TARTU ÜLIKOOL

Arvutiteaduse instituut

Tõnu Tamme

Loogilise programmeerimise meetod

Tartu 2003

Õpiku valmimist toetasid EITSA Tiigriülikool ja HESP [1]

Toimetaja: Ülo Kaasik

Sisukord

ISBN 9985-56-819-2

Tartu Ülikooli Kirjastus
www.tyk.ut.ee
Tellimus nr. 768

[1] Autor kasutas küljendamiseks süsteemi LaTeX ning trükikirju Times, MathTime ja Computer Modern Typewriter. Joonised valmisid paketiga PSTricks ja aineregister programmiga xindy.

1. peatükk

Programm kui teadmiste baas

Faktid, reeglid ja päringud. Sugupuu kirjeldamine. Inimestevaheliste mõistete defineerimine. Töö Prologis. Programmi testimine ja silumine.

1.1. Loogiline programmeerimine ja Prolog

Loogiline programmeerimine tekkis 1970-ndate aastate alguses Prantsusmaal Marseille's. Seal kohtusid lingvistikaprobleemidest huvituv Alain Colmerauer ja automaatse teoreemitõestamisega tegelev Robert Kowalski. Neist esimene oli teinud lootustandvaid katsetusi ilmaennustuste tõlkimisel prantsuse keelest inglise keelde ja vastupidi ning teine oli kirjeldanud uusi resolutsioonistrateegiaid matemaatiliste väidete efektiivsemaks tuletamiseks. Nende koostöö viljana tekkis uus programmeerimiskeel, mis sai nimeks Prolog (ingl. k. *PROgramming in LOGic, pr. k. PROgrammation et LOGique*). Colmerauer huvitus automaatse teoreemitõestamise meetodite (unifitseerimise ja resolutsiooni) kasutamisest tegeliku elu ülesannete lahendamiseks. Kowalski tutvustas talle neid matemaatilise loogika meetodeid ja näitas, et loodavat keelt võib teatud mõttes vaadelda esimest järku predikaatloogika alamhulgana, nn. Horni loogikana.[2]

[2] Robert Kowalskilt pärineb ütlus *algorithm = logic + control*.

Prolog ise, vähemalt selle põhiosa, on äärmiselt lihtne — nii matemaatilise loogika tundjale kui ka programmeerijale. Formaliseerime Prologis näiteks klassikalise süllogismi

> Kõik inimesed on surelikud. Sokrates on inimene. Järelikult Sokrates on surelik.

Tulemuseks on kahe predikaadi kirjeldused: *inimene* ja *surelik*, millest mõlemale antakse ümarsulgudes ette üks argument:

```
inimene(sokrates).
surelik(X):-inimene(X).
```

Predikaatide **kirjeldused** koosnevad faktidest ja reeglitest. Predikaadi *inimene* kirjeldus koosneb ühest **faktist**, mis väidab, et Sokrates on inimene. Predikaadi *surelik* kirjeldus koosneb ühest **reeglist**, mis ütleb, et *X* on surelik tingimusel, et *X* on inimene. Reegel erineb faktist selle poolest, et koosneb kahest osast — koolonile ja miinusele eelnevast **päisest** ja sellele järgevast **sisust**. Reegli sisuks on komadega eraldatud kitsendused, mis tagavad reegli päises esitatud predikaadi tõesuse. Fakt koosneb ainult päisest ning selle sisu on tühi. Fakti on lihtne esitada reeglina, kasutades sisuna samaselt tõest predikaati *true*. Näiteks saab predikaadi *inimene* kirjelduse ümber kirjutada ekvivalentsel kujul reeglina

```
inimene(sokrates):-true.
```

Formaliseerimisel tuleb tähele panna, et kõik programmis esinevad kirjeldused peavad lõppema punktiga. Tavaprogrammeerimise mõttes ei ole **predikaatide** kirjeldused protseduurid, vaid funktsioonid, mis väljastavad töö lõpus vastuse: kas “jah” (ingl. k. *yes*) või “ei” (ingl. k. *no*). Näiteks argumendi *sokrates* korral annab predikaat *inimene* vastuseks “jah”. Kuid võttes argumendiks Robert Kowalski või Alain Colmeraueri selgub, et nad ei ole inimesed — predikaadi *inimene* kirjelduses ei nimetata ühtegi teist inimest peale Sokratese. Selguse huvides võib selle predikaadi kirjelduse ümber kirjutada ekvivalentsel kujul, kus sisendargument on tähistatud muutujaga *X* ja kirjeldustes loetletakse argumendi väärtused, mis annavad vastuse “jah”; ülejäänud argumentide korral on vastuseks “ei”:

```
inimene(X):-X=sokrates.
```

Prologis eristatakse muutujaid ja konstante tähise esitähe järgi: **aatomid** ehk konstandid algavad väiketähega ja **muutujad** suurtähega.[3] Aatom võib koosneda ka vaid kirjavahemärkidest või olla apostroofides sõne. Muutuja või

[3] Aluseks on inglise keele tähestik, mistõttu Prolog jääb hätta eestikeelsete suurte täpitähtedega.

aatomi esimesele järgnevateks sümboliteks võivad olla nii suurtähed ja väiketähed kui ka numbrid. Erandina käsitletakse suurtähena alakriipsu '_'. Ülaltoodud predikaatide kirjeldustes kasutatakse ühte muutujat *X* ja kolme aatomit *inimene*, *sokrates* ja *surelik* (neist esimene ja kolmas tähistavad predikaate ja teine indiviidi, milleks võib olla suvaline objekt). Oletame, et me soovime täiendada inimeste loetelu. Sellisel juhul on, sõltuvalt programmeerija eelistusest, kaks teed. Esiteks, dubleerida fakti erinevate indiviidide nimedega:

```
inimene(sokrates).
inimene(robert).
inimene(alain).
```

Teine võimalus on esitada predikaadi kirjelduses positiivsete vastuste loetelu, eraldades need üksteisest semikooloniga, mis väljendab alternatiive:

```
inimene(X):-
    X=sokrates
    ; X=robert
    ; X=alain.
```

Nagu näha, kirjutatakse reegli sisus semikoolonid taandrea algusesse, et nad eristuksid visuaalselt komadest, mis on tavaliselt rea lõpus. Mõlema predikaadi kirjelduste transleerimisel saadakse loogiliselt samaväärsed sisemised esitused, kuid predikaadi nime dubleeriv esitus on esimese argumendi automaatse indekseerimise tõttu kiirem.

Vaatleme nüüd predikaadi *surelik* kirjeldust leheküljel 4. See ütleb, et vastuse saamiseks käivitatakse predikaadi *inimene* kirjeldus, kusjuures "jah" vastatakse kõigil sellistel argumendi väärtustel, mille jaoks predikaat *inimene* annab vastuse "jah". Süllogismile järelduse arvutamiseks käivitame nüüd päringu

```
?-surelik(sokrates).
yes
```

Küsimärgist ja miinusest koosnev konstruktsioon tähistab Prologis **päringut** ehk käsku, s.t. pöördumist olemasolevate predikaatide poole.[4] Toodud päringu korral käivitatakse esmalt predikaat *surelik*, mis käivitab omakorda predikaadi *inimene*. Et viimane vastab "jah", siis on ka lõppvastuseks "jah", s.t. olemasolevatest predikaatidest järeldub, et Sokrates on surelik. Vastus *yes* on siin välja toodud, kuid hiljem eeldame päringute tõesust sageli vaikimisi.

[4] Interaktiivses töös Prologiga pole küsimärki ja miinust vaja eraldi sisestada, sest interaktiivne töö käibki vaid päringutega. Soovides kirjeldada fakte või reegleid, loeme need kas olemasolevast tekstifailist või sisestame töö käigus, minnes predikaadiga `[user]` programmide kirjeldamise režiimi. Tagasi saab klahvikombinatsiooniga *ctrl-D*.

Alternatiivina oleksime võinud käivitada predikaadi *surelik*, andes argumendiks muutuja *X*:

```
?-surelik(X).
X=sokrates
```

Päringu täitmise skeem on eelmisega sarnane: muutuja *X* edastatakse predikaadile *inimene*, mis väljastab lisaks vastusele "jah" ka muutuja sobiva väärtuse, antud juhul $X = sokrates$. Saadud vastus on ka algpäringu lõpptulemuseks.

Ülesanne 1. Formaliseerige Prologis süllogism

Kõik linnud lendavad. Hani ja part on linnud. Järelikult part lendab.

Ülesanne 2. Esitage eelmises ülesandes koostatud programmile päringud:

a) Kas part lendab?
b) Kes lendab?

1.2. Genealoogilise teadmiste baasi loomine

Prologis on lihtne kirjeldada ka inimestevahelisi seoseid. Võtame näiteks perekonna, kus Mary on abielus Johniga ning neil on lapsed Alice, Bob ja Carol. Johni ema on Ann, kes on abielus Peteriga ning neil on lisaks Johnile ka tütar Linda. Linda on abielus Pauliga ning neil on poeg Fred:

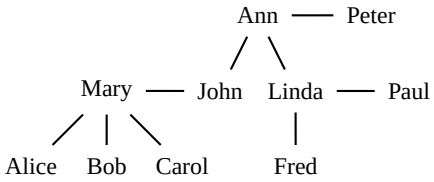
```
married(mary,john).
married(linda,paul).
married(ann,peter).

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

Kirjeldatud suguvõsa illustreerib skeem joonisel 1.1, kus horisontaaljooned tähistavad naise ja mehe vahelist suhet *married* ning vertikaal- ja diagonaaljooned lapse ja ema vahelist suhet *mother*. Lisaks suhetele kirjeldame ära ka inimese soo: *male* tähistab meessugu ja *female* naissugu:

```
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. Nüüd saame esitada päringuid. Küsime näiteks: kas Mary on abielus?



Joonis 1.1. Inimestevahelised seosed suguvõsas.

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

Vastus `yes` ütleb, et Prologil õnnestus **unifitseerida** see päring mingi predikaadiga oma sisemises andmebaasis, mis koosneb sisse loetud predikaatide kirjelduste faktidest ja reeglitest. Avaldiste unifitseerimine tähendab nende süntaktilise kokkulangemise kontrollimist, kasutades vabade muutujate sobivaid asendusi.

Alakriips '_' tähistab siin argumenti, mille väärtus meid ei huvita — soovime lihtsalt teada, kas Mary on abielus või ei ole; alakriipsu kutsutakse nimetuks muutujaks (ingl. k. *anonymous variable*). Hiljem näeme, et lisaks vastuste leidmise kiirendamisele — kaob vajadus vastuste väljastamiseks — on alakriipsul muidki rakendusi. Ent kirjutades päringus argumendid vastupidises järjekorras saame teistsuguse vastuse:

```
?-married(_,mary).
no
```

Nimelt on kirjeldamata seose *married* sümmeetrilisus.

Küsime nüüd: kes on Mary lapsed?

```
?-mother(Child,mary).
Child = alice ;
Child = bob ;
Child = carol ;
no
```

Prolog unifitseerib esmalt muutuja *Child* nimega Alice ning jääb ootama. Et me tahame teada ka Mary teisi lapsi, siis vajutame semikoolonile ning saame uueks vastuseks Bob; Mary kolmandaks lapseks on Carol; lõpuks saame teate `no`, s.t. antud päringul ei ole rohkem vastuseid. Järgmises punktis selgitame, kuidas need vastused arvutatakse.

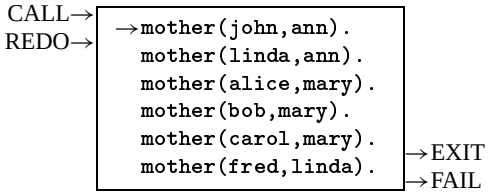
Ülesanne 3. Looge fail *perekond.pl*[5] ning kirjutage sellesse faktid oma sugulaste kohta, kasutades predikaate *mother*, *married*, *male* ja *female*.

Ülesanne 4. Lugege oma perekonna andmebaas sisse ja esitage sellele päringud (vt. punkt 1.6 lk. 12):

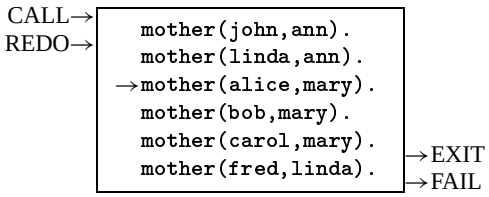
a) Kes on minu ema?
b) Kes on minu ema lapsed?

1.3. Byrdi kastimudel

Kuidas Mary lapsed leitakse? Prologi tööd võib kirjeldada Lawrence Byrdi kastimudeliga (ingl. k. *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. Näiteks



Esmakordsel sisenemisel kasti (CALL) tõstetakse selle sisemine viit predikaadi esimesele kirjeldusele[6], mida püütakse päringuga **unifitseerida**, s.t. muuta kirjeldust ja päringut muutujate asendamise teel süntaktiliselt võrdseks. Kastis liigutakse senikaua allapoole, kuni unifitseerimine õnnestub ning väljutakse pordi EXIT kaudu. Päringu `?-mother(Child,mary)` korral saame esimeseks vastuseks Alice, sest muutuja *Child* saab asendada aatomiga *alice*:



[5] Prologi programmide levinud laienditeks on '.pl' ja '.pro'. Neist esimene kipub segi minema Perli programmide laienditega.

[6] Kirjelduste järjekord on sama, mis programmis.

Vajutamine semikoolonile tähendab, et soovime saada uusi vastuseid. Et kasti sisemine viit jäi väljumisel kolmandale faktile, siis siseneme samasse kasti uuesti (REDO) ja jätkame allapoole liikumist, saades vastused Bob ja Carol. Vastuste lõppemisel väljutakse kastist läbi pordi FAIL.

Aga kes on Alice'i isa? Kui oletada, et selleks on inimene, kes on tema emaga abielus, siis võib isa leidmiseks esitada **liitpäringu**, kus leiame esmalt vaadeldava isiku ema *_M* ning seejärel ema abikaasa:

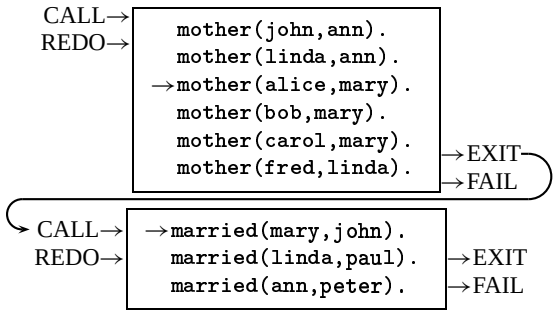
```
?-mother(alice,_M), married(_M,Father).
Father = john
```

Muutuja *_M* alguses oleva alakriipsu tõttu jäetakse väljastamata ema nimi. Kuidas Prolog täidab kahest järjestikusest päringust koosnevat liitpäringut? Seda võib selgitada Prologi enda keeles — päringutega:

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

Liitpäringu korral käivitatakse kõigepealt selle vasakpoolseim alampäring. Antud juhul leitakse sisemisest andmebaasist vasakpoolse päringu vastuseks, et Alice'i ema on Mary. Seejärel tehakse liitpäringus unifitseerimisel saadud asendused ja kustutatakse vasakpoolne päring. Sama tegevust korratakse liitpäringu allesjäänud osaga, s.t. esialgse liitpäringu parempoolse päringuga, leides sisemisest andmebaasist unifitseerimisega Mary abikaasa, kelleks osutub John. Pärast ainsa allesjäänud päringu kustutamist saadakse tühipäring, mis tähendab, et saadi positiivne vastus. Lisaks vastusele *yes* trükitakse välja ka **arvutatud vastus**, mis koosneb esialgse liitpäringu (alakriipsuta) muutujate väärtustest.

Kastimudeli keeles tähendab vasakpoolse päringu kustutamine selle kasti pordi EXIT ühendamist parempoolse päringu kasti pordiga CALL:



1.4. Teadmiste esitamine reeglitena

Teadmised ei koosne mitte ainult faktidest, vaid siia kuulub ka teatud hulk uusi mõisteid ja nende vahelisi seoseid. Ülal leidsime kahe päringuga Alice'i isa kui tema ema abikaasa. Niisuguse liitpäringu kasutamise lihtsustamiseks on mõttekas defineerida predikaat, mille sisuks on see liitpäring. Antud juhul saame selliselt isa mõistet defineeriva reegli *father*, nõudes täiendavalt, et isa oleks meessoost:[7]

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

Paneme tähele, et sõna "isa" kasutatakse eesti keeles kahes tähenduses — kahekohalise seosena "*x* on *y* isa" ja omadusena "*x* on isa", täpsustamata isa ja lapse vahelist seost. Isa omaduse saame, võttes kaheargumendilise isa reegli *father*/2 projektsiooni teise argumendi järgi:[8]

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

Lapsele viitav muutuja *_Child* algab alakriipsuga, sest ta esineb isaksolemise omadust kirjeldavas reeglis *father*/1 ainult üks kord, s.t. ta on selles reeglis ühekordne muutuja (ingl. k. *singleton variable*). Vastasel juhul annab Prologi kompilaator hoiatuse: suure tõenäosusega on tegemist trükiveaga; näiteks nimed *Father*, *father* ja *FaTher* on kõik erinevad. Alternatiivina võib sellisel juhul kasutada nimetut muutujat '_'.

Ülesanne 5. Lõpetage ülesandes 3 alustatud oma suguvõsa andmebaas, lisades sellele reeglid venna, õe, tädi, onu, vanaisa ja vanaema leidmiseks (vastavalt predikaadid *brother*, *sister*, *aunt*, *uncle*, *grandfather* ja *grandmother*).

Ülesanne 6. Kirjutage programm, mis kirjeldab noormeeste ja neidude huvialad ning leiab, milline noormees ja neid sobivad omavahel.

Ülesanne 7. Realiseerige Prologi teadmistebaasina oma märkmik (nimi, aadress, telefon, sünnipäev). Esitage selle põhjal päringuid, näiteks

a) Kes elab Anne tänaval?
b) Kes on sündinud veebruaris?
c) Kes on samal kuul sündinud?

[7] Me kasutame programmides predikaatide ja muutujate nimedena ingliskeelseid termineid, et olla kooskõlas Prologi predikaatide standardsete nimetamistavadega.

[8] Predikaadile järgnev kaldkriipsuga eraldatud number tähistab predikaadi argumentide arvu, eristamaks seda teistest samanimelistest predikaatidest.

1.5. Prologi süntaks

Prologi süntaksiks on esimest järku loogika alamkeel — Horni loogika, milles atomaarsed predikaadid seotakse komadega ja tagurpidi implikatsioonidega ← või :−. **Atomaarsed predikaadid** on loogilise **termi** kujul: kas *aatom* või $aatom(term_1, \ldots, term_n)$. Tabelisse 1.1 on koondatud osa termide konstrueerimiseks vajalikke Prologi lihtsümboleid.

Element	Selgitus	Näide
muutuja	nimi, mis algab suurtähega	*X*, *X112*, *XYZ*
	või alakriipsuga	*_X*, *_*
aatom	nimi, mis algab väiketähega	*a*, *andres*
	või apostroofides sõne	*'Tõnu'*
	või kirjavahemärgid	=? =:
arv	nimi, mis algab numbriga	1, 10*a*, 3.14*E*1
kommentaar	sulgudes	/*...*/
	või rea lõpp alates sümbolist %	%rea lõpuni
loogikatehe	konjunktsioon	*p*, *q*
	disjunktsioon	*p* ; *q*
	eitus	\+*p*
		(varem *not p*)
	implikatsioon	*p*−>*q* ; *true*
		q:−*p*
võrdus	unifitseerimine	=, \= (eitus)
	objektide võrdus	==, \== (eitus)

Tabel 1.1. Valik Prologi baaselemente.

Lisaks Horni loogikas realiseeritud implikatsiooni- ja konjunktsioonitehtele on Prologile lisatud ka teised loogikatehted nagu disjunktsioon ja eitus. Sellegipoolest on eituse realiseerimine osutunud probleemiks — Horni disjunktid $p \leftarrow q_1, \ldots, q_n$ ei sisalda standardkujul negatiivseid atomaarseid predikaate, sest nool ← on lihtsalt positiivsete ja negatiivsete literaalide[9] eraldaja: positiivsed literaalid on viidud noolest vasakule ja negatiivsed literaalid noolest paremale. Puuduvad ka kvantorid ja ekvivalentsiseos — neid saab Horni disjunktide abil väljendada vaid kaudselt.

[9] Literaal on atomaarne predikaat või selle eitus.

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	$?{-}p_1, \ldots, p_n.$
		$:{-}p_1, \ldots, p_n.$
$\leftarrow$	tühipäring	*true*

Tabel 1.2. Horni loogika põhikonstruktsioonid.

1.6. Prologi kasutamine

Enne tööle asumist Prologi konkreetse süsteemiga on hea tunda Prologi lihtsamaid süsteemseid predikaate. Mõned neid sisaldavad päringud on koondatud tabelisse 1.3. Paneme tähele, et kõik päringud peavad lõppema punktiga — muidu ei võta süsteem meid jutule.

Tähtsamad vabavaralised Prologi süsteemid on hetkel SWI-Prolog ja GNU Prolog. Kui varem loeti programm arvuti mällu ja interpreteeriti seda täitmisel ühe alampäringu kaupa, siis tänu David Warrenile **kompileerib** tänapäeval enamus Prologe programmi teatavasse vahekeelde — **Warreni abstraktseks masinaks** (WAM), mille interpreteerimine on tunduvalt efektiivsem. Reeglina säilitatakse mälus ka predikaatide esialgsed kirjeldused, mida saab vaadata süsteemse predikaadiga *listing*.

Kuigi Prologi programmi saab kompileerida masinkoodi, käib töö programmiga tavaliselt interaktiivselt, kompileerides selle käsuga `?-[foo]` või `?-consult(foo)` Warreni abstraktseks masinaks.[10] Tuleb hoiatada, et Prologis programmeerimine erineb oluliselt programmeerimisest levinud imperatiivsetes keeltes nagu Pascal, C ja Java. Kui viimastes täidetakse programme determineeritult, arvutades sisendargumentide väärtuste põhjal väljundeid, siis Prologis ei tehta tihti vahet predikaatide sisend- ja väljundargumentidel. Näiteks predikaat *father*/2 on kasutatav nii lapse järgi isa arvutamiseks kui ka isa järgi lapse leidmiseks:[11]

```
?-father(alice,X).
```

[10] Eeldame, et kompileeritakse programmi *foo.pl*.

[11] Predikaadi *father* kasutusviisiks on seega *father(?Child,?Father)*. Küsimärk muutuja ees tähistab sisend- ja väljundargumenti. Plussmärk tähistab sisendargumenti, mis peab olema piisavalt määratud, ja miinusmärk väljundargumenti, mis peab olema väärtustamata muutuja. Järelikult on predikaadi *father* lubatud kasutusviisideks ka *father(+Child,–Father)* ja *father(–Child,+Father)*.

Süsteemse predikaadi kasutus[12]	Selgitus
`?-halt.`	Prologist väljumine
`?-X is 1059*513/100.`	aritmeetilise avaldise arvutamine
`?-atom_chars(aatom,List).`	aatomi teisendamine sümbolite listiks
`?-[foo].`	programmi *foo.pl* (või *foo*) kompileerimine
`?-pwd.`	töökataloogi näitamine
`?-chdir('h:/prolog').`	töökataloogi muutmine
`?-ls.`	failide kuvamine
`?-edit(foo).`	programmi redigeerimine
`?-listing.`	näita kõigi predikaatide kirjeldusi
`?-listing(member).`	näita kõigi predikaatide *member* kirjeldusi
`?-listing(member/2).`	näita predikaadi *member*/2 kirjeldusi
`?-trace.`	programmi jälgimise alustamine
`?-notrace.`	programmi jälgimise lõpetamine
`?-spy(married/2).`	silumispunkt predikaadil *married*/2
`?-nospy(married/2).`	silumispunkti väljalülitamine
`?-debug.`	silumise alustamine
`?-nodebug.`	silumise lõpetamine
`?-apropos(list).`	kogu informatsioon sõna "list" kohta
`?-help.`	abiteave
`?-help(member).`	abiteave predikaadi *member* kohta
`?-sort([1,...,10],L).`	listi [1, . . . , 10] järjestamine listiks *L*
`?-statistics.`	päringute täitmise statistika

Tabel 1.3. Prologi süsteemseid predikaate.

```
X=john
?-father(X,john).
X=alice ;
X=bob ;
X=carol ;
no
```

Samuti võivad predikaadil sisendargumendid üldse puududa. Sellisel juhul töötab predikaat oma määramispiirkonna väärtuste generaatorina:

```
?-father(X,Y).
X=john  Y=peter ;
X=linda Y=peter ;
...
```

[12] Aluseks on võetud SWI-Prolog.

Tuleb märkida, et Prologi predikaatide selline mittedetermineeritus on põhimõtteline — Colmerauer ehitas mittedetermineerituse Prologile sisse juba alguses, sest tavakeele konstruktsioonid on reeglina mitmeti mõistetavad.

Prologis programmeerimise erilisuse kohta väidab Richard O'Keefe, et teistes keeltes programmeerides kirjutab ta detailseid instruktsioone, Prologis kirjeldab ta aga objektide vahelisi seoseid — on selline tunne nagu häälestaks mingit süsteemi.

1.7. Programmi testimine ja silumine

Kui oleme programmi teksti valmis kirjutanud, siis algab selle testimine, s.t. korrektsuse kontroll. Programm on **korrektne** siis, kui see väljastab õiged ja ainult õiged vastused täpselt üks kord. Seda saab kontrollida vastuseid ühekaupa väljastades, vajutades iga vastuse järel semikoolonile:

```
?-father(X,Y).
X=john  Y=peter ;
...
```

Teiseks võimaluseks on kasutada *fail*-tsüklit:

```
?-father(X,Y), write(X-Y), nl, fail.
john-peter
...
no
```

Predikaat *write* väljastab oma argumendiks oleva termi ekraanile, *nl* teeb reavahetuse ja samaselt väär predikaat *fail* ütleb, et saadud vastus meid ei rahulda, ning käsib teha sama järgmise vastusega. Selle tulemusena väljastatakse kõik antud päringu vastused reakaupa ekraanile ja lõpetatakse teatega `no`. Kolmas võimalus on kasutada kõigi vastuste üheaegseks väljastamiseks spetsiaalseid **teist järku predikaate** *findall* ja *bagof*, mis koguvad kokku mingi päringu kõik vastused:

```
?-findall(X-Y,father(X,Y),List).
List=[john-peter,...]
```

Nende predikaatide kasutus on sarnane: nurksulgudes esitatav list moodustatakse muutuja *List* väärtuseks kõigist sellistest termidest $X-Y$, mis on koostatud päringu `?-father(X,Y)` vastustest X ja Y. Ka predikaat *bagof* annaks antud juhul sama tulemuse.

Ülesanne 8. Testige oma perekonna teadmistebaasi (ülesanne 5, lk. 10) reeglid nimetatud kolme meetodiga.

Juhul, kui programm annab täitmisel vigu või ootamatuid vastuseid, tuleks seda **siluda**, s.t. otsida sellest vigade põhjuseid. Jälgimisega silumisrežiim lülitatakse sisse predikaadiga *trace*:

```
?-trace, father(alice,X).
   Call: (7) father(alice, _G152) ? creep
   Call: (8) mother(alice, _G221) ? creep
   Exit: (8) mother(alice, mary) ? creep
   Call: (8) married(mary, _G152) ? creep
   Exit: (8) married(mary, john) ? creep
   Call: (8) male(john) ? creep
   Exit: (8) male(john) ? creep
   Exit: (7) father(alice, john) ? creep
X = john ;
   Fail: (8) male(john) ? creep
   Fail: (8) married(mary, _G152) ? creep
   Fail: (8) mother(alice, _G221) ? creep
   Fail: (7) father(alice, _G152) ? creep
no
```

Vasakul näeme portide nimesid, mille kaudu kasti siseneti või väljuti, sulgudes number näitab, kui sügaval kastides me oleme ja sõna “creep” täidetava päringu kõrval paremal ütleb, et silumine toimub sammhaaval (vajutati klahvile *enter* või *c*). Silumise käigus näeme tegelikke sisemisi muutujaid prefiksiga ’_G’, mitte esialgseid programmi ja päringu muutujaid: Prolog asendab esialgsed muutujad kompileerimise käigus ära.

Kui programmi sammhaaval silumine on tülikas, võib seda siluda hüpetega, vajutades klahvi *c* (või *enter*) asemel klahvile *l* (ingl. k. *leap*). Peatumiskohtade sisse- ja väljalülitamiseks on predikaadid *spy* ja *nospy*. Sellisel juhul peatutakse ainult sisselülitatud kastide portidel:

```
?-spy(mother/2).
?-father(alice,X).
   Call: (7) mother(alice, _G332) ? leap
   Exit: (7) mother(alice, mary) ? leap
X = john ;
   Fail: (7) mother(alice, _G332) ? leap
no
```

2. peatükk

Tehted

Avaldiste sisemised ja välimised esitused. Avaldiste suupärasemaks muutmine. Tehete tüübid. Tehte lisamine ja muutmine. Infikstehete kasutamine. Sümbolavaldised ja nende teisendamine.

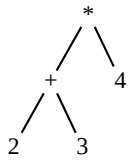
2.1. Avaldiste puud ja suluesitused

Programm koosneb atomaarsetest predikaatidest *aatom*($term_1, \ldots, term_n$) ja neile lisatud süsteemsetest tehetest nagu päise ja sisu eraldaja ':-' reegli keskel ja täidetava predikaadi tunnus '?-' (või ':-') päringu alguses. Programmis esinevad päringud täidetakse kohe kompileerimise ajal. Prologi kompilaator teisendab programmi standardsele sisemisele kujule, milleks on programmi süntaksipuule vastav suluesitus, mida saab vaadata predikaadiga *display* või *write_canonical*. **Suluesitus** vastab süntaksipuu läbimisele preorderis, s.t. puu iga tipu märgendi järel on sulgudes selle alampuude komadega eraldatud suluesitused. Nii näiteks aritmeetilise avaldise $(2 + 3) * 4$ suluesitust saame vaadata päringuga

```
?-display((2+3)*4).
*(+(2, 3), 4)
```

Suluesitus määrab üheselt avaldise süntaksipuu. Suluesitus on teksti lineaarsest esitusest parem, sest on üheselt mõistetav ning selles on lihtsam liikuda.

Avaldise sisemine ja välimine esitus ei lange tavaliselt kokku: kui sisemise esituse ülesandeks on esitada avaldist üheselt ja efektiivselt, siis välimise esituse eesmärgiks on lihtsustada programmi ja päringute kirjutamist ja vastuste



Joonis 2.1. Aritmeetilise avaldise $(2 + 3) * 4$ süntaksipuu.

lugemist. Eespool vaadeldud avaldise välimise esituse saame näiteks predikaadiga *write*:

```
?-write((2+3)*4).
(2+3)*4
```

Tuleb välja, et me kasutasimegi argumendina avaldise $(2 + 3) * 4$ välimist esitust, ent tulemus jääb samaks, kasutades argumendina avaldise sisemist esitust:

```
?-write(*(+(2,3),4)).
(2+3)*4
```

Resultaat on määratud spetsiaalsete teisendusreeglitega. Juhul, kui vastus ei ole ette nähtud üksnes inimesele lugemiseks, vaid seda kasutatakse ka mingi predikaadi argumendina, siis võib tekkida probleem vastuse uuesti sisselugemisega, sest teisendusreeglid ei tarvitse olla pööratavad. Suluesitus on aga alati üheselt mõistetav.

Ülesanne 9. Kirjutage programm, mis väljastab kompileerimisel sõna “Tere”.

2.2. Süsteemsed tehted

Avaldiste esitamiseks kasutatakse tehteid. mida saab vaadata predikaadiga *current_op*/3, kus esimeseks argumendiks on tehte prioriteet, teiseks argumendiks tehte tüüp ja kolmandaks argumendiks tehte nimi. Tehte nimeks (ehk tehtemärgiks) võib põhimõtteliselt olla suvaline aatom, millele tuleb lihtsalt määrata prioriteet (selleks on täisarv 1—1200) ja tüüp: kas prefiks-, infiks- või postfikstehe (vt. tabel 2.1). Näiteks plussmärk on defineeritud kahel viisil:

```
?-current_op(P,T,+).
P = 500  T = fx ;
P = 500  T = yfx
```

prefikstehe	*fx*, *fy*
infikstehe	*xfx*, *xfy*, *yfx*
postfikstehe	*xf*, *yf*

Tabel 2.1. Tehete tüübid.

Esiteks, unaarse plussina argumendi ees, ja teiseks, binaarse plusstehtena argumentide vahel, mõlemad on prioriteediga 500. Täht *f* näitab tehtemärgi asukohta argumentide *x* ja *y* suhtes: tüüp *fx* ütleb, et märk asub oma argumendi ees ja tüüp *xfx*, et märk on kahe argumendi vahel. Tehte prioriteet omistatakse ka selle poolt moodustatud alamavaldisele. Näiteks avaldise $2 + 3$ prioriteet on 500. Suluesituse moodustamisel on nõue, et tähega *f* tähistatud tehte prioriteet oleks rangelt suurem kui tähega *x* tähistatud alamavaldise prioriteet. Seda nõuet saab nõrgendada, kasutades tähe *x* asemel tähte *y*, mis ütleb, et argumendi prioriteet on kas tehte prioriteedist rangelt väiksem või on sellega võrdne. See võimaldab kasutada sama tehet iseenda alluvuses, kusjuures kõigepealt analüüsitakse tähega *y* tähistatud alamavaldis:

```
?-display(1+2+3).
+(+(1, 2), 3)
```

Et esmalt võeti sulgudesse summa $1 + 2$, siis tüübi *yfx* tõttu on binaarne plusstehe vasakassotsiatiivne. Sama käib ka korrutamistehte kohta, kuid selle prioriteet on väiksem, sest korrutamised sooritatakse vaikimisi enne liitmisi:

```
?-current_op(P,T,*).
P = 400  T = yfx
```

Avaldise vaikimisi analüüsi saab muuta ümarsulgudega, mis muudavad oma alamavaldise prioriteedi nulliks:

```
?-display(1+(2+3)).
+(1, +(2, 3))
```

Paneme tähele, et liitmise vasak- ja paremassotsiatiivne esitus on erinevad:

```
?-(1+2)+3=1+(2+3).
no
```

Võrdus tähendab avaldiste süntaksipuude võrdsustamist ehk **unifitseerimist**.

Juhul kui mingi aatom on defineeritud mitme erineva tehtemärgina, siis kompilaator valib nende hulgast ühe sobiva ja fikseerib selle antud avaldise süntaksianalüüsi jaoks. Valida on maksimaalselt kolme tehete põhitüübi vahel. Iga tüübi sees saab muuta tehte prioriteeti ja argumente, kuid muutmisel kustutatakse vana definitsioon ning asendatakse see uuega.

Sageli on probleemiks kahe päringu vahelise koma puudumine. Oletame, et me soovime pärast aritmeetilise valemi $2 + 3 = 5$ väljastamist sooritada reavahetuse predikaadiga *nl*:

```
?-write(2+3=5) nl.
ERROR: Syntax error: Operator expected
```

Viga tekib sellest, et **tühikul** on Prologis eritähendus: lisaks süntaktiliste elementide eraldamisele kasutatakse tühikuid ka lisavahede jätmiseks programmi ridades ja reaalguste tabuleerimiseks. Seepärast tuleks võtta tühik koma sarnase tehtena defineerimiseks apostroofidesse. Vastasel juhul oleks tühiku käsitlemine tehtena loogilises vastuolus liigsete tühikute kõrvaldamise algoritmiga. Eritähendus on ka **komal**: lisaks predikaatidevahelisele konjunktsioonitehtele eraldab koma atomaarse predikaadi argumente ja listi elemente.

Ülesanne 10. Koostage aritmeetiliste avaldiste süntaksipuud ja nende suluesitused:

a) $1 + (2 + 3)$;
b) $1 + 2 + 3$;
c) $2 * 8 + 5 * 10/4 - 7$;
d) $2 * X\hat{}2 + A * X - B$.

2.3. Tehte lisamine ja muutmine

Uut tehet saab defineerida või olemasoleva tehte parameetreid muuta predikaadiga *op*, millel on samad kolm argumenti, mis predikaadil *current_op*, kuid enam ei tohi kasutada väärtustamata muutujaid. Näiteks saab ära muuta binaarse liitmistehte prioriteedi ja assotsiatiivsuse:

```
?-op(300,xfy,+).
```

Nüüd võib jätta avaldises $(2+3)*4$ sulud kirjutamata, sest liitmise prioriteet on korrutamise omast madalam. Sulgude vajalikkust saab kontrollida infikspredikaadiga =/2, mis testib oma alamavaldiste unifitseeritavust. Erijuhul, muutujate puudumisel, kontrollitakse avaldiste süntaksipuude kokkulangevust:

```
?-2+3*4=(2+3)*4.
yes
```

Ümber saab defineerida ka näiteks unaarse plussmärgi, öeldes, et avaldise ees võib olla mitu plussi:

```
?-op(500,fy,+).
```

Nüüd võib $+(+3)$ asemel kirjutada $++3$, kus arvu märgi automaatse eemaldamise keelamiseks kasutatakse plussmärkide ümber tühikuid.

Selline tehete süsteem muudab programmid loetavamaks ning lähendab Prologi programmi matemaatilisele ja tavakeelele.

Ülesanne 11. Defineerige predikaat *ema_on* kui infikstehe nii, et võiksime fakti *mother(john, ann)* asemel (vt. ül. 5) kirjutada:

```
john ema_on ann.
```

Ülesanne 12. Kirjeldage postfikspredikaat *on_mees* (vt. ül. 5) nii, et võiks esitada järgmise päringu:

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

Ülesanne 13. Tehke oma sugulaste teadmistebaas ringi, kasutades predikaatide standardsete suluesituste asemel infiks- ja postfiksesitusi (vt. ül. 5).

Ülesanne 14. Lubage oma sugulaste teadmistebaasis järgmiste tavakeelsete infikspäringutega (vt. ül. 4 ja 5) sarnaseid päringuid:

```
?-Kes on mary lapsed.
?-Kes on Kellega abielus.
?-nimeta mõni Naine.
?-palun teata mulle johni Õe nimi.
```

Ülesanne 15. Kirjeldage oma sugulaste teadmistebaasis tehted ja predikaadid, mis lubavad tavakeelseid fakte ja päringuid (vt. ül. 4 ja 5), näiteks

```
Minu nimi on alice.
Minu ema on mary ja isa on john.
Mul on õde carol ja vend bob.

?-Mis on sinu nimi.
Mis = alice
?-Kes on sinu Isa ja Ema.
?-Kes on sinu õde.
?-Kes on sinu vend.
```

Ülesanne 16. Kirjeldage eestikeelsed arvuväljendid nullist miljonini, kasutades sobivaid tehteid nii, et Prolog oskab neid süntaktiliselt analüüsida. Näiteks

```
?-display(kolmsada viiskümmend kuus).
kolmsada(viiskümmend(kuus))
```

Ülesanne 17. Defineerige lausearvutuse tehted kujul '∼', and, or, '->' ja '<->' (vastavalt eitus, konjunktsioon, disjunktsioon, implikatsioon ja ekvivalents) nii, et neid saab kasutada predikaatide argumentidena. Näiteks

```
?-display(a and ~b -> ~(a<->b)).
->(and(a,~(b)), ~(<->(a, b)))
```

Predikaat	Selgitus
atom(X)	*X* on aatom
atomic(X)	*X* on aatom või täisarv
integer(X)	*X* on täisarv
number(X)	*X* on arv
float(X)	*X* on ujukoma-arv
var(X)	*X* on väärtustamata muutuja
nonvar(X)	*X* ei ole väärtustamata muutuja
compound(X)	*X* on liitterm
ground(X)	*X* on kinnine term

Tabel 2.2. Süsteemsed tüübipredikaadid.

2.4. Sümbolavaldised

Ehkki Prologi peetakse tüüpimata keeleks, on selles siiski teatud võimalused elemendi tüübi esitamiseks ja selle kontrollimiseks. Mõningad tüübid on ka süsteemselt defineeritud (vt. tabel 2.2).

Vaatleme näiteks polünoomide klassi defineerimist, millesse kuuluvad aritmeetilised avaldised, mis võivad sisaldada lisaks liitmise, lahutamise ja korrutamise tehetele ja täisarvudele ka muutujaid *a*, *b* ja *c*. Sellesse klassi kuuluvad näiteks avaldised $1 + 2$, $1 + a$ ja $2 * (a + b)$, kuid ei kuulu $1 + x$. Kõigepealt kirjeldame polünoomides kasutatavad muutujad *a*, *b* ja *c* ning täisarvud:

```
polynomial(a).
polynomial(b).
polynomial(c).
polynomial(X):-integer(X).
```

Täisarvulisuse kontrolli predikaat *integer* on tavaliselt süsteemselt defineeritud. Seejärel kirjeldame rekursiivselt polünoomides esinevad aritmeetilised tehted ja nende kasutamisreeglid:

```
polynomial(X+Y):-polynomial(X), polynomial(Y).
polynomial(X-Y):-polynomial(X), polynomial(Y).
polynomial(X*Y):-polynomial(X), polynomial(Y).
```

Seega tehte rakendamise tulemus on polünoom siis, kui selle kumbki operand osutub polünoomiks.

Ülesanne 18. Kontrollige predikaadi *polynomial* tööd eespool toodud polünoomidel.

Ülesanne 19. Kirjeldage eestikeelsete arvuväljendite klass arvudele nullist miljonini (vt. ül. 16) nii, et saame kontrollida arvuliste väljendite süntaktilist korrektsust. Näiteks

```
?-arv(kolmsada viiskümmend kuus).
yes
?-arv(kolmsada kakssada).
no
```

Lisaks arvuliste väljendite süntaksianalüüsile peab programm töötama nende generaatorina:

```
?-arv(X).
X=null ;
...
X=miljon
```

Ülesanne 20. Andke lausearvutuse valemi tüübikirjeldus (vt. ül. 17):

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

2.5. Sümbolteisendused

Prolog sobib hästi süntaktiliste avaldiste teisendamiseks. Lihtsustame näiteks eelmises punktis kirjeldatud polünoome:

$$(a+0)+2a = a+2a = a+a+a = 3a$$
$$1a+0b+(0+1)c = a+0+0c+1c = a+0+c = a+c$$

Lihtsustamise realiseerimiseks paneme kirja paar lihtsat teisendusreeglit:

```
simplify(X+0,X).  % x + 0 = x
simplify(1*X,X).  % 1x = x
simplify(X*1,X).  % x1 = x
```

Lisaks realiseerime aritmeetilise avaldise osade lihtsustamise ja teisendamist mittevajavad juhud nagu muutujad ja arvud:

```
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).
```

Tulemus on “suurepärane, kuid mitte lootusetu”:

```
?-simplify((a+0)+2*a,S).
S = a+2*a
?-simplify(1*a+0*b+(0+1)*c,S).
S = a+0*b+(0+1)*c
```

Ülesanne 21. Täiustage programmi *simplify*.

Teiseks näiteks võtame klassikalise avaldiste diferentseerimise. Püüame ära programmeerida järgmised näited:

$$2' = 0$$
$$x' = 1$$
$$(2x)' = 2$$
$$(x^2)' = 2x$$
$$(\sin x)' = \cos x$$

Vastavaks programmiks sobib näiteks

```
% diff(P,X,DP) on tõene siis,
%       kui DP on avaldise P tuletis X järgi
diff(X,X,1):-atomic(X).
diff(N,_X,0):-integer(N).
diff(F+G,X,DF+DG):-diff(F,X,DF), diff(G,X,DG).
diff(F*G,X,DF*G+F*DG):-diff(F,X,DF), diff(G,X,DG).
diff(X^2,X,2*X).
diff(sin(X),X,cos(X)).
```

Ülesanne 22. Testige programmi *diff* eespool toodud näidetel.

Ülesanne 23. Realiseerige predikaat *linearize* aritmeetilise avaldise teisendamiseks vasakassotsiatiivsele kujule. Näiteks

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

Ülesanne 24. Kirjutage programm, mis viib lausearvutuse valemi disjunktiivsele normaalkujule (vt. ül. 17 ja 20). Näiteks

```
?-dnk(a<->b, a and b or ~a and ~b).
```

Ülesanne 25. Kirjutage programm, mis tõlgib eestikeelsed arvuväljendid inglise keelde ja tagasi (vt. ül. 16 ja 19). Näiteks

```
?-tõlgi(kolmsada viiskümmend kuus,Eng).
Eng=three hundred and fifty-six
```

Ülesanne 26. Kirjutage programm oma sugulasi käsitlevate lausete tõlkimiseks inglise keelde ja tagasi (vt. ül. 4, 5 ja 15). Näiteks

```
?-tõlgi(alice on naine,T).
T=alice is female
?-tõlgi(mary õde on alice,alice is the sister of mary).
yes
```

Ülesanne 27. (A. Colmerauer) Defineerige sobivad sõnatehted, nii et predikaat *display* koostab suluesituse toodud ingliskeelsele lausele:

```
?-display(the expansion of government activities
          in canada as in many other countries
          is not something new).
```

Tõlkige lause eesti keelde ja defineerige ka selle jaoks sobivad sõnatehted. Kirjutage predikaat *translate*, mis tõlgib ingliskeelse lause automaatselt eesti keelde ja vastupidi:

```
?-translate(the expansion of government activities
            in canada as in many other countries
            is not something new, Estonian).
```

3. peatükk

Aritmeetika

Prologi sisseehitatud aritmeetika. Unifitseerimine ja resolutsioon. Peano aritmeetika. Programmi täielikkus.

3.1. Sisseehitatud aritmeetika

Arvutuste kiirendamiseks on Prologi keelde arvatud aritmeetikatehted ja -relatsioonid, neist olulisemad on toodud tabelites 3.1 ja 3.2. Binaarne predi-

Relatsioon	Prioriteet	Tüüp	Selgitus
is	700	*xfx*	parema poole arvutamine
=:=	700	*xfx*	aritmeetiline võrdlus
=\=	700	*xfx*	aritmeetiline mittevõrdus
<	700	*xfx*	väiksem
=<	700	*xfx*	väiksem või võrdne
>	700	*xfx*	suurem
>=	700	*xfx*	suurem või võrdne

Tabel 3.1. Prologi aritmeetikarelatsioonid.

kaat *is* arvutab välja oma paremal pool asuva aritmeetilise avaldise väärtuse ning unifitseerib saadud väärtust termiga endast vasakul. Predikaat =:= erineb predikaadist *is* selle poolest, et arvutab enne unifitseerimist välja ka vasakpoolse aritmeetilise avaldise väärtuse. Näiteks

```
?-2+3=:=3+2.
yes
```

Analoogiliselt töötavad relatsioonid suurem, väiksem, mittesuurem (=<) ja mitteväiksem (>=). Predikaat =\= tähendab aritmeetilise võrdluse =:= eitust. Paneme tähele, et unifitseerimispredikaat = ja selle eitus \= erinevad aritmeetilistest predikaatidest, sest nad ei arvuta oma alamavaldisi välja, vaid püüavad võrdsustada avaldisi süntaktiliselt, asendades nendes vabu muutujaid:

```
?-2+3=3+2.
no
?-2+3\=3+2.
yes
```

Objektide võrdlemise predikaat == ja selle eitus \== võrdlevad samuti avaldisi süntaktiliselt, kuid jätavad vabad muutujad asendamata:

```
?-2+X==2+Y.
no
```

Tehe	Prioriteet	Tüüp	Selgitus
+	500	*fx*	plussmärk
+	500	*yfx*	liitmine
–	500	*fx*	miinusmärk
–	500	*yfx*	lahutamine
*	400	*yfx*	korrutamine
/	400	*yfx*	jagamine
//	400	*yfx*	täisarvuline jagamine
mod	400	*yfx*	jääk
^	200	*xfy*	astendamine
**	200	*xfx*	astendamine

Tabel 3.2. Valik Prologi aritmeetikatehteid.

Aritmeetikatehetena on defineeritud tehted liitmisest astendamiseni ja trigonomeetriast bititeheteni. Aritmeetilised tehted ja predikaadid on Prologile lisatud selleks, et muuta programmide täitmist efektiivsemaks. Näiteks saab kirjeldada predikaadi, mis summeerib kaks arvu:

```
sum(X,Y,Z):-Z is X+Y.
?-sum(1,2,X).
X=3
```

Katse kasutada sama programmi lahutamiseks annab aritmeetilise vea:

```
?-sum(X,2,3).
ERROR: Arguments are not sufficiently instantiated
```

Probleem on selles, et ei ole võimalik välja arvutada aritmeetilist avaldist $X+2$, milles muutuja X on väärtustamata. Hiljem näeme, et Peano aritmeetikas on liitmine pööratav, kuid tehteid täidetakse aeglasemalt.

Ülesanne 28. Koostage programm, mis väljastab sisestatud arvude korrutise. Näiteks

```
?-korrutis.
sisesta X: 2.
sisesta Y: 3.
2*3=6
```

Ülesanne 29. Koostage programm, mis kontrollib, kas on tegemist paarisarvuga. Näiteks 4 on paarisarv:

```
?-even(4).
yes
```

Ülesanne 30. Koostage rekursiivne programm, mis arvutab etteantud arvu faktoriaali. Näiteks

```
?-fact(5,X).
X=120
```

Ülesanne 31. Koostage programm, mis leiab esimese n arvu ruutude summa. Näiteks

```
?-sum(5,X).
X=55
```

Ülesanne 32. Teisendage arv rooma numbriks. Näiteks

```
?-roman(2003,'MMIII').
yes
?-roman(1999,R).
R=MIM ;
R=MXMIX ;
R=MCMIC ;
R=MCMXCIX
```

3.2. Unifitseerimine

Loogilise programmeerimise ja loogilise teoreemitõestamise aluseks on termide võrdsustamine ehk unifitseerimine. **Robinsoni determineeritud unifitseerimisalgoritm** püüab võrdsustada terme s ja t, asendades nendes indiviidmuutujaid sobivate termidega:

Sisend: termid s ja t[13]
Väljund: termide s ja t unifitseerija üldkuju θ
Algoritm:

1. $\theta := \{\}$.
2. Leia termide $s\theta$ ja $t\theta$ vasakult esimene erinevus.
3. Konstrueeri erinevust põhjustavatest alamtermidest ebakõlapaar (w, u).
4. Ebakõlapaar on lihtne siis, kui see on pööratav kujule (x, v), kus x on muutuja ja v on term, mis ei sisalda muutujat x.
5. $\theta := \theta\{x/v\}$.
6. Kui $s\theta = t\theta$, siis termid on unifitseeritavad.
7. Kui ebakõlapaar pole lihtne, siis termid pole unifitseeritavad.
8. Korda samme 2—8.

Algoritmi alguses on otsitav substitutsioon θ tühi. Algoritmi igal sammul leitakse indiviidmuutuja x, mille asendamine termiga v lähendab terme s ja t teineteisele, arvutatakse θ kompositsioon substitutsiooniga $\{x/v\}$ ja korratakse sama protsessi uuesti. Iteratsioon lõpeb kas termide s ja t unifitseerija üldkuju väljastamisega sammul 6 või tõdemusega sammul 7, et antud termid ei ole unifitseeritavad. Püüame näiteks võrdsustada termid

$$f(x, g(b)) \quad \text{ja} \quad f(a, y).$$

Eeldame, et f on funktsionaalsümbol, a ja b (kui tähestiku esimesed tähed) indiviidkonstandid ning x ja y (kui tähestiku viimased tähed) indiviidmuutujad. Meie näite korral muutub termide $f(x, g(b))$ ja $f(a, y)$ võrdus[14] algoritmi täitmise käigus järgmiselt:

$$\begin{array}{ll} f(x, g(b)) = f(a, y) & \text{(leitakse substitutsioon } \{x/a\}) \\ f(a, g(b)) = f(a, y) & \text{(leitakse substitutsioon } \{y/g(b)\}) \\ f(a, g(b)) = f(a, g(b)) & \text{(termid on võrdsed).} \end{array}$$

Algoritmi töö lõpus väljastatakse termide unifitseerija üldkuju kui leitud substitutsioonide kompositsioon

$$\theta = \{x/a\}\{y/g(b)\} = \{x/a, y/g(b)\}.$$

Ülesanne 33. Tooge näide termidest, mis ei ole unifitseeritavad.

[13] Term on muutuja või aatom või muutujatest ja aatomitest koosnev struktuur.
[14] Eesmärgiks on termide süntaktiline kokkulangemine.

Ülesanne 34. Arvutage välja aritmeetilise avaldise väärtus, milles numbrite asemel on nende ingliskeelsed nimetused: one, two, three, four, five, six, seven, eight, nine, zero. Näiteks

```
?-compute(one+(two+three*eight),X).
X=27
?-compute((one-five)-three*seven,X).
X=-25
?-compute(one+(two-three*eight)/four,X).
X=-4.5
```

3.3. Peano aritmeetika

Peano aritmeetikas ei kasutata tavalisi arve, vaid sümbolarve, mis saadakse, rakendades konstandile 0 lõplik arv kordi ühekohalist funktsiooni s:[15]

$$0, s(0), s(s(0)), s^3(0), s^4(0), \ldots$$

Me interpreteerime sümbolarvu $s^n(0)$ kui harilikku täisarvu n, kus astendaja n näitab, mitu s-i on nulli ees. Lihtne on ära kirjeldada kõik süntaktiliselt korrektsed sümbolarvud:

```
nat(0).
nat(s(X)):-nat(X).
```

S. t. null on sümbolarv ja $s(X)$ on sümbolarv siis, kui X on sümbolarv. Sümbolarvude liitmise defineerime Peano aritmeetika reeglitega:

```
add(X,0,X).                    % x + 0 = x
add(X,s(Y),s(Z)):-add(X,Y,Z).  % x + s(y) = s(x + y)
```

Predikaadiga *add* saab liita näiteks sümbolarvud, mis interpreteerivad tavalisi arve 1 ja 2:

```
?-add(s(0),s(s(0)),X).
X = s(s(s(0)))
```

Tulemuseks on arvule 3 vastav sümbolarv. Kuid predikaat *add* sobib ka arvude lahutamiseks:

```
?-add(X,s(s(0)),s(s(s(0)))).
X = s(0)
```

Järgmises punktis demonstreerime, kuidas see tulemusi saadi.

[15] Peano aritmeetika antakse konstandiga 0 ja ühekohalise funktsiooniga $s(x)$, nii et kehtivad järgmised aksioomid: 1) iga $x \in \mathbb{N}$ korral $0 \neq s(x)$, 2) kui $s(x) = s(y)$, siis $x = y$, 3) hulga $\mathbb{N}$ iga alamhulk, mis sisaldab elemendi 0 ja koos oma mistahes elemendiga x ka elemendi $s(x)$, ühtib hulgaga $\mathbb{N}$; sellise operatsiooniga hulka $\mathbb{N}$ nimetatakse naturaalarvuvallaks, elementi $s(x)$ elemendi x järglaseks ja aksioomi 3 induktsiooniprintsiibiks.

Ülesanne 35. Kirjeldage predikaat *even*, mis kontrollib, kas sümbolarvu interpretatsiooniks on paarisarv. Näiteks

```
?-even(s(s(s(s(0))))).
yes
```

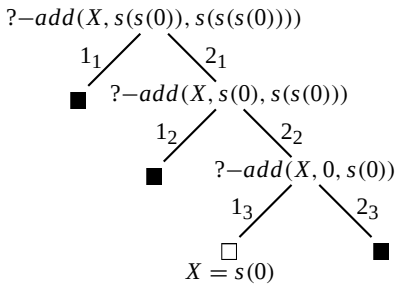
Ülesanne 36. Koostage programm, mis arvutab Peano aritmeetikas faktoriaali. Näiteks

```
?-factorial(s(s(s(0))),X).
X=s(s(s(s(s(0)))))
?-factorial(X,s(s(s(s(s(0)))))).
X=s(s(s(0)))
```

3.4. Resolutsioon

Päringu täitmist saab kirjeldada resolutsioonimeetodiga. Vaatame, kuidas täidetakse päringut `?-add(X,s(s(0)),s(s(s(0))))`. Selle jälgimiseks nummerdame programmi *add* kirjeldused:

1. `add(X,0,X).`
2. `add(X,s(Y),s(Z)):-add(X,Y,Z).`



Joonis 3.1. Päringu $?{-}add(X, s^2(0), s^3(0))$ resolutsioonipuu.

Vastav resolutsioonipuu on joonisel 3.1. Üks **resolutsioonisamm** seisneb päringu unifitseerimises temale vastava predikaadi mingi kirjelduse päisega ja päringu asendamises selle kirjelduse sisuga (rakendades viimasele unifitseerimisel tehtud asendusi). **Resolutsioonipuu** saame siis, kui koondame päringu täitmise kõikvõimalikud sammud üheks puuks, kus mingi tipu alluvad esitavad päringu täitmist sama predikaadi erinevate kirjelduste korral ning puu juurest lehtedesse minevad teed esitavad järjestikuseid päringu asendusi. Eesmärgiks on jõuda allapoole liikudes faktideni, muutes esialgse päringu tühjaks. Tühipäringut väljendatakse **tühja kastiga**. Juhul, kui päringut ei õnnestu vastava predikaadi mõne kirjeldusega unifitseerida, joonistame seest **täidetud kasti**.

Puu igast tipust väljub nii mitu serva, kui palju on päringu esimesele alampäringule vastaval predikaadil kirjeldusi. Antud puu korral väljub igast tipust (v.a. lehest) kaks serva, vasakpoolne neist on tähistatud numbriga 1 ning vastab tipus asuva päringu unifitseerimisele predikaadi *add* esimese kirjeldusega ja parempoolne on tähistatud numbriga 2 ning vastab päringu unifitseerimisele predikaadi *add* teise kirjeldusega. Numbrite 1 ja 2 alaindeksid määravad indeksi, mida me kasutame vastava predikaadi kirjelduse **muutujate eraldamiseks**: kirjeldustes esinevad muutujad peavad olema unikaalsed kõigi päringute ja kirjelduste suhtes resolutsioonipuu juurest vastava tipuni.

Puu harus märgendiga 1_1 on kirjelduse *add(X,0,X)* muutujate eraldamise tulemuseks fakt *add(X_1,0,X_1)*, sest muutujat X kasutatakse esialgses päringus. Tulemuseks on täidetud kast, sest saadud kirjeldus ja esialgne päring pole unifitseeritavad.

Naaberharus märgendiga 2_1 saame programmi teise kirjelduse muutujate eraldamisel reegli *add(X_1,s(Y),s(Z)):–add(X_1,Y,Z)*, mis annab päringuga unifitseerides substitutsiooni $\{X_1/X,\ Y/s(0),\ Z/s(s(0))\}$ ja päringu asendamisel reegli parema poolega uue päringu `?-add(X,s(0),s(s(0)))`.

Korrates saadud päringuga sama tegevust harus 1_2, saame muutujate eraldamisel kirjelduse *add(X_2,0,X_2)*, mis ei unifitseeru antud päringuga. Harus 2_2 saame muutujate eraldamisel kirjelduse *add(X_2,s(Y_2),s(Z_2)):–add(X_2,Y_2,Z_2)*, mis annab substitutsiooni $\{X_2/X,\ Y_2/0,\ Z_2/s(0)\}$ ja uue päringu `?-add(X,0,s(0))`.

Puuharus märgendiga 1_3 annab programmi esimese kirjelduse muutujate eraldamine fakti *add(X_3,0,X_3)*, mille unifitseerimine viimase päringuga annab substitutsiooni $\{X/s(0),\ X_3/s(0)\}$ ja tühipäringu $\square$. Kokkuvõttes saime esialgse päringu jaoks vastuse $X = s(0)$.

Vastava parempoolse puuharu 2_3 täitmisel saame muutujate eraldamisel kirjelduse *add(X_3,s(Y_3),s(Z_3)):–add(X_3,Y_3,Z_3)*, mis ei unifitseeru antud päringuga.

Programm töötab ka juhul, kui kasutame päringus arvu $s^3(0)$ asemel arvu $s^3(3)$, mis ei ole sümbolkujul:

```
?-add(X,s(s(0)),s(s(s(3)))).
X=s(3)
```

Selle puuduse saab kõrvaldada, lisades predikaadi *add* kirjeldustele tüübikontrolli, mis lubab liita ainult sümbolarve:

```
add(X,0,X):-nat(X).
add(X,s(Y),s(Z)):-add(X,Y,Z).
```

Võttes predikaadi *add* poole pöördumisel muutujateks nii esimese kui ka teise argumendi, näeme, et see oskab lisaks sümbolarvude liitmisele ja lahutamisele genereerida summade vastavaid kombinatsioone:

```
?-add(X,Y,s(s(s(0)))).
X = s(s(s(0)))  Y = 0 ;
X = s(s(0))  Y = s(0) ;
X = s(0)  Y = s(s(0)) ;
X = 0  Y = s(s(s(0)))
```

Ülesanne 37. Koostage päringu `?-add(X,Y,s(s(s(0))))` resolutsioonipuu.

3.5. Täielikkus

Mingit programmi kirjutades peame tavaliselt silmas teatavat argumentide ja tulemuse väärtuste korteežide hulka, millel programm peab töötama õigesti. Seda hulka nimetatakse programmi **taotletud mudeliks**. Näiteks predikaati *nat* kirjeldades pidasime silmas kõigi sümbolarvude hulka $\mathbf{N} = \{0, s(0), s^2(0), s^3(0), \ldots\}$. Sellisel juhul võib proovida tõestada programmi **täielikkust**, s.t. programmi võimet arvutada taotletud mudeli väärtuseid ja ainult neid. Näitame, et predikaat *nat* on täielik kõigi sümbolarvude hulga **N** jaoks.

Joonistades välja päringu `?-nat(X)` resolutsioonipuu näeme, et programm *nat* on **korrektne**, sest väljastab vastustena ainult hulga **N** elemente.

Programmi *nat* täielikkuse jaoks peame veel näitama, et selle vastuste hulk katab ära kogu hulga **N**. Valime suvalise arvu $s^n(0)$ ($n \geqslant 0$) ja joonistame resolutsioonipuu päringule `?-nat(s^n(0))`. Rakendades n korda reeglit *nat(s(X)):–nat(X)* jõuame puus alla liikudes päringuni `?-nat(0)`. Rakendades viimasele fakti *nat(0)*, saamegi vastuse *yes*.

Ülesanne 38. Tõestage predikaadi *even* täielikkus mudeli $\{0, s^2(0), s^4(0), \ldots\}$ jaoks (vt. ül. 35).

4. peatükk

Listid

Listi ehitus. Standardsed listipredikaadid. Listi summeerimine ja järjestamine. Listide kasutamine lausete süntaksianalüüsis.

4.1. Listi esitus

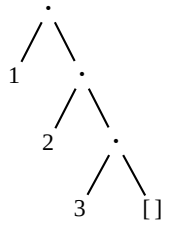
Lisaks struktuuride suluesitustele on list Prologi tähtsuselt teine andmestruktuur. See on praktiliselt ainus võimalus andmete koondamiseks massiividesse — nii ühe- kui ka kahemõõtmelistesse — sest massiivi mõiste Prologis reeglina puudub.

List on elementide järjestatud kogum, mis sisemiselt moodustatakse binaarse tehte '.' abil rekursiivse struktuurina. Listi lõpetab **tühilist**, mida tähistatakse tühjade nurksulgudega [ ]. Tehte esimeseks operandiks on listi esimene element ehk **päis** (ingl. k. *head*) ja teiseks operandiks listi ülejäänud elemendid ehk **saba** (ingl. k. *tail*). Põhimõtteliselt saab sellise rekursiivse struktuuri moodustada suvalise kahekohalise funktsiooniga. Näiteks

$$f(a,\ f(b, nil))$$

on käsitletav kui list, mis koosneb elementidest *a* ja *b*. Lisaks sisemisele '.'-esitusele saab liste programmis esitada ka komadega eraldatud elementide loeteludena, mis on võetud nurksulgudesse. Nii saab moodustada näiteks arvude listi

[1, 2, 3],



kuid võimalikud on ka aatomite ja listide listid:

[a,b,c],
['Aadam','Eeva'],
[[a,b],f(2,1),[[ ]]].

Nagu eespool mainitud, on listis tähtis elementide järjekord. Näiteks listid $[a, b, c]$ ja $[b, c, a]$ koosnevad samadest elementidest, kuid on listidena erinevad:

```
?-[a,b,c]=[b,c,a].
no
```

Listina on lihtne esitada kahemõõtmelist massiivi, võttes listi elementideks massiivi ridade elementide listid. Näiteks

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = [[1, 0, 0], [0, 1, 0], [0, 0, 1]].$$

Lisaks listi esitusele nurksulgudesse paigutatud loendina kasutab nurksulgusid ka listi teine välimine Lispi-laadne esitus, milles listi saba eraldatakse listi esimestest elementidest tehtemärgiga '|' (ingl. k. *construction*). Tabelis 4.1 on toodud mõnede listide ekvivalentsed esitused.

Sisekuju	Märgiga '|'	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 4.1. Listi ekvivalentsed esitused.

Niisiis on üks võimalus listi moodustamiseks esitada see elementide loendina nurksulgudes. Alternatiivina võiks listi moodustada predikaadi *list*/1 tõesuspiirkonnana, kus predikaadi kirjelduste järjestus annab ka elementide järjestuse listis:

```
list(a).
list(b).
list(c).
```

Sellise listi saab viia standardkujule predikaadiga *findall* või *bagof* (millest oli juttu eespool):

```
?-findall(X,list(X),List).
List = [a, b, c]
```

Listi moodustamiseks ei sobi aga analoogiline predikaat *setof*, mis moodustab elementidest järjestatud hulga, kaotades sellest korduvad elemendid.

Lispi-laadses tähistuses saab anda listi tüübikirjelduse:

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

See tähendab, et tühilist on list ja suvalisest päisest ning sabast koosnev struktuur osutub listiks siis, kui seda on saba.[16]

Listid jagunevad **täielikeks**, mille elementide arv on määratud, ja **osalisteks**, mille pikkus pole teada. Näiteks list $[a, b, c, X]$ on täielik, sest see koosneb neljast elemendist, kuid list $[a, b, c|Xs]$ pole täielik, sest ei ole teada saba *Xs* pikkus. Osalised listid on ka $[X|Ys]$ ja $[X, X|\_]$.

Predikaat *is_list* loeb listiks ka tavalise muutuja *X*. Sellise kõdunud juhu saab kõrvaldada süsteemset predikaati *nonvar* kasutades, mis kontrollib, kas tegemist pole väärtustamata muutujaga:

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

Lisaks listide moodustamisele, saab nendega sooritada ka standardseid tehteid: elemente liita, muuta ja kustutada. Üldjuhul tuleb selleks moodustada uus list, kopeerides osa vaadeldava listi elemente uude listi. Olgu meil näiteks olemas list $Xs = [a, b, c]$. Kõige lihtsam on lisada elemente selle algusesse. Elemendi *d* lisamiseks listi *Xs* algusesse moodustame uue listi *Ys*:

```
?-Xs=[a,b,c], Ys=[d|Xs].
Ys=[d,a,b,c]
```

Kuid vajaduse korral saab moodustada ka listi, kus uus element on lisatud etteantud listi elementide lõppu. Näiteks, kui me teame, et listis *Xs* on kolm elementi, siis saame need unifitseerimisega kätte ja moodustame listi *Ys*, kus nende järel on element *d*:

```
?-Xs=[a,b,c], Xs=[A,B,C], Ys=[A,B,C,d].
Ys=[a,b,c,d]
```

[16] Muuutuja *Xs* sufiks 's' näitab, et see muutuja tähistab listi (täht 's' sõna lõpus on inglise keeles mitmuse tunnus).

Märk '|' annab üldisema võimaluse listile lisamiseks: peale elementide lisamise olemasoleva listi lõppu saab moodustada ka lisamiste ahelaid, lisades kõigepealt listi lõppu ühe elemendi ja seejärel teise elemendi:

```
?-Ys=[a,b,c|Zs], Zs=[d|Ws], Ws=[e].
Ys=[a,b,c,d,e]
```

Listi $Xs = [a, b, c]$ muutmine käib põhimõtteliselt sama moodi. Asendame näiteks listis *Xs* elemendi *b* elemendiga *d*, saades tulemuseks listi *Ys*:

```
?-Xs=[a,b,c], Xs=[A,B,C], Ys=[A,d,C].
Ys=[a,d,c]
```

Ja lõpuks kustutame listist *Xs* teise elemendi, saades tulemuseks listi *Ys*:

```
?-Xs=[a,b,c], Xs=[A,B,C], Ys=[A,C].
Ys=[a,c]
```

Me andsime näidetes uutele listidele uued nimed, sest samast listist ei saa põhimõtteliselt kustutada ega seda listi muuta. Lisada saab vaid osalisse listi, mille elementide arv võib suureneda. Viimaseid kasutatakse sisend- ja väljundvoogude kirjeldamiseks, näiteks teksti sisselugemiseks.

Ülesanne 39. Milline on järgmise päringu korral saadav vastus?

a) `?-[a,b,c,d]=[X|Y].`
b) `?-[a,b,c,d]=[X,Y,Z].`
c) `?-[a,b,c,d]=[X,Y|Z].`
d) `?-[a,[b,c],d]=[X,Y,Z].`
e) `?-[a,[b,c]]=[X,Y|Z].`

Ülesanne 40. Koostage programm listi elementide reakaupa trükkimiseks. Näiteks

```
?-print_list([a,b,c]).
a
b
c
yes
```

4.2. Standardpredikaadid

Et list on Prologis andmete hoidmise standardstruktuur, siis tutvustame lühidalt olulisi süsteemseid listipredikaate. Predikaat *member* kontrollib elemendi kuulumist listi:

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

Predikaadi teiseks argumendiks on list ja esimeseks argumendiks objekt, mille kuulumist listi kontrollitakse. Predikaadi *member* kirjeldus ütleb, et objekt on kas listi esimene element (kirjelduse 1. rida) või esineb listi sabas. Predikaadi *member* standardsel kasutusel kontrollitakse etteantud objekti kuulumist listi:

```
?-member(b,[a,b,c]).
yes
```

Lisaks saab predikaati *member* kasutada ka listi elementide genereerimiseks:

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

Samuti saab seda kasutada listide genereerimiseks:

```
?-member(a,Xs).
Xs=[a] ;
Xs=[_,a] ;
...
```

Teise standardpredikaadina vaatame listist eemaldamist. Predikaadi kirjelduse alguses on hea kommentaarina ära tuua selle argumentide tähendused ja predikaadi eeldatav kasutus, sest Prologi mittedetermineerituse tõttu võib predikaatidel olla mitmeid kõrvalkasutusi, mis ei ole algselt ette planeeritud:

```
% select(X,Xs,Ys) on tõene siis,
%       kui eemaldades listist Xs elemendi X saame listi Ys
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).
```

Predikaat *select* erineb predikaadist *member* selle poolest, et võtab elemendi leidmisel listist kolmanda argumendi väärtusena välja ka listi saba, lõpetades sellega listi eelmistest elementidest moodustatud osalise listi:

```
?-select(b,[a,b,c],Xs).
Xs=[a,c]
```

Lisaks elementide eemaldamisele saab predikaati *select* kasutada ka etteantud listi lisamiseks. Näiteks lisame listile [a, c] elemendi b, saades tulemuseks:

```
?-select(b,Xs,[a,c]).
Xs=[b,a,c]
```

Huvitav on see, et predikaat *member* avaldub predikaadi *select* kaudu:

```
member(X,Xs):-select(X,Xs,_).
```

Kui mingi element esineb listis kahes kohas, siis predikaat *select* eemaldab neist alguses esimese ja seejärel teise, taastades esimese:

```
?-select(a,[a,b,a,c],Xs).
Xs=[b,a,c] ;
Xs=[a,b,c]
```

Mõlema elemendi üheaegseks eemaldamiseks listist saab kasutada predikaati *delete*:

```
% delete(Xs,X,Ys) on tõene siis,
%       kui eemaldades listist Xs elemendi X kõik esinemised saame listi Ys
delete([],_,[]).
delete([X|Xs],X,Ys):-delete(Xs,X,Ys).
delete([X|Xs],Y,[X|Zs]):-delete(Xs,Y,Zs).
```

Predikaat *delete* kopeerib uude listi kõik esialgse listi elemendid peale elemendi *X*; listi läbimine katkeb alles selle lõpus, mitte konkreetsel elemendil nagu predikaatide *member* ja *select* korral:

```
?-delete([a,b,a,c],a,Xs).
Xs=[b,c]
```

Kahe listi ühendamiseks kasutatakse predikaati *append*, mille kirjeldus teatab, et tühilisti ühend listiga on sama list, mittetühja listi korral on aga ühendiks list, mille esimeseks elemendiks on listi esimene element ja sabaks on listi ülejäänud elementide ühend etteantud listiga:

```
% append(Xs,Ys,Zs) on tõene siis,
%       kui Zs on listide Xs ja Ys ühend
append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).
```

Predikaadi *append* põhifunktsiooniks on ühendada kaks etteantud listi:

```
?-append([a,b,c],[d],Xs).
Xs=[a,b,c,d]
```

Samas on *append* kasutatav ka teistes olukordades nagu listi eraldamiseks kaheks listiks. Richard O'Keefe väidab, et kuigi *append* töötab vastavalt kirjeldusele, ei ole selle käitumine alati intuitiivne. Näiteks tühilisti ühendamisel muutujaga *Xs* saame ühe üldlahendi:

```
?-append([],Xs,Xs).
Xs = _G239 ;
no
```

Kuid muutuja *Xs* ühendamisel tühilistiga saame mitu erilahendit:

```
?-append(Xs,[],Xs).
Xs = [] ;
Xs = [_G296] ;
Xs = [_G296, _G299] ;
...
```

Ülesanne 41. Millised vastused väljastab (ja millises järjekorras) päring:

```
?-append(Xs,Ys,[a,b,c,d]).
```

Ülesanne 42. Avaldage predikaat *member*/2 predikaadi *append*/3 kaudu.

Listide kasutamisel on oluline veel predikaat *length*, mis leiab etteantud listi pikkuse (tühilisti [ ] pikkus on 0 ja listi $[X|Xs]$ pikkus on ühe võrra suurem listi *Xs* pikkusest):

```
length([],0).
length([X|Xs],N):-
  length(Xs,N1),
  N is N1+1.
```

Näiteks:

```
?-length([a,b,c],N).
N=3
```

Kuid tihti kasutatakse seda predikaati etteantud pikkusega listi moodustamiseks:

```
?-length(Xs,3).
Xs=[_,_,_]
```

Kolmest sellisest listist saab moodustada 3×3 ruudustiku trips-traps-trulli mängimiseks, et seda hiljem täita ristide ja nullidega.

Ülesanne 43. Realiseerige predikaat *last*, mis väidab, et antud element on listis viimasel kohal. Näiteks

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

Ülesanne 44. Kirjutage programm, mis trükib listi elemendid välja ridade kaupa (vt. ül. 40):

a) lisades iga elemendi ette selle järjekorranumbri listis;
b) lisades iga elemendi ette selle järjekorranumbri listis ja nummerdades alamlistid omaette, väljastades need suurema taandega kui põhilisti elemendid;
c) lisades iga elemendi ette selle järjekorranumbri listis ja nummerdades alamlistid samas numeratsioonis, mis põhilist, kuid suurema taandega kui põhilisti elemendid.

Näiteks päringu `?-print_list([a,[1,2],b,c])` korral saame järgmised tulemused:

a)
```
1. a
2. [1,2]
3. b
4. c
yes
```

b)
```
1. a
   1. 1
   2. 2
2. b
3. c
yes
```

c)
```
1. a
   2. 1
   3. 2
4. b
5. c
yes
```

Osaliste listide abil on lihtne kopeerida ühte listi teiseks: tühilisti kopeerime tühilistiks, aga päisest ja sabast koosneva listi kopeerime elementide *X* kaupa rekursiivselt:

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

Näiteks listi [*a*, *b*, *c*] kopeerimisel on tulemuseks esialgne list:

```
?-copy_list([a,b,c],List).
List=[a,b,c]
```

Kopeerimise suund ei ole antud juhul oluline. Muidugi oleks saanud listi kopeerimise teha lihtsamini unifitseerimise abil:

```
?-List=[a,b,c].
List=[a,b,c]
```

Kuid listide kopeerimisprogrammil on ka teisi rakendusi, sest kopeerimise ajal on võimalik elemente teisendada.

Ülesanne 45. Teisendage programmi *copy_list* nii, et teise argumendi väärtuseks on esimese argumendiga sama pikk list

a) mis koosneb ainult elementidest *a*;
b) milles iga element *E* on asendatud struktuuriga *el(E)*.

Näiteks päringu `?-copy_list([a,b,c],List)` korral saame järgmised tulemused:

a) `List=[a,a,a]`

b) `List=[el(a),el(b),el(c)]`

Ülesanne 46. Kirjutage programm, mis teisendab ingliskeelsete numbritega listi eestikeelseks ja vastupidi. Näiteks

```
?-trans([one,two,three],Est).
Est=[üks,kaks,kolm]
?-trans(Eng,[üks,kaks,kolm]).
Eng=[one,two,three]
```

Ülesanne 47. Realiseerige predikaat *flatten*, mis kaotab listist alamlistid, tõstes nende elemendid põhilisti. Näiteks

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

Ülesanne 48. Realiseerige Prologis trips-traps-trulli mäng.

4.3. Summeerimine

Arvude listi elemente on võimalik kokku liita või järjestada. Liitmiseks vaatame listi elementhaaval läbi vasakult paremale, liites igal sammul listi esimese elemendi hetkesummale:

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

Muutuja *Sum* sisaldab listi läbivaadatud elementide liitmisavaldist. Jõudes tühilistini arvutame selle avaldise summa predikaadiga *is* välja ja väljastame ekraanile. Päringuga `?-sum([1,2,3],0)` alustame listi $[1, 2, 3]$ elementide summeerimist nullist, saades järgmise alampäringute loetelu:

```
?-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).
```

Jõudes rekursiooniga tühilistini, oleme konstrueerinud avaldise $0 + 1 + 2 + 3$, mille arvutame välja predikaadiga *is*. Tegelikult on see programm kasutu, sest ei tee arvutatud summat teistele programmidele kättesaadavaks. Traditsiooniliselt kasutatakse Prologis listi summeerimiseks **akumulaatorit**, lisades predikaadile lisaargumendi, et väljastada selle abil tulemus. Akumulaatori mõte on asendada osalise vastuse leidmine täieliku vastuse leidmisega. Selleks arvutame igal rekursiooni sammul muutuja *Acc* väärtusena listi vaadeldud elementide summa, andes akumulaatorile algväärtuseks nulli:

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

Summeerides listi [1, 2, 3] päringuga `?-sum([1,2,3],Sum)`, saame järgmised alampäringud:

```
?-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
```

Kuid listi elemendid saab kokku liita ka ilma akumulaatorita, tühjendades esmalt rekursiooniga listi ja liites seejärel elemendid rekursioonist väljumise käigus tagurpidi järjekorras:

```
sum([],0).
sum([X|Xs],Sum):-
  sum(Xs,Sum2),
  Sum is Sum2+X.
```

Päringu `?-sum([1,2,3],Sum)` täitmisel saame nüüd järgmised alampäringud ja vastuse:

```
?-sum([2,3],Sum).
?-sum([3],Sum).
?-sum([],0).
?-sum([3],3).
?-sum([2,3],5).
?-sum([1,2,3],6).
Sum=6
```

Viimane programm raiskab küll rohkem mälu kui eelmised, sest jätab arvutamata summad päringute magasini ootele.

Ülesanne 49. Realiseerige listi ümberkeeramine kahe programmina: esimene kasutab predikaati *append* ja teine akumulaatorit. Otsustage, kumb variant töötab efektiivsemalt. Näiteks

```
?-reverse([a,b,c],[c,b,a]).
yes
```

Ülesanne 50. Kontrollige, kas list on sama, lugedes selle elemente eest tahapoole ja tagant ettepoole. Näiteks

```
?-palindroom([s,a,m,m,a,s]).
yes
```

Ülesanne 51. Kirjutage kaks programmi, mis moodustavad antud listi põhjal korduvate elementideta listi: esimene programm jätab alles iga elemendi viimase esinemise listis ja teine programm jätab alles iga elemendi esimese esinemise listis, säilitades ülejäänud elementide järjestust. Näiteks

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

Ülesanne 52. Kirjutage programm, mis koostab antud listi põhjal uue listi, milles iga element on paaris oma kordsusega esialgses listis. Säilitage elementide loomulikku järjestust. Näiteks

```
?-ntimes([a,b,c,a],List).
List=[(a,2),(b,1),(c,1)]
```

4.4. Järjestamine

Hoopis huvitavam on listi elementide järjestamine, näiteks mittekahanevas järjekorras: [1, 2, 2, 3]. Tundub, et Prolog on ainuke keel, kus omab mõtet sorteerimismeetod *slowsort*, milles listi järjestamiseks leitakse selle kõikvõimalikud permutatsioonid ja kontrollitakse neis elementide järjestatust (vt. ül. 54):

```
% slowsort(Xs,Ys) on tõene siis,
%       kui Ys on listi Xs järjestatud kuju
slowsort(List,SortedList):-
  integer_list(List),
  permutation(List,SortedList),
  ordered(SortedList).
```

Näiteks

```
?-slowsort([1,3,2,4,2],S).
S=[1,2,2,3,4]
```

Et järjestatakse arvude liste, siis anname kõigepealt vastava tüübikirjelduse:

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

Predikaat *integer*/1 on täisarvu kontrolliv süsteemne predikaat.

Permutatsioonide genereerimiseks valime listist eespool kirjeldatud predikaadiga *select* järgemööda ühe elemendi ning asetame selle permutatsioonilistis esimeseks elemendiks:

```
% permutation(Xs,Ys) on tõene siis,
%       kui Ys on listi Xs permutatsioon
permutation([],[]).
permutation(List,[X|Xs]):-
  select(X,List,List2),
  permutation(List2,Xs).
```

Ülesanne 53. Mis järjekorras väljastatakse päringu tulemused?

```
?-permutation([a,b,c],Xs).
```

Ülesanne 54. Realiseerige predikaat *ordered*, mis kontrollib listi elementide mittekahanevat järjestatust. Näiteks

```
?-ordered([1,2,3]).
yes
?-ordered([1,3,2,4,2]).
no
```

Ülesanne 55. Testige predikaadi *slowsort* kiirust erinevatel listidel. Kui suurtel listidel annab *slowsort* tulemuse? Kuidas muuta programmi tööd kiiremaks?

Ülesanne 56. Kirjutage programm, mis järjestab listi mullimeetodil.

Ülesanne 57. Kirjutage programm, mis järjestab listi pistemeetodil.

Ülesanne 58. Kirjutage programm, mis järjestab aatomite listi tähestikuliselt. Näiteks

```
?-alphsort([tiina,mari,külli],Xs).
Xs = [külli,mari,tiina]
```

4.5. Diferentslist

Diferentslisti käsitletakse tavaliselt termina kujul $x - y$, s.t. listide x ja y vahena, kus y langeb kokku listi x lõpuga. Juhul, kui diferentslisti tagumine liige on väärtustamata muutuja, saame vastava hariliku listi kõige üldisema esituse diferentslistina, mis sarnaneb osalise listiga. Näiteks listi [1, 2, 3] kõige üldisem esitus diferentslistina on $[1, 2, 3|Xs] - Xs$. On selge, et iga listi Xs saab esitada diferentslistina kujul $Xs - [\,]$.

Diferentslistide tähtsaks omaduseks on nende töötlemise efektiivsus. Näiteks predikaat *concat* ühendab diferentslistid $[1, 2, 3|Xs] - Xs$ ja $[4, 5|Ys] - Ys$ ühe resolutsioonisammuga, saades tulemuseks listi $[1, 2, 3, 4, 5|Ys] - Ys$:

```
concat(Xs-Ys,Ys-Zs,Xs-Zs).
```

Samas harilike listide [1, 2, 3] ja [4, 5] ühendamiseks predikaadiga *append* kulub neli resolutsioonisammu, sest enne ühendamist eemaldatakse ühekaupa kõik esimese listi elemendid.

Diferentsliste võib vaadelda kui vasakult paremale töötavat konveierit. mida on hea kasutada teksti grammatiliseks analüüsimiseks. Analüüsime näiteks eestikeelseid arvuväljendeid (vt. ül. 16 ja 19). Esitame arvuväljendid listi kujul, näiteks fraasile “sada viiskümmend kuus” vastab list [*sada*, *viiskümmend*, *kuus*]. Süntaktiliseks analüüsimiseks võib jagada esialgse väljendi alamväljenditeks, mida analüüsitakse eraldi ja ühendatakse seejärel predikaadiga *append*. Näiteks sajalistefraas algab sajalisi väljendava sõnaga (sada, kakssada jne.), millele järgneb kümnelistefraas:

```
sajalistefraas(Xs):-
    sajalised(Ys), kümnelistefraas(Zs), append(Ys,Zs,Xs).
```

Kümnelistefraas algab kümnelisi väljendava sõnaga (kümme, kakskümmend jne.), millele järgneb ühelistefraas:

```
kümnelistefraas(Xs):-
    kümnelised(Ys), ühelistefraas(Zs), append(Ys,Zs,Xs).
```

Ühelistefraas koosneb ühelisi väljendavast sõnast (üks, kaks jne.):

```
ühelistefraas(Xs):-ühelised(Xs).
```

Sajalised, kümnelised ja ühelised esitame üheelemendiliste listidena. Näiteks

```
sajalised([sada]).
kümnelised([viiskümmend]).
ühelised([kuus]).
```

Seejuures võivad kümnelised ja ühelised ka puududa:

```
kümnelised([]).
ühelised([]).
```

Analüsaatori käivitame päringuga

```
?-sajalistefraas([sada,viiskümmend,kuus]).
yes
```

Sama analüüsi saab realiseerida efektiivsemalt, kasutades diferentsliste, mis on erinevalt predikaadist *append* ühendatavad ühe sammuga:

```
sajalistefraas(Xs-Ys):-
    sajalised(Xs-Zs), kümnelistefraas(Zs-Ys).
kümnelistefraas(Xs-Ys):-
    kümnelised(Xs-Zs), ühelistefraas(Zs-Ys).
ühelistefraas(Xs-Ys):-ühelised(Xs-Ys).
```

Väljendites esinevad konkreetsed sõnad esitame seekord listide vahedena:

```
sajalised([sada|Xs]-Xs).
kümnelised([viiskümmend|Xs]-Xs).
kümnelised(Xs-Xs).
ühelised([kuus|Xs]-Xs).
ühelised(Xs-Xs).
```

Programmi käivitame päringuga

```
?-sajalistefraas([sada,viiskümmend,kuus]-[]).
```

Et miinusmärki sisaldava termi unifitseerimine tähendab sageli ühte lisaoperatsiooni,[17] siis asendatakse miinusmärk efektiivsuse huvides komaga, suurendades niiviisi predikaatide argumentide arvu — diferentslist muutub ühest termist kaheks komaga eraldatud termiks:

```
sajalistefraas(Xs,Ys):-
    sajalised(Xs,Zs), kümnelistefraas(Zs,Ys).
kümnelistefraas(Xs,Ys):-
    kümnelised(Xs,Zs), ühelistefraas(Zs,Ys).
ühelistefraas(Xs,Ys):-ühelised(Xs,Ys).
```

Arvsõnad esitame samuti “koma”-listidena:

```
sajalised([sada|Xs],Xs).
kümnelised([viiskümmend|Xs],Xs).
kümnelised(Xs,Xs).
ühelised([kuus|Xs],Xs).
ühelised(Xs,Xs).
```

Programmi käivitame päringuga

```
?-sajalistefraas([sada,viiskümmend,kuus],[]).
```

List ei tarvitse põhimõtteliselt lõppeda tühilistiga, näiteks võib ühelistefraasile järgneda veel sõnu nagu tekstilõigus “sada viiskümmend kuus õuna”. Sellisel juhul saame vastuseks *Xs* analüüsimata sõnade listi:

```
?-sajalistefraas([sada,viiskümmend,kuus,õuna],Xs).
Xs=[õuna]
```

Ülesanne 59. Koostage predikaat, mis annab diferentslisti tüübikirjelduse.

Ülesanne 60. Lisage arvuväljendite grammatikale arvude 11—19 analüüs.

Ülesanne 61. Koostage diferentslistide abil programmeerimiskäsu `if x > 0 then a:=a+2 else a:=1` analüsaator.

[17] Meie unifitseerimisalgoritmi korral jääb sammude arv samaks.

5. peatükk

Keerdülesannete lahendamine

Vastuste genereerimine ja testimine. Eituse kasutamine ja sobivate andmestruktuuride valimine.

5.1. Mis nädalapäev täna on?

Prolog sobib hästi klassikaliste intellektitehnika probleemide, näiteks keerdülesannete lahendamiseks. Võtame hakatuseks ülesande, milles Alice püüab unustuse metsas välja selgitada, mis päev parasjagu on: *Alice kohtas metsas Lõvi ja Ükssarve. On teada, et Lõvi valetab esmaspäeval, teisipäeval ja kolmapäeval ning räägib tõtt ülejäänud nädalapäevadel. Ükssarv valetab neljapäeval, reedel ja laupäeval ning räägib tõtt ülejäänud päevadel. Alice päris loomadelt, kas nad eile rääkisid tõtt või valetasid. Mõlemad loomad väitsid, et olid eile valetanud. Alice suutis vastuste põhjal otsustada, mis nädalapäevaga on tegemist. Mis nädalapäeval Alice neid küsitles?*

Ülesande lahendamiseks formuleerime kõigepealt faktid selle kohta, kes millal valetab. On teada, et Lõvi valetab esmaspäeval, teisipäeval ja kolmapäeval, seega saame kolm fakti:

```
lõvivaletab(esmaspäev).
lõvivaletab(teisipäev).
lõvivaletab(kolmapäev).
```

Ükssarve kohta on teada, et ta valetab neljapäeval, reedel ja laupäeval:

```
ükssarvvaletab(neljapäev).
ükssarvvaletab(reede).
ükssarvvaletab(laupäev).
```

Ülejäänud päevadel räägivad mõlemad loomad tõtt. Selle saab defineerida teadaolevate faktide põhjal, kasutades eitust (vt. lk. 11):

```
lõvitõtt(X):- \+lõvivaletab(X).
ükssarvtõtt(X):- \+ükssarvvaletab(X).
```

Programmile tuleb lisaks selgitada, mida tähendab mõiste *eile*, s.t. esmaspäevale eelneb pühapäev, teisipäevale esmaspäev jne.:

```
eile(esmaspäev,pühapäev).
eile(teisipäev,esmaspäev).
...
eile(pühapäev,laupäev).
```

Formaliseerime nüüd Lõvi väite. Kui oletada, et Lõvi rääkis tõtt, siis eelmisel päeval ta valetas:

```
lõvi(X):-eile(X,Y), lõvitõtt(X), lõvivaletab(Y).
```

Ja kui oletada, et Lõvi valetas, siis eelmisel päeval rääkis ta tõtt:

```
lõvi(X):-eile(X,Y), lõvivaletab(X), lõvitõtt(Y).
```

Paneme tähele, et predikaat *eile* töötab siin nädalapäevade generaatorina temale järgnevatele pärigutele, mis kontrollivad vajalike tingimuste täidetust. Vabade muutujate puudumine tagab ka eituse kasutamise korrektsuse predikaadis *lõvitõtt*. Ükssarve väitele saame analoogilise kirjelduse:

```
ükssarv(X):-eile(X,Y),ükssarvtõtt(X),ükssarvvaletab(Y).
ükssarv(X):-eile(X,Y),ükssarvvaletab(X),ükssarvtõtt(Y).
```

Käivitades predikaadi *lõvi*, saame võimalikeks nädalapäevadeks neljapäeva ja esmaspäeva:

```
?-lõvi(X).
X = neljapäev ;
X = esmaspäev
```

Ükssarve korral on võimalikud pühapäev ja neljapäev:

```
?-ükssarv(X).
X = pühapäev ;
X = neljapäev
```

Ülesande lahenduse saamiseks moodustame nende tulemuste ühisosa, realiseerides selle reeglina *lahenda*/1:

```
lahenda(X):-lõvi(X), ükssarv(X).
?-lahenda(X).
X = neljapäev
```

Ülesanne 62. (J. Henno) Millisel päeval (päevadel) oleks Alice saanud küsimusele "Kas Sa räägid homme tõtt?" mõlematelt kinnituse, et nad räägivad homme tõtt?

Ülesanne 63. (R. Smullyan) Teisel päeval kohtas Alice ainult Lõvi, kes väitis kahte asja: (1) Ma valetasin eile. (2) Ma valetan jälle kaks päeva pärast homset. Mis nädalapäev see oli?

Ülesanne 64. (R. Smullyan) Millisel nädalapäeval saab Lõvi väita kahte asja: (1) Ma valetasin eile. (2) Ma valetan homme jälle.

Ülesanne 65. (R. Smullyan) Millistel nädalapäevadel saab Lõvi väita kahte asja ühe lausega: "Ma valetasin eile ja ma valetan homme jälle." (Hoiatus: vastus pole sama, mis eelmisel ülesandel!)

5.2. Ahv ja Banaan

Võtame teiseks ülesandeks probleemi, kus Ahv püüab kätte saada Banaani: *Ahv on toas, mille keskel on lakke riputatud Banaan. Ahv asub vasakul ukse juures ja paremal akna juures on suur Kast. Ahv ei ulatu põrandalt Banaanini; küll on võimalik selleni ulatuda Kasti pealt. Kuidas Ahv saab Banaani kätte?*

Probleemi lahendamiseks võtame kasutusele seisukirjelduste ruumi, mis koosneb kõikvõimalikest konstruktsioonidest kujul

$$\mathit{seis}(\mathit{AhvHor}, \mathit{AhvVer}, \mathit{Kast}, \mathit{OmabBanaani}),$$

mis määravad ära Ahvi ja Kasti parameetrid ruumis. Ahv saab olla horisontaalselt kas ukse juures, toa keskel või akna juures (vastavalt *uks*, *keskel*, *aken*). Sama kehtib ka Kasti kohta. Ahv saab vertikaalselt olla kas põrandal või Kasti peal (vastavalt *põrand*, *kastil*). Lisaks saab Ahv olla kas Banaaniga või ilma (vastavalt *omab*, *eioma*). Niisiis on võimalikke seise kokku $3^2 \cdot 2^2 = 36$. Ülesande algseisus on Ahv ukse juures põrandal ilma Banaanita ning Kast on akna juures, seega:

$$\mathit{seis}(\mathit{uks}, \mathit{põrand}, \mathit{aken}, \mathit{eioma}).$$

Võimalikke lõppseise on kokku 18, s.t. kõik seisud, milles Ahv omab Banaani:

seis(_, _, _, *omab*).

Ahvi võimalikud tegevused on Banaani võtmine laest, Kastile ronimine, Kasti lükkamine ühest kohast teise ja ruumis liikumine. Kirjeldame need tegevused kolmekohalise predikaadiga *käik*/3, mis näitab, kuidas jooksev seis muutub tegevuse tulemusena. Et Banaan ripub toa keskel laes, siis saab Ahv seda võtta vaid juhul, kui tema ja Kast asuvad mõlemad toa keskel ja Ahv on roninud Kasti otsa:

```
käik(seis(keskel,kastil,keskel,eioma),
     võta,
     seis(keskel,kastil,keskel,omab)).
```

Predikaadi *käik* esimeseks argumendiks on seis enne tegevuse sooritamist, teiseks argumendiks tegevuse nimi ja kolmandaks argumendiks seis pärast tegevuse sooritamist.

Kastile ronimise tingimuseks on, et Ahv ja Kast asuksid samas kohas ja Ahv asuks põrandal:

```
käik(seis(Koht,põrand,Koht,Omab),
     roni,
     seis(Koht,kastil,Koht,Omab)).
```

Selliseid seise on kokku $3 \cdot 2 = 6$. Muutuja *Koht* esinemine mõlemas seisus ütleb meile, et nii Ahv kui ka Kast jäävad pärast Ahvi Kastile ronimist samasse kohta, kus nad olid enne Kastile ronimist. Muutuja *Omab* ütleb, et Ahv saab Kastile ronida nii Banaaniga kui ka ilma Banaanita.

Lükkamise eeltingimuseks on, et Ahv ja Kast asuksid enne ja pärast lükkamist samas kohas:

```
käik(seis(Koht1,põrand,Koht1,Omab),
     lükka(Koht1,Koht2),
     seis(Koht2,põrand,Koht2,Omab)).
```

Selliseid seise on $3 \cdot 3 \cdot 2 = 18$.

Viimaseks tegevuseks on liikumine ruumis:

```
käik(seis(Koht1,põrand,Kast,Omab),
     liigu(Koht1,Koht2),
     seis(Koht2,põrand,Kast,Omab)).
```

Selleks on $3 \cdot 3 \cdot 3 \cdot 2 = 54$ võimalust, sõltuvalt liikumise algpunktist *Koht1*, lõpp-punktist *Koht2*, Kasti asukohast *Kast* ja Banaani omamisest *Omab*.

Järgnevalt konstrueerime käikude mootori *saab_kätte*/1, mis teeb käike senikaua, kuni jõutakse lõppseisu:

```
saab_kätte(seis(_,_,_,omab)).
saab_kätte(Seis1):-
   käik(Seis1,Käik,Seis2),
   write(Käik), nl,
   saab_kätte(Seis2).
```

Programmi käivitame, võttes argumendiks algseisu:

```
?-saab_kätte(seis(uks,põrand,aken,eioma)).
liigu(uks, _G461)
roni
lükka(aken, _G469)
roni
võta
```

Predikaat *write* väljastab täidetavate tegevuste nimed: kõigepealt liigub Ahv ukse juurest mingisse kohta *_G461*[18] ja ronib seal Kasti peale. Et Kast asub akna juures, siis ei saa Ahv Banaani kätte ning jääb Kasti peal patiseisu, sest ta ei saa täita ühtegi tegevust. Nüüd tühistatakse viimane tegevus, s.t. **tagurdatakse** programmi täitmine seisu, kus Ahv oli enne Kastile ronimist ja Ahv lükkab ronimise asemel Kasti akna juurest mingisse kohta *_G469*. Seejärel ronib Ahv Kastile ja võtab Banaani. Paneme tähele, et Ahv liikus akna juurde ja lükkas Kasti ruumi keskele tänu muutujate asendustele unifitseerimise käigus.

Ülesanne 66. Lahendage sama ülesanne teistsuguses seades: Ahv soovib saada kätte puu otsas rippuvat Banaani. Tal on kasutada laud, tool ja kepp, kusjuures ainuke võimalus Banaan kätte saada on ronida laua otsa tõstetud toolile ja lüüa Banaan kepiga puu otsast alla.

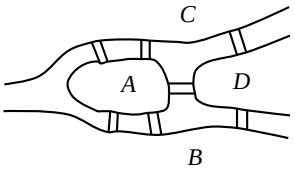
Ülesanne 67. Lahendage keerdülesanne: neli viikingit peavad liikuma ohtlikust tsoonist ohutusse tsooni. On pime ja kasutada saab ainult ühte tõrvikut. Kahe tsooni vahel on takistusriba, mida saab ületada kas ühe- või kahekaupa. Esimesel viikingil kulub takistusriba ületamiseks 5 minutit, teisel 10 minutit, kolmandal 20 minutit ja neljandal 25 minutit. Ohutusse tsooni liikumiseks on aega kuni 60 minutit. Kahe liikuja korral võetakse üleminekuaeg aeglasema paarilise järgi. Programm käivitatakse päringuga

```
?-saab_minna(seis(ohtlik,ohtlik,ohtlik,ohtlik,ohtlik,0)).
```

[18] Et saaks ronida, peab see koht olema aken.

5.3. Jalutuskäik Königsbergis

Lahendame nüüd Königsbergi sildade kuulsa probleemi: *ülesandeks on koostada pühapäevane jalutuskäik, mille käigus on vaja ületada seitse üle Pregeli jõe viivat silda, igaühte täpselt üks kord, nii et lõpuks jõuda koju tagasi.* Leonhard Euler (1707–1783) tõestas, et sellist ringkäiku ei leidu.



Probleemi lahendamiseks genereerime listi [1, 2, . . . , 7] kõikvõimalikud arvuga 1 lõppevad permutatsioonid, mis määravad ära seitsme silla ületamise järjekorra. Tee sulgemiseks lisame listi algusesse arvu 1 ja kontrollime, kas leitud tee pikkusega kaheksa on läbitav, s.t. kas igal kahel järjestikusel sillal on ühine ots:

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

Alustame sildade märgendamisest numbritega:

```
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 68. Parandage programmi nii, et tulemuseks on õige vastus — sellist jalutuskäiku ei leidu.

Ülesanne 69. (Neljavärviprobleem) Värvige etteantud maakaart nelja värviga nii, et naaberriigid on erinevat värvi.

Ülesanne 70. Kandke $n \times n$ malelauale n lippu nii, et ükski lipp ei asuks teise lipu tules.

Ülesanne 71. Kirjutage programm, mis näitab, kuidas mõõta 7- ja 5-liitrise anuma abil 4 liitrit vedelikku.

5.4. Kolm sõpra

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

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

Vaja on vastata järgmistele küsimustele:

- *Kes on austraallane?*
- *Millist sporti teeb Richard?*

Tegemist on tavalise nuputamisülesandega, millel on täpselt üks lahend. Ülesande lahendamisel on tähtis valida sobiv struktuur, mis lihtsustab oluliselt ülesande lahendamist. Seejärel omistame struktuuri muutujatele võimalikke väärtusi, tehes seda seni, kuni leiame väärtustuse, mis on kooskõlas nii faktidega kui ka küsimustega. Seda võib tinglikult nimetada struktuuri ühestamiseks. Struktuuriks on list kolmest elemendist, milleks on termid *sõber*/3:

```
struktuur([sõber(_,_,_), sõber(_,_,_), sõber(_,_,_)]).
```

Termi *sõber*/3 esimene argument määrab sõbra nime, teine kodakondsuse ja kolmas spordiala:

```
nimi(sõber(A,_,_),A).
kodakondsus(sõber(_,B,_),B).
sport(sõber(_,_,C),C).
```

Listi elemendid on järjestatud paremuse järgi, mispärast selle esimene element oli parem teisest ja kolmandast liikmest ja teine liige oli parem kolmandast:

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

Predikaat *first*/1 võtab listist välja vasakult esimese elemendi:

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

Sõprade probleemi lahendamiseks valime nüüd süsteemse predikaadiga *member*/2 struktuurist ühekaupa elemente ning ühestame faktide ja küsimuste abil struktuuri muutujate väärtused:

```
mõistatus:-
  struktuur(Sõbrad),
  oli_parem(S11,S12,Sõbrad),
      nimi(S11,michael),
      sport(S11,korvpall),
      kodakondsus(S12,ameerika),
  oli_parem(S21,S22,Sõbrad),
      nimi(S21,simon),
      kodakondsus(S21,iisraeli),
      sport(S22,tennis),
  first(Sõbrad,S3),
      sport(S3,kriket),
  member(S4,Sõbrad),
      nimi(S4,Nimi),
      kodakondsus(S4,austraalia),
  member(S5,Sõbrad),
      nimi(S5,richard),
      sport(S5,Sport),
  write('Austraallane on '), write(Nimi), nl,
  write('Richard mängib '), write(Sport), nl.
```

Programmi käivitamisel on tulemus järgmine:

```
?-mõistatus.
Austraallane on michael
Richard mängib tennis
```

Ülesanne 72. Lahendage keerdülesanne: kaks sõpra elavad eri linnades: üks elab Tartus ja teine elab Pärnus. On teada, et Tõnu elab Tartus. Kus elab Priit?

Lahendage sama ülesanne kolme sõbra ja kolme linna korral.

Ülesanne 73. Lahendage keerdülesanne[19]: ühel ja samal tänaval on viis maja, mis on erinevat värvi, iga majaomanik on eri rahvusest, joob erinevat jooki, suitsetab erinevat marki sigarette ja omab erinevat lemmiklooma. Küsimus: kellel on lemmikloomaks kalad? Faktid on järgmised:[20]

a) Britt elab punases majas.
b) Rootslasel on lemmikloomaks koer.
c) Taanlane joob teed.
d) Roheline maja asub valgest majast vasakul.
e) Rohelise maja omanik joob kohvi.
f) Inimene, kes suitsetab Pall Malli, kasvatab linde.
g) Kollase maja omanik suitsetab Dunhilli.
h) Inimene, kes elab keskmises majas, joob piima.
i) Norrakas elab esimeses majas.
j) See, kes suitsetab Blendi, on selle naaber, kelle lemmikloomaks on kass.
k) See, kellel on hobune, on selle naaber, kes suitsetab Dunhilli.
l) Inimene, kes suitsetab Bluemastersi, joob õlut.
m) Sakslane suitsetab Prince'i.
n) Norrakas elab sinise maja kõrval.
o) Inimesel, kes suitsetab Blendi, on naabriks see, kes joob vett.

Ülesanne 74. Lahendage keerdülesanne: Titanicu uppumise järel triivisid neli noormeest ja neli neidu üksikule saarele. Mõne aja möödudes armus neist igaüks ühte oma kaaslasesse, kusjuures kedagi ei armastanud mitu kaaslast.

Neiu, keda armastas John, armus Jimi. Neiu, keda armastas Allan, armus Billi. Noormees, keda armastas Rose, armus Marysse. Gloria armastas noormeest, kes armastas Rose'i. Gloriale ei meeldinud Jim ega John. Billile ei meeldinud Mary. Jillile ei meeldinud noormees, kes teda armastas. Keda armastas Allan?

Ülesanne 75. Lahendage keerdülesanne: neli endist koolivenda püüdsid meenutada, millistel nädalapäevadel olid neil koolis toimunud ringi koosolekud. Peeter arvas, et esmaspäeviti ja reedeti; Madis väitis, et esmaspäeviti ja neljapäeviti; Juhan oli veendunud, et teisipäeviti ja neljapäeviti; Elmar aga kinnitas, et kolmapäeviti ja reedeti. Hiljem selgus, et Peeter ja kaks tema koolivendadest eksisid ühe, neljas aga mõlema nädalapäevaga. Millisel nädalapäeval ringi koosolekuid kindlasti ei toimunud?

Ülesanne 76. (Ramsey arvud) Olgu meil kuueliikmeline seltskond, milles suvalised kaks inimest kas tunnevad teineteist või ei tunne. Kirjutage programm, mis tõestab, et antud seltskonnas on kas 3 inimest, kes tunnevad üksteist või 3 inimest, kellest ükski teisi ei tunne.

Kirjutage teine programm, mis tõestab, et viie inimese korral see väide ei kehti.

---

[19] Selle mõistatuse olevat sõnastanud Albert Einstein oma karjääri alguses ning väitnud, et 98% inimkonnast ei oska seda puhast loogikaülesannet lahendada.

[20] Ülesande teksti saatis ja soovitas loengutes kasutada Antti Andreimann.

6. peatükk

Metapredikaadid

Lõikepredikaat ja eitus. Programmide teisendamine ja metainterpreteerimine. Dünaamilised ja teist järku predikaadid.

6.1. Lõikepredikaat

Lõikepredikaat (ingl. k. *cut*, tähiseks !) elimineerib tagurdamisest (ingl. k. *backtracking*) kõik talle vastavas reeglis või päringus eelnevad predikaadid. Vaatleme programmi

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

Lõikepredikaadi täitmine predikaadi *b* esimeses kirjelduses seisneb temast vasakule jäävate predikaatide *b* ja *d* väärtuste fikseerimises, mistõttu need eemaldatakse tagurdamisest. Lõikepredikaadi töö jälgimiseks esitame päringu `?-a` täitmisel saadud alampäringud:

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

Vastus *no* saadi seetõttu, et lõikepredikaat lõpetab reeglis *b:-d,!* temale eelnevate predikaatide *d* ja *b* täitmise. Kõrvaldame nüüd sellest reeglist lõikepredikaadi. Selgub, et sel juhul leitakse veel kaks vastust, sest päringu `?-a` täitmisel jõutakse predikaatide *d* ja *b* ülejäänud kirjeldusteni:

```
?-b.
?-d.
yes ;
?-true.
yes ;
yes ;
no
```

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

Siit peaks selguma lõikepredikaadi olemus. Deklaratiivses mõttes võib seda vaadelda kui predikaati, mis on alati tõene. Protseduraalses mõttes muudab ta aga otsingupuud, lõigates sealt välja kirjelduses talle eelnevate predikaatide harud. Sõltuvalt programmi tähenduse muutumisest liigitatakse lõikepredikaadi kasutused rohelisteks (ingl. k. *green*) ja punasteks (ingl. k. *red*). Lõikepredikaadi **punased** kasutused on ohtlikud, sest nad muudavad programmi semantikat, lõigates välja osa vastuseid. Lõikepredikaadi **rohelised** kasutused on aga ohutud. Neid lisatakse predikaatide determineerimiseks, millega tõuseb nende töökiirus. Kui me teame, et päringul on ainult üks vastus, siis saab selleni jõudmisel lõikepredikaadiga vaadeldava predikaadi kõik järgnevad kirjeldused tagurdamisest välja jätta. Sellega vähendame kontrollide arvu ning hoiame kokku aega ja mälu. Olgu meil vaja sooritada kolme erinevat 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 tarvis kontrollida. Rohelise lõikepredikaadi võib ära jätta või asendada selle alati tõese predikaadiga *true*.

Punase lõikepredikaadiga on asi keerulisem. Näiteks predikaat *max* ütleb: kui $X \leqslant Y$, siis maksimaalne X-ist ja Y-ist on Y, muidu aga X.

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

Eemaldades predikaadist *max* lõikepredikaadi, saame vigase programmi. See programm töötab ainult juhul, kui kolmandaks argumendiks on muutuja, s.t. kui ta arvutab maksimaalse kahe esimese argumendi väärtustest. Juhul kui kõik argumendid on arvulised, saame unifitseerimise ebaõnnestumise tõttu reegli esimese kirjelduse päises vale vastuse:

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

Lõikepredikaadi punase kasutuse saame muuta roheliseks sel teel, et lisame predikaadi *max* teisele kirjeldusele tarviliku tingimuse:

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

Nüüd annab sama päring õige vastuse:

```
?-max(2,5,2).
no
```

Reegli päise argumentide unifitseerimise ebaõnnestumise vältimiseks võime kirjutada programmi ümber kujul, kus reegli päises on sõltumatud muutujad:

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

Ent selguse huvides oleks õigem kirjutada programm alguses valmis ilma lõikepredikaatideta ning lisada need efektiivsuse huvides alles hiljem. Ideaalkujul võiks predikaadi *max* teise kirjelduse esitada kujul

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

Vahel võib tarvis minna ka efektiivset *member*-predikaati:

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

See predikaat annab ainult päringu esimese vastuse, lõigates ülejäänud vastused välja. Näiteks

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

Et tegemist on standardpredikaadiga, siis on õigem lisada lõikepredikaat päringu lõppu:

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

Lõikepredikaadi 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.
```

Predikaat *if_then_else* tähendab deklaratiivselt, et tõesed on kas P ja Q või P eitus ja R. Protseduraalselt aga tõestame P ja kui tõestamine õnnestus, siis tõestame Q, muidu tõestame R. Predikaati *if_then_else* kirjutatakse tavaliselt kujul $P -> Q; R$.

Lõikepredikaati võib kasutada ka programmi teisendamiseks efektiivsele sabarekursiivsele kujule (ingl. k. *tail recursion optimization*). Kui on teada, et päringu $\leftarrow p_1, \ldots, p_n$ esimesed $n-1$ predikaati annavad ühese vastuse, siis võib rekursiivse predikaadi p_n eraldada neist lõikepredikaadiga, mistõttu rekursioonimagasini viimase taseme võib allapoole liikudes pidevalt üle kirjutada. Paremad Prologi-süsteemid teevad seda automaatselt. Sabarekursiivsed on näiteks standardpredikaadid *member* ja *append*, võimaldades kuitahes sügavat rekursiooni. Seda teades, saame kirjutada lõpmatult töötava predikaadi *loop*:

```
loop:-loop.
loop.
```

6.2. Eitus

Eitusega on loogilistes programmides isesugune suhe. Tegemist ei ole mitte eitusega klassikalise loogika mõttes, vaid selle intuitsionistliku analoogiga: väide on väär siis, kui seda ei õnnestu tõestada. Puhtas Prologis (ingl. k. *pure Prolog*) ei ole üldse eitust ja teisi loogikaväliseid vahendeid. Loogilised programmid pannakse kirja **Horni reeglitena** pööratud implikatsiooni kujul $p \leftarrow q_1, \ldots, q_n$, mille vasakule poole on viidud esialgse disjunkti ainus eituseta atomaarne predikaat ja paremale eitusega atomaarsed predikaadid ilma eituseta. Lubades eituseid (\+) ka parempoolsete predikaatide ees, väljume puhtast Prologist. Sama kehtib ka kvantorite kohta, mis esinevad Horni reeglites varjatud kujul: reegli vasaku poole muutujad seotakse üldisuskvantoriga ja ülejäänud muutujad olemasolukvantoriga.

Prologis esitatakse ainult positiivsed faktid, eeldades, et kirjeldamata faktid on väärad. Asi töötab siis, kui predikaatides ei ole muutujaid:

```
tudeng(andres).
tudeng(juhan).
tudeng(peeter).
lektor(mariliis).
```

Antud juhul saame teada, et Mariliis ei ole tudeng:

```
?-\+ tudeng(mariliis).
yes
```

Eitust on lihtne realiseerida lõikepredikaadi abil:[21]

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

Muutujat *X* kasutatakse siin **metamuutujana**, s.t. päringu kujul oleva muutujana, mida on lubatud käsuna täita. Muutujat saab ka täita süsteemse predikaadiga *call*/1. Kombinatsioon ! ja *fail* ütleb, et vastust ei tasu edasi otsida, sest see puudub.

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

```
vallaline_tudeng(X):-not abielus(X), tudeng(X).
tudeng(juhan).
abielus(andres).
```

Selgub, et andmebaasis pole vallalisi tudengeid:

```
?-vallaline_tudeng(X).
no
```

Päringu jälgimine predikaadi *not* kirjelduse alusel selgitab, kuidas eitust selles päringus käsitletakse:

```
?-not abielus(X), tudeng(X).
?-abielus(X), !, fail, tudeng(X).
?-!, fail, tudeng(andres).
?-fail, tudeng(andres).
no
```

Lõikepredikaat lõikab predikaadi *not* teise kirjelduse otsingupuust välja. Õige vastuse saame siis, kui kirjutame päringus predikaadid teises järjekorras:

```
?-tudeng(X), not abielus(X).
```

Seda päringut käsitletakse nii:

```
?-not abielus(juhan).
?-abielus(juhan), !, fail.
X = juhan
```

Lõikepredikaadini praegu ei jõuta ning täidetakse *not* teine kirjeldus.

Eitusele on iseloomulik, 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):- \+ \+ Goal.
```

[21] Eeldame, et *not* on defineeritud prefikstehtena.

Predikaati *verify* võib muuhulgas kasutada eelkontrollideks ristsõnade koostamisel. Oletame, et me soovime koostada 3×3 täielikku ristsõna, näiteks

a	h	i
r	a	t
k	e	s

Sõnad esitame tähtede loeteludena ja ruudustiku üheksa muutujana:

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

$C11$	$C12$	$C13$
$C21$	$C22$	$C23$
$C31$	$C32$	$C33$

Kandes sõnastiku esimese sõna "rat" ruudustiku esimesse ritta, kontrollime liigse töö vältimiseks enne järgmise rea täitmist, kas sõnaraamatus leiduvad sõnad veergude täitmiseks. Seejärel kanname sõna ruudustiku esimesse veergu:

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

Muutujad *C22*, *C23*, *C32* ja *C33* jäävad väärtusmata. Antud juhul sõna "rat" ei sobinudki esimesse ritta ning see asendati sõnaga "ahi". Sama sõna sobis ka esimesse veergu. Ülejäänud lahtrid jäid tühjaks, kuigi veerud kontrolliti esimese rea täitmise järel ära.

Ülesanne 77. Kirjutage programm, mis koostab 3×3 täieliku ristsõna.

Ülesanne 78. Kirjutage programm, mis koostab 8×8 aukudega ristsõna.

Ülesanne 79. Kirjutage programm, mis koostab ristsõna, täites ruudustiku lahtreid mitte sõnakaupa, vaid tähthaaval, täites lahtrid tähestiku erinevate tähtedega ning kontrollides nende sobivust sõnaraamatu sõnadega.

Testige, kumb ruudustiku täitmisviis on efektiivsem: kas sõnakaupa või tähekaupa täitmine.

6.3. Teist järku predikaadid

Predikaadid annavad tavaliselt päringu vastuseid ühekaupa. Näiteks

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

Et erinevad vastused saadakse tagurdamisega, siis jõudes järgmise vastuseni, kaob eelmine vastus ära. Vastuste säilitamise probleemi lahendavad süsteemsed teist järku predikaadid *bagof*, *setof* ja *findall*, mis võimaldavad koguda kokku otsingupuu eri harudes saadud vastuste hulga.

Predikaat *bagof* moodustab lihtsalt päringu vastuste listi:

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

Predikaat *setof* teeb sama, kuid lisaks veel järjestab vastuste listi, kõrvaldades sellest elementide kordused. Predikaat *findall* leiab päringu vastused esimeseks argumendiks oleva muutuja järgi, sidudes päringu teised muutujad olemasolukvantoriga '^'. Kui predikaadid *bagof* ja *setof* käituvad üldiselt loogiliselt, siis predikaadil *findall* puudub hea loogiline kirjeldus. Järgmises näites fikseerib predikaat *bagof* päringus vaba muutuja *Y* esimese väärtuse, andmata kõiki vastuseid 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 muutuja *Y* olemasolukvantoriga, saame soovitud tulemuse:

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

Predikaat *setof* annab päringu vastused järjestatult:

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

Predikaadi *findall* korral pole vahet, kas *Y* on seotud olemasolukvantoriga 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 predikaatide abil saab ka kirjeldada eitusepredikaadi *not*/1 ja predikaadi *var*/1, mis kontrollib, kas tema argumendiks on vaba muutuja või mitte. Näiteks eituse defineerimiseks paneme tähele, et vääral päringul vastused puuduvad:

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

Ülesanne 80. Realiseerige predikaat *var*/1, kasutades teist järku predikaate.

6.4. Metaprogrammeerimine

Metaprogrammeerimise all mõistetakse manipuleerimist programmidega kui andmetega. Metaloogiliste predikaatide hulka kuuluvad näiteks *clause*, *=..*, *functor* ja *arg*.

Prologi termid kujutavad endast sümbolitest moodustatud suluavaldisi. Termidega manipuleerimiseks on predikaadid *functor*/3 ja *arg*/3. Neist esimene annab termi $aatom(term_1, \ldots, term_n)$ korral sulgudele eelneva aatomi ja termi argumentide arvu ning teine leiab sulgude seest n-nda komadega eraldatud argumendi. Näiteks

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

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 argumendina nädalapäeva:

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

Selleks moodustame sama funktsiooniniga uue termi, võttes selle viimaseks argumendiks nädalapäeva ning kopeerides ülejäänud argumendid eelmisest termist:

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

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

Sama võib teha ka binaarse infikstehte =.. abil, mis teisendab vasakul asuva n argumendiga termi $(n+1)$-elemendiliseks listiks, mille esimeseks elemendiks on termi sulgude ees olev aatom ja sellele järgnevad termi argumentide väärtused, ja vastupidi, $(n+1)$-elemendilise listi n argumendiga 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 81. Realiseerige predikaat *map_list*, mis rakendab oma teiseks argumendiks olevat funktsiooni esimese argumendina näidatud listi kõikidele elementidele. Kirjeldage predikaadid *succ* ja *square*, mis leiavad vastavalt antud arvule järgneva arvu ja antud arvu ruudu. Näiteks, et tõese vastuse annaksid päringud

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

Prologis on ka vahendid programmide teisendamiseks. Mällu loetud programmi predikaatide kirjeldusi saab lugeda predikaadiga *clause*/2, mis väljastab reegli *Head:-Body* päise *Head* järgi selle sisu *Body*. Näiteks predikaadi *member* korral on tulemuseks

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

Et faktidel eeldused puuduvad, siis väljastatakse nende korral samaselt tõene predikaat *true*. Metamuutuja abil on lihtne koostada ka uusi päringuid, sest predikaat *functor* moodustab termi nime ja argumentide arvu järgi abstraktse termi, mida on võimalik käivitada:

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

Kui metamuutujas T on predikaadi nimi määramata, siis saame veateate:

```
?-T, functor(T,member,2).
ERROR: Arguments are not sufficiently instantiated
```

Prologi-süsteemides, mis toetavad päringute edasilükkamist (ingl. k. *delay*), jäetakse päring ootele, täites selle alles siis, kui see on piisavalt määratud. Näiteks ECLiPSe'is käivitatakse päringute edasilükkamine päringuga

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

Nii võib lahti saada ka aritmeetikavigadest väärtustamata muutujate korral:

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

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

6.5. Sugulaste märgendamine

Kasutame nüüd varem kirjutatud programmi ja teist järku predikaate oma sugulaste määramiseks vastava teadmiste baasi põhjal (vt. ül. 3 ja 5). Sugulasi on mõistlik uurida kauguse järgi: kõigepealt määrame ära oma esimese ringi sugulased, siis teise ringi sugulased jne. Esimese ringi sugulasteks loeme lapsed, abikaasa ja vanemad. Kõigepealt koostame programmi, mis väljastab etteantud arvu *N* ja isiku nime järgi tema *N*-nda ringi sugulased. Vastav otsing toimub rekursiivselt ringide kaupa:[22]

```
relative(0,X,X).
relative(N,X,Y):-
  N>0,
  ( son(X,Z)
  ; daughter(X,Z)
  ; mother(X,Z)
  ; father(X,Z)
  ; wife(X,Z)
  ; husband(X,Z)
  ),
  N1 is N-1,
  relative(N1,Z,Y).
```

[22] Sugulaste leidmise programm valmis loengusisese diskussiooni käigus koostöös Targo Tennisbergiga.

Ülesanne 82. Defineerige predikaadid *son*/2, *daughter*/2, *wife*/2 ja *husband*/2, mis leiavad sugulaste kohta moodustatud teadmiste baasist vastavalt isiku poja, tütre, naise ja mehe. Loetlege oma nullinda, esimese ja teise ringi sugulased.

Järgmiseks märgendame oma sugulased ära. Selleks lisame programmi predikaatidele sugulaste **märgendid**, mis moodustatakse miinusmärkidega ja öeldistäitega *on* ühendatud sõnade fraasidest. Lisaks tõlgendame kaugust N mitte täpse kaugusena, vaid vahemikuna nullist N-ni, lisades predikaadi *relative*/4 esimesele kirjeldusele tingimuse $N \geqslant 0$:

```
relative(N,X,X,on-X):-N>=0.
relative(N,X,Y,Def-Def1):-
  N>0,
  ( son(X,Z), Def=poeg
  ; daughter(X,Z), Def=tütar
  ; mother(X,Z), Def=ema
  ; father(X,Z), Def=isa
  ; wife(X,Z), Def=naine
  ; husband(X,Z), Def=mees
  ),
  N1 is N-1,
  relative(N1,Z,Y,Def1).
```

Ülesanne 83. Loetlege oma esimese kahe ringi sugulased koos vastavate märgenditega.

Selgub, et igal inimesel võib olla mitu märgendit. Näiteks Alice on tema ise, kuid samal ajal on ta ka oma ema tütar ja oma isa tütar:

```
?-relative(2,alice,alice,Def).
Def = on-alice ;
Def = ema-tütar-on-alice ;
Def = isa-tütar-on-alice ;
no
```

Ülesanne 84. Koostage programm, mis leiab iga sugulase jaoks unikaalse märgendi. Näiteks

```
?-relative-definite(2,alice,john,X).
X=alice-isa-on-john ;
no
```

6.6. Programmi metainterpreteerimine

Metainterpreteerimine on programmi täitmine ebastandardsete vahenditega, näiteks kasutades enda kirjutatud interpretaatorit. Prologis on süsteemsed vahendid programmide täitmise juhtimiseks, mis võimaldavad realiseerida lisaks programmide täitmise standardsele ülevalt-alla ja vasakult-paremale strateegiale ka alternatiivseid täitmisstrateegiaid.

Lihtsaima metainterpretaatori saame siis, kui kasutame metamuutujat, usaldades käsu täitmise standardsele interpretaatorile:

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

Puhta Prologi metainterpretaatori saame predikaati *clause* kasutades, mis lahutab Prologi reeglid vasakuks ja paremaks pooleks:

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

Seega liitpäringu täitmine toimub ühe päringu kaupa vasakult paremale ja reegli korral täidetakse selle päisele vastavad sisukirjeldused. Lõikepredikaadid '!' keelavad predikaadi *true* käsitlemise reeglina predikaadi *solve* kirjelduse kolmandas reas. Metainterpreteerime päringu `?-member(X,[a,b,c])` täitmist predikaadiga *solve*:

```
?-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
```

Predikaat *clause* leiab predikaadile vastava reegli päise ja asub seda täitma. Fakti korral on paremaks pooleks lihtsalt predikaat *true*. Liitpäringu A, B korral täidetakse aga selle kumbki päring eraldi.

Puhta Prologi metainterpretaatorit on lihtne laiendada kogu Prologile, kuhu on lisatud aritmeetika, listid, süsteemsed predikaadid jms. Selleks selgitame, mida teha süsteemsete predikaatide, eituse ja lõikepredikaadi korral:

```
solve(true):-!.
solve((A,B)):-!, solve(A), solve(B).
solve(!):-!, !(reduce(A)).          % Ancestor cut
solve(\+ A):-!, \+ solve(A).
solve(setof(X,Goal,Xs)):-!, setof(X,solve(Goal),Xs).
solve(A):-system(A), !, A.
solve(A):-reduce(A).

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)).
```

Lõikepredikaadi jaoks toome sisse predikaadi *reduce*, mis võimaldab otsingupuus õigesse kohta tagasi pöörduda:

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

Ülesanne 85. Prolog otsib standardselt vastust süviti, otsides vastust kõigepealt otsingupuu vasakpoolseimast harust, seejärel selle naaberharust jne. Millise tulemuse annab päring `?-loop`? Joonistage vastav otsingupuu.

```
loop:-loop.
loop.
```

Ülesanne 86. Predikaat *loop* annab vastuse laiuti otsides, s.t. vaadates otsingupuu harusid läbi paralleelselt. Realiseerige laiuti otsimise strateegia. Võrrelge süviti- ja laiutiotsingu strateegiate efektiivsust.

Ülesanne 87. Oletame, et te konstrueerite kompilaatorit üheregistrilisele protsessorile, mis oskab täita järgmisi käske:

- ONE – omistada registrile väärtus 1
- DOUBLE – kahekordistada registri sisu
- ADD – liita registri sisule 1
- SUB – lahutada registri sisust 1
- DIVIDE – jagada registri sisu 3-ga

Esimest nelja käsku saab täita ainult juhul, kui registri sisu ei jagu kolmega. Kui registri sisu jagub kolmega, siis saab täita ainult käsku DIVIDE. Ülesandeks on kirjutada registrisse etteantud arv. (Võite eeldada, et registri algväärtuseks on 1.)

Kirjutage programm, mis simuleerib sellise protsessori tööd arvu 27 leidmisel. Koostage selle programmi jaoks metainterpretaator, mis kasutab järgmisi otsingustrateegiaid:

a) laiutiotsing;
b) iteratiivne süvitiotsing;
c) A*otsing.

Valige A*otsingus hinnangufunktsiooniks $f(n) = g(n) + h(n)$, kus $g(n)$ on läbitud tee maksumus, kui DIVIDE maksumus on $2n/3$ ja teiste käskude maksumus on 1 ning $h(n) = |27 - n|$ on heuristiline funktsioon.

Väljastage ka tee vastuseni. (Võimalusel vältige tsükleid.) Otsustage, milline neist strateegiatest on antud probleemi jaoks kõige efektiivsem.

6.7. Dünaamilised andmebaasid

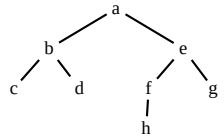
Kuigi programmi muutmist selle täitmise käigus loetakse halvaks tehnikaks, pakub predikaatide *assert*, *retract* ja *retractall* kasutamine huvitavaid võimalusi. Predikaat *assert* lisab predikaatide dünaamilise andmebaasi lõppu uusi fakte ja reegleid. Predikaat *retract* loeb ja kustutab sealt kirjeldusi ühekaupa. Predikaat *retractall* kustutab dünaamilisest andmebaasist kõik antud predikaadi kirjeldused.

Nende predikaatidega saame oma käsutusse globaalsed muutujad — loendajad ja summaatorid — ning mugava vahendi programmi täitmise juhtimiseks. Leiame näiteks teist järku predikaate *bagof*, *setof* ja *findall* kasutamata kahendpuu lehtede arvu. Esmalt kirjeldame, kuidas me puud läbime. Teeme seda näiteks rekursiivselt postorderis: kõigepealt läbime alampuu vasakpoolse haru, siis parempoolse haru ja seejärel tegeleme alampuu juurega:

```
traverse(tree(_,Left,_)):-traverse(Left).
traverse(tree(_,_,Right)):-traverse(Right).
traverse(tree(_,leaf,leaf)).
```

Kirjeldame 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)))).
```



Dünaamilisi predikaate saab kirjeldada käsuga *dynamic*. Võtame kasutusele dünaamilise loenduri *lehtede_arv*/1:

```
:-dynamic lehtede_arv/1.
```

Loenduri *lehtede_arv*/1 abil loeme *fail*-tsükliga kokku, kui mitu korda jõuti puu läbimisel leheni, suurendades iga kord leheni jõudes loenduri väärtust ühe võrra:

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

Lehtede arvu väljastame disjunktsiooniga pärast kahendpuu kõigi harude läbimist. Eespool kirjeldatud puu lehtede arvu leidmiseks loeme selle sisse ja leiame puu lehtede arvu:

```
?-tree(Tree), number_of_leaves(Tree,N).
N=4
```

Ülesanne 88. Andke eespool toodud kahendpuu tüübikirjeldus.

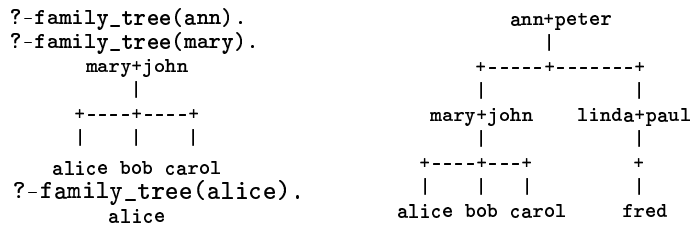
Ülesanne 89. Koostage predikaat *number_of_nodes*, mis leiab kahendpuu kõigi tippude arvu.

Ülesanne 90. Kirjutage programm, mis moodustab kahendpuust listi. (See on üks võimalus teist järku predikaatide *bagof*, *setof* ja *findall* realiseerimiseks.)

Ülesanne 91. Koostage programm, mis trükib kahendpuu välja taanetega. Näiteks eespool kirjeldatud kahendpuu väljastatakse 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 92. Koostage ja trükkige oma sugupuu. Näiteks joonisel 1.1 lk. 7 esitatud suguvõsa korral trükitakse järgmised puud:



Ülesanne 93. Koostage programm seente määramiseks.

6.8. Programmi efektiivsuse hindamine

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

```
?-dynamic test/2.
```

Genereerime predikaadi *test* jaoks esimese argumendi järgi miljon fakti arvudega 1—1 000 000:

```
?-between(1,1000000,X), assert(test(X,a)), fail.
```

Sama teeme predikaadi *test* teise argumendi järgi:

```
?-between(1,1000000,X), assert(test(a,X)), fail.
```

Testime predikaadiga *time*/1, kui palju kulub aega eelviimase faktini jõudmiseks. Selgub, et SWI-Prolog indekseerib predikaatide kirjeldusi vaikimisi esimese argumendi järgi:

```
?-time(test(a,999999)).
% 1 inferences in 0.71 seconds (1 Lips)
?-time(test(999999,a)).
% 1 inferences in 0.00 seconds (Infinite Lips)
```

Sama tulemuse oleksime ilmselt saanud, kasutades Prologi jälgijat (*trace*). Predikaat *time* töötab nagu predikaat *once*, täites päringut täpselt ühe korra ning väljastab päringu täitmiseks kulunud aja ning sekundis sooritatud resolutsioonisammude arvu (*lips*). Kirjutame predikaadi *test*/2 kirjeldused dünaamilisest andmebaasist faili *test.pl* ja teeme sama testi kompileeritud programmiga:

```
?-open('test.pl',write,S,[alias=s]).
?-test(X,Y),write(s,test(X,Y)),write(s,'.'),nl(s),fail.
?-close(s).
?-retractall(test(X,Y)).
?-[test].
% test compiled 13.54 sec, 144,000,288 bytes
?-time(test(a,999999)).
% 1 inferences in 0.11 seconds (9 Lips)
?-time(test(999999,a)).
% 1 inferences in 0.00 seconds (Infinite Lips)
```

Selgub, et erinevalt üldlevinud arvamusest töötavad dünaamilised predikaadid küllaldase efektiivsusega. Kasutatud mälu hulka näeme predikaadiga *statistics*. Täpsemat informatsiooni mingi predikaadi täitmise kohta saame süsteemse predikaadiga *profile*/3:

```
?-profile(test(a,999999),plain,4).
% 1 inferences in 0.09 seconds (11 Lips)

  Predicate   Box Entries =   Calls+Redos = Exits+Fails   Time
  ============================================================
  test/2          999,999 =     1+999,998 = 999,999+0   100.0%
  statistics/2          4 =           4+0 = 4+0           0.0%
  is/2                  3 =           3+0 = 3+0           0.0%
  =/2                   3 =           3+0 = 3+0           0.0%

?-profile(test(999999,a),plain,3).
% 1 inferences in 0.00 seconds (Infinite Lips)

  Predicate   Box Entries =   Calls+Redos = Exits+Fails   Time
  ============================================================
  statistics/2          4 =           4+0 = 4+0           0.0%
  =/2                   4 =           4+0 = 4+0           0.0%
  test/2                1 =           1+0 = 1+0           0.0%
```

7. peatükk

Tekstitöötlus

Teksti sisselugemine ja muutmine. Sõnaraamatu koostamine.

7.1. Sõnad, tähed ja failid

Teksti saab sisse lugeda nii aatomite, struktuursete termide kui ka tähtede kaupa. Aatomite ja termide lugemiseks on predikaat *read*, mis eeldab, et sisseloetavad objektid lõpevad punktiga. Loeme näiteks klaviatuurilt aatomi *üks*:

```
?-read(X).
üks.
X = üks
```

Analoogiliselt loeme sisse sulgudega termi *arv(üks,1)*:

```
?-read(X).
arv(üks,1).
X = arv(üks, 1)
```

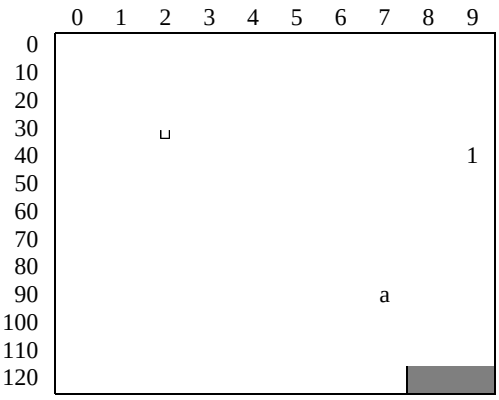
Teksti tähekaupa lugemine on keerulisem, sest teksti hoitakse arvuti mälus ja failides ASCII-koodidena. Samuti on tekstifailides nähtamatud ja mittetrükitavad sümbolid nagu reavahetused ja tühikud. Mingi sümboli ASCII-koodi lugemiseks on predikaadid *get*/1 (loeb järgmise trükitava sümboli koodi ning unifitseerib selle oma argumendiga) ja *get0*/1 (loeb järgmise sümboli koodi ning unifitseerib selle oma argumendiga). Predikaat *put*/1 väljastab ühe sümboli. Selgitame nende predikaatidega välja tähe *a* koodi:

```
?-get(X), put(X).
a
a
X = 97
```

Me vajutasime klahvile *a* ja reavahetusklahvile, mispeale predikaat *put* kirjutas tähe *a* ja väljastas muutuja *X* väärtusena tähe *a* koodi 97. Järelikult sisendis ja väljundis teksti kujul olev informatsioon väljendub programmi siseselt koodidena. Teeme ka kindlaks numbri 1 ja tühiku '␣' koodid:

```
?-get(X).
1
X = 49
?-get0(X).
␣
X = 32
```

Kanname uuritud sümbolid ASCII-koodide tabelisse 7.1, milles on standardselt sümbolid koodidega 0–127.



Tabel 7.1. Tähtede ASCII-koodide tabel.

Ülesanne 94. Kirjutage ASCII-tabelisse ülejäänud väiketähed *a–z* ning suurtähed *A–Z*. Kirjutage tabelisse ka ülejäänud numbrid 0—9. Selgitage, miks asuvad vastavad tähed ja numbrid tabelis rühmiti.

Ülesanne 95. Kirjutage programm, mis trükib sümbolite ASCII-tabeli.

Selgitame veel reavahetusklahvile vastava koodi, mida läheb tarvis teksti reakaupa lugemisel:

```
?-get0(X).
<enter>
X = 10
```

Et erinevates operatsioonisüsteemides on tekstifaili formaadid erinevad, siis võib ka reavahetuse sümboli kood olla erinev.

Et Ameerikas ei kasutata täpitähti, siis puuduvad ASCII standardtabelis eesti keele täpitähed. Sellepärast on kasutusele võetud **laiendatud ASCII-tabel** koodidega 0–255, mille esimene pool langeb kokku standardse ASCII-tabeliga ja teises pooles on enamiku euroopa keelte erisümbolid, sealhulgas eesti keele täpitähed. Et baidis on 8 bitti ja sümbolikoode hoitakse baithaaval, siis oli tabeli laiendamine võimalikule 256 sümbolile igati loomulik.

Ülesanne 96. Koostage laiendatud ASCII-tabel, milles on eesti keele täpitähed.

Tabelis 7.2 on toodud mõned teksti töötlemise standardpredikaadid.

Predikaat	Selgitus
char_code(X,Y)	tähe *X* kood on *Y*
atom_codes(X,Ys)	aatomi *X* koodide list on *Ys*
name(X,Ys)	aatomi või arvu *X* koodide list on *Ys*
atom_chars(X,Ys)	aatomi *X* tähtede list on *Ys*
upcase_atom(X,Y)	aatomi *X* suurtäheline vorm on *Y*
downcase_atom(X,Y)	aatomi *X* väiketäheline vorm on *Y*
atom_concat(X,Y,Z)	aatomite *X* ja *Y* ühend on *Z*

Tabel 7.2. Predikaadid aatomite ja tähtede teisendamiseks.

Teksti töötlemine sümbolite ASCII-koodidega tundub esialgu vaevaline. Püüdke näiteks lugeda listile *Xs* vastavat teksti, kui listile on omistatud predikaadiga *atom_codes*/2 sõna “aabits” tähekoodid:

```
?-atom_codes(aabits,Xs).
Xs = [97, 97, 98, 105, 116, 115]
```

Sama koodide listi annavad jutumärkides **sõned** ja predikaat *name*/2:

```
?-Xs="aabits".
Xs = [97, 97, 98, 105, 116, 115]
?-name(aabits,Xs).
Xs = [97, 97, 98, 105, 116, 115]
```

Samas on SWI-Prologis ka vahendid koodide asemel tähtedega opereerimiseks. Predikaadiga *char_code*/2 saame etteantavale tähele vastava ASCII-koodi. Nagu eespool nägime, vastab tähele *a* kood 97:

```
?-char_code(a,X).
X = 97
```

Reavahetusele vastab täheline kood '\n':

```
?-char_code(X,10).
X = '\n'
```

Aatomile vastava tähtede listi saame predikaadiga *atom_chars*/2:

```
?-atom_chars(aabits,Xs).
Xs = [a, a, b, i, t, s]
```

Predikaat[23]	Selgitus
get(X)	loeb aktiivsest sisendvoost esimese trükitava sümboli ja unifitseerib selle koodi *X*-ga
get0(X)	loeb aktiivsest sisendvoost järgmise baidi ja unifitseerib selle *X*-ga
put(X)	kirjutab tähe *X* või koodile *X* vastava sümboli aktiivsesse väljundvoogu
get_byte(X)	loeb aktiivsest sisendvoost järgmise baidi ja unifitseerib selle *X*-ga
get_char(X)	loeb aktiivsest sisendvoost järgmise tähe ja unifitseerib selle *X*-ga
get_code(X)	loeb aktiivsest sisendvoost järgmise sümboli koodi ja unifitseerib selle *X*-ga
peek_byte(X)	unifitseerib aktiivse sisendvoo viimati loetud baidi *X*-ga
peek_char(X)	unifitseerib aktiivse sisendvoo viimati loetud tähe *X*-ga
peek_code(X)	unifitseerib aktiivse sisendvoo viimati loetud sümboli koodi *X*-ga
put_byte(X)	kirjutab baidi *X* aktiivsesse väljundvoogu
put_char(X)	kirjutab tähe *X* aktiivsesse väljundvoogu
put_code(X)	kirjutab sümboli koodi *X* aktiivsesse väljundvoogu

Tabel 7.3. Sümbolite sisselugemise ja väljastamise predikaadid.

Teksti tähekaupa lugemiseks ja kirjutamiseks on predikaadid *get_char*/1 ja *put_char*/1 (vt. tabel 7.3):

[23] Kõigist neist predikaatidest on olemas ka kahekohalised variandid, mille esimeseks argumendiks on andmevoo nimi.

```
?-get_char(X), put_char(X).
a
a
X = a
```

Antud juhul on muutuja *X* väärtuseks mitte tähe *a* kood, vaid täht *a* ise. Mugav on ka võimalus muuta väiketähelist aatomit suurtäheliseks ja vastupidi, suurtähelist aatomit väiketäheliseks:

```
?-upcase_atom(väiKE,X), downcase_atom(X,Y).
X = 'VÄIKE'
Y = väike
```

Ülesanne 97. Kirjutage programm eestikeelsete sõnade poolitamiseks. Näiteks

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

Ülesanne 98. Kirjutage programm, mis käänab etteantud sõna mingi teise sõna reeglite järgi. Näiteks sõna "mees" käänamisel sõna "naine" järgi annab see programm järgmised vastused:

```
?-kääna(mees,naine,K).
K = mene ;
K = mese ;
K = mest ;
K = mesesse ;
K = mesed ;
K = meste ;
K = mesi ;
K = mestesse ;
no
```

Käänake selle programmiga ka sõna "tuli" järgi sõna "lumi".

Ülesanne 99. Realiseerige predikaat *rot13*, mis roteerib tähtede *a–z* ASCII-koode kolmteist kohta edasi:

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

Ülesanne 100. Kirjutage programm, mis kontrollib, kas etteantud sõnaraamatu abil saab teisendada ühe sõna teiseks sõnaks, muutes igal sammul ära täpselt ühe sõna tähe ning säilitades sõna pikkust. Näiteks olgu sõnaraamatus *sõna*/1 ainult üks sõna:

```
sõna(mets).
```

Predikaat *sõnaredel* annab järgmise teisenduste ahela:

```
?-sõnaredel(mees,kets).
mees-mets-kets
```

Predikaat	Selgitus
see(F)	muudab faili *F* aktiivseks sisendvooks, avades vajadusel faili *F* lugemiseks
seeing(F)	unifitseerib faili *F* aktiivse sisendvoo nimega
seen	sulgeb aktiivse sisendvoo ning aktiviseerib andmevoo *user*
tell(F)	muudab faili *F* aktiivseks väljundvooks; vajadusel loob ja avab faili *F* väljastamiseks
append(F)	nagu *tell*/1, kuid olemasoleva faili korral kirjutab selle lõppu
telling(F)	unifitseerib faili *F* aktiivse väljundvoo nimega
told	sulgeb aktiivse väljundvoo ning aktiviseerib andmevoo *user*
open(F,R,S)	avab faili *F* režiimis *R* ning seostab sellega andmevoo *S*
open(F,R,S,P)	avab faili *F* režiimis *R* ning seostab sellega andmevoo *S* parameetritega *P*
close(S)	sulgeb andmevoo *S*
close(S,P)	sulgeb andmevoo *S* parameetritega *P*
current_stream(O,R,S)	näitab andmevoo *S* režiimi *R* ja osutust *O*
at_end_of_stream(S)	sisendvoog *S* on jõudnud faili lõppu

Tabel 7.4. Failide avamise ja sulgemise predikaadid.

Tabelis 7.4 on toodud mõned failide avamise ja sulgemise predikaadid. Tekstifailide avamiseks ja sulgemiseks saab kasutada predikaate *see*/1, *seen*/0, *tell*/1 ja *told*/0, mis suunavad aktiivse sisendi ja väljundi ajutiselt ümber vastavasse faili. Näiteks soovides luua jooksvasse kataloogi faili *proov* ning kirjutada sellesse lause “Tere!”, võime kirjutada päringu:

```
?-tell(proov), write('Tere!'), nl, told.
```

Väljundi selline ümbersuunamine on tavaliselt tülikas, mistõttu tuleks eelistada andmevoogude (ingl. k. *stream*) kasutamist. Sama töö saame teha andmevoogude abil päringutega

```
?-open(proov,write,S).
S = '$stream'(629834)
?-S='$stream'(629834), write(S,'Tere!'), nl(S).
?-close('$stream'(629834)).
```

Predikaadiga *open* avame faili *proov* kirjutamiseks (ingl. k. *write*), sidudes sellega andmevoo *S*. Kirjutamis- ja reavahetuspredikaatides *write* ja *nl* anname esimese argumendina andmevoo nime ′$*stream*′(629834). Faili kirjutamise

Predikaat[24]	Selgitus
read(X)	loeb aktiivsest sisendvoost termi ja unifitseerib selle termiga *X*
write(X)	kirjutab termi *X* aktiivsesse väljundvoogu
writeq(X)	kirjutab termi *X* aktiivsesse väljundvoogu, nii et see on predikaadiga *read*/1 sisse loetav, kasutades vajadusel sulgusid, tehteid ja apostroofe
write_canonical(X)	kirjutab termi *X* aktiivsesse väljundvoogu, ignoreerides tehete kirjeldusi ning kasutades väljastamiseks termide standardset suluesitust
tab(N)	kirjutab aktiivsesse väljundvoogu *N* tühikut
nl	kirjutab aktiivsesse väljundvoogu reavahetuse

Tabel 7.5. Termide sisselugemise ja väljastamise predikaadid.

lõppedes tuleb fail sulgeda predikaadiga *close*. Andmevoogude nimede meelespidamise lihtsustamiseks võib võtta predikaadis *open*/4 kasutusele globaalse nime *s* ja kasutada seda voo tähistajana kirjutamis- ja sulgemispredikaatides:

```
?-open(proov,write,S,[alias=s]).
S = s
?-write(s,'Tere!'), nl(s).
?-close(s).
```

Lugemiseks avatud failist saame lugeda üksikuid tähti:

```
?-open(proov,read,S,[alias=s]).
S = s
?-get_char(s,C1), get_char(s,C2), close(s).
C1 = 'T'
C2 = e
```

Samuti saame kirjutada rekursiivse predikaadi, mis loeb ära faili kõik tähed, väljastades need samal ajal ekraanile:

```
loe_tähed:-
  get_char(s,C),
  writeq(C), nl,
  loe_tähed.
```

[24] Kõigist neist predikaatidest on olemas ka kahekohalised variandid, mille esimeseks argumendiks on andmevoo nimi.

Kahjuks läheb see programm lõpmatusse tsüklisse. Probleem seisneb selles, et faili tähed saavad otsa ja pärast hüüumärgile järgnevat reavahetust väljastatakse lõpmatult konstanti *end_of_file*:

```
?-open(proov,read,S,[alias=s]), loe_tähed.
'T'
e
r
e
!
'\n'
end_of_file
end_of_file
...
```

Lõpmatu tsükli vältimiseks kirjutame predikaadi *loe_tähed* kirjelduste algusesse veel faili lõppu märkivat predikaati *at_end_of_stream*/1 kasutava kontrolli:

```
loe_tähed:-at_end_of_stream(s), !.
```

Nüüd lõpeb failist lugemine ilusti ära ja saame selle sulgeda:

```
?-open(proov,read,S,[alias=s]), loe_tähed, close(s).
...
'\n'
yes
```

Sümbolite ASCII-koode lugevates predikaatides *get_code*/2 ja *get_byte*/2 saame faili lõpu korral koodi -1.

7.2. Sümbolite asendamine

Kirjutame programmi, mis loeb sisse tekstifaili ja asendab selles esinevad numbrid tärnidega. Olgu näiteks sisendfailiks joonisel 7.1 toodu.

> Palun helista minu telefonile 7 375 497.
> Aitäh!

Joonis 7.1. Fail *sisend.txt*.

Kõigepealt küsime sisend- ja väljundfaili nime, avame nimetatud failid globaalsete voonimedega *in* ja *out* ning seejärel asendame predikaadiga *asenda_numbrid* numbrid tärnidega ja sulgeme failid:

```
asenda_tärnidega:-
  write('Kirjuta sisendfaili nimi: '),
  read(Fail),
  open(Fail,read,S,[alias=in]),
  write('Kirjuta väljundfaili nimi: '),
  read(Fail2),
  open(Fail2,write,S2,[alias=out]),
  asenda_numbrid,
  close(S),
  close(S2).
```

Predikaat *asenda_numbrid* loeb rekursiivselt sisendfailist ühe tähe, teisendab selle predikaadiga *asenda_number_tärniga* soovitud kujule ja väljastab saadud tähe väljundfaili. Sisendfaili lõppu kontrollime süsteemse predikaadiga *at_end_of_stream*/1:

```
asenda_numbrid:-at_end_of_stream(in), !.
asenda_numbrid:-
  get_char(in,X),
  asenda_number_tärniga(X,Y),
  put_char(out,Y),
  asenda_numbrid.
```

Tähe teisendamise teeme esialgu fiktiivselt, s.t. kanname iga tähe üle sellisena nagu see loeti:

```
asenda_number_tärniga(X,X).
```

Käivitame nüüd prooviks predikaadi *asenda_tärnidega*:

```
?-asenda_tärnidega.
Kirjuta sisendfaili nimi: 'sisend.txt'.
Kirjuta väljundfaili nimi: 'väljund.txt'.
yes
```

Et vastuseks saime *yes*, siis peab väljundfailis *väljund.txt* olema täpselt sama tekst, mis sisendfailis.

Kirjutame nüüd predikaadi *asenda_number_tärniga* kirjelduste algusesse sümboli 7 asendamise tärniga:

```
asenda_number_tärniga('7',*).
```

Palun helista minu telefonile * 3*5 49*.
Aitäh!

Joonis 7.2. Fail *väljund.txt*.

Predikaadi *asenda_tärnidega* täitmise tulemusena saame nüüd ülaltoodud faili.

Ülesanne 101. Täiendage programmi *asenda_tärnidega* nii, et tärnidega asendatakse kõik numbrid.

7.3. Alamsõnade asendamine

Järgmiseks kirjutame programmi *asenda_sõnad*, mis loeb sisse tekstifaili ja asendab selles reakaupa alamsõned etteantud sõnedega:

```
asenda_sõnad:-
  open('sisend.txt',read,S,[alias=in]),
  open('väljund.txt',write,S2,[alias=out]),
  write('sisend.txt'->'väljund.txt'), nl,
  asenda_read,
  close(S),
  close(S2).
```

Predikaat *asenda_read* loeb rekursiivselt sisendfailist ühe rea, asendab selles predikaadiga *asenda_sõnad*/2 alamsõned ja väljastab saadud tekstirea väljundfaili:

```
asenda_read:-at_end_of_stream(in), !.
asenda_read:-
  loe_rida(R),
  asenda_sõnad(R,R2),
  kirjuta_rida(R2),
  asenda_read.
```

Teksti reakaupa lugemisel on vaja ära tunda realõpud, milleks kasutame predikaati *end_of_line*/1:

```
end_of_line('\n').
```

Rea sisselugemisel kasutame osalist listi $[C|Cs]$, mille esimeseks sümboliks on loetav täht *C* ja mille sabale *Cs* omistame rea lõppedes tühilisti:

```
loe_rida([]):-at_end_of_stream(in), !.[25]
loe_rida([C|Cs]):-
  get_char(in, C),
  (end_of_line(C)->Cs=[]
  ;
  loe_rida(Cs)).
```

Alamsõnede asendamise teeme esialgu fiktiivselt, kandes kõik tähed muutmata kujul sisendlistist väljundlisti:

```
asenda_sõnad([A|As],[A|Bs]):-
  asenda_sõnad(As,Bs).
asenda_sõnad([],[]).
```

[25] Osa tekstiredaktoreid lubab mittetühja rea lõpus ka faili lõpu tunnust.

Muudetud tekstirea väljastamiseks kirjutame selle sümbolid väljundfaili:

```
kirjuta_rida([]).
kirjuta_rida([C|Cs]):-
  put_char(out,C),
  kirjuta_rida(Cs).
```

Predikaadi *asenda_sõnad*/0 täitmine annab nüüd tulemuse

```
?-asenda_sõnad.
sisend.txt->väljund.txt
yes
```

Et vastuseks saime *yes*, siis peab väljundfailis *väljund.txt* olema täpselt sama tekst, mis sisendfailis. Alamsõne "helista" asendamiseks sünonüümiga "kõlista" kirjutame nüüd predikaadi *asenda_sõnad*/2 kirjelduste algusesse reegli, mis kontrollib, kas sisendlisti seitse esimest sümbolit moodustavad sõna "helista" ja asendab need sõna "kõlista" sümbolitega:

```
asenda_sõnad([h,e,l,i,s,t,a|As],[k,õ,l,i,s,t,a|Bs]):-
  asenda_sõnad(As,Bs).
```

Predikaadi *asenda_sõnad*/0 täitmise tulemusena saame nüüd kõrvaltoodud faili *väljund.txt*.

Palun kõlista minu telefonile 7 375 497.
Aitäh!

Joonis 7.3. Fail *väljund.txt*.

Ülesanne 102. Täiendage programmi *asenda_sõnad* nii, et selles asendatakse sobivate sünonüümidega kõik sisendfailis esinevad sõnad.

7.4. Sõnaraamatu koostamine

Järgmiseks kirjutame programmi *leia_sõnad*, mis koostab etteantud faili põhjal selle sõnade nimekirja. Võtame aluseks faili joonisel 7.1. Sellest sõnade väljaeraldamiseks peame määrama, millistest sümbolitest moodustuvad sõnad ja millised sümbolid on sõnade eraldajad. Valitud sisendfailis on eraldajateks tühikud, punkt ja hüüumärk. Lisame nendele ka realõpusümboli '\n', mis on kirjeldatud programmis *asenda_sõnad*/0 eespool:

```
end_of_word(' ').
end_of_word('.').
end_of_word('!').
end_of_word(C):-end_of_line(C).
```

Sõnades kasutatakse tähtedena väiketähti, suurtähti ja numbreid:

```
word_character(X):-chars(Tähed), member(X,Tähed).
chars([a,e,f,h,i,l,m,n,o,s,t,u]).
chars(['A','P']).
chars([ä]).
chars(['0','1','2','3','4','5','6','7','8','9']).
```

Programm *leia_sõnad* sarnaneb programmiga *asenda_sõnad*, kuid väljastab leitud sõnad mitte otse, vaid predikaati *sõna*/1 kirjeldavate faktidena. Valitud sisendfaili korral väljastatakse joonisel 7.4 toodud fail *sõnad.pl*:

```
leia_sõnad:-
  open('sisend.txt',read,S,[alias=in]),
  open('sõnad.pl',write,S2,[alias=out]),
  write('sisend.txt->sõnad.pl'), nl,
  leia_sõnad_ridadest,
  close(S),
  close(S2).
```

Predikaat *leia_sõnad_ridadest* loeb rekursiivselt sisendfailist ühe rea, eraldab sellest predikaadiga *rea_sõnad* sõnad ja kirjutab need formaaditud kujul väljundfaili:

```
leia_sõnad_ridadest:-at_end_of_stream(in),!.
leia_sõnad_ridadest:-
  loe_rida(R),
  rea_sõnad(R,Sõnad),
  kirjuta_sõnad(Sõnad),
  leia_sõnad_ridadest.
```

```
sõna('Palun').
sõna(helista).
sõna(minu).
sõna(telefonile).
sõna('7').
sõna('375').
sõna('497').
sõna('Aitäh').
```

Joonis 7.4. Fail *sõnad.pl*.

Predikaat *rea_sõnad* kontrollib, kas listi esimene element *C* on sõnas kasutatav täht ja koostab sellistest järjestikustest tähtedest listi *Tähed* ja sõna *Sõna*:

```
rea_sõnad([C|Cs],[Sõna|Sõnad]):-
  word_character(C),
  moodusta_sõna([C|Cs],Tähed,Cs2),
  atom_chars(Sõna,Tähed),
  rea_sõnad(Cs2,Sõnad).
```

Kui oleme jõudnud sõna lõppu, siis jätame kõik eraldajad vahele, kuni jõuame järgmise sõnani või listi lõppu:

```
rea_sõnad([C|Cs],Sõnad):-
  end_of_word(C),
  rea_sõnad(Cs,Sõnad).
rea_sõnad([],[]).
```

Predikaat *moodusta_sõna*/3 tõstab tähed listi algusest moodustatava sõna tähtedeks, kuni jõuatakse esimese eraldajani või listi lõppu:

```
moodusta_sõna([C|Cs],[C|Ws],Cs2):-
  word_character(C),
  moodusta_sõna(Cs,Ws,Cs2).
moodusta_sõna([C|Cs],[],[C|Cs]):-
  end_of_word(C).
moodusta_sõna([],[],[]).
```

Tekstireast leitud sõnad kirjutame ükshaaval väljundfaili:

```
kirjuta_sõnad([]).
kirjuta_sõnad([W|Ws]):-
  writeq(out,sõna(W)),
  write(out,'.'),
  nl(out),
  kirjuta_sõnad(Ws).
```

Programmi *leia_sõnad*/0 käivitamisel saame tulemuseks

```
?-leia_sõnad.
sisend.txt->sõnad.pl
yes
```

Loodud sõnadefaili saame programmina sisse lugeda ja moodustada selle põhjal sõnade listi *Sõnaraamat*:

```
?-[sõnad].
?-bagof(X,sõna(X),Sõnaraamat).
```

Ülesanne 103. Täiendage programmi *leia_sõnad*, lisades sellele puuduvaid tähti ja kirjavahemärke.

Ülesanne 104. Kirjutage programm, mis koostab antud tekstis esinevate sõnade sagedussõnaraamatu.

Ülesanne 105. Kirjutage programm, mis pakib sõnaraamatu Cooperi meetodil, asendades sõnade kokkulangevad prefiksid kokkulangevate tähtede arvudega. Näiteks

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

Ülesanne 106. Kirjutage programm, mis väljastab tekstis esinevate tähtede sagedustabeli mittekasvavas järjestuses. Näiteks

```
?-frequency(['Siin',on,tekst]).
s - 2 (18%)
i - 2 (18%)
n - 2 (18%)
t - 2 (18%)
o - 1 (9%)
e - 1 (9%)
k - 1 (9%)
```

7.5. Eliza

Joseph Weizenbaum realiseeris aastatel 1964–1966 programmi Eliza, mis imiteeris psühhoterapeudi esimest vestlust patsiendiga. Vestluse iseloom oli muudetav vastavate skriptidega.

Joonisel 7.5 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 perekonnast. Mõned patsiendi laused sõnastab Eliza ringi, tehes nendes sobivad asendused. Näiteks asendab ta lauses

> Te ei vaidle minuga.

sõna “Te” sõnaga “Ma” ja sõna “minuga” sõnaga “teiega”, võimaldades patsiendil endast rohkem rääkida:

> Ma ei vaidle teiega?

Ülesanne 107. Kirjutage triviaalne tõlkimisprogramm, mis asendab ingliskeelsed sõnad eestikeelsete vastetega. Näiteks

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

Ülesanne 108. Formuleerige joonisel 7.5 toodu põhjal Eliza vastamisreeglid.

Ülesanne 109. Kirjutage programm, mis jäljendab joonise 7.5 dialoogi.

Ülesanne 110. Mõelge välja dialoog ja kirjutage sellele vastav programm.

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 7.5. Patsiendi ja ELIZA vaheline dialoog.

8. peatükk

Grammatikad

Lause süntaksi kirjeldamine. Arvu ja käände ühildumise arvestamine. Süntaksipuu konstrueerimine. Lause semantika esitamine ja lausete tõlkimine.

8.1. Eesti keele lihtgrammatika

Süntaksianalüüsis on hästi tuntud kontekstivabad grammatikad, mis esitatakse asendusreeglite ehk produktsioonide abil, kus noolest vasakul on asendatav mitteterminaal ja paremal asenduse tulemus: terminaalidest ja mitteterminaalidest moodustatud sõne:

<mitteterminaal> $\longrightarrow$ <sõne>

Tegelikult on Prolog kontekstivabadest grammatikatest võimsam. Saab näidata, et Prologis on kirjeldatavad kõik Turingi masinad. Järelikult on Prolog nagu teisedki programmeerimiskeeled Turingi masinatega ekvivalentne.[26]

Ülesanne 111. Tõestage, et Prologi programmidena saab realiseerida kõik rekursiivsed funktsioonid.

Prologile on sisse ehitatud **grammatikareeglid** kujul *a*`-->`*b*, mis vastavad Chomsky grammatikate produktsioonireeglitele. Grammatikareeglite abil esitatud grammatikaid nimetatakse Horni grammatikateks. Nendes esitatakse mitteterminaalid tavalisel atomaarse predikaadi kujul ja terminaalid nurksulgudes listina.

[26] Jutt on grammatikate ja keelte Chomsky hierarhiast.

Kirjutame grammatika analüüsimaks eestikeelset lauset “noor tubli mees kannab kive”. Kõigepealt kirjeldame mitteterminaalidega lause põhikonstruktsioonid. Lause koosneb noomenifraasist ja sellele järgnevast verbifraasist:

```
lause --> noomenifraas, verbifraas.
```

Noomenifraas koosneb nimisõnast, mille ees on kaks omadussõna:[27]

```
noomenifraas --> omadussõna, omadussõna, nimisõna.
```

Verbifraas koosneb tegusõnast ja sellele järgnevast osastavas käändes nimisõnast, mille tähistame segaduse vältimiseks mitteterminaaliga *nimisõna2*:

```
verbifraas --> tegusõna, nimisõna2.
```

Nimetavas ja osastavas käändes nimisõnadeks ja tegusõnaks on vastavalt terminaalid “mees”, “kive” ja “kannab”:

```
nimisõna --> [mees].
nimisõna2 --> [kive].
tegusõna --> [kannab].
```

Omadussõnaks võivad olla terminaalid “noor” ja “tubli”:

```
omadussõna --> [noor] ; [tubli].
```
[28]

Mingi lause (süntaktilise) korrektsuse kontrollimiseks esitame sõnad aatomitena ja lause sõnade listina. Seejärel kasutame predikaati *phrase*/2, mille esimeseks argumendiks on analüüsitavale lausekonstruktsioonile vastav mitteterminaal ja teiseks argumendiks sõnade list:

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

Predikaat *phrase*/2 on lühendatud vorm predikaadist *phrase*/3, mille kolmandaks argumendiks on listi analüüsimata lõpp, s.t. antud juhul tühilist:

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

Ent analüüsides sama lauset noomenifraasina, jääb lause lõpp analüüsimata:

```
?-phrase(noomenifraas,[noor,tubli,mees,kannab,kive],Xs).
Xs=[kannab,kive]
```

Et grammatikareeglid esitatakse sisemiselt kaheargumendiliste diferentstlistidena, siis võib predikaadi *phrase*/3 esimese argumendi tõsta sulgude ette sõna “phrase” asemele:

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

[27] Noomen ehk käändsõna on kas nimisõna, arvsõna, asesõna või omadussõna.

[28] Grammatikareegli paremas pooles saab esitada ka semikooloniga eraldatud alternatiive.

Grammatikareegli terminaalides ja mitteterminaalides jäetakse diferentslisti standardsed argumendid ära, sest Prolog võimaldab kirjutada kontekstivabu grammatikaid loomulikumas esituses, hoides grammatikareeglеid sisemises andmebaasis predikaatidena, millel on kaks lisaargumenti. Vastav teisendus tehakse grammatikareeglite sisselugemisel automaatselt.

Ülesanne 112. Muutke grammatikat nii, et saaksime sama grammatikaga analüüsida laused “mees kannab” ja “mees kannab kive”.

Ülesanne 113. Muutke grammatikat nii, et selles oleksid neli nimetavas käändes nimisõna: Kalevipoeg, Sulevipoeg, Olevipoeg ja Linda, neli osastavas käändes nimisõna: kive, Kalevipoega, Sulevipoega ja Olevipoega ja neli verbi: kannab, korjab, veeretab ja viskab.

Ülesanne 114. Kirjutage grammatika, et saaks analüüsida lauset “kena sale tüdruk kohendab rahulikult juukseid”.

Ülesanne 115. Laiendage esialgset grammatikat lausetele, milles nimisõnal võib olla suvaline arv täiendeid. Näiteks “noor tubli osav ilus mees kannab kive”.

Ülesanne 116. Laiendage esialgset grammatikat lausetele, mis võivad sisaldada osalauseid. Näiteks “mees ütleb, et noor mees kannab kive”.

Ülesanne 117. Koostage grammatika, mis tunneb ära sõnad kujul $a^n b^n$ ($n \geqslant 0$). Näiteks tunnistatakse süntaktiliselt korrektseks sõna a^2b^2:

```
?-phrase(ab,[a,a,b,b]).
```

Ülesanne 118. Koostage aritmeetiliste avaldiste grammatika.

Ülesanne 119. Koostage lauseloogika valemite grammatika.

8.2. Grammatikareeglite teisendamine

Quintus Prologis ei kasutata tavalisi kaheargumendilisi diferentsliste, vaid sisselugemisel töödeldakse grammatikareeglеid predikaadiga *C* (ingl. k. *connects*). Näiteks reegel

```
nimisõna --> [mees].
```

teisendatakse kujule

```
nimisõna(Ls,Ls2):-'C'(Ls,mees,Ls2).
```

Predikaat *C* on kirjeldatud kui

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

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

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

Reegliga analüüsitakse kõigepealt sõna "sest", väljastatakse teade "Käes!" 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).
```

Vanemates Prologisüsteemides on reegli sisekujuks

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

See on eksitav, sest väljastab teate alles pärast mõlema sõna analüüsimist. Lõikepredikaadi '!' kasutamisel võivad tulemused oluliselt erineda.

Ülesanne 120. Koostage grammatika, mis tunneb ära sõnad kujul $a^n b^n c^n$ ($n \geqslant 0$). Näiteks

```
?-phrase(abc,[a,a,b,b,c,c]).
```

Ülesanne 121. (M. Covington) Prologi grammatikareeglid teisendatakse programmi kompileerimise käigus automaatselt predikaadiga *expand_term*/2 sisekujule, muutmata ülejäänud terme. Seda predikaati kasutavad programmi sisselugemiseks ja automaatseks teisendamiseks süsteemsed predikaadid *consult* ja *reconsult*. Kontrollige, kas teie Prologisüsteem tunneb predikaati *expand_term*:

```
?-expand_term((lause --> nf, vf), Tulemus).
?-expand_term(imeline(veeb), Tulemus).
```

Ülesanne 122. (M. Covington) Uurige, kuidas teie süsteem teisendab reeglit

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

Millised on teisenduse eelised ja puudused?

8.3. Grammatika täiustamine

Punktis 8.1 toodud grammatikat on võimalik parameetrite lisamisega laiendada, sest grammatikareeglite mitteterminaalidel võivad olla ka argumendid. Diferentslisti kaks sisemist lisaargumenti lisatakse nende järele. Horni grammatikates on lihtne tagada ühilduvust arvus: lause alus ja öeldis peavad olema samaaegselt kas ainsuses või mitmuses. Sihitis võib olla alusest ja öeldisest sõltumatult kas ainsuses või mitmuses.

Erinevalt inglise keelest on eesti keeles tähtsad ka sõnade käänded. Nimelt on lause alus tavaliselt nimetavas käändes, osasihitis osastavas ja täissihitis nimetavas või omastavas käändes. Aluse ja sihitisega samas käändes peavad olema ka nende täiendid:

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

Järgmised laused loeme aga grammatiliselt vigaseks:

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

Sõnade arvu ja käände ühildamiseks lisame kõigile sõnaliikidele arvu lisaargumendi ja nimisõnale ja omadussõnale veel käände lisaargumendi.

Argumendiga *Arv* tagame, et lause alus ja öeldis on samaaegselt kas ainsuses või mitmuses:

```
lause --> noomenifraas(Arv), verbifraas(Arv).
```

Argument *Arv* kandub edasi predikaadi *nimisõna* päisesse. Lisaks öeldisega arvus ühildumisele peab alus ühilduma käändes ka täiendiga:

```
noomenifraas(Arv) --> omadussõna(Arv,Kääne),
                      nimisõna(Arv,Kääne).
```

Verbifraasis peab noomenifraasiga arvus ühilduma öeldis, ent sihitis ja selle täiend ühilduvad arvus ja käändes omaette, sõltumata aluse ja öeldise ainsusest või mitmusest:

```
verbifraas(Arv) --> tegusõna(Arv), noomenifraas(_).
```

Sõnaraamatus märgime ära iga sõna arvu ja käände:

```
nimisõna(ainsus,nimetav) --> [mees].
nimisõna(mitmus,nimetav) --> [mehed].
nimisõna(ainsus,osastav) --> [kivi].
nimisõna(mitmus,osastav) --> [kive].
tegusõna(ainsus) --> [kannab].
tegusõna(mitmus) --> [kannavad].
omadussõna(ainsus,nimetav) --> [noor] ; [suur].
omadussõna(mitmus,nimetav) --> [noored] ; [suured].
omadussõna(ainsus,osastav) --> [suurt].
omadussõna(mitmus,osastav) --> [suuri].
```

Arvu ja käänet arvestava grammatika käivitame samamoodi kui eelmise grammatika:

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

8.4. Lause semantika

Grammatikareeglitega saab defineerida ka lausete semantika ehk tähenduse. Lause kõige üldisemat semantikat esindab selle **süntaksipuu** (ingl. k. *parse tree*), millesse kogutakse kõik lause analüüsimisel kasutatud terminaalid ja mitteterminaalid. Sisuliselt on see tõestus, et lause on antud grammatikaga analüüsitav. Lause süntaksipuust saab moodustada lause mitmesuguseid konkreetseid semantikaid ehk lauseosade tähendusi.

Lause süntaksipuu ehitamiseks moodustame termi *lause(N,V)*, millesse salvestame noomenifraasi süntaksipuu *N* ja verbifraasi süntaksipuu *V*:

```
lause(lause(N,V)) --> noomenifraas(N), verbifraas(V).
```

Noomenifraasi analüüsimisel salvestame termi *nfraas(O1,O2,N)* omadussõnade ja nimisõna süntaksipuud:

```
noomenifraas(nfraas(O1,O2,N)) -->
          omadussõna(O1), omadussõna(O2), nimisõna(N).
```

Samuti toimime verbifraasis termiga *vfraas(T,N)*:

```
verbifraas(vfraas(T,N)) --> tegusõna(T), nimisõna2(N).
```

Terminaalide korral lõpetame süntaksipuu ära, kirjutades iga sõna ette sulgudesse tema liigi:

```
nimisõna(nimisõna(mees)) --> [mees].
nimisõna2(nimisõna2(kive)) --> [kive].
tegusõna(tegusõna(kannab)) --> [kannab].
omadussõna(omadussõna(noor)) --> [noor].
omadussõna(omadussõna(tubli)) --> [tubli].
```

Analüüsitava lause süntaksipuu saame nüüd päringuga

```
?-phrase(lause(Puu),[noor,tubli,mees,kannab,kive]).
Puu = lause(nfraas(omadussõna(noor),
                   omadussõna(tubli),
                   nimisõna(mees)),
            vfraas(tegusõna(kannab),
                   nimisõna2(kive)))
```

Lause “noor tubli mees kannab kive” **kompositsionaalse semantikana** võib käsitleda atomaarset predikaati *kannab(mees)*. Selle predikaadi konstrueerimiseks leiame vaadeldavast lausest aluse *A* ja öeldise *Q* ning moodustame metapredikaadiga =../2 nende listist [*Q*, *A*] struktuuri *Q(A)*:

```
lause(Sem) --> noomenifraas(A), verbifraas(Q),
               {Sem=..[Q,A]}.
```

Noomeni- ja verbifraasist eraldame vajalikud sõnad, kasutades lisaargumenti:

```
noomenifraas(A) --> omadussõna, omadussõna, nimisõna(A).
verbifraas(Q) --> tegusõna(Q), nimisõna2.
```

Samuti tuleb lisaargument lisada mitteterminaalidele *nimisõna* ja *tegusõna*:

```
nimisõna(mees) --> [mees].
nimisõna2 --> [kive].
tegusõna(kannab) --> [kannab].
omadussõna --> [noor] ; [tubli].
```

Lause semantika arvutamiseks anname järgmise päringu:

```
?-phrase(lause(Sem),[noor,tubli,mees,kannab,kive]).
Sem = kannab(mees)
```

Lause semantikat saab kasutada **küsimustele vastamisel**. Näiteks küsimusele “kes” vastamisel asendame mitteterminaali *lause* kirjelduse reegliga, mis võtab lausest välja ainult aluse:

```
lause(kes(A)) --> noomenifraas(A), verbifraas(Q).
```

Vastus küsimusele “kes” on vaadeldava lause korral “mees”:

```
?-phrase(lause(kes(X)),[noor,tubli,mees,kannab,kive]).
X = mees
```

Ülesanne 123. Realiseerige semantilised küsimused “mis”, “mida” ja “milline”.

Ülesanne 124. Modifitseerige grammatikat nii, et tegusõna “kannab” kirjeldatakse transitiivse tegusõnana kujul

```
tegusõna(kannab(X,Y),X,Y) --> [kannab].
```

Noomeni- ja verbifraasi analüüsimisel asendatakse termis *kannab(X,Y)* muutujad *X* ja *Y* lause aluse ja sihitisega ilma metapredikaati =../2 kasutamata. Näiteks

```
?-phrase(lause(Sem),[noor,tubli,mees,kannab,kive]).
Sem = kannab(mees,kive)
```

8.5. Lausete tõlkimine

Üks loogilise programmeerimise eesmärke on olnud lausete ühest keelest teise tõlkimine. Seda on mõistlik teha, fikseerides erinevates keeltes lausetele sarnase semantilise termi, mida on lihtsam tõlkida. Vastaku näiteks eestikeelsele lausele “noor tubli mees kannab kive” semantiline term

lause(nsem(mees, noor, tubli), vsem(kannab, kive)).

Selle termi saab tõlkida sõnahaaval inglise keelde:

lause(nsem(man, young, good), vsem(carries, stones)).

Seejärel anname analoogilise grammatika vastavale ingliskeelsele lausele "a good young man carries stones", mille abil saab leitud ingliskeelse semantilise termi jaoks konstrueerida korrektse ingliskeelse lause.

Lause semantilise termi saamiseks konstrueerime termi *lause(Nsem,Vsem)*, millesse salvestame noomenifraasi semantika *Nsem* ja verbifraasi semantika *Vsem*. Lisaks peavad noomenifraas ja verbifraas ühilduma omavahel keeles. Kui inglise keeles on noomenifraas verbifraasi ees, siis eesti keeles võib noomenifraas olla ka verbifraasi taga:

```
lause(lause(Nsem,Vsem)) -->
         noomenifraas(eesti,Nsem),
         verbifraas(eesti,Vsem).
lause(lause(Nsem,Vsem)) -->
         verbifraas(eesti,Vsem),
         noomenifraas(eesti,Nsem).
lause(lause(Nsem,Vsem)) -->
         noomenifraas(inglise,Nsem),
         verbifraas(inglise,Vsem).
```

Eestikeelne noomenifraas erineb ingliskeelsest selle poolest, et ingliskeelses fraasis võib lisaks olla kas määrav või umbmäärane artikkel:

```
noomenifraas(eesti,nsem(A,T1,T2)) -->
         omadussõna(eesti,T1), omadussõna(eesti,T2),
         nimisõna(eesti,A).
noomenifraas(inglise,nsem(A,T1,T2)) -->
         artikkel(inglise), omadussõna(inglise,T1),
         omadussõna(inglise,T2), nimisõna(inglise,A).
```

Eesti keeles ei ole ka määratud öeldise ja sihitise järjekord:

```
verbifraas(eesti,vsem(Q,S)) -->
         tegusõna(eesti,Q), nimisõna2(eesti,S).
verbifraas(eesti,vsem(Q,S)) -->
         nimisõna2(eesti,S), tegusõna(eesti,Q).
verbifraas(inglise,vsem(Q,S)) -->
         tegusõna(inglise,Q), artikkel(inglise),
         nimisõna2(inglise,S).
```

Sõnastiku moodustame paralleelselt kahes keeles, tuues predikaatide teise argumendina välja semantilise termi elemendi:

```
artikkel(inglise) --> [] ; [a] ; [the].
nimisõna(eesti,mees) --> [mees].
nimisõna(inglise,man) --> [man].
nimisõna2(eesti,kive) --> [kive].
nimisõna2(inglise,stones) --> [stones].
tegusõna(eesti,kannab) --> [kannab].
tegusõna(inglise,carries) --> [carries].
omadussõna(eesti,noor) --> [noor].
omadussõna(inglise,young) --> [young].
omadussõna(eesti,tubli) --> [tubli].
omadussõna(inglise,good) --> [good].
omadussõna(inglise,capable) --> [capable].
```

Nüüd on meil olemas nii eesti- kui ka ingliskeelse lause semantikad:

```
?-phrase(lause(Sem),[noor,tubli,mees,kannab,kive]).
Sem=lause(nsem(mees,noor,tubli), vsem(kannab,kive))
?-phrase(lause(Sem),[a,good,young,man,carries,stones]).
Sem=lause(nsem(man,good,young), vsem(carries,stones))
```

Tõlkimiseks kasutame predikaati *tõlgi*/2, mis seab eestikeelsele lausele *Est* vastavusse ingliskeelse lause *Eng*, ühildades sõnakaupa nende semantilised termid *SemEst* ja *SemEng*:

```
tõlgi(Est,Eng):-
  phrase(lause(SemEst),Est),
  tõlgi_semantika(SemEst,SemEng),
  phrase(lause(SemEng),Eng).
```

Lause semantilise termi tõlgime predikaadiga *tõlgi_semantika* fraaside kaupa: noomeni- ja verbifraasi eestikeelsed semantikad *Nsem* ja *Vsem* tõlgime vastavateks ingliskeelseteks semantikateks *Nsem2* ja *Vsem2*:

```
tõlgi_semantika(lause(Nsem,Vsem),
                lause(Nsem2,Vsem2)):-
   tõlgi_semantika(Nsem,Nsem2),
   tõlgi_semantika(Vsem,Vsem2).
```

Fraaside semantikad tõlgime sõnakaupa predikaadiga *tõlgi_sõna*:

```
tõlgi_semantika(nsem(A,T1,T2),nsem(A2,T12,T22)):-
   tõlgi_sõna(A,A2),
   tõlgi_sõna(T1,T12),
   tõlgi_sõna(T2,T22).
tõlgi_semantika(vsem(Q,S),vsem(Q2,S2)):-
   tõlgi_sõna(Q,Q2),
   tõlgi_sõna(S,S2).
```

Lõpuks anname eesti- ja ingliskeelsete sõnade vastavuse, kus ühel sõnal võib olla mitu vastet:

```
tõlgi_sõna(mees,man).
tõlgi_sõna(kannab,carries).
tõlgi_sõna(noor,young).
tõlgi_sõna(tubli,good).
tõlgi_sõna(tubli,capable).
tõlgi_sõna(kive,stones).
```

Eesti keelest inglise keelde tõlkimisel saame lausele “noor tubli mees kannab kive” mitu vastet, sest aluse ja sihitise ees võib olla artikkel ja võib mitte olla:

```
?-tõlgi([noor,tubli,mees,kannab,kive],Eng).
Eng = [young, good, man, carries, stones] ;
Eng = [young, good, man, carries, a, stones] ;
...
```

Ingliskeelse lause tõlkimisel saame neli eestikeelset vastet, sest eesti keeles on lubatud muuta sõnade ja fraaside järjekorda:

```
?-tõlgi(Est,[the,good,young,man,carries,the,stones]).
Est = [tubli, noor, mees, kannab, kive] ;
Est = [tubli, noor, mees, kive, kannab] ;
Est = [kannab, kive, tubli, noor, mees] ;
Est = [kive, kannab, tubli, noor, mees] ;
no
```

Üllataval kombel töötab tõlkeprogramm ka eesti- ja ingliskeelsete lausete tõlkepaaride generaatorina:

```
?-tõlgi(Est,Eng).
Est = [noor, noor, mees, kannab, kive]
Eng = [young, young, man, carries, stones] ;
...
```

Ülesanne 125. Muutke grammatikat nii, et saaksime sama grammatikaga tõlkida lausedd “mees kannab” ja “mees kannab kive”.

Ülesanne 126. Kirjutage tõlkimisprogramm lause jaoks “kena sale tüdruk kohendab rahulikult juukseid”.

8.6. Arvsõnade analüüsimine

Horni grammatika standardse näitena toome grammatika arvude 0–999 ingliskeelsete nimetuste süntaksianalüüsiks joonisel 8.1 (vt. lk. 99). Lisaks arvsõnaväljendi analüüsimisele annab grammatika välja ka väljendi arvulise väärtuse. Näiteks

```
?-phrase(number(N),[twenty,four]).
N = 24
```

Grammatika töötab ka tagurpidi, väljastades arvule vastava väljendi:

```
?-phrase(number(24),Xs).
Xs = [twenty, four]
```

Samuti töötab grammatika arvude ja nendele vastavate väljendite generaatorina:

```
?-phrase(number(N),Xs).
N = 0    Xs = [zero] ;
N = 100  Xs = [one, hundred] ;
N = 101  Xs = [one, hundred, and, one] ;
N = 102  Xs = [one, hundred, and, two] ;
...
```

Ülesanne 127. Kirjutage eestikeelsete arvsõnaväljendite grammatika arvudele 0–999, mis leiab nii arvule vastava väljendi kui ka väljendile vastava arvu.

8.7. Grammatika väärkasutused

Programmeerijad kasutavad meelsasti 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) --> [].
```

See predikaat analüüsib tegelikult 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]
```

```
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 8.1. Ingliskeelsete arvude Horni grammatika.

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 seda saab sooritada, ilma et analüüsitaks ühtegi listi elementi.

Ülesanne 128. (M. Covington) Kirjutage programm, mis analüüsib grammatikareeglite abil läbi kõik listi elemendid ja leiab nende summa.

Ülesanne 129. (M. Covington) Kas `dog --> "dog"` on korrektne grammatikareegel? Kui on, siis mida see tähendab ja kuidas seda kasutada?

Ülesanne 130. (M. Covington) Kirjutage Horni grammatika, mis jaotab tähtede listi sõnadeks. Näiteks teisendab stringi `"this is it"` kujule `[this,is,it]`. Tähtede listi sõnadeks jaotamist võib vaadelda kui teatud liiki süntaktilist analüüsi.

8.8. Prologi grammatika

```
programm            --> programmielemendid.
programmielemendid  --> [] ;
                        programmielement, programmielemendid.
programmielement    --> kirjeldus ; päring.
kirjeldus           --> fakt ; reegel ; grammatikareegel.
fakt                --> atomaarne_predikaat, ['.'].
reegel              --> atomaarne_predikaat, [':-'],
                        alternatiivid, ['.'].
päring              --> ['?-'], alternatiivid, ['.'].
päring              --> [':-'], alternatiivid, ['.'].
alternatiivid       --> konditsionaal.
alternatiivid       --> konditsionaal, [';'], alternatiivid.
alternatiivid       --> konditsionaal, ['|'], alternatiivid.
alternatiivid       --> ['('], alternatiivid, [')'].
konditsionaal       --> literaalid.
konditsionaal       --> literaalid, ['->'], konditsionaal.
literaalid          --> literaal.
literaalid          --> literaal, [','], literaalid.
```

```
literaal            --> atomaarne_predikaat.
literaal            --> ['\+'], atomaarne_predikaat.
atomaarne_predikaat --> predikaadinimi, argumendid.
atomaarne_predikaat --> term, võrdlus, term.
atomaarne_predikaat --> ['!'] ; [true] ; [fail] ; [repeat].
predikaadinimi      --> aatom.
argumendid          --> [] ; ['('], termid, [')'].
termid              --> term ; term, [','], termid.
term                --> ['('], termid, [')'].
term                --> ['{'], termid, ['}'].
term                --> konstant ; struktuur ; muutuja ;
                        list ; tekst.
konstant            --> aatom ; arv ; ['[',']'].
struktuur           --> funktsiooninimi, argumendid.
struktuur           --> term, tehe, term.
funktsiooninimi     --> aatom.
list                --> ['[',']'] ; ['['], termid, [']'].
list                --> ['['], termid, ['|'], termid, [']'].
tekst               --> ['"'], sümbolid, ['"'].

aatom               --> väiketäht, tähed.
aatom               --> kirjavahemärgid.
aatom               --> [''''], sümbolid, [''''].
kirjavahemärgid     --> kirjavahemärk.
kirjavahemärgid     --> kirjavahemärk, kirjavahemärgid.
sümbolid            --> [] ; sümbol, sümbolid.
sümbol              --> [W], { on_ASCII_sümbol(W) }.
arv                 --> märk, numbrid, punkt, tähed.
numbrid             --> number ; number, numbrid.
muutuja             --> (suurtäht ; ['_']), tähed.
tähed               --> [] ; täht, tähed.
täht                --> väiketäht ; suurtäht ; number ; ['_'].

väiketäht           --> ['a'] ; ['b'] ; ['c'] ; ... ; ['z'].
väiketäht           --> ['õ'] ; ['ä'] ; ['ö'] ; ['ü'].
suurtäht            --> ['A'] ; ['B'] : ['C'] ; ... ; ['Z'].
suurtäht            --> ['Õ'] ; ['Ä'] ; ['Ö'] ; ['Ü'].
number              --> ['0'] ; ['1'] ; ['2'] ; ... ; ['9'].
märk                --> [] ; ['+'] ; ['-'].
punkt               --> [] ; ['.'].
võrdlus             --> ['='] ; ['\='] ; ['=='] ; ['\=='].
võrdlus             --> ['is'] ; ['=:='] ; ['=\='].
võrdlus             --> ['<'] ; ['=<'] ; ['>'] ; ['>='].
tehe                --> ['+'] ; ['-'] ; ['*'] ; ['/'] ; ['//'].
tehe                --> ['mod'] ; ['**'] ; ['^'].
```

```
kirjavahemärk       --> ['+'] ; ['-'] ; ['*'] ; ['/'] ; [':'].
kirjavahemärk       --> ['<'] ; ['='] ; ['>'] ; ['#'] ; ['$'].
kirjavahemärk       --> ['@'] ; ['&'] ; ['^'] ; ['~'] ; ['`'].
kirjavahemärk       --> ['\'] ; ['.'] ; ['?'].

grammatikareegel    --> reegli_päis, ['-->'], reegli_sisud.
reegli_päis         --> mitteterminaal.
reegli_päis         --> mitteterminaal, [','], terminaal.
reegli_sisud        --> reegli_sisu.
reegli_sisud        --> reegli_sisu, [';'], reegli_sisud.
reegli_sisud        --> reegli_sisu, ['|'], reegli_sisud.
reegli_sisu         --> reegli_elemendid.
reegli_sisu         --> reegli_elemendid, ['->'],
                        reegli_elemendid.
reegli_elemendid    --> reegli_element.
reegli_elemendid    --> reegli_element, [','], reegli_elemendid.
reegli_element      --> muutuja.
reegli_element      --> mitteterminaal.
reegli_element      --> terminaal.
reegli_element      --> ['{'], alternatiivid, ['}'].
reegli_element      --> ['!'].
reegli_element      --> ['('], reegli_sisud, [')'].

mitteterminaal      --> aatom, argumendid.
terminaal           --> list.
```

Tühikut ja mittetrükitavaid sümboleid (tabulatsioon, reavahetus) ei tohi kirjutada predikaadi- või funktsiooninime ja selle argumentide sulu vahele. Kommentaare % ja `/*...*/` võib kirjutada programmis samadesse kohtadesse, kuhu tohib kirjutada tühikut.

9. peatükk

Loogilise programmeerimise laiendused

Hulkade kasutamine otsingustrateegias. Interaktiivsed programmid. Teadmiste tuletamine andmetest.

9.1. Tabelimeetod

Prologi interpretaator jääb hätta mõningate päringute täitmisega, sest kasutab süvitiotsingut. Probleeme tekitavad muuhulgas relatsioonide sellised omadused nagu sümmeetria ja transitiivsus, mille kontrollimine võib jääda lõpmatusse tsüklisse. Oletame näiteks, et predikaat *ühendatud*/2 on sümmeetriline, kusjuures *a* olgu ühendatud *b*-ga:

```
ühendatud(X,Y):-ühendatud(Y,X).
ühendatud(a,b).
```

Tavaline interpretaator jääb hätta päringuga `?-ühendatud(a,X)`:

```
?-ühendatud(X,a).
?-ühendatud(a,X).
...
```

Tulemuseks on lõpmatu päringute jada, milles *a* ja *X* vahetavad pidevalt kohti.

Oletame nüüd, et predikaat *ühendatud*/2 on transitiivne, kusjuures *a* olgu ühendatud *b*-ga:

```
ühendatud(X,Y):-ühendatud(X,Z), ühendatud(Z,Y).
ühendatud(a,b).
```

Ka nüüd saame päringuga `?-ühendatud(a,X)` lõpmatu tsükli, sest muutuja *Z* number pidevalt suureneb:

```
?-ühendatud(a,Z1).
?-ühendatud(a,Z2).
?-ühendatud(a,Z3).
...
```

Selle vältimiseks kasutavad B-Prolog ja XSB-Prolog **tabelimeetodit** ehk puhverdamist, mis salvestatab teatud predikaatide päringud ja nende vastused tabelisse. Kui mõnda salvestatud päringut püütakse korrata, siis jäetakse päring täitmata ning loetakse vastus või selle puudumine tabelist.

Tabelimeetodiga saame sümmeetrilise predikaadi *ühendatud*/2 korral ülaltoodud päringule kohe vastuse:

```
:- table ühendatud/2.
ühendatud(X,Y):-ühendatud(Y,X).
ühendatud(a,b).
?-ühendatud(a,X).
X=b
```

Esmalt salvestatakse tabelisse päring `?-ühendatud(a,X)` ja seejärel alampäring `?-ühendatud(X,a)`. Järgmise alampäringu `?-ühendatud(a,X)` täitmisel selgub tabelist, et seda päringut püüti juba täita, kuid ilma tulemuseta. Seejärel katkestatakse lõpmatu rekursioon ja saadakse sümmeetriareeglile järgneva fakti *ühendatud(a,b)* toimel vastus $X = b$.

Transitiivse predikaadi *ühendatud*/2 korral saame samuti vastuse:

```
:- table ühendatud/2.
ühendatud(X,Y):-ühendatud(X,Z), ühendatud(Z,Y).
ühendatud(a,b).
?-ühendatud(a,X).
X=b
```

Esmalt salvestatakse tabelisse päring `?-ühendatud(a,X)`. Selle täitmisel saadav alampäring `?-ühendatud(a,Z1)` langeb muutujate ümbernimetamise täpsusega kokku salvestatud päringuga `?-ühendatud(a,X)`, millel tabelis puudub tulemus. Lõpmatu haru täitmine katkestatakse ja transitiivsusreeglile järgneva faktiga *ühendatud(a,b)* saadakse uus päring `?-ühendatud(b,X)`. Viimane annab alampäringu `?-ühendatud(b,Z2)`, mis on ekvivalentne salvestatud päringuga. Lõpmatu haru täitmine katkestatakse ja transitiivsusreeglile järgneva faktiga *ühendatud(a,b)* saadakse vastus $X = b$.

Ülesanne 131. Kirjutage metainterpretaator, mis arvutab päringu tabelimeetodil.

9.2. Kitsendustega loogiline programmeerimine

Lisaks aritmeetiliste avaldiste väärtuste arvutamisele saab mitmes Prologi-süsteemis kasutada ka erinevaid võrrandite lahendamise meetodeid, näiteks lineaarsete võrrandite süsteemi lahendamist elimineerimismeetodil ja optimaalse lahendi leidmist simpleksmeetodil.

Kitsendustega loogilise programmeerimise näitena kirjutame punktis 4.3 esitatud arvulise listi elementide summeerimise predikaadi ümber, kasutades liitmise asemel lahutamist ja võttes tühilisti elementide summaks nulli:[29]

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

Kahjuks ei anna päring `?-sum([1,2,3],Sum)` soovitud tulemust, sest predikaadi *is* täitmisel on muutuja *Sum* väärtustamata, põhjustades aritmeetikavea:

```
?-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 ?
```

Uuemad loogilise programmeerimise süsteemid nagu ECLiPSe, SICStus Prolog ja GNU Prolog võimaldavad kasutada kitsendustega programmeerimise meetodeid (ingl. k. *constraint programming*), mistõttu need saavad hakkama ka lihtsamate sümbolvõrrandite lahendamisega. ECLiPSe'is piisab lõplike doomenite (ingl. k. *finite domains*) sisselülitamisest ja vastavate '#'-võrdluste kasutamisest:

```
:-lib(fd).
sum([],Sum):-0 #= Sum.
```

Pärast neid täiustusi hakkab programm *sum* ilusti tööle:

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

Lahendame ECLiPSe'is krüptoaritmeetilise ülesande, milles erinevatele tähtedele vastavad erinevad numbrid ja sõna esimene täht peab olema nullist erinev:

$$\begin{array}{r} SEND \\ +MORE \\ \hline MONEY \end{array}$$

[29] Summa arvutamise lahutamise abil pakkus loengu käigus välja Oleg Mürk.

Lõplike doomenite meetodi ideeks on seada igale muutujale vastavusse lõplik väärtuste piirkond, millest arvutuste käigus kustutatakse kõik väärtused, mida muutuja ei saa kitsenduste põhjal omada. Kitsendusteks võivad olla nii võrdlused ja võrratused kui ka doomenipredikaadid. Näiteks võttes muutuja *X* doomeniks arvud 1—4 ning keelates selles kahest väiksemad arvud ja arvu 3, saame tulemuseks arvuhulga {2, 4}:

```
:-lib(fd).
?-X::1..4, X#>=2, X#\=3.
X = X[2, 4]
yes
```

Toodud krüptoaritmeetilise ülesande lahendamiseks loeme sisse lõplike doomenite teegi ja määrame predikaadiga *solve* tähtede vahelised seosed ja kitsendused:

```
:-lib(fd).
solve([S,E,N,D,M,O,R,Y]):-S #\= 0, M #\= 0,
                1000* S +100* E +10* N + D
               +1000* M +100* O +10* R + E
  #= 10000* M +1000* O +100* N +10* E + Y.
```

Predikaadi *puzzle* kirjeldus ütleb, et tema argumendiks on list numbritest 0—9, mille elemendid on paarikaupa erinevad ning mis rahuldab eespool kirjeldatud predikaati *solve*:

```
puzzle(Xs):-Xs::0..9, alldifferent(Xs), solve(Xs),
            labeling(Xs).
```

Süsteemne predikaat *labeling* genereerib listi iga muutuja jaoks ühekaupa doomeni kõik väärtused. See on vajalik juhul, kui ülesandel on mitu vastust, või kui eelnevatest kitsendustest ei piisa muutujate doomenite ühestamiseks. Antud juhul selgub siiski, et ülesandel on ühene vastus:

```
?-puzzle([S,E,N,D,M,O,R,Y]).
S = 9  E = 5  N = 6  D = 7
M = 1  O = 0  R = 8  Y = 2
more (0.01s cpu) ? ;
no (0.01s cpu)
```

Lõplike doomenite meetodi efektiivsuses veendume siis, kui kasutame sama ülesande lahendamiseks harilikke muutujaid: genereerime numbrite 0—9 kõik permutatsioonid ja omistame saadud numbritejärjendid ülesande kaheksale tähele:

```
?-Xs=[0,1,2,3,4,5,6,7,8,9], permutation(Xs,[_,_|Ys]),
  solve(Ys).
Xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Ys = [9, 5, 6, 7, 1, 0, 8, 2]
more (9.62s cpu) ? ;
Xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Ys = [9, 5, 6, 7, 1, 0, 8, 2]
more (12.34s cpu) ? ;
no (27.47s cpu)
```

Nagu lahendusaegadest näha, saadakse seekord tulemus tuhandeid kordi aeglasemalt.

Lisaks lõplike doomenite ja lineaarsete võrrandite lahendamise tehnikatele võimaldavad kitsendustega programmeerimise keeled määrata ka muutujate ja väärtuste valiku järjekorrad muutujate väärtuste genereerimisel. Tavaliselt valitakse väärtustamiseks muutuja, mis on seotud suurima arvu kitsendustega, väärtuse valik sõltub aga konkreetsest ülesandest. Näiteks ristsõna koostamisel püütakse esmalt valida võõrtähti sisaldavad sõnad, minimeerides niiviisi doomenite suurusi. Lisaks on kitsendustega programmeerimise keeltes vahendid minimaalse vastuse leidmiseks, mis on vajalik optimaalsete plaanide koostamiseks. Näiteks süsteemne predikaat *minimize* leiab antud predikaadi tõesuspiirkonnast antud termi järgi minimaalse vastuse:

```
?-lib(fd), X::1..10, minimize(indomain(X),X).
X :: 1..10, minimize(indomain(X), X).
Found a solution with cost 1
X = 1
```

Ülesanne 132. Kirjutage programm, mis lahendab järgmise krüptoaritmeetilise ülesande:

$$\begin{array}{r} \ddot{U}\,K\,S \\ +\,\ddot{U}\,K\,S \\ +\,\ddot{U}\,K\,S \\ +\,\ddot{U}\,K\,S \\ \hline N\,E\,L\,I \end{array}$$

Täiendage programmi, et leida ülesande kõigi vastuste arvu.

Ülesanne 133. Kirjutage programm, mis koostab etteantud sõnadest etteantud mõõtudega ristsõna, kasutades lõplike doomenite meetodit (vt. punkt 6.2, aga samuti ka ül. 77 ja 78).

9.3. Abduktiivne loogiline programmeerimine

Kui tavaliselt kasutatakse reegleid vasakult paremale, s.t. eelduste tõesuse põhjal saadakse järelduse tõesus, siis Peirce'i poolt kirjeldatud **abduktiivne meetod** annab järelduse tõesuse ja reegli põhjal selle eelduse. Näiteks, kui muru on märg, siis võime teha abduktiivse järelduse, et sajab vihma, sest vihmasadu muudab muru märjaks. Samuti võib teha abduktiivse järelduse, et muru kasteti, sest muru kastmisel saab see samuti märjaks.

Abduktiivseid järeldusi tehakse elus sageli. Näiteks ennustame abduktsiooniga ilma. Võtame aluseks vanasõnad "Kui Mart külmetab, siis Kadri sulatab" ja "Kui Mart on must, siis Kadri on valge". Realiseerime need vanasõnad reeglitena:

```
soojus(kadri,soe):-soojus(mart,külm).
maa(kadri,valge):-maa(mart,must).
```

Püüame leida nende reeglite põhjal vastuse küsimustele "Kas kadripäeval on lumi maas?" ja "Milline on maa kadripäeval?" Selleks kirjutame predikaadi *ennusta*/1, mis püüab leida reeglite põhjal neile küsimustele vastuse, esitades vajadusel täpsustavaid lisaküsimusi:

```
ennusta(G):-solve(G), !.
ennusta(G):-
  write('Kahjuks ei oska vastata.'), nl,
  write('Palun täiendage teadmistebaasi.'), nl,
  write('50% tõenäosusega on '), write(G), nl.
```

Ennustaja sisemiseks mootoriks on küsiv metainterpretaator *solve*, mis esitab teatud tingimustel lisaküsimusi ning kasutab vaikimisiteadmusi:

```
:-op(700,xfx,if).
solve(true):-!.
solve((A,B)):-!, solve(A), solve(B).
solve(A):-askable(A), ask(A), read(jah), nl.
solve(A):-clause(A,B), solve(B), !.
solve(A):-default(A if B), solve(B), !.
```

Lisaküsimuseks võib olla iga predikaat, milles ei ole vabu muutujaid:

```
askable(X):-ground(X).
```

Dialoogi ilmestamiseks muudame osa lisaküsimusi suupärasemaks:

```
ask(maa(mart,must)):-
  write('Kas mardipäeval oli maa must'), !.
ask(maa(mart,valge)):-
  write('Kas mardipäeval oli maa valge'), !.
ask(maa(kadri,must)):-
  write('Kas kadripäeval oli maa must'), !.
ask(maa(kadri,valge)):-
  write('Kas kadripäeval oli maa valge'), !.
ask(soojus(mart,külm)):-
  write('Kas mardipäeval oli külm'), !.
ask(soojus(mart,soe)):-
  write('Kas mardipäeval oli soe'), !.
ask(soojus(kadri,külm)):-
  write('Kas kadripäeval oli külm'), !.
ask(soojus(kadri,soe)):-
  write('Kas kadripäeval oli soe'), !.
ask(A):-write(A).
```

Lisame programmile ka vaikimisiteadmuse “Kui on soe, siis ei ole lund”:

```
default(maa(X,must) if soojus(X,soe)).
```

Dialoog “ilmatargaga” näeb nüüd välja järgmiselt:

```
?-ennusta(maa(kadri,valge)).
Kas kadripäeval oli maa valge?- eitea.
Kas mardipäeval oli maa must?- ei.
Kas mardipäeval oli soe?- ei.
Kahjuks ei oska vastata.
Palun täiendage teadmistebaasi.
50% tõenäosusega on maa(kadri, valge)
yes
?-ennusta(maa(kadri,Maa)).
Kas mardipäeval oli maa must?- ei.
Kas mardipäeval oli soe?- ei.
Kas kadripäeval oli soe?- eitea.
Kas mardipäeval oli külm?- jah.
Maa = must
```

Ülesanne 134. Täiustage ilmaennustamise programmi, lisades sellele järgmised rahvatarkusest pärinevad reeglid:

a) Kui näärikuul sula teeb, siis on küünlakuu külm.
b) Soe küünlakuu, külm kevade.
c) Kui hani mardipäeva ajal jää peale astub, siis astub ta jõulu ajal sopa sisse.
d) Mis Kadrid lahti sulatavad, seda tinutab Andres kinni.
e) Kuidas tali, nõnda suvi.
f) Mida külmem jõulu, seda palavam Jaan.

9.4. Induktiivne loogiline programmeerimine

Induktiivse loogilise programmeerimise eesmärgiks on koostada etteantud sisendi ja väljundi paaridele vastav faktidest ja reeglitest koosnev programm. Selleks saab kasutada Milli eksperimentaalseid induktsioonimeetodeid. Uurime mingi nähtuse *p* ilmnemise põhjuseid, valides nende kirjeldamiseks välja mingi hulga tingimusi. Nähtuseks võib näiteks olla vee jäätumine ja valitud tingimusteks temperatuuri langemine alla nulli, soola lisamine veele, õhurõhu tõstmine jms. Sooritame teatud arvu eksperimente ja moodustame nende põhjal tabeli, mille read vastavad sooritatud eksperimentidele ja veerud näitavad, kas antud tingimus oli täidetud või kas vaadeldav nähtus ilmnes.

Milli **ühildumis- ja erinevusmeetod** väidab, et kui mingi tingimus on täidetud eksperimentides, kus nähtus *p* ilmneb, ja pole täidetud eksperimentides, kus nähtus ei ilmne, siis see tingimus ongi nähtuse *p* põhjuseks. Järgmises kuues eksperimendis, kus välja on valitud kolm tingimust *A*, *B* ja *C*, vastab nendele kriteeriumitele ainult tingimus *A*:

Eksperiment	Tingimused			Nähtus
1	A	B	C	p
2	A		C	p
3	A	B		p
4			C	
5		B		
6		B	C	

Järelikult Milli meetod annab meile reegli $A \rightarrow p$ ehk Prologi kujul

```
p:-'A'.
```

Reegleid arvutava programmi kirjutamiseks esitame eksperimentide tulemuste tabeli predikaadina *eksperiment*/4, mille esimene argument vastab tingimusele *A*, teine tingimusele *B*, kolmas tingimusele *C* ja neljas nähtusele *p*. Plussmärk tähendab tingimuse või nähtuse esinemist ja miinusmärk selle mitte esinemist:

```
eksperiment(+,+,+,+).
eksperiment(+,-,+,+).
eksperiment(+,+,-,+).
eksperiment(-,-,+,-).
eksperiment(-,+,-,-).
eksperiment(-,+,+,-).
```

Reegli $A \rightarrow p$ kehtivust kontrollime predikaadiga *reegel_A*, mis leiab esmalt eksperimentide arvu, milles on täidetud tingimus *A*, seejärel eksperimentide arvu, milles ilmneb nähtus *p*, ja lõpuks eksperimentide arvu, milles on täidetud tingimus *A* ja ilmneb nähtus *p*. Juhul, kui saadud arvud on võrdsed, salvestame kontrollitava reegli dünaamilisse andmebaasi:

```
reegel_A:-
  findall(A,(eksperiment(A,_,_,_),A='+'),As),
  length(As,N),
  findall(P,(eksperiment(_,_,_,P),P='+'),Ps),
  length(Ps,N),
  findall(A+P,(eksperiment(A,_,_,P),A='+',P='+'),APs),
  length(APs,N),
  assert(p:-'A').
```

Käsu *reegel_A* täitmine annab tulemuseks reegli `p:-'A'`:

```
?-reegel_A.
yes
```

Ülesanne 135. Kirjutage predikaadid, mis kontrollivad predikaadi *eksperiment*/4 põhjal reeglite $B \rightarrow p$ ja $C \rightarrow p$ kehtivust.

Tänapäeval kasutatakse reeglite moodustamiseks enam statistilisi meetodeid. **Assotsiatsioonireeglite** korral lubatakse ka reegleid, mis on etteantud täpsuse ja usalduslävega. Olgu eksperimentideks mingis kaupluses sooritatud ostud ja vastaku iga tingimus mingi kauba sisaldumisele konkreetses ostukorvis. Eesmärgiks on selgitada välja, millised kaubad võiksid asuda kaupluses lähestiku — ostja ostab tavaliselt poest mitut kaupa, kuid ta ei viitsi läbi käia poe kõiki soppe. Näiteks lapsemähkmete ostja ostab tavaliselt ka lapsetoitu ja veini ostja ostab tavaliselt juustu. Reegli eelduste järjekord ei ole oluline, kuid järeldumine ei ole sümmeetriline. Näiteks juustu ostja ei tarvitse osta veini.

Reegli $p \rightarrow q$ **täpsuse** t ja **usaldusläve** u arvutame valemitega

$$t = \frac{l}{m} \qquad \text{ja} \qquad u = \frac{m}{n},$$

kus n on kõigi ostukorvide arv, m on nende ostukorvide arv, kus sisaldub p, ja l on nende ostukorvide arv, kus sisalduvad p ja q (allikas: [8]).

Märgime mingi kauba sisaldumist ostukorvis plussmärgiga ja mittesisaldumist miinusmärgiga. Vaatame ostukorvide tabelit, mille veerud näitavad piima, leiva ja õlle sisaldumist konkreetsetes ostukorvides:

Ostukorv	Piim	Leib	Õlu
1	+	−	+
2	−	+	+
3	+	+	+

Antud tabeli korral saame reegli *piim* → *leib* jaoks täpsuse ja usaldusläve

$$t = \frac{1}{2} \quad \text{(ehk 50\%)}, \qquad u = \frac{2}{3} \quad \text{(ehk 66\%)}.$$

Reeglite *õlu* → *piim* ja *õlu* → *leib* jaoks saame

$$t = \frac{2}{3} \quad \text{(ehk 66\%)}, \qquad u = \frac{3}{3} \quad \text{(ehk 100\%)}$$

ja kahe eeldusega reegli *leib,õlu* → *piim* jaoks

$$t = \frac{1}{2} \quad \text{(ehk 50\%)}, \qquad u = \frac{2}{3} \quad \text{(ehk 66\%)}.$$

Ülesanne 136. Kirjutage programm, mis antud tabeli põhjal leiab kõik etteantavat täpsust ja usaldusläve ületavad assotsiatsioonireeglid. Näiteks täpsuse 60% ja usaldusläve 80% korral väljastatakse ostukorvide toodud tabeli põhjal reeglid *õlu* → *piim* ja *õlu* → *leib*.

Ülesannete lahendused

Ülesanne 1.

```
lind(hani).
lind(part).
lendab(X):-lind(X).
```

Ülesanne 2. 1. `?-lendab(part).` 2. `?-lendab(X).`

Ülesanne 5 (vihje). Kasutage eitust kujul $X\backslash{=}Y$.

Ülesanne 5. Keegi on kellegi vend, kui ta on sama ema laps, ta on meessoost ning ta erineb esialgsest isikust:

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

Ülesanne 7 (vihje). Kasutage kirjeldusi kujul

```
aadress(anne,60-34).
```

Ülesanne 9. `?-write('Tere').`

Ülesanne 11. `:-op(900,xfx,ema_on).`

Ülesanne 12.

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

Ülesanne 14 (vihje). Viimane päring ei tööta, sest suurtähte 'Õ' ei käsitleta muutuja esitähena.

Ülesanne 17 (vihje). Uurige Prologi sisemiste loogikatehete kirjeldusi.

Ülesanne 17.

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

Ülesanne 22 (vihje).

```
?-diff(2*x,x,D).
?-diff(sin(x),x,D).
```

Ülesanne 28 (vihje). Kasutage predikaate *write*/1 (väljastamine), *read*/1 (lugemine) ja *nl*/0 (reavahetus).

Ülesanne 29. `even(X):-X mod 2 =:= 0.`

Ülesanne 31.

```
sum(N,S):-sum(N,0,S).

sum(0,S,S).
sum(N,Acc,S):-
    N2 is N-1,
    Acc2 is Acc+N*N,
    sum(N2,Acc2,S).
```

Ülesanne 32 (vihje). Reeglid:[30]

1. Viielisi ei tohi lahutada. Õige on näiteks XCV, mitte VC.
2. Kaks lahutamist ei tohi olla järjest. Näiteks XCVIII, mitte IIC.
3. Ei tohi olla neli samasugust tähte järjest: IIII, XXXX jne.
4. Ükski viieline ei tohi esineda arvus rohkem kui üks kord. Näiteks LIX ja LV, kuid mitte LVIV.
5. Ei tohi lahutada ja liita samaaegselt: $5 \neq$ IVI (4 ja 1). $100 \neq$ ICI, XCX jne.

Ülesanne 33. $a \neq f(a), a \neq b$.

Ülesanne 42. `member(X,Xs):-append(_,[X|_],Xs).`

Ülesanne 43.

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

Ülesanne 52 (vihje). Läbige list vasakult paremale, salvestades akumulaatorisse elemendid koos loendaja väärtusega. Loendaja suurendamiseks tehke parandus akumulaatori vastavas liikmes.

[30] Reegleid aitasid sõnastada Ülo Kaasik ja Dmitri Lepp.

Ülesanne 54.

```
% ordered(Xs) on tõene siis,
%       kui arvude list Xs on järjestatud mittekahanevalt
ordered([]).
ordered([_]).
ordered([X,Y|Ys]):-
    X=<Y,
    ordered([Y|Ys]).
```

Ülesanne 58 (vihje). Kasutage süsteemset predikaati *atom_chars*/2, mis annab aatomile vastava tähtede listi:

```
  ?-atom_chars(külli,[k,ü,l,l,i]).
```

Ülesanne 69.

```
color(1).  color(2).  color(3).

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):-
    \+ (
    member(color(X,CX),S), member(color(Y,CY),S),
    X\=Y,
    neighbour(X,Y),
    CX=CY).

?-coloring(S).
```

Ülesanne 71 (vihje). Anumatega saab sooritada kolme operatsiooni: anumat täita, anumat tühjendada ja kallata ühe anuma sisu teise anumasse. Probleemi lahendamiseks leidke nendele operatsioonidele vastav seisude järjend:

$$\begin{gathered} \mathit{seis}(\mathit{seitse}(0), \mathit{viis}(0)), \\ \mathit{seis}(\mathit{seitse}(7), \mathit{viis}(0)), \\ \ldots \end{gathered}$$

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 71.

```
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 72.

```
struktuur([sõber(N1,L1),sõber(N2,L2)]):-
    L1='Tartu',
    L2='Pärnu'.

linn('Tartu').
linn('Pärnu').

nimi(sõber(Nimi,_),Nimi).
linn(sõber(_,Linn),Linn):-linn(Linn).

mõistatus:-
    struktuur(Sõbrad),
    member(Mees1,Sõbrad),
    nimi(Mees1,'Tõnu'),
```

```
    linn(Mees1,'Tartu'),
    member(Mees2,Sõbrad),
    nimi(Mees2,'Priit'),
    linn(Mees2,Linn),
    write('Priit elab: '), write(Linn), write('s'), nl.
?-mõistatus.
```

Ülesanne 77 (vihje). Korduste vältimiseks pidage listi kujul meeles kõik ruudustikku kantud sõnad.

Ülesanne 80 (vihje). Term X on vaba muutuja, kui järgmisel päringul on kaks vastust:

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

Ülesanne 80.

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

Ülesanne 81.

```
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 82 (vihje).

```
?-relative(2,alice,X).
  X = bob ;
  X = alice ;
  X = carol ;
  X = john ;
  X = bob ;
  X = alice ;
  X = carol ;
  X = ann ;
  X = peter ;
  X = mary ;
  no
```

Ülesanne 83 (vihje).

```
?-relative(2,alice,X,Def).
  X = alice  Def = on-alice ;
  X = mary   Def = ema-on-mary ;
  X = bob    Def = ema-poeg-on-bob ;
  X = alice  Def = ema-tütar-on-alice ;
  X = carol  Def = ema-tütar-on-carol ;
  X = john   Def = ema-mees-on-john ;
```

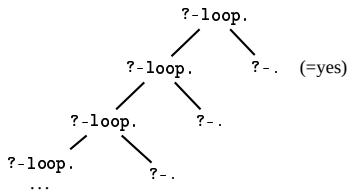
```
  X = john   Def = isa-on-john ;
  X = bob    Def = isa-poeg-on-bob ;
  X = alice  Def = isa-tütar-on-alice ;
  X = carol  Def = isa-tütar-on-carol ;
  X = ann    Def = isa-ema-on-ann ;
  X = peter  Def = isa-isa-on-peter ;
  X = mary   Def = isa-naine-on-mary ;
  no
```

Ülesanne 84 (vihje).

```
:-op(500,xfy,-).

relative-definite(N,X,Y,Label):-
    findall(X-Def,relative(N,X,_,Def),List),
    remove_duplicate_labels(List,List2),
    member(Label,List2),
    member_name(Y,Label).
label_name(_-Y,Z):-
  !,
  label_name(Y,Z).
label_name(X,X).
?-member_name(john,[alice-ema-mees-on-john,alice-isa-on-john]).
```

Ülesanne 85.



Vastust ei leita, sest otsingupuu vasakpoolne haru on lõpmatu.

Ülesanne 86 (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 nimetatakse järjekorraks (ingl. k. *queue*) või FIFO-magasiniks (ingl. k. *first-in-first-out*).)

Ülesanne 88.

```
is_tree(leaf).
is_tree(tree(E,L,R)):-is_tree(L), is_tree(R).
```

Ülesanne 90.

```
tree_to_list(tree(X,L,R),Xs):-
    tree_to_list(L,Ls), tree_to_list(R,Rs), append([X|Ls],Rs,Xs).
tree_to_list(leaf,[]).
```

Ülesanne 98.

```
kääna(X,Tüüp,Y):-
  sõna_tüvi(X,Xs,_),
  vorm(Tüüp,_+Ys),
  atom_concat(Xs,Ys,Y).

sõna_tüvi(X,X1,Y1):-
  atom_chars(X,Cs),
  Ys=[_,_],
  append(Xs,Ys,Cs),
  atom_chars(X1,Xs),
  atom_chars(Y1,Ys).

vorm(naine,nai+ne).
vorm(naine,nai+se).
vorm(naine,nai+st).
vorm(naine,nai+sesse).
vorm(naine,nai+sed).
vorm(naine,nai+ste).
vorm(naine,nai+si).
vorm(naine,nai+stesse).
```

Ülesanne 102 (vihje).
palun = ole hea, ole meheks
minu = mu
telefon = kõnetraat, mobiil
aitäh = tänan

Ülesanne 110 (vihje).
Kas lasteaias on suuri lapsi ka?
Sa pabistad vist natuke lasteaeda mineku pärast.
Suured poisid võivad mind lüüa.
Sa kardad haiget saada.
Jah, ja nad ei mängi minuga.
Sa tunned end üksildasena, kui nad sinuga ei mängi.

Ülesanne 117.

```
ab-->[].
ab-->a,ab,b.
a-->[a].
b-->[b].
```

Ülesanne 120 (vihje). Kasutage lisaargumentide unifitseerimist ja Peano aritmeetikat.

Ülesanne 120.

```
abc-->a(N),b(N),c(N).
a(0)-->[].
a(s(N))-->[a], a(N).
b(0)-->[].
b(s(N))-->[b], b(N).
c(0)-->[].
c(s(N))-->[c], c(N).
```

Ülesanne 122.

```
?-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 124.

```
lause(Sem) --> noomenifraas(A), verbifraas(Sem,A).

noomenifraas(A) --> omadussõna, omadussõna, nimisõna(A).
verbifraas(Q,A) --> tegusõna(Q,A,S), nimisõna2(S).

nimisõna(mees) --> [mees].
nimisõna2(kive) --> [kive].
tegusõna(kannab(X,Y),X,Y) --> [kannab].
omadussõna --> [noor] ; [tubli].
```

Ülesanne 132 (vihje). Kasutage vastuste loendamiseks dünaamilisi predikaate, millega saab luua globaalse sideme erinevate otsinguharude vahel:

```
?-asserta(arv(0)).
?-arv(N).
?-retract(arv(N)).
```

Kirjandus

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

[2] P. Blackburn, J. Bos, K. Striegnitz. *Learn Prolog now*!, 2001. [http://www.coli.uni-sb.de/~kris/learn-prolog-now/]

[3] 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]

[4] I. Bratko. *Prolog programming for artificial intelligence.* 3rd edition. Addison-Wesley, 2001.

[5] W. F. Clocksin, C. S. Mellish. *Programming in Prolog.* 3rd edition. Springer-Verlag, 1987.

[6] I. M. Copi, K. Burgess-Jackson. *Informal logic.* Prentice Hall, 1996.

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

[8] D. Hand, H. Mannila, P. Smyth. *Principles of data mining.* MIT Press, 2001.

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

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

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

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

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

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

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

[16] J. W. Lloyd. *Foundations of logic programming.* 2nd edition. Springer-Verlag, 1987.

[17] K. Marriott, P. J. Stuckey. *Programming with constraints: an introduction.* MIT Press, 1998.

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

[19] D. Poole, A. Mackworth, R. Goebel. *Computational intelligence: a logical approach.* Oxford University Press, 1998.

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

[21] L. Sterling, E. Shapiro. *The art of Prolog: advanced programming techniques.* MIT Press, 1986.

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

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

Aineregister