Prologi kursus lingvistidele *

Tõnu Tamme
TÜ Arvutiteaduse instituut

17. veebruar 2003. a.

1 Prologi programm kui teadmiste baas

Faktid, reeglid ja päringud. Sugupuu kirjeldamine. Inimestevaheliste mõistete defineerimine.

1.1 Sissejuhatus

Programmeerimiskeel Prolog (*Programmation et logique, Programming in logic*) loodi 1970-ndate aastate alguses Prantsusmaal Marseille's lingvistilistel eesmärkidel. Alain Colmerauer ja Philippe Roussel konstrueerisid esimest järku loogikal põhineva järeldusreeglite keele, mille abil oli lihtne kirjeldada erinevaid grammatikaid, kasutades oskuslikult ära automaatse teoreemitõestamise valdkonnas selleks ajaks saadud kogemused. Tänu David Warreni efektiivsele Prologi realisatsioonile Edinburghis 1977. a. sai Prolog peagi tuntuks hea vahendina intellektitehnika probleemide lahendamiseks.

Oluliseks seigaks Prologi ajaloos loetakse 1981. a. oktoobris Jaapanis välja kuulutatud viienda põlvkonna arvuti loomise projekti, sest uute arvutite baaskeeleks valiti Prolog.

Prologi süntaksiks on esimest järku loogika alamkeel — Horni loogika, milles atomaarsed predikaadid seotakse komadega ja tagurpidi implikatsioonidega: $\leftarrow$ või $:-$. Lisatud on ka teised loogikatehted: disjunktsioon

*Kursus valmis programmi HESP (*Higher Education Support Program*) toetusel 27. augustil 1998. a.

Implikatiivne kuju	Konstruktsioon	Prologi kuju
$p \leftarrow$	fakt	$p.$
$p \leftarrow q_1, \ldots, q_n$	reegel	$p{:}{-}q_1, \ldots, q_n.$
$\leftarrow p_1, \ldots, p_n$	päring	$?{-}p_1, \ldots, p_n.$
$\leftarrow$	vastuolu	*fail*

Tabel 1: Prologi põhikonstruktsioonid.

ja eitus. Sellegipoolest on Eituse realiseerimine osutunud tõsiseks problee-

Loogikatehe	Tehtemärk	Prologi kuju
konjunktsioon	&	,
disjunktsioon	v	;
implikatsioon	$\rightarrow$	$p{-}{>}q; true$
		$q{:}{-}p$
eitus	$\neg$	*not*
		\+ (ISO standard)

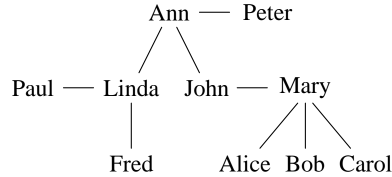
Tabel 2: Loogikatehted Prologis.

miks, sest Horni disjunktid $p \leftarrow q_1, \ldots, q_n$ ei sisalda standardkujul negatiivseid aatomeid — positiivsed literaalid on noolest vasakul ja negatiivsed literaalid on viidud noolest paremale. Puuduvad ka kvantorid ja ekvivalents — neid saab Horni disjunktide abil väljendada vaid kaudselt.

1.2 Andmebaas

Kirjeldame Prologi abil inimestevahelised seosed. Olgu näiteks Mary abielus Johniga, Linda abielus Pauliga ja Ann Peteriga. Olgu veel Ann Johni ja Linda ema, Mary lasteks olgu Alice, Bob ja Carol ning Fred olgu Linda poeg. Teame ka seda, kes on meessoost (*male*) ja kes on naissoost (*female*).

```
married(mary,john).
married(linda,paul).
married(ann,peter).
```



Joonis 1: Inimestevahelised seosed.

```
mother(john,ann).      mother(linda,ann).
mother(alice,mary).    mother(bob,mary).    mother(carol,mary).
mother(fred,linda).

female(mary).   female(alice).   female(carol).
female(ann).    female(linda).
male(john).     male(bob).       male(peter).
male(paul).     male(fred).
```

Toodud lihtne Prologi programm koosneb ainult faktidest. Paneme tähele, et iga fakt peab kindlasti lõppema punktiga. Nüüd saame esitada päringuid. Kas Mary on abielus?

```
?-married(mary,_).
yes
```

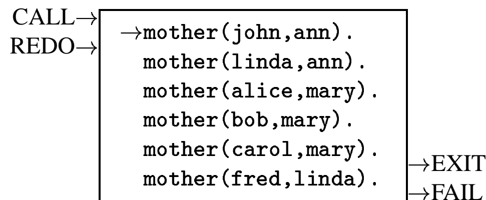
Vastus `yes` ütleb, et Prologil õnnestus unifitseerida predikaat *married(mary,_)* mingi predikaadiga oma sisemises andmebaasis. Alakriips '`_`' tähistab siin argumenti, mille väärtus meid ei huvita — me soovime lihtsalt teada, kas Mary on abielus või ei ole; alakriipsu kutsutakse nimetuks muutujaks (*anonymous variable*). Muutujaid eristab nimedest e. aatomitest see, et nad algavad suure tähega. Muutujate või aatomite esimesele järgnevateks sümboliteks võivad olla nii suurtähed ja väiketähed kui ka numbrid. Niisiis loetakse suurtähtede hulka erandina ka alakriips, mis muutuja esitähena kiirendab andmebaasist otsimist — kaob vajadus vastuste väljastamiseks.

Küsime nüüd: kes on Mary lapsed? Prolog unifitseerib esmalt muutuja `Child` nimega Alice ja jääb ootama. Kuna me tahame teada ka teisi Mary lapsi, siis trükime semikooloni ning saame uueks vastuseks Bob; Mary kolmandaks lapseks on Carol; lõpuks saame teate `no`, s.t. antud päringul ei ole rohkem vastuseid.

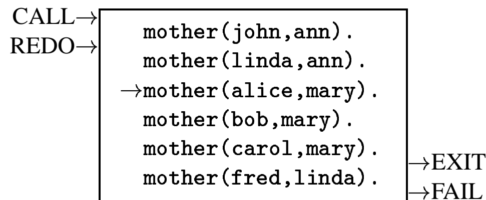
```
?-mother(Child,mary).
```

```
Child = alice;
Child = bob;
Child = carol;
no
```

Kuidas see tulemus saadi? Prologi töö kirjeldamiseks kasutatakse Lawrence Byrdi kastimudelit (*Byrd's box model*), milles predikaate kujutatakse neid defineerivaid fakte ja reegleid sisaldavate kastidena. Igal kastil on kaks sisendit — CALL ja REDO — ning kaks väljundit — EXIT ja FAIL. Esmakordsel kasti sisenemisel (CALL) tõstetakse selle sisemine viit esimesele predikaadi definitsioonile ja püütakse seda päringuga unifitseerida, s.t. muuta neid muutujate asendamise teel võrdseks.



Kastis liigutakse niikaua allapoole, kuni unifitseerimine õnnestub ja väljutakse EXIT pordi kaudu. Praegu saime vastuseks Alice:



Vajutades semikoolonile, soovime saada uusi lahendusi. Kuna kasti sisemine viit jäeti väljumisel kolmandale faktile, siis siseneme samasse kasti uuesti (REDO) ja jätkame allapoole liikumist, saades lahendid Bob ja Carol. Lahendite lõppemisel väljutakse kastist läbi FAIL pordi.

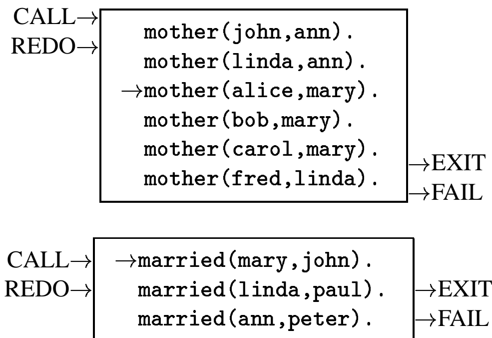
Aga kes on Alice'i isa? Ilmselt inimene, kes on tema emaga abielus.

```
?-mother(alice,_M), married(_M,Father).
Father = john
```

Hästi, aga kuidas Prolog töötleb kahte järjestikust päringut? Seda on mõistlik üles kirjutada Prologi enda keeles — Horni disjunktidena:

```
?-mother(alice,_M), married(_M,Father). % esialgne päring
{_M=mary}                                - muutuja unifitseerimine
?-married(mary,Father).                  % uus päring
{Father=john}                            - unifitseerimine
?-                                       % tühipäring
Father = john                            - arvutatud vastus
```

Kõigepealt leitakse vasakpoolse päringu vastus: Alice'i ema on Mary. Seejärel leitakse Mary abikaasa, kelleks osutub John, ning väljastatakse arvutatud vastus. Kastimudeli keeles tähendab see minekut esimese päringu kasti EXIT'ist teise päringu kasti sisendisse:



Lahendusprotsessi saab mingil määral jälgida käsuga `trace`. SWI-Prologis näeb dialoog välja järgmiselt:

```
?-trace.
?-mother(alice,_M), married(_M,Father).
Call:  (  7) mother(alice, _L119) ? creep
Exit:  (  7) mother(alice, mary) ? creep
Call:  (  7) married(mary, _G289) ? creep
Exit:  (  7) married(mary, john) ? creep
Father = john ;
no
```

Kuid ESL-Prolog annab tulemuseks hoopis

```
(1) 0 CALL: mother(alice,_412)?
(1) 0 EXIT: mother(alice,mary)
(2) 0 CALL: married(mary,_502)?
(2) 0 EXIT: married(mary,john)
```

```
_M = mary
Father = john
More (y/n)? y
(2) 0 REDO: married(mary,john)?
(2) 0 FAIL: married(mary,_502)
(1) 0 REDO: mother(alice,mary)?
(1) 0 FAIL: mother(alice,_412)
no
```

ESL-Prolog arvab, et peale Johni võib Maryl olla ka teisi abikaasasid, SWI-Prolog on aga veendunud, et rohkem abikaasasid ei ole, sest ta arvutab predikaadi *married* indeksi esimese argumendi järgi ja näeb sellest, et Maryl ei ole peale Johni ühtegi abikaasat.

1.3 Prologi kasutamine

Enne konkreetse Prologi süsteemiga tööle asumist on hea tunda Prologi lihtsamaid juhtkäske. Paneme tähele, et kõik käsud peavad lõppema punktiga — muidu ei võta süsteem meid üldse jutule.

Erinevalt teistest programmerimiskeeltest loeb Prolog kõik programmid mällu, mistõttu Prologis programmeerimine on omapärane interaktiivne dialoog. Süsteemi piisaval tundmisel tekib programmi kasutajal mõnus peremehetunne.

1.4 Teadmiste esitamine

Teadmised ei ole mitte ainult faktid, vaid ka teatud hulk mõisteid ja nende vahelisi seoseid. Eespool leidsime kahe päringuga Alice'i isa — kui tema ema abikaasa. Niisugune kompleksne päring on mõttekas vormistada eraldi predikaadina, antud juhul isa mõistet defineeriva reegliga, nõudes lisaks, et ema abikaasa oleks meessoost:

```
father(Child,Father):-
    mother(Child,Mother),
    married(Mother,Father),
    male(Father).
```

Paneme tähele, et sõnal "isa" on keeles kaks tähendust — seos olla "kellegi isa" ja omadus olla lihtsalt isa, täpsustamata isa ja lapse vahelist seost.

```
father(Father):-father(_Child,Father).
```

Prologi käsk	Selgitus
`?-[user].`	programmi sisestamine klaviatuurilt
...	
ˆD	(Ctrl-D) lõpetab sisestamise
`?-halt.`	Prologist väljumine
`?-X is 1059*513/100.`	aritmeetilise avaldise arvutamine
`?-name(atom,List).`	aatomi teisendamine sümbolite listiks
`?-[foo].`	programmi foo/foo.pl sisselugemine
`?-edit(foo).`	programmi redigeerimine (SWI-Prologis)
`?-listing.`	defi neeritud predikaatide kirjelduste väljastamine
`?-listing(member).`	predikaadi *member* kirjeldus
`?-listing(member/2).`	predikaadi *member*/2 kirjeldus
`?-trace.`	programmi täitmise jälgimine
`?-notrace.`	jälgimise lõpetamine
`?-spy(married/2).`	kahekohalise predikaadi *married* jälgimine
`?-nodebug.`	silumise lõpetamine
`?-apropos(list).`	kogu informatsioon sõna "list" kohta (SWI-Prologis)
`?-help.`	abiteave
`?-help(member).`	abiteave predikaadi *member* kohta
`?-sort([1,...,10],L).`	listi $[1, \ldots, 10]$ sorteerimine
`?-statistics.`	Prologi tööstatistika

Tabel 3: Prologi juhtkäsud.

Predikaatidele ja nende argumentidele on soovitav anda mnemoonilised nimed, hõlbustamaks programmide lugemist. Lapsele viitav muutuja `Child` esineb isa omaduse reeglis *father*/1 ainult üks kord — ta on reeglis ühekordne muutuja (*singleton variable*). Prologi kompilaator annab sellisel juhul hoiatuse, sest suure tõenäosusega on tegemist trükiveaga — näiteks `Father`, `father` ja `FahTer` on erinevad muutujad. Aga kui muutuja on tõepoolest ühekordne, nagu omaduse "olla isa" definitsioonis, tuleb seda alustada kas alakriipsuga või piirdudagi ainult nimetu muutujaga '`_`'.

Ülesanne 1 Esitage oma sugupuu Prologi teadmistebaasina (vend, õde, tädi, onu, vanaisa, vanaema).

1.5 Taust

Käesolev õppematerjal on mõeldud Prologi loengute lugemiseks, aga ka iseõppimiseks kõigile programmeerimishuvilistele, kujutades endast kontsentraati Tartu Ülikooli arvutiteaduse instituudis loetud aine "Loogiline programmeerimine" loengutest aastatel 1993, 1995 [66] ja 1997 [67], samuti ainete "Programmeerimislabor: Prolog" (1994) ja "Programmeerimiskeeled" (1997 [68], 1998) loengutest ja materjalidest. Käesoleva konspekti autor loeb loogilise programmeerimise kursust alates 1993. a. kevadest. Enne seda luges seda Tartu Ülikoolis Tanel Tammet. Et kursus on mõeldud lingvistidele, siis pööratakse selles võrreldes tavalise Prologi kursusega rohkem tähelepanu tekstitöötlusele ja grammatikate realiseerimisele.

Suuremateks kursuse mõjutajateks võib lugeda külalislektori Michael Hanusi (Saarbrücken) loengutsüklit Tartu Ülikoolis 1994. a., aga ka Chris Hoggeri [33] ja Richard O'Keefe [54] raamatuid.

Kuidas siis nende materjalide järgi õppida? Loogiline programmeerimine, s.h. Prolog on kiiresti arenev valdkond. Tihti puudub valmis retsept mingi probleemi lahendamiseks. Seepärast on kasulik proovida lahendada konspektis toodud ülesandeid iseseisvalt, enne kui lugeda lahendusi. Loogilised programmid on probleemide kontsentreeritud esitused predikaatarvutuses. Lahenduse leidmine ja struktuuride valik jääb valmis programmi puhul tihti varjatuks. Sageli osutub kompaktne loogiline esitus ka efektiivseks. Samas tuleb arvestada Prologi programmide kompileerimise ja interpreteerimise iseärasustega. Neid küsimusi valgustab põhjalikult raamatus [54] Richard O'Keefe, kes on arendanud mitut Prologi süsteemi. Seetõttu võib teda momendil pidada üheks paremaks Prologi siseelu asjatundjaks.

Niisiis Teie ülesandeks on loengumaterjalidest arusaamine. Põhirõhk pange kindlasti ülesannete lahendamisele, püüdke iseseisvalt ilma konspektita uuesti realiseerida loengus esitatud programmid. Leidke seejärel võimalusi nende täiustamiseks, testige programmide kiirust ja kasutamismugavust. Prologis programmeerimine erineb oluliselt programmide kirjutamisest teistes keeltes. Richard O'Keefe nendib [54], et teistes keelte programmeerides kirjutab ta detailseid instruktsioone, Prologis kirjeldab ta aga objektide vahelisi seoseid — on selline tunne nagu häälestaks mingit süsteemi. Eriti oluline on õigete andmestruktuuride valik: see lihtsustab programmide kirjutamist, muutes samal ajal nende töö efektiivsemaks. Hea on olla kursis ka programmide kirjutamise tehnoloogiaga — algoritm, mis on efektiivne teistes keeltes, on tavaliselt efektiivne ka Prologis.

Materjalid on järjestatud selliselt, et nende järgi saaks lugeda kaheksa loengut, kusjuures iga loeng eeldab ka praktiliste ülesannete lahendamist — kas enne või pärast loengut. Praktika näitab, et ülesannete lahendamine võtab tunduvalt rohkem aega kui loengus istumine või raamatute lugemine. Seepärast ei piisa Prologi omandamiseks ainult praktikumidest, vaid tuleb teha ka palju kodutööd.

Jõudu tööle!

Ajalugu

Võib öelda, et Prolog kui programmeerimiskeel sündis katsetest rakendada automaatse teoreemitõestamise (*automatic theorem proving*) praktilisi ja teoreetilisi tulemusi intellektitehnika probleemide lahendamiseks. Oluliseks tõukejõuks sai resolutsioonimeetodi kirjeldamine Alan Robinsoni poolt 1965. aastal [59]. Tänu sisse ehitatud variantide läbivaatusele (*backtracking*) ja sümbolteisendustele (*unification*) sobis Prolog lingvistikaprobleemide lahendamiseks peaaegu ideaalselt. Nii sündiski aastatel 1970–1973 Marseille Prolog. Seda on aastate jooksul täiendatud tehisintellekti probleemide efektiivseks lahendamiseks tarvilike vahenditega. Lisati näiteks lõpmatud termid ja erinevatel lõplikel ja lõpmatutel objektihulkadel töötavad kitsendustega programmeerimise meetodid. Marseille Prologist on välja arendatud keeled Prolog II–IV.

Samaaegselt praktiliste probleemide lahendamisega arendati ka Prologi teooriat. Robert Kowalski näitas, et Horni disjunkte $p \leftarrow q_1, \ldots, q_n$ saab lugeda nii *deklaratiivselt*:

> Kui predikaadid $q_1, \ldots, q_n$ on tõesed, siis on tõene ka predikaat p.

kui ka *protseduraalselt*:

> Programmi p täitmiseks tuleb täita tema alamprogrammid $q_1, \ldots, q_n$.

Keith Clark jt. on teinud palju tööd Prologi eituse teooria kirjeldamiseks. Uuritud on ka Prologi laiendusi kvantoritega ja tüübiteooria vahenditega (näiteks Gödeli projekt Inglismaal Bristolis [31] ja Mercury projekt Austraalias [49]).

Praktilise programmeerimiskeelena sai Prolog tuule tiibadesse alles Šotimaal Edinburghis tehtud pingutuste tulemusena. David Warren jt. realiseerisid 1977. a. oma Prologi versiooni ja vastava kompilaatori, muutes Prologi peaaegu sama efektiivseks kui teised intellektitehnikas kasutatavad keeled, näiteks Lisp. Arvutil VAX realiseeritud keel sai tuntuks Edinburghi Prologina (Prolog-10) ja levis üle maailma. William Clocksin ja Chris Mellish kirjutasid 1981. a. esimese Prologi õpiku [14]. David Warren tutvustas 1982. a. ka uudset transleerimismeetodit, kus Prologi programm teisendatakse teatud virtuaalseks masinaks — Warreni abstraktseks masinaks (WAM) —, mida on hiljem lihtsam interpreteerida kui esialgset Prologi programmi. Prologi kommertssüsteemides (näiteks Sicstus ja Quintus Prologis [62], LPA Prologis [45], ALS Prologis [2], Arity Prologis, Amzi Prologis [3] ja Visual Prologis [73]) transleeritakse Prolog siiski kas programmeerimiskeelde C või masinkoodi, saades Warreni abstraktsest masinast umbes kolm korda kiirema programmi.

Suure tõuke Prologi arenguks andis Jaapanis 1981. a. oktoobris välja kuulutatud viienda põlvkonna arvuti loomise projekt [22], mille baaskeeleks valiti Prolog. Arvati, et tänu keeles algselt sisalduvale mittedetermineeritusele — lubatakse kirjeldada predikaadi mitu alternatiivset varianti — sobib Prolog mitmel protsessoril (või mitmel üheaegselt töötaval arvutil) paralleelselt töötavate programmide realiseerimiseks paremini kui teised keeled. Nüüdseks on konstrueeritud terve hulk paralleelsust toetavaid Prologi versioone.

Kirjandus

Hea ülevaate Prologi ajaloost saab Alain Colmeraueri ja Philippe Rousseli artiklist [16]. Tasub ka lugeda Robert Kowalski artikleid [39], [70], [40], [42], [34] ja raamatut [41]. Warreni abstraktse masina kohta võib lugeda Hassan Ait-Kaci [1] ja David Maieri ja David Scott Warreni [46] raamatust, samuti Peter van Roy artiklist [72], kus antakse põhjalik ülevaade Prologi arengust ning vaagitakse tema tulevikuperspektiive.

Prologi õpikutest soovitatakse lugeda William Clocksini ja Chris Mellishi raamatut [14], millest on juba ilmunud neli trükki. Hea on ka Prolog-10 manuaal [9]. Matemaatikute ja programmeerijate jaoks sobib paremini Sterlingi ja Shapiro raamat [64], samuti Nilssoni ja Maluszinski [53], Ivan Bratko [10], John Malpase [47] ja Michael Covingtoni jt. [19] õpikud. Sterlingi-Shapiro ja Ivan Bratko raamatutes käsitletakse lisaks põhjalikult

intellektitehnika probleemide lahendamist Prologis. Tasub lugeda ka Prologi ISO standardit [36].

Edasijõudnutele võib soovitada Richard O'Keefe monograafiat [54]. Prologi lingvistilistest rakendustest saab ülevaate Michael Covingtoni raamatust [18].

Loogilise programmeerimise teooriat käsitletakse John Lloydi [43], Chris Hoggeri [32], David Maieri ja David Scott Warreni [46] ja Krzysztof Apti [6] raamatutes, samuti Apti artiklis [5]. Huvitav on tutvuda Prologi ja predikaatloogika vaheliste seostega. Sellise vaatenurga alt on Prologi käsitletud Steve Reevesi ja Michael Clarki [58], Richard Spencer-Smithi [63], Chris Hoggeri [33] ja Peter Gibbinsi [24] õpikutes.

Automaatset teoreemitõestamise ja resolutsioonimeetodi kohta võib lugeda Changi ja Lee [13], Uwe Schöningi [61] ja Melvin Fittingi [23] raamatutest.

Tingimustega loogilist programmeerimist käsitletakse ajakirja "Byte" 1987. a. augustinumbris [11] ning Pascal van Hentenrycki [71] ja Kim Marriotti ja Peter Stuckey [48] raamatutes.

Kindlasti võib soovitada Dennis Merritti artiklit "Adventure in Prolog" [51], milles räägitakse Prologi eelistest mängude realiseerimisel. Artikkel on kaasas Amzi Prologiga. Tasub lugeda ka David Hoveli artiklit [35] Prologi kasutamisest operatsioonisüsteemis Windows NT. Ära ei tohi unustada ka materjale veebis: uudistegruppi news:comp.lang.prolog ja selle Korduvalt Küsitud Küsimuste faili [4]. Huvitavaid materjale võib leida Loogilise Programmeerimise Assotsiatsiooni (*ALP*) leheküljelt [7] ning ajakirjadest Logic Programming Newsletter, The Journal of Logic Programming ja New Generation Computing. Suurepärane on ka PC AI Magazine'i Prologi lehekülg [56].

Eestikeelseid materjale on kirjutanud Jaak Henno [27], [30] (kahjuks ei olnud veebimaterjalid kursuse koostamise ajal kättesaadavad) ja Tanel Tammet [69]. Jaak Hennolt on ilmunud ka soomekeelne õpik [28] ja raamat formaalsetest keeltest [29]. Mare Koit on kirjutanud resolutsioonimeetodit käsitleva õppematerjali [38].

Prologisüsteemid

Prologisüsteemid jaotatakse vabatarkvaraks (*freeware*) ja kommertssüsteemideks. Kuna Prolog arenes välja akadeemilistes ringkondades, siis on paljud vanemad Prologisüsteemid vabalt kätte saadavad. Vaba on näiteks Tartu

Ülikooli serveritele installeeritud SB-Prolog (Stony-Brooks Prolog). Vabad on ka C-Prolog ja Jan Wielemakeri SWI-Prolog [65]. Niisugused süsteemid kompileerivad tavaliselt programmi Warreni abstraktseks masinaks ja ei anna omaette täidetavat faili. Paljud Prologisüsteemid töötavad nii MS Windowsi keskkonnas kui ka mitmesugustes Unixi versioonides, näiteks Linuxi ja Solarise all. Ent on olemas ka MS Dosi all töötavaid versioone: SB-Prolog [60], ESL-Prolog [21], Quintus Prolog, Turbo Prolog jt. Viimane saavutas Borlandi firma kaudu Prologidest suurima populaarsuse. Nüüdseks on Borland Turbo Prologi toetamisest loobunud ja on andnud selle tagasi Taani emafirmale Prolog Development Center, kes müüb seda PDC Prologi nime all. Tuntud on ka selle Windowsi versioon Visual Prolog [73]. Kahjuks on Turbo Prolog ja tema sugulaskeeled ehitatud üles rangele tüübisüsteemile, mis muudab nende õppimise teistest Prologidest vaevalisemaks. Lisaks Borlandile on Prologi arendanud ka IBM: tuntud on tema suurarvutitel kasutatud keel IBM Prolog.

Akadeemilistest süsteemidest on laialdaselt levinud firma ICL poolt toetatav tingimustega loogilise programmeerimise süsteem Eclipse [20] ja Rootsis arendatav Sicstus Prolog [62]. Nende süsteemide kasutamiseks tuleb sõlmida litsentsileping, mis ei ole õppeasutuste jaoks eriti kallis või on päris tasuta (Eclipse). Mõlemad süsteemid töötavad nii Windows 95-s kui ka paljudes Unixi versioonides. Sicstus Prolog on tuntud oma kiiruse poolest — tema Solarise versioon võimaldab kompileerida Prologi programmi otse masinkoodi. Rootsi riigi poolt toetatav instituut SICS müüb ka Quintus Prologi MS Dos-ile.

Tingimustega loogilise programmeerimise süsteemidest on lisaks Eclipse'ile ja Sicstus Prologile tuntud keeled CLP(R) ja Prolog III.

MS Windowsi kommertssüsteemidest pakutakse lisaks Visual Prologile ka Amzi Prologi [3] ja LPA Prologi [45], mis vastavad paremini Prologi standarditele (Edinburghi Prolog ja ISO standard [36]). Kommertssüsteemidel on kaasas teegid (*library*) Prologi ühendamiseks teiste C-programmidega ja veebiga, samuti Windowsi graafiliste võimaluste kasutamiseks. Nad võimaldavad ka kompileerida Prologi programmi omaette täidetavaks failiks (*runtime application*). Paul Tarau on lisanud BinPrologile [8] toeka veebiliidese. NB! Amzi, LPA ja Visual pakuvad kuuajaliseks või pikemaks testimiseks välja oma täisfunktsionaalsed *time-locked* Prologi versioonid.

Tartu Ülikooli serveritele on installeeritud lisaks SB-Prologile ja Eclipse'ile eksperimentaalne loogilise programmeerimise süsteem Gödel [25].

Viimasel ajal on jõudsalt arenenud Austraalias välja töötatud rangel tüübisüsteemil baseeruv loogilise programmeerimise süsteem Mercury [49], milles püütakse sarnaselt Gödeliga suurendada Prologi deklaratiivsust, tõstes samaaegselt programmide efektiivsust.

2 Andmete kirjeldamine

Rekursiivsed kirjeldused ja termistruktuurid.

2.1 Kasutamise nipid

Sõltuvalt kasutatavast Prologist ei tarvitse eespool toodud käsud alati töötada. Näiteks Eclipse'is ei anna käsk

```
?-listing(father).
```

soovitud tulemust — predikaadi *father* kirjeldust — sest programm kompileeritakse Warreni abstraktse masina (WAM) käskudeks. Programmi teksti aitab Eclipse'is säilitada käsk

```
:-set_flag(all_dynamic,on).
```

mis ütleb, et Prologi programm kujutab endast dünaamilist andmebaasi, mida on võimalik täitmise käigus muuta. Alternatiivina võib dünaamilisena kirjeldada iga predikaadi eraldi.

```
:-dynamic father/2, father/1.
```

Dünaamiliste predikaatide täitmise efektiivsus sõltub suuresti kasutatavast Prologist, kuid üldreeglina on nende täitmine aeglasem — neid tuleb ju täiendavalt interpreteerida. SWI-Prologis, mis transleerib samuti programmi WAMi käskudeks, töötab aga käsk `listing` juba vaikimisi, ilma globaalsete parameetrite sättimiseta.

Mary laste teadasaamiseks esitasime päringu

```
?-mother(Child,mary).
```

ja vajutasime iga vastuse korral semikoolonile. Kuidas sellest tüütust tegevusest lahti saada? Tuletame meelde, kuidas Prolog otsib lahendit. Ta liigub senikaua mööda predikaatide "kaste" allapoole, kuni kõik päringud on saanud positiivse vastuse. Lahendus tekib unifitseerimise käigus kõrvalproduktina. Kasutame seda ära ja väljastame leitud lahendused `write` käsuga:

```
?-mother(Child,mary), write('Mary laps on '),
  write(Child), nl, fail.
```

Sellist konstruktsiooni nimetatakse *fail*-tsükliks. Käsk `fail` on vastand käsule `true` — kui viimane on alati tõene, siis `fail` kinnitab, et vaadeldaval predikaatide kogumil ei ole lahendit ja käsib nihutada eelnevate predikaatide kastide viitasid allapoole. Nii saame üksteise järel kõik esimese predikaadi lahendid, väljastame need ja käsime edasi otsida:

```
Mary laps on alice
Mary laps on bob
Mary laps on carol
no
```

Käsk `nl` (*new line*) palub trükkida lahendid eraldi ridadele, et neid oleks lihtsam lugeda.

Loomulikult oleks hea kirjutada nimed suurte tähtedega: Mary, Alice, Bob, Carol jne. Prologis on ka selle peale mõeldud. Kui suurtähega algavad sõnad tähistavad tavaliselt muutujaid, siis apostroofides sõnu käsitletakse aatomitena Näiteks võib küsida

```
?-mother('Alice','Mary').
```

Kuna Prolog eristab suur- ja väiketähti, tuleb nüüd parandada ka meie andmebaasi faktid.

2.2 Prologi plussid

Püüame kokku võtta, milleks Prolog hea on. Millised on tema rakendused?

- Intellektitehnika. Prolog on kujunenud intellektitehnika ja tehisintellekti programmeerimiskeeleks. Prologi õpikutes lahendatakse intellektitehnika probleeme ja vastupidi: intellektitehnika õpikud kasutavad baaskeelena keelt Prolog kui esimest järku loogika efektiivset nurgakest.

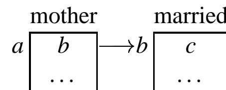
- Andmebaasid. Prologis on lihtne esitada andmeid ja seoseid nende vahel. Samuti on Prologi sisse ehitatud andmebaasist otsimine — automaatse teoreemitõestamise meetoditel põhinev päringutele vastamise mehhanism.

  Prologi fakte võib ette kujutada tabelitena ja reegleid seostena tabelite vahel. Näiteks predikaadid *mother*/2 ja *married*/2 punktis 1.2 on esitatavad tabelitena.

	mother
john	ann
linda	ann
alice	mary
bob	mary
carol	mary
fred	linda

	married
mary	john
linda	paul
ann	peter

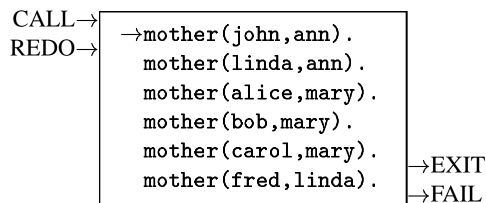
Isa predikaat on nüüd seos kahe tabeli vahel.



$$father(a,c){:-}mother(a,b), married(b,c).$$

Prologi on lihtne ühildada ka relatsioonilise ja objektorienteeritud andmebaasi mudeliga.

- Lingvistika. On teada, et Prolog loodigi lingvistilistel eesmärkidel. Ta on peaaegu ideaalne vahend generatiivsete grammatikate produktsioonide esitamiseks Horni loogikareeglitena. Prologi sisseehitatud Horni grammatikad võimaldavad säilitada grammatikate esialgset kuju, tõstes veelgi grammatikate kirjutamise tulemuslikkust.
- Paralleelsus. Seoses Jaapani viienda põlvkonna arvuti projektiga sai Prolog tuntuks paralleelsete algoritmide realiseerimise keelena. Paralleelsus on näiliselt Prologi juba sisse kodeeritud. Näiteks predikaadi *mother*/2 kastis võib piisava arvu protsessorite korral vaadata samaaegselt läbi mitut fakti või definitsiooni.



Pole ju tähtis, et me alustame kindlasti kasti esimesest definitsioonist ja liigume järkjärgult allapoole.

- Prototüüpimine. Prologi kasutatakse ka uute programmeerimiskeelte mudelite loomiseks ja nendega katsetamiseks. Keele mudelit täiustatakse, kuni kasutajad jäävad sellega rahule. Seejärel programmeeritakse keel uuesti, tavaliselt C's, arvestades klientide soove ja kehtivaid standardeid.

2.3 Sugulaste loetlemine

Asume nüüd sugupuu kirjeldamise põhieesmärgi kallale. Kes on meie sugulased? Kes on lähemad sugulased? Kuidas nad on meile sugulased? Küsimusi on palju. Püüame leida nendele vastused.

Sugulaste hulk on üldreeglina lõpmatu. Seepärast on mõistlik järjestada nad kauguse järgi: kõigepealt loetleme lähemad sugulased — vanemad, lapsed, vennad, õed ja abikaasa; seejärel loetleme varem loetletud sugulaste lähemad sugulased jne., liikudes ringide kaupa sammhaaval kaugemate sugulaste poole. Lõpmatut hulka saab defineerida ainult rekursiivselt —

```
% relative(N,Man,R) on tõene, kui
%         R on isiku Man kuni N taseme sugulane
relative(N,Man,R):-
    N>0,
    N1 is N-1,
    (parent(Man,R)
    ; child(Man,R)
    ; brother(Man,R)
    ; sister(Man,R)
    ; abikaasa(Man,R)
    ; relative(N1,X,R), relative(1,Man,X)).

parent(Man,R):-mother(Man,R).
parent(Man,R):-father(Man,R).

abikaasa(Man,R):-
    married(Man,R)
    ;
    married(R,Man).
```

Joonis 2: Programm sugulaste loetlemiseks.

predikaat *relative*/3 pöördub enda poole tagasi, genereerides järjest uusi

sugulasi. Nüüd saame teada, kes on Fredi kahe esimese ringi sugulased:

```
?-relative(2,fred,Kes).
Kes = linda;
Kes = paul;
Kes = ann;
Kes = peter;
Kes = paul;
Kes = linda;
no
```

Vastus on selline, sest defineeritud on ainult vanema ja abikaasa mõisted. Teised seosed on defineerimata: veateadete vältimiseks tuleks kompileerimisel öelda, et need predikaadid on dünaamilised.

2.4 Sugulaste märgendamine

Sugulase mõiste rekursiivne definitsioon annab meile hoopis suuremad võimalused — lisaks sugulaste loetlemisele saame väljastada, kuidas need härrased on meile sugulased. Selleks lisame mõistete definitsioonidele vastavad märgendid: ema, isa, abikaasa jne. Nüüd on sugulaste loetelu tunduvalt informatiivsem:

```
?-relative(2,fred,Kes,Def).
Kes = linda, Def = ema;
Kes = paul.  Def = isa;
Kes = ann,   Def = [ema,ema];
Kes = peter, Def = [ema,isa];
Kes = paul,  Def = [ema,abikaasa];
Kes = linda, Def = [isa,abikaasa];
no
```

Me kasutasime siin ära Prologi võimet konstrueerida rekursiooni käigus mitmesuguseid termistruktuure. Termide standardseks näiteks on listid, millest räägime täpsemalt järgmises punktis.

Ülesanne 2 Lõpetage oma sugulaste teadmistebaas.

2.5 Infikspredikaadid

Lisaks predikaatide esitusele prefikskujul lubab Prolog ka nende infiks- ja postfiksesitust. Näiteks fakti

```
% relative(N,Man,R,Def) on tõene, kui
%         R on isikuga Man kuni N taseme sugulusseoses Def
relative(N,Man,R,Def):-
    N>0,
    N1 is N-1,
    (parent(Man,R,Def)
    ; child(Man,R,Def)
    ; brother(Man,R,Def)
    ; sister(Man,R,Def)
    ; abikaasa(Man,R,Def)
    ; relative(N1,X,R,Def1),
    relative(1,Man,X,Def2), Def=[Def2,Def1]
    ).

parent(Man,R,ema):-mother(Man,R).
parent(Man,R,isa):-father(Man,R).

abikaasa(Man,R,abikaasa):-
    married(Man,R)
    ;
    married(R,Man).
```

Joonis 3: Täiustatud sugulaste loetlemisprogramm.

```
mother(john,ann).
```

asemel võib kirjutada

```
john ema_on ann.
```

Selleks defineerime *ema_on* kui infikspredikaadi.

```
:-op(900,xfx,ema_on).
```

Arv 900 on siin infiksfunktori *ema_on* prioriteet, mida arvestatakse programmi käskude süntaksianalüüsil. See peaks olema madalam operaatori `:-` prioriteedist, milleks on tavaliselt 1200. Määrang `xfx` ütleb, et funktor ei ole assotsiatiivne. Näiteks ei tohi kirjutada

```
fred ema_on linda ema_on ann.
```

s.t. Fredi ema on Linda ja Linda ema on Ann. Infikspredikaate on lubatud kirjutada nii definitsiooni kohaselt, argumentidest ümbritsetuna, kui ka prefikskujul.

```
ema_on(alice,mary).

?-X ema_on Y.
X = john, Y = ann ;
X = alice, Y = mary ;
no
```

Ülesanne 3 Kirjeldage postfi kspredikaat *on_mees*.

```
?-X on_mees.
X = john
```

Ülesanne 4 Tehke oma sugulaste andmebaas ringi, kasutades infi ks- ja postfi kspredikaate.

Lihtne on muuta ka olemasolevaid infiksoperaatoreid ja lisada uusi operaatoreid. Näiteks lausearvutuse tehted võib defineerida kujul

```
:-op(900,fy,'~'). %eitus
:-op(910,xfy,and).
:-op(920,xfy,or).
:-op(930,xfy,'->').
:-op(940,xfy,'<->').
```

Määratlused `fy` ja `xfy` ütlevad, et eitus ja binaarsed loogilised tehted (konjunktsioon, disjunktsioon, implikatsioon ja ekvivalents) assotsieeruvad paremale.

```
a and b and c = a and (b and c)
```

Loogiliste tehete erinevad prioriteedid võimaldavad kasutada tavapäraseid sulgude ärajätmisreegleid

```
a and b or c = (a and b) or c
```

sest analüüsi alustatakse väiksema prioriteediga tehetest. Loomulikult võib kasutada ka Prologi siseseid lausearvutuse tehete tähistusi (`not`, `\+`, `','`, `';'`, `':-'`).

Ülesanne 5 Andke lausearvutuse valemi tüübikirjeldus.

```
?-wff(a and b <-> ~c).
yes
?-wff(a+1).
no
?-wff(true and false).
yes
```

Ülesanne 6 Kirjutage programm, mis kontrollib lausearvutuse valemite loogilist tõesust ja väärust.

2.6 Sümbolteisendused

Prolog on eriti sobiv vahend süntaktiliste avaldiste teisendamiseks. Võtame näiteks aritmeetiliste avaldiste lihtsustamise.

$$(x+0)+2x = x+x+x = 3x$$
$$1*x+0y+(0+1)*z = x+z$$

Lihtsustamise realiseerimiseks paneme kirja paar lihtsat teisendusreeglit.

```
simplify(X+0,X). % x+0=x
simplify(1*X,X). % 1*x=x
simplify(X*1,X). % x*1=x
```

Lisaks realiseerime ka süntaksipuus liikumise ja teisendamist mittevajavad juhud (näiteks muutujad).

```
simplify(X+Y,SX+SY):-simplify(X,SX), simplify(Y,SY).
simplify(X*Y,SX*SY):-simplify(X,SX), simplify(Y,SY).
simplify(S,S). %muutujad jms.
```

Tulemus on "suurepärane, kuid mitte lootusetu".

```
?-simplify((x+0)+2*x,S).
S = x+2*x
?-simplify(1*x+0*x+(0+1)*z,S).
S = x+0*y+(0+1)*z
```

Ülesanne 7 Lõpetage programm *simplify*.

Ülesanne 8 Realiseerige predikaat *linearize* aritmeetiliste avaldiste teisendamiseks vasakassotsiatiivsele kujule.

```
?-linearize(a+b+c+d,((a+b)+c)+d).
yes
?-linearize((a+b)+(c+d),X).
X = a+b+c+d
```

Ülesanne 9 Kirjutage programm, mis viib infi kskujulise aritmeetikavalemi postfi kskujule.

$$(a+b/c)*(d-e*f) = abc/+def*-*$$

2.7 Taust

Sugulaste leidmise programm valmis loengusisese diskussiooni käigus koostöös Targo Tennisbergiga. Infikspredikaatide idee on pärit Jaak Hennolt [27] ja Bratkolt [10].

3 Listid

Standardsed listipredikaadid. Listi summeerimine ja sorteerimine.

3.1 Sugulusmõistete genereerimine

Sugulaste märgendatud teadmistebaas pakub huvitavaid võimalusi. Eespool väljastasime Fredi kõik esimese ja teise ringi sugulased. Käsuga `findall` saab neist moodustada loendi (*list*).

```
?-findall(X,relative(2,fred,X),List).
```

Võimalusi on veel. Näiteks võib küsida: kes on Fredi jaoks Linda? Saame kaks vastust: Linda on Fredi ema ja tema isa abikaasa.

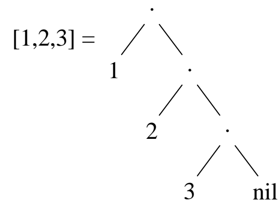
```
?-findall(X,relative(2,fred,linda,X),List).
List = [ema,[isa,abikaasa]]
```

Loetledes aga Fredi kuni kaheksanda taseme sugulasi annab `findall`'iga analoogiline käsk `setof`, mis kaotab listist kordused ja järjestab vastused tähestikuliselt, 3406 erinevat ema defi nitsiooni.

Ülesanne 10 Nimetage Fredi ja Linda näite varal kümmekond ema defi nitsiooni. Korrake katset käsuga `setof`.

3.2 Listid

List defineeritakse binaarse '.'-predikaadi abil rekursiivse struktuurina, mille esimeseks argumendiks on listi element (*head*) ja teiseks elemendiks listi ülejäänud osa (*tail*). Nurksulgudes avaldis on '.'-struktuuri väline Lispi-



Joonis 4: Listi sisemine esitus.

laadne esitus, milles listi ülejäänud osa (*cdr*) eraldatakse listi peast (*car*) cons-operaatoriga '`|`'. Nurksulgudega antakse ka listi definitsioon, mida on hea kasutada listide kontrollimiseks.

Sisekuju	Cons-operaatoriga	Elementide loendina
.(a,[])	[a|[]]	[a]
.(a,.(b,[]))	[a|[b|[]]]	[a,b]
.(a,.(b,.(c,[])))	[a|[b|[c|[]]]]	[a,b,c]
.(a,X)	[a|X]	[a|X]
.(a,.(b,X))	[a|[b|X]]	[a,b|X]

Tabel 4: Listi ekvivalentsed esitused.

```
is_list([]).
is_list([_|Xs]):-
    is_list(Xs).
```

`[]` tähistab siin tühilisti `[]`. Näiteks

```
?-is_list([1,2,3]).
?-is_list([X|_]).
?-is_list([1,2|_]).
?-is_list(X).
```

Kolm viimast näidet on osalised listid, sest nende saba pikkus on defineerimata. Viimases näites on tegemist kõdunud juhuga, sest listiks kuulutatakse tavaline muutuja. Selle variandi saab kõrvaldada, kasutades metapredikaati `nonvar`:

```
proper_list(X):-nonvar(X), is_list(X).
```

Arvulisi liste on võimalik kokku liita ja järjestada. Liitmiseks vaatame listi elementhaaval vasakult paremale läbi:

```
sum([],Sum):-X is Sum, write(X).
sum([X|Xs],Sum):-
    sum(Xs,Sum+X).
```

Sufiks 's' muutuja `Xs` järel ütleb, et tegemist on elementide loeteluga[1]. Muutuja `Sum` sisaldab predikaadi *sum* täitmisel listi läbivaatatud elementide summat. Jõudes listi lõppu, arvutatakse summa käsuga `is` välja ja kuvatakse ekraanile:

```
?-sum([1,2,3],0).
?-sum([2,3],0+1).
?-sum([3],0+1+2).
?-sum([],0+1+2+3).
?-X is 0+1+2+3, write(X).
?-write(6).
```

Tegelikult on see programm täiesti kasutu, sest ta ei tee arvutatud summat teistele programmidele kättesaadavaks. Proovime programmi ümber kirjutada kujul

```
sum([],Sum):-0 is Sum.
sum([X|Xs],Sum):-
    sum(Xs,Sum-X).
```

Kahjuks ei saa selliselt soovitud tulemust, sest käsu `is` täitmisel on muutuja `Sum` väärtustamata, põhjustades aritmeetikavea:

```
?-sum([1,2,3],Sum).
?-sum([2,3],Sum-1).
?-sum([3],Sum-1-2).
?-sum([],Sum-1-2-3).
?-0 is Sum-1-2-3.
[WARNING: Arguments are not sufficiently instantiated]
 ^ Exception: ( 12) 0 is Sum-1-2-3 ?
```

[1] 's' on inglise keeles mitmuse tunnus.

Uuemad loogilise programmeerimise süsteemid kasutavad tingimustega programmeerimise meetodeid (*constraint programming*), mistõttu nad saavad hakkama ka lihtsamate sümbolvõrrandite lahendamisega. Näiteks Eclipse'is piisab muudatustest

```
:-lib(fd). %finite domains :-use_module(...)?
sum([],Sum):-0 #= Sum.

?-sum([1,2,3],Sum).
Sum = 6
```

Prologis kasutatakse tulemuse väljastamiseks akumulaatorit ja lisaargumenti. Lisame ka listi tüübikontrolli, sest summeerimine omab mõtet vaid arvuliste listide korral:

```
sum(List,Sum):-integer_list(List), sum(List,0,Sum).
sum([],Sum,Sum).
sum([X|Xs],Acc,Sum):-
    Acc2 is Acc+X,
    sum(Xs,Acc2,Sum).

integer_list([]).
integer_list([X|Xs]):-
    integer(X),
    integer_list(Xs).
```

Käsk `trace` annab listi summeerimisel järgmise tulemuse:

```
?-sum([1,2,3],Sum).
?-integer_list([1,2,3]), sum([1,2,3],0,Sum).
?-integer(1), integer_list([2,3]), sum([1,2,3],0,Sum).
?-integer_list([2,3]), sum([1,2,3],0,Sum).
?-integer(2), integer_list([3]), sum([1,2,3],0,Sum).
?-integer_list([3]), sum([1,2,3],0,Sum).
?-integer(3), integer_list([]), sum([1,2,3],0,Sum).
?-integer_list([]), sum([1,2,3],0,Sum).
?-sum([1,2,3],0,Sum).
?-Acc2 is 0+1, sum([2,3],Acc2,Sum).
?-sum([2,3],1,Sum).
?-Acc2 is 1+2, sum([3],Acc2,Sum).
?-sum([3],3,Sum).
?-Acc2 is 3+3, sum([],Acc2,Sum).
?-sum([],6,Sum).
Sum = 6
```

Listi elemendiks olemist kontrolliv predikaat *member* on Prologis tavaliselt juba olemas. Ta ütleb, et objekt on kas listi esimene element (1. disjunkt) või ta esineb listi sabas (2. disjunkt).

```
member(X,[X|_]).
member(X,[_,Ys]):-
    member(X,Ys).
```

Ülesanne 11 Realiseerige predikaat *last*, mis leiab listi viimase elemendi.

```
?-last(3,[1,2,3]).
```

Ülesanne 12 Realiseerige predikaat *flatten*, mis kaotab listist alamlistid.

```
?-flatten([1,[[2,3],4],5],[1,2,3,4,5]).
```

Hoopis huvitavam on listi elementide järjestamine, näiteks mittekahanevas järjekorras: `[1,2,2,3]`. Järjestamiseks tuleb leida järjestamistingimust rahuldav listi elementide permutatsioon e. järjestus.

```
slowsort(List,SortedList):-
    integer_list(List),
    permutation(List,SortedList),
    ordered(SortedList).
```

Permutatsioonide genereerimiseks eemaldame listist järgemööda kõik elemendid ning asetame nad listis esimesele kohale. Eemaldamispredikaat *select* sarnaneb predikaadiga *member*.

```
permutation([],[]).
permutation(List,[X|Xs]):-
    select(List,X.List2),
    permutation(List2,Xs).

%select([],_,[]).
select([X|Xs],X,Xs).
select([X|Xs],Y,[X|Zs]):-
    select(Xs,Y,Zs).
```

Ülesanne 13 Realiseerige programm `slowsort` ja testige selle kiirust.

Ülesanne 14 Mõelge välja ja realiseerige mõni teine sorteerimisalgoritm.

Ülesanne 15 Realiseerige mittearvulise listi sorteerimine (näiteks aatomite tähestikuline järjestamine).

```
?-sort([tiina,mari,külli],L).
L = [külli,mari,tiina]
```

3.3 Prologi miinused

Loomulikult on Prologil ka miinuseid.

- Aeglane. Warreni abstraktseks masinaks kompileeritud programmi täitmine on umbes kümme korda aeglasem kui imperatiivsetes keeltes. Masinkoodi kompileerimisel on vahe umbes kolmekordne.
- Raiskab mälu. Mälu on muutunud odavaks, mistõttu mälupiirangute osatähtsus väheneb pidevalt.
- Puudub tüübikontroll. Efektiivseks kompileerimiseks on see muidugi tähtis. Kuid siis on tegemist teise keelega — Prologi käsitletakse tavaliselt kui mittedetermineeritud tüüpideta programmeerimiskeelt.
- Puuduvad alamprogrammide teegid. Amzi, LPA ja Visual Prologi jaoks on juba loodud mitmeid teeke.
- Raske siluda. Prologi programmi silumine ei erine oluliselt teistes keeltes kirjutatud programmide silumisest.
- Kasutajaliidese puudumine. Paljud Prologi süsteemid sisaldavad ka tänapäevast kasutajaliidest.
- Ei anna täidetavat programmi. Paljud Prologi kommertsversioonid võimaldavad kompileerida programmi iseseisvalt täidetavale kujule. SWI-Prolog võimaldab ühendada Prologi programme C-programmidega. Uuemates Prologi versioonides on vahendid suhtlemiseks veebi programmidega.

Ülesanne 16 Kirjutage programm, mis näitab, kuidas mõõta 7- ja 5-liitriste anumate abil 4 liitrit vedelikku.

3.4 Taust

Summa arvutamine lahutamise abil pakkus loengu käigus välja Oleg Mürk. Programm `slowsort` kujutab endast Prologi klassikat. Selle nime all on seda analüüsitud John Lloydi monograafias [43].

4 Lausete süntaksianalüüs

Grammatika kirjeldamine.

4.1 Sorteerimise sisend

Sorteerimisprogrammi on tarvis ka testida. Sisendandmed saab lihtsasti genereerida süsteemse predikaadiga *range* või *between*.

```
?-between(1,100,X).
X = 1 ;
X = 2 ;
...
```

Moodustame näiteks listi $[1,\ldots,100]$ ja pöörame selle ringi — et sorteerimisel oleks mõtet.

```
?-findall(X,between(1,100,X),List), reverse(List,Newlist),
  sort(Newlist).
```

Arvude segiajamiseks võib ka liita kaks järjestatud listi kokku.

```
?-findall(X,range(1,500,X),List1),
  findall(X,range(1,500,X),List2),
  append(List1,List2,List), sort(List).
```

Programmi efektiivsuse testimiseks võib kasutada käskusid `cputime` või `time(Goal)`.

```
?-Goal=sort(List),
  X is cputime, Goal, Y is cputime, Time is Y-X, write([X,sec]).
```

Muutujale `Goal` omistati siin Prologi päring, mistõttu teda nimetatakse metamuutujaks (*metavariable facility*). Samuti võib kasutada Prologisüsteemi sisemist statistikat päringule kulutatud aja ja kasutatud mälu kohta. Näiteks käsk *time*/1 täidab nagu *once*/1 päringut ainult korra, kustutades predikaatide kastide sisemised viidad (*choice point*). Teatud juhtudel võib saada statistikat ka iga predikaadi kohta eraldi (*profiling*).

4.2 Lihtne eesti keele grammatika

Arvutiteaduses on hästi tuntud kontekstivabad grammatikad (*context-free grammar*), mis esitatakse teisendusreeglite abil, kus vasakul on asendatav mitteterminaal ja paremal on asenduse tulemus: terminaalide ja mitteterminaalide jada — mitteterminaalse fraasi võimalikud kujud.

<mitteterminaal>⟶ <fraas>
<mitteterminaal>⟶ <fraas>
...

Igale kontekstivabale grammatikale G vastab teatav keel L, mille sõnadeks on grammatika algsümbolist S genereeritavad terminaalidest koosnevad tähekombinatsioonid.

$$L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}$$

Prolog on programmeerimiskeelena kontekstivabadest grammatikatest tunduvalt võimsam. Lihtne on näidata, et Prologiga saab sooritada kõiki Turingi masina tehteid. Järelikult on Prolog programmeerimiskeelena — nagu teisedki programmeerimiskeeled — Turingi masinaga ekvivalentne.

Kuna Prolog loodi 1970-ndate aastate alguses lingvistilistel eesmärkidel, siis ei valmista temas grammatikate realiseerimine tavaliselt probleeme. Võtame näiteks kontekstivaba grammatika, mis kirjeldab imepisikese osa eestikeelsetest lausetest. Esimene reegel ütleb, et lause koosneb aluserühmast ning sellele järgnevast öeldiserühmast.

lause	⟶	aluserühm öeldiserühm
aluserühm	⟶	alus
aluserühm	⟶	täiend alus
öeldiserühm	⟶	öeldis
öeldiserühm	⟶	öeldis sihitis_määrus
sihitis_määrus	⟶	sihitis
sihitis_määrus	⟶	määrus
sihitis_määrus	⟶	sihitis määrus
alus	⟶	nimisõna
öeldis	⟶	tegusõna
täiend	⟶	omadussõna
sihitis	⟶	nimisõna
määrus	⟶	määrsõna

Esitame ka grammatikale vastava sõnaraamatu, kus terminaalid on tähistatud nurksulgudega.

nimisõna	$\longrightarrow$	[mees]
nimisõna	$\longrightarrow$	[kive]
nimisõna	$\longrightarrow$	[marju]
tegusõna	$\longrightarrow$	[kannab]
tegusõna	$\longrightarrow$	[korjab]
omadussõna	$\longrightarrow$	[noor]
omadussõna	$\longrightarrow$	[tubli]
määrsõna	$\longrightarrow$	[hästi]

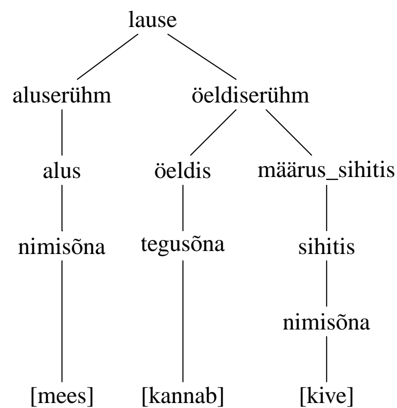
Vaadeldava grammatika järgi võib lause koosneda kas alusest ja öeldisest

Mees kannab.

alusest öeldisest ja sihitisest

Mees kannab kive.

vms. Grammatika realiseerimiseks Prologis kujutame sõnu aatomitena ja lauseid sõnade listidena. Terminaalid esitame faktidena. Laused lõhume sõnadeks ja osalauseteks käsuga `append`. Erineva struktuuriga lausete ana-



Joonis 5: Süntaksipuu.

lüüsimiseks kasutame Prologi variantide läbivaatamise võimet (*backtracking*). Alguses kontrollime, kas lause on lihtsama struktuuriga; kui ei ole, siis võtame vaatluse alla keerukamad lausekonstruktsioonid. Selliselt toimime kuni leiame lausele vastava grammatilise struktuuri või vaatame läbi kõik grammatikaga lubatud lausekonstruktsioonid ja kuulutame lause antud grammatikale mittevastavaks. Käsuga `append` realiseeritud grammatikad ei ole kahjuks efektiivsed — `append` töötab lausete analüüsimisel generaatorina, andes analüüsitava lause kõikvõimalikud algused

[], [mees], [mees, kannab], [mees, kannab, kive].

Lausete analüüsimise kiirendamiseks kasutatakse erilisi struktuure: diferentsliste (*difference list*), mis jaotavad listi kaheks pooleks — vasakpoolseks, juba analüüsitud osaks, ja parempoolseks, analüüsimata osaks. Meie näite korral saame diferentslisti

[mees, kannab, kive] – [kannab, kive],

kus miinusmärgi ees on kogu list ja selle järgneb listi parem pool; listi vasakpoolne osa koosneb sõnast “mees” ja parempoolne osa sõnadest “kannab” ja “kive”.

Diferentslisti käsitletakse tavaliselt termina kujul $x - y$, s.t. listide x ja y vahena. Näiteks

$$[1,2,3|Xs] - Xs = [1,2,3]$$

on listi $[1,2,3]$ kõige üldisem esitus diferentslistina. On selge, et iga listi L saab esitada diferentslistina $L - []$. Diferentslistide tähtsaks omaduseks on nende efektiivsus. Näiteks käsk

```
concat(x-y,y-z,x-z).
```

ühendab diferentslistid $[1,2,3|Xs] - Xs$ ja $[4,5|Ys] - Ys$ ühe sammuga, saades tulemuseks listi

$$[1,2,3,4,5|Zs] - Zs.$$

Diferenstlistid sarnanevad osaliste struktuuridega (*partial structure*), sest määramata teise argumendi korral on diferentslist $x - y$ alati unifitseeritav diferentslistiga y. Diferentslistide abil saame ülevalt-alla vasakult-paremale rekursiivse analüsaatori, mis leiab lahendi variantide läbivaatamisega (*backtracking*).

```
lause(L):-
    aluserühm(X),
    öeldiserühm(Y),
    append(X,Y,L).
aluserühm(A):-alus(A).
aluserühm(A):-
    täiend(X),
    alus(Y),
    append(X,Y,A).
öeldiserühm(Q):-öeldis(Q).
öeldiserühm(Q):-
    öeldis(X),
    sihitis_määrus(Y),
    append(X,Y,Q).
sihitis_määrus(S):-sihitis(S).
sihitis_määrus(M):-määrus(M).
sihitis_määrus(S):-
    sihitis(X),
    määrus(Y),
    append(X,Y,S).
alus(A):-nimisõna(A).
öeldis(Q):-tegusõna(Q).
täiend(T):-omadussõna(T).
sihitis(S):-nimisõna(S).
määrus(M):-määrsõna(M).

nimisõna([mees]).
nimisõna([kive]).
nimisõna([marju]).
tegusõna([kannab]).
tegusõna([korjab]).
omadussõna([noor]).
omadussõna([tubli]).
määrsõna([hästi]).

?-lause([mees,kannab,kive]).
```

Joonis 6: Grammatika `append`'i abil.

```
lause(L-L0):-aluserühm(L-L1), öeldiserühm(L1-L0).
aluserühm(X-Y):-alus(X-Y).
aluserühm(X-Y):-täiend(X-Z), alus(Z-Y).
öeldiserühm(X-Y):-öeldis(X-Y).
öeldiser"uhm(X-Y):-öeldis(X-Z), sihitis_määrus(Z-Y).
sihitis_määrus(X-Y):-sihitis(X-Y).
sihitis_määrus(X-Y):-määrus(X-Y).
sihitis_määrus(X-Y):-sihitis(X-Z), määrus(Z-Y).

alus(X-Y):-nimisõna(X-Y).
öeldis(X-Y):-tegusõna(X-Y).
täiend(X-Y):-omadussõna(X-Y).
sihitis(X-Y):-nimisõna(X-Y).
määrus(X-Y):-määrsõna(X-Y).

nimisõna([mees|S]-S).
nimisõna([kive|S]-S).
nimisõna([marju|S]-S).
tegusõna([kannab|S]-S).
tegusõna([korjab|S]-S).
omadussõna([noor|S]-S).
omadussõna([tubli|S]-S).
määrsõna([hästi|S]-S).

?-lause([mees,kannab,kive]-[]).
```

Joonis 7: Grammatika üheargumendiliste diferentslistide abil.

Lähme nüüd oma näitega edasi. Muudame grammatika diferentslistide abil efektiivsemaks. Esitame terminaalid faktidena, mis eemaldavad listi peast elemente, lühendades järkjärgult listi analüüsimata osa. Iga grammatikareegel analüüsib nüüd sisendsõnade jada vasakult paremale, kuni leiab sobimatu sõna, või eraldab jada algusest reegliga analüüsitud sõnad, jättes diferentslisti parempoolse osa analüüsimise teistele reeglitele.

Kuna miinusmärki sisaldava termi unifitseerimine tähendab ühte lisaoperatsiooni, asendatakse see efektiivsuse huvides komaga, suurendades niiviisi predikaatide argumentide arvu — diferentslist muutub ühest argumendist kaheks komaga eraldatud argumendiks.

```
lause(L,L0):-aluserühm(L,L1), öeldiserühm(L1,L0).
aluserühm(X,Y):-alus(X,Y).
aluserühm(X,Y):-täiend(X,Z), alus(Z,Y).
öeldiserühm(X,Y):-öeldis(X,Y).
öeldiserühm(X,Y):-öeldis(X,Z), sihitis_määrus(Z,Y).
sihitis_määrus(X,Y):-sihitis(X,Y).
sihitis_määrus(X,Y):-määrus(X,Y).
sihitis_määrus(X,Y):-sihitis(X,Z), määrus(Z,Y).

alus(X,Y):-nimisõna(X,Y).
öeldis(X,Y):-tegusõna(X,Y).
täiend(X,Y):-omadussõna(X,Y).
sihitis(X,Y):-nimisõna(X,Y).
määrus(X,Y):-määrsõna(X,Y).

nimisõna([mees|S],S).
nimisõna([kive|S],S).
nimisõna([marju|S],S).
tegusõna([kannab|S],S).
tegusõna([korjab|S],S).
omadussõna([noor|S],S).
omadussõna([tubli|S],S).
määrsõna([hästi|S],S).

?-lause([mees,kannab,kive],[]).
```

Joonis 8: Grammatika kaheargumendiliste diferentslistide abil.

Grammatikatega töötamiseks on Prologis käsud *phrase*/2 ja *phrase*/3, mis kontrollivad, kas teine argument on analüüsitav kui esimeseks argumendiks olev grammatiline konstruktsioon.

```
?-phrase(lause,[mees,kannab,kive]).
?-phrase(lause,[mees,kannab,kive],[]).
```

Käsk *phrase*/2 eeldab, et diferentslisti analüüsimata osa on tühilist. Terminaalides ja mitteterminaalides jäetakse argumendid ära, sest Prolog võimaldab kirjutada kontekstivabu grammatikaid loomulikumas — nooletähistuses — hoides neid grammatikareegleid sisemises andmebaasis predikaatidena, millel on diferentslisti jaoks kaks lisaargumenti. Vastav teisendus

tehakse grammatika sisselugemisel automaatselt. Grammatikareeglite abil

```
lause --> aluserühm, öeldiserühm.
aluserühm --> alus.
aluserühm --> täiend, alus.
öeldiserühm --> öeldis.
öeldiserühm --> öeldis, sihitis_määrus.
sihitis_määrus --> sihitis.
sihitis_määrus --> määrus.
sihitis_määrus --> sihitis, määrus.

alus --> nimisõna.
öeldis --> tegusõna.
täiend --> omadussõna.
sihitis --> nimisõna.
määrus --> määrsõna.

nimisõna --> [mees]; [kive]; [marju].
tegusõna --> [kannab]; [korjab].
omadussõna --> [noor]; [tubli].
määrsõna --> [hästi].

?-lause([mees,kannab,kive],[]).
```

Joonis 9: Horni grammatika.

esitatud grammatikaid nimetatakse Horni grammatikateks (*definite clause grammar*). Nende rakendustest räägime aga edaspidi.

Ülesanne 17 Laiendage `append`'il, diferentslistidel ja grammatikareeglitel põhinevaid grammatikaid lausetele, milles alusel võib olla mitu täiendit (või piiramatu arv täiendeid):

> Noor tubli mees kannab kive.

Ülesanne 18 (rekursioon) Laiendage `append`'il, diferentslistidel ja grammatikareeglitel põhinevaid grammatikaid lausetele, mis võivad omakorda sisaldada uusi lauseid:

> Mees ütleb, et noor mees kannab kive.

Ülesanne 19 Koostage grammatika, mis tunneb ära sõnu kujul $a^n b^n$.

Ülesanne 20 Koostage grammatika, mis tunneb ära sõnu kujul $a^n b^n c^n$.

Ülesanne 21 Koostage aritmeetikavalemite grammatika.

Ülesanne 22 Koostage loogikavalemite grammatika.

4.3 Grammatikareeglite teisendamine

Quintus Prologis ei kasutata mitte tavalisi kaheargumendilisi diferentsliste, vaid töödeldakse grammatikareegleid käsuga `C` (*connects*). Näiteks reegel

```
nimisõna --> [mees].
```

teisendatakse kujule

```
nimisõna(L1,L):-'C'(L1,mees,L).
```

kus `C` on defineeritud kui

```
'C'([X|Y],X,Y).
```

Probleem on nimelt selles, et grammatikareeglites võib kasutada ka tavalisi Prologi käske, ümbritsedes need loogeliste sulgudega. Vaatleme näiteks grammatikareeglit

```
p --> [sest], {write('Käes!')}, [et].
```

milles analüüsitakse kõigepealt sõna "sest", väljastatakse teade ja analüüsitakse seejärel sõna "et". Teadete trükkimine on kasulik programmide silumiseks. Quintus Prolog teisendab reegli kujule

```
p(L1,L):-'C'(L1,sest,L2), write('Käes!'), 'C'(L2,et,L).
```

Kuid vanemates Prologides on reegli sisekujuks eksitav

```
p([sest,et|L],L):-write('Käes!').
```

sest ta väljastab teate alles pärast mõlema sõna analüüsimist. Lõikepredikaadi (*cut*) kasutamisel võivad aga tulemused oluliselt erineda.

Ülesanne 23 Prologi grammatikareeglite sisselugemine toimub käsuga `expand_term`, mis teisendab grammatikareeglid sisekujule, muutmata ülejäänud terme. Seda käsku kasutavad programmi sisselugemiseks ja automaatseks teisendamiseks süsteemsed predikaadid `consult` ja `reconsult`.

Kontrollige, kas teie Prolog tunneb käsku `expand_term`.

```
?-expand_term((s --> np, vp), Tulemus).
?-expand_term(green(kermit), Tulemus).
```

Ülesanne 24 Uurige, kuidas teie Prolog teisendab reeglit

```
p --> {write('Töötlen sest-et')}, [sest,et].
```

Kuidas te selle teada saite? Millised on teisenduse eelised ja puudused?

4.4 Prologi võrdlus imperatiivsete keeltega

Prolog on huvitav võrrelda C'ga, Pascaliga ja teiste imperatiivsete keeltega. Esmapilgul tundub, et keeltel on üsna vähe ühiseid jooni. "Imperatiivsed programmeerijad" tunnevad ennast deklaratiivsetes keeltes esialgu küllalt ebakindlalt.

Selgub aga, et Prologil ja Pascalil on palju ühist. Ühes keeles kirjutatud efektiivse programmi saab suure tõenäosusega lihtsasti ümber kodeerida piisavalt efektiivseks programmiks teises keeles. Maksimaalse tulemuse saavutamist takistavad keelte iseärasused. Näiteks Prologi standardis pole indekseeritavaid ühe- ja kahemõõtmelisi massiive, samuti puudub muutujate ja protseduuride otsene tüübikontroll.

4.5 Taust

Tavakeele süntaktilist analüüsi kirjeldatakse pajudes Prologi õpikutes. Sellegipoolest tasub lugeda Horni grammatikate esimest populaarset esitust William Clocksini ja Chris Mellishi raamatus [14]. Samuti Leon Sterlingi ja Ehud Shapiro raamatu teist trükki [64] ja Michael Covingtoni lingvistidele mõeldud [18].

Toodud eesti keele grammatika põhineb Jaak Henno [27] ja Eduard Vääri [74] kooliõpikutel. Sõnastiku valikul said määravaks suvised tegemised Tammeoru talus Tartumaal ja ansambli The Rolling Stones kontsert 8. augustil Tallinna lauluväljakul.

5 Teksti sisestamine

Andmete sisestamine ja väljastamine. Interaktiivsed programmid.

	Pascal	Prolog
muutuja	lokaalne/globaalne	lokaalne
	tüübitud	tüüpimata
	saab korduvalt omistada	saab üks kord omistada
	`var x: integer`	
omistamine	`x:=y`	`x=y` (unifi tseerimine)
	`Inc(x) {x:=x+1}`	puudub
tüübikontroll	tugev	kaudne
hargnemine	`case x of`	mitu defi nitsiooni (;)
	`a:`	`nat(1).`
	`b:`	`nat(2).`
	`end`	
	`if x then y else z`	`x->y;z`
tsükkel	`for x:=1 to 100 do`	rekursioon
	`while y do`	`repeat.`
	`repeat ... until`	`repeat:-repeat.`
protseduur	`function nat(X:integer):boolean`	`nat(X).`

Tabel 5: Prologi ja Pascali võrdlus.

5.1 Teksti jaotamine sõnadeks

Kuna tavalisel ASCII-tekstil puudub loogiline struktuur, tuleb see süntaktiliseks analüüsiks eelnevalt muuta sõnade listiks.

Teisendus võiks välja näha umbes niimoodi:

```
?-read_words(Words).
See on  lause.
Words = ['See',on,lause,'.']
```

Tähtede ja sümbolite sisselugemiseks kasutatakse käsku `get0(C)`, mis loeb aktiivsest sisendvoost järgmise sümboli. Meie jaoks ei sobi käsk `get(C)`, sest ta loeb sisse ainult trükitavaid sümboleid, jättes teksti juhtsümbolid (tühik, realõpp, faililõpp, tabulatsioon jms.) vahele. Vaikimisi on sisendvooks *user*, s.t. klaviatuur, kuid selle saab ära muuta käsuga `see`:

```
?-see('file.txt').
?-seeing(X).
X='file.txt'
?-seen.
```

```
?-seeing(X).
X = user
```

Teksti on võimalik sisse lugeda kahte moodi: kas ridade kaupa või lausete kaupa. Realiseerime neist esimese. Kui teksti ridade kaupa lugemine on selge, ei tohiks probleeme tekitada ka sealt lausete välja eraldamine.

Kõigepealt määrame ära kirjavahemärgid (*punctuation*) ja teksti juhtsümbolid (*not_other*). Ülejäänud sümbolitest moodustame siis sõnad. Kirjavahemärkideks loeme jutumärgid, koma, semikooloni, kooloni, punkti, küsimärgi ja hüüumärgi. Apostroofi kasutatakse tekstis kahel otstarbel: sõnade esiletõstmiseks ja sufiksi eraldamiseks (näiteks: sõna 'mets', härra X'ga). Seepärast käsitleme apostroofi lihtsalt sõna osana.

Reavahetuse tunnuseks on sõltuvalt Prologist ja kasutatavast operatsioonisüsteemist kas CR (*carriage return*), LF (*line feed*) või nende kombinatsioon. Lõikepredikaat (!) on lisatud predikaatide definitsioonidele efektiivsuse huvides — ta keelab ära töödeldava predikaadi ülejäänud variantide läbivaatamise.

```
punctuation(34). /*"*/
%punctuation(39). /*'*/
punctuation(44). /*,*/
punctuation(59). /*;*/
punctuation(58). /*:*/
punctuation(46). /*.*/
punctuation(63). /*?*/
punctuation(33). /*!*/

not_other(_):-at_end_of_stream, !.
not_other(13):-!.  /*CR*/
not_other(10):-!.  /*LF*/
%not_other(-1):-!.  /*EOF*/
not_other(9):-!.   /*tabulatsioon*/
not_other(32):-!.  /*tühik*/
not_other(C):-punctuation(C), !.
```

Sõnade sisselugemise programmeerime kaheolekulise automaadina: esimeses olekus *read_words* liigume sõna algusesse, s.t. juhtsümbolitest ja kirjavahemärkidest erineva sümbolini, teises olekus *read_rest_of_word* liigume sõna lõppu ja moodustame kogutud sümbolitest sõna — antud juhul Prologi aatomi — kas käsuga `name(Atom,List)` või käsuga `string_to_list(Atom,List)`. Käsk *name* töötab ka teisipidi, andes aatomi järgi vastava stringi.

```
?-name('Tõnu',String).
String = [84, 245, 110, 117]
?-name(T,"Tõnu").
T = 'Tõnu'
```

Kuna käsk *get0* ei võimalda sama sümbolit korduvalt sisse lugeda, tuleb sõnale järgnevat sümbolit esimesse olekusse tagasi pöördudes argumendina meeles pidada. Sama teeme ka pöördudes vaheldumisi predikaatide *read_words*/1 ja *read_words*/2 poole — jätame tühikud ja tabulatsioonid vahele, realõpu korral lõpetame aga töö. Lõikepredikaati kasutame sõna alguse tuvastamiseks — kui oleme leidnud juhtsümbolist ja kirjavahemärgist erineva sümboli. Kirjavahemärgid jätame aga meelde ühesümboliliste sõnadena.

```
% read_words(Words)
%   loeb sisendvoost ühe tekstirea
%   ja moodustab neist sõnade listi Words
read_words(_):- at_end_of_stream, !, fail.
read_words(Words):-
    get0(C),
    read_words(C,Words).

read_words(_,[]):-at_end_of_stream, !. /*EOF*/
read_words(13,[]):-!. /*CR*/
read_words(10,[]):-!. /*LF*/
%read_words(-1,[]):-!, fail. /*EOF*/
read_words(9,Words):-!, /*tab*/
    read_words(Words).
read_words(32,Words):-!, /*tühik*/
    read_words(Words).
read_words(C,[Word|Words]):-
    punctuation(C),
    !,
    name(Word,[C]),
    read_words(Words).
read_words(C,[Word|Words]):-
    /*C ei ole failil&otilde;pp, reavahetus, tühik, tabulatsioon
      ega kirjavahemärk */
    read_rest_of_word(Chars,LeftOver),
    name(Word,[C|Chars]), /*string_to_list*/
    read_words(LeftOver,Words).
```

```
read_rest_of_word(Chars,LeftOver):-
    get0(C),
    read_rest_of_word(C,Chars,LeftOver).

read_rest_of_word(C,[],C):-
    not_other(C),
    !.
read_rest_of_word(C,[C|Chars],LeftOver):-
    read_rest_of_word(Chars,LeftOver).
```

Nüüd on meil klaviatuurilt teksti sisestamine selge. Failist teksti lugemisel peab ära tundma ka faili lõpu (*end of file*). SWI-Prologis on faililõpu sümboli koodiks -1, kuid selle aitab ära tunda ka predikaat *at_end_of_stream*. Failist lugemiseks sobib niisiis päring

```
?-repeat_stream, read_words(Words), write(Words), nl, fail.
```

kus predikaat *repeat_stream* loeb failist uusi ridu kuni jõutakse faili lõppu.

```
repeat_stream.
repeat_stream:-at_end_of_stream, !.
repeat_stream:-repeat_stream.
```

Enne tuleb muidugi tekstifail avada ja pärast lõpetamist fail sulgeda (käskudega *see*/1 ja *seen*/0). SWI-Prologis saab sisend- ja väljundvoo määrata käskudega *open*/3 ja *close*/1. Faili avamisel (*open*) on esimeseks argumendiks faili nimi, teiseks töörežiim (*read, write, append*) ja kolmandaks argumendiks andmevoo (*stream*) nimi.

```
?-open(foo,read,in), get0(in,C), close(in).
```

Ülesanne 25 Testige teksti sisselugemise programmi kiirust ja püüdke seda muuta efektiivsemaks.

Ülesanne 26 Realiseerige teksti sisselugemine lausete kaupa. Kasutage predikaati *read_words*/1, mis loeb teksti ridade kaupa.

Ülesanne 27 Koostage sõnade sagedussõnaraamat.

Ülesanne 28 Pakkige sõnaraamat Cooperi meetodil, mis asendab sõnade kokkulangevad prefi ksid kokkulangevate tähtede arvudega.

```
?-cooper([aabits,aabitsatõde,aadel],Zip).
Zip = [aabits,6,atõde,2,del]
```

Ülesanne 29 Kirjutage programm eestikeelsete sõnade poolitamiseks.

```
?-hyphenate(maasikas,maa-si-kas).
```

Ülesanne 30 Genereerige mingi käändkonna sõnade käändevormid.

```
?-vormid(tuba,[tuba,toa,tuba,tuppa]).
```

Ülesanne 31 Realiseerige teisendus rot13, mis roteerib tähtede ascii-koode kolmteist kohta edasi.

```
?-rot13(pasunakoor,cnfhanxbbe).
```

5.2 Eliza

Joseph Weizenbaum realiseeris aastatel (1964–1966) MIT'is programmi Eliza [77], mis imiteeris psühhoterapeudi esimest vestlust patsiendiga. Vestluse iseloom oli muudetav vastavate skriptidega. Edinburghis kasutati hiljem Eliza lihtsustatud varianti Prologi õpetamiseks.

Joonisel 10 on katkend Eliza ja patsiendi vahelisest vestlusest. Eliza ei püüa mitte tekstist aru saada, vaid otsib sellest teatavaid sõnu ja väljendeid. Näiteks sõna “ema” esinemisel palub ta patsiendil rääkida rohkem oma perest. Aga vahel ta sõnastab patsiendi lause ringi, tehes seal sobivad asendused. Näiteks asendab ta lauses

> Kuidas sul läheb?

sõna “sul” sõnaga “mul”, võimaldades patsiendil endast rohkem rääkida.

> Kuidas mul läheb?

Ülesanne 32 Kirjutage triviaalne tõlkimisprogramm, mis asendab ingliskeelsed sõnad eestikeelsete vastetega.

> What time is it?
> Mis aeg on see?

Ülesanne 33 Formuleerige toodud katkendi põhjal Eliza vastamisreeglid.

Ülesanne 34 Kirjutage Prologi programm, mis jäljendab Eliza ja patsiendi vahelist dialoogi.

Men are all alike.
IN WHAT WAY
They're always bugging us about something or other.
CAN YOU THINK OF A SPECIFIC EXAMPLE
Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE
He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED
It's true. I am unhappy.
DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY
I need some help, that much seems certain.
WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
Perhaps I could learn to get along with my mother.
TELL ME MORE ABOUT YOUR FAMILY
My mother takes care of me.
WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU
My father.
YOUR FATHER
You are like my father in some ways.
WHAT RESEMBLANCE DO YOU SEE
You are not very aggressive but I think you don't want me to notice that.
WHAT MAKE YOU THINK I AM NOT VERY AGGRESSIVE
You don't argue with me.
WHY DO YOU THINK I DON'T ARGUE WITH YOU
You are afraid of me.
DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU
My father is afraid of everybody.
WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR FATHER
Bullies.
DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR BOYFRIEND MADE YOU COME HERE

Joonis 10: Patsiendi ja ELIZA vaheline dialoog.

5.3 Ekspertsüsteem

Faktide ja reeglite esitamine, dünaamilised andmebaasid ja sisseehitatud otsingumootor (*inference engine*) teevad Prologist ideaalse vahendi ekspertsüsteemide programmeerimiseks.

Ekspertsüsteemi peamiseks osaks on eksperdi teadmised: andmed ja faktid mingi valdkonna kohta ning nendevahelised seosed. Kuigi Prologis on olemas algelised vahendid andmete talletamiseks ja kasutamiseks, loobutakse tavaliselt andmete esitamisest Prologi programmina ning realiseeritakse infosüsteem — luuakse vahendid andmete salvestamiseks, muutmiseks ja otsimiseks. Michael Hanus realiseeris näiteks Saksamaal Saarbrückenis Sicstus Prologi abil oma instituudi bibliograafilise andmebaasi.

Hoopis huvitavam on tegelda veinidega. Konstrueerime lihtsa andmebaasi, mille kasutajaliides näeb ette operatsioonid nii veinidega kui ka andmebaasiga tervikuna.

```
% Kasutajasõbralik veinide andmebaas
menu:-
    repeat,
    write('Valige number'),nl,nl,
    write(' 1. Lisan veini'), nl,
    write(' 2. Otsin veini'), nl,
    write(' 3. Loetle veinid'), nl,
    write(' 4. Eemaldan veini'), nl,
    write(' 5. Kustutan andmebaasi'), nl,
    write(' 6. Salvestan andmebaasi'), nl,
    write(' 7. Loen andmebaasi'), nl,
    write(' 8. Exit'), nl, nl,
    get(N), code(N,Choice),
    do_it(Choice).
```

Andmebaasi juhitakse numbriklahvidega. Kõigepealt teisendame sisseloetud sümboli koodi vastavaks numbriks ning seejärel sooritame nõutud tegevuse. Kui pole tegemist numbritega 1–8, anname veateate (*error*). Tegevused pöörduvad rekursiivselt tagasi esialgse menüü poole, tekitades lõpmatu tsükli. Mälu see siiski ei raiska, sest rekursiooni minnakse tegevuse viimases käsus ja eelmisi käskusid on vaja täita ainult üks kord (*tail recursion optimization*). Näiteks optimiseerivates Prologisüsteemides töötab rekursiivne predikaat *repeat* lõpmatult, suurendamata kasutatavat mälu.

```
code(49,1):-!.
code(50,2):-!.
```

```
code(51,3):-!.
code(52,4):-!.
code(53,5):-!.
code(54,6):-!.
code(55,7):-!.
code(56,8):-!.
code(_X,0):-error.

do_it(1):-add_to_kb.
do_it(2):-find_a_wine.
do_it(3):-list_all_wines.
do_it(4):-retract_from_kb.
do_it(5):-retract_all.
do_it(6):-save_the_file.
do_it(7):-consult_the_expert.
do_it(8):-exit.

error:-
    write('Vigane valik - Valige uuesti'), nl,
    menu.
```

Igale tegevusele vastab mingi predikaat. Veinid salvestatakse `fail`-tsükliga tekstifaili `wines.txt`, kasutades predikaati *writeq*, lisades faktide automaatseks sisselugemiseks nende lõppu punkti.

```
save_the_file:-
    write('Salvestan andmebaasi'), nl,
    tell('wines.txt'),
    save_wines,
    told,
    menu.

save_wines:-
    wine(Name,Region,Characteristics,Color),
    writeq(wine(Name,Region,Characteristics,Color)),
    write(.), nl,
    fail,
    menu.
save_wines:-!.
```

Andmebaasi loeme sisse käsuga *consult*, nagu harilikult.

```
consult_the_expert:-
```

```
    write('Loen andmebaasi'), nl,
    consult('wines.txt'),
    menu.
```

Lisatava veini karakteristikud loeme käsuga *read*, mistõttu neile peab kindlasti järgnema punkt. Sisestatavad nimed tuleks ümbritseda apostroofidega: 'Põltsamaa kuldne'. Veinid lisatakse sisemise andmebaasi lõppu käsuga *assertz*.

```
add_to_kb:-
    write('Lisan veini'), nl,
    write('Sisestage veini nimi:'), nl,
    read(Name), nl,
    write('Piirkond:'), nl,
    read(Region), nl,
    write('Iseloomustus:'), nl,
    read(Characteristics), nl,
    write('Värv:'), nl,
    read(Color), nl,
    assertz(wine(Name,Region,Characteristics,Color)),
    menu.
```

Veinide dünaamilisest andmebaasist otsimiseks kasutatakse Prologi tavalist otsimismehhanismi.

```
find_a_wine:-
    write('Otsin veini'), nl,
    write('Sisestage veini nimi:'), nl,
    read(Name), nl,
    wine(Name,Region,Characteristics,Color),
    write('Nimi= '), write(Name), nl,
    write('Piirkond= '), write(Region), nl,
    write('Iseloomustus= '), write(Characteristics), nl,
    write('Värv= '), write(Color), nl,
    menu.

list_all_wines:-
    write('Loetle veinid'), nl,
    find_wine,
    nl,
    menu.

find_wine:-
```

```
    wine(Name,Region,Characteristics,Color),
    write('Nimi= '), write(Name), nl,
    write('Piirkond= '), write(Region), nl,
    write('Iseloomustus= '), write(Characteristics), nl,
    write('Värv= '), write(Color), nl,
    nl, fail,
    menu.
find_wine:-!.
```

Veinide kustutamiseks dünaamilisest andmebaasist sobivad käsud *retract* ja *retractall*.

```
retract_from_kb:-
    write('Eemaldan veini'), nl,
    write('Sisestage veini nimi:'), nl,
    read(Name), nl,
    retract(wine(Name,_,_,_)),
    menu.

retract_all:-
    write('Kustutan andmebaasi'), nl,
    retract(wine(_,_,_,_)),
    fail,
    menu.
```

Kõige keerulisem on andmebaasist väljumine. Tegu on ju lõpmatu tsükliga. SWI-Prologis lõpetab programmi töö näiteks klahvikombinatsioon Ctrl-C.

```
exit:-
    write('Väljumiseks vajutage Ctrl-C'), nl,
    menu.
```

Ülesanne 35 Kirjutage veinide ekspertsüsteemi sisendfail.

Ülesanne 36 Sisestage veinide ekspertsüsteem ja õppige seda kasutama.

Ülesanne 37 Realiseerige Prologi teadmistebaasina oma märkmik (nimed, aadressid, telefonid jms.).

Ülesanne 38 Realiseerige Prologis mõni lihtne mäng: mõttemeister, trips-traps-trull vms. Konstrueerige mängu jaoks kasutajaliides.

5.4 Taust

Teksti sisselugemise programm on võetud Richard O'Keefe raamatust [54]. Eliza dialoog pärineb Joseph Weizenbaumi populaarsest raamatust [76]. Omal ajal võtsid paljud, erinevalt programmi autorist, Elizat väga tõsiselt, uskudes tehisintellekti peatsesse võidukäiku. Hiljem leiti, et lihtsatest süntaktilistest teisendustest siiski ei piisa inimvestluse modelleerimiseks — Turingi testi läbimiseks on vaja ka tekstist aru saada. Sellegipoolest on levinud mitmesugused tekstirobotid, mis kasutavad põhimõtteliselt sama strateegiat. Veinide teadmistebaas on võetud Ralph Cafolla ja Daniel Albert Kauffmani käsiraamatust [12].

6 Horni grammatikate rakendused

Lause analüüsipuu konstrueerimine. Ainsuse, mitmuse ja käänete arvestamine.

6.1 Grammatika täiustamine

Punktis 4 toodud eesti keele Horni grammatikat joonisel 9 on võimalik parameetrite lisamisega laiendada, sest grammatikareeglite predikaatidel võivad olla ka argumendid. Diferentslisti kaks sisemist lisaargumenti lisatakse nende järele. Genereerime näiteks Horni grammatikaga süntaksipuu (*parse tee*).

Ülesanne 39 (raske) Kirjutage programm, mis trükib süntaksipuu tekstina nagu joonisel.

Horni grammatikates on lihtne tagada ka ainsuse ja mitmuse ühilduvust. Selleks peavad lause alus ja öeldis olema samaaegselt kas ainsuses või mitmuses. Sihitis võib olla alusest ja öeldisest sõltumatult nii ainsuses kui ka mitmuses. Näiteks laused

Noor mees kannab kive.
Noored mehed kannavad kive.

on grammatiliselt korrektsed. Vigased on aga laused

Noored mees kannavad kive.
Noored mehed kannab kive.

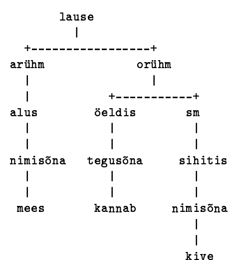
```
lause(lause(A,Q)) --> aluserühm(A), öeldiserühm(Q).
aluser"uhm(arühm(A)) --> alus(A).
aluserühm(arühm(T,A))--> täiend(T), alus(A).
öeldiserühm(orühm(Q)) --> öeldis(Q).
öeldiserühm(orühm(Q,S)) --> öeldis(Q), sihitis_määrus(S).
sihitis_määrus(sm(S)) --> sihitis(S).
sihitis_määrus(sm(M)) --> määrus(M).
sihitis_määrus(sm(S,M)) --> sihitis(S), määrus(M).

alus(alus(N)) --> nimisõna(N).
öeldis(öeldis(T)) --> tegusõna(T).
täiend(t"aiend(O)) --> omadussõna(O).
sihitis(sihitis(N)) --> nimisõna(N).
määrus(määrsõna(M)) --> määrsõna(M).

nimisõna(nimisõna(mees)) --> [mees].
nimisõna(nimisõna(kive)) --> [kive].
nimisõna(nimisõna(marju)) --> [marju].
tegusõna(tegusõna(kannab)) --> [kannab].
tegusõna(tegusõna(korjab)) --> [korjab].
omadussõna(omadussõna(noor)) --> [noor].
omadussõna(omadussõna(tubli)) --> [tubli].
määrsõna(määrsõna(hästi)) --> [hästi].

?-lause(Puu,[mees,kannab,kive],[]).
Puu = lause(arühm(alus(nimisõna(mees))),
            orühm(öeldis(tegusõna(kannab)),
                  sm(sihitis(nimisõna(kive)))))
```

Joonis 11: Süntaksipuu konstrueerimine.



Joonis 12: Süntaksipuu väljatrükk.

Argumendiga `Num` tagame, et lause alus, öeldis ja täiend on samaaegselt kas ainsuses või mitmuses.

Erinevalt inglise keelest tuleb eesti keeles panna rõhku ka sõnade käänetele. Nimelt on lause alus tavaliselt nimetavas käändes ja sihitis osastavas käändes. Aluse ja sihitisega samas käändes peavad olema ka nende täiendid.

> Suur mees kannab suurt kivi.
> Suured mehed kannavad suuri kive.

Järgmised laused loeme aga grammatiliselt vigaseks:

> Suur mees kannab suurt kive.
> Suured mees kannavad suur kive.

Sõnade arvu ja käände ühildamiseks lisame täiendi, aluse ja sihitise predikaatidele teise argumendi, mis väljendab nende käändeatribuuti. Sihitise täiendi jaoks toome sisse sihitiserühma mõiste.

Ülesanne 40 Väljastage lause analüüsipuu asemel öeldise semantiline predikaat:

> kannab(mees,kive)

6.2 Arvude äratundmine

Horni grammatikate kasutamise standardse näitena toome grammatika arvude 0–999 ingliskeelsete väljendite süntaksianalüüsiks. Programm töötab ka tagurpidi, leides antud arvule vastava tekstikuju. Kuna arvude tekstikujusid genereeritakse ühekaupa alt-üles, siis pole programmi selline kasutamine eriti efektiivne. Veenduge selles.

```
lause --> aluserühm(Num), öeldiserühm(Num).
aluserühm(Num) --> alus(Num).
aluserühm(Num)--> täiend(Num), alus(Num).
öeldiserühm(Num) --> öeldis(Num).
öeldiserühm(Num) --> öeldis(Num), sihitis_määrus.
sihitis_määrus --> sihitis.
sihitis_määrus --> määrus.
sihitis_määrus --> sihitis, määrus.

alus(Num) --> nimisõna(Num).
öeldis(Num) --> tegusõna(Num).
täiend(Num) --> omadussõna(Num).
sihitis --> nimisõna(_).
määrus --> määrsõna.

nimisõna(ainsus) --> [mees].
nimisõna(mitmus) --> [mehed].
nimisõna(_) --> [kive].
nimisõna(_) --> [marju].
tegusõna(ainsus) --> [kannab]; [korjab].
tegusõna(mitmus) --> [kannavad]; [korjavad].
omadussõna(ainsus) --> [noor]; [tubli].
omadussõna(mitmus) --> [noored]; [tublid].
määrsõna --> [hästi].

?-lause([mehed,Teevad,Mida],[]).
Teevad = kannavad
Mida = kive
```

Joonis 13: Ainsuse ja mitmuse ühildamine.

```
lause --> aluserühm(Num), öeldiserühm(Num).
aluserühm(Num) --> alus(Num,_).
aluserühm(Num)--> täiend(Num,Case), alus(Num,Case).
öeldiserühm(Num) --> öeldis(Num).
öeldiserühm(Num) --> öeldis(Num), sihitis_määrus.
sihitis_määrus --> sihitiserühm.
sihitis_määrus --> määrus.
sihitis_määrus --> sihitiserühm, määrus.
sihitiserühm --> sihitis(_,_).
sihitiserühm --> täiend(Num,Case), sihitis(Num,Case).

alus(Num,nimetav) --> nimisõna(Num,nimetav).
öeldis(Num) --> tegusõna(Num).
täiend(Num,Case) --> omadussõna(Num,Case).
sihitis(Num,osastav) --> nimisõna(Num,osastav).
määrus --> määrsõna.

nimisõna(ainsus,nimetav) --> [mees]; [kivi]; [mari].
nimisõna(ainsus,osastav) --> [meest] ; [kivi]; [marja].
nimisõna(mitmus,nimetav) --> [mehed]; [kivid]; [marjad].
nimisõna(mitmus,osastav) --> [mehi]; [kive]; [marju].
tegusõna(ainsus) --> [kannab]; [korjab].
tegusõna(mitmus) --> [kannavad]; [korjavad].
omadussõna(ainsus,nimetav) --> [suur]; [noor]; [tubli].
omadussõna(ainsus,osastav) --> [suurt]; [noort]; [tubli].
omadussõna(mitmus,nimetav) --> [suured]; [noored]; [tublid].
omadussõna(mitmus,osastav) --> [suuri]; [noori]; [tublisid].
määrsõna --> [hästi].

?-lause([mees,kannab,Mida],[]).
Mida = meest
?-lause([Millised,mehed,kannavad,Milliseid,kive],[]).
Millised = suured
Milliseid = suuri
```

Joonis 14: Arvu ja käände ühildamine.

```
number(0) --> [zero].
number(N) --> xxx(N]).

xxx(N) --> digit(D), [hundred], rest_xxx(N1), {N is D*100+N1}.
xxx(N) --> xx(N).

rest_xxx(0) --> [].
rest_xxx(N) --> [and], xx(N).

xx(N) --> digit(N).
xx(N) --> teen(N).
xx(N) --> tens(T), rest_xx(N1), {N is T+N1}.

rest_xx(0) --> [].
rest_xx(N) --> digit(N).

digit(1) --> [one].       teen(10) --> [ten].
digit(2) --> [two].       teen(11) --> [eleven].
digit(3) --> [three].     teen(12) --> [twelve].
digit(4) --> [four].      teen(13) --> [thirteen].
digit(5) --> [five].      teen(14) --> [fourteen].
digit(6) --> [six].       teen(15) --> [fifteen].
digit(7) --> [seven].     teen(16) --> [sixteen].
digit(8) --> [eight].     teen(17) --> [seventeen].
digit(9) --> [nine].      teen(18) --> [eighteen].
                          teen(19) --> [nineteen].
tens(20) --> [twenty].
tens(30) --> [thirty].
tens(40) --> [fourty].
tens(50) --> [fifty].
tens(60) --> [sixty].
tens(70) --> [seventy].
tens(80) --> [eighty].
tens(90) --> [ninety].
```

Joonis 15: Ingliskeelsete arvude Horni grammatika.

Ülesanne 41 Kirjutage programm eestikeelsete arvude 0–999 süntaksianalüüsiks.

```
?-arv([kakskymmend,neli]).
?-kodeeri(Kakskymmend_neli,24).
Kakskymmend_neli = [kakskymmend,neli]
```

6.3 Grammatikate väärkasutused

Programmeerijad kasutavad meelsasti programmeerimiskeelte kõrvalefekte. Horni grammatikate mittesihipärase rakendamise näitena toome predikaadi, mis leiab listi elementide arvu.

```
count_off(N) --> [_], count_off(M), {N is M+1}.
count_off(0) --> [].
```

Tegelikult analüüsib see predikaat listi algusest etteantud arvu elemente, jättes listi ülejäänud elemendid vaatluse alt välja.

```
?-count_off(N,[a,b,c],[]).
N = 3
?-count_off(N,[a,b,c,d,e,f],[e,f]).
N = 4
?-count_off(2,[a,b,c,d,e],What]).
What = [c,d,e]
```

Prologi grammatikareeglite kasutamine antud juhul pigem eksitab kui selgitab predikaadi *count_off* töötamist. Parem oleks kirjutada otse

```
count_off(N,[_|Rest],Tail):-count_off(M,Rest,Tail), N is M+1.
count_off(0,Tail,Tail).
```

Paneme tähele, et reegel `count_off(0) -> []` ei määra tegevust listi lõpus. Reegel ütleb lihtsalt, et teda saab sooritada, ilma et analüüsitaks ühtegi listi elementi. Sellegipoolest on see ainuke käsk, mis on rakendatav tühilistile.

Ülesanne 42 Kirjutage grammatikareeglite abil programm, mis analüüsib läbi kõik listi elemendid ja leiab nende summa.

Ülesanne 43 Kas `dog -> "dog"` on korrektne grammatikareegel? Kui on, siis mida see tähendab ja kuidas seda kasutada?

Ülesanne 44 Kirjutage Horni grammatika, mis jaotab tähtede jada sõnadeks. Näiteks teisendab stringi `"this is it"` kujule `[this,is,it]`. Tähtede jada sõnadeks jaotamist võib vaadelda kui teatud liiki süntaktilist analüüsi (*parsing*).

6.4 Taust

Horni grammatikaid on hästi kirjeldatud Michael Covingtoni [18] ja Leon Sterlingi ja Ehud Shapiro [64] raamatutes. Viimasest pärineb ka ingliskeelsete arvude grammatika.

7 Metapredikaadid

Dünaamilised predikaadid. Lõikeoperaator ja eitus. Programmide teisendamine ja metainterpreteerimine.

7.1 Efektiivsus

Prolog on huvitav keel selle poolest, et ta vabastab programmeerija tehnilisest tööst nagu mäluhaldamisest, indekseerimisest jms. Seepärast sõltub programmi töökiirus konkreetsest Prologi realisatsioonist. Programmide kirjutamisel tuleb arvestada seda, et predikaatide definitsioone indekseeritakse tavaliselt esimese argumendi järgi. Paremates süsteemides on võimalik indekseerida ka teiste argumentide järgi. Tihti vaieldakse selle üle, kas dünaamilisi predikaate, mille definitsioone saab programmi täitmise käigus muuta, tasub kasutada või mitte. Teeme lihtsa eksperimendi. Defineerime uue dünaamilise kahekohalise predikaadi

```
?-dynamic esimene/2.
```

ja genereerime predikaadi mõlema argumendi järgi 100 000 erinevat fakti.

```
?-between(1,100000,X), assert(esimene(a,X)), fail.
?-between(1,100000,X), assert(esimene(X,a)), fail.
```

Selgub, et SWI-Prolog indekseerib predikaatide definitsioone vaikimisi esimese argumendi järgi. Sama tulemuse oleksime ilmselt saanud ka kasutades Prologi silumisvahendeid (*trace*).

```
?-time(esimene(a,99999)).
3 inferences in 37.62 seconds (0 Lips)
?-time(esimene(99999,a)).
3 inferences in 0.05 seconds (60 Lips)
```

Käsk *time* töötab nagu käsk *once*, täites päringut täpselt ühe korra ning väljastab päringu täitmiseks kulunud aja ja sekundis sooritatud resolutsioonisammude arvu (*lips*).

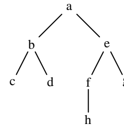
7.2 Dünaamilised andmebaasid

Kuigi programmi struktuuri muutmist selle täitmise käigus loetakse halvaks programmeerimistehnikaks, pakuvad käsud *assert* ja *retract* huvitavaid võimalusi. Nende abil saame oma käsutusse globaalsed muutujad — loendajad ja summaatorid — ning mugava vahendi programmi täitmise juhtimiseks (*flag*). Leiame näiteks teist järku predikaate *bagof*, *setof* ja *findall* kasutamata kahendpuu lehtede arvu. Esmalt kirjeldame, kuidas me puud läbime. Näiteks rekursiivselt postorderis: kõigepealt läbime puu vasakpoolse haru, siis parempoolse haru ja kõige lõpuks tegeleme puu juurega.

```
traverse(tree(_Element,Left,_Right)):-traverse(Left).
traverse(tree(_Element,_Left,Right)):-traverse(Right).
traverse(tree(_Element,leaf,leaf)).
```

Valime välja ka sobiva puu.

```
tree(tree(a,tree(b,tree(c,leaf,leaf),
                   tree(d,leaf,leaf)),
            tree(e,tree(f,tree(h,leaf,leaf),
                          leaf),
                   tree(g,leaf,leaf)))).
```



Dünaamilise loenduri *lehtede_arv*/1 abil loeme *fail*-tsükliga kokku, mitu korda jõuti puu läbimisel leheni.

```
number_of_leaves(Tree,N):-
    assert(lehtede_arv(0)),
    traverse(Tree),
    retract(lehtede_arv(N)),
    N1 is N+1,
    assert(lehtede_arv(N1)),
    fail
    ;
    lehtede_arv(N), write(lehtede_arv(N)).
```

Lehtede arvu väljastame disjunktsiooniga pärast kahendpuu kõigi harude läbimist.

```
?-tree(Tree), number_of_leaves(Tree,N).
```

Ülesanne 45 Muutke programmi, nii et see leiab puu kõigi tippude arvu.

Ülesanne 46 Koostage programm, mis trükib puu välja taanetega. Näiteks kujul

```
tree
  a
  tree
    b
    tree
      c
      leaf
      leaf
    tree
      d
      leaf
      leaf
  tree
    e
    tree
      f
      tree
        h
        leaf
        leaf
      leaf
    tree
      g
      leaf
      leaf
```

Ülesanne 47 Kirjutage programm, mis moodustab kahendpuust listi. (See on üks võimalus teist järku predikaatide *bagof*, *setof* ja *fi ndall* realiseerimiseks.)

7.3 Cut-operaator

Lõikeoperaator (*cut*) *user* elimineerib variantide läbivaatusest (*backtracking*) kõik talle reeglis või päringus eelnevad predikaadid. Vaatleme programmi

```
a:-b.
b:-d,!.
b.
d.
d:-fail.
```

Predikaadi *b* esimeses disjunktis esineva lõikeoperaatori (!) täitmine seisneb antud juhul selles, et ta eemaldab variantide läbivaatusest predikaadid *b* ja *d*. Lõikeoperaatori töö jälgimiseks trasseerime päringut `?-a.`

```
?-a.
?-b.
?-d,!.
?-!.
yes ;
no
```

Kõrvaldame nüüd predikaadi *b* definitsioonist lõikeoperaatori. Selgub, et ilma lõikeoperaatorita tehakse kaks resolutsioonisammu rohkem.

```
?-a.
?-b.
?-d.
yes ;
?-fail.
yes ;
no
```

Paremate Prologisüsteemide korral võib juhtuda, et ülejäänud lahendeid lihtsalt ei otsita, sest predikaadil *a* puuduvad argumendid. Sellisel juhul võime lisada *a*-le vajaliku argumendi ja täita päringu `?-a(X).`

Siit selgub lõikeoperaatori olemus. Deklaratiivses mõttes võib teda vaadelda kui predikaati, mis on alati tõene (*true*). Protseduraalses mõttes muudab ta aga otsingupuud, lõigates sealt välja definitsioonis talle eelnevatele predikaatide harud. Sõltuvalt sellest, kuidas niisugune väljalõikamine programmi tähendust muudab, liigitatakse lõikeoperaatori kasutused rohelisteks (*green*) ja punasteks (*red*). Punased kasutused on ohtlikud, sest nad muudavad progammi semantikat, lõigates välja osa lahendeid. Rohelised lõikeoperaatorid on aga ohutud (*safe*). Neid lisatakse programmidele töökiiruse tõstmiseks — predikaatide determineerimiseks. Kui me teame, et päringul on ainult üks vastus, siis saab lõikeoperaatoriga kõik järgnevad predikaadi definitsioonid variantide läbivaatusest välja lõigata (*committed choice*). Sellega vähendame kontrollide arvu ning hoiame kokku aega ja mälu. Olgu meil vaja sooritada kolme eri tegevust sõltuvalt sellest, kas tegu on mehe, naise või lapsega.

```
valik(X):-mees(X), !, ...
valik(X):-naine(X), !, ...
valik(X):-laps(X), !, ...
```

Kui me oleme tuvastanud, et tegemist on mehega, pole teisi variante enam tarvis kontrollida. Rohelise lõikeoperaatori võib kas ära jätta või asendada selle alati tõese predikaadiga *true*.

Punase lõikeoperaatoriga on asi keerulisem. Predikaat `max` ütleb järgmist: kui $X \leq Y$, siis on maksimaalne Y, muidu on maksimaalne X.

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

Eemaldades siit *cut*'i, saame lihtsalt vigase programmi. Kahjuks töötab programm ainult juhul, kui kolmandaks argumendiks on muutuja, s.t. kui ta töötab determineeritult, arvutades maksimaalse kahest esimesest argumendist. Kui kõik argumendid on määratud, saame aga reegli vasakul poole unifitseerimise tõttu vale vastuse.

```
?-max(2,5,2).
yes
```

Rohelise *cut*'i saame, kui lisame teisele disjunktile tarviliku eelduse.

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

Vasaku poole argumentide unifitseerimise vältimiseks võime kirjutada programmi ümber kujul

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

Roheliseks muutub lõikeoperaator alles siis, kui me lisame teisele reeglile kontrollid.

```
max(X,Y,Z):-X>Y, Z=X.
```

Vahel võib tarvis minna ka efektiivset *member*-predikaati, mis annab ainult ühe lahendi.

```
member(X,[X|Xs]):-!.
member(X,[Y|Ys]):-member(X,Ys).

?-member(X,[1,2,3]).
X = 1 ;
no
```

Kuna tegemist on standardpredikaadiga, siis on õigem lisada *cut* päringu lõppu.

```
?-member(X,[1,2,3]), !.
```

Lõikeoperaatori abil on lihtne realiseerida imperatiivsetest keeltest tuntud konstruktsioon *if_then_else*.

```
if_then_else(P,Q,R):-P, !, Q.
if_then_else(P,Q,R):-R.
```

Deklaratiivselt tähendab see, et tõesed on kas P ja Q või P eitus ja R. Protseduraalselt aga tõestame P; kui tõestamine õnnestus, siis tõestame Q, muidu tõestame R. Prologis kirjutatakse *if_then_else* tavaliselt kujul `P -> Q; R`.

Lõikeoperaatorit võib kasutada ka programmi teisendamiseks efektiivsele sabarekursiivsele kujule (*tail recursion optimization*). Kui on teada, et päringu $\leftarrow p_1, \ldots, p_n$ eelmised predikaadid omavad ühest lahendit, siis võib

viimase rekursiivse predikaadi p_n eraldada neist lõikeoperaatoriga. Paremad Prologisüsteemid teevad seda automaatselt. Sabarekursiivsed on näiteks standardpredikaadid *member*, *append* ja *repeat*, võimaldades kuitahes sügavat rekursiooni.

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

7.4 Eitus

Eitusega on loogilistes programmides hoopis isesugune suhe. Tegemist ei ole mitte eitusega klassikalise loogika mõttes, vaid selle teatava intuitsionistliku analoogiga: väide on väär, kui seda ei õnnestu tõestada. Puhtas Prologis (*pure Prolog*) ei ole üldse eitust (ja teisi loogikaväliseid vahendeid). Loogilised programmid pannakse kirja Horni reeglitena, mille vasakul pool asuv predikaat on esialgses disjunktis eitusega ning paremal asuvad eeldus-predikaadid ilma eituseta. Parempoolsetesse predikaatidesse on eitus lisatud alles hiljem (*not*, `\+`). Sama kehtib ka kvantorite kohta, mis esinevad Horni reeglites varjatud kujul: reegli vasaku poole muutujad seotakse üldisusekvantoriga ja ülejäänud muutujad eksistentsikvantoriga. Prologis esitatakse ainult positiivsed faktid, eeldades et kõik kirjeldamata faktid on väärad. Asi töötab, kui predikaatides ei ole muutujaid.

```
student(joe).
student(bill).
student(jim).
teacher(mary).
```

Saame teada, et Mary ei ole õpilane.

```
?-not student(mary).
yes
```

Eitust on lihtne realiseerida *cut* ja *fail* abil.

```
not X:-X, !, fail.
not X.
```

Kombinatsioon *cut-fail* ütleb lihtsalt, et lahendit ei tasu edasi otsida — lahend puudub.

Probleem tekib siis, kui päringus on vabu muutujaid. Näiteks järgmises programmis sõltub vastus predikaatide järjekorrast.

```
unmarried_student(X):-not married(X), student(X).
student(bill).
married(joe).

?-unmarried_student(X).
no
```

Me tahame teada, kas andmebaasis on õpilasi, kes ei ole abielus. Päringu trasseerimine meie *not* definitsiooniga selgitab, kuidas Prolog käsitleb eitust.

```
?-unmarried_student(X).
?-not married(X), student(X).
?-married(X), !, fail, student(X).
?-!, fail, student(joe).
?-fail, student(joe).
no
```

Cut lõikab predikaadi *not* teise definitsiooni otsingupuust välja. Õige vastuse saame, kirjutades päringud teises järjekorras.

```
?-student(X), not married(X).
?-not married(bill).
?-married(bill), !, fail.
X = bill
```

Cut'ini praegu ei jõuta ning täidetakse *not* teine definitsioon.

Eitusele on iseloomulik see, et vastuseni jõudmisel on päringu vabad muutujad väärtustamata. Predikaat *verify* erinebki metamuutujast ja predikaadist *call* selle poolest, et ta ei riku ära muutujate väärtuseid.

```
verify(Goal):-not not Goal.
```

Predikaati *verify* võib näiteks kasutada eelkontrollide tegemiseks ristsõna koostamisel.

<table>
<tr><td>a</td><td>h</td><td>i</td><td rowspan="3">=</td><td>C11</td><td>C12</td><td>C13</td></tr>
<tr><td>r</td><td>a</td><td>t</td><td>C21</td><td>C22</td><td>C23</td></tr>
<tr><td>k</td><td>e</td><td>s</td><td>C31</td><td>C32</td><td>C33</td></tr>
</table>

Sõnad esitame tähtede loeteludena.

```
word(r,a,t).
word(a,h,i).
word(a,r,k).
...
```

Kandes esimese sõna "rat" tabelisse, on liigse töö vältimiseks hea enne järgmise sõna täitmist kohe kontrollida, kas sõnaraamatus leiduvad esimese sõnaga ristuvad sõnad. Seejärel asume teise sõna täitmisele.

```
?-word(C11,C12,C13),
  verify(word(C11,C21,C31)),
  verify(word(C12,C22,C32)),
  verify(word(C13,C23,C33)),
  word(C11,C21,C31).
C11 = a, C12 = h, C13 = i
C21 = h, C31 = i
C22 = C22, C32 = C32, C23 = C23, C33 = C33
```

Antud juhul sõna "rat" ei sobinudki ja esimesse ritta kanti "ahi". Sama sõna sobis ka esimesse veergu. Ülejäänud lahtrid jäid tühjaks, kuigi veergude sõnad kontrolliti esimese rea täitmise järel ära.

Ülesanne 48 Konstrueerige Prologiga lihtne ristsõna.

7.5 Metaprogrammeerimine

Metaprogrammeerimise all mõistetakse programmidega kui andmetega manipuleerimist. Metaloogilised predikaadid on näiteks *clause*, *functor*, *arg* ja =.. . Erinevalt ekstraloogilistest predikaatidest (*var*, *integer*, *atom*, *read*, *write*, *tell*, *get*, *system*) ei saa neid esitada faktide loeteluna.

Prologi termid kujutavad endast sümbolite suluavaldisi. Ka Prologi programm ise on teatud liiki suluavaldis, s.t. Prologi programm on term. Termidega manipuleerimiseks võib kasutada käskusid *functor*/3 ja *arg*/3. Neist esimene annab sulgudele eelneva aatomi ja argumentide arvu ning teine leiab sulgude seest *n*-nda komadega eraldatud argumendi.

```
?-functor(member(a,[a,b,c]),member,2).
?-arg(N,member(a,[a,b,c]),[a,b,c]).
N = 2
```

Predikaatidel *functor* ja *arg* on huvitav omadus — nende abil saab mitte ainult terme lahti lõhkuda, vaid ka uusi terme luua. Oletame, et me tahame lisada kuupäevatermile neljanda atribuudi — nädalapäeva.

```
?-add_argument(kolmapäev,date(5,märts,1997),Term).
Term = date(5,märts,1997,kolmapäev)
```

Selleks moodustame sama funktoriga uue termi, võttes selle viimaseks argumendiks nädalapäeva ja kopeerides eelmised argumendid vanast termist.

```
add_argument(X,Y,Z):-
    functor(Y,F,N),
    N1 is N+1,
    functor(Z,F,N1),
    arg(N1,Z,X),
    args(1,N,Y,Z).

args(M,N,Y,Z):-
    M=<N,
    arg(M,Y,X),
    arg(M,Z,X),
    M1 is M+1,
    args(M1,N,Y,Z).
args(M,N,_,_):-M is N+1.
```

Sama võimaldab ka operaator =.., mis teisendab vasakul asuva n argumendiga termi $n+1$ elemendiliseks listiks ja vastupidi — listi vastavaks termiks.

```
add_argument(X,Y,Z):-
    Y=..List,               % [date,5,märts,1997]
    append(List,[X],List2), % [date,5,märts,1997,kolmapäev]
    Z=..List2.              % date(5,märts,1997,kolmapäev)
```

Ülesanne 49 Realiseerige predikaat *map_list*, mis rakendab oma teiseks argumendiks olevat funktsiooni listi kõikidele elementidele. Kirjutage predikaadid *succ* ja *square*.

```
?-map_list([1,2,3],succ,[2,3,4]).
?-map_list([1,2,3],square,[1,4,9]).
```

Prologis on ka vahendid, mis võimaldavad teisendada programme. Mällu loetud programmi disjunkte saab lugeda käsuga `clause`, mis väljastab reegli *Head:–Body* parempoolse osa (*body*). Näiteks programmi

```
member(X,[X|Xs]).
member(X,[Y|Ys]):-
    member(X,Ys).
```

korral on tulemuseks

```
?-clause(member(X,Ys),Body).
Body = true ;
Body = member(X,Ys1)
```

Kuna *faktidel* eeldused puuduvad, väljastatakse nende korral predikaat *true*. Programmi lõppu saab lisada uusi disjunkte lisada käsuga *assert*. Reegli lisamisel peab aga vasak pool (*head*) olema piisavalt määratud. Metamuutujate abil on lihtne ka koostada uusi päringuid.

```
?-functor(T,member,2), T.
T = member(X,[X|Xs]) ;
T = member(X,[Y,X|Xs]) ;
T = member(X,[Y1,Y2,X|Xs]) ;
...
```

Juhul kui metamuutujas on predikaadi nimi määramata, saame vastava veateate.

```
?-T, functor(T,member,2).
```

Prologisüsteemides, mis toetavad päringute edasilükkamist (*delay*) jäetakse päring ootele, täites selle alles pärast piisavat määratlemist. Näiteks Eclipse'is käivitatakse päringute edasilükkamine käsuga

```
:-set_flag(coroutine,on).
```

Nii võib lahti saada ka aritmeetikavigadest.

```
?-X>0, X=1.
X = 1
```

Harilik Prolog annab esimese päringu korral veateate, sest ta ei suuda välja arvutada seose vasaku poole väärtust. Viivitamismehhanismiga süsteem jätab aga esimese päringu vahele ja täidab kõigepealt teise päringu, omistades muutujale `X` väärtuse 1. Seejärel saab täita ka esimese päringu.

7.6 Teist järku programmeerimine

Käsuga `assert` saab realiseerida hulki moodustavad predikaadid *bagof*, *setof* ja *findall*. Prolog annab tavaliselt päringu vastused ühekaupa.

```
?-member(X,[a,b,c]).
X = a;
X = b;
X = c;
no
```

Käsk `bagof` annab päringule vastava lahendite loendi.

```
?-bagof(X,member(X,[a,b,c]),List).
List = [a,b,c]
```

Käsk `setof` järjestab lahendid, kõrvaldades kordused, ning `findall` leiab lahendid esimeseks argumendiks olev muutuja järgi, sidudes teised päringu muutujad eksistentsikvantoriga `^`. Näiteks *bagof* fikseerib vaba muutuja `Y` esimese väärtuse, andmata kõiki lahendeid korraga.

```
?-bagof(X,append(Y,X,[1,2,3,4]),List).
List = [[1,2,3,4]];
List = [[2,3,4]];
List = [[3,4]];
List = [[4]];
List = [[]];
no
```

Sidudes `Y` eksistentsikvantoriga, saame aga soovitud tulemuse.

```
?-bagof(X,Y^append(Y,X,[1,2,3,4]),List).
List = [[1,2,3,4], [2,3,4], [3,4], [4], []]
```

Käsk `setof` annab lahendid järjestatult.

```
?-setof(X,Y^append(Y,X,[1,2,3,4]),List).
List = [[], [1,2,3,4], [2,3,4], [3,4], [4]]
```

Käsu `findall` korral pole vahet, kas `Y` on seotud eksistentsikvantoriga või mitte.

```
?-findall(X,append(Y,X,[1,2,3,4]),List).
List = [[1,2,3,4], [2,3,4], [3,4], [4], []]
```

Teist järku loogikas saab realiseerida predikaadid *not* ja *var*. Eitust tõlgendatakse päringu väärusena, s.t. päringul puuduvad lahendid.

```
not X :- setof(Y,X,[]).
```

Term X on aga vaba muutuja, kui päringul

```
X=1; X=2
```

on kaks lahendit.

Ülesanne 50 Realiseerige *var*/1, kasutades Prologi teist järku predikaate.

7.7 Programmi metainterpreteerimine

Prologis on ka vahendid programmide silumiseks ja nende täitmise juhtimiseks. Seda nimetatakse programmide metainterpreteerimiseks. Lisaks programmide täitmise standardsele ülevalt-alla vasakult-paremale strateegiale saab lihtsalt realiseerida ka teisi strateegiaid.

Asume siis realiseerima Prologi metainterpretaatorit. Lihtsaima metainterpretaatori saame, kasutades metamuutujat.

```
solve(A):-A.
```

Puhta Prologi metainterpretaator kasutab käsku `clause`, mis lahutab Prologi reeglid vasakuks ja paremaks pooleks.

```
solve(true):-!.
solve((A,B)):-!, solve(A), solve(B).
solve(A):-clause(A,B), solve(B).
```

Trasseerime sellega predikaati *member*.

```
?-solve(member(X,[a,b,c])).
?-clause(member(X,[a,b,c]),B), solve(B).
?-solve(true).
X = a ;
?-solve(member(X,[b,c])).
?-clause(member(X,[b,c]),B), solve(B).
?-solve(true).
X = b ;
?-solve(member(X,[c])).
?-clause(member(X,[c]),B), solve(B).
?-solve(true).
X = c;
?-solve(member(X,[])).
?-clause(member(X,[]),B), solve(B).
no
```

Käsk `clause` leiab predikaadile vastava reegli parema poole ja asub selle täitmisele. Fakti korral on paremaks pooleks lihtsalt predikaat *true*. Liitpäringu korral täidetakse aga selle iga päring eraldi.

Puhta Prologi metainterpretaatorit on lihtne laiendada kogu Prologile. Selleks selgitame, mida teha süsteemsete predikaatide, eituse ja lõikeoperaatori korral.

```
solve(true):-!.
solve((A,B)):-!, solve(A), solve(B).
solve(!):-!, !(reduce(A)). /* Ancestor cut */
solve(not A):-!, not solve(A).
solve(setof(X,Goal,Xs)):-!, setof(X,solve(Goal),Xs).
solve(A):-system(A), !, A.
solve(A):-reduce(A).

reduce(A):-clause(A,B), solve(B).

system(A is B).  system(A < B).
system(read(X)).  system(write(X)).
system(integer(X)).  system(functor(T,F,N)).
system(clause(A,B)).  system(system(X)).
```

Lõikeoperaatori jaoks toome sisse predikaadi *reduce*: siis saame *ancestor cut*'i abil otsingupuus õigesse kohta tagasi pöörduda.

Ülesanne 51 Prolog otsib lahendit sügavuti, otsides lahendit kõigepealt otsingupuu (*SLD-tree*) vasakpoolseimast harust, seejärel selle naaberharust jne. Millise tulemuse annab programm *loop*? Joonistage vastav otsingupuu.

```
loop:-loop.
loop.

?-loop.
```

Ülesanne 52 Programm *loop* annab lahendi laiuti otsides: vaadates otsingupuu harusid läbi paralleelselt. Realiseerige laiuti otsimise strateegia. Võrrelge sügavuti ja laiuti otsimise strateegiate efektiivsust.

7.8 Taust

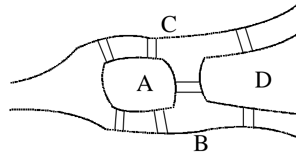
Metapredikaatide materjal on võetud põhiliselt Sterlingi ja Shapiro raamatust.

8 Keerdülesannete lahendamine

Tee otsimine graafis ja vastuseruumi ühestamine.

8.1 Jalutuskäik Königsbergis

Eriti huvitav on Prologiga keerdülesannete lahendamine. Võtame näiteks Königsbergi sildade probleemi. Ülesandeks on koostada pühapäevane ja-



Joonis 16: Königsbergi sillad.

lutuskäik (*Spaziergang*), mille käigus on vaja ületada seitse üle Pregeli jõe viivat silda — igaühte täpselt üks kord —, nii et lõpuks jõutakse koju tagasi. Leonhard Euler (1707–1783) tõestas, et sellist ringkäiku ei leidu.

Esimene võimalus on genereerida sarnaselt sugulaste märgendamisega kõikvõimalikud seitmesillased teed, mis jõuavad samasse kohta tagasi ja kus ükski sild ei kordu.

```
?-relative(7,1,Who,How), length(How,8),
  last(1,How), different(How).
```

Efektiivsem on genereerida listi $[1,\ldots,7]$ kõikvõimalikud permutatsioonid, mis määravad ära sildade ületamise järjekorra. Seejärel sulgeme tee, lisades listi lõppu arvu 1 ja kontrollime, kas tee on läbitav, s.t. kas kahel järjestikusel sillal on ühine ots.

```
euler_walk:-
    permutation([1,2,3,4,5,6,7],Path),
    last(1,Path),
    Spaziergang=[1|Path],
    euler_walk(Spaziergang),
    write(Spaziergang).
```

Alustame sildade märgendamisest (*labeling*).

```
bridge(a,c,1).
bridge(a,c,2).
bridge(a,b,3).
bridge(a,b,4).
bridge(d,a,5).
```

```
bridge(d,c,6).
bridge(d,b,7).
```

Seejärel kontrollime, kas igal kahel järjestikusel sillal on ühine otspunkt.

```
euler_walk([_]).
euler_walk([X,Y|Path]):-
    bridge(AX,BX,X),
    bridge(AY,BY,Y),
    ( AX=AY
    ; AX=BY
    ; BX=AY
    ; BX=BY),
    euler_walk([Y|Path]).
```

Tulemuseks on aga vale vastus.

```
?-euler_walk.
[1,2,3,4,5,7,6,1].
```

Ülesanne 53 Parandage programmi, nii et tulemuseks on õige vastus —sellist jalutuskäiku ei leidu.

8.2 Sõprade probleem

Võtame nüüd järgmise keerdülesande. Kolm sõpra said programmeerimisvõistlusel esimese, teise ja kolmanda koha. Igaühel neist on erinev eesnimi, spordiala ja kodakondsus (*nationality*). Teada on faktid (*clue*):

1. Michael mängib korvpalli ja ta oli parem kui ameeriklane.
2. Simon on israellane ja ta oli parem kui tennisemängija.
3. Kriketimängija tuli esimeseks.

Vaja on vastata järgmistele küsimustele (*query*):

- Kes on austraallane?
- Millist sporti teeb Richard?

Tegemist on tavalise nuputamisülesandega, millel on täpselt üks lahend. Nende lahendamisel on tähtis valida sobiv struktuur, mis lihtsustab oluliselt ülesande lahendamist. Seejärel omistame struktuuri muutujatele võimalikke väärtusi, tehes seda seni, kuni leiame väärtustuse, mis rahuldab üheaegselt nii fakte kui ka küsimusi. Seda võib tinglikult nimetada struktuuri ühestamiseks. Antud juhul on struktuuriks 3×3 tabel, mille read vastavad võistlusel saadud kohtadele ja veergudes on võistlejate atribuudid: nimi, kodakondsus ja spordiala.

```
structure([friend (_,_,_),
           friend (_,_,_),
           friend (_,_,_)]).

nam(friend (A,_,_),A). %name/2 ei tohi muuta!
nationality(friend(_,B,_),B).
sport(friend(_,_,C),C).
```

Struktuuri põhjal paneme paika järjestuse.

```
did_better(A,B,[A,B,_]).
did_better(A,C,[A,_,C]).
did_better(B,C,[_,B,C]).

first([X|_],X).
```

Sõprade probleemi lahendamiseks valime nüüd struktuurist ühekaupa võistlejaid ning ühestame faktide ja küsimuste abil struktuuri muutujate väärtused.

```
puzzle:-
    structure(Friends),
    did_better(M1C1,M2C1,Friends),
        nam(M1C1,michael),
        sport(M1C1,basketball),
        nationality(M2C1,american),
    did_better(M1C2,M2C2,Friends),
        nam(M1C2,simon),
        nationality(M1C2,israeli),
        sport(M2C2,tennis),
    first(Friends,ManClue3),
        sport(ManClue3,cricket),
    member(Q1,Friends),
        nam(Q1,Name),
```

```
        nationality(Q1,australian),
    member(Q2,Friends),
        nam(Q2,richard),
        sport(Q2,Sport),
    write('Austraallane on '), write(Name),
    nl,
    write('Richard mängib '), write(Sport),
    nl.
```

```
?-puzzle
```

Ülesanne 54 Lahendage Prologiga järgmine ülesanne. Kaks sõpra elavad eri linnades: üks elab Tartus ja teine elab Pärnus. Tõnu elab Tartus. Kus elab Priit?

Lahendage sama ülesanne kolme sõbra ja kolme linna korral.

Ülesanne 55 Proovige lahendada sama probleem eituse abil.

Ülesanne 56 (Neljavärviprobleem). Värvige maakaart nelja värviga, nii et naaberriigid on erinevat värvi.

Ülesanne 57 Kandke $n \times n$ malelauale n lippu, nii et ükski lipp ei asuks teise lipu tules.

8.3 Taust

Königsbergi sildade probleem on võetud Oystein Ore raamatust [55] ja sõprade probleem Leon Sterlingi ja Ehud Shapiro raamatust [64]. Sõprade ja linnade ülesannet püüdsime tagajärjetult lahendada 1993. a. loogilise programmeerimise kursuses. Neljavärviprobleem on klassikaline matemaatikaülesanne, millega pistsid 20. sajandi alguses rinda ka eesti matemaatikud Jaan Sarv, Jüri Nuut, Hermann Jaakson ja Edgar Krahn [37]. Lippude probleem on standardne programmeerimisülesanne, mida on mõnus lahendada $n = 4$ ja $n = 8$ korral. Tänapäeval lahendatakse seda edukalt tunduvalt suuremate arvude korral.

Ülesannete lahendused

Ülesanne 1 (vihje). Kasutage eitust kujul `X\=Y`.

Ülesanne 1. Keegi on kellegi vend, kui ta on sama ema laps ja ta on meessoost.

```
brother(Brother):-brother(_,Brother).
brother(Child,Brother):-
    mother(Child,Mother),
    mother(Brother,Mother),
    male(Brother),
    Child\=Brother.
```

Ülesanne 2. Kirjeldame puuduvad mõisted: laps, vend ja õde.

```
child(Parent,Child):-parent(Child,Parent).
daughter(Parent,Child):-child(Parent,Child), female(Child).
son(Parent,Child):-child(Parent,Child), male(Child).

child(Man,R,tytar):-daughter(Man,R).
child(Man,R,poeg):-son(Man,R).

brother(Man,R,vend):-brother(Man,R).
sister(Man,R,ode):-sister(Man,R).
```

Ülesanne 3.

```
:-op(900,xf,on_mees).
john on_mees.
```

Ülesanne 5.

```
% wff(X) on tõene, kui
%       X on lausearvutuse valem
wff(X):-atom(X). %numbreid ei luba!
wff(true).
wff(false).
wff(~X):-wff(X).
wff(X and Y):-wff(X), wff(Y).
wff(X or Y):-wff(X), wff(Y).
wff(X -> Y):-wff(X), wff(Y).
wff(X <-> Y):-wff(X), wff(Y).
```

Ülesanne 6.

```
% valid(X) on tõene, kui
%          X on loogiliselt tõene lausearvutuse valem
valid(true).
valid(~X):-invalid(X).
valid(X and Y):-valid(X), valid(X).
valid(X or Y):-valid(X) ; valid(X).
valid(X -> Y):-invalid(X) ; valid(X).
valid(X <-> Y):-valid(X -> Y) ; valid(Y -> X).

% invalid(X) on tõene, kui
%            X on loogiliselt väär lausearvutuse valem
invalid(false).
invalid(~X).
invalid(X and Y):-invalid(X) ; invalid(Y).
invalid(X or Y):-invalid(X), invalid(Y).
invalid(X -> Y):-valid(X) ; invalid(Y).
invalid(X <-> Y):-valid(X), invalid(Y) ; invalid(X), valid(Y).

?-satisfiable(true and true).
?-invalid(true and false).
```

Ülesanne 10 (vihje). Listi pikkuse saab kindlaks teha käsuga *length*/2.

Ülesanne 10.

```
?-setof(Ema,relative(8,fred,linda,Ema),List), length(List,N).
N=3046
?-setof(Ema,relative(4,fred,linda,Ema),List), length(List,N).
List = [
 ema,
[ema, [abikaasa, abikaasa]],
[ema, [abikaasa, [poeg, ema]]],
[ema, [ema, tytar]],
[ema, [ema, [abikaasa, tytar]]],
[ema, [ema, [poeg, ode]]],
[ema, [isa, tytar]],
[ema, [isa, [abikaasa, tytar]]],
[ema, [isa, [poeg, ode]]],
[ema, [poeg, ema]],
[ema, [poeg, [isa, abikaasa]]],
[ema, [vend, ode]],
[ema, [vend, [ema, tytar]]],
[ema, [vend, [isa, tytar]]],
[isa, abikaasa],
[isa, [abikaasa, [abikaasa, abikaasa]]],
[isa, [abikaasa, [ema, tytar]]],
[isa, [abikaasa, [isa, tytar]]],
```

```
[isa, [abikaasa, [poeg, ema]]],
[isa, [abikaasa, [vend, ode]]],
[isa, [poeg, ema]],
[isa, [poeg, [isa, abikaasa]]]
]
N = 22
```

Ülesanne 11.

```
last(X,[X]).
last(X,[_|Xs]):-last(X,Xs).
```

Ülesanne 13.

```
ordered([]).
ordered([_]).
ordered([X,Y|Ys]):-
    X=<Y,
    ordered([Y|Ys]).
```

Ülesanne 15 (vihje). Kasutage süsteemset predikaati *name*/2.

Ülesanne 16 (vihje). Anumatega saab sooritada kolme operatsiooni: anumat täita, anumat tühjendada ja kallata ühe anuma sisu teise anumasse. Probleemi lahendamiseks tuleb leida neile operatsioonidele vastav seisude jada:

$$seis(seitse(0), viis(0)),$$
$$seis(seitse(7), viis(0)),$$
$$\ldots$$

7l	5l	Operatsioon
0	0	algseis
7	0	täida seitse
2	5	kalla seitse viide
2	0	tühjenda viis
0	2	kalla seitse viide
7	2	täida seitse
4	5	kalla seitse viide

Ülesanne 16.

```
pour([seis(seven(M),five(4))|[]]).
pour([seis(seven(M),five(0))|Solution]):-
    Solution=[seis(seven(M),five(5))|Solution2],
    pour(Solution).
```

```
pour([seis(seven(7),five(N))|Solution]):-
    Solution=[seis(seven(0),five(N))|Solution2],
    pour(Solution).
pour([seis(seven(M),five(N))|Solution]):-
    M=\=7, N=\=0,
    R is M+N,
    minimum(R,7,R1),
    C is R1-M,
    M1 is M+C, N1 is N-C,
    Solution=[seis(seven(M1),five(N1))|Solution2],
    pour(Solution).

minimum(X,Y,X):-X =< Y.
minimum(X,Y,Y):-X > Y.

?-pour([seis(seven(0),five(0))|Solution]).
```

Ülesanne 24.

```
?-expand_term((p --> {write('Töötlen sest-et')}, [sest,et]), Tulemus).
Tulemus = p(_G605, _G608):-(write('Töötlen sest-et'), _G626=_G605),
          append([sest, et], _G608, _G626)
```

Ülesanne 34.

```
eliza:-
    repeat_stream,
    read_words(Sentence), nl,
    list_to_sentence(Sentence,Patient), write(Patient), nl,
    eliza(Sentence,Eliza),
    list_to_sentence(Eliza,Reply), write('Eliza: '), write(Reply), nl,
    fail.
eliza_file:-
    see(eliza), % sisendfail 'eliza'
    eliza;
    seen. % sulgeb streami

% Eliza's reply to patient's sentence
eliza(Sentence,Reply):-
    keyword(Word,Reply),
    (atomic(Word)
    ->member(Word,Sentence)
    ; sublist(Word,Sentence)
    ), !.
eliza(Sentence,Reply):-
    transform(C,_RC),
```

```
    member(C,Sentence),
    !,
    translate(Sentence,Reply).
eliza(_Sentence,['Does',that,have,anything,to,do,with,the,fact,
                 that,your,boyfriend,made,you,come,here]).

transform('I',you).
transform(am,are).
transform('My','Your').
transform(my,your).
transform(me,you).

keyword([afraid,of,me],
    ['Does',it,please,you,to,believe,'I',am,afraid,of,you]).
keyword(aggressive,
    ['What',make,you,think,'I',am,not,very,aggressive]).
keyword(alike,['In',what,way]).
keyword(argue,['Why',do,you,think,'I','don''t',argue,with,you]).
keyword(bugging,['Can',you,think,of,a,specific,example]).
keyword(care,['Who',else,in,your,family,takes,care,of,you]).
keyword(depressed,['I',am,sorry,to,hear,you,are,depressed]).
keyword(everybody,
    ['What',else,comes,to,mind,when,you,think,of,your,father]).
keyword(help,['What',would,it,mean,to,you,if,you,got,some,help]).
keyword(like,['What',resemblance,do,you,see]).
keyword(mother,['Tell',me,more,about,your,family]).
keyword(unhappy,
    ['Do',you,think,coming,here,will,help,you,not,to,be,unhappy]).

translate([],[]).
translate(['.'],[]):-!. %punkt lõpust maha!
translate(['Well',',',my|Xs],['Your'|Ys]):- %Well, algusest maha!
    !, translate(Xs,Ys).
translate([C|Xs],[RC|Ys]):-
    transform(C,RC),
    translate(Xs,Ys).
translate([X|Xs],[X|Ys]):-
    \+ transform(X,_),
    translate(Xs,Ys).

/*?-sublist([2,3],[1,2,3,4]).*/
sublist(Xs,Ys):-
    prefix(Xs,Ys).
sublist(Xs,[_|Ys]):-
    sublist(Xs,Ys).
/*?-prefix([2,3],[2,3,4]).*/
prefix([],_).
```

```
prefix([X|Xs],[X|Ys]):-
    prefix(Xs,Ys).

% replace(List,C,RC,List2)
% List2 is the result of replacing symbol C with RC in List
replace([],_,_,[]).
replace([C|Xs],C,RC,[RC|Xs2]):-
    replace(Xs,C,RC,Xs2).
replace([X|Xs],C,RC,[X|Xs2]):-
    X\==C,
    replace(Xs,C,RC,Xs2).

/*?-list_to_sentence(['Men',are,all,alike,'.'],
     'Men are all alike.').*/
list_to_sentence(List,Sentence):-
    list_to_sentencelist(List,List2),
    concat_atom(List2,Sentence).
list_to_sentencelist([],[]).
list_to_sentencelist([X],[X]).
list_to_sentencelist([X,Y|XYs],[X|XYs2]):-
    atom_chars(Y,[CY]),
    punctuation(CY),
    !,
    list_to_sentencelist([Y|XYs],XYs2).
list_to_sentencelist([X,Y|XYs],[X,' '|XYs2]):-
    list_to_sentencelist([Y|XYs],XYs2).

?-eliza.        % sisendist ridade kaupa
?-eliza_file.   % failist 'eliza' ridade kaupa
```

Ülesanne 35.

```
% wine(Name,Region,Characteristics,Color)
%     veini nimi, piirkond, iseloomustus ja värv
wine('Pouilly-Fuisse','Saone-et-Loire','dry and heady','Red').
wine('Gewurztraminer','Alsasce','sweet and light','White').
wine('St. Emilion','Bordeau','full-bodied and dark','Red').
wine('Graves','Bordeau','not too sweet','White').
wine('Sauternes','Bordeau','sweet and fruity','White').
wine('Yonne','Burgundy','light and subtle','White').
wine('Muscadet','Loire','pale and delicate','White').
```

Ülesanne 42.

```
count_sum(N,[E|Rest],Tail):-count_sum(M,Rest,Tail), N is M+E.
count_sum(0,Tail,Tail).
```

Ülesanne 47.

```
% preorder(Tree,Pre) <-
%   Pre is a preorder traversal of the binary tree Tree.
preorder(tree(X,L,R),Xs):-
    preorder(L,Ls), preorder(R,Rs), append([X|Ls],Rs,Xs).
preorder(void,[]).
```

Ülesanne 49.

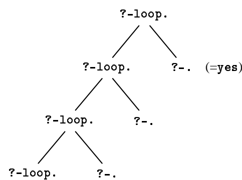
```
map_list([],_,[]).
map_list([X|Xs],Func,[Y|Ys]):-
    Goal=..[Func,X,Y],
    Goal,
    map_list(Xs,Func,Ys).

succ(X,Y):-Y is X+1.
square(X,Y):-Y is X*X.
```

Ülesanne 50.

```
var(X):-setof(X,(X=1; X=2),List), length(List,2).
```

Ülesanne 51.



Prolog ei leia lahendit, sest otsingupuu vasakpoolne haru on lõpmatu.

Ülesanne 52 (vihje). Metainterpreteerige programmi listi abil. Alguses koosneb list esialgsest päringust. Laiuti otsimiseks eemaldage igal sammul listist esimene element, leidke kõik päringud, mis on sellest saadavad ühe resolutsioonisammuga, ja kirjutage saadud päringud listi lõppu. (Niisugust loendit kutsutakse arvutiteaduses järjekorraks (*queue*) või FIFO-magasiniks (*fi rst-in-fi rst-out*).)

Ülesanne 54.

```
structure([friend(N1,T1),friend(N2,T2)]):-
    T1='Tartu',
    T2='Pärnu'.
structure3([friend(N1,T1),friend(N2,T2),friend(N3,T3)]):-
```

```
    T1='Tartu',
    T2='Pärnu',
    T3='Tallinn'.

town('Tartu').
town('Pärnu').
town('Tallinn').

nam(friend(Name,_Town),Name). %name ei saa kasutada!
town(friend(Name,Town),Town):-town(Town).

puzzle:-
    structure(Friends),
    member(Man1,Friends),
    nam(Man1,'Tõnu'),
    town(Man1,'Tartu'),
    member(Man2,Friends),
    nam(Man2,'Priit'),
    town(Man2,Town),
    write('Priit elab: '), write(Town), write('s').
puzzle3:-
    structure3(Friends),
    member(Man1,Friends),
    nam(Man1,'Tõnu'),
    town(Man1,'Tartu'),
    member(Man2,Friends),
    nam(Man2,'Priit'),
    town(Man2,Town),
    member(Man3,Friends),
    nam(Man3,'Jaak'),
    town(Man3,Town3),
    write('Priit elab: '), write(Town), write('s'), nl,
    write('Jaak elab: '), write(Town3), write('s').

?-puzzle.
?-puzzle3.
```

Ülesanne 56.

```
color(1).   % +-+---+            a - b
color(2).   % |A|  B|            | / |
color(3).   % +-+---+            c - d
            % | C| D|
country(a). % +--+--+
country(b).
country(c).
country(d).
```

```
neighbour(a,b).  neighbour(a,c).
neighbour(b,a).  neighbour(b,c).  neighbour(b,d).
neighbour(c,d).

structure([color(a,X),color(b,Y),color(c,Z),color(d,W)]).

coloring(S):-structure(S), colors(S), check(S).

colors([]).
colors([color(X,CX)|S]):-
    color(CX),
    colors(S).

check(S):-
    not (
    member(color(X,CX),S), member(color(Y,CY),S),
    X\=Y,
    neighbour(X,Y),
    CX=CY).

?-coloring(S).
```

Prologi standardpredikaadid[2]

Tüübid:

atom(Atom)	Atom on aatom.
integer(I)	I on täisarv.
atomic(A)	A on aatom või täisarv.
constant(X)	Sama mis atomic.
functor(St,F,A)	F on termi St funktor ja A on tema aarsus.
arg(N,S,Sn)	Sn on S'i N's argument.
var(V)	V on muutuja.
nonvar(C)	C ei ole muutuja.

Programmi teisendamine:

assert(Clause)	Lisa disjunkt programmi lõppu.
asserta(Clause)	Lisa disjunkt programmi algusesse.
assertz(Clause)	Sama mis assert.
retract(Clause)	Kustuta Clause'iga unifi tseeruv disjunkt.
abolish(F,A)	Kustuta kõik disjunktid funktoriga F ja aarsusega A.
consult(File)	Loe sisse programmifail File.
reconsult(File)	Sama mis consult, kuid programmist kustutatakse vanade protseduuride disjunktid.
clause(G,B)	B on G-ga unifi tseeruva disjunkti keha,
listing	Trüki aktiivne progamm.
listing(Name)	Trüki välja programmi predikaadid nimega Name.

Programmi sisselugemine:

[fi le1,fi le2,...]	Sama mis reconsult/1.
[+fi le1,+fi le2,...]	Sama mis consult/1.
[user] või [+user]	Protseduuride sisestamine klaviaaturilt.

[2]Loetelu on võetud Leon Sterlingi ja Ehud Shapiro raamatu '*The art of Prolog: advanced programming techniques*" esimesest väljaandest (MIT Press, 1986). Vt. samuti Prolog-10 manuaali [9].

Sisend/väljund:

read(T)	T on järgmine punktiga lõppev term aktiivses sisendvoos, millele järgneb reavahetus või tühik.
sread(T,Vs)	Loe term T, Vs on paaride loend; muutuja nimi ja muutuja väärtus.
write(T)	Kirjuta T aktiivsesse väljundfaili.
writeq(T)	Kirjuta T, kasutades vajaduse korral apostroofe (*quotes*).
display(T)	Väljasta T terminaalile.
displayq(T)	Väljasta T terminaalile, kasutades vajaduse korral apostroofe.
displayl(L)	Väljasta objektide list.
print(T)	"Pretty printingu"realisatsioon; kirjutage oma "portray(T)".
see(File)	File on uus akviivne sisendfail.
seeing(File)	File unifi tseeritakse aktiivse sisendfaili nimega.
seen	Suleb aktiivse sisendvoo.
tell(File)	Olgu File aktiivne väljundvoog.
telling(File)	File unifi tseeritakse aktiivse väljundfaili nimega.
told	Sule aktiivne väljundfail.
flush	Tühjenda aktiivse väljundfaili puhver.
get(C)	Loe aktiivsest sisendvoost trükitav sümbol.
get0(C)	Loe aktiivsest sisendvoost sümbol.
put(C)	Kirjuta väljundfaili sümbol C.
tab(N)	Trüki väljundfaili N tühikut.
nl	Trüki väljundfaili reavahetuse sümbol.
skip(C)	Jäta vahele kõik sisendvoo sümbolid kuni sümbolini C.
ttyget(C)	Loe terminaalilt C.
ttyput(C)	Kirjuta terminaalile ascii-sümbol.
ttynl	Kirjuta terminaalile reavahetuse sümbol.
ttyskip(C)	Jäta vahele kõik terminaali sümbolid kuni sümbolini C.
ttytab(N)	Trüki terminaalile N tühikut.
op(P,Type,Op)	Defi neeri operator Op tüüpi Type prioriteediga P.
save(File)	Salvesta konfi guratsioon faili File.
log	Trüki andmebaas faili ’log’.

Silumine:

trace	Silu (trasseeri) klaviatuurilt sisestatavaid päringuid.
trace(G)	Silu (trasseeri) päringut G.

Üldotstarbelised predikaadid:

true	On alati tõene.
fail	On alati väär.
!	Cut.
exit	Lõpeta seanss.
abort	Katkesta täitmine.
call(G)	Täida G.
not(G)	Kui G on tõene, siis not(G) on väär (ja vastupidi).
name(A,L)	L on A sümbolite list.
repeat	Korda suvaline arv kordi.
,	Konjunktsioon.
;	Disjunktsioon.
=..	Teisendab termi aatomite listiks.
X=Y	X unifi tseerub Y'ga.
\=	Eitab =.
T1==T2	T1 ja T2 on identsed.
T1\==T2	Eitab == .
system(Comm)	Täidab operatsioonisüsteemis käsu Comm.
systemp(F,A)	F on süsteemse predikaadi funktor ja A on tema aarsus.
save_term(T)	Tõsta magasini.
unsave_term(T)	Eemalda magasinist.
member(X,Xs)	X on listi Xs element.
append(Xs,Ys,Zs)	Zs on Xs ja Ys ühend.

Eriotstarbelised predikaadid:

iterate(G)	Täida G'd kuni ta muutub vääraks (efektiivselt).
fork_exec(fi le,Comm)	C-sarnane käsk.
ancestor(G,N)	G on aktiivse päringu N's eellane.
cutg(G)	Ancestor cut.
retry(G)	Korda päringut G.

Aritmeetilised predikaadid:

T1<T2	Aritmeetilise avaldise T1 väärtus on väiksem kui aritmeetilisel avaldisel T2.

T1=\=T2	Pole võrdsed.
T1>T2	Suurem kui.
T1>=T2	Suurem või võrdne.
T1=<T2	Väiksem või võrdne.
T1=:=T2	Pole võrdsed.
R:=Exp	R on aritmeetilise avaldise Exp tulemus.
R is Exp	Sama mis :=.

Aritmeetilised tehted:

X+Y	Liitmine.
X−Y	Lahutamine.
X*Y	Korrutamine.
X/Y	Jagamine.
X mod Y	Jääk X jagamisel Y'ga.
−X	Unaarne miinus.
+X	Unaarne pluss.

Operaatorite definitsioonid:

```
:−(op(1200,fx,[:−,?−])).
:−(op(1200,xfx,[(:−),<−,−>])).
:−(op(1100,xfy,';')).
:−(op(1000,xfy,',')).
:−(op(900,fx,[not])).
:−(op(700,xfx,[=,\=,is,:=,=..,==,\==,=:=,=\=,<,>,=<,>=])).
:−(op(500,yfx,[+,−])).
:−(op(500,fx,[(+),(−)])).
:−(op(400,yfx,[*,/,//])).
:−(op(300,xfx,[mod])).
```

Tähtsamate mõistete sõnastik

aatom (*atom*) Konstant. Prologi aatomid algavad tavaliselt väikse tähega ja neil puudub loogiline struktuur.

arvutiteadus (*computer science*) Informaatika haru, mis tegeleb informatsiooni esitamise ja programmeerimise teoreetiliste küsimustega.

atomaarne valem (*atomic formula*) Predikaatsümbol koos vastava arvu alamtermidega: $p(t_1,\ldots,t_n)$.

automaatne teoreemitõestamine (*automated theorem proving*) Intellektitehnika ja arvutiteaduse haru, mis konstrueerib meetodeid teoreemide tõestamiseks arvutil.

Chomsky hierarhia (*Chomskian hierarchy*) Lõplikel automaatidel ja Turingi masinatel põhinev formaalsete keelte ja grammatikate keerukushierarhia.

disjunkt (*clause*) Literaalide disjunktsioon.

ekspertsüsteem (*expert system*) Mingit valdkonda käsitlevate faktide ja reeglite kogum koos vastuste tuletusmootoriga.

fakt (*fact*) Ühest atomaarsest lausest koosnev Horni disjunkt: $p\leftarrow$.

generatiivne grammatika (*generative grammar*) Produktsioonireeglite hulk, mille abil saab genereerida kas formaalse või tavakeele sõnu.

grammatikareegel (*grammar rule*) Asendusreegel kujul `p -> q`.

Horni disjunkt (*definite clause*) Disjunkt, milles on kuni üks positiivne literaal.

Horni grammatika (*definite clause grammar, DCG*) Grammatikareeglite kogum.

Horni loogika (*Horn logic*) Predikaatloogika osa, mille valemiteks on Horni disjunktid.

Horni programm (*definite program*) Horni disjunktide kogum.

informaatika Kõik, mis seondub informatsiooni hoidmise ja töötlemisega. Tänapäeval nimetatakse seda infotehnoloogiaks (IT).

intellektitehnika (artificial intelligence) Inimmõtlemise modelleerimisega tegelev arvutiteaduse valdkond.

list Binaarse '.'-operaatori või nurksulgude abil moodustatud rekursiivne struktuur. Loogilise programmeerimise põhiline andmestruktuur.

literaal (*literal*) Atomaarne valem või selle eitus.

loogiline programm (*logic program*) Horni disjunktide kogum.

loogiline programmeerimine (*logic programming*) Automaatse teoreemitõestamise meetoditel põhinev programmeerimistehnika. Tihti samastatakse loogiline programmeerimine programmeerimisega Prologis, eristamaks seda programmide kirjutamisest teistes predikaatloogikal ja lambda-arvutusel baseeruvates keeltes.

metainterpreteerimine (*metainterprete*) Programmi täitmise juhtimine.

metamuutuja (*metavariable*) Muutuja, mida saab käivitada kui programmi.

metapredikaat (*metapredicate*) Süsteemne predikaat.

muutuja (*variable*) Tavaline loogiline muutuja, mille väärtust ei ole pärast väärtustamist võimalik muuta. Prologi muutujad algavad suure tähega.

nimetu muutuja (*anonymous variable,* _) Muutuja, millele ei taheta omistada konkreetset väärtust: tal kas on väärtus või ei ole väärtust.

predikaat (*predicate*) Objektide vaheline seos. Näiteks emad on seotud oma lastega: *ema(Ema,Laps)*.

Prolog Horni loogikal põhinev programmeerimiskeel. Kõneldakse ka puhtast Prologist (*pure Prolog*), Edinburghi Prologist ja ISO standardile vastavast Prologist.

päring (*query*) Horni disjunkt, mis koosneb ühest või mitmest komadega eraldatud negatiivsest literaalist: $\leftarrow p$.

reegel (*rule*) Horni disjunkt, milles on üks positiivne literaal ja üks või mitu negatiivset literaali: $p \leftarrow q_1, \ldots, q_n$.

resolutsioon (*resolution*) Resolutsioonireeglil põhinev tuletusmeetod.

substitutsioon (*substitution*) Ühe või mitme muutuja kõigi esinemiste asendamine mingi termiga.

tehisintellekt Mingi konkreetne intellektitehnika seade või programm. Näiteks malemänguprogramm.

term (*term*) Tavaline predikaatloogika term — muutuja, konstantsümbol või funktsioonisümbol koos sobiva arvu alamtermidega: $f(t_1, \ldots, t_n)$. Loogilises programmeerimises käsitletakse termina ka Prologi programmi ja selle osi.

tingimustega loogiline programmeerimine (*constraint logic programming, CLP*) Mitmesuguste optimeerimis- ja lineaarse planeerimise meetodite kasutamine loogilisel programmeerimisel.

tühidisjunkt (*empty clause*) Disjunkt, milles pole ühtegi literaali. Tühidisjunkt tähistab loogilist väärust, sest seda ei saa tõeseks muuta.

unifitseerimine (*unify, match*) Kahe või enama termi muutujate viimine samale sümbolkujule, kasutades substitutsioone.

variantide läbivaatamine (*backtracking*) Lihtsaim otsingustrateegia, mis vaatab järjest läbi kõik variandid.

ühikdisjunkt (*unit clause*) Ühest literaalist koosnev disjunkt.

Viited

[1] H. Ait-Kaci. *Warren's Abstract Machine: a tutorial reconstruction.* MIT Press, 1991. [http://www.isg.sfu.ca/~hak/documents/wam.html]

[2] ALS Prolog. [http://www.als.com/]

[3] Amzi! Prolog. [http://www.amzi.com]

[4] J. Andrews (toim.). Answers to some Frequently Asked Questions (FAQ) often seen in *comp.lang.prolog*. [http://fas.sfu.ca/0/cs/people/ResearchStaff/jamie/personal/prolog-faq.1]

[5] K. R. Apt. *Introduction to logic programming.* 1988. (*Logic programming.* Raamatus: J. van Leeuwen (toim.). *Handbook of theoretical computer science.* North-Holland, 1990, 493–574.)

[6] K. R. Apt. *From Logic Programming to Prolog.* Prentice-Hall, 1997.

[7] The Association of Logic Programming Web Site. [http://www-lp.doc.ic.ac.uk/alp/archive.html]

[8] BinProlog. [ftp://clement.info.umoncton.ca/BinProlog3.0.tar.gz]

[9] D. L. Bowen (toim.), L. Byrd, F. C. N. Pereira, L. M. Pereira, D. H. D. Warren. *DECsystem-10 Prolog user's manual.* 1982. [http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/doc/intro/prolog.doc]

[10] **I. Bratko. *Prolog programming for artificial intelligence.* Addison Wesley, 1986.

[11] *Byte,* August (9), 1987.

[12] R. Cafolla, D. A. Kauffman. *Turbo Prolog: step by step. With Turbo Prolog 2.0.* Merrill, 1988.

[13] C.-L. Chang, R. C.-T. Lee. *Symbolic logic and mechanical theorem proving.* Academic Press, 1973.

[14] **W. F. Clocksin, C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[15] J. Cohen. *A view of the origins and development of Prolog.* Comm. ACM 31(1), 1988, 26–36.

[16] A. Colmerauer, P. Roussel. *The birth of Prolog.* ACM SIGPLAN Notices 28(3), 1993, 37–52. Raamatus: T. J. Bergin, R. G. Gibson *History of programming languages.* ACM Press, 1996, 331–352.

[17] M.A. Covington. *Efficient Prolog: a practical guide.* 1989. [ftp://ai.uga.edu/pub/ai.reports/ai198908.ps.Z]

[18] *M. A. Covington. *Natural language processing for Prolog programmers.* Prentice Hall, 1994.

[19] *M. Covington, D. Nute, A. Vellino. *Prolog programming in depth.* Scott, Foresman, 1988.

[20] ECLiPSe. [http://www-icparc.doc.ic.ac.uk/eclipse]

[21] ESL Prolog-2. [http://www.cs.ut.ee/~tqnu/eslpdpro.zip]

[22] E. A. Feigenbaum, P. McCorduck. *The fifth generation: artificial intelligence and Japan's computer challenge to the world.* Addison-Wesley, 1983.

[23] M. Fitting. *First-order logic and automated theorem proving.* Springer-Verlag, 1996.

[24] P. Gibbins. *Logic with Prolog.* Oxford University Press, 1988.

[25] Gödel. [ftp://ftp.cs.bris.ac.uk/goedel]

[26] M. Hanus. *Problemlösen mit PROLOG.* Stuttgart, 1987.

[27] J. Henno. *PROLOG – see on imelihtne.* Huvitav informaatika III, Tallinn, 1989.

[28] J. Henno. *Prolog ja olympoksen jumalat.* Helsinki, 1991.

[29] J. Henno. *Formaalsed keeled ja abstraksed automaadid.* Tallinn, 1991.

[30] J. Henno. *Prologi kursus*. TTÜ, kevad 1997. [http://www.ttu.ee/info/inf/conference/courswre/prolog.htm]

[31] P. Hill, J. Lloyd. *The Gödel programming language.* MIT Press, 1994.

[32] C. J. Hogger. *Introduction to logic programming.* Academic Press, 1984.

[33] C. J. Hogger. *Essentials of Logic Programming.* Clarendon, 1990.

[34] C. J. Hogger, R. A. Kowalski. *Logic programming.* Raamatus: S. C. Shapiro (toim.). *Encyclopedia of artificial intelligence.* Wiley, 1992, 873–891.

[35] D. Hovel. *Using Prolog in Windows NT network configuration.* [http://www.research.microsoft.com/research/dtg/davidhov/pap.htm]

[36] ISO Prolog Standard. [http://www.als.com/nalp/prolog_std.html]

[37] M. Kilp. *Neljavärviprobleem: ühe matemaatikaprobleemi lahenduse lugu*. Tallinn, 1984.

[38] M. Koit. *Resolutsioonimeetod.* Tartu, 1989.

[39] R. Kowalski. *Predicate logic as a programming language.* Raamatus: J. Rosenfeld (toim.). *Proc. IFIP Congress.* North-Holland, 1974, 569–574.

[40] R. Kowalski. *Algorithm = Logic + Control.* Comm. ACM 22, 1979, 424–436.

[41] R. Kowalski. *Logic for problem solving.* North-Holland, 1979.

[42] R. Kowalski. *The early years of logic programming.* Comm. ACM 3(1), 1988, 38–43.

[43] J. W. Lloyd. *Foundations of logic programming.* Springer-Verlag, 1987.

[44] D. W. Loveland. *Automated theorem proving: a quarter century review.* Am. Math. Soc. 29, 1984, 1–42.

[45] LPA Prolog. [http://www.lpa.co.uk/]

[46] D. Maier, D. S. Warren. *Computing with logic.* Benjamin Cummings, 1988.

[47] *J. Malpas. *Prolog: a relational language and its applications.* Prentice Hall, 1987.

[48] K. Marriott, P. J. Stuckey. *Programming with constraints: an introduction.* MIT Press, 1998. [http://www.cs.mu.oz.au/~pjs/book/progs.html]

[49] Mercury. [http://www.cs.mu.oz.au/~zs/mercury.html]

[50] D. Merritt. *Adventure in Prolog.* Springer-Verlag, 1990.

[51] D. C. Merritt. *Exploring Prolog: adventures, objects, animals and taxes.* PC AI magazine, Volume 7, Number 5 September/October 1993, 18–23. [http://www.amzi.com/profun.htm], [http://www.cs.auckland.ac.nz/~j-hamer/07.363/explore.html]

[52] *Natural Language Processing FAQ.* [ftp://rtfm.mit.edu/pub/usenet-by-hierarchy/comp/ai/nat-lang/Natural_Language_Processing_FAQ]

[53] *U. Nilsson, J. Małuszyński. *Logic Programing and Prolog.* Wiley, 1990.

[54] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[55] O. Ore. *Graafid ja nende kasutamine.* Tallinn, 1976.

[56] PC AI Magazine. *The Prolog Programming Language.* [http://www.pcai.com/pcai/New_Home_Page/ai_info/pcai_prolog.html]

[57] PDC Prolog. [http://www.pdc.dk/]

[58] S. Reeves, M. Clarke. *Logic for computer science.* Addison Wesley, 1990.

[59] J. A. Robison. *A machine-oriented logic based on the resolution principle.* J. ACM 12, 1965, 23–41.

[60] SB-Prolog. [http://www.cs.ut.ee/~tqnu/sb_prolog.zip]

[61] U. Schöning. *Logic for computer scientists.* Birkhäuser, 1989.

[62] SICStus Prolog. [http://www.sics.se/sicstus.html]

[63] R. Spencer-Smith. *Logic and Prolog.* Harvester, 1991.

[64] **L. Sterling, E. Shapiro. *The art of Prolog: advanced programming techniques.* MIT Press, 1994. [ftp://ftp.cwru.edu/ArtOfProlog/]

[65] SWI-Prolog. [http://www.swi.psy.uva.nl/usr/jan/SWI-Prolog.html]

[66] T. Tamme. *Loogilise programmeerimise kursus.* Kevad 1995. [ftp://romulus.cs.ut.ee/pub/info/loogiline.progr]

[67] T. Tamme. *Loogilise programmeerimise kursus.* Kevad 1997. [http://www.cs.ut.ee/~tqnu/logpr97.html]

[68] T. Tamme. *Programmeerimiskeelte kursuse Prologi osa konspekt.* Kevad 1997. [http://www.cs.ut.ee/~tqnu/prolog97.html]

[69] *T. Tamme, T. Tammet, R. Prank. *Loogika: mõtlemisest tõestamiseni.* Tartu, 1997.

[70] M. van Emden, R. Kowalski. *The semantics of predicate logic as a programming language.* Comm. ACM 23, 1976, 733–742.

[71] P. van Hentenryck. *Constraint satisfaction in logic programming.* MIT Press, 1989.

[72] P. van Roy. *1983–1993: the wonder years of sequential Prolog implementation.* DEC, 1993. [ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/SequentialPrologImp.ps.Z]

[73] Visual Prolog. [http://www.visual.com], [http://www.pdc.dk/vip/]

[74] E. Vääri. *Eesti keele õpik keskkoolile.* Tallinn, 1976.

[75] M. Wallace. *Communicating with databases in natural language.* Ellis Horwood, 1984.

[76] J. Weizenbaum. *Computer power and human reason.* Freeman, 1976.

[77] J. Weizenbaum. *ELIZA — a computer program for the study of natural language communication between man and machine.* CACM 9, 1966, 36–45.

Indeks

*, 83
+, 83
,, 19, 82, 83
-, 83
->, 84
->, 83
., 22
/, 83
//, 83
:-, 18, 19, 83
:=, 83
;, 19, 82, 83
<, 82, 83
<-, 83
=, 82, 83
=.., 61, 62, 82
=:=, 83
=<, 83
==, 82, 83
>, 83
>=, 83
?-, 83

aatom, 3, 14, 29, 38, 84
abolish, 80
abort, 82
aeg, 27, 54, 57
ainsus ja mitmus, 47, 50
Ait-Kaci, Hassan, 10
akumulaator, 24
alamprogrammide teegid, 26
algoritm, 8
ALP, 11
ALS Prolog, 10
aluseruhm, 28
Amzi Prolog, 10–12, 26
analusaator, 30
ancestor, 82
ancestor cut, 66
andmebaas, 14, 43, 60
andmestruktuur, 8
andmevoog, 40
apostroof, 14, 38, 45
append, 29–31, 34, 59, 82
Apt, Krzysztof, 11
arg, 61, 80
aritmeetikaviga, 63
aritmeetiliste avaldiste lihtsustamine, 20
aritmeetiliste avaldiste teisendamine, 20
Arity Prolog, 10
arvud, 49
arvutatud vastus, 5
arvutiteadus, 28, 84
ascii tekst, 37
assert, 55, 63, 80
asserta, 80
assertz, 45, 80
assotsiatiivsus, 18
assotsieerumine paremale, 19
at_end_of_stream, 40
atom, 61, 80
atomaarne valem, 84
atomic, 80
atribuut, 69
Austraalia, 9, 13
automaat, 38
automaatne teoreemitõestamine, 9, 14, 84

bagof, 55, 56, 63, 64
between, 27
bibliograafi line andmebaas, 43
BinProlog, 12
Borland, 12
Bratko, Ivan, 10, 21
Bristol, 9
Byrd, Lawrence, 4
Byrdi kastimudel, 4

C, 10, 12, 16, 26, 36
C, 35
C-Prolog, 12
Cafolla, Ralph, 47
CALL, 4

call, 60, 82
Chomsky hierarhia, 84
Clark, Keith, 9
Clark, Michael, 11
clause, 61, 62, 65, 80
Clocksin, William, 10, 36
close, 40
CLP(R), 12
Colmerauer, Alain, 1, 10
comp.lang.prolog, 11, 87
cons-operaator, 22
constant, 80
consult, 35, 44, 80
Cooperi meetod, 40
count_off, 53
Covington, Michael, 10, 11, 36, 54
cputime, 27
CR, 38
cut, 56
cut, 57–60
cut-fail, 59
cutg, 82

deklaratiivne keel, 36
deklaratiivsus, 9, 13, 57, 58
determineeritus, 57
diferentslist, 30, 32, 35
disjunkt, 62, 63, 84
disjunktsioon, 19
display, 81
displayl, 81
displayq, 81
dunaamiline andmebaas, 13, 43
dunaamiline predikaat, 13, 17, 54

Eclipse, 12, 13, 24, 63
Edinburgh, 1, 10, 41
Edinburghi Prolog, 10, 12, 85
eesti keel, 28
eesti keele kaanded, 49
efektiivsus, 13, 14, 26, 27, 30, 32, 38, 49, 54, 57, 58, 66, 67
eitus, 2, 9, 19, 59, 60, 64, 65
eksistentsikvantor, 59, 64
ekspertsusteem, 43, 84
ekstraloogiline predikaat, 61
ekvivalents, 2, 19
ekvivalentsus Turingi masinaga, 28
Eliza, 41, 42, 47
ema_on, 18
ESL-Prolog, 6, 12
Euler, Leonhard, 67
EXIT, 4, 5
exit, 82
expand_term, 35

FAIL, 4
fail, 14, 55, 59, 82
fail-tsukkel, 14, 44
faililõpp, 37, 40
fakt, 4, 14, 29, 43, 63, 65, 68, 69, 84
father, 7, 13, 15
findall, 21, 55, 56, 63, 64
Fitting, Melvin, 11
flatten, 25
flush, 81
fork_exec, 82
functor, 61, 80
funktor, 61
funktsioonisumbol, 86
fy, 19

generaator, 30
generatiivne grammatika, 15, 84
get, 61, 81
get0, 39, 81
Gibbins, Peter, 11
globaalne muutuja, 55
globaalne parameeter, 13
Godel, 9, 12, 13
graafi ka, 12
grammatika, 8, 15, 33, 84
grammatika algsumbol, 28
grammatikareegel, 33–35, 47, 53, 84

Hanus, Michael, 8, 43
Henno, Jaak, 11, 21, 36
Hogger, Chris, 8, 11
Horni disjunkt, 2, 4, 9, 84
Horni grammatika, 15, 34, 53, 54, 84
Horni loogika, 1, 84
Horni programm, 84

Horni reegel, 59
Hovel, David, 11

IBM, 12
IBM Prolog, 12
ICL, 12
if_then_else, 58
imperatiivne keel, 26, 36, 58
implikatsioon, 19
indekseerimine, 54
infi ksfunktor, 18
infi kskuju, 17
infi ksoperaator, 19
infi kspredikaat, 18, 19
informaatika, 85
infosusteem, 43
infotehnoloogia, 85
inimvestlus, 47
integer, 61, 80
intellektitehnika, 1, 9, 11, 14, 85
interpreteerimine, 8, 13
intuitsionistlik loogika, 59
is, 23, 83
ISO standard, 12, 85
IT, 85
iterate, 82

Jaakson, Hermann, 70
Jaapan, 1, 10
jarjekord, 77
Journal of Logic Programming, 11

kaandkond, 41
kahendpuu, 55
kast, 13
kasutajaliides, 26, 43
Kauffman, Daniel Albert, 47
keerdulesanne, 67, 68
kirjavahemark, 38
kitsendustega programmeerimine, 9
klassikaline loogika, 59
Koit, Mare, 11
kompilaator, 10
kompileerimine, 8, 12, 13, 26
Konigsbergi sildade probleem, 67, 70
konjunktsioon, 19
kontekstivaba grammatika, 28, 33

Kowalski, Robert, 9, 10
Krahn, Edgar, 70
kusimus, 68, 69
kvantor, 2, 9, 59

lahend, 13
lahendi otsimine, 13
laiuti otsimine, 66
last, 25
lause, 29
lausearvutuse tehted, 19
lausearvutuse valem, 19, 20
lehtede_arv, 55
length, 72
LF, 38
linearize, 20
lingvistika, 8, 9, 11, 15, 28, 36
Linux, 12
lippude probleem, 70
lips, 54
Lisp, 10, 22
list, 17, 21, 22, 29, 37, 53, 85
listing, 13, 80
literaal, 85
Lloyd, John, 11, 27
loend, 21
log, 81
Logic Programming Newsletter, 11
loogika, 14
loogikareegel, 15
loogikavaline predikaat, 59
loogiline programm, 8, 59, 85
loogiline programmeerimine, 8, 11, 85
loogiline tõesus, 20
loogiline vaarus, 20
Loogilise Programmeerimise Assotsiatsioon, 11
loogilised tehted, 19
loop, 66
LPA Prolog, 10, 12, 26
lõikeoperaator, 56–59, 65, 66
lõikepredikaat, 35, 38

Maier, David, 10, 11
Malpas, John, 10
malu, 26, 27, 43, 57
maluhaldamine, 54

mang, 46
mangude realiseerimine, 11
map_list, 62
margendamine, 17, 67
married, 6, 7, 14, 15
Marriott, Kim, 11
Marseille Prolog, 9
masinkood, 10, 26
massiiv, 36
max, 57
Mellish, Chris, 10, 36
member, 7, 25, 58, 59, 82
Mercury, 9, 13
Merritt, Dennis, 11
metainterpretaator, 65
metainterpreteerimine, 65, 85
metaloogiline predikaat, 61
metamuutuja, 27, 60, 63, 65, 85
metapredikaat, 85
metaprogrammeerimine, 61
mittedetermineeritus, 10, 26
mitteterminaal, 28, 33
mod, 83
mother, 14, 15
MS Dos, 12
MS Windows, 12
Murk, Oleg, 27
muutuja, 14, 85
mõiste, 6

name, 38, 73, 82
neljavarviprobleem, 70
New Generation Computing, 11
nimetu muutuja, 3, 85
nl, 14, 81
nonvar, 23, 80
not, 19, 59, 60, 64, 82, 83
nuputamisulesanne, 69
Nuut, Juri, 70

O'Keefe, Richard, 8, 11, 47
objektorienteeritud andmebaas, 15
oeldiseruhm, 28
omadus, 6
on_mees, 19
once, 27, 54
op, 81
open, 40
Ore, Oystein, 70
osaline list, 23
osaline struktuur, 30
otsingumootor, 43
otsingupuu, 57, 60, 66, 77

paralleelsus, 10, 15
paring, 4, 54, 60, 63, 85
paringu edasilukkamine, 63
Pascal, 36, 37
PC AI Magazine, 11
PDC Prolog, 12
permutatsioon, 25, 67
phrase, 33
postfi kskuju, 17
postfi kspredikaat, 19
postorder, 55
predikaat, 85
predikaatarvutus, 8
predikaatloogika, 11
prefi kskuju, 17, 18
print, 81
prioriteet, 18, 19
produktsioon, 15
programmi taitmine, 12
Prolog, 1, 6, 8–11, 13, 14, 20, 26, 28, 33, 36, 37, 41, 43, 54, 59, 63, 65, 85
Prolog II, 9
Prolog III, 12
Prolog IV, 9
Prolog-10, 10, 80
Prologi ISO standard, 11
Prologi programm, 61
Prologi rakendused, 14
Prologi standard, 12, 36
prototuupimine, 16
protseduraalsus, 9, 57, 58
puhas Prolog, 59, 65, 85
punane lõikeoperaator, 57
put, 81
puu labimine, 55

Quintus Prolog, 10, 12, 35

range, 27

read, 45, 61, 81
read_rest_of_word, 38
read_words, 38–40
realõpp, 37
reavahetus, 38
reconsult, 35, 80
REDO, 4
reduce, 66
reegel, 4, 14, 43, 62, 63, 65, 86
Reeves, Steve, 11
rekursiivne struktuur, 22
rekursioon, 16, 17, 43, 55, 59
relative, 16
relatsiooniline andmebaas, 15
repeat, 43, 59, 82
repeat_stream, 40
resolutsioon, 86
resolutsioonimeetod, 9, 11
retract, 46, 55, 80
retractall, 46
retry, 82
ristsõna, 60
Robinson, Alan, 9
roheline lõikeoperaator, 57, 58
Rolling Stones, 36
Rootsi, 12
rot13, 41
Roussel, Philippe, 1, 10

sabarekursioon, 58, 59
Saksamaa, 43
Sarv, Jaan, 70
save, 81
save_term, 82
SB-Prolog, 12
Schoning, Uwe, 11
see, 37, 40, 81
seeing, 81
seen, 40, 81
seis, 73
seitse, 73
select, 25
seos, 6, 43
setof, 21, 55, 56, 63, 64
Shapiro, Ehud, 36, 54, 70, 80
SICS, 12
Sicstus Prolog, 10, 12, 43
sihitis, 47
sihitiseruhm, 49
silumine, 26, 35, 65
simplify, 20
sisendvoog, 37, 40
skip, 81
slowsort, 25, 27
Solaris, 12
sorteerimine, 25
Spencer-Smith, Richard, 11
square, 62
sread, 81
statistika, 27
Sterling, Leon, 36, 54, 70, 80
strateegia, 65, 66
string, 38, 53
struktuur, 69
Stuckey, Peter, 11
substitutsioon, 86
succ, 62
sugavuti otsimine, 66
sugulaste andmebaas, 19
sugulaste margendamine, 67
sugulaste teadmistebaas, 17
sulgude arajatmisreegel, 19
sum, 23
suntaksianaluus, 18, 37, 49
suntaksipuu, 20, 47
suntaktiliste avaldiste teisendamine, 20
susteemne predikaat, 65
suurtaht, 14
SWI-Prolog, 6, 12, 13, 26, 40, 46, 54
system, 61, 82
systemp, 82
sõna, 29
sõnade poolitamine, 41
sõnaraamat, 29, 61
sõnaraamatu pakkimine, 40
sõprade probleem, 68, 70

Taani, 12
tab, 81
tabulatsioon, 37
taidetav programm, 26
Tammeoru talu, 36
Tammet, Tanel, 8, 11

Tarau, Paul, 12
Tartu Ulikool, 8, 12
teadmised, 43
tehisintellekt, 14, 47, 86
teisendusreegel, 20, 28
teist jarku loogika, 64
teist jarku predikaat, 56, 63
teksti juhtsumbol, 37, 38
tekstirobot, 47
tell, 61, 81
telling, 81
Tennisberg, Targo, 21
teooria, 11
term, 9, 17, 61, 86
terminaal, 28, 29, 32, 33
time, 27, 54
time(Goal), 27
tingimustega loogiline programmeerimine, 11, 12, 86
tingimustega programmeerimine, 24
told, 81
trace, 5, 54, 81
transleerimismeetod, 10
trasseerimine, 56, 60
true, 14, 57, 63, 65, 82
ttyget, 81
ttynl, 81
ttyput, 81
ttyskip, 81
ttytab, 81
tuhidisjunkt, 86
tuhik, 37
tuhilist, 22, 33
Turbo Prolog, 12
Turingi masin, 28, 84
Turingi test, 47
tuubikontroll, 19, 24, 26, 36
tuubiteooria, 9
tuupimine, 12, 13
tõlkimine, 41

uhekordne muutuja, 7
uhestamine, 69
uhikdisjunkt, 86
uldisusekvantor, 59
unifi tseerimine, 3, 4, 13, 57, 58, 86
Unix, 12
unsave_term, 82
user, 37, 56

Vaari, Eduard, 36
vaarus, 86
vaba muutuja, 59
vaiketaht, 14
valjundvoog, 40
van Hentenryck, Pascal, 11
van Roy, Peter, 10
var, 61, 64, 80
variantide labivaatamine, 30, 38, 56, 86
vasakassotsiatiivne kuju, 20
vastus, 57
VAX, 10
veeb, 26
veebiliides, 12
veinide andmebaas, 43
veinide teadmistebaas, 47
verify, 60
viienda põlvkonna arvuti projekt, 1, 10, 15
viis, 73
viivitamine, 63
Visual Prolog, 10, 12, 26

WAM, 10, 13
Warren, David, 1, 10
Warren, David Scott, 10, 11
Warreni abstraktne masin, 10, 12, 13, 26
Weizenbaum, Joseph, 41, 47
Wielemakeri, Jan, 12
Windows, 12
Windows 95, 12
Windows NT, 11
wines.txt, 44
write, 13, 61, 81
writeq, 44, 81

xfx, 18
xfy, 19