs],Xs).
izvuci(X,[Y|Ys],[Y|Zs]):-izvuci(X,Ys,Zs).

lista(M,N,[M|Ns]):-M<N,M1 is M+1, lista(M1,N,Ns).
lista(N,N,[N]).

napada(X,Xs):-napada(X,1,Xs).
napada(X,N,[Y|Ys]):- X=:=Y+N; Y=:=X+N.
napada(X,N,[Y|Ys]):-N1 is N+1,napada(X,N1,Ys).
```

Taj program u stvari radi sasvim slično kao prethodno opisan,ali za razliku od njega ne odnosi se jedino na 8 kraljica , već ma koliko njih. Tako, na pitanje

```
?-kraljica(4,X),write(X),nl,fail.
```

će se pojaviti svi rasporedi od 4 kraljice, međusobno netukućih. Opisujemo razne pojedinosti u raspravljanju tog pitanja:

Prvo, formula kraljica(4,X) se računa "jednačinom" :

kraljica(4,X)=lista(1,4,Lis),kraljica(Lis,[],X)
= kraljica([1,2,3,4],[],X), jer predikat lista pomoću svojih članaka pravi listu. Bliže, pri računu formule oblika lista(1,n,X),gde n=1, 2,..., promenljiva X dobije vrednost: [1,2,3,...,n]. Primetimo da formula kraljica([1,2,3,4],[],X) ima tri argumenta.Konačni rezultat se postupno gradi i čuva u drugom argumentu,nazvanom Priv.

(*1) = izvuci(Q,[1,2,3,4],Lis1),not napada(Q,[]),kraljica(Lis1,[Q|[]],X)
Sada na pozornicu stupa predikat izvuci.Njegovom uslugom u ovom koraku se iz liste [1,2,3,4] izvlači njen prvi član, odnosno tačnije rečeno važe jednakosti

(*2) Q=1, Lis1=[2,3,4]

Već tu istaknimo da taj predikat pri eventualnom "backtracking" je sposoban da proizvede i ostala "izvlačenja", opisana ovim jednakostima

(*3) Q=2, Lis1=[1,3,4]; Q=3,Lis1=[1,2,4]; Q=4,Lis1=[1,2,3]

U ovom koraku koristeći jednakosti (*2),algoritam dalje ovako teče. Računamo formulu napada(Q,[]).Tu se pojavljuje predikat napada,koji je "zadužen da brine o među-napadanju kraljica".Ovde imamo trivijalan slučaj i vrednost formule: not napada(Q,[]) je da. Međutim,kasnije pred sam kraj će nam se pojaviti računanje ove formule

napada(3,[1,4,2])

Njen smisao je ovaj:
Zamislimo da su do sada već postavljene tri kraljice prema ovom crtežu

```
X   Kr  X
X   X   X
X   X   Kr
Kr  X   X
```

gde znak X predstavlja prazno polje, a Kr postavljenu kraljicu. Tom rasporedu kraljica, gledajući brojeve redova u kojima se one nalaze, odgovara lista [1,4,2]. Sada vrednost formule

napada(3,[1,4,2])

je da, ukoliko nova kraljica dodata u stubac ispred tih kraljica i stavljena u treći red "se tuče" sa nekom desnom od nje. Nije teško videti da je vrednost te formule ne.U stvari, ta postavka kraljica je već jedno traženo rešenje.

Vratimo se algoritmu na mestu na kome smo bili zastali i prešli na taj opis predikata napada, odnosno vratimo se jednačini (*1). Sada nas čeka računanje formule kraljica(Lis1,[Q|[]],X), tj. prema (*2) formule

kraljica([2,3,4],[1],X)

To se može ovako shvatiti. Privremeni rezultat je lista [1],tj. trenutno je jedna kraljica postavljena i to u poslednji stubac i prvi red.Uopšte, kraljice se postavljaju "zdesna nalevo",odnosno od poslednjeg stubca prema prvom. Pored toga preostale su nam još tri neraspoređene kraljice, i njihovi redovi smeju biti 2,3,4. Međutim, nakon određenog prološkog računa se zaključuje da je ta formula netačna[1]. Stoga, prema jednačini (*1) vraćamo se na deo izvuci(Q,[1,2,3,4],Lis1) i tražimo mu nov dokaz, odnosno (videti (*3)) imamo Q=[2],Lis1=[1,3,4].Deo not napada(...) je opet tačan i tako nam se sada pojavi ova jednačina

kraljica(4,X)=kraljica([1,2,3,4],[],X)
=kraljica([1,3,4],[2],X)

Kratko opisano u daljem će se prvi argument i drugi, tj. Priv ovako postupno menjati:

---

[1]U stvari, jasno je da poslednja kraljica ne sme biti u prvom redu, jer bi nas inače čekao podproblem sa tri kraljice, a taj nema potvrdno rešenje.

```
[1,3,4],[2] -->[1,3], [4,2]
            -->[3], [1,4,2]
            -->[], [3,1,4,2]
```

i onda će se postaviti pitanje računanja formule

```
kraljica([],[3,1,4,2],X)
```

koja se može računati samo pomoću drugog kraljica-članka i tako se konačno dobije X=[3,1,4,2].

Zadatak 11.18. Kako u rasponu od 1 do datog prirodnog broja N naći neki "slučajan" broj ?

Rešenje. Reč slučajan smo stavili pod navodnike jer:

> To je jedan od divnih matematičkih pojmova, koji je poput pojma beskonačnosti u velikoj meri tajanstven, pošto nam nikako ne može poći za rukom da mu damo ikakvu strogu definiciju.

Jedan program glasi:

```
izvor(13).
sluc(Y,X):-izvor(S),X is (S mod Y) +1, retract(izvor(S)),
           Novi is (125*S+1) mod 4096, asserta(izvor(Novi)),!.

ajde:-write('Od 1 do kog broja dati slucajne '),
           read(Y),repeat,sluc(Y,X),write(X),tab(2),X=5.
```

se pokreće pitanjem ?-ajde i onda se zadaje neki prirodan broj Y nakon čega se programom izračunavaju razni "slučajni" brojevi u rasponu 1 .. Y. Program se zaustavlja kad se među njima pojavi broj[2] 5. Inače, ti "slučajni" brojevi, uz pomoć predikata izvor, nastaju ovako:

> Neka Y=20. Prvo se pojavi računanje formule sluc(20,X). Pošto je dato izvor(13) to dobijemo S=13. Dalje
>
> ```
> X is (13 mod 20) +1
> ```
>
> tj. X je 14, jer pri delenju 13 sa 20 ostatak je 13.
> I tako, prvi "slučajan" broj će biti 14.
> Dalje, se izbacuje članak izvor (S), tj. izvor (13) i umesto njega dodaje izvor(Novi), gde
>
> ```
> Novi is (125*13 +1) mod 4096
> ```
>
> što daje Novi=1626. Tako sada imamo članak
>
> ```
> izvor(1626)
> ```
>
> pa će se nov "slučajan" broj X računati pomoću njega, itd.

Napominjemo da u LPA-prologu ima ugrađen predikat irand kojim se prave razni "slučajni" brojevi. Recimo, na pitanje

```
?- X is irand(20).
```

dobiće se neki prirodan broj od 1 do 20.

[2] To znači da za Y ne smemo dati broj manji od 5, jer program se neće zaustaviti.

Zadatak 11.19. Koristeći okolnost da su u Arity-prolog ugrađeni "brojački predikati" napisati program kojim se može odrediti broj rešenja date formule.

Rešenje. Neka je primera radi relacija a data člancima

```
a(2).   a(77).   a(88).
```

Tada formula a(X), po X, ima tri rešenja. Sada opisujemo program koji je sposoban da obavi taj "posao" brojanja rešenja. Jedan takav program glasi

```
res:-ctr_set(0,0), a(X),ctr_inc(0,Y) fail.
res:-write('Ima '),ctr_is(0,Y),write(Y),write(' resenja').
```

i uključuje se sa ?-res. kada štampa poruku: Ima 3.resenja

Objašnjenje:

> Arity prolog ima ukupno 31 ugrađen brojač, i njihovi nazivi su brojke 0,1,2,...,30. Formulom ctr_set(0,0) je 0-brojač postavljen na početnu vrednost 0. Recimo, sa ctr_set(21,7) se 21-brojač postavlja na 7. Pored tog predikata ctr_set sa brojačima ima još nekih:
>
> ctr_inc -povećavanje za 1; slično, ctr_dec smanjivanje za 1
> ctr_is -za saznavanje tekuće vrednosti.
>
> Gornji program na očigledan način koristi predikate ctr_inc,ctr_is.

Zadatak 11.20. Napisati program kojim se može nacrtati ma koji pravougaonik poput navedenog na slici

uz pretpostavku datosti koordinata levog gornjeg i desnog donjeg ugla.

Rešenje. Evo najpre rešenja u Arity-prologu. Jedan program glasi:

```
kutija(Y,X,Y1,X1):-
        C is X1-X, tmove(Y,X), wc(C,196), tmove(Y,X1),
        wc(1,191), C1 is Y1-Y,Y2 is Y+1,
        vert(X1,Y2,Y1), tmove(Y,X), wc(1,218),
        tmove(Y2,X), vert(X,Y2,Y1), tmove(Y1,X),
        wc(1,192), X2 is X+1,tmove(Y1,X2),
        wc(C,196),tmove(Y1,X1), wc(1,217).

vert(X,Y,Y1):-
        ctr_set(0,Y), repeat,ctr_inc(0,Z),
        tmove(Z,X), wc(1,179), Z==Y1.
```

i pokreće se pitanjem, kao

```
?-kutija(3,5,15,20).
```

Tada levi gornji ugao nacrtane "kutije" je u 3-čem redu i 5-tom stubcu, a donji desni ugao u 15-om redu i 20-tom stubcu ekrana.
U programu učestvuju[3] ovi ,do sada nepomenuti, predikati

[3] Pojavljuju se i brojački predikati opisani u prethodnom zadatku.

tmove, wc

Predikat tmove je pravi "kursorski" predikat. Računanjem formule oblika

tmove(prvi,drugi)

se kursor seli u ekransku tačku (prvi,drugi).
Predikat wc služi za višestruko pisanje datog znaka. Tako sa

?-wc(20,65). ili sa ?-wc(20,'A').

će 20 puta zaredom biti napisano slovo A. U programu se wc predikat koristi nekoliko puta:

Za crtanje znaka vodoravne crtice, njegov ASCII kod je 196,
Za crtanje "coškova" kutije, ASCII kodovi su 218,191,192,217.
Za crtanje znaka uspravne crtice, njegov ASCII kod je 179.

Članak imena vert je "zadužen" da uz pomoć znaka uspravne crtice crta bočne stranice kutije.

Sada prelazimo na rešenje u LPA-prologu. Uočimo ovaj program

```
ajde:-gdev(1),fill([100,150,100,3500,3000,3000,3000,100),
                                        get0('TRM:',Ch),gdev(0).
```

koji se pokreće sa ?-ajde. i tada se na ekranu nacrta mnogougao čija temena su u ovim tačkama[4] (koordinate redom uzimamo iz liste u fill-formuli):

```
(100,150),(100,3500),
   (3000,3000), (3000,100)
```

Evo podrobnijeg objašnjenja. LPA-prolog "gleda" ekran kao koordinatni sistem tačaka od (0,0) -donji levi ugao,do (32767,32767)- gornji desni ugao. Svi grafički predikati LPA-prologa počivaju na tzv. gsx-sistemu. U vezi sa tim direktorija LPA-prologa po pravilu sadrži fajlu gsx.exe.Da bi se koristili grafički predikati LPA-prologa prvo taj sistem na određen način,opisan u Priručniku, mora biti instaliran.Posle toga, pre ulaska u LPA-prolog kucamo gsx tako da pokrenemo gsx-sistem. Tek iza toga kad uđemo u LPA-prolog i učitamo gornji program pitanjem ?-ajde. postiže se ono što smo već opisali. Naime:

Prvo se računa formula gdev (1), što se svodi na prelaz na grafički "mode"[5].
Drugo, računa se data fill-formula i sledstveno na ekranu se nacrta pomenuti mnogougao, tj. ovde baš četvorougao. Primetimo da u opštem slučaju pomoću fill/1 predikata se može crtati mnogougao ali koji ima paran broj stranica, od 2 do 256.
Treće, taj mnogougao "ne beži" sa ekrana, jer na redu je računanje formule get0('TRM:',Ch),što praktično znači da se čeka korisnik da pretisne ma koju tipku tastature.
Nakon takvog pretiskanja računa se formula gdev (0), što se svodi na izlazak iz grafičkog moda i vraćanje u "običan", tj. tekstovni mode.

U LPA-prologu pored navedenih gdev,fill ima još nekih grafičkih predikata:

[4] Temena smo izabrali skoro proizvoljno.

[5] Ukoliko je gsx-sistem instaliran.

line -za crtanje linije (tj. duzi).
text -za upis teksta od odredenog mesta
inxy -za rad se kursorom
mark -za crtanje ne "punog" poligona, već njegovih temena.Broj temena mora biti paran broj, od 2 do 256.
gmod -za brisanje grafičke slike, kao i za izbor grafičkog moda.[6]
gsx -za proveru da li je GSX-sistem instaliran; dalje za direktno pozivanje funkcija GSX-sistema,koje nisu uključene kao prološki grafički predikati.

Pomenimo još da fill i mark predikati se takođe koriste i za podešavanje "atributa" znakova, boje i dr. Tako, ako u gornjem programu ispred fill-formule umetnemo formulu

fill(2,3)

onda nacrtani poligon će biti na poseban način "osenčen".
Evo primera za text-predikat. Na pitanje

```
?-gdev(1),fill([100,100,100,3000,3000,3000,3000,100]),
    text([100,120],'Hej'),get0(Ch),gdev(0).
```

na ekranu će se pojaviti pravougaonik i u njemu od mesta (100,200), u tom redu, tekst 'Hej'.

Zadatak 11.21. Uočimo program

```
((a 1))
((b _x)(a _x)(SUM _x 1 _x1)(c _x1))
((a 2))
((c 2))
```

i postavimo ovo, očigledno, neprološko pitanje:
Koje od relacija _rel datog programa zadovoljavaju uslov:

(_rel 1)

Da li se prološki može rešiti taj i slični "relacijski" problemi, odnosno oni u kojima se traže relacije koje zadovoljavaju date uslove.

Rešenje. Dato pitanje bi se slobodnije moglo ovako napisati

?((_rel 1)(PP _rel)FAIL)

Da li Prolog može odgovoriti na njega ? Zaista, ako bismo to pokušali već na samom početku bi se pojavilo računanje formule

(_rel 1)

pri čemu _rel nema nikoju vrednost, i Prolog bi prekinuo algoritam sa porukom o grešci.

Osnovna zamisao rešenja koje ćemo dalje opisati je ova:

Označimo tekući program sa P. Na izvestan način od njega ćemo napraviti novi program, tzv. sverel od P, kraća oznaka sverel(P) i to tako da ćemo od svakog članka iz P napraviti po jedan članak iz sverel(P).To pravljenje je opisano ovim zamenama:

((a 1))

[6] Tome služi pitanje ?-gsx.

```
        ----> ((sverel a 1))

    ((b _x)(a _x)(SUM _x 1 _x1)(c _x1))
        ---->((sverel b _x)(sverel a _x)(SUM _x 1 _x1)(sverel c _x1))

    ((a 2))
        ----> ((sverel a 2))

    ((c 2)
        ----> ((sverel c 2))
```

Znači, program sverel(P) se sastoji iz tih sverel-članaka.
Ako pretpostavimo da su ti članci nekako napravljeni i dodati tekućem programu P, onda pitanje postavljeno u zadatku se može proloski ispravno postaviti ovako

```
(Σ)          ?((sverel _rel 1)(PP _rel)FAIL)
```

i jasno je da će se kao odgovori pojaviti a, b.

Inače, odabrano je upravo ime sverel, jer u toj ideji se

> sve relacije polaznog programa ujedinjuju ("svereluju") u jednu relaciju.

Međutim, sada je osnovno pitanje kako napraviti sverel(P) ? Evo kako je to urađeno u programu koji ćemo ubrzo navesti:

> Prvo, tekući program se snimi pod imenom PRIV777.LOG. To ime je odabrano kao uslužno.
> Drugo, učita se fajla SVEREL.LOG koja, kratko rečeno, sadrži sverel-program, koji "obavlja glavni posao".
> Treće, pomoću sverel-programa se postupno, korak za korakom, obavlja "sverelovanje" članaka iz fajle PRIV777.LOG. Detalje kako se to radi ćemo objasniti malo kasnije. Sverelovani članci se privremeno upisuju u novu fajlu imena PRIV888.LOG.
> Četvrto, konačno se ta fajla učita tako da tekući program pored starih članaka sadrži i njihove "sverele".
> Naravno iza toga korisnik može postaviti neko "relacijsko"pitanje poput pitanja (Σ).

Sada navodimo sadržaj fajle SVEREL.LOG :

```
((ajde)
      (KILL (Sver sve svefor dobro citaj ajde))
       (SAVE "PRIV777.LOG")
        (LOAD "SVEREL.LOG")
         (OPEN "PRIV777.LOG")
          (CREATE "PRIV888.LOG")
           (citaj "PRIV777.LOG")
            (CLOSE "PRIV777.LOG")
             (CLOSE "PRIV888.LOG")
              (KILL (Sver sve svefor dobro citaj ajde))
               (LOAD "PRIV888.LOG")
      (PP Sverelovanje zavrseno))

((citaj _X)(READ _X _Y)(sve _Y _Z)
            (WRITE "PRIV888.LOG" (_Z))(citaj _X))
((citaj _X))
```



```
((svefor (NOT|_A) (NOT|_A1))(svefor _A _A1))
((svefor (_A _B) (_A _B1))
         (CON _A)(ON _A (ADDCL DELCL CL ?))(sve _B _B1))
((svefor (OR _A _B) (OR _A1 _B1))(sve _A _A1)(sve _B _B1))
((svefor (IF _A _B _C) (IF _A1 _B1 _C1))
                       (svefor _A _A1)(sve _B _B1)(sve _C _C1))
((svefor (FORALL _A _B) (FORALL _A1 _B1))
                        (sve _A _A1)(sve _B _B1))
((svefor (ISALL _A _B|_C) (ISALL _A _B|_C1))(sve _C _C1))
((svefor (_A|_B) (sverel|(_A|_B)))(dobro _A))
((svefor _A _A))

((dobro _A)(CON _A)(NOT SYS _A))

 ((sve ((_A _B)) ((_A _B1)))
           (CON _A)(ON _A (ADDCL DELCL CL ?))(sve _B _B1))
 ((sve (_X|_Y) (_P|_Q))(svefor _X _P)(sve _Y _Q))
 ((sve () ()))
```

U tom programu osnovnu ulogu imaju predikati citaj i sve. Naime, tokom algoritma uslugom citaj-članaka se redom čitaju članci iz fajle PRIV777.LOG. To se bukvalno obavlja pomoću formule (READ _X _Y), gde _X je upravo ime te fajle. Svaki dobijeni članak _Y se "svereluje" pri "računanju" formule (sve _Y _Z). Taj _Z je sverel od _Y. Čim se napravi _Z on se upisuje u fajlu PRIV888.LOG.
Preostaje da objasnimo kako se "računa" (sve _Y _Z). Slobodnije rečeno to je unekoliko slično sa računanjem formalnog izvoda date formule, odnosno to je primerak "simboličkog programiranja". Pri tome se koristi okolnost da se u Micro-prologu svi objekti prave iz I-termova, pa se predikat sve definiše rekurzivno po dužini terma. U stvari, se pojavljuje još jedan predikat svefor, pri čemu ukratko rečeno:

> sve se odnosi na sverelovanje članaka, ili "njihovih repova",
> dok svefor se odnosi na sverelovanje formula.

Evo nekoliko primera sverelovanja, tj. računanja sve-formula:

1) Neka je Clanak neki dati članak. Tada je jasno ako treba da "svereluje" članak oblika ((ADDCL Clanak)), onda sverelovanje ne sme da "uhvati" predikat ADDCL. Znači, tu treba da bude ovakav rezultat

   ```
   ((ADDCL Clanak1))
   ```

   gde je Clanak1 upravo sverel od Clanak.
   To što rekosmo upravo poručuje prvi sve-članak. Inače, podsećamo da je ON u Micro-prolog ugrađen predikat za "biti element od".

2) Uočimo sada ovaj običan članak

   ```
   ((a 1)(b 2))
   ```

   i potražimo _X tako da važi

   ```
   (sve ((a 1)(b 2)) _X)
   ```

   Sada se odmah upošljava drugi sve-članak, prema kome imamo:
   _X je oblika (_P|_Q) pri čemu treba da važe formule

   ```
   (svefor (a 1) _P)
   ```

(sve ((b 2)) _Q)

tj. treba da nademo "svefor" za (a 1) i dalje "sve" za rep, odnosno za ((b 2)). Na osnovu predposlednjeg svefor-članka odmah se dobije da _P je (sverel a 1). Dalje za formulu(sve ((b 2)) _Q) opet se upošljava drugi sve-članak i dobije se

_Q je oblika (_R|_S) pri čemu treba da važe formule
(svefor (b 2) _R)
(sve () _S)

Lako se dobije _R je (sverel b 2), a _S , prema poslednjem sve-članku je (). Tako konačno za _X imamo:

_X=(_P|_Q)
=((sverel a 1)|_Q)
=((sverel a 1)|((sverel b 2)|()))
=((sverel a 1) (sverel b 2)))

3) Neka Clanak bude ((a _x)(IF (EQ _x 1) ((PP 1)) ((PP Nije 1)))). Potražimo _X tako da važi (sve Clanak _X).
Slično prethodnom :

_X je oblika (_P|_Q), gde treba da važe formule
(svefor (a _x) _P) i (sve ((IF ...)) _Q).

Tako, _P je (sverel a _x), dok za drugu formulu moramo uposliti drugi sve-članak, po kome dobijemo:

_Q je oblika (_R|_S) pri čemu treba da važe formule
(svefor (IF (EQ _x 1) ((PP 1)) ((PP Nije 1))) _R)
(sve () _S)

_S je očigledno jednako (), dok za traženje _R moramo uposliti četvrti svefor-članak. Prema njemu:

_R je oblika (IF _for1 _CL1 _C12), gde treba da važe formule
(svefor (EQ _x 1) _for1)
(sve ((PP 1)) _C11)
(sve ((PP Nije 1)) _C12)

Dalje, lako se dobije _for1 je (EQ _x 1), jer EQ je u Prolog ugrađen predikat, odnosno on je "sistemski" predikat, tj. za njega važi formula (SYS EQ) pa se u traženju _for1 koristi poslednji svefor-članak. Slično, lako se dobije _C11 je ((PP 1)), dok _C12 je ((PP Nije 1)). Inače tu se usput koristi okolnost da je PP "sistemski" predikat tj. da važi (SYS PP).

Na osnovu svega lako se dobije

_X=((sverel a _x)(IF (EQ _x 1) ((PP 1)) ((PP Nije 1))))

Na kraju ukratko ponovimo kako se koristi dati sverel-program, po pretpostavci da je on smešten u fajli imena SVEREL.LOG:

Najpre se od strane korisnika ili ukuca neki program ili se to učini učitavanjem (preko LOAD-predikata) neke fajle koja sadrzi program. Označimo taj tekuci program sa P. Zatim se sa LOAD SVEREL uključuje pomenuta fajla za sverelovanje. Ona se pušta u dejstvo pitanjem

?((ajde))

čijim raspravljenjem se obavlja sverelovanje programa P.

Nakon toga korisnik može postaviti neko "relacijsko" pitanje poput(Σ).

Završavamo navođenjem još jednog primera korišćenja SVEREL-programa. Neka program P sastoji iz ovih fakt-članaka

```
((fakt 0 1))
((fakt _n _m)(LESS 0 _n)
             (SUM 1 _nn _n) (fakt _nn _mm) (TIMES _n _mm _m))
```

Sverelovanjem tog programa nastaje sledeći program

```
((sverel fakt 0 1))
((sverel fakt _n _m)(LESS 0 _n)
             (SUM 1 _nn _n)(sverel fakt _nn _mm)(TIMES _n _mm _m))
```

Primetite, što je bitno, sverelovanje ne "hvata" nijednu od sistemskih relacija. Primera radi, u vezi sa prethodnim programom smemo postaviti i ovo pitanje

```
?((sverel _rel 6 120)(PP _rel))
```

i kao odgovor cemo dobiti: fakt.

Zadatak 11.22. Pretpostavimo da su nam date neke term_jedinke kao a, b, c, d i da hoćemo da od njih napravimo neki *-term, ali koji bi bar u nekom smislu bio "slučajan". Možda je jedan takav term ovaj: ((a*(c*a))*(d*b)) ?

Rešenje (u LPA-prologu). Koristimo irand-funkciju koju smo koristili u Zadatku 11.18. Algoritam -"utisnut" u program koji dalje navodimo - se može kratko opisati ovako:

Na polazu imamo izvesnu listu L = [a1, a2, ..., ak] zadanih term-jedinki. U prvom koraku u rasponu 0...k-1 uslugom irand-a biramo dva slučajna broja p, q. Neka je, recimo, p<q. Tada listu L zamenjujemo listom L1 dobijenom iz nje tako što joj se "skloni" q-ti član i uz to kao novi p-ti član se uzme ovaj *-term, napisan kao lista: [ap, *, aq]. Slično se čini i kad q<p, tada se p-ti uklanja, a za novi q-ti se uzima lista [aq, *, ap]. Ukoliko, p=q, onda se ne uklanja nijedan član, a za novi p-ti član se uzima lista [ap, *, ap].

Napravivši listu L1, u narednom koraku nju uzimamo kao polaznu i slično nastavljamo dalje . Algoritam se zaustavlja ukoliko se desi da smo stigli do jednočlane liste; ona je traženi slučajan term.

Program glasi:

```
duz([],0).                          % duz-predikat je za odredivanje dužine
duz([A|B],X):-duz(B,Y),X is Y+1.    %liste

slucaj(A,B):-duz(A,M),sluc(M,A,B).

sluc(1,A,A).
sluc(M,A,B):- P is irand(M), Q is irand(M), skupi(P,Q,A,AA),
              P=Q -> sluc(M,AA,B); M1 is M-1,sluc(M1,AA,B).

skupi(P,P,A,B):-obradi(P,P,A,B).
```

Slično, važe formule (SYS NOT), (SYS IF), (SYS FAIL) i slično, ali ne važi (SYS a).

```
skupi(P,Q,A,B):-P<Q,obradi(P,Q,A,B).
skupi(P,Q,A,B):-obradi(Q,P,A,B).

obradi(0,Q,X,Y):-spoj(Q,X,Y).
obradi(P,Q,[A|X],[A|Y]):-P1 is P-1,Q1 is Q-1, obradi(P1,Q1,X,Y).

spoj(0,[A|B],[[A,*,A]|B]).
spoj(N,[A|AA],[[A,*,C]|B]):-N1 is N-1,izvuci(N1,AA,[C|B]).

izvuci(0,X,X).
izvuci(N,[A|B],[C,A|X]):- N1 is N-1,izvuci(N1,B,[C|X]).
```

Program se pokreće pitanjem oblika ?-slucaj(A,B). gde je A zadana lista, a B je promenljiva, koja će "zapamtiti" rezultat, tj. slučajan *-term. Najpre se izračunava M, tj. dužina liste A, a potom se pomoću sluc(M,A,B) izračunava B. Pri računu te sluc-formule se dešava upravo ono gore rečeno o algoritmu, i AA je nova ulazna lista za sluc-formulu. Pri pravljenju AA se koriste skupi-članci. Inače skupi-predikat je definisan pomoću predikata obradi. A taj predikat -kratko rečeno- je "zadužen" da u listi A prvo dođemo do p-tog do p-tog člana, a onda uslugom spoj-predikata da obavimo "spajanje p-tog sa q-tim" i izbacimo q-ti.

Zadatak 11.23. Neka je recimo zadan ovaj MICRO-prološki program Prog

```
((p)(a))
((p)(q)(r))
((p)(r)(s))
((q)(a)(b))
((a)(b))
((b)FAIL)
((r)(s)(t))
((s)(u))
((u))
((t)(s))
```

Ako se postavi neko prološko pitanje, kao ?((p)), Prolog će tokom njegovog raspravljanja usput na svoj način dokazati neke formule, koje su naravno logičke posledice datog programa Prog. Kako saznati šta je sve na takav način (prološki) dokazano?

Rešenje. Jedna zamisao je da dati program Prog zamenimo ovim drugim programom Prog' osposobljenim za takav zadatak:

```
((p)(a)(zapisi a)(zapisi p))
((p)(q)(zapisi q)(r)(zapisi r)(zapisi p))
((p)(r)(zapisi r)(s)(zapisi s)(zapisi p))

((q)(a)(zapisi a)(b)(zapisi b)(zapisi q))

((a)(b)(zapisi b)(zapisi a))

((b)FAIL)

((r)(s)(zapisi s)(t)(zapisi t)(zapisi r))

((s)(u)(zapisi u)(zapisi s))

((u)(zapisi u))

((t)(s)(zapisi s)(zapisi t))
```



```
((zapisi _X)(PP (_X)))
```

U tom programu se koristi pomoćni predikat zapisi, čiji "zadatak" je da ispiše usput dokazanu formulu. Recimo, na pitanje ?((r)) na ekranu će se štampati (s), (t), (r) -ali neke i više puta, svaki put kad budu dokazane. ne. Međutim, ako umesto ispisa želimo da dokazane formule budu dodate u obliku (elementarnih) članaka, onda gornji zapisi -članak treba zameniti sa ova dva

```
((zapisi _X)(NOT CL ((_X))))(ADDCL ((_X)))) .
((zapisi _X))
```

Njihovom uslugom će se ,ali samo **novodokazana**[1] formula dodati u obliku odgovarajućeg članka.

12. PREGLED ZNAČAJNIJIH RELACIJA Micro, Arity ,LPA - PROLOGA

Pregled se odnosi na, u Prolog, ugrađene relacije(predikate). Prvo napomenimo da premda je Prolog u osnovi relacijski jezik, neke od njegovih zvaničnih relacija(predikata) u stvari nisu prave relacije. Primera radi, formulom

```
write('Milan'),     odnosno    (P Milan)
```

se nalaže, "nareduje" da se nešto uradi, odnosno upravo da se reč Milan ispiše na ekranu. U Prologu, inače, ima priličan broj sličnih "uradi" predikata. Takvi su:

```
read, assert, consult,...; R,ADDCL,LOAD   itd.
a takođe i "predikati" !,fail, odnosno /,FAIL.
```

Dalje ističemo, da ćemo se u izlaganjima o svakoj posebnoj relaciji (predikatu) uglavnom držati ovakvog rasporeda:

Prvo ćemo navesti osnovni način pisanja u obliku formule; i uz to dati kako se obično čita ta formula.
Dalje, upotrebom znakova +,- , na način koji objašnjavamo, ćemo opisati prirodu relacije, u smislu da li je proverna, da li, i po kojim argumentima, je generatorna.

Evo kao uzor jedan konkretran primer iz Micro prologa, koji se odnosi na dobro poznatu relaciju SUM. Objašnjenje o toj relaciji će uglavnom sadržati ove zapise:

```
 (SUM A B C)          C je A+B
-(SUM + + +)
 (SUM + + -)
 (SUM + - +)
 (SUM - + +)
```

Evo kakav je njihov smisao . U prvom redu je prvo navedena uobičajena SUM-formula, gde su A,B,C argumenti. Desno je naveden način čitanja te formu-

[1] U tu svrhu nam pomaže predikat CL- o člancima.

le. U drugom redu je rečeno da ako su sva tri argumenta A,B,C zadana, onda je izračunljiva njihova SUM-formula; rezultat je naravno: jeste ili nije . Recimo, vrednost od (SUM 3 4 8) je nije. U trećem redu je navedeno da je je navedeno da je formula (SUM A B C) generativna po C, tj. za zadane A, B se može izračunati C. Na primer, pri računanju vrednosti formule

(SUM 3 4 _x)

gde je _x promenljiva koja još nema neku vrednost, _x dobije vrednost 7, a sama formula se sračuna na vrednost jeste. Slično objašnjenje se odnosi i na preostale redove . Može se primetiti da znak - upravo stoji na onim mestima gde treba stavljati promenljive, čije vrednosti se inače dobijaju tokom računanja dotične formule.
U daljem ćemo argumente označavati sa A,B,... ili upotrebom običnih reči u uglastim zagradama. Recimo, assert(<clanak>).
Takođe ističemo da za svaki od prološkim predikata koji ima neke argumente po pravilu je definisan pod izvesnim ograničenjima za njih. Recimo nemoguće je "računanje" ovih formula

(SUM pera 3 6), assert(56)

jer pera nije broj, a nije ni promenljiva, dok 56 nije nikakav članak. U takvim slučajevima Prolozi uglavnom prijavljuju grešku i čak prekidaju prološki algoritam[2]. Međutim da izlaganje ne bi bilo previše dugačko, u opisima koje dalje navodimo pomenuta ograničenja u vezi sa argumentima obično sem ponekad, nećemo posebno isticati. Sada prelazimo na opise predikata.

12.1 M i c r o p r o l o g

ABORT ili (ABORT)
To nije prava relacija. Kad u prološkom algoritmu na nju dođe red, čitav algoritam se završava[3]. Primer:

```
((ajde)(PP Daj)(R _x)
     (IF (EQ _x 5) (ABORT) ((PP Evo _x)(ajde))))
```

Ako na pitanje ?((ajde)) za _x damo 5 algoritam će se završiti.

(ADDCL <clanak>) ili (ADDCL <clanak> <mesto>)
(ADDCL +), (ADDCL + +)
Nije prava relacija, ima smisao :
Neka je **<clanak>** dati članak. U tekućem programu uočimo sve članke sa imenom istim kao taj **<clanak>**. Tada sa (ADDCL <clanak>) taj se članak dodaje tekućem programu i stavlja na kraj takvih članaka, dok sa(ADDCL **<clanak>** **<mesto>**) on se među njih stavlja na mesto **<mesto>**.
Primer: Neka je P

```
((a 2)(b 6)(c _x))  ((a 2 3))
((b 2 4)(c _x))  ((a 6))
```

Tada nakon računanja formula (ADDCL ((a 33))), (ADDCL ((a 4)(b 5)) 2)
P se preobraća u ovaj program

```
((a 2)(b 6)(c _x))  ((a 4)(b 5))  ((a 2 3))
((b 2 4)(c _x))  ((a 6))  ((a 33))
```

[2] Doduše to ponašanje zavisi od verzije Prologa. Tako Arity će "ćutke" prihvatiti "formulu" assert(56), dok LPA će je sračunati na nije.

[3] Znači taj abort slobodnije rečeno je "izađi iz sklopa u kome si i završi algoritam".

(BINTERM <putanja-fajle> <podatak>)
(BINTERM + -)
BINTERM je osnovni predikat za čitanje podataka iz binarne fajle zadanog imena (odnosno putanje). Videti Zadatak 11.15.

(BWRITE <putanja-fajle> <lista-podataka>)
(BWRITE + +)

BWRITE je osnovni predikat za upis podataka (u obliku liste) u fajlu datog imena, koja je najpre napravljena. Videti Zadatak 11.15.

(CHAROF <znak> <broj>)
-(CHAROF + +), (CHAROF - +), (CHAROF + -)
Smisao: Broju <broj> kao ASCI-kodu odgovara znak **<znak>**.
Primeri: Formula (CHAROF A 65) je tačna, jer 65 je ASCI-kod znaka A.

(CHDIR <dir>)
(CHDIR -), (CHDIR +)
Nije prava relacija, smisao blizak komandi cd u DOS-u. Recimo, pri računu ?((CHDIR _x)(PP _x)) na ekranu se štampa ime direktorije u kojoj se nalazimo, dok pri računu ?((CHDIR <dir>)), gde je <dir> putanja izvesne direktorije idemo u tu direktoriju. Primeri:

```
?((CHDIR "\JEZICI"))     Za prelaz u direktoriju JEZICI, koja je prva
                         poddirektorija "vrha".
?((CHDIR "\JEZICI\PROLOG"))  Za prelaz u direktoriju PROLOG
```

(CL A) ili (CL A B C)
O predikatu CL videti tekst na kraju tačke 4.

(CLOSE A) Zatvori A
-(CLOSE +)
Nije prava relacija. Služi za zatvaranje već prethodno otvorene fajle imena A. Takođe se koristi i za "zatvaranje" prozora, što praktično znači da prethodno "otvoren", tj. postavljen prozor nestaje sa ekrana.

(CLS <ime-rel> <arg> <repclanka> <mesto>)
(CLS + - - +)
Za dato ime članka, čak i onih ugrađenih u Prolog, i za dat njegov redni broj pomoću CLS-predikata su odredljivi:
<arg> -lista argumenata glave članka
<repclanka> -rep članka.
Primeri: Neka program P sadrži članke

```
((a 1)(b 2)(c 3))
((a _x _y)(d _x _y)(e _y _x))
```

Tada pri računu formule

```
(CLS a _X _Y 2)
```

promenljive _X,_Y će dobiti ove vrednosti
(_x _y), odnosno ((d _x _y)(e _y _x))
jer drugi a-članak ima oblik: (Glava|Rep), gde Glava=(a|(_x _y)), tj. ima oblik ((a|<arg>)|Rep).
Drugi primer:
Na ?((CLS PP _x _y 1)(PP _x _y))
u vezi sa predikatom PP, na ekranu će se štampati
_A kao argument, i ((WRITE "WND:" _A)) kao rep PP-članka.

(CON A)
-(CON +)
Smisao: A je konstanta, tj. konstantska reč.
Primer: (CON pera) je tačno, (CON 23), (CON (1 2)) su netačne.

(CREATE <ime>)
(CREATE +)
Nije prava relacija. Pri računu formule (CREATE <ime>) pravi se nova fajla imena <ime> i ujedno se otvara. Ako već postoji fajla istog imena, ona se ne gubi već se preimenuje u .BAK oblik.

(CRMOD <ime> <lista-izvoznih> <lista-uvoznih>
(CRMOD + + +)
CRMOD je predikat za pravljenje tzv. modula. Bez pravljenja ikojeg novog modula u Micro-prologu se radi u osnovnom modulu imena &. Modul je u neku ruku kao zaseban "proloski svet". Svaki modul sadrži izvesne svoje članke, imena recimo označenih sa a,b,c, .Neka od tih imena mogu da budu tzv izvozna, što znači da članak takvog imena može biti korišćen u osnovnom modulu &, kao i ma kom drugom ako pripada listi njegovih uvoznih imena. Recimo, sa ?((CRMOD pera (a b) (p q))) se pravi modul imena pera, lista izvoznih imena je (a b), a uvoznih (p q). Već postavljanjem tog pitanja na ekranu se pojavi ovakva slika

pera

tj. znak & se zameni imenom novog modula u koji smo prešli. Tu sada možemo uvesti neke a-, b-članke, recimo kao

```
pera((a _x)(PP Evo _x))
pera((b _x _y)(a _x)(a _y))
```

U modulu pera možemo uvesti i članke nekog drugog imena, neprisutnog u izvoznoj listi. Recimo, sa

```
pera((c 0 1))
pera((c _x _y)(LESS 0 _x)(SUM _x _y 5))
```

su uvedena dva c-članka. To su primeri članaka sa tzv. lokalnim imenima, oni su "prepoznatljivi" samo u modulu u kome su uvedeni.
Modul pera ima p,q kao uvozna imena. To znači taj model može koristiti p-, q- članke, koji su uvedeni ili u osnovnom modulu & ili u nekom drugom modelu, u kome su p,q prisutni u listi izvoznih imena.
U svakom od modula možemo koristiti sve sistemske predikate Prologa. Tako, ako smo u pera-modulu, onda sa LIST ALL se "listaju" njegovi članci, sa ADDCL, DELCL se dodaju, brišu članci u tom modulu, itd. Ako hoćemo da iz modula pera pređemo u neki drugi modul onda koristimo CUMOD predikat (videti). Tako sa CUMOD & vraćamo se osnovnom modulu.
Uopšte, u vezi sa modulima su ovi predikati CRMOD, CUMOD, MDICT, DICT.
Napomenimo da o modulima, doduše u LPA-prologu- što je slično- se podrobnije izlaže u tački 10.

(CRWIND <ime> <red> <stubac> <broj-redova> <broj-stubaca>)
(CRWIND + + + + +)
Nije prava relacija. Pri računu CRWIND-formule, uz datost svih argumenata pravi se prozor pod imenom <ime> čiji levi gornji vrh je u redu <red>, i stubcu <stubac>, koja ima redova <broj-redova> i stubaca <broj-stubaca>. Sam ekran se shvata kao prozor čije ime je &: levi gornji vrh ima koordinate 0,0, broj redova je 23, a broj stubaca je 78.
Primer. Pri raspravi pitanja ?((CRWIND Primer 10 10 10 10)(PP Zdravo)) pra-

vi se prozor[4] imena Primer, kao sto je niže prikazano

i uz to u prozoru se ispisuje reč Zdravo. U daljem radu taj prozor- dok ga ne promenimo- postaje "novi ekran", tj. sve se u njemu "odigrava". S tim u vezi primetite da je osnovni znak & Micro-prologa smešten u napravljeni prozor. O prozorima još videti u opisu CUWIND-predikata.

(CUMOD <ime-modula>)
(CUMOD +), (CUMOD -)
CUMOD je jedan od predikata za module (eventualno, najpre videti o CRMOD predikatu). Primeri[5]:

Sa ?((CUMOD pera)) iz tekućeg modula prelazimo u model imena pera.
Sa ?((CUMOD _x)(PP _x)) na ekranu se štampa ime modula u kome smo.

(CURSOR <ime> <red> <vrsta>)
(CURSOR + - -), (CURSOR + + +)
Nije prava relacija. Odnosi se na kursor u okviru prozora zadanog svojim imenom <ime>. Recimo, na pitanje oblika

?((CURSOR &: _x _y)(PP _x _y))

na ekranu će se štampati trenutne koordinate kursora. Tu je &: ime koje odgovara običnom ekranu. Međutim, na pitanje oblika

?((CURSOR &: 5 23)(PP Pera))

će se prvo kursor preseliti na mesto sa koordinatama 5,23 i počev od tog mesta u njegovom redu će biti štampana reč Pera.

(CUWIND <ime>)
(CUWIND +), (CUWIND -)
Predikat CUWIND služi za[6] prelaz iz prozora u prozor, kao i za određivanje imena prozora u kome se nalazimo. Recimo, uz pretpostavku da se nalazimo u nekom prozoru Pr i hoćemo da pređemo u već postojeći[7] prozor imena <ime>, onda se to postiže računanjem formule (CUWIND <ime>). Primera radi, računanjem formule (CUWIND &:) se iz prozora Pr vraćamo u osnovni ekran. Ako ujedno želimo da stari prozor Pr nestane sa ekrana, onda moramo to po-

---

4 Primetimo da prozor ne bi bio napravljen, ukoliko ne bi mogao " da se smesti na ekran". Ovde to nije slučaj jer važe nejednakosti
10+10<24, 10+10<79.

5 CUMOD predikat je unekoliko sličan prozorskom predikatu CUWIND.

6 Pogledajte i opis za CRWIND.

7 Ako je polazni prozor običan ekran, onda je on "već postojeći", ali u drugom slučaju polazni prozor je morao prethodno biti napravljen uslugom CRWIND -predikata.

stići upotrebom CLOSE-predikata. Naime računanjem formule (CLOSE <ime>), gde zamišljamo da je <ime> ime prozora Pr, će se skloniti, obrisati prozor Pr. Druga mogućnost CUWIND-predikata je sledeca. Pretpostavimo da se nalazimo u nekom prozoru i želimo da otkrijemo njegovo ime, primera radi koje je ime ekrana, kao prozora ? Dosta je u tu svrhu postaviti ovo pitanje

```
?((CUWIND _X)(PP _X))
```

i na ekranu će se pojaviti &: kao traženo ime.
O prozorima dodajmo još da se u slučaju raznih ispisnih predikata kao P, PP i dr. koristi ime "WND:" kao opšte za ma koji prozor. Tako, predikat PP se u Micro-prologu ovako definiše

```
((P|_X)(WRITE "WND:" _X))
```

što znači da ga smemo koristiti kad se nalazimo u ma kom prozoru.

(DATE <dan> <mesec> <godina>) ili (DATE <dan> <mesec>)
(DATE - - -), (DATE - -)
Predikat pomoću koga možemo saznati dan, mesec i godinu, odnosno dan i mesec, koje trenutno računar pamti.
Primer. Na pitanje ?((DATE _x _y _z)(PP _x _y _z)) na ekranu će se ispisati datum.

(DEF <rel>) **<rel> je ime u tekućem programu uvedene realacije, ili ime u Prolog ugradjene relacije.**
-(DEF +)
Pomoću predikata DEF možemo saznati da li u tekućem programu ili u jeziku Prolog je nekim članom uvedena, definisana relacija imena <rel>.
Primer. Formula (DEF IF) je tačna jer IF je prologova relacija. Formula (DEF a) će biti tačna ukoliko tekući program ima bar jedan a-članak, kao ((a 1) (b 3)) i sl.

(DEL <fajla>)
(DEL +)
Pomoću DEL predikata se može izbrišе fajla imena <fajla>. Ako je to fajla iz direktorije u kojoj je Prolog, onda navodimo samo ime fajle, a inače moramo zadati čitavu putanju fajle.
Primer:

```
?((DEL "PERA.LOG"))          Za brisanje fajle PERA.LOG iz Prologove direktorije
?((DEL "\JEZICI\TC2\CPRIM1.C"))  Za brisanje fajle CPRIM1.C iz direktorije TC2 koja je poddirektorija direktorije JEZICI.
```

(DELCL <clanak>) ili (DELCL <ime-clanka> <mesto>)
(DELCL +) (DELCL + +)
Služi za brisanje odgovarajućih članaka. Da bi se iz tekućeg programa uklonio neki članak, kao recimo ((a _x)(b 8)) koristi se formula oblika (DELCL <clanak>), gde <clanak> je ili potpuno naveden članak ili naveden samo njegov oblik, odnosno dat u vidu ((ime|_X)|_Y)), gde je sa ime označeno ime članka[8]. Tako, navedeni članak se može obrisati ili ovako

```
?((DELCL ((a _x)(b 8)))),        ili ovako
?((DELCL ((a|_X)|_Y))))          Pazite, tako se briše prvi a-članak, pa
```

[8] Tj. ime relacije njegove glave.

taj način se može upotrebiti ako dati članak je takav.

Oblik (DELCL <ime-clanka> <mesto>) se slično koristi kao prethodni, ali onda se zadaju ime članka za brisanje, kao i njegovo mesto u nizu svih imečlanaka. Recimo, sa ?((DELCL a 5)) se briše 5-ti a-članak.

(DICT <ime-modula> <izvozni> <uvozni>|<ostali>)
(DICT|_)
DICT, skraćeno od dictionary (rečnik), je predikat koji je "zadužen" da u okviru nekog modula[9] zapamti
ime, listu izvoznih, listu uvoznih, listu lokalnih, kao i razne usput korišćene konstante, imena promenljivih i drugo.
Primeri:
1) Recimo, da se nalazimo u glavnom modulu &. Tada, ako na samom početku postavimo pitanje
   ?((DICT|_x)(PP _x))
   dobićemo ovu listu
   (& () () _x "<USER>" "?ERROR")
   gde prvi član & je ime modula, liste uvoznih i izvoznih su prazne, dalje dolazi _x jer njega smo upravo iskoristili u postavljanju pitanja. Konačno dolaze konstante "<USER>", "?ERROR"" koje u Micro-prologu imaju posebnu ulogu i značaj.
   Međutim, ako ubrzo uvedemo neki članak imena ime, onda će predikat DICT i to zapamtiti, i sl. dalje.
2) Ako smo napravili modul nekog imena i ako uđemo u taj modul onda na gornje pitanje na ekranu će se pojaviti ovakva lista
   (jova (a b c) (r s) _x)
   gde zamišljamo da jova je ime modula, izvozna lista je (a b c), uvozna lista je (r s), dok _x je ono _x iz postavljenog pitanja.
   Kao što se vidi kad se nalazimo u nekom modulu onda pomoću DICT-predikata recimo možemo saznati uvoznu i izvoznu listu.

(DIR <oblik-imena-fajli> <spisak-fajli>)
(DIR + -)
DIR predikat se koristi potpuno slično kao DOS-ova dir-komanda. Recimo, na pitanje

```
?((DIR "\JEZICI\*.BAT" _X)(PP _X)
```

će se u obliku liste dati spisak svih fajli koje se nalaze u direktoriji JEZICI i čije ime ima "ekstenziju" BAT.

(EOF <ime>)
(EOF +)
U radu sa fajlama se koristi taj predikat EOF (kraj fajle). Tada <ime> mora biti zadana putanja fajle, odnosno samo ime ako je fajla u okviru tekuće direktorije.

(EQ A B) **A je ujednačivo sa B**
(EQ + -) (EQ - +) (EQ - -) -(EQ + +)
Predikat EQ je predikat algoritma unifikacije (ujednačavanja). Ako su A, B dva ma koja |-terma, tada pri računu formule (EQ A B) se vrši ujednačavanje A sa B, ako može, i pritom promenljive dobiju odgovarajuće zamene.
Primeri:

[9] Eventualno najpre pogledati o CRMOD- predikatu.

Pitanje ?((EQ 2 3)) Odgovor Nije
Pitanje ?((EQ _x _y), gde _x, _y su promenljive[10].
Odgovor Jeste. Tada _x dobije zamenu[11] _y
Pitanje ?((EQ (1 2 _X 5 6) (1 2 (3 4 5) 5 6))(PP _X))
Odgovor je Jeste. Na ekranu se štampa (3 4 5)

(ERRM <broj> <opis greške>)
(ERRM + -)

ERRM predikat pri zadavanju vrednosti prvog argumenta, koja može biti 0,1,2,3,...,20 u drugom argumentu sadrži opis odgovarajuće greške. Recimo, na pitanje

```
?((ERRM 2 _x)(PP _x))
```

na ekranu će se pojaviti ovaj opis

"Predicate not defined"

Smisao je sledeći. Ako se nekom prilikom tokom proloških algoritma pojavi računanje formule sa čijim imenom nema nijednog članka, onda Micro-prolog prekida algoritam. Prekidu je dodeljen broj 2, a njegov slovni opis je

"Predicate not defined"

Vrednost prvog argumenta je najviše 20; drugim rečima ima ukupno 21-a tzv. "obrojena" greška. Ovim malim programom možete saznati sve slovne opise "obrojenih" grešaka

```
((ajde)(PP Daj broj)
    (R _x)
    (ERRM _x _y)(PP _y)
    (ajde))
```

Program se pokreće pitanjem ?((ajde)). Recimo, ako se _x-u da vrednost 8 dobiće se poruka: Path not found, što se može desiti pri pokušaju otvaranja fajle kojoj je zadana pogrešna putanja. Dalje, za _x=11 dobićemo poruku Break!. To se dešava kad sam korisnik tipkom Del prekine algoritam. Ako sa _x damo neku od vrednosti 21,22,.... dobićemo poruku Unknown error. U vezi sa obrojenim greškama videti i o "?ERROR?" predikatu. Pored obrojenih grešaka, kojima odgovaraju i slovni opisi tokom Proloških algoritma se može pojaviti i neka druga za koju u Prologu nije predviđen broj (pa ni slovni opis). Recimo, može se dogoditi da iz nekog razloga, koji čak može biti slučaj opšte greške proloških algoritma, računar prekine algoritam, izađe iz Prologa i pređe u DOS.

(EXEC <putanja_izvršne_fajle> <lista_parametara> <promenljiva>)
(EXEC + + -)

EXEC predikat služi da se ne izlazeći iz Micro-prologa izvrši neka .COM ili .EXE fajla.
Primeri:

1) Zamislimo da smo recimo pomoću C-jezika napravili fajlu saberi.exe koja ima ulazne podatke oblika
   broj1 operacija broj2
   pa se shodno tome iz DOS-a može na primer ovako izvršiti[12]
   >saberi 5 + 7
   Taj posao se može obaviti i u okviru Micro-prologa postavljanjem ovog pitanja
   ?((EXEC "\CE\SABERI.EXE" (5 + 7) _x))
   gde pretpostavljamo da se fajla saberi.exe nalazi u direktoriji CE. Drugi argument je lista ulaznih podataka[13], a treći je promenljiva, "zadužena" da zapamti da li će se pojaviti neka greška.
2) Sada zamislimo da iz DOS-a hoćemo da saznamo vreme, u obliku čas-minut-sekunda. Kao što znamo to se postiže ovom DOS-ovom komandom
   (*) >time
   To je prilično neobično jer, proverite, među DOS-ovim fajlama nema nijedne kao time.com ili time.exe.
   Ostavimo tu neobičnost i istaknimo da se umesto "nameštenog oblika" (*) u DOS-u može ovako pitati
   >command.com /c time
   Znači, u stvari, time je samo jedan od parametara glavne izvršne fajle računara, odnosno fajle command.com. Istaknimo da je neophodno da prvi ulazni podatak bude /c.
   Shodno rečeno, iz Micro-prologa možemo obaviti isti time-posao postavljanjem ovog pitanja[14]
   ?((EXEC "\command.com" ("/c" time) _x))
   Naravno pretpostavlja se da se command.com nalazi "na vrhu".
3) Uočimo ovaj članak
   ((ajde)(EXEC "\command.com" ("/c" copy "\pera" "mile") _x))
   pretpostavljajući da se fajla imena pera nalazi na vrhu, gde je i command.com. Tada, pitanjem ?((ajde)) će se fajla pera prekopirati u fajlu mile.

(EXIT <broj>)
(EXIT +)

EXIT predikat u osnovi služi za izlaženje iz Micro-prologa i vraćanje u DOS. Navedeni <broj> može biti jedan od brojeva 1,2,3,...,255. Računanjem formule, kao (EXIT 4), se obavi opisano vraćanje. Istaknimo, da se tada i sve korisnikom-otvorene fajle automatski zatvaraju.

FAIL

To je po dogovoru formula čija vrednost je fiksirana i iznosi nije.

(FDICT|<lista-fajli>)
(FDICT -)

Pomoću FDICT predikata možemo naći listu svih fajli, koje su otvorene. Tako zamislimo da smo sa

```
?((CREATE "mile1.log"))
```

i sa

```
?((CREATE "mile2.log"))
```

otvorili dve nove fajle, a da smo sa

```
?((OPEN "pera.log"))
```

otvorili već postojeću fajlu. Tada, na pitanje

```
?((FDICT|_Lista)(PP _Lista))
```

na ekranu će se štampati lista od te tri fajle. Međutim, ako recimo sa

```
?((CLOSE "pera.log"))
```

zatvorimo fajlu pera.log, onda će ona automatski biti sklonjena sa te

---

[10] Tj. još nisu dobili vrednosti.

[11] U stvari, _x i _y su prvo bile različite promenljive, u smislu da imaju različite adrese, a nakon ujednačavanja (EQ _x _y), adresa od _x se menja i ujednačuje se sa adresom od _y.

[12] Tu je > tipičan DOS-ov prompt(znak).

[13] Da je saberi.exe bila bez ulaznih podataka taj argument bismo postavili na ().

[14] Podatak /c je stavljen pod navodnike jer sadrži "DOS-osetljiv" znak /.

liste.

(FLDATA <fajla> <ime> <ekstenzija> <vreme-nastanka> <datum-nastanka> <veličina> <atribut>)
(FLDATA + - - - - - -)
Predikatom FLDATA za fajlu date putanje možemo saznati mnoštvo podataka o njoj, odnosno:
njeno ime, ekstenziju, vreme i dan nastanka, veličinu i atribut, koji određuje da li je ta fajla direktorija, "obična" fajla, fajla samo za čitanje, sakrivena i sl.
Primer: Na pitanje
?((FLDATA "\micro\prolog.exe" _I _E _V _D _B _A)(PP _I _E _V _D _B _A))
na ekranu se štampa:
PROLOG EXE (19 39 20) (26 6 1985) (34 368) 32

(FLUSH <bafer>) Briši bafer <bufer>
(FLUSH +)
U mnogo slučajeva kompjuteri izvesne podatke čuvaju za izvesno vreme, da bi ih kasnije određenim redom koristili. U svrhu takvog čuvanja uopšte služe posebne fajle koje se obično nazivaju baferi ("smestači"[15]). Među baferima Micro-prologa imamo i ove dve:
"BUF:" -tzv. editor-bafer. Svi upisi preko tastature idu u njega, i uz to sa EHO-om, tj. na ekranu se pojavljuje upisna, odnosno ukucana reč.
"TRM:" -bafer tastature, slično kao BUF:, ali upisi u njega idu bez EHO-a, tj. na ekranu se ne pojavljuje upisna reč.
Gotovo u svakom jeziku korisnik u nekim slučajevima, koji se nažalost ne opisuju potpuno, ponekad sam mora da vodi računa o baferima, u smislu da li jesu ili nisu prazni. Za brisanje bafera služi FLUSH-predikat.
Primer: Pretpostavimo, da u nekom programu želimo da na nekom mestu algoritam stoji neko vreme, po želji korisnika, a da se dalje nastavi kucanjem ma koje tipke na tastaturi.
To recimo može biti korisno u radu sa prozorima. Na jednom primeru pokazujemo kako se takva zamisao "privremenog zastoja" može ostvariti.
Uočimo program

```
((ajde)(CRWIND "okvir" 5 10 10 30)
       (PP Jovan)(PP Za izlazak kucati ma koju tipku)
       (FLUSH "TRM:")(GETB "TRM:" _X)(CUWIND &:)(CLOSE "okvir"))
```

Evo šta se dešava na pitanje ?((ajde)):
Prvo se otvori prozor imena okvir sa naznačenim dimenzijama.
Drugo, budući da se PP odnosi na tekući prozor, a to je sada okvir, to se u njemu ispisuje reč Jova.
Treće, ispisuje se poruka o načinu izlaženja. Iza toga je osnovno da se uposli (GETB "TRM:" _X) sa smislom
(*) Čekam da se sa tastature učita[16] _X, ali na ekranu se neće ništa pojaviti(tj. upis bez EHOA).
U stvari, to je ujedno i naša želja. Međutim, ima jedno ALI.
Naime, (GETB "TRM:" _X) radi po opisu (*), ukoliko bafer "TRM:" je prazan. Ali, ako kojim slučajem u njemu ima neka vrednost onda se GETB-formula umesto po (*) računa ovako: iz bafera se uzme postojeća vrednosti veže se za _X. To praktično znači da u takvom slučaju nam se ne ispunjava naš naum.
Da bismo se obezbedili od takvog neželjenog ponašanja GETB-formule moramo podesiti da pre računanja te formule bafer "TRM:" bude prazan. Upravo iz tog razloga je u navedenom programu pre GETB-formule je stavljeno (FLUSH "TRM:").

(FORALL A B) Kad god važi A "uradi" B
(FORALL + +)
O tom predikatu videti u Zadatku 3.11 (rešenje (iii)) i Primeru 4.1.4.

(FREAD <ime-fajle> <lista-formata> <lista-podataka>)
(FREAD + + -)
Videti Zadatak 11.14.

(FWRITE <ime-fajle> <lista-formata> <lista-podataka>)
(FWRITE + + +)
Videti Zadatak 11.14.

(GETB <putanja-fajle> <bajt>)
(GETB + -)
Jedan od osnovnih predikata za "input, read, čitanje" i to bajt po bajt iz fajle sa navedenom putanjom. Videti Zadatak 11.12. Jedan posebno zanimljiv slučaj u vezi sa baferom "TRM:" je izložen pri opisu FLUSH predikata.

(GRNHOL Var Term) ili **(GRNHOL Var1 Term Var2)**
(GRNHOL - +) **(GRNHOL - + -)**
Videti Zadatak 6.18.

(I <var>)
(I -)
Predikat sličan uobičajenom predikatu R, za "input, read, učitavanje", ali sa razlikom da kad dodeljena vrednost je neka promenljiva, onda je njeno ime zapamćeno i uz upotrebu P, PP i sl. predikata može biti štampano. Primer:
Ako na pitanje
?((PP Daj)(I _x)(PP _x))
za _x damo neku od ovih vrednosti 89, _Pera, gde je druga promenljiva, onda na ekranu će se pojaviti 89, odnosno reč _Pera. Kada bi umesto I u tom pitanju stajalo R onda bi se u drugom slučaju štampala adresa promenljive _Pera.

(IF <uslov> <uradi1> <uradi2>)
(IF + + +)
Videti Primer 4.1.2.

(INT <broj>) <broj> je ceo broj
ili **(INT <broj> <rezultat>)** <rezultat> je ceo deo broja <broj>
(INT +), **(INT + -)**
Primeri: (INT 5) je tačno, (INT 6.3 _x) je tačno za _x=6.

(INTERM <putanja-fajle> <term>)
(INTERM + -)
Jedan od osnovnih predikata za čitanje fajle članak po članak. Tako, ako je pera.log prološka fajla, onda računanjem formule[17]
(INTERM "pera.log" _X)

[15] Rekli smo smestači da bismo istakli njihov pravi smisao.

[16] To _X je uslužna promenljiva, i dalje više nema nikakvu ulogu.

[17] Pretpostavljamo da je fajla "pera.log" prethodno otvorena.

_X se veže za prvi članak iz te fajle, pri čemu slično kao kod I-predikata promenljive se pojavljuju preko svojih imena. Recimo, članak kao

```
((a _x)(b _x _y))
```

će se pojaviti bukvalno tako. Za ispis svih članaka date fajle može se koristiti ovaj program

```
((pisi _Ime) (OPEN _Ime)(ispis _Ime))
((pisi _Ime)(CLOSE _Ime))
((ispis _Ime)(INTERM _Ime _X)(PP _X) (ispis _Ime))
```

koji se pokreće pitanjem oblika ?((pisi <ime>)) , gde se zadaje putanja <ime>. Primetimo, da u stvari INTERM predikat nije bitno vezan za prološke članke, odnosno on uopšte ne proverava "člankost". Primera radi, ako uočimo fajlu "\mile" čiji je sadržaj ovaj

```
(p r s)
pera
(a ((b )()))
```

onda sa ?((pisi "\mile")) ćemo dobiti potpuno dobar ispis. Razlog: INTERM predikat odvaja "parčiće" date fajle i pri tom osnovni uslov je da u svakom parčetu bude isti broj levih i desnih zagrada. Ako se desi da naiđe na parče koje ne zadovoljava taj uslov onda na tom mestu prekida dalje traganje i putovanje po fajli.

(INTOK <putanja-fajle> <reč>)
(INTOK + -)

Predikat za čitanje fajle "reč po reč". Tako, ako je "\pera" fajla sa ovim sadržajem

```
Pera ide
u (dan! _a)
```

onda pri računu formule

```
?((OPEN "\pera")(INTOK "\pera" _X)(PP _X)(CLOSE "\pera"))
```

na ekranu će se pojaviti prva reč Pera. Upotrebom programa

```
((pisi _Ime) (OPEN _Ime)(ispis _Ime))
((pisi _Ime)(CLOSE _Ime))
((ispis _Ime)(INTOK _Ime _X)(PP _X) (ispis _Ime))
```

na pitanje ?((pisi "\pera")) na ekranu će se pojaviti sve "reči" te fajle, odnosno

```
Pera  ide  u  (  dan  !  a  )
```

Pri otkidanju reči "razdvajači" su
znak beline, znak novog reda, znaci (,),!
Poslednja tri znaka se takođe ubrajaju u reči.

(ISALL Lista X for1 for2 ... fork) Lista je lista svih vrednosti promenljive X za koje su tačne formule for1 for2 ...fork

(ISALL - + + + ... +)

O ISALL predikatu videti u Zadatku 4.1.5.

(KILL <ime-članka>)
(KILL +)

Predikat za "brisanje" svih članaka sa imenom <ime-članka>. Tako, sa ?((KIL a)) iz tekućeg programa uklanjamo, brišemo sve članke imena a.
Za brisanje "celog programa", tj. svih članaka koristimo isti predikat, ali kao ime damo reč ALL.

(LESS A B) A je manji od B
-(LESS + +)

LESS predikat se može koristiti za poređenje brojeva, kao i reči. Primeri:

```
?((LESS 4 6.7))      Jeste
?((LESS 6 6))        Nije
?((LESS mika pera))  Jeste
```


(LIST <šta>)
(LIST +)

Predikat LIST služi za ekranski ispis
svih članaka datog imena <šta> (Recimo[18]: LIST a)
svih članaka nekoliko datih imena (Recimo: LIST (a b c))
svih članaka tekućeg programa (sa[19]: LIST ALL)
svih članaka modula datog imena

(LISTP <ime> <šta>)
(LISTP + +)

LISTP je jedan od osnovnih sistemskih predikata; pomoću njega su definisani LIST i SAVE. Tako definicija za SAVE glasi

```
((SAVE _x)(CREATE _x)(LISTP _x ALL)(CLOSE _x))
```

Tu se kao _x daje neko ime, recimo "mile.log". Tada se pravi fajla tog imena, dalje pomoću LISTP se u tu fajlu "izlista" ALL, tj. sve iz tekućeg programa i konačno ta se fajla zatvori. Primetimo da smo umesto (LISTP _x ALL) imali (LISTP _x (a b)) onda bi se u tu fajlu "izlistali", tj. snimili samo a- i b-članci tekućeg programa, ako ih ima.
Definicija predikata LIST je slična prethodnoj za SAVE i glasi

```
((LIST _x)(LISTP "WND:" _x))
```

LISTP ima još neke zanimljivosti. Recimo, ako u tekućem programu imamo a-, b- članke onda na pitanje

```
?((CREATE "mile.log")
  (LISTP "mile.log"
         (a b (PP Zdravo) "jovan-mod")
  )
 (CLOSE "mile.log"))
```

se obzirom na drugi LISTP-argument koji je

```
(a b (PP Zdravo) "jovan-mod")
```

dešava sledeće:
Usnimavaju se a- i b-članci, dalje u fajlu "mile.log" se ostvaruje komanda (PP Zdravo), tj. piše reč Zdravo i konačno u "mile.log" se usnimavaju i svi članci modula "jovan-mod"

Primetimo da umesto (PP Zdravo) može biti ma koja prološka komanda, što pruža široke mogućnosti. Recimo, mogli smo staviti (LOAD "pera.log"). Tada pri kasnijem učitavanju fajle "mile.log" usput zbog te LOAD-formule bi se učitala i fajla "pera.log".

(LOAD <putanja-fajle>)
(LOAD +)

Predikat LOAD služi da se tekućem programu[20] dodaju svi članci sadržani u fajli čija putanja je <putanja-fajle>. Recimo sa LOAD PERA, pretpostavljajući da je fajla[21] PERA.LOG član micro-prološke direktorije, se na kraj tekućeg programa dodaju svi članci te fajle.

---

[18] Naravno može i ovako ?((LIST a)).

[19] Može i ovako ?((LIST ALL))

[20] Može biti i prazan.

[21] ili fajla PERA.BIN

Bitno je da pri tom dodavanju vrši se provera "člankosti" i dejstvo LOAD se može prekinuti ako se u putovanju kroz fajlu naide na ne-članak.

(LOGIN <disk>)
(LOGIN +), -(LOGIN +)
LOGIN predikat je osnovni disk-predikat. Recimo, sa
?((LOGIN _X)(PP _X))
na ekranu se pojavi oznaka diska na kome smo, kao C,A, i dr. Dalje,sa
?((LOGIN A))
bez obzira gde smo trenutno prealzimo na disk A.

(LST A) A je lista
(LST +)
Primeri: (LST 5) nije tačno, dok (LST (1 (23 66)) jeste.

(MDICT|<lista otvorenih modula>)
(MDICT|-)
Primer:
Na pitanje ?((MDICT|_x)(PP _x)) će se štampati lista svih modula koje smo otvorili. Tu se ne računa osnovni modul &.
U vezi sa modulima pogledati deo o predikatu CRMOD.

(MKDIR <novi-dir>)
(MKDIR +)
Predikat pomoću koga,slično kao u DOS-u, možemo da napravimo novu direktoriju imena <novi-dir>.Primeri:
?((MKDIR JOVA)) Pravi se micro-prološka poddirektorija JOVA
?((MKDIR "/jezici/fortran") U direktoriji jezici pravi se poddirektorija fortran

(NOT|<formula>)
-(NOT|+)
Predikat proloske negacije. Videti Primer 4.1.3.

(NUM A) **A je broj**
-(NUM +)
Primer: (NUM 5) je tačno, (NUM pera) nije.

(ON <element> <lista>)
-(ON + +) (ON - +)
ON je osnovni član-predikat.Primeri:
Sa ?((ON 5 (1 2 5 7)) saznajemo da li je 5 član liste (1 2 5 7)
Sa ?((ON _X (1 2 5)) _X se vezuje za 1, tj. prvi član liste (1 2 5))
Sa ?((ON _X (1 2 5)(PP _X)FAIL) _X se redom vezuje sa 1,2,5 tj. svim članovima liste (1 2 5).

(OPEN <putanja-fajle>)
(OPEN +)
OPEN je osnovni predikat za otvaranje već postojeće fajle zadanog "punog imena", tj. putanje.

(OR A B)
(OR + +)
Jedan od osnovnih logičkih predikata (ili-predikat). Videti Primer 4.1.1.

(P|<dato>)
(P|+)
P je najraznovrsniji predikat za ispisivanje datog podatka. Ispisivanje je

po pravilu na tekući prozor, često baš na ekran. Recimo, sa
?((P Pera)) ?((P "Pera"))
se u oba slučaja na ekranu ispisuje reč Pera, ali bez prelaza u novi red kao u slučaju PP predikata).Kao što se primećuje znaci navoda se ne ispisuju. Razlog je što oni mogu imati veoma posebnu ulogu. Recimo, da želimo da se navedeni ispis obavi na printeru. Tada se ispred te formule o ispisu mora staviti podatak da se ispis usmerava na printer. To se čini uz pomoć pomoć formule (P "~P"), koja bi mogla da se čita "štampaj control P". Shodno rečenom, pitanjem[22]
?((P "~P")(P Pera))
se reč Pera štampa na printeru. Pored kontrolnog znaka <Ctrl P> mogu se koristiti i neki drugi. Tako sa
?((P "~D" Pera ide u skolu))
se reči Pera ide u skolu štampaju u okviru čija osnovna boja je ona ekranska druga boja("boja pozadine", ili "naličje").Ako nastavimo dalje sa nekim štampanjima ostajemo u takvom načinu ("mode"). Ali, ako nešto kasnije računamo formulu (P "~R") vraćamo se na prethodni, stari način- bez uokviravanja.

(PP|<dato>)
(PP|+)
Predikat PP za ispis podataka <dato> na tekući prozor, recimo ekran. Ako <dato> sadrži znake navoda oni se ignorišu. Na kraju se obavlja prelaz u novi red.

(PQ|<dato>)
(PQ|+)
Predikat PQ za ispis podataka <dato> na tekući prozor, recimo ekran. Ako <dato> sadrži znake navoda i oni se štampaju.Na kraju se obavlja prelaz u novi red.

(PSIWIND <ime> <red> <stubac>) ili
(PSIWIND <ime> <red> <stubac> <broj-redova> <broj-stubaca>)
PSWIND predikat se koristi u vezi sa već postojećim prozorom imena <ime>. Recimo da je ime Izlog. Pomoću PSWIND predikata možemo
1) Saznati koordinate datog prozora.Tako na pitanje
?((PSWIND Izlog _x _y)(PP _x _y))
na ekranu će se ispisati koordinate levog gornjeg ugla prozora Izlog, dok na pitanje
?((PSWIND Izlog _x _y _z _u)(PP _x _y _z _u))
pored njih štampaće se i broj vrsta i broj stubaca.
2) Možemo -bez menjanja veličine- dati prozor pomerati u željeni položaj.Tako, pitanjem
?((PSWIND Izlog 15 20))
se prozor Izlog seli na nov položaj tako da mu levi gornji ugao bude u tački (15,20).

(PUTB <putanja-fajle> <ASCII-kod>)
PUTB predikat se prvo može služiti da se u dati prozor,ekran, fajlu ispiše znak čiji ASCI-kod je <ASCII-kod>
Recimo,
Sa ?((PUTB "WND:" 65)) se na tekući prozor, a to može biti i ekran

---

[22] To smo naravno mogli i ovako iskazati ?((P "~P" Pera)).

ispisuje slovo A.
Sa ?((CREATE PERA)(PUTB PERA 65)(CLOSE PERA)) se stvara nova fajla[23] PERA i u nju ispisuje slovo A.
Sa ?((PUTB "WND:" 12)) se briše ceo tekući prozor.

(R <ime-promenljive>)
Predikat R služi za učitavanje podataka iz osnovnog bafera "BUF:", što se ostvaruje uz pomoć tastature. U R-formuli argument mora biti reč koja počinje podcrtom, tj. biti kao ime promenljive. Tako, pitanjem
?((R _x)(PP _x))
se nakon sa tastature zadavanja neke vrednosti, kao 77, stvara prava promenljiva sa imenom _x i njoj se pridružuje ta vrednost 77.

(READ <putanja-fajle> <term>)
READ predikat je u stvari uopštenje R-predikata i predstavlja određeno preinačenje INTERM predikata. Tako zamislimo da izvesna fajla putanje "\micro\pera.log" ima ovakav početak
((a _x)(b _x _y)) ((f 2)(g 3))
tada na pitanje oblika
?((OPEN "\micro\pera.log")(INTERM _X)(PP _X)(CLOSE "\micro\pera.log"))
će se na ekranu ispsati njen prvi članak, tj. ((a _x)(b _x _y)) bukvalno u takvom obliku. Ali, na pitanje
?((OPEN "\micro\pera.log")(READ _X)(PP _X)(CLOSE "\micro\pera.log"))
na ekranu će se ispisati isti članak, ali _x, _y će postati prave promenljive i umesto njih će se štampati njihove adrese.

(REN <staro-ime> <novo-ime>)
(REN + +)
REN predikat služi za promenu imena fajle. Recimo, sa
?((REN "\jezici\mile.txt" "\jezici\jova.txt")
se fajla mile.txt u direktoriji \jezici preimenuje u fajlu jova.txt.

(RMDIR <dir>)
(RMDIR +)
RMDIR predikat služi za uklanjanje date prazne direktorije.

(RLST <lista> <učitano> <ostatak>) ili (RLST <lista> <učitano>)
(RLST + - -) **(RLST + -)**
Predikat RLST u osnovi služi da iz date liste<lista> pojedinačnih znakova, kao (a b c . j k) odvoji znak po znak i od njih sklopi najdužu moguću reč <učitano>, a eventualno može dati i <ostatak>- preostatak polazne liste. Recimo, na pitanje
?((RLST (a b c . j k) _x _y))
_x će biti reč abc, jer je znak . prekinuo građenje reči, a _y će biti preostatak liste tj. lista (. j k). Primetimo, da se potpuno slično dešava i sa opštim R-predikatom. Tako, ako na pitanje
?((R _x)(PP _x))
za _x dajemo ovaj niz znakova
abc.jk
na ekranu će se pojaviti samo
abc.

(SAVE <ime-fajle>) ili (SAVE <ime-fajle> <lista-imena-članaka>)
(SAVE +) **(SAVE + +)**

[23] U micro-prološkoj direktoriji.

Predikat za snimanje u fajlu datog imena
ili svih članaka tekućeg programa ,
ili onih sa imenima navedenih u <lista-imena-članaka>.
Recimo, sa
?((SAVE PERA)), odnosno ?((SAVE PERA (a b c)))
čitav tekući program snimamo u fajlu PERA.LOG, odnosno u istu fajlu snimamo sve a-, b- i c- članke.

(SDICT! <lista>)
(SDICT! -)
Predikat za dobijanje svih u Micro-prolog ugrađenih predikata, odnosno kaže se i sistemskih predikata.
Relacija pomoću koje dobijamo listu "sistemskih" relacija (članovi liste zadovoljavaju relaciju SYS). Tako, na pitanje ?((SDICT! _X)(PP _X)) na ekranu će se pojaviti ova dugačka lista

(& NUM DEF VAR LST CON FAIL SYS / /* CUMOD CRMOD HIDE SPACE CHAROF STRINGOF ERRM KILL ABORT TIME DATE WLST RLST ALL ADDCL CLS SUM TIMES LESS SIGN INT W WQ WRITE INTERM INTOK LISTP GRNHOL GETB PUTB BWRITE BINTERM BLISTP CRWIND CUWIND PSWIND CURSOR VIDEO FWRITE FREAD "BUF:" "LST:" "AUX:" "TRM:" "WND:" &: FLUSH EOF EXIT LOGIN CHDIR MKDIR RMDIR OPEN CREATE CLOSE SEEK DEL REN DIR FLDATA EXEC VER NOT EQ IF OR CL DELCL EDIT LOAD SAVE LIST ? DICT SDICT MDICT FDICT WDICT GDICT READ I R P PP PQ ! ON FORALL ISALL)

(SEEK <fajla> <blok> <bajt>)
(SEEK + + +)
Predikat pomoću koga se po fajli možemo "kretati" bajt po bajt. Fajla ne sme da bude duža od 32Kb, dogovorno je podeljena na "kriške", blokove a jedan blok ima 1024 bajta, tj. 1Kb. Pretpostavimo da fajla imena "pera" ima ovaj sadržaj
Jovan idd
Onda, smatrajući da je EOF-kraj fajle odmah iz poslednjeg slova, odnosno iza drugog slova d redom ima ove "bajtove"- u stvari, navodimo njihove sadržaje
0:J , 1:o, 2:v, 3:a, 4:n, 5: (Tu je znak beline, ASCII kod mu je 32)
6:i, 7:d, 8:d
gde smo naveli i redosledni broj svakog bajta. Znači ta fajla ima 8 bajtova[24]. Cela staje u jedan blok. U vezi sa tim , recimo, pri računu formule
(SEEK "pera" 0 2)
"ćemo biti upravljeni" na 2-bajt, odnosno v. Kako možemo popraviti grešku u fajli "pera" ? Evo jednog rešenja. Postavimo pitanje
?((OPEN "pera")(SEEK "pera" 0 8)(PUTB "pera" 101)(CLOSE "pera"))
Znači, prvo se otvori fajla"pera", onda se u 0-tom bloku pređe na 8-mi bajt. Iza toga sa (PUTB "pera" 101) taj bajt se promeni i "napuni" sa 101, koji je ASCII kod za e. Konačno se zatvori fajla "pera".

(SIGN <broj> <znak-broja>)
(SIGN + -) **(SIGN + +)**
Predikat u vezi sa znakom datog broja. Primeri:
(SIGN 5.5 1) je tačno

[24] U stvari, ako takvu fajlu napišete pomoću nekog editora onda joj on na kraj doda još 3 bajta: 2 u vezi sa novim redom i 1 bajta za označavanje kraja fajle. U taj bajt prosto stavi 1A, tj. 26 što je ASCI kod za Ctrl-Z.

Formule (SIGN 8 _x), (SIGN -4 _y), (SIGN 0 _z) su tačne pri ovim "vezama" _x=1, _y=-1, _z=0.

(SPACE <var>)
(SPACE -)

Predikat SPACE služi za saznavanje koliko kilobajta memorije je još slobodno za rad. Recimo, ako se na pitanje

```
?((SPACE _X)(PP _X))
```

na ekranu pojavi 62, to znači da je preostalo još 62 bajta.

(STRINGOF <lista-znakova> <reč-znakova>)
(STRINGOF + -),(STRINGOF - +), -(STRINGOF + +)

Predikat koji se odnosi na listu znakova (a1 a2 ... ak) i reč čija slova su redom a1,a2,...,ak. Primeri:

(STRINGOF _X Pera) je tačno za _X=(P e r a)
(STRINGOF (P e r a) _X) je tačno za _X=Pera
(STRINGOF (A "2" "3") _X) je tačno za _X=A23

(SUM A B C) C je A+B
-(SUM + + +), (SUM - + +), (SUM + - +), (SUM + + -)

Primeri:
(SUM 2 3 5) je tačno; (SUM 2 3 _x) je tačno pri _x=5
(SUM 5 _x 8.2) je tačno pri _x=6.

(SYS <ime-relacije>)
-(SYS +)

(SYS A) znači: A je ime neke sistemske, tj. u Prolog ugrađene relacije (predikata). Tako,
(SYS IF) je tačno, dok (SYS a) je netačno, čak i ako smo recimo prethodno uveli neke a-članke.

(TIME <casova> <minuta> <sekundi> <stotinki>) ili
(TIME <casova> <minuta> <sekundi>) ili
(TIME <casova> <minuta>)

Pomoću TIME predikat možemo santai vreme koje računar trenutno pamti.

(TIMES A B C) C je A * B
-(TIMES + + +) (TIMES - + +) (TIMES + - +) (TIMES + + -)

Primeri:
(TIMES 2 3 7) je netačno; (TIMES 2 _x 7) je tačno za _x=3.5
(TIMES 4 5 _x) je tačno za _x=20.

(VAR <zapis>)
-(VAR +)

(VAR A) je tačno upravo ako je A reč koja počinje podcrticom _ , tj. reč izgleda neke proloske promenljive i uz to ,toj promenljivoj još nije dodeljena nikoja vrednost, ili joj je dodeljena vrednost ,ali ona je neka druga promenljiva.
Primer:
(VAR 2), (VAR (3 5 _x)), (VAR (_x 5)) su netačni
(VAR _x4) je tačno.
Dalje, pri raspravi pitanja

```
?((VAR _x)(EQ _x 3)(VAR _x))
```

prva VAR-formula je tačna, dok druga je netačna.

(VER A B C)
(VER - - -)

Primer: Na pitanje


```
?((VER _X _Y _Z)(PP _X _Y _Z))
```

na ekranu će se štampati verzija korišćenog Micro-prologa, kao

micro-PROLOG 1 2

(W <ime-fajle> <lista-koju-ispisujemo>)
(W + +)

Jedan od predikata za upisivanje date liste podataka (drugi argument) u fajlu datog imena (prvi argument). Posle ispisivanja ne obavlja se automatski prelaz u novi red. Slično relacijama P i PP znaci navoda se ignorišu.
Primeri: Na pitanje

```
?((EQ _X 55)(W "WND:" (Evo x= _X)))
```

na ekranu se pojavi

```
Evo x = 55&
```

Na pitanje

```
?((EQ _X 55)(W "WND:" ("Evo x= _X")))
```

na ekranu se pojavi Evo x= _X.
Na pitanje

```
?((W "LST:" (Sada cu ovo napisati pomocu printera))
```

printerom će se oštampati rečenica navedena u listi.

(WDICT|<lista-prozora>)
(WDICT|-)

Predikat pomoću koga možemo dobiti listu trenutno otvorenih (definisanih) prozora. Tako na pitanje

```
?((WDICT|_X)(PP _X))
```

na ekranu će se pojaviti ta lista. Ako se tokom rada neki prozor ukloni (pomoću relacije CLOSE) on se briše iz
liste koju dobijamo sa WDICT.

(WLST <razslov> <lista-reči>)
(WLST - +)

Neka je data <lista-reči> kao (Pera ide tamo). Ako bismo želeli da joj svaki član "usitnimo" do slova, reći ćemo da je "razslovimo", onda bismo od nje dobili ovu listu

(P e r a " " i d e " " t a m o)

gde znaci navoda " " zamenjuju "beline" u polaznoj listi. Za takav posao je napravljen predikat WLST. Naime, ako je <lista-reči> zadana, tada na pitanje oblika

```
?((WLST _x <lista-reči>)
```

promenljiva _x se vezuje za odgovarajući "razslov" te liste reči. Kao i maločas znacima "beline" odgovaraju navodi " ".

(WQ <ime-fajle> <lista-koju-ispisujemo>)
(WD + +)

Predikat potpuno sličan W-predikatu, uz razliku da se znaci navoda ne ignorišu. Primeri:
Na pitanje ?((EQ _X 55)(WQ "WND:" ("Evo x= _X"))) na ekranu se štampa

```
"Evo x= _X"
```

Na pitanje ?((WQ "WND:" ("Pa sta je to?"-pitao se on.))) na ekranu se štampa

```
"Pa sta je to?" -pitao se on .
```

(WRITE <ime-fajle> <lista-koju-ispisujemo>)
(WRITE + +)

Predikat WRITE u osnovi služi da se u već postojeću fajlu upiše neka lis-

ta reči, članaka. Nakon svakog takvog upisa automatski se dodaju i "naznake novog reda". Dalje videti u Zadatku 11.11.

Posebni znaci[25]:

/
Znak reza, videti izlaganje : 2.3 Pravilo o REZU.

(/* <ma-šta>)
Predikat /* je uvek tačan, tj njegova definicija glasi ((/* _x)). Sledstveno formula oblika (/* _x) se može umetati među druge prološke formule bez mogućnosti da može remetiti tok proloških algoritma. Primera radi, prološki su ekvivalenta ova dva a-članka

```
Prvi:    ((a _x)(/* Ovo je a-članak)(LESS 0 _x)(LESS _x 4))
Drugi:   ((a _x)(LESS 0 _x)(LESS _x 4))
```

U skladu sa rečenim predikat /* se, kao i u tom primeru, po pravilu koristi za beleženje raznih napomena, komentara.

(!|<formula>)
(!|+)
U Edinburgškoj sintaksi ! je znak reza, a u Micro-prologu je to poseban predikat "prvog rešenja date formule". Tako, ako je predikat a uveden ovim člancima: ((a 1)) ((a 2)) ((a 3)) onda sa ((b _x)(! a _x)) je uveden predikat b. Predikat b važi samo u "tački" 1, tj. jedino važi (b 1). Inače, opšta definicija !-predikata glasi

```
    ((!|_For) _For /)
```

Recimo, zamenjujući _For sa (a _x) prema njoj imamo

```
    ((!|(a _x)) (a _x) /)
```

što prema definiciji |-terma je ekvivalentno sa

```
     ((! a _x) (a _x) /)
```

Koristeći taj članak potražimo sva _x-rešenja za (! a _x), tj, postavimo pitanje

```
     ?((! a _x)(PP _x)FAIL)
```

Nije teško ideti da zbog /-FAIL kombinacije _x će dobiti samo jednu vrednost: prvo _x-rešenje formule (a _x).

"?ERROR?"
je jedno posebno ime i korisnik po želji može tekućem programu dodati nekoliko "?ERROR?" -članaka. Te članke Prolog tretira na poseban način. Naime, kad god se tokom proloških algoritma pojavi neka greška, umesto da prekine čitav algoritam Prolog izvršava "?ERROR?" -članke.
Primer:
Kao što smo u opisu ERRM-predikata videli, broj 2 odgovara slučaju pojave nepoznatog predikata. Pretpostavimo da u takvom slučaju hoćemo da se na ekranu prvo pojavi poruka

```
        Nepoznat predikat
```

dalje da se algoritam prekine, i uz to na ekranu pojavi znak ? ("nedokazanosti"). U tu svrhu je dosta da uvedemo ovaj "?ERROR?" -članak

```
   (("?ERROR?" 2 _x)(PP Nepoznat predikat)(PP ?))
```

Međutim, ukoliko želimo da se slučaj pojave nepoznatog predikata tretira samo kao slučaj nedokazane formule[26], onda to možemo postići uvođenjem

---

[25] tj, oni koji ne počinju nekim običnim slovom, pa nije jasno kojim redom da se redaju.

[26] Takav pristup koristi ARITY-prolog.

ovog "?ERROR?" -članka

```
   (("?ERROR?" 2 _x)FAIL)
```

"BUF:"
je ime specijalne fajle koja predstavlja jedan od dva bafera[27] tastature. Drugi je "TRM:". Svaki na tastaturi kucan znak ide u tu fajlu, i to sa ehom, tj. kucani znak se pojavi i na ekranu. Ta fajla može maksimalno da primi 1920 karaktera.

"LST:"
je bafer[28] printera, tj. ime fajle čiji sadržaj se štampa na printeru. informacija na štampač.

"TRM:"
je ime specijalne fajle koja je pored "BUF:" takođe bafer tastature. Međutim, za razliku od "BUF:" kucani znak na tastaturi je bez ehoa, tj. ne pojavljuje se na ekranu. Koristi se za direknu kontrolu toka programa pomoću tastature (kod rada sa menijima, prozorima).

"<USER>"
je ime posebno ime za članke koje korisnik ("user") može sam uvesti. Ti "<USER>" članci se ponašaju na veoma poseban način. Recimo, zamislimo da smo smo napravili fajlu imena "duzina.log" ovog sadržaja

```
        ((duz () 0))
        ((duz (_x|_y) _Rez)(duz _y _Rez1)(SUM 1 _Rez1 _Rez))
    (("<USER>")(PP Ovo je program za duzinu liste))
```

Tada ako smo ušli u Micro-prolog i učitali tu fajlu[29] onda:
Prvo se izvršava "<USER>" članak, tj. na ekranu se pojavljuje poruka

```
        Ovo je program za dužinu liste
```

Međutim, ako posle bilo šta radimo u tom programu, kao postavimo pitanja

```
        ?((PP Pera))    ?((duz (1 2 3) _x)(PP _x))
        ?((PP Jova)ABORT)
```

na kraju kao "dodatak" izvršiće se gornji "<USER>" članak, tj. na ekranu će se svaki put pojaviti navedena poruka. Slično, ako usput dodamo nov članak kucajući recimo

```
        ((p _x _y)(duz _y _x))
```

opet će se štampati navedena poruka.
U skladu sa izloženim, obično se kaže da korisnikovi "<USER>" članci " u svoje ruke" preuzimaju kontrolu sveg delanja u okviru programa u kome se pojavljuju.

"WND:" je oznaka tekućeg prozora[30]. Razni predikati ispisivanja kao P, PP, LIST dejstvuju u tekućem prozoru "WND:"

&
ime koje je u Micro-prologu dato osnovnom modulu, u kome se nalazimo čim uđemo u Micro-prolog, učitamo neku fajlu, upišemo neke članke, postavimo

---

[27] U vezi sa pojmom "bafer" videti o FLUSH-predikatu.

[28] Videti prethodnu fusnotu.

[29] Recimo, sa LOAD DUZINA ili LOAD "duzina.log".

[30] O prozorima videti u opisu CRWIND- predikata.

neko pitanje i dr. sve dok ne predemo u neki drugi modul[1].

&:
ime koje je u Micro-prologu dodeljeno osnovnom prozoru (prozor koji je ceo ekran). U tom prozoru se nalazimo čim uđemo u Micro-prolog, i ostajemo u njemu sve dok ne pređemo u neki drugi prozor[2].

?
To je osnovni Micro-prološki predikat. Njegova definicija glasi
((? _X)|_X)
pa je primenljiv ne na formule, već na njihove "sastave" kao
(for1 for2 ...fork) ,gde su for1,for2,...,fork formule
odnosno opštije na "repove članaka" (videti u poglavlju 4 deo 4.1).

12.2 Arity- prolog

U ovom opisu izlažemo o predikatima Arity-prologa[3]. Najpre sledi spisak[4] posebnih znakova i funkcija[5], a onda spisak predikata poređanih abecedno. Pomenimo da među tim predikatima ima i pravih funkcija, čija upotreba pretpostavlja korišćenje is-predikata. Primer: X is 2+3. Tu je + oznaka sabiranja, koje je operacija, odnosno funkcija. Pri navođenju predikata recimo imena ime obično pišemo ili ovako: ime / k, gde k, tzv. arnost, tj. broj argumenata na koji se odnosi predikat ili pišemo samo ime, a onda običnim rečima navodimo arnost.

P o s e b n i z n a c i i f u n k c i j e:

!
Znak reza, videti izlaganje: 2.3 Pravilo o REZU. Videti i o call-predikatu.

/* */
Između ta dva znaka može se staviti ma koji tekst, komentar, koji se od strane Prologa ignoriše.

, Arnost je 2
Znak konjunkcije (videti (4.2.1))

; Arnost je 2
Znak disjunkcije (videti (4.2.2)).

*
Oznaka množenja

---

[1] Videti o CRMOD i o CUMOD predikatu.

[2] Videti o CRWIND i o CUWIND predikatu.

[3] Napominjemo da mnogi od tih opisa odgovaraju i LPA-prologu. Tu činjenicu ćemo iskoristiti u tački 12.3 pri opisu predikata LPA-prologa.

[4] U spisku se nalazi veći deo predikata Arity-prologa (vezrija 6.0).

[5] Tu dolaze +,-,sin,log i dr.

+
Oznaka sabiranja

-
Oznaka oduzimanja

/
Oznaka delenja

//
Oznaka celobrojnog delenja

mod
Oblik korišćenja: X mod Y . Smisao: X mod Y je ostatak pri delenju X sa Y. Primer: U pitanju ?- X is 23 mod 7. X se vezuje za 2.

^ pri upotrebi oblika X^Y, gde X,Y realni brojevi označava stepen "iks na ipsilon". Primer: U pitanju X is 3^2. X se vezuje za 9.

sin, cos, tan, log, ln ,abs, asin,acos, atan, exp, sdrt
su oznake dobro poznatih funnkcija. Primeri: U pitanjima
?- X is sin(1). ?-X is abs(-2). ?-X is sqrt(2)
X se vezuje za 0.84147098 ,odnosno 2, odnosno 1.41421356.

X/\Y 2-konjukcija X i Y
Logička konjunkcija 2-zapisa brojeva X,Y. Primer (račun korak za korakom):
9/\27= 1 0 0 1 Prvo smo 9 i 27 iskazali u osnovi 2
/\ 1 1 0 1 1
= 0 1 0 0 1 2-Cifre smo redom "spajali" po pravilima konjunkcije: 1/\1=1, 1/\0=0 i sl.
= 9 Rezultat iskazujemo u 10-zapisu (desetično).

X\/Y 2-diskjunkcija X i Y
Ta je funkcija slična prethodnoj s tim što se 2-cifre pri računu spajaju po pravilima disjunkcije: 0\/1= 1\/0= 1\/1=1, 0\/0=0.

X<<Y 2-zapis od X se ulevo pomeri za Y mesta.
Videti i dve funkcije ispred. Primer
5<<1= 1 0 1 pomaknuto ulevo za 1 (5 u 2-zapisu glasi 1 0 1)
=1 0 1 0 Pri pomaku smo na kraj dopisali 0
=10.
X>>Y 2-zapis od X se udesno pomeri za Y mesta.
Slično prethodnoj funkciji, ali pomak je udesno.

= Arnost je 2
Predikat = u potpunosti odgovara unifikaciji. Videti više u opisu EQ-predikata u delu 12.1 ,tj. opisu Micro-proloških predikata.

\= Negacija od =

== Bukvalno jednaki

Primeri: Tačne su formule 2==2, [1,3]==[1,3], X==X, ali nije tačna
X==Y
jer promenljive X,Y su različite, tj. sa različitim adresama (u memoriji). Međutim, ova konjunkcija

X=Y,X==Y

je tačna, jer zbog = najpre se X,Y unificiraju, što se svodi na :

Y promeni[6] svoju adresu i kao novu uzme adresu od X

\== Negacija od ==

:- / 2
Znak "grlo" za građenje nejednočlanih članaka, koji se mogu gledati kao ovakve strukture ':-' (Glava,Rep). Videti (4.2.5).

izraz1 < izraz2 < je znak "manji"

Smisao: vrednost izraza izraz1 je manja od vrednosti izraza izraz2. Tačne su formule 2<3, 6+5<7*4-1.

izraz1 > izraz2 > je znak "veći"
Smisao: vrednost izraza izraz1 je veća od vrednosti izraza izraz2.

izraz1 =:= izraz2 =:= je znak "jednakovrednosni"
Smisao: vrednosti izraza izraz1,izraz2 su jednake.Recimo, tačna je formula 2+3*4=:=7*2.

=\= Negacija za =:=

=< manji ili jednak, tj. negacija za >

>= veći ili jednak, tj. negacija za <

=.. Strukt=.. Lista
O tom predikatu videti u tački 1,deo 2).

@< leksički manji
Primer: ana @< beba tačno, dok pera @< mika netačno

@> leksički veći
Primer: pera @> mila je tačno, dok a23 @> a32 je netačno.

@=< leksički manji ili jednak,tj. negacija za @>.

@>= leksički veći ili jednak, tj. negacija za @<.

[! !] "snip" -predikat
Videti Zadatak 6.8.

\+ drugi zapis za not.

[fajla1,fajla2,...] Ako smo već ušli u Arity-prolog onda takvim pitanjem se učitava jedna za drugom niz fajli fajla1, fajla2,...
Primer: Na pitanje
?- [pera,mile,jova].[7]
učitavaju se prvo fajla pera, pa iza nje mile i najzad fajla jova.

^ Znak kvantora[8] postoji. Videti (4.2.12).

A b e c e d n i s p i s a k p r e d i k a t a

a b o l i sh(ime_predikata/arnost)

[6] Kod Micro- i LPA-prologa na takvom mestu se dešava obratno, tj. X menja adresu uzevši adresu od Y.

[7] Tj. pera.ari ,ako je Arity-prolog ili pera.dec ako je LPA-prolog.

[8] Napred smo videli da se ^ koristi i kao znak stepena.

abolish(+)
Predikat abolish služi za brisanje svih članaka datog imena: ime_predikata i uz to date arnosti (videti (4.2.10)). LPA-prolog ima nekih proširenja:
Tako sa ?-abolish(ime). brišu se svi članci imena 'ime' -bez obzira na arnost. Dalje, sa ?-abolish ([ime1,ime2,...,imek]) se brišu svi članci imena: ime1,ime2,...,imek (bez obzira da li jesu ili nisu dinamički).

a b o r t(<broj>)
abort(+)
Videti ABORT u 10.1. Primer (u Arity-prologu):
ajde:-write('Daj '),read(X),
ifthenelse(number(X),write('Broj'),(abort(4),write('Kraj'))).
Ako na pitanje ?-ajde, za X damo ne-broj, kao: pera.,onda zbog abort(4) će se računanje formule ajde završiti na mestu abort(4), i na ekranu se neće pojaviti reč Kraj.
U LPA-prologu abort predikat ima arnost 0, tj. abort-formula ima oblik abort.

a p i _ m o n o
Uslugom tog predikata ,ako smo ušli u Arity-prolog i još koristimo monitor u boji, onda pitanjem ?-api_mono. prelazimo na crno-beli ekran.

a r g(<broj>,<struktura>,<član>)
arg(+,+,-), -arg(+,+,+)
Smisao: Ako je dat <broj>, kao 1,2,..., i data struktura, odnosno izraz oblika ime(a1,a2,...,ak) onda <član> je upravo ai, gde i=<broj>.
Recimo, pri računu formule arg(2,f(p,q,r,s),X) promenljiva X se vezuje za q.Videti (4.2.6).

a r g 0(<broj>,<struktura>,<član>)
Slično kao arg, ali <član> treba da bude 1+<broj> -ti član strukture.
Primer: ?-arg0(1,g(2,3),X). X se veže za 3.

a r g r e p(<struktura>,<broj>,<zamena>,<rezultat>)
argrep(+,+,+,-), -argrep(+,+,+,+)
Smisao: U datoj strukturi , član po redu: <broj> zameni datom zamenom.
Primer: ?-argrep(f(2,3,4),2,pera,X). X se vezuje za f(2,pera,4).

a ss e r t(<članak>), a ss e r t a(<članak>), a ss e r t z(<članak>)
Predikati za dodavanje članaka. Videti (4.2.7) i (4.2.8).

a t o m(<dato>)
atom(+)
Smisao: <dato> je konstantska reč, kao: a,b,pera, 'Mile',ali ne i 23,f(3).
U LPA-prologu predikat atom može da bude i u ovom obliku
a t o m(<dato>,<tip>)
atom(+,-)
Smisao: <dato> je atom (vid. prethodno) čiji tip je <tip>.Evo mogućih vrednosti za <tip> i odgovarajući smisao
0 Atom nema nikakvo posebno značenje, tj.ne nastupa nijedan od donjih slučajeva.
1 Atom je ime tekućeg definisanog modula
2 Atom je ime neke već otvorene fajle.
3 Atom je ime nekog već napravljenog prozora.

[9] Znači kad na red dođe računanje atom-formule atom(<ime>,Prom), onda je prethodno bila otvorena fajla imena <ime>.

4 Atom je ime neke binarne operacije, kao '+','*' i sl. ugradene u Prolog.
5 Atom je ime neke unarne operacije,kao 'sin','cos','log' i sl.ugradene u Prolog.
6 Atom je ime nekog nekog u Prolog ugradenog predikata,koji je pisan ili na Asembleru ili na C- jeziku.
7 Atom je ime nekog dinamičkog predikata tekućeg programa
8 Atom je ime nekog statičkog predikata tekućeg programa

U atom-formuli <tip> mora biti promenljiva (koja nema vrednost).
Primeri:
1) Pretpostavimo da odmah pri ulasku u Prolog postavimo pitanje
?-atom('pera',X).
Tada će X biti vezano za 0, jer 'pera' je "običam" atom. Ali, ako iza toga pitamo
?-create('pera').
tj. napravimo fajlu imena 'pera', i zatim pitamo
?-atom('pera',X).
X će biti vezano za 2.
2) ?-atom(&:,X). X se vezuje za 3
3) ?-atom(write,X). X se vezuje za 8.

a t o m i c(<reč>
Smisao: <reč> je konstantska ili je oznaka broja.

a t o m _ s t r i n g(<atom>,<string>)
atom_string(+,-), atom_string(-,+), -atom_string(+,+)
Taj predikat je "zadužen" za
Prebacivanje atoma u string,
Prebacivanje stringa u atom,
Proveru da li su atom i string medu-odgovarajući.
Primeri:
?-atom_string(pera,X); X se veže za $pera$
?-atom_string($pera$,X); X se veže za atom pera
?-atom_string(pera,$pera$) je tačno.

b a g o f(<uzorak>,<formula>,<torba>)
bagof(+,+,-), -bagof(+,+,+)
Smisao: <torba> je lista svih objekata oblika <uzorak> tako da važi formula <formula>. Videti 4.2.12. Primer:
Uz pretpostavku da je a-relacija definisana člancima
a(1). a(2). a(1).
na navedena pitanja imamo date odgovore
?- bagof(X,a(X),Torba); Torba se vezuje za listu [1,2,1]. Pazite, 1 se se pojavljuje dvaput.Tako ne bi bilo da je umesto bagof predikata korišćen setof predikat.
?-bagof(f(X),a(X),Torba); Torba se vezuje za listu [f(1),f(2),f(1)].

b e t w e e n b, b e t w e e n k e y s b
Videti deo 8.3 u tački 8.

b r e a k
Prekida tok programa, koji se onda može nastaviti pritiskom Ctrl-Z tipke.
Primer:

```
ajde:-write('Pera'),nl,write('Za nastavak pritisnuti Ctrl-Z'),
      break,write('Kraj').
```

Na pitanje ?-ajde. će se štampati


```
Pera
Za nastavak pritisnuti Ctrl-Z
```

i tok algoritma će zastati,a nastaviće se posle datog Ctrl-Z znaka.Iza tog će se na ekranu još pojaviti reč

```
Kraj
```

b t r e e _ c o u n t(ime_drveta,broj_ključeva)
Videti deo 8.3 u tački 8.

c a l l(<formula>)
Smisao: (prološki) izračunati formulu <formula>
Primeri: Navedeni Članak1 je prološki ekvivalentan sa Članak2

Članak1	a(X):-b(X), call(c(X)).	Članak2	a(X):-b(X),c(X).
Članak1	a(X):-call((b(X),c(X))	Članak2	a(X):-b(X),c(X).

pa se može pomisliti da je call-predikat nepotreban, odnosno uvek izbaciv. Međutim, da to nije tako pored ostalog može se videti iz donjeg programa u kome se javlja call(!):

```
proba1:-write('Pera'),call(!).
proba1:-write('Dara').
proba2:-write('Pera'),!.
proba2:-write('Dara').
```

Na pitanje ?-proba2,fail. na ekranu će se štampati samo reč Pera, dok na pitanje ?-proba1,fail. će se štampati[10] i Dara i Pera. Pravi razlog je što je call-predikat u stvari definisan sa

```
call(X):-X.
```

pa su algoritamska drveta prvog i drugog pitanja strukturno različita.

c a s e
predikat za logičko razlikovanje slučajeva. Javlja se u dva oblika:
case([A1->B1,...,An->Bn|Ostatak])
Smisao: Ako A1 izračunaj(uradi) B1, inače ...,Ako An izračunaj Bn, a inače,tj.kad nijedan od A1,...,An nije tačan, izračunaj Ostatak.
case([A1->B1,...,An->Bn]).
Smisao: Ako A1 izračunaj B1,inače ..,Ako An izračunaj Bn,a ako nije nijedan od A1,...,An, onda case-formula je,po definiciji,tačna.
Videti Zadatak 6.9.

ch d i r(<putanja_direktorije>)
Smisao: ili da se pređe u novu direktoriju date putanje, ili kad se ne navede putanja, tj. ona je promenljiva da se sazna u kojoj smo direktoriji. Recimo, sa ?-chdir(X), X će biti vezano sa putanjom tekuće direktorije,kao c:\jezici\arity6, dok pitanjem ?-chdir('c:\turbo') se prelazi u naznačenu direktoriju.

ch m od(<ime_fajle>,<fajlin_atribut>)
chmod(+,+), chmod(+,-)
Predikat chmod je u vezi sa vrednošću (read-only, hidden, archive, system) atributa date fajle; ili nam daje tu vrednost ili ako je sami izaberemo, onda je pridružuje fajli.

c l a u s e(<glava_članka>,<rep_članka>)
Videti (4.2.4). Istaknimo da clause predikat nije "jednogran", odnosno pri proceduri vraćanje(backtracking) u upotrebu mogu ući i druge clause-grane.

---

[10] To je blisko Micro-prološkom predikatu '?'.

c l o s e(<ručka>)
Zatvara fajlu zadane ručke ("handle"). Videti Zadatak 11.12 (deo za Arity-prolog).

c l s
Predikat "zadužen" da obriše tekući prozor (recimo,ceo ekran) i još da kursor postavi u položaj 0,0, tj. krajnji levi gornji ugao.
Primer: Pitanjem ?-write('Pera'),cls,write('Dara'). prvo će se na "zatečenom" mestu štampati reč Pera, iza tog ceo prozor će biti obrisan, i sa početkom u tački 0,0 će biti štampana reč Dara.

c o d e _ w o r l d(<stari_svet>,<novi_svet>)
code_world(+,+), code_world(-,-)
Predikat code_world je u vezi sa svetovima(videti deo 8.2 tačke 8).Primer:
Sa ?-create_world(pera) se pravi svet imena pera
Dalje sa
code_world(main,pera) se iz polaznog sveta (zove se 'main') prelazi u taj svet pera. Sada na pitanje ?-code_world(X,X) za X bismo dobili pera, tj. ime sveta u kome smo trenutno.
Ako smo već u svetu imena pera, onda recimo sa ?-assert(a(3)) članak a(3) se unosi kao članak upravo u taj code_world. Tako, ako nalazivši se u tom svetu pitamo ?-listing. onda će se i članak a(3) pojaviti.

c o m p a r e(<rez>,<reč1>,<reč2>)
Smisao: Leksikografski poredi <reč1> i <reč2>. Rezultat <rez> je jedan od ovih = < >. Primer:
Pitanjem ?-compare(X,pera,jov) X se veže za >.
Formula compare(=,per,per) je tačna.

c o n c a t
predikat za dopisivanje stringa na string(pa i više njih). Koristi se u dva oblika
c o n c a t(<string1>,<string2>,<rezultat>)
c o n c a t([string1,string2,...],<rezultat>)
Primeri:
?-concat(pe,ra,X); X se vezuje za string $pera$. Tako će Vam Arity odgovoriti; string je izmedu znakova $ $
?-concat('pera',$dara$,X); X će biti $peradara$
?-concat('pera',65,X); X će biti $peraA$. Jednočlani string, kao 'A' može se zadati i svojim ASCII-kodom.
?-concat([pera,dara,mara],X); X će biti $peradaramara$.

c o n s u l t(<ime_fajle>)
Predikat consult služi da tekućim člancima doda i sve one iz fajle datog imena <ime_fajle>).

c r e a t e(<ručka>,<ime_fajle>)
Predikat create služi za pravljenje nove fajle datog imena <ime_fajle>, u u koju se može vršiti upis.Kao rezultat se daje vrednost <ručka>("handle") koja se uglavnom dalje koristi pri radu sa napravljenom fajlom. Inače, ako se create-predikat koristi u slučaju već postojeće fajle ona se gubi,odnosno umesto nje se pravi nova pod istim imenom. Videti i Zadatak 11.12.

c r e a t e _ w o r l d(<ime_sveta>)
Predikat create_world služi za pravllenje sveta datog imena.Videti deo 8.2 u tački 8.

c t r _ d e c, c t r _ i n c, c t r _ i s, c t r _ s e t
To su "brojački predikati". Videti Zadatak 11.19.

c u r r e n t _ o p(<predn>,<tip>,<ime>)
Predikat current_op je u vezi sa op-izrazima;videti deo 4.3 u tački 4. Recimo, pitanjem
?-curent_op(X,Y,Z),write(X),tab(2),write(Y),tab(2),write(Z),fail.
će se dati spisak svih op-izraza[11] koji su u bazi podatataka tj. u fajli API.IDB. To znači, tu će biti i svi u Prolog ugrađeni, kao i oni koje je korisnik dodato.

c u r r e n t _ p r e d i c a t e(<predikat>)
Predikat current_predicate "pamti" sve predikate u osnovnom programskom svetu (taj code_world ima naziv 'main'), ili uopšte u nekom drugom takvom svetu. Primer:
Ako udemo u Arity-prolog i sa ?-[vidi]. učitamo fajlu imena vidi. ari, tada na pitanje
?-current_predicate(X),write(X),nl,fail.
će se javiti spisak svih u toj fajli definisanih predikata. Recimo, ako se u njoj javlja članak oblika
a(X,Y):-...
gde tačkice označavaju nenaveden deo onda će u rečenom spisku biti i:
a / 2
Tu je prvo navedeno ime, zatim službeni znak /, i zatim arnost predikata a. U stvari, spisak se sastoji od delova oblika
ime_predikata / arnost
Međutim, ako umesto gornjeg postavimo pitanje
?-current_predicate(a / X),write(X),fail.
dobićemo spisak svih a-predikata sa njihovom arnošću, dok na pitanje
?-current_predikate(X / 2),write(X),fail.
dobiće se spisak svih predikata arnosti 2 (naravno broj 2 nije bitan).
Predikat current_predicate se može koristiti uopšte u ma kom programskom svetu[12]

c u r r e n t _ w i n d o w(<prozor1>,<prozor2>)
current_window(-,-); current_window(+,+)
Predikat current_window je jedan od prozorskih predikata (videti i define_window predikat).
Pitanjem
?-current_window (X,X). X se veže za ime tekućeg prozora, tj. prozora u kome smo.To u krajnjem slučaju može biti prozor 'main'; osnovni u kome smo kad se "uključimo" u Arity-prolog
Pitanjem
?-current_window(main,proz).
uz pretpostavku: da smo trenutno u osnovnom prozoru main, da smo već sa define_window predikatom napravili prozor imena proz,se iz prozora main prebacujemo u prozor proz.Slično pitanjem:
?-current_window(proz1,proz2)
se iz prozora proz1 prelazi u prozor proz2; pretpostavljajući da su oba prozora već napravljena.

---

[11] tj. njihovih brojeva "prednosti", njihovih tipova i imena.

[12] Osnovni svet je 'main', a sami možemo napraviti neki drugi.Videti deo 8.2 u tački 8.

d a t a _ w o r l d(<svet1>,<svet2>)
Predikat data_world je jedan od predikata o svetovima(videti deo 8.2 u tački 8).
Pitanjem
?-data_world (X,X). X se veže za ime tekućeg sveta, tj. onoga u kome smo.To je u krajnjem slučaju svet 'main' u koji "ulazimo" čim se uključimo u Arity-prolog.
Pitanjem
?-data_world(svet1,svet2)
se iz sveta svet1 prelazi u svet svet2 - uz pretpostavku da su to već napravljeni svetovi[13]

d a t e, d a t e _ d a y
Predikati u vezi sa tekućim datumom, i danom u nedelji. Primer:
Pitanjem
?-date(X). za X se dobije nešto kao date(1992,8,31) sa smislom 31-i avgust 1992 godine
Pitanjem
?-date_day(date(1992,8,31),X). za X se dobije 1, sa smislom Ponedeljak. Naime, 0,1,...,6 redom ogovaraju danima Nedelja,Ponedeljak,...,Subota.

d e b u g
Pored **trace**, primer tzv. "debuger"-predikata, odnosno predikata koji omogućuje praćenje toka programa, korak po korak. Za razliku od trace pomoću debug predikata tokom izvršenja ne prate se svi predikati već samo oni koji se za to naznače uslugom predikata **spy**.Primera radi zamislimo da tekući program sadrži predikate a,b,c,d, i da tokom raspravljanja pitanja
?-a(X).
hocemo da vidimo ponašanje predikata a i c. To možemo postići ovako
Prvo pitanje: ?-debug.
i tako smo "uključili" debug-predikat.
Drugo pitanje: ?-spy([a,c]).
i tako naznačujemo želju da "špijuniramo" a i c.
Sada su obavljene pripreme za glavno pitanje:
?-a(X).
Na ekranu će se korak-za-korakom objasniti njegovo raspravljenje sa isticanjem dešavanja u vezi sa predikatima a i c.
Nakon završenog raspravljanja možemo recimo sa
?-nospy(a). ili ?-nospy(c). ili nospy([a,c]).
ukiniti "špijunski" status za a, odnosno c ,odnosno i za a i za c.
Takođe, ako u nastavku rada nećemo uslugu debug-predikata onda pitanjem
?-nodebug.
ga "ukidamo".
Inače uopšte tokom "skroziranja" toka programa po pravilu se ističu ovakvi detalji:

call	<for>	-kad se dođe do računanja formule <form>
exit	<for>	-kad se sa uspehom završi računanje formule <for>, tj. dokaže <for>.
redo	<for>	-kad se pri vraćanju (backtracking) ponovo dođe do formule <for>.
fail	<for>	-ako se dogodi da se računanje formule <for> završi sa ne.

U Arity-prologu se uslugom tzv. leash predikata može postići da se pri "skroziranju" istaknu samo neki od tih detalja.Evo kako se to može učiniti. Uočimo ovu malu tabelu sa četiri polja

Call Exit Redo Fail

gde svako polje može biti popunjeno sa 0 ili 1. Recimo jedno popunjenje glasi

1 1 0 0

Reći ćemo slobodnije da su uključena polja Call,Exit a isključena Redo, Fail (jer na njima su 0). Gledajući te brojke kao sastavke binarnog zapisa lako vidimo da im odgovara broj 12 (u desetičnom zapisu).
Zamislimo sada da se postavi ovo pitanje
?-leash(12).
To znači da će se iza toga pri dejstvu debug-predikata na ekranu isticati samo Call i Exit detalji jer,kako rekosmo njihova "polja" na uočenom popunjenju su uključena. Ali, to ujedno znači da recimo nećemo "videti" kad se desi da neka formula trenutno ne bude dokazana.
Recimo, ako bismo želeli da vidimo samo ona mesta na koja se proloski algoritam vraća onda to postižemo pitanjem
?-leash(2).
jer broju 2 odgovara ovaj 2-zapis sa četiri cifre
0 0 1 0
i tada je uključeno samo Redo polje.

d e c(<broj>,<broj - 1>)
Predikat dec služi da se od datog broja <broj> pređe na broj za 1 manji.
Primer: ?-dec(5,X). X se veže za 4.

d e b u gg e r
Pitanjem ?-debugger.na ekranu se pojave pojedinosti u vezi sa debug- odnosno trace- predikatom, kao koji s predikati naznačeni kao spy i dr. Videti o debug-predikatu.

d e f i n e b (ime, broj_raklji_u_čvoru, jedinstvenost_ključeva, redosled)
Jedan od predikata za B-drveta, o kojima se govori u delu 9.2 tačke 9.

d e f i n e _ w i n d o w
(naziv,zaglavlje, (leva_gornja_vrsta,levi_gornji_stubac),
(desna_donja_vrsta,desni_donji_stubac),(proz_atr,ivica_atr))
Predikat define_window služi za pravljenje prozora. Primer:

Pitanjem ?-define_window(pera,'ovo je pera-prozor',(2,3),(20,30),(10,15))
se pravi prozor naziva pera, sa okvirom koji sadrži tekst
ovo je pera-prozor
čiji levi gornji ugao je u tački (2,3),desni_donji u tački (20,30), dok 10 i 15 odgovaraju bojama kursora (zelena) i boji okvira (bela)[14].

d e l e t e(<ime_fajle>)
Predikat delete služi za brisanje fajle datog imena. Primer:
Sa ?-delete('fakt.ari'). se briše fajla fakt.ari koja je u direktoriji gde i Arity-jezik.
Sa ?-delete('c:\jezik\tc2\vidi.c'). se briše fajla vidi.c, kojoj je nave-

[13] Jedino za svet main se to ne traži- on je po postavci napravljen.

[14] Arity-prolog pored opisanog ima predikat define_window sa još dva dodatna argumenta,koji su "zaduženi" za pozadinu boje prozora i za osnovne delice iz kojih je sam okvir napravljen.

dena cela putanja.

d e l e t e _ w i n d o w(<ime_prozora>)
Služi za brisanje(uklanjanje) prozora datog imena.

d e l e t e _ w o r l d(<ime_sveta>)
Služi za brisanje sveta datog imena.

d i r e c t o r y(<putanja_fajle>,<ime>,<mode>,<vreme>,<datum>,<veličina>)
Ako se zada putanja fajle onda ostali argumenti redom daju
ime fajle
mode fajle (1 ako je read-only; 2 ako je hidden;
4 ako je system fajla; 16 ako je directorija;
32 ako je archive.)
vreme (u toku dana) i datum pravljenja
veličina (u bajtovima).

d i s k(<ime_diska>)
Primer: Sa ?-disk(X). X se veže recimo sa C ako smo na C-disku.
Sa ?-disk(a). prelazimo na gipki disk a.

d i s p l a y(<izraz>)
Dati izraz preobraća na prefiksni poljski oblik. Primer:
?-display(2+3). Na ekranu se pojavi '+'(2,3)
?-display(2+3*4). Na ekranu se pojavi '+'(2,'*'(3,4))

d i s p l a y(<ručka>,<izraz>)
Deluje slično kao display-predikat,ali zapis umesto ne ekran obavlja u fajlu zadane "ručke" ("handle"). Videti Zadatak 12.12 (deo za Arity-prolog).

e q(<predmet1>,<predmet2>)
Predikat eq ("adresno jednaki") je sličan predikatu ==. Tako, formule
eq(3,3), 3==3
eq(X,X), X==X
su tačne. Ali, dok je recimo formula [1,2]==[1,2] tačna, dotle netačna je formula eq([1,2],[1,2]). Naime, eq je "lenj", i neće da proverava bukvalnu jednakost složenih predmeta.

e r a s e(<ref_broj>)
Služi za brisanje objekta datog referentnog broja (vid.deo 8.2 u tački 8).

e r a s e a l l(<ime_ključa>
Služi za brisanje svih objekata zapisanih pod datim ključem (vid.deo 8.2 u tački 8).

e x p a n d _ t e r m(<gramatički_izraz>,<proloski_prevod>)
Služi da se za dati gramatički izraz napravi odgovarajući proloski prevod, tj. članak .Primer:
U pitanju ?-expand_term((a-->b,c),X). se javlja gramatički izraz[15]
a-->b,c (kao dat). Tada se X vezuje za prevod kao
a(P,Q):-b(Q,R),c(R,Q) (Doduše umesto promenljivih P,Q,R pojavljuju se odgovarajuće adrese)

e x p u n g e / 0

Ako se u radu sa svetovima neki podatak uklanja, onda uz upotrebu tog

[15] O gramatičkim izrazima videti u delu 4.4 tačke 4.

predikata se oslobađa njegova memorija(što znači da je Prolog može kasnije koristiti). Drugim rečima expunge je neke vrste "čistač".

f a i l
Opšta prološka "formula" koja po definiciji je uvek netačna.

f i l e _ l i s t(<ime_fajle>)
Taj predikat je sličan predikatu SAVE kod Micro- i kod LPA-prologa. Naime, program sa kojim radimo u Arity možemo snimiti kao jednu fajlu. Recimo, sa ?-file_list(pera) ta fajla se napravi pod imenom pera.ari.

f i l e _ l i s t(<ime_fajle>,<ime_članka/arnost>)
Slično kao prethodni predikat, ali snimanje se odnosi samo na članke datog imena i date arnosti.

f i l e e r r o r s(<staro_stanje>,<novo_stanje>)
To je predikat "zadužen" da vodi računa o usput napravljenim greškama u vezi sa fajlama. Tako, ako pri ulazu u Arity-prolog postavimo pitanje

?-fileerrors(X,X).

videćemo da je X vezano za 'on', tj. da je taj predikat "budan" za greške. Shodno tome, ako odmah iza tog pitanja pitamo

?-open(_,'c:\jova',r)

gde fajla 'c:\jova' ne postoji, na ekranu će se pojaviti takva opomena o nepostojanju. Slično bi nam se uopšte desilo tokom izvršenja nekog programa u kome se naiđe na otvaranje nepostojeće fajle. Ali, na tom mestu Prolog ujedno prekida čitav algoritam. Međutim, to može biti nepraktično. Da li bi Prolog mogao biti nateran da u slučaju takvih formula "ne iskače", već jedino "smatra" da je takva formula netačna ? Upravo toj svrsi služi

fileerrors-predikat.

Recimo, ako umesto gornjeg pitanja postavimo ovo

?-fileerrors(_,off),open(_,'c:\jova',r).

onda će odgovor biti no (tj. ne), ali bez prekida programa i pojave poruke na ekranu.

f i n d a l l(<uzorak>,<formula>,<lista>)
Videti (4.2.11)

f l o a t(<realan_broj>)
Primer: formula float(3.4) je tačna, a formula float(6) netačna.

f l o a t _ t e x t(<realan_broj>,<string>,<format>)
float_text(+,-,+), float_text(-,+,-), -float_text(+,+,+)
Predikat float_text uglavnom služi za "prelaz" od realnog broja na odgovarajući string, kao i obratno. Treći argument <format> može imati jednu od ovih vrednosti
general, fixed(N), scientific(N) (N je broj decimalskih cifara)
Primeri:
?-float_text(3.45,X,general). X se vezuje za string 3.45
?-float_text(3.45,X,fixed(2)). X se vezuje za string 3.45.
?-float_text(3,45,X,fixed(1)). X se vezuje za string 3.5
?-folat_text(X,5.67,_). X se vezuje za 5.67. Format se ignoriše.

f l u s h / 0
Predikat koji služi da se "briše" tekući bafer (koji prihvata i čuva znake

kucane sa tastature).

f u n c t o r / 3
Videti (4.2.3)

g c
Predicat gc ("garbage colector"-skupljač smeća) je zadužen da brine o korišćenju memorije za stog("stack"), tako da, recimo, uzima na raspolaganje memoriju podacima koji se više ne moraju čuvati na stogu.

g e t(<prom>), g e t(<ručka>,<prom>)
To su predikati predikati za "input",za "učitavanje".Učitava se promenljiva <prom>, pri učitavanju se mora dati "štampiva" vrednost, a samoj promenljivoj se pridružuje njena ASCII-vrednost.Primer:

Ako na pitanje ?-get(X),write(X). kucamo znak A, onda X dobije vrednost 65, a ako kucamo recimo "neštampiv" znak Esc, onda se to ignoriše, "preskače".
Ako je pera.ari neka fajla, onda pitanjem
?-open(Rucka,'pera.ari',r),get(Rucka,Y),write(Y),close(Rucka).
se prvo ta fajla otvara za čitanje[16], pri tom otvaranju je fajli dodeljena Rucka, dalje posredstvom te ručke u narednoj get-formuli se iz fajle "uzme" prvi štampiv sastavak[17] Y, štampa se njegova ASCII-vrednost i na kraju se zatvara fajla pera.ari (ponovo posredstvom ručke).

g e t 0(<prom>), g e t 0(<ručka>,<prom>)
Predikati slični sa get-predikatima[18], ali se učitavaju i "neštampivi" znaci. Promenljivoj <prom> se pridružuje odgovarajuća ASCII- vrednost.

g e t 0 _ n o e ch o(<prom>)
Predikat getOnoecho služi za učitavanje znaka sa tastature, promenljivoj <prom> se pridružuje njegova ASCCI-vrednost. Međutim, učitavanje je bez ehoa, tj. kucani znak se ne javlja na ekranu.

g e t _ c u r s o r(<prom1>,<prom2>)
get_cursor(-,-)
Postavljanjem pitanja ?-get_cursor(X,Y) X i Y se vezuju za širinu i visinu korišćenog kursora.

h a l t
Predikat za izlaženje iz Prologa.

h i d e _ w i n d o w(<tekući>,<novi>)
Pitanjem oblika ?-hide_window(<tekući>,<novi>) se pravi prozor imena<novi> prelazi u njega, a uklanja se prozor imena <tekući>.

i f th e n(<for1>,<for2>)
Smisao: Ako važi formula <for1> izračunaj ("uradi") formulu <for2>.
Dopunski, ifthen-formula je tačna ukoliko formula <for1> nije tačna.
Primer:?-read(X),ifthen(X=2,(write('Jeste'),tab(3),write(X)),write('Kraj')
Ako se X-u da vrednost 2, onda će se sračunati navedena konjunkcija ( ) tj.štampaće se : Jeste 2, i iza tog štampaće se reč Kraj, a ako se X-u ne da vrednost 2, samo će se štampati reč Kraj.

i f th e n e l s e(<for1>,<for2>,<for3>)
Slično sa ifthen predikatom.Smisao:Ako važi formula <for1> izračunaj("uradi") formulu <for2>, a u protivnom izračunaj <for3>.

i n c(<broj>,<broj + 1>)
Predikat inc služi da se od datog broja <broj> pređe na broj za 1 veći.
Primer: ?-inc(6,X). X se vezuje za 7.

i n s t a n c e(<referentni_broj>,<podatak>)
instance(+,-)
Predikatom instance (vid. deo 8.2 u tački 8) se uz poznatost referentnog broja pronalazi odgovarajući podatak.

i n t _ t e x t(<broj>,<string>)
int_text(-,+), int_tex(+,-), -int_text(+,+)
Taj predikat služi za
"prebac" broja u odgovarajući string
"prebac" string u odgovarajući broj
proveru da li broj <broj> odgovara stringu <string>
Primeri:
?-int_text(23,X). X se veže za string '23'.
?-int_text(X,'567'). X se veže za broj 567.
?-int_text(45,'45') je tačna formula.

i n t e g e r(<dato>)
Smisao: <dato> je ceo broj.
Recimo, integer(34) je tačna formula, dok integer([1,2]) nije.

i s ; oblik formule: Prom is Izraz
Predikat is je "zadužen" za izračunavanja vrednosti aritmetičkih izraza. Naime, izračunava se izraz Izraz, a Prom se veže za tu vrednost.

k e y(<ključ>,<referentni_broj>)
key(+,-)
Predikatom key uz datost ključa možemo naći odgovarajući referentni broj.

k e y b(<Ascii-kod>,<Scan-kod>)
Predikat keyb služi za "dvo-kodno" čitanje znakova[19]. Tako se mogu čitati svi znaci sa tastature. Primer: Ako na pitanje ?-keyb(X,Y).
kucamo

A,	X će biti 65, a Y tzv. SCAN-kod će biti 30
Ctrl_Z	X će biti 26, a Y 44
Esc	X će biti 27, a Y 1
F9	X će biti 0, a Y 67.

k e y s(<ključ>)
Videti Primer 8.2.2, kao i iskaz (8.4.3.

k e y s o r t(<lista>,<rezultat>)
Taj predikat služi pri "poslu sređivanja po ključevima". Elementi polazne liste se moraju zadati u obliku ključ-vrednost. Primer:
?-keysort([c-23,a-45,b-22,a-5,c-7,b-67,b-9],X).
X se vezuje za listu [a-45,a-5,b-22,b-67,b-9,c-23,c-7].

---

[16] Treći open-argument je r, tj. read.

[17] Upravo sastavak na prostoru jednog bajta.

[18] Videti iznad.

[19] U Arity-prologu u vezi sa tim ima još nekih predikata,kao keyb sa 3 argumenta, i keyb_peek sa 2 i sa 3 argumenta.

l e n g th(<lista>,<duzina>)
length(+,-), -length(+,+)
Smisao: <duzina> je broj članova liste <lista>.

l i s t _ t e x t(<lista>,<string>)
list_text(+,-), list_text(-,+), -list_text(+,+)
Smisao: <string> je string čija slova,odnosno njihovi ASCI-kodovi po redu čine listu <lista>.Primer:
?-list_text([65,66],X). X se vezuje za string 'AB'

l i s t i n g, **l i s t i n g**(ime),
listing(ime/arnost),listing([ime1/arnost1],...,[imek/arnostk])
Iz tekućeg programskog sveta (code world) na ekran ispisuje članke datih imena, arnosti.Primeri:
Pretpostavimo da smo ušli u Arity-prolog i učitali fajlu imena 'pera.ari'
Tada sa
?-listing, ?-listing(a), ?-listing(a/2), ?-listing([a/3,b/2])
se na ekran ispisuju
svi članci te fajle, svi a-članci, svi a(_,_)-članci, tj. u kojima a ima arnost 2, odnosno svi a(_,_,_)- i svi b(_,_)-članci.

l o a d _ k e y(<ime_fajle>,<ključ>)
Tim predikatom se sastavci date fajle pod navedenim ključem zapisuju u tekući svet podataka[20] (data world).Inače jedan sastavak je napravljen upravo od svih znakova u jednom redu.Primer:
Pretpostavimo da fajla imena 'podaci' ima ovaj sadržaj (red za redom)
a b
c p:-q,
r,s,t.
Dalje, neka smo ušavši u Arity-prolog pitali
?-load_key('podaci',pera).
Tada su svi redovi fajle podaci, jedan za drugim, zapisani pod ključem pera. To se može proveriti postavljanjem ovog pitanja
?-recorded(pera,X,_),write(X),nl,fail.

l o ck / 0
Predikat kojim se može isključiti dejstvo Ctrl-Break signala. Primer:
Recimo, da imamo program sa ovim člancima
a:-write('pera'),tab(2),a.
b:-lock,a.
Tada na pitanje ?-a. na ekranu će se bez prestanka štampati reč pera sa međurazmakom 2 . Međutim, to "dešavanje" se može prekinuti uslugom Ctrl-Break signala, tj pritiskanjem i tipke Ctrl i tipke Break.Međutim, ako se pita ?-b. tada zbog lock predikata ne možemo sa Ctrl-Break da prekinemo tok algoritma.

m k d i r(<ime_direktorije>)
Predikat zadužen za pravljenje nove direktorije. Primer:
?-mkdir('jovica').Tako se u okviru tekuće direktorije pravi poddirektorija imena jovica.
?-mkdir('c:\jezik\prolog'). Tako se u direktoriji c:\jezik pravi nova poddirektorija imena prolog.

[20] Videti deo 8.2 u tački 8.

m o v e _ w i n d o w(<x>,<y>)
Taj predikat služi da se tekući prozor translira tako da mu nov levi gornji vrh bude u tački <x>,<y>.

n a m e(<reč>,<lista>)
name(+,-), name(-,+), -name(+,+)
Smisao: <lista> je lista ASCI-kodova pojedinih znakova reči <reč>.
Primeri:
?-name('A4f',X). X se vezuje za [65,52,102] jer ASCII-kodovi za A, 4 i f su redom 65, 52, 102.

n l
Znak za novi red pri ispisu.

n l(<ručka>)
Upisuje znak novog reda u fajlu zadane ručke (videti Zadatak 12.12, deo o Arity-prologu).

n o n v a r(<dato>)
nonvar(+)
Smisao: <dato> nije promenljiva ili jeste oblika promenljive koja već ima vrednost.

n o d e b u g
Prekida dejstvo debug-predikata(videti o njemu).

n o t(<for>)
Smisao: Formula not(<for>) je prološka negacija formule <for>

n o s p y(<ime>), **n o s p y**(<ime>/arnost)
Njegovom uslugom predikat zadan sa <ime> ili <ime>/arnost prestaje da tokom izvršenja programa više bude sa statusom "špijuniran". Taj status znači da se tokom programa na ekranu izlažu svi međukoraci pri "računanju" formule tog predikata. Videti o debug-predikatu.

n o t r a c e
Prekida dejstvo trace-predikata.Videti o njemu.

n r e f(<referentni_broj>,<sledeći>)
nref(+,-)
To je jedan od predikata o svetovima i record-predikatima. Videti deo 8.2 u tački 8.

n th _ ch a r(<broj>,<string>,<znak>)
nth_char(+,+,-)
Smisao: Uz dati <broj> i datu reč <string> sa <znak> je označen ASCII-kod slova te reči koje je od početka na mestu 1+<broj>. Primer:
?-nth_char(2,'petkovic',X). X se veže za ASCII-kod slova t,tj. 112

n th _ r e f(<ključ>,<broj>,<referentni_broj>)
Smisao: Uz dati ključ i dati <broj> sa <referentni_broj> je označen referentni broj podatka koji je pod tim ključem od početka na mestu <broj>. Videti deo 8.2 u tački 8.

n u m b e r(<dato>)
Smisao: <dato> je broj (ceo ili realan).

o p(<predn>,<tip>,<ime-izraz>)
Taj predikat služi za definisanje op-izraza, pomoću njegovog imena, tipa i broja "prednosti". Videti deo 4.3 u tački 4.

o p e n(<ručka>,<ime_fajle>,<pristup>)
open(-,+,+)

Predikat open služi za otvaranje već postojeće fajle i to prema vrednosti trećeg argumenta može biti:

r - za "čitanje" (read);
w - za "pisanje" (write);
a - za dopisivanje na kraj (append);
rw - za read i za write;
ra - za read i za append.

Pri otvaranju se kao rezultat dobija <ručka>. Videti i Zadatak 11.12.

p r e f(<referentni_broj>,<prethodni>)
pref(+,-)

To je jedan od predikata o svetovima i record-predikata. Videti deo 8.2 u tački 8.

p u t(<ascii>)
put(+)

To je predikat kojim se za zadan <ascii>, tj. neki ASCII-kod štampa odgovarajući znak, ako tom <ascii> odgovara neki znak. Ukoliko broju <asscii> ne odgovara neki znak, onda se ili ništa ne štampa ili se uradi nešto posebno. Recimo, sa ?-put(7) se dešava zviždanje (za kratko vreme). Dalji primeri:

?-put(65). Na ekranu se štampa A.
?-put(0). Na ekranu se ne pojavljuje ništa, a sama formula put(0) je tačna.
?-put(10) ili ?-put(13). "Štampa" se nov red, tj. prelazi se u nov red.

p u t(<ručka>,<ascii>)
put(+,+)

Predikat kao put-predikat (opisan iznad), ali ispis se obavlja u fajlu određenu njenom ručkom ("handle"). Videti Zadatak 11.12.

r e a d(<prološki_izraz>)

To je osnovni predikat uopšte za čitanje (read, input) nekog prološkog izraza, koji se od strane korisnika zadaje kao niz jedan za drugim kucanih znakova, pri čemu read-predikat pojavu tačke smatra za kraj izraza. Uz to Prolog proverava ispravnost ukucanog podatka, i ukoliko on nije ispravan prekida se algoritam sa porukom o grešci. Primer:

Ako na pitanje ?-read(X),write(X). kao podatak
damo[21] pera. onda će X biti vezano za atom pera, i na ekranu će se pojaviti ta reč;
damo f(1). X će biti vezano za f(1), što će se štampati na ekranu;
damo "pera". X će biti vezano za listu [112,101,114,97], čiji članovi su ASCII-kodovi redom za slova p,e,r,a , što je u skladu sa okolnošću kako uopšte Prolozi tretiraju stringove. Inače, na ekranu će se pojaviti ta lista.
damo Z. X će biti vezano za promenljivu Z, tj. za određenu adresu, koja će se posle štampati na ekranu;
damo f(). Na ekranu će se pojaviti poruka o sintaksnoj grešci i prekid toka algoritma.

[21] Mora se na kraju kucati znak tačke.

Međutim, ako želimo, možemo Prolog naterati da ne prijavljuje sintaksnu grešku već jedino da smatra da je tada formula read(X) netačna. Evo kako:
Koristimo uslugu syntaxerrors(_,_) predikata, odnosno pitanjem
?-syntaxerrors(_,off)
se postiže željeni cilj, tj. "isključivanje sintaksne analize". Iza toga postavimo gornje pitanje.
Inače, sa ?-syntaxerrors(_,on) možemo ponovo "uključiti sintaksni analizator.

r e a d(<ručka>,<prološki_izraz>)

Predikat sličan sa read (videti ispred), s tim što se ovim predikatom čita podatak po podatak iz neke fajle, posredstvom njene ručke ("handle"). Podaci moraju biti sintaksno ispravni. Primer:

Prepostavimo da smo već napravili fajlu imena 'jova' koja sadrži ove proloske članke

```
pera.   mile.   f(1).   g(1):-h(2).
```

Tada program[22]

```
ajde:-open(Rucka,'jova',r),
      ispisi(Rucka).
ispisi(Rucka):-repeat, read(Rucka,Izraz),
               write(Izraz),tab(3), Izraz==end_of_file,
                                    close(Rucka).
```

na pitanje ?-ajde. će na ekran ispisati pojedine sastavke (članke) fajle jova.

r e a d _ l i n e(<ručka>,<red>)

Predikat sličan predhodnom , ali ovaj čita red po red fajle zadane ručkom.

r e a d _ s t r i n g(<najveća_duzina>,<string>)

Predikat za učitavanje stringova do date dužine. Primer:
Ako na pitanje ?-read_string(3,X),write(X).
kucamo zoran. X će biti vezano za string 'zor' i to će biti štampano na ekranu;
kucamo z oran. (pazite sa medubelinom) X će biti vezano sa 'z o' što će biti štampano na ekranu.

r e a d _ s t r i n g(<ručka>,<najveća_dužina>,<string>)

Sličan prethodno opisanom read_string(_,_) predikatu, ali njime se obavlja čitanje iz date fajle, određene ručkom (videti o read-predikatu).

r e c o n s u l t(<ime_fajle>)

Sličan predikatu consult, s tim što "stare članke zamenjuje novim", u smislu:
Ako već u programu sa kojim radimo imamo recimo neke f / k članke, tj. članke imena f, arnosti k, onda ukoliko pri
?-reconsult('ime')
fajla 'ime' ima neke f / k članke onda se iz tekućeg programa izbacuju njegovi "stari" f / k članci i dodaju se takvi članci iz fajle 'ime'

r e c o r d _ a f t e r, r e c o r d a, r e c o r d e d, r e c o r d z

To su neki predikata o record-zapisima. Videti deo 8.2 u tački 8.

r e c o r d b / 3

[22] Primetite kako je kombinacijom
repeat,..., Izraz==end_of_file.
obezbeđeno "putovanje do kraja fajle".

Predikat za gradenje B-drveta. Videti deo 8.3 u tački 8.

r e c o r d h / 3
Predikat za gradenje hash-tabela. Videti deo 8.4 u tački 8.

r e f(<dati_broj)
-ref(+)
Smisao: Da li je <dati_broj) referentni broj nekog podatka. Primer:
Pretpostavimo da smo ušavši u Arity-prolog prvo napravili svet 'pera' pitanjem
```
?-create_world(pera).
```
a da smo zatim pitanjem
```
?-data_world(main,pera).
```
"ušli" u svet 'pera' i onda pitanjem
```
?-recorda(a,a(333),X).
```
upisali pod ključem a podatak a(333). Njemu će biti pridružen referentni broj X i na ekranu će se, recimo, pojaviti
```
X=~001B03C7
```
Onda možemo pitati
```
?-ref(~001B03C7).
```
i odgovor će biti YES ,tj. da.
Inače ref predikat je jedan od predikata o svetovima(videti deo 8.2 u tački 8).

r e f r e sh / 0
Pitanjem ?-refresh. svi trenutno postojeći prozori se ponovo grade. To pitanje može biti korisno ako pri radu sa raznim prozorima se dogodi neka "ekranska" nezgoda.

r e g i o n _ c, r e g i o n _ c a, r e g i o n _ c c
Ti predikati , ukratko rečeno, služe da se u okviru tekućeg prozora snimi pod nekim ključem željeno "parče" (region).Primer:
Recimo, da smo u nekom prozoru, nije važno kog imena. Tada sa
```
?-region_c((5,5),(15,20),String),recordz(proz,String,X).
```
pod ključem 'proz' je od tog prozora snimljeno parče čiji levi gornji ugao je u tački (5,5), a desni donji u[23] (15,20).
Snimljeno parče možemo "pozvati" sa recorded predikatom, tj. pitanjem
```
?-recorded(proz,X,_).
```

r e l a b e l _ w i n d o w(<novo_zaglavlje>)
Tim predikatom možemo promeniti zaglavlje tekućem prozoru. Primer:
Pretpostavimo da se trenutno nalazimo u prozoru čije je zaglavlje
```
Danas je sreda
```
i da hoćemo da novo zaglavlje bude
```
Danas je petak
```
Tada, prosto postavimo pitanje
```
?-relabel_window('Danas je petak').
```

r e m o v e a ll b(<ime_b_drveta>)
Služi za brisanje b-drveta datog imena. Videti deo 8.3 u tački 8.

r e m o v e b(<ime_b_drveta>,<clan>,<podatak>))
Briše u b-drvetu datog imena i datog "ključa" <clan> prvi njemu pridružen

[23] Ukoliko te tačke upadaju u tekući prozor; u protivnom snimanje se ne vrši.

podatak. Videti deo 8.3 u tački 8.

r e m o v e a ll h / 1
To je "hash" predikat. Videti deo 8.4 u tački 8.

r e m o v e h / 3
To je "hash" predikat. Videti deo 8.4 u tački 8.

r e n a m e(<prva_fajla>,<druga_fajla>)
Služi za promenu imena fajle. Recimo sa
```
?- rename('c:\pera','c:\jova').
```
fajla imena[24] 'pera' postaje fajla imena 'jova'.

r e p e a t
Osnovni predikat koji je tačan i koji je tačan i ukoliko se procedurom vraćanja (backtracking) dođe na njega. To sve važi zahvaljujući njegovoj definiciji:
```
repeat.
repeat:-repeat.
```

r e p l a c e(<referentni_broj>,<podatak>)
replace(+,-)
To je jedan od predikata o svetovima. Videti deo 8.2 u tački 8.

r e p l a c e b (<ime_b_drveta>, <ključ>, <stari_podatak>, <novi_podatak>)
Pomoću tog predikata u datom b-drvetu, uz poznat ključ možemo neki stari podatak zameniti željenim novim. Videti deo 8.3 u tački 8.

r e p l a c e h (<ime_hash_tabele>,<ključ>,<stari_podatak>,<novi_podatak>)
Sličan sa replaceb -predikatom, ali se odnosi na hash-tabele. Videti deo 8.4 u tački 8.

r e s i z e _ w i n d o w(<red>,<stubac>)
Služi da se u tekućem prozoru promeni "širina" i "dužina", tj. broj redova i stubaca. Primer:
Ako se nalazimo u nekom prozoru, onda pitanjem
```
?-resize_window(2,-5)
```
tom prozoru se za 2 povećava broj redova a ,za 5 smanjuje broj stubaca

r e s t o r e, r e s t o r e(<ime>)
Videti deo 8.2 u tački 8.

r e t r a c t(<izgled_članka>)
r e t r a c t(+)
Briše se prvi članak koji ima dati oblik[25], određen sa <izgled_članka>. Ime članka mora biti zadano. Primer:
Pitanjem ?-retract(f(X,Y)). se briše (ako ga ima) prvi članak datog izgleda. U takve dolaze
```
f(2,3)., f(3,A).
```
ali ne recimo članak f(2,3):-g(6,8). Ako želimo baš njega da obrišemo onda to postižemo pitanjem
```
?-retract((f(2,3):-g(6,8)).
```
ili pitanjem
```
?-retract((f(X,Y):-Z)).
```

[24] Nalazi se u direktoriji "na vrhu".

[25] "Oblicnost" je u stvari drugo ime za "ujednačiv" , gden se misli na ujednačavanje u smislu algoritma ujednačavanja(unifikacije).

ali ako smo sigurni da je taj članak prvi koji je tog oblika.

r e t r i e v e b / 3
To je predikat za B-drveta. Videti deo 8.3 u tački 8.

r e t r i e v e h / 3
Predikat o hash-tabelama. Videti deo 8.4 u tački 8.

r m d i r(<putanja>)
Služi za uklanjanje (prazne) direktorije date putanje.

s a v e, s a v e(<ime>)
Videti deo 8.2 u tački 8.

s c r ee n _ h e i gh t(<stara>,<nova>)
Tim predikatom se može menjati širina ekrana (uz pretpostavku prisustva određene grafičke kartice). Primeri:
?-screen_height(X,X). X će biti vezano za širinu ekrana
?-screen_height(_,50). Nova širina će biti 50, ako imamo VGA-karticu.
Inače, 25 i 43 su širine koje se mogu koristiti ako računar na kome radimo ima EGA, ili VGA karticu.

s e e(<ime_fajle>)

Tim predikatom se ,ukoliko je postojeća, otvara fajla datog imena i to otvara za "čitanje" (read); ona postaje tekuće input- sredstvo. Primer:
Pitanjem ?- see('fakt.ari'),read(X),write(X),seen.
se (za čitanje) otvara fajla 'fakt.ari', iz nje se čita X, zatim se to X štampa na ekran, i konačno uslugom seen-predikata se ta fajla zatvara. Inače za samo čitanje X-a koristi se predikat read (videti),sa svim pratećim ogradama.

s e e _ h(<ručka>)
Predikat sličan sa see, ali za njega se traži "ručka" (handle) fajle.

s e e i n g(<promenljiva>)
Pitanjem oblika ?-seeing(X). X se vezuje za ime fajle koja je prethodno već bila otvorena predikatom see (videti).

s e e k(<ručka>,<pomak>,<odakle>,<novo_mesto>)
seek(+,+,+,-)
Predikat seek, kratko rečeno, služi za pomicanje bajt-po-bajt po datoj fajli, zadanoj njenom "ručkom" (handle). Sa <pomak> je označen broj 0,1, 2,..., a <odakle> može biti:
bof ("od početka fajle")
eof ("od kraja fajle")
current ("od tekućeg mesta")
Neka je recimo <pomak>=5 tada:
Ako <odakle>=bof, onda <novo_mesto> je 5
Ako <odakle>=eof, onda <novo_mesto> je : dužina[1] fajle manje 5
Ako <odakle>=current, onda <novo_mesto> je u fajli za 5 bajtova "napred"
Primer:
Pretpostavimo da smo napravili fajlu imena 'podaci.dat' i da se ona sastoji iz ovakvih redova ,svaki od 38 znakova[2]

---

[1] U bajtovima.

[2] Na kraju svakog reda stoji * na 38-om mestu. Inače svaki red, budući da u

Danas je ponedeljak,radni dan *
Sutra treba da odemo tamo *
Iduce srede je praznik *
itd.

Tada program:
```
daj_podatak(X):-open(Rucka,'podaci.dat',r),X1 is (X-1) * 40,
                seek(Rucka,X1,bof,_),read_string(Rucka,40,Y),
                write(Y),close(Rucka).
```
je sposoban da iz fajle 'podaci.dat' izvuče ma koji red i štampa ga. Tako, pitanjem ?-daj_podatak(3). štampa se treći red

s e e n
Videti see(<ime_fajle>)

s e t _ c u r s o r(<širina>,<visina>), set_cursor(<izgled>)
set_cursor(+,+),set_cursor(<izgled>)
Tim predikatima[3] možemo menjati veličinu "kursora" (osnovnog znaka koji se pojavljuje na ekranu), odnosno njegov izgled; za <izgled> ima 4 mogućnosti
0,1,2,3
kojima odgovara ovakav izgled
nevidljiv, kao blok, kao polu-blok, podvučen

s e t o f
Videti (4.2.12)

sh e ll
Na pitanje ?-shell. iz Arity-prologa se izlazi u DOS, a iz njega možemo se vratiti u Arity kucanjem iz DOS-a: exit

sh e ll(<DOS-komanda>)
Predikat za izvršenje željene DOS-komande. Primeri:
?-shell('dir'). Izvršava se (DOS-ova) dir komanda
?-shell('c:\jezici\tc2\cprim1.exe'). Izvršava se fajla cprim1.exe

s k i p(<dat_znak>)
Predikat u vezi sa čitanjem (read) i preskakanjem znaka <dat_znak>. Primer:
Ako na pitanje ?-get0(X),skip('e'),get0(Y).
kucamo
abcdef.
onda X će biti vezano za 'a', dalje do 'e' uključujući i njega biće "preskakanje" i na kraju Y će biti vezano sa 'f'.

s k i p(<ručka>,<dati_znak>)
Sličan sa prethodnim skip-predikatom ali čitanje i preskakanje se odnosi na datu fajlu, određenu "ručkom" (handle).

s o r t(<lista>,<sređena_lista>)
sort(+,-), -sort(+,+)
Predikat sort služi za sređivanje (sortiranje) date liste; ponovljeni članovi se odbacuju. Primer:
?-sort([4,1,3,1],X). X se veže za [1,3,4].
?-sort([2,1],[1,2]. Odgovor je da (yes).

---

fajli moramo računati znak novog reda kao i znak čuvanja reda("line-feed") formalno ima po 40 znakova.

[3] Upotreba tih predikata takođe zavisi i od računara sa kojim radite.

s p y
Videti o debug-predikatu.

s t d i n(<iz_fajla>,<pitanje>),
s t d i n o u t(<iz_fajla>,<u_fajla>,<pitanje>)
s t d o u t(<u_fajla>,<pitanje>)
Standardna iz_fajla, fajla za čitanje (read) podataka je "tastatura", dok standardna u_fajla za ispis podataka je "ekran". Međutim, ponekad može biti poželjno da jedna ili obe te fajle budu neke po volji odabrane. To se može ostvariti korišćenjem tih predikata. Primer:

Pretpostavimo da smo već u Arity-prologu i da radimo sa nekim programom P. Tada, naravno možemo sa ?-listing. "izlistati" program P. Ali, ako uradimo ovako
?-stdout('mile.ari',listing).
onda se prvo pravi fajla imena 'mile.ari' i prehodni "listing" se smešta u nju. U stvari, na taj način smo program P snimili u fajlu imena 'mile.ari'.

s t o r e _ w i n d o w s / 0
Ako se nalazimo u nekom prozoru i nakon izvesnog rada u njemu hoćemo da sačuvamo sve ispise na njemu ,onda u tu svrhu služi taj predikat. Jednostavno postavimo pitanje ?-store_windows.

s t r i n g(<dato>)
Smisao: <dato>) je string, tj. to je reč kojoj prvo i poslednje slovo su znak $. Primeri:
?-string($pera$). Odgovor da.
?-string('pera'). Odgovor ne.

s t r i n g _ l e n g th(<string>,<duzina>)
string_length(+,-), -string_length(+,+)
Smisao: <duzina> je duzina stringa <string>. Primeri:
?-string_length($pera$,X). X se veže za 4.
?-string_length($pera$,4). Odgovor je da.

s t r i n g _ s e a r ch(<pod_string>,<string>,<mesto>)
string_search(+,+,-), -string_search(+,+,+)
Smisao: <pod_string> je podreč, podstring za <string> i ta podreč počinje od mesta <mesto>. Primeri:
?-string_search(rad,$prerada$,X). X se vezuje za 3, jer početno mesto se broji sa 0.
?-string_search(rad,$prerada$,3). Odgovor je da.
Predikat string_search nije "jednogran", tj. pri proceduri vraćanje (backtracking) je sposoban da nađe sve vrednosti za <mesto>. Recimo, pri raspravljanju pitanja
?-string_search(as,$perasasd$,X),write(X),fail.
X će biti vezan prvo za 3, a zatim za 5.

string_search(<vel_slova>,<pod_string>,<string>,<mesto>)
Predikat potpuno sličan prethodnom, s tim da argument <vel_slova> može biti 0 ili 1. Ako je 1 to znači da se ne pravi razlika između malih i velikih slova. Primer:
?-string_search(1,Per,$kopernik$,X). X se veže za 2
?-string_search(0,Per,$kopernik$,X). Nema tog X, odgovor je ne.
Razlog: prvi argument je 0, pa se pravi razlika između malih i velikih slova.

I taj predikat nije "jednogran", tj. pri backtacking-u je sposoban da da i i druge (ako ih ima) vrednosti za <mesto>.

s t r i n g _ t e r m(<string>,<izraz>)
string_term(+,-), string_term(-,+), -string_term(+,+)
Smisao: <izraz> je pravilan proloski izraz, a <string> je njegov odgovarajući string. Primeri:
?-string_term($pera$,X). X se vezuje za atom pera.
?-string_term(X,pera). X se vezuje za string $pera$
?-string_term($pera()$,X). Odgovor je ne, jer reč pera() nije pravilan proloski izraz.
?-string_term($pera(5)$,X). X se vezuje za pera(5).

s y n t a x e r r o r s(<staro_stanje>,<novo_stanje>)
Taj predikat je "zadužen" da brine o eventualnim sintaksnim greškama, pojavljenim tokom rada programa. Videti o read-predikatu, kao i o predikatu fileerrors koji umesto o sintaksnim greškama brine o greškama u radu sa fajlama.

s y s t e m(<ime> / arnost)
-system(+), system(-).
Smisao: Da li je <ime> ime nekog sistemskog, tj. u Prolog ugrađenog predikata. Primeri:
?-system(float / 1). Odgovor je da.
?-system(call / 1). Odgovor je da.
?-system(X),write(X),nl,fail. X će redom biti vezano za svaki od proloških sistemskih predikata.

t a b(<brojka>), **t a b**(<ručka>,<brojka>)
Predikat tab služi za ispis <brojka> znakova "belina" ("razmaka"). U drugom gom slučaju taj ispis se odnosi na fajlu zadane ručke.

t e l l(<ime_fajle>), **t e l l i n g**(<promen>), **told**
Smisao: Ako je <ime_fajle> dato ime, pitanjem
?-tell(<ime_fajle>)
se fajla tog imena stvara i otvara za pisanje, i postaje tekuće "upisno sredstvo", što znači da recimo, write predikat ima dejstvo u tu fajlu, a ne na ekran. Pitanjem ?-telling(X). X se vezuje sa imenom fajle koja je tekuća upisna fajla ("upisno sredstvo"). Pitanjem ?-told. se zatvara fajla koja je prethodno bila sa tell otvorena. Videti Zadatak 11.11.
Primeri:
Ako odmah pri ulasku u Arity-prolog pitamo ?-telling(X). X će biti vezano za ime 'user' (tako se zove "ekranska fajla").
Ako pitamo
?-tell('pera.ari'),write('Danas'),telling(X),told.
onda prvo fajla imena 'pera.ari' se stvara i otvara u nju se upisuje reč Danas, dalje zbog telling(X), X se vezuje za ime 'pera.ari' i na kraju, zbog told fajla 'pera.ari' se zatvara i izlazi se iz nje.

t e l l _ h(<ručka>)
Slično kao tell predikat, ali fajla se zadaje ručkom. Međutim, za razliku od tell predikata iza korišćenja tell_h bezuspešno je korišćenje telling predikata.

t g e t(<red>,<stubac>)
Na pitanje ?-tget(X,Y). X se vezuje za brojku reda, a Y za brojku stupca u kome se trenutno nalazi kursor.

t m o v e(<red>,<stubac>)

Seli kursor u ekransku tačku (<red>,<stubac>). Videti Zadatak 12.18.

t o l d
Videti o tell-predikatu.

t r a c e / 0
Osnovni među tzv. "debuger"-predikatima, odnosno predikatima koji omogućuju praćenje toka programa, korak po korak. Primer:
Pretpostavimo da u okviru Arity-prologa smo učitali neki program i da onda postavimo pitanje ?-trace. što će prouzrokavati "buđenje", odnosno uključivanje "debugera"[4]. Ako iza toga postavimo neko pitanje
?-ϕ
onda korak za korakom ćemo videti razvoj razmatranja tog pitanja.
Ako hoćemo da "se oslobodimo" dejstva trace-predikata onda možemo postaviti pitanje ?-notrace, tj. koristiti uslugu notrace-predikata (videti primer u objašnjenju za naredni predikat).

t r a c e(<ime_fajle>)
Sličan sa trace ali se čitav tok razvoja programa usnimava i na fajlu datog imena. Primer:
Pretpostavimo da smo u Arity-prologu i da imamo ovaj program za faktorijel

```
fakt(0,1).
fakt(X,Y):-X>0,X1 is X-1,fakt(X1,Y1),Y is X*Y1.
```

Dalje, zamislimo da hoćemo da se pitanje ?-fakt(3,X). raspravi upotrebom trace-predikata, tj. raspravi do svakog međukoraka, i da uz to želimo da ta rasprava, taj tok bude zapamćen, snimljen u fajli imena 'tok'. To se može postići postavljanjem ovih pitanja:

```
Prvo:    ?-trace('tok').
Drugo:   ?-fakt(3,X).
Treće:   ?-notrace.      (Ako hoćemo da se oslobodimo trace-a).
```

t r u e
"Formula" true je uvek tačna.

v a r(<dato>)
Smisao: Da li je <dato> promenljiva, koja još nije dobila vrednost ili koja je dobila, ali ona je neka druga promenljiva. Primeri:
Pitanje ?- X is 3,var(X). Odgovor je ne, jer X ima vrednost 3.
Pitanje ?- X=Y, var(X). Odgovor je da, jer vrednost X-a je promenljiva.

w c(<broj>,<znak>)
Služi da se <znak> na ekranu ispiše <broj>-puta. Videti Zadatak 12.18.

wh a t _ b t r ee s(<promen>)
Videti deo 8.3 u tački 8. Na pitanje oblika
?-what_btrees(X),write(X),nl,fail.
se dobija spisak svih već napravljenih B-drveta.

wh a t _ w i n d o w s(<promen>)
Jedan od prozorskih predikata. Na pitanje oblika
?-what_windows(X),write(X),nl,fail.
se dobija spisak svih već napravljenih prozora (videti o predikatima define_window, current_window).

[4] To je ustvari dopunski program.

wh a t _ w o r l d s(<promen>)
Videti Primer 8.2.2. Na pitanje oblika
?-what_worlds(X),write(X),nl,fail.
se dobija spisak svih već napravljenih svetova.

w i n d o w _ i n f o / S
Taj predikat ima argumente iste kao i predikat define_window(videti gore).
Primer:
Na pitanje

```
?-window_info(A,B,C,D,E,F),write(A),tab(2),write(B),tab(2),
        write(C),tab(2),write(D),tab(2),write(E),fail.
```

dobiće se spisak svih već otvorenih prozora sa njihovim odrednicima:
naziv, zaglavlje, itd.
Kao obavezno javiće se podaci o ova dva prološka prozora:
main i debuger

w r i t e(<dato>)
Osnovni predikat za ispis (na ekran) argumenta <dato>. On mora da bude ispravan prološki izraz. Primeri:

?-write(pera).	Ispisaće se reč (atom) pera.
?-write(X).	Ispisaće se nešto kao _0084, što je u stvari pridružena adresa za X ispred koje stoji podcrtica.
?-write(pera(3)).	Ispisaće se pera(3).
?-write("jova").	Ispisaće se lista [106,111,118,97], jer Arity-prolog reč između navodnica " " shvata kao listu ASCII kodova njenih slova (ne računajući navodnice).

w r i t e(<ručka>,<dato>)
Taj predikat je sličan sa write-predikatom, prethodno opisanim, ali ispis se obavlja u fajlu određenu njenom "ručkom" (handle).

12.3 LPA - prolog

Taj prolog , uprkos znatnoj sličnosti sa Micro-prologom, pripada prolozima Edinburgške sintakse. Shodno tome ima veoma mnogo zajedničkih predikata LPA- i Arity-prologa. Predikate Arity-prologa smo opisali u prethodnom izlaganju 12.2 i u ovom opisu često ćemo se pozivati na njih i koristiti ih tj. tokom ovog opisa nećemo ih ponovo izlagati. To praktično znači da se ovaj opis uglavnom odnosi na predikate u kojima se LPA-prolog razlikuje od Arity-prologa. Najpre navodimo spisak posebnih znakova i funkcija, a nakon toga sledi spisak predikata redan abecedno.

P o s e b n i z n a c i i f u n k c i j e

&
Oznaka za osnovni modul ("workspace"). Ako pri radu ne pređemo u neki modul koji je prethodno posebno napravljen, onda smo stalno u tom osnovnom modulu.

&:
Oznaka za osnovni prozor("ceo ekran") u kome se nalazimo odmah pri prelazu u Prolog i ostajemo tu dok ne napravimo neki drugi prozor i pređemo u njega:

'LST:'	oznaka za printer(njegovu "fajlu").
'TRM:'	oznaka za tastaturu(njenu "fajlu").

'WND:' oznaka za tekući prozor

!
Znak reza, videti izlaganje: 2.3 Pravilo o REZU. Videti i o call-predikatu.

\(Prom) duboki rez
Videti u tački 2, deo 2.3 odnosno program 2.3.2.

%
Iza tog znaka može se -u oviru jednog reda- staviti ma koji tekst, komentar, koji se od strane Prologa ignoriše.

, Arnost je 2
Znak konjunkcije (videti (4.2.1))

; Arnost je 2
Znak disjunkcije (videti (4.2.2)).

*
Oznaka množenja

+
Oznaka sabiranja

-
Oznaka oduzimanja

/
Oznaka delenja

//
Oznaka celobrojnog delenja

mod
Oblik korišćenja: X mod Y . Smisao: X mod Y je ostatak pri delenju X sa Y. Primer: U pitanju ?- X is 23 mod 7. X se vezuje za 2.

^ pri upotrebi oblika X^Y, gde X, Y realni brojevi
označava stepen "iks na ipsilon". Primer: U pitanju X is 3^2. X se vezuje za 9.

sin, cos, tan, log, ln ,abs, asin, acos, atan, alog, aln, sdrt
su oznake dobro poznatih funnkcija. Primeri: U pitanjima
?- X is sin(1). ?-X is abs(-2). ?-X is sqrt(2)
X se vezuje za 0.84147098 ,odnosno 2, odnosno 1.41421356, dok u pitanju ?-X is alog(2). X se vezuje za 100 (tj. 10 na kvadrat), jer log(100)=2.
Vidi se da u LPA-prologu sa asin,acos,atan,alog,aln su označene inverzne funkcije za sin,cos,tan,log,ln.

irand u obliku irand(X)
služi za nalaženje "slučajnog" celog broja između 0 i X-1. Primer:
U pitanju ?-X is irand(7.6)
može se dogoditi da X bude vezano za 6 (ili uopšte neki od 0,1,...,7).

f l o o r(X) označava najveći ceo broj manji ili jednak X

f p(X) označava razlomljeni deo realnog broja X. Recimo, fp(6.3) je 0.3.

i p(X) označava celi deo realnog broja X

X/\Y 2-konjukcija X i Y
Logička konjunkcija 2-zapisa brojeva X,Y. Primer (račun korak za korakom):


```
9/\27=     1 0 0 1   Prvo smo 9 i 27 iskazali u osnovi 2
     /\ 1 1 0 1 1
     =  0 1 0 0 1   2-Cifre smo redom "spajali" po pravilima konjunk-
                    cije: 1/\1=1, 1/\0=0 i sl.
     = 9  Rezultat iskazujemo u 10-zapisu (desetično).
```

X\/Y 2-diskjunkcija X i Y
Ta je funkcija slična prethodnoj s tim što se 2-cifre pri računu spajaju po pravilima disjunkcije: 0\/1= 1\/0= 1\/1=1, 0\/0=0.

X<<Y 2-zapis od X se ulevo pomeri za Y mesta.
Videti i dve funkcije ispred. Primer
5<<1= 1 0 1 pomaknuto ulevo za 1 (5 u 2-zapisu glasi 1 0 1)
=1 0 1 0 Pri pomaku smo na kraj dopisali 0
=10.

X>>Y 2-zapis od X se udesno pomeri za Y mesta
Slično prethodnoj funkciji, ali pomak je udesno.

= Arnost je 2
Predikat = u potpunosti odgovara unifikaciji. Videti više u opisu EQ-predikata u delu 12.1 ,tj. opisu Micro-proloških predikata.

\= Negacija od =

== Bukvalno jednaki

Primeri: Tačne su formule 2==2, [1,3]==[1,3], X==X, ali nije tačna
X==Y
jer promenljive X,Y su različite, tj. sa različitim adresama (u memoriji).
Međutim, ova konjunkcija

X=Y, X==Y

je tačna, jer zbog = najpre se X,Y unificiraju, što se svodi na :
Y promeni svoju adresu i kao novu uzme adresu od X

\== Negacija od ==

:- / 2
Znak "grlo" za građenje nejednočlanih članaka, koji se mogu gledati kao ovakve strukture ':-' (Glava, Rep). Videti (4.2.5).

izraz1 < izraz2 < je znak "manji"

Smisao: vrednost izraza izraz1 je manja od vrednosti izraza izraz2. Tačne su formule 2<3, 6+5<7*4-1.

izraz1 > izraz2 > je znak "veći"
Smisao: vrednost izraza izraz1 je veća od vrednosti izraza izraz2.

izraz1 =:= izraz2 =:= je znak "jednakovrednosni"
Smisao: vrednosti izraza izraz1, izraz2 su jednake. Recimo, tačna je formula 2+3*4=:=7*2.

=\= Negacija za =:=

=< manji ili jednak, tj. negacija za >

>= veći ili jednak, tj. negacija za <

=.. Strukt=.. Lista
O tom predikatu videti u tački 1, deo 2.

@< leksički manji
Primer: ana @< beba tačno, dok pera @< mika netačno

@> leksički veći
Primer: pera @> mila je tačno, dok a23 @> a32 je netačno.

@=< leksički manji ili jednak, tj. negacija za @>.

@>= leksički veći ili jednak, tj. negacija za @<.

\+ drugi zapis za not.

[fajla1,fajla2,...] Ako smo već ušli u LPA-prolog onda takvim pitanjem se učitava jedna za drugom niz fajli fajla1, fajla2,...
Primer: Na pitanje
        ?- [pera,mile,jova].
učitavaju se prvo fajla pera[5], pa iza nje mile i najzad fajla jova.

^ Znak kvantora[6] postoji. Videti (4.2.12).

Abecedni spisak predikata

a b o l i sh
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

a b o r t
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

a r g
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

a ss e r t, a ss e r t a, a ss e r t z a
O tim predikatima videti u delu 12.2, tj. opisu predikata Arity-prologa.

a ss e r t x(<članak>,<mesto>)
Pored gore navedenih assert-predikata, LPA-prolog ima i taj. Njegovom uslugom se željeni <članak> može dodati, i da pritom u spisku svih članak istog imena bude stavljen na mesto <mesto>. Primer:
  Pretpostavimo da tekući program sadrži ove a-članke
    a:-b.
    a(X):-write(X).
  Tada pitanjem
     ?-assertx(:-(a,c),2).
  članak a:-c. postaje novi drugi a-članak. Znači, spisak a-članaka glasi
    a:-b.
    a:-c.
    a(X):-write(X).

a t o m
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

a t o m i c
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

---

[5] Tj. pera.dec .

[6] Videli smo napred da se ^ koristi i kao znak stepena.

b a g o ff
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

c a ll
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

c d e f<dato>
Smisao: da li je <dato> ime statičkog predikata. Videti deo 10.2 u tački 10 (Izvršne fajle u prologu).

ch a r o f(<slovo>,<broj>)
Smisao: <broj> je ASCCI kod za <slovo>. Recimo, u pitanju
     ?-charof('A',X),charof(Y,66).
X se vezuje za 65, a Y za 'B'.

ch d i r
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

c l a u s e(<glava>,<rep>)
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa. U LPA-prologu koristi se i sledeća clause/3 formula:
    clause(<glava>,<rep>,<lista>)
sa smislom
    <rep> je rep članka čija glava je <glava>, i uz to <lista> je lista svih promenljivih u članku. Svuda se umesto promenljivih pojavljuju njihovi ground-zamene, tj. pojavljuju se imena promenljivih. Inače <glava> mora biti zadana.
Primer:
    Neka tekući ima ove p-članke
       p:-q.
       p(X,Y):-e(X),f(Y).
    Tada na pitanje
      ?-clause(p(X,Y),Z,U).
    imaćemo ova "vezivanja"
      X='X',Y='Y'
      Z=e('X'),f('Y').

c l a u s e x(<glava>,<rep>,<odakle>,<mesto>)
c l a u s e x(<glava>,<rep>,<lista>,<odakle>,<mesto>)
Ti su predikati slični sa clause-predikatima, opisanim prethodno. Proširenje je što ovi sadrže
    <odakle> -to je redno mesto od koga se traži članak
    <mesto> -to je mesto članka
    <lista> -spisak ground-oblika svih promenljivih članka.
Primer:
   Neka tekući program ima ove p-članke
     p:-q1.
     p:-q2.
     p(X):-q(X).
     p:-q3.
   Tada na pitanje ?-clausex(p,X,2,N),write(X),tab(2),write(N),fail.
   imaćemo redom ova "vezivanja"
     X=q2, N=2
     X=q3, N=4   Pazite N se odnosi na sve p-članke, pa se u članku
                 p(X):-q(X) smatra N=3.

c l o a d

Videti deo 9.2 u tački 9, o izvršnim fajlama Prologa.

c l o s e(<ime>)
Smisao: zatvori fajlu ili prozor zadanog imena <ime>.

c o m p a r e
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

c o m p o u n d<dato>
Smisao: <dato> je složen term(struktura) kao što su f(1), g([1,X],3),a nisu 23, [1,2] i sl.

c o n c a t
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

c o n s u l t
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

c r e a t e<ime>
Smisao: Napraviti(kreirati) fajlu imena <ime>, za upis-ispis.

c r m o d
Predikat za pravljenje modula, videti tačku 10, (10.2).

c r w i n d
Taj predikat zajedno sa cuwind-i cursor-predikatom pripada prozorskim predikatima.U osnovi su slično uvedeni kao istoimeni predikati Micro-prologa. Videti deo 12.1, tj. opis Micro-proloških predikata. Pored njih u LPA ima ima i pcwind -predikat(videti niže).U njegovom opisu ima i primera upotrebe crwind i cuwind predikata.

c s a v e
Videti deo 9.2 u tački 9 o izvršnim fajlama Prologa.

c u m o d
Predikat za module. Videti tačku 10, (10.1).

c u r r e n t _ o p
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

c u r s o r
Prozorski predikat (slično kao u Micro-prologu). Videti napomenu o crwind-predikatu.

c u w i n d
Prozorski predikat. Videti napomenu o crwind-predikatu

d a t e(<dan>,<mesec>), **date**(<dan>,<mesec>,<godina>)
date(-,-), **date**(-,-,-)
Videti o DATE predikatu u delu 12.1, tj. opisu Micro-proloških predikata.

d e b u g
Pored **trace**, primer tzv. "debuger"-predikata, odnosno predikata koji omogućuje praćenje toka programa, korak po korak. Za razliku od trace pomoću debug predikata tokom izvršenja ne prate se švi predikati već samo oni koji se za to naznače uslugom predikata **spy**. Primera radi zamislimo da tekući program sadrži predikate a,b,c,d, i da tokom raspravljanja pitanja

```
?-a(X).
```

hoćemo da vidimo ponašanje predikata a i c. To možemo postići ovako
Prvo pitanje: ?-debug.
i tako smo "uključili" debug-predikat.

Drugo pitanje: ?-spy([a,c]).
i tako naznačujemo želju da "spijuniramo" a i c. Da recimo želimo da špijuniramo sve predikate onda bismo umesto tog mogli postaviti ovo pitanje ?-spy(all).
Sada su obavljene pripreme za glavno pitanje:

```
?-a(X).
```

Na ekranu će se korak-za-korakom objasniti njegovo raspravljanje, sa isticanjem dešavanja u vezi sa predikatima a i c.
Nakon završenog raspravljanja možemo recimo sa

```
?-nospy(a). ili ?-nospy(c). ili nospy([a,c]).
```

ukiniti "špijunski" status za a, odnosno c ,odnosno i za a i za c. Da su svi predikati bili sa "špijunskim" statusom onda bismo im ga mogli ukinuti ovim pitanjem ?-nospy(all).
Takođe, ako u nastavku rada nećemo uslugu debug-predikata onda pitanjem

```
?-nodebug.
```

ga "ukidamo".
Inače uopšte tokom "skroziranja" toka programa po pravilu se ističu ovakvi detalji:

call	<for>	-kad se dođe do računanja formule <for>
exit	<for>	-kad se sa uspehom završi računanje formule <for>, tj. dokaže <for>.
redo	<for>	-kad se pri vraćanju (backtracking) ponovo dođe do formule <for>.
fail	<for>	-ako se dogodi da se računanje formule <for> završi sa ne.

U LPA-prologu se uslugom tzv. debug_ports predikata može postići da se pri "skroziranju" istaknu samo neki od tih detalja. Evo kako se to može že učiniti. Uočimo ovu malu tabelu sa četiri polja

Call Exit Redo Fail

gde svako polje može biti popunjeno sa jednom od reči: on, off.
Recimo jedno popunjenje glasi

on on off off

Reći ćemo slobodnije da su uključena polja Call, Exit(na njima stoji on) a isključena Redo, Fail (jer na njima stoji off). Ako s tim u vezi želimo da se na ekranu ističu samo Call i Exit detalji, onda se to može postići postavljanjem ovog pitanja

```
?-debug_ports(on,on,off,off).
```

Recimo,ako bismo želeli da vidimo samo ona mesta na koja se prološki algoritam vraća onda to postižemo pitanjem

```
?-debug_ports(off,off,on,off).
```

jer tada je "uključeno" samo Redo polje.

d e b u gg i n g
Pitanjem ?-debugging. na ekranu se pojave pojedinosti u vezi sa debug-odnosno trace- predikatom, kao koji s predikati naznačeni kao spy i dr. Videti o debug-predikatu.

d e b u g _ p o r t s
Videti o predikatu debug.

d e b u g _ s t r e a m(<ime>)
Taj predikat je potpuno sličan sa debug-predikatom, opisanim gore ali sa razlikom što se sve "debug" informacije upisuju u fajlu imena <ime>, koja se inače najpre napravi.Međutim, ta fajla može biti i "uopštenija kao printer, prozor i sl. Podrobije rečeno, <ime>

'LST:' se odnosi na printer

'WND:' na tekući prozor
'&:' na osnovni prozor ("ceo ekran")

d e b u g _ w i n d o w(<red>,<kolona>,<dubina>,<širina>)
To je jedan od debug-predikata. Videti napred o predikatima: debug i debug _stream. Pomoću debug_window predikata možemo sami po želji napraviti prozor, tako da pri debug-procesu sve informacije se pojave u njemu.
Primer: Pitanjem
?-debug_window(0,40,20,30)
se pravi prozor za "smeštaj" debug-informacija. Levi gornji ugao tog prozora je u početnom redu, i koloni(stubcu) 40. Dalje, njegova "dubina" je 20, tj. u njemu ima 20 redova, a širina mu je 30.

d e f
Videti o DEF predikatu u delu 12.1, tj. opisu predikata Micro-prologa.

d e l(<ime_fajle>)
Smisao : obristai fajlu datog imena <ime_fajle>).

d i c t
Videti tačku 10, deo oko formule (10.5).

d i r
Videti o DIR predikatu u delu 12.1, tj. opisu predikata Micro-prologa.

d i s p l a y(<izraz>)
Dati izraz preobraća na prefiksni poljski oblik. Primer:
?-display(2+3). Na ekranu se pojavi '+'(2,3)
?-display(2+3*4). Na ekranu se pojavi '+'(2,'*'(3,4))

d r i v e(<disk>
Predikat u vezi sa diskom (na kome radimo ili na koji hoćemo da pređemo). Primeri:
Pretpostavimo da smo ušli u LPA-prolog, koji se inače nalazi recimo na disku C. Tada pitanjem
?-drive(X).
X se vezuje za 'C'.Međutim, ako hoćemo da pređemo na disk 'A', onda to možemo postići pitanjem ?-drive('A').Evo nešto složenijeg primera.
Pretpostavimo da smo ušli u LPA-prolog, koji je na 'C'disku, i da sa diska 'A' hoćemo da učitamo fajlu imena 'mile.dec' (puna putanja je 'A:\mile.dec'). To možemo postići ovim pitanjem
?-drive('A'),consult('mile.dec'),drive('C').

e o f(<ime_fajle>)
Pri radu sa fajlama taj predikat sluzi za naznačavanje kraja fajle datog imena.

e x p a n d _ t e r m(<gramatički_izraz>,<prološki_prevod>)
Videti opis tog predikata u delu 12.2, tj. opisu predikata Arity prologa.

f a i l
Ta "formula" je po definiciji netačna.

f a l s e
Uvek netačna "formula"(sinonim za fail).

f d i c t
Pitanjem oblika ?-fdict(X). X se vezuje za listu svih fajli koje su do tog trenutna otvorene, pomoću open-predikata ili su novo-napravljene pomoću

create-predikata.

f i l l
Jedan od grafičkih predikata. Videti Zadatak 11.20.

f i n d(<ime_fajle>,<dato>)
Smisao: da li je <dato> reč ,tj. sastavak fajle imena <ime_fajle>. Primer:
Neka tekstovna fajla imena 'dati' ima na početku ovakav sadržaj
Danas je subota i lepo je vreme
Tada, na pitanje
?-open('dati'),find('dati','lepo'),close('dati').
odgovor je da. Međutim,pritom se ujedno fajlin pokazivač postavlja na kraj nađene reči. Stoga na primer uz uslugu fr-predikata možemo iza te "otkinuti" podreč date dužine. Recimo, zamenom tog pitanja ovim
?-open('dati'),find('dati','lepo'),fr('dati',[c(7),[X]),close('dati').
X se vezuje za reč ' je vre'

f i n d a l l(<uzorak>,<formula>,<lista>)
Smisao <lista> je lista svih (ne nužno različitih) objekata oblika<uzorak> za koje je tačna formula <formula>. Primer:
Neka neki program pored ostalog sadrži ove a-članke
a(1). a(4). a(2). a(1).
Tada u pitanju
?-findall(X,a(X),L). L se vezuje za listu [1,4,2,1]
a u pitanju
?-findall(f(X),a(X),L. L se vezuje za listu [f(1),f(4),f(2),f(1)].
Videti i u tački 4 deo oko formule (4.2.11).

f l d a t a(<dato_ime>,<ime>,<eksten>,<vreme>,<datum>,<veličina>,<atribut>)
fldata(+,-,-,-,-,-,-)
Smisao: Ako se zada prvi argument,tj. puno ime neke fajle onda redom ostali argumenti sadrže
ime fajle(bez ekstenzije), ekstenziju imena, vreme u časovima,minutima sekundama kad je fajla napravljena, datum, tj. dan njenog pravljenja, veličinu i konačno atribut fajle.
Atribut je jedan od brojeva 0,1,2,...,7 i recimo ako je 0 to znači da je fajla samo za čitanje, ako je 1 da fajla "sakrivena", ako je 2 da je fajla sistemska, i dr.

f l o a t(<dato>
Smisao: <dato> je realan broj. Recimo: formula float (8.9) je tačna, a netačne su formule float(8), float('Pera').

f o r a l l
Videti Zadatak 3.24.

f r
Formatirano čitanje podataka. Videti Zadatke 11.13 i 11.14, kao i Napomene 11.3 i 11.4.

f u n c t o r
Videti o tom predikatu u delu 12.2, tj. opisu Arity predikata.

f w
Formarirano pisanje podataka. Videti Zadatke 11.14 i 11.13, kao i Napomene 11.14 i 11.13.

g d e v
Jedan od grafičkih predikata. Videti Zadatak 11.20.

g e t(<prom>), g e t(<ime_fajle>,<prom>),
g e t0(<prom>), g e t0(<ime_fajle>,<prom>)
O tim predikatima videti u delu 12.2, tj. opisi Arity predikata. Jedino, za razliku od Arity-a LPA-prolog umesto "ručke" korist ime fajle.

g m o d
Jedan od grafičkih predikata. Videti Zadatak 11.20.

g r e a d(<prom>)
Pored read-predikata predikat za "učitavanje", ali za razliku od njega ako se za učitavanje zada promenljiva onda <prom> kao vrednost dobije odgovarajući atom (pod znacima ' ').Primer:
Ako na pitanje ?-gread(X). kucamo
J.
tj. kucamo promenljivu, onda X se vezuje za atom 'J'.S druge strane ako na pitanje ?-read(X) kucamo
J.
onda se X vezuje za izvesnu adresu, odnosno na ekranu se štampa nešto kao _4A93

g r e a d(<ime_fajle>,<prom>,<lista>)
g r e a d(+,-,-)
Sličan sa gread/1 predikatom s tim što tim predikatom se vrši učitavanje iz fajle zadanog imena, <prom> se vezuje za odgovarajući atom, dok <lista> je lista promenljivih (kao atoma, kao reči) učestvujućih u učitanom podatku. Primeri:
Ako na pitanje[7] ?-gread('BUF:',X,Y). kucamo f(A,B) onda
X se vezuje za f('A','B'), a Y za listu ['A','B'].

g s x
Jedan od grafičkih predikata. Videti Zadatak 11.20.

h a l t
Predikat za izlaženje iz Prologa. Naime, pitanjem ?-halt. se obavlja izlaženje.

i d e f<dato>
Smisao: da li je <dato> ime dinamičkog (interpreterskog) predikata.
Videti deo 9.2 u tački 9 o izvršnim fajlama.

i n t e g e r(<dato>)
Smisao: <dato> je ceo broj.

i n x y
Jedan od grafičkih predikata. Videti Zadatak 11.20.

k i ll(<ime_predikata>)
k i ll(<lista_imena_predikata>)
k i ll(all)
Predikat kill služi za uklanjanje, brisanje svih članaka naznačenih imena.
Primeri:
?-kill(all). Brišu se svi članci tekućeg programa.
?-kill(fakt). Brišu se svi članci sa imenom fakt.
?-kill([per,der]). Brišu se svi članci imena per i imena der.

[7] Podsećamo da sa 'BUF:' je označena "fajla tastature".

l e n g th(<dato>,<duz>)
l e n g th(+,-), -l e n g th(+,+)
Smisao: <duz> je dužina od <dato>, koje može biti lista, atom, struktura.
Primeri:
?-length(pera,X). X se vezuje za 4
?-length([a,b,c],X). X se vezuje za 3
?-length(f(a,b,c),X). X se vezuje za 4, jer strukturi f(a,b,c) odgovara lista[8] [f,a,b,c] dužine 4.

l i n e
Jedan od grafičkih predikata. Videti Zadatak 12.20.

l i s t i n g, l i s t i n g(ime),
listing(ime/arnost),listing([ime1/arnost1],...,[imek/arnostk])
Iz tekućeg modula na ekran ispisuje članke[9] datih imena, arnosti.
Primeri:
Pretpostavimo da smo ušli u LPA-prolog i učitali fajlu imena 'pera.deci'
Tada sa
?-listing, ?-listing(a), ?-listing(a/2), ?-listing([a/3,b/2])
se na ekran ispisuju
svi članci te fajle, svi a-članci, svi a(_,_)-članci, tj. u kojima a ima arnost 2, odnosno svi a(_,_,_)- i svi b(_,_)-članci.

l s t(<dato>)
Smisao: <dato> je lista (ili string).

m a r k
Jedan od grafičkih predikata. Videti Zadatak 11.20.

m d i c t
Videti u tački 10 deo oko formule (10.5).

m k d i r
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

n a m e
O tom predikatu videti u delu 12.2, tj. opisu predikata Arity-prologa.

n l , n l(<ime_fajle>)
Predikat za ispisivanje novog reda na ekran, odnosno u zadanu fajlu.

n o d e b u g
Videti o debug-predikatu.

n o n v a r(<dato>)
nonvar(+)
Smisao: <dato> nije promenljiva ili jeste oblika promenljive koja već ima vrednost.

n o s p y
Videti o debug-predikatu

n o s p y a ll
Ukidaju se sve prethodno postavljene spy-instrukcije, tj. svi predikati

[8] Videti o predikatu =.. na početku.

[9] Listing "deluje" samo na dinamičke, a ne i statičke predikate. O pomenutim vrstama predikata videti deo 10.2 tačke 10).

gube status "špijuniranog". Videti o debug-predikatu.

n o t(<dato>)
Smisao: Prološka negacija formule <dato>.

n o t r a c e
Videti o trace-predikatu.

n u m b e r(<dato>)
Smisao: <dato> je broj. Recimo, tačne su formule number(2) i number(3.4).

n u m b e r v a r s(<izraz>,<pocetak>,<kraj>)
n u m b e r v a r s(+,+,-)
Neka je dat proloski izraz kao f(a,[B,C],B,E) u kome učestvuju tri promenljive B,C,E . Parametru <pocetak> recimo dajmo vrednost 7. Tada na pitanje nje

```
?-numbervars(f(a,[B,C],B,E),7,X).
```

odgovor će biti da, i uz to
Promenljive B,C,E će redom biti zamenjene ovim posebnim LPA-atomima
$VAR(7) (počinje od 7, jer drugi parametar je 7)
$VAR(8), odnosno $VAR(9)
Dopunski treci parametar X će dobiti vrednost 10. Naime,počeli smo od $VAR(7), imali smo ukupno tri promenljive, odnosno "potrošili" smo oznake $VAR(7), $VAR(8), $VAR(9) i X dobija vrednost za 1 više od poslednjeg VAR-argumenta, tj. X=1+9=10.
U opštem slučaju pri računu formule oblika
numbervars(izraz,pocet,X) (X mora biti promenljiva)
sve promenljive prvog argumenta redom se zamenjuju ovakvim atomima
$VAR(pocet), $VAR(pocet+1), itd.
Ako je recimo $VAR(k) poslednji tako upotrebljen atom ,onda X dobija vrednost k+1.

o n(<elem>,<lista>)
-o **n**(+,+), **o n**(-,+)
Smisao: <elem> je element liste <lista>

o n e(<formula>)
Predikat prvog rešenja date formule <formula>. Recimo, neka tekući program sadrži redom ove a-članke

```
a(33).   a(44).   a(2).
```

Tada,očigledno prvo X za koje važi formula a(X) je X=33. To se može dobiti primenom one predikata. Naime, u pitanju

```
?-one(a(X)).
```

X se vezuje za 33.

o p(<predn>,<tip>,<ime-izraz>)
Taj predikat služi za definisanje op-izraza, pomoću njegovog imena, tipa i broja "prednosti". Videti deo 4.3 u tački 4.

o p e n(<ime_fajle>)
o p e n(+)
Smisao: Otvoriti (već postojeću) fajlu datog imena . To otvaranje je i za čitanje i pisanje (read i write).

p s w i n d(<ime_prozora>,<red>,<kolona>)
Služi za postavljanje ili nalaženje pozicije dato prozora. Drugi i treci argument moraju biti brojevi 1,2,3,... ili promenljive.Primeri:
Pretpostavimo da smo ušli u LPA prolog. Tada pitanjem


```
?-crwind(joca,5,7,10,20).
```

se pravi prozor imena 'joca' sa levim gornjim uglom u "tački"[10] (5,7) a desnim donjim uglom u tački (5+10,7+20).Jednovremeno "se selimo" u taj prozor. Ako želimo da se vratimo u osnovni prozor("ceo ekran"),onda možemo dati pitanje ?-cuwind(&:).
Evo primera korišćenja pswind- predikata:

```
?-pswind(joca,X,Y).
```

X se vezuje za 5, Y za 7.

```
?-pswind(joca,6,9).
```

Prozor 'joca' se seli na nov položaj. Sada je levi gornji ugao u tački (6,9).

```
?-pswind(joca,X,10).
```

Sada se druga koordinata od 9 menja na 10, a prva ostaje 6.

U vezi sa crwind-predikatom dodajemo i ovo. Pored već korišćenog crwind-a sa 5 argumenata, postoje još dva : crwind/7 i crwind/8. Prva dva nova argumenta se odnose na boju okvira i unutrašnjosti prozora, a treci u slučaju crwind/8 predikata se odnosi na tzv. 'mode' prozora. Taj mode može biti
sadržaj prozora se ne vidi,
ne vidi se okvir prozora
(*) pri korišćenju listing-predikata ne obavlja se odmah čitavo "listanje" već se popuni prozor, i onda čeka korisnik da pretisne ENTER-tipku.
i još neki slični.
Primeri:
Sa ?-crwind(mile,0,0,10,50,48,137). se pravi prozor 'mile' čiji okvir je plav, unutrašnjost crvena a kursor žut.
Sa ?-crwind(proz,0,0,10,50,48,137,5)
se pravi prozor 'proz' isti kao prethodni sa dodatkom da mu je 'mode' 5, što znači da ima svojstvo (*).

p u t(<ascii>)
put(+)
To je predikat kojim se za zadan <ascii>, tj. neki ASCII-kod štampa odgovarajući znak, ako tom <ascii> odgovara neki znak. Ukoliko broju <asscii> ne odgovara neki znak, onda se ili ništa ne štampa ili se uradi nešto posebno.Recimo, sa ?-put(7) se dešava zviždanje (za kratko vreme).Dalji primeri:

```
?-put(65).
```

Na ekranu se štampa A.

```
?-put(0).
```

Na ekranu se ne pojavljuje ništa, a sama formula put(0) je tačna.

```
?-put(10) ili ?-put(13).
```

"Štampa" se nov red, tj. prelazi se u nov red.

p u t(<ime>,<ascii>)
put(+,+)
Predikat kao put-predikat (opisan iznad), ali ispis se obavlja u fajlu,ili prozor imena <ime>.Videti Zadatak 11.12.

r e a d(<proloski_izraz>)
To je osnovni predikat uopšte za čitanje (read, input) nekog prološkog izraza, koji se od strane korisnika zadaje kao niz jedan za drugim kucanih znakova, pri čemu read-predikat pojavu tačke smatra za kraj izraza. Uz to Prolog proverava ispravnost ukucanog podatka, i ukoliko on nije ispravan prekida se algoritam sa porukom o greški. Primer:
Ako na pitanje ?-read(X),write(X). kao podatak

---

[10] Znači, "po dubini" 5, a "po širini" 7.

damo[11] pera. onda će X biti vezano za atom pera, i na ekranu će se pojaviti ta reč;
damo f(1). X će biti vezano za f(1), što će se štampati na ekranu;
damo "pera". X će biti vezano za listu [112,101,114,97], čiji članovi su ASCII-kodovi redom za slova p,e,r,a , što je u skladu sa okolnošcu kako uopšte Prolozi tretiraju stringove. Inače, na ekranu će se pojaviti ta lista.
damo Z. X će biti vezano za promenljivu Z, tj. za određenu adresu, koja će se posle štampati na ekranu;
damo f(). Na ekranu će se pojaviti poruka o sintaksnoj grešci i prekid toka algoritma.

r e a d(<ime>,<prološki_izraz>)
Predikat sličan sa read (videti ispred), s tim što se ovim predikatom čita podatak iz neke fajle datog imena ili obavlja učitavanje u prozor datog imena. Podaci moraju biti sintaksno ispravni.
U slučaju fajli ako se procedurom vraćanje (backtracking) ponovo dođe do read-formule ,onda se po fajli nastavlja dalje čitanje. Primer:
Prepostavimo da smo već napravili fajlu imena 'jova' koja sadrži ove prološke članke

```
pera.   mile.    f(1).   g(1):-h(2).
```

Tada program[12]

```
ajde:-open('jova'),
      ispisi('jova').
ispisi(A):-repeat, read(A,Izraz),
                write(Izraz),tab(3), Izraz==end_of_file,
                                              close(A).
```

na pitanje ?-ajde. će na ekran ispisati pojedine sastavke (članke) fajle jova.

r e c o n s u l t(<ime_fajle>)
Sličan predikatu consult, s tim što "stare članke zamenjuje novim", u smislu:
Ako već u programu sa kojim radimo imamo recimo neke f / k članke, tj. članke imena f, arnosti k, onda ukoliko pri

```
?-reconsult('ime')
```

fajla 'ime' ima neke f / k članke onda se iz tekućeg programa izbacuju njegovi "stari" f/ k članci i dodaju se takvi članci iz fajle 'ime'.

r e n(<staro_ime>,<novo_ime>)
r e n(+,+)
Služi da se fajli datog imena promeni ime. Primer:
Pitanjem ?-ren('pera.txt','jova.abc').
se fajla 'pera.txt' preimenuje u fajlu 'jova.abc'.

r e p e a t
Osnovni predikat koji je tačan i koji je tačan i ukoliko se procedurom vraćanja (backtracking) dođe na njega. To sve važi zahvaljujući njegovoj definiciji:

---

[11] Mora se na kraju kucati znak tačke.

[12] Primetite kako je kombinacijom

```
repeat,..., Izraz==end_of_file.
```

obezbeđeno "putovanje do kraja fajle".


```
repeat.
repeat:-repeat.
```

r e t r a c t, r e t r a c t a l l, r e t r a c t x
r e t r a c t(<izgled_članka>),
r e t r a c t a l l(<izgled_članka>),
r e t r a c t a l l(<izgled_članka>,<koji>)
Predikati za brisanje članaka, ali se ne mogu koristiti u programima koji su napravljeni kao izvršni (tj. sa ekstezijom .com) o kojima se govori u tački 9, odnosno drugim rečima odnose se na dimačike predikate. Ime članka mora biti zadano.
Uslugom prvog predikata briše se prvi članak koji ima dati oblik[13], određen sa <izgled_članka>. Primer:
Pitanjem ?-retract(f(X,Y)). se briše (ako ga ima) prvi članak datog izgleda. U takve dolaze

```
f(2,3)., f(3,A).
```

ali ne recimo članak f(2,3):-g(6,8). Ako želimo baš njega da obrišemo onda to postižemo pitanjem

```
?-retract((f(2,3):-g(6,8)).
```

ili pitanjem

```
?-retract((f(X,Y):-Z)).
```

ali ako smo sigurni da je taj članak prvi koji je tog oblika.
Ostala dva retract-predikata deluju ovako:
Pitanjem ?-retractall(<izgled_članka>) se brišu svi članci datog izgleda, a pitanjem ?-retract(<izgled_članka>,<koji>) samo onaj među njima koji je po redu <koji>, gde <koji> je 1,2,3, i sl.

r m d i r(<ime>)
Njegovom uslugom može se obrisati (prazna) direktorija datog imena

s a v e(<ime_fajle>)
s a v e(<ime_fajle>,šta)
Videti Zadatak 11.8. Primeri za drugi oblik:

```
?-save(mile,a).    ?-save(jova,[a,b]).
```

Prvim pitanjem se pravi fajla mile.dec i u nju unose a-članci(tekućeg programa), a drugim se svi a-, i b-članci unose u novu fajlu jova.dec.
Istaknimo da je save-predikat u Arity-prologu prilično drukčiji. Takođe se save predikat koristi i za snimanje modula. Videti tačku 10, formulu (10.4).

s d i c t(<lista_sistem_imena>)
Na pitanje oblika

```
?-sdict(X).
```

X se vezuje za listu svih sistemskih imena (atoma) Prologa, gde recimo dolaze 'read','write','assert', tj. imena svih u Prolog ugrađenih predikata, a takođe imena kao &, &:, user i dr. Kratko, X se vezuje za listu svih Y za koje važi sys(Y).

s e e , s e e i n g, s e e n
Videti opise tih predikata u delu 12.2, tj. opisu predikata Arity-prologa.

s e e k
Videti Zadatak 11.13.

s e t o f

---

[13] "Oblićnost" je u stvari drugo ime za "ujednačiv" , gden se misli na ujednačavanje u smislu algoritma ujednačavanja(unifikacije).

Videti o tom predikatu u delu 12.2 , tj. opisu predikata Arity-prologa.

s k i p(<dat_znak>)
Predikat u vezi sa čitanjem (read) i preskakanjem znaka <dat_znak>. Primer:
Ako na pitanje ?-get0(X),skip('e'),get0(Y).
kucamo
abcdef.
onda X će biti vezano za 'a', dalje do 'e ' uključujući i njega biće "preskakanje" i na kraju Y će biti vezano sa 'f'.

s k i p(<ime>,<dati_znak>)
Sličan sa prethodnim skip-predikatom ali čitanje i preskakanje se odnosi na datu fajlu ili dati prozor.

s o r t(<lista>,<sređena_lista>)
sort(+,-), -sort(+,+)
Predikat sort služi za sređivanje (sortiranje) date liste; ponovljeni članovi se odbacuju. Primer:
?-sort([4,1,3,1],X). X se veže za [1,3,4].
?-sort([2,1],[1,2]. Odgovor je da (yes).

s o r t(<lista_od_lista>,<sređena>,<ključ>)
sort(+,-,+), -sort(+,+,+)
Ako je prvi argument oblika
[[a1,a2,...,ak],[b1,b2,...],...]
tj. lista od lista i ako je ključ recimo [2], onda se sređivanje obavlja u odnosu na druge članove podlisti [a1,a2,...], [b1,b2,...],... tj. u odnosu su na a2,b2,c2,..., tj. oni se srede i podliste "prihvate" to sređivanje.
Primer:
?-sort([[1,4,6,5],[0,2,55,7],[8,1,9,6]],X,[2]).
X se vezuje za [[8,1,9,6],[0,2,55,7],[1,4,6,5]]
Članovi prvog argumenta mogu biti i strukture, tj. izrazi oblika
f(a1,...,ak)
ali i sa njima radimo slično, zamišljajući ih kao liste
[f,a1,...,ak]
Primeri:
?-sort([starost(pera,45),starost(jova,23),starost(mile,17)],X,[3]).
X se vezuje za listu
[starost(mile,17),starost(jova,23),starost(pera,45)]
jer sređivan je ovaj niz 45,23,17 -trećih "članova" datih struktura.
?-sort([f(2,3),g(4,5),a(pera,b)],X,[1]).
X se vezuje za listu
[a(pera,b),f(2,3),g(4,5)]

s o r t(<lista_od_lista>,<sređena>,<ključ>,<pravac>)
sort(+,-,+,+), -sort(+,+,+,+)
Predikat kao prethodni, s tim da ima dodatni argument <pravac>, koji može biti 1,-1. Ako je 1 onda se sređivanje obavlja "normalno", tj. rastući, a ako je -1 obavlja se sređivanje u opadajućem redosledu.

s p i e d(<ime>)
Smisao: da li je predikatu imena <ime> dodeljen status "špijuniranja".
Videti o debug-predikatu.

s p y
Videti o debug-predikatu.

s t r i n g o f(<lista_char>,<atom>)
stringof(+,-), stringof(-,+), -stringof(+,+)
Smisao: Drugi argument je atom, a prvi lista njegovih slova (karaktera).
Primeri:
?-stringof(X,pera). X se vezuje za [p,e,r,a].
?-stringof([p,e,'2',k],X). X se vezuje za atom pe2k
?-stringof(X,'Pet'). X se vezuje za ['P',e,t].

s y s(<ime>)
Smisao: <ime> je ime neke sistemske reči(atoma) prologa. Videti i o sdict-predikatu.

t a b(<brojka>), t a b(<ime>,<brojka>)
Predikat tab služi za ispis <brojka> znakova "belina" ("razmaka"). U drugom slučaju taj ispis se odnosi na fajlu ili prozor zadanog imena.

t e l l(<ime_fajle>), t e l l i n g(<promen>), t o l d
Smisao: Ako je <ime_fajle> dato ime, pitanjem
?-tell(<ime_fajle>)
se fajla tog imena stvara i otvara za pisanje, i postaje tekuće "upisno sredstvo", što znači da recimo, write predikat ima dejstvo u tu fajlu, a ne na ekran. Pitanjem ?-telling(X). X se vezuje sa imenom fajle koja je tekuća upisna fajla ("upisno sredstvo"). Pitanjem ?-told. se zatvara fajla koja je prethodno bila sa tell otvorena. Videti Zadatak 11.11.
Primeri:
Ako odmah pri ulasku u LPA-prolog pitamo ?-telling(X). X će biti vezano za ime 'WND:' (tako se zove "ekranska fajla").
Ako pitamo
?-tell('pera.dec'),write('Danas'),telling(X),told.
onda prvo fajla imena 'pera.dec' se stvara i otvara u nju se upisuje reč Danas, dalje zbog telling(X), X se vezuje za ime 'pera.dec' i na kraju, zbog told fajla 'pera.ari'se zatvara i izlazi se iz nje.

t e x t
Jedan od grafičkih predikata. Videti Zadatak 11.20.

t i m e /2 /3 /4
Predikati u vezi sa vremenom: časovi, minuti, sekunde, stotinke. Odnosi se na vreme koje daje časovnik računara.
Primeri:
Na pitanje ?-time(X,Y). moguć odgovor X=9,Y=23 sa smislom: 9 časova i 23 minuta
Na pitanje ?-time(X,Y,Z). moguć odgovor X=9,Y=23 kao prethodno, a Z=12, što znači 12 sekundi.
Da smo pitali ?-time(X,Y,Z,U) U bi bilo vezano sa broj stotinki(sekunde).

t o g r o u n d, t o h o l l o w
O ta dva predikata videti Napomenu 6.3 (tačke 6)

t r a c e / 0
Osnovni među tzv. "debuger"-predikatima, odnosno predikatima koji omogućuju praćenje toka programa, korak po korak. Primer:
Pretpostavimo da u okviru LPA-prologa smo učitali neki program i da onda postavimo pitanje ?-trace. što će prouzrokavati "buđenje", odnosno

sno uključivanje "debugera"[14]. Ako iza toga postavimo neko pitanje
?-ϕ
onda korak za korakom cemo videti razvoj razmatranja tog pitanja(videti i o debug-predikatu).
Ako hocemo da "se oslobodimo" dejstva trace-predikata onda možemo postaviti pitanje ?-notrace, tj. koristiti uslugu notrace-predikata.

t r u e
Ta "formula", po definiciji, je uvek tačna.

v a r(<dato>)
Smisao: Da li je <dato> promenljiva,koja još nije dobila vrednost ili koja je dobila, ali ona je neka druga promenljiva. Primeri:
Pitanje ?- X is 3,var(X). Odgovor je ne, jer X ima vrednost 3.
Pitanje ?- X=Y, var(X). Odgovor je da, jer vrednost X-a je promenljiva.

v i d e o /3, /4
Predikat u vezi sa bojama okvira prozora ,njegove unutrašnjosti, odnosno i tzv. 'mode' (videti o pswind-predikatu). Ti 'atributi' se ili mogu saznati ili po želji postaviti[15].
Primer:
Ako smo recimo predikatom crwind/5 napravili prozor 'jova' (videti o pswind-predikatu) i ušli u njega, tada pitanjem
?-video(jova,X,Y).
X,Y ce biti vezani za 0,0 ("crno-beli" slučaj) . Ali, ako zatim recimo pitamo
?-video(jova,3,7)
onda okvir postaje beo, a unutrašnjost plava. Ako se umesto tog postavi ovo pitanje
?-video(jova,3,7,5).
onda se dešava sve isto uz dodatak da je 5 'mode' tog prozora.

w d i c t(<prom>)
Na pitanje kao ?-wdict(X). X se vezuje za listu svih do tada otvorenih prozora.

w r i t e(<dato>)
Osnovni predikat za ispis (na ekran) argumenta <dato>. On mora da bude ispravan proloski izraz. Primeri:

?-write(pera).	Ispisace se reč (atom) pera.
?-write(X).	Ispisace se nešto kao _0084,što je u stvari pridružena adresa za X ispred koje stoji podcrtica.
?-write(pera(3)).	Ispisace se pera(3).
?-write("jova").	Ispisace se lista [106,111,118,97],jer Arity-prolog reč izmedu navodnica " " shvata kao listu ASCII kodova njenih slova (ne računajući navodnice).

w r i t e(<ime>,<dato>)
Taj predikat je sličan sa write-predikatom, prethodno opisanim, ali ispis se obavlja u fajlu ili prozor odreden datim imenom.

---

[14] To je ustvari dopunski program.

[15] Istićemo da se postavljanje atributa za neki prozor može postici tokom njegove gradnje i tada se koristi predikat crwind/7, odnosno crwind/8.

w r i t e d /1 /2
Predikat sličan prethodno opisanom,ali pri ispisu atoma koji neophodno sadrži znake navoda , kao u slučaju 'Pera', ispis obavlja i sa tim navodom (za razliku od write-predikata).
Primer: Uočimo ovaj program

```
ajde:-write('Daj ime '),read(Ime),
      see(Ime), uradi, seen.
uradi:-gread(Term), pokazi(Term).

pokazi(end_of_file):-!.
pokazi(Term):-write(' Sa znakom navoda  '),writeq(Term),nl,
              write(' I jos bez navoda  '),write(Term),nl, uradi.
```

i pretpostavimo da imamo fajlu imena 'jova' sa ovim sadržajem
pera. Mile. dragan.
Tada, ako na pitanje ?-ajde. damo 'jova' kao ime fajle na ekranu,usled korišcenja oba predikatai writeq i write, ce se u vezi sa fajlinim "sastavkom" Mile. pojaviti ova dva ispisa

'Mile'	(tako radi writeq)
Mile	(tako radi write)

L i t e r a t u r a

Prvo, pre navodenja savremenih knjiga ukazujemo na dodadaje koji su doveli do Prologa i bitno uticali na njegov razvoj

1965 g. J.A.Robinson: A Machine-Oriented Logic Based on the Resolution Principle , J. ACM 12, strane 23-41, Januar 1965
1972g. A. Colmerauer u Marselju izmišlja prvi Prolog .
1974g. R. Kowalski: Predicate Logic as a Programming Language, 1974
1977g. D.H.D. Warren, F. Pereira, L. Pereira pišu tzv. DEC-prolog.To je ubrzo postao standard Prologa.University of Edinburgh
1980. K.L.Clark, F.G.McCabe pišu Micro-prolog

Knjige i članci
R.A. Kowalski: Logic for Problem Solving, Elsevier North-Holland,1979
D.H.Warren: Prolog on DEC System 10, University of Edinburgh, 1979
W.F.Clocksin, C.S.Mellish: Programming in Prolog,Spinger Verlag, New York, 1981
K.L.Clark, F.G.McCabe: micro-PROLOG PROGRAMMING in LOGIC, Prentice Hall International, 1984
Gh.J.Hogger: Introduction to Logic Programming, Acad. Press, 1984
A. Colmerauer: Prolog in 10 figures, CACM,vol. 28, no. 12,dec. 1985
F. Giannesini,H. Kanoui, R. Pasero, M. van Caneghem: Prolog, Addison-Wesley Publ. Company, 1986
L.Sterling, E.Shapiro: The Art of Prolog - Advanced Programming Techniques, MIT Press, Cambridge,...,London, 1986

Časopisi
Artificial Inteligence, North-Holland
The Journal of Logic Programming, North-Hollana
New Generation Computing, Springer Verlag

I n d e x

Napomena: Ovaj indeks, tj. ukaznik na glavnije pojmove prisutne u knjizi je nešto kraći, jer u njemu se ne nalaze imena u Prolog ugradenih predikata, kao što su assert, write, PP, read, LOAD, save i dr. Njihovi opisi se redom izlažu u tački 12.