ITI0211 - Loogiline programmeerimine
Loeng 5: Loogilise programmi täitmise juhtimine

J.Vain

Sügis 2021

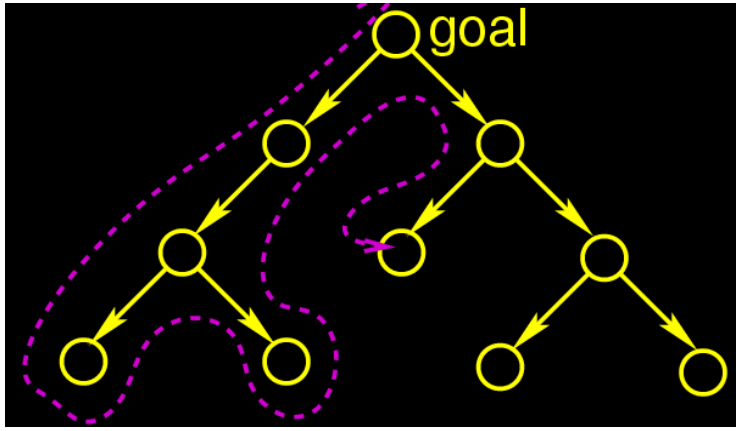
Loengu eesmärk

Anda

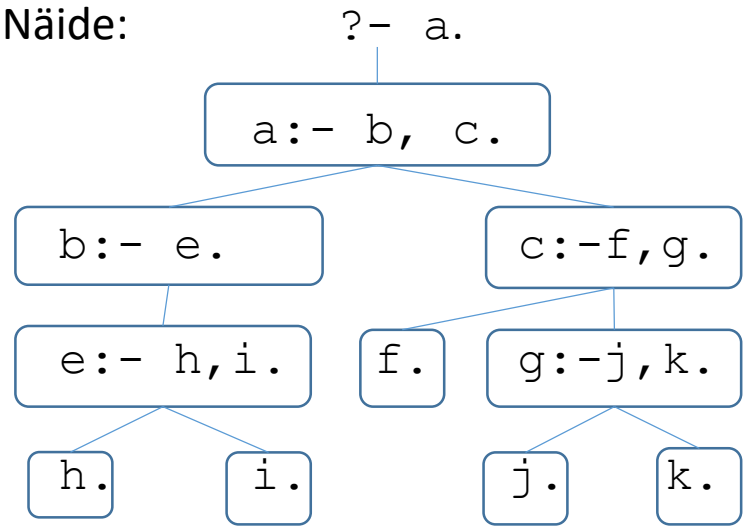
- Teadmised sellest kuidas töötab Prologi otsingumootor
- Oskus kasutada otsingujuhtimise predikaate `cut, fail, repeat`
- Teadmised otsingujuhtimise predikaatide kasutamisega kaasnevatest riskidest

Tagurdamisega otsing

Loogilise programmi täitmine on esitatav (sügavuti) otsingu puuna, kus
· juurtipuks on päring ja kõik ülejäänud tipud on täidetavad Horni laused
· iga kaar esivanemtipust alamtippu vastab pöördumisele esivanemtipule vastava reegli kehast
· puu lehed on faktid



Näide:

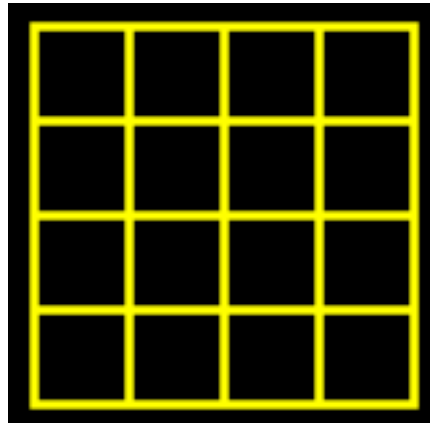


Lahendi otsingu näide: 4 lipu ülesanne

- Olgu 4 × 4 ruuduga malelaud
- Eesmärk:

  paigutada lauale 4 lippu nii, et ükski lipp ei satuks teise lipu tule alla st lipud ei satuks samale reale, veerule ja diagonaalile.
- Olgu ülesande lahendamiseks antud reegel `solution(Q1,Q2,Q3,Q4),` kus

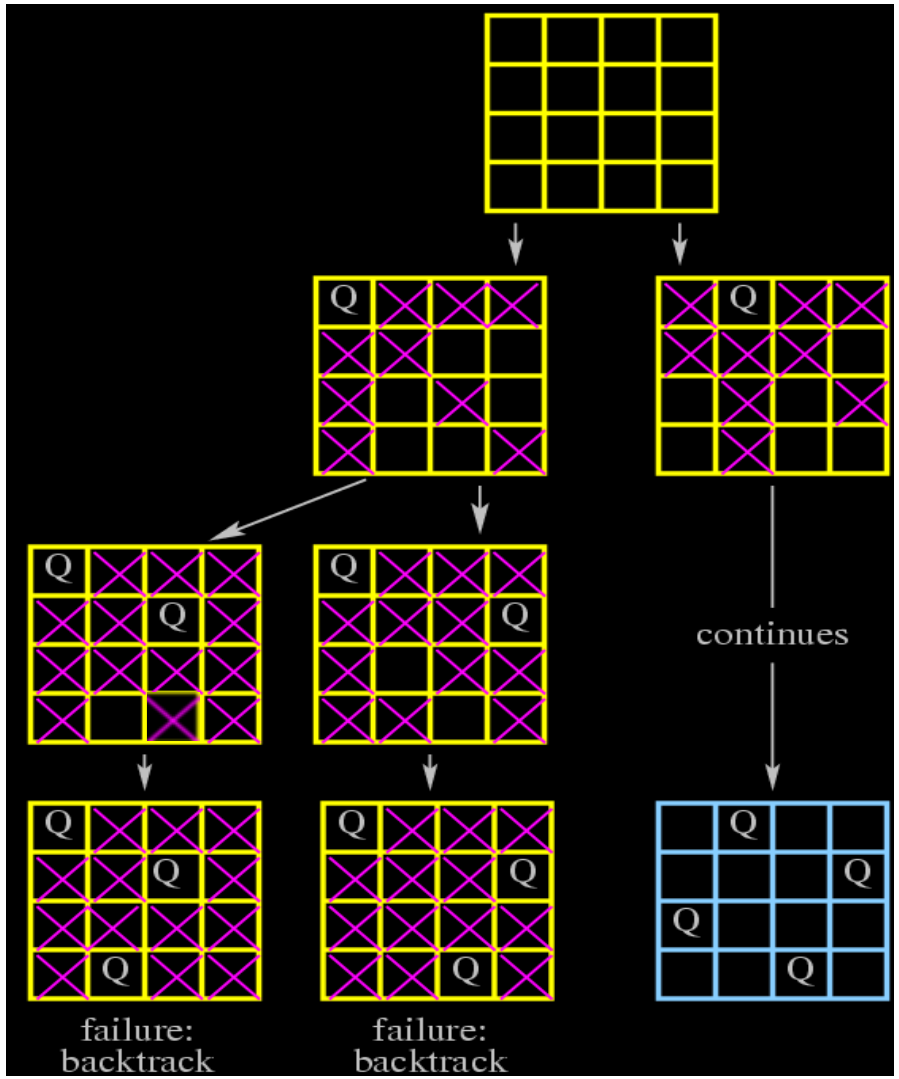
  `Q1,Q2,Q3,Q4` on lippude positsioonid laual.
- Kuidas toimub reegli `solution(Q1,Q2,Q3,Q4)` täitmine?



4 lipu paigutamine: otsingu puu

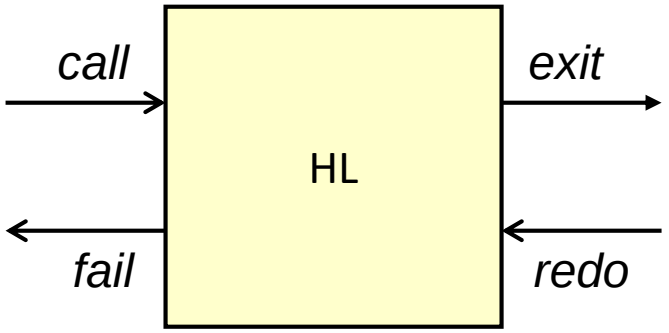
- Sügavuti otsing toimub otsingupuul vasakult paremale
- Otsing **peatub** kui päringu kõik muutujad on väärtustatud nii, et otsingu kitsendused on rahuldatud.
- **Tagurdamine**:

  kui jõutakse olekusse, kust edasi lahendit ei leidu, siis võetakse muutujate viimane väärtustus tagasi ja otsitakse muutujatele uut väärtustust otsingupuus tagurdamisega.



Pinumälu kasutus loogilise programmi täitmisel

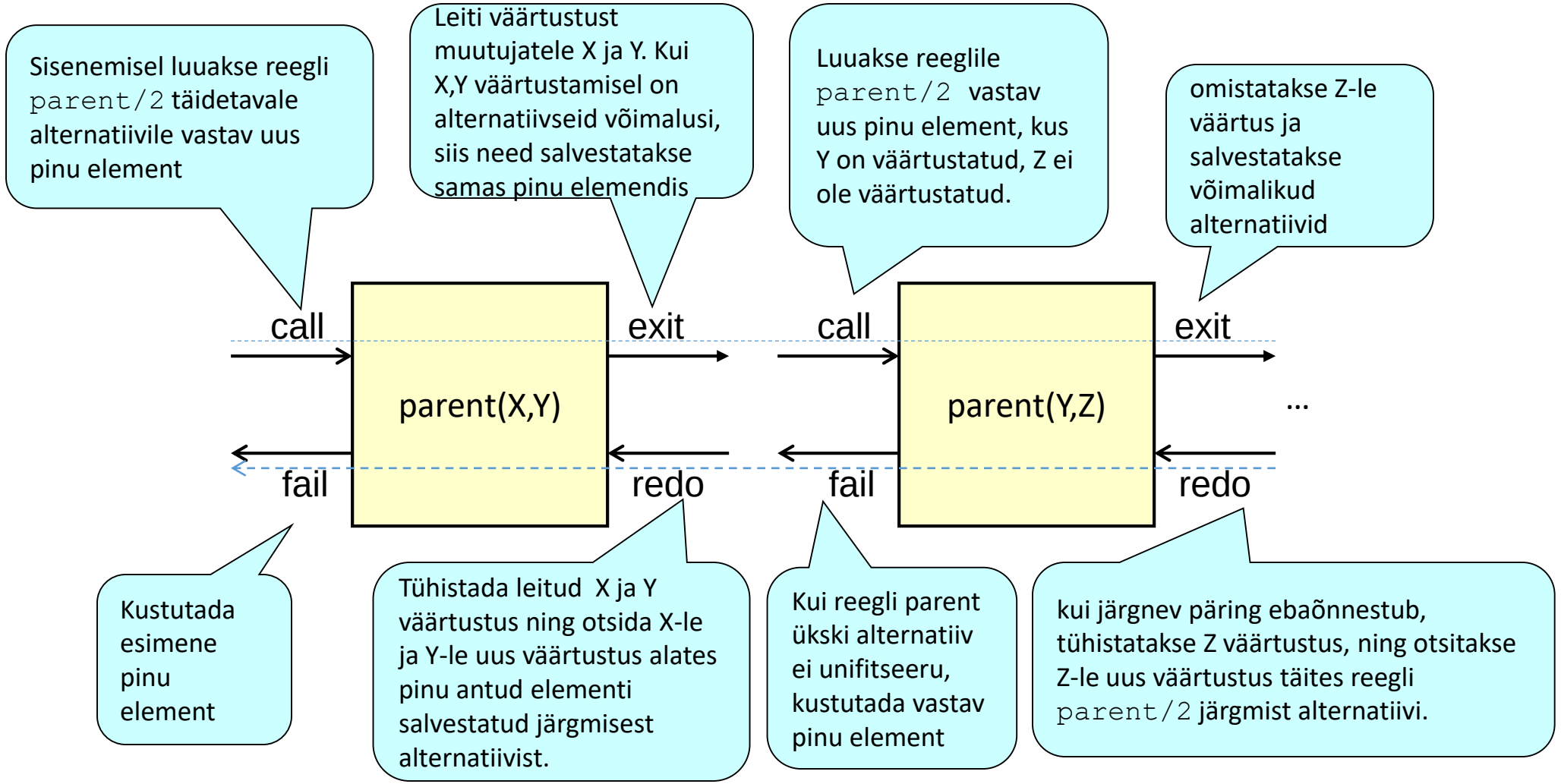
- Iga HL täitmisel lisatakse programmi pinusse element, millel on 2 sisenemise ja 2 väljumise varianti
- Graafiliselt:



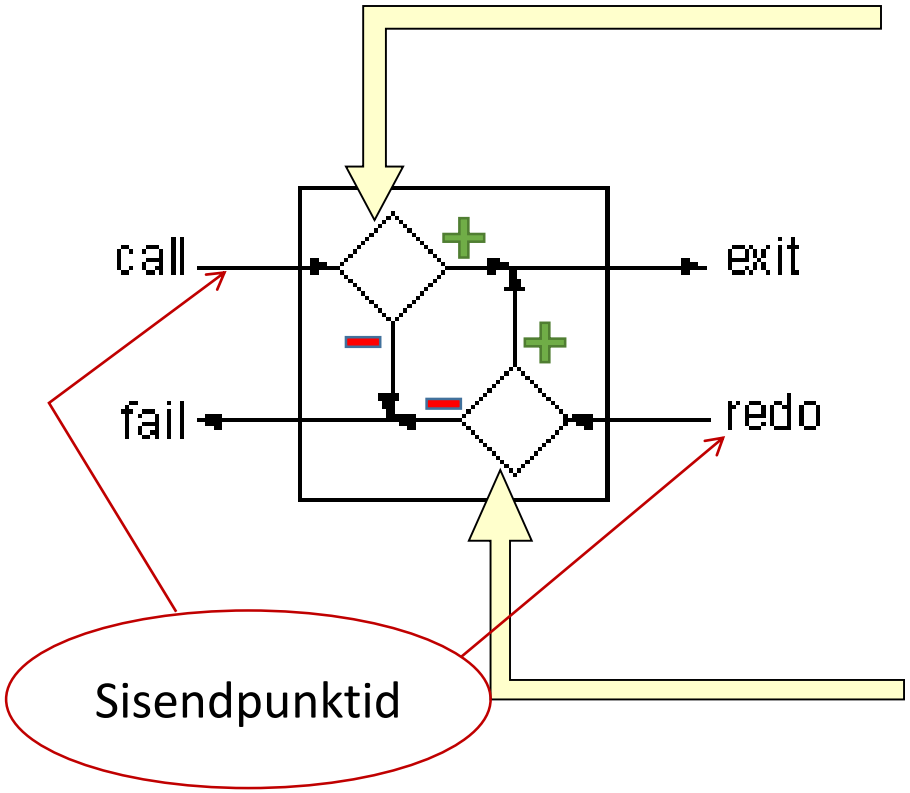
- Sisenemine:
  - *call* (väljakutsel)
  - *redo* (tagurdamisel)
- Väljumine:
  - *exit* (kui reegel tagastab *true*)
  - *fail* (kui reegel tagastab *false*)

- Iga lause täitmine võib lõppeda kas edasiminekuga (programmi järgmise HL lause täitmine) - *exit*
- või ebaõnnestumisega ja tagurdamisega – *fail*
- Kui järgmisena täidetavast HL-st toimub omakorda väljumine *fail*-ga, toimub tagurdamisel sisenemine *redo*-ga

Näide: reegli `anchestor/2` täitmise mudel



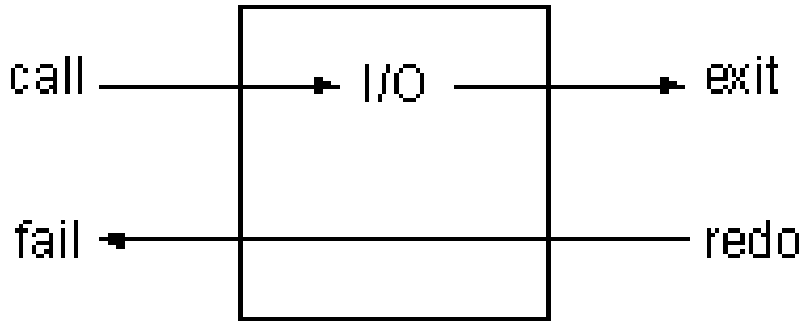
Sisemine juhtimisvoog reegli täitmisel



call – võtab reegli esimene alternatiiv, mis unifitseerib päringuga.
Kui see leidub, siis väärtustatakse muutujad ja väljumine toimub **exit**-ga, muul juhul väljumine **fail**-ga.

redo – annulleeri viimati täidetud alternatiivi muutujate väärtustus ja otsi uus alternatiiv, mis unifitseerub päringuga.
Kui leidub unifitseeruv alternatiiv, siis väärtusta muutujad ja välju **exit**-ga, muul juhul välju **fail**-ga.

I/O predikaatide täitmine



I/O predikaatide täitmisel juhtimisvoo **suund** ei muutu!

- Kui sisenemine CALL, siis väljumine EXIT.
- Kui sisenemine REDO, siis väljumine FAIL.

Täitmist suunavad predikaadid

fail – reverseerib juhtimisvoo,
sisenemine: *call* väljumine: *fail*
(täitmine ei lähe käsust *fail* kaugemale)

repeat – reverseerib tagurdamise.
sisenemine: *call*-ga , siis väljumine: *exit*-ga
sisenemine: *redo*-ga, siis väljumine: *exit*-ga
(tagurdada ei saa *repeat-le* eelnevatele lausetele)

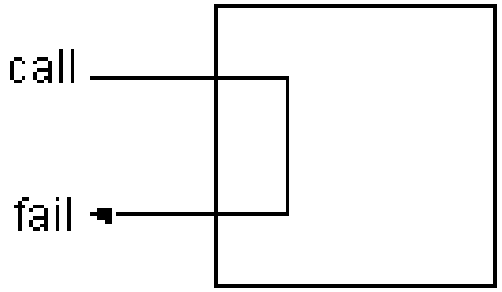
X, Y (and) – väljumine *exit*-ga, kui X ja Y mõlema väljumine on *exit*-ga, muul juhul väljumine *fail*-ga.
REDO tagurdab esmalt lausesse Y ja seejärel X-i.

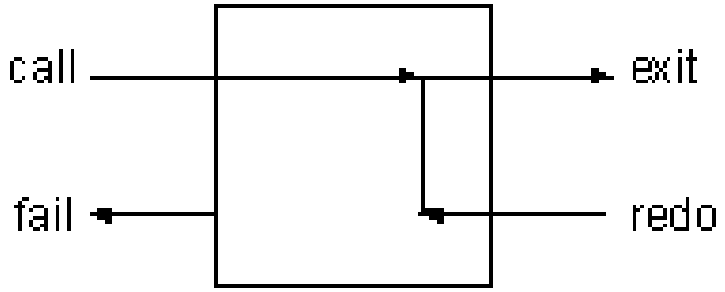
X; Y (or) – väljumine *exit*-ga kui vähemalt ühe X või Y väljumine toimub *exit*-ga.

REDO tagurdab esmalt lausesse X ja seejärel Y-sse.

not(X) – sisenemine *call*-ga ja väljumine *exit*-ga, kui X-st toimub väljumine *fail*-ga. Muul juhul väljumine *fail*-ga. Kui sisenemine *redo*-ga, siis väljumine *fail*-ga

Täitmist suunavad predikaadid: fail ja repeat





fail – tagastab alati *false*, reverseerib juhtimisvoo paremalt vasakule

repeat – tagastab alati *true*, reverseerib juhtimisvoo tagurdamisel paremalt paremale.

Näiteid predikaatide `fail, repeat, not` kasutamisest

Teadmusbaas

```
vanem(mihkel, tiit).
vanem(anna, tiit).
vanem(pille, anna).

mees(mihkel). mees(tiit).

naine(anna). naine(pille).
```

Päringud

```
?- vanem(X, tiit), write(X), nl, fail.
mihkel
anna
false

?- repeat, write('> '), read(quit).
> hi.                    % käsurealt loeti sisend hi
> quit.                  % käsurealt loeti sisend quit
true

?- (mees(X) ; naine(X)).
X = mihkel ;
X = tiit ;
X = anna ;
X = pille ;
false

?- not(vanem(pille,tiit)).
true
```

Predikaatide `cut` ja `fail` kooskasutamine

Küsimus:

Kuidas esitada erandiga reeglit: „Peeter armastab kõiki loomi välja arvatud madusid“?

- Lahendus 1:

```
armastab(peeter,X) :- madu(X),fail.
armastab(peeter,X) :- loom(X).
```

Selgitus

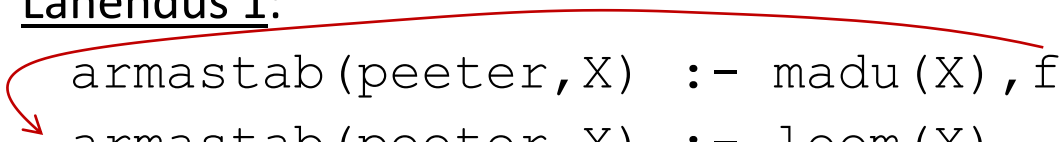
1. alternatiiv kirjeldab erandit, kui X väärtus unifitseerib predikaadi `madu` parameetriga
2. alternatiiv esitab üldist reeglit, kus X väärtus unifitseerib predikaadi `loom` parameetriga

Predikaatide `cut` ja `fail` kooskasutamine

Küsimus:

Kuidas kirjutada reeglit: „Peeter armastab kõiki loomi välja arvatud madusid“?

- Lahendus 1:



```
armastab(peeter,X) :- madu(X),fail.
armastab(peeter,X) :- loom(X).
```

See ei ole korrektne lahendus!

`fail` sunnib tagurdamisel valima reegli `armastab` järgmise alternatiivi, mis tagastab `true` päringule `?- armastab(peeter, madu).`

Predikaatide `cut` ja `fail` kooskasutamine

Küsimus:

Kuidas kirjutada reeglit: „Peeter armastab kõiki loomi välja arvatud madusid“?

- Lahendus 1:

```
armastab(peeter,X) :- madu(X),fail.
armastab(peeter,X) :- loom(X).
```

- See ei ole korrektne, sest fail sunnib tagasivõtuga valima järgmise alternatiivi, mis tagastab `true`.

Kuidas vältida reegli järgmise alternatiivi täitmist, kui eelmine alternatiiv lõpetab `fail`-ga?

- Lahendus 2: („ !, cut “ operaatorite paari kooskasutamine)

```
armastab(peeter,X) :- madu(X),!,fail.
armastab(peeter,X) :- loom(X).
```

CUT-operaatori `!/0` kasutamine

- Tagurdamine on Prologi otsingumootori oluline omadus,
- kuid võib anda ebaefektiivse programmi täitmise:
  - Prolog võib kulutada aega ja mälu otsingupuu nende harude läbimiseks, mis ei anna vajalikku tulemust.
  - Otsingu juhtimiseks tagurdamisel tuleks kärpida otsingupuud
- Tagasivõttu saab juhtida cut-operaatoriga - “ ! “
- cut-operaatoril ei ole argumente

cut-operaatori kasutamise näide

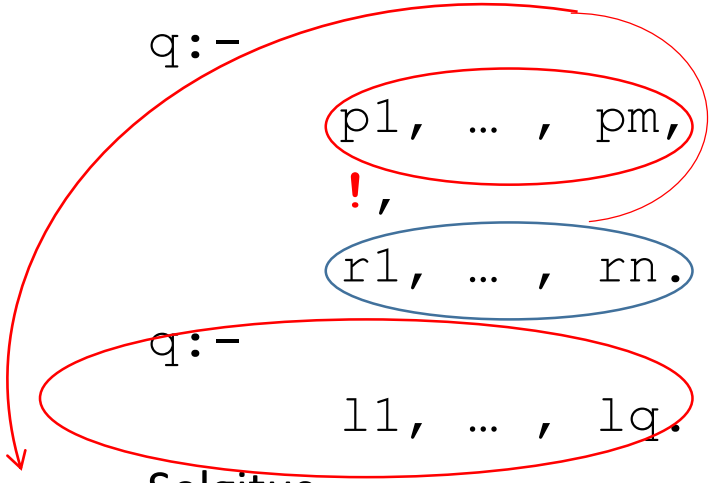
- “!“ saame lisada reegli kehasse nagu iga teise predikaadi:
- Näide:

```
p(X):- b(X), c(X), !, d(X), e(X).
```

- Cut’i täitmine õnnestub alati
- Cut’i kasutamine võimaldab säilitada otsingul lahendi, mis on leitud enne cut-operaatorini jõudmist.

Kuidas cut toimib?

- Olgu reegel kahe alternatiiviga



Kui päringutele `r1,…,rn` rohkem alternatiive ei leidu, siis päringutest `p1, … ,pm` hüpatakse tagurdamisel üle

Nendele päringutele otsitakse tagurdamisel uusi lahendeid

Ka cut-i alternatiivile järgnevatest alternatiividest hüpatakse tagurdamisel üle

Selgitus

- Tagurdamise korral toimub alternatiivsete lahenduste otsimine päringutele r1,…,rn, st. päringutele, mis on reegli alternatiivi kehas peale !-operaatorit,
- predikaadid `p1, … , pm` läbitakse tagurdamise korral ilma neile uut lahendit otsimata,
- samuti ei otsita uut lahendit reegli q/0 järgmisetele alternatiividele.

Näide: reegel ilma cut-operaatorita

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1). b(2).
c(1). c(2).
d(2).
e(2).
f(3).
```

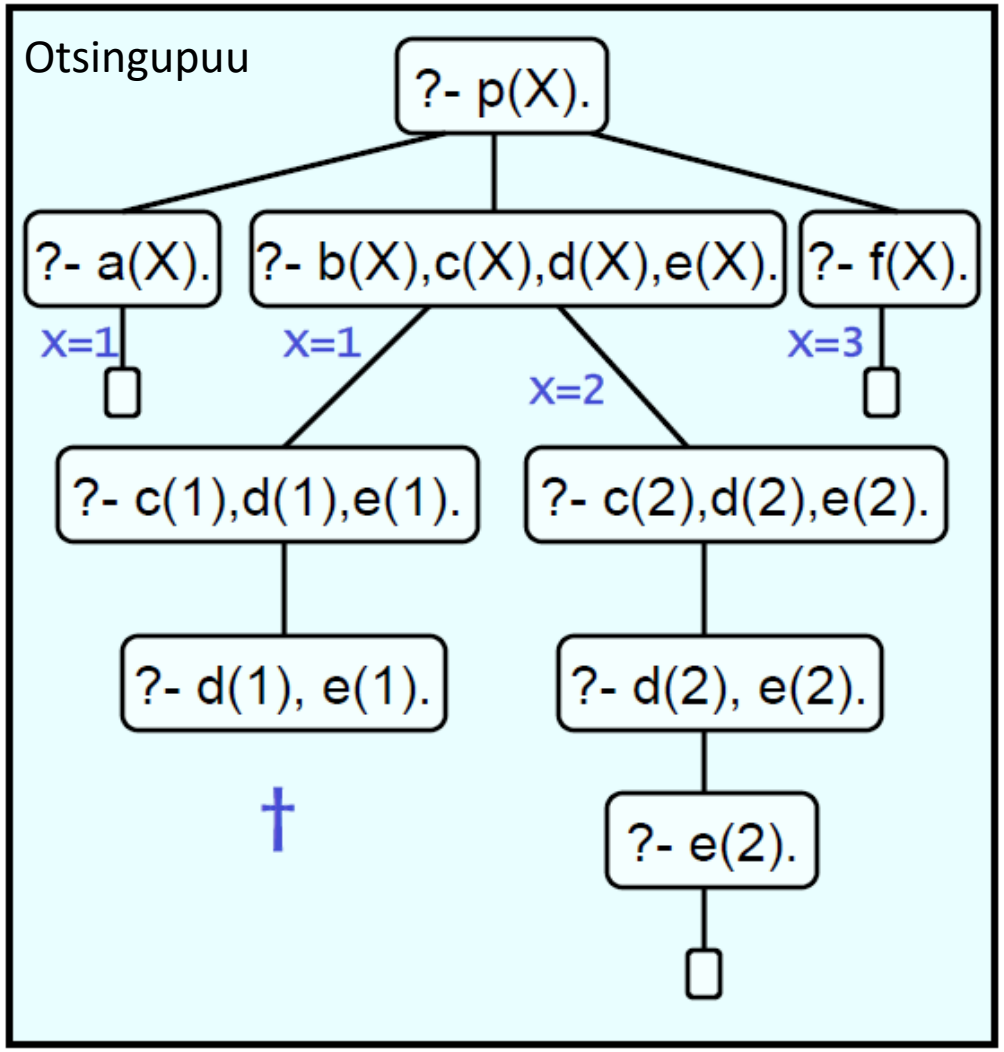
Esimene lahend

Teine lahend

Kolmas lahend

Kuidas toimub päringu täitmine?

```
?- p(X).
X=1;
X=2;
X=3;
no
```



cut-operaatori lisamine

- Lisame cut-operaatori reegli `p/1` teise alternatiivi

```
p(X):- a(X).
p(X):- b(X), c(X),!, d(X), e(X).
p(X):- f(X).
```

- Päring `?- p(X).` Tagastab nüüd eelnevaga võrreldes erineva tulemuse.

```
?- p(X).
X=1;
no
```

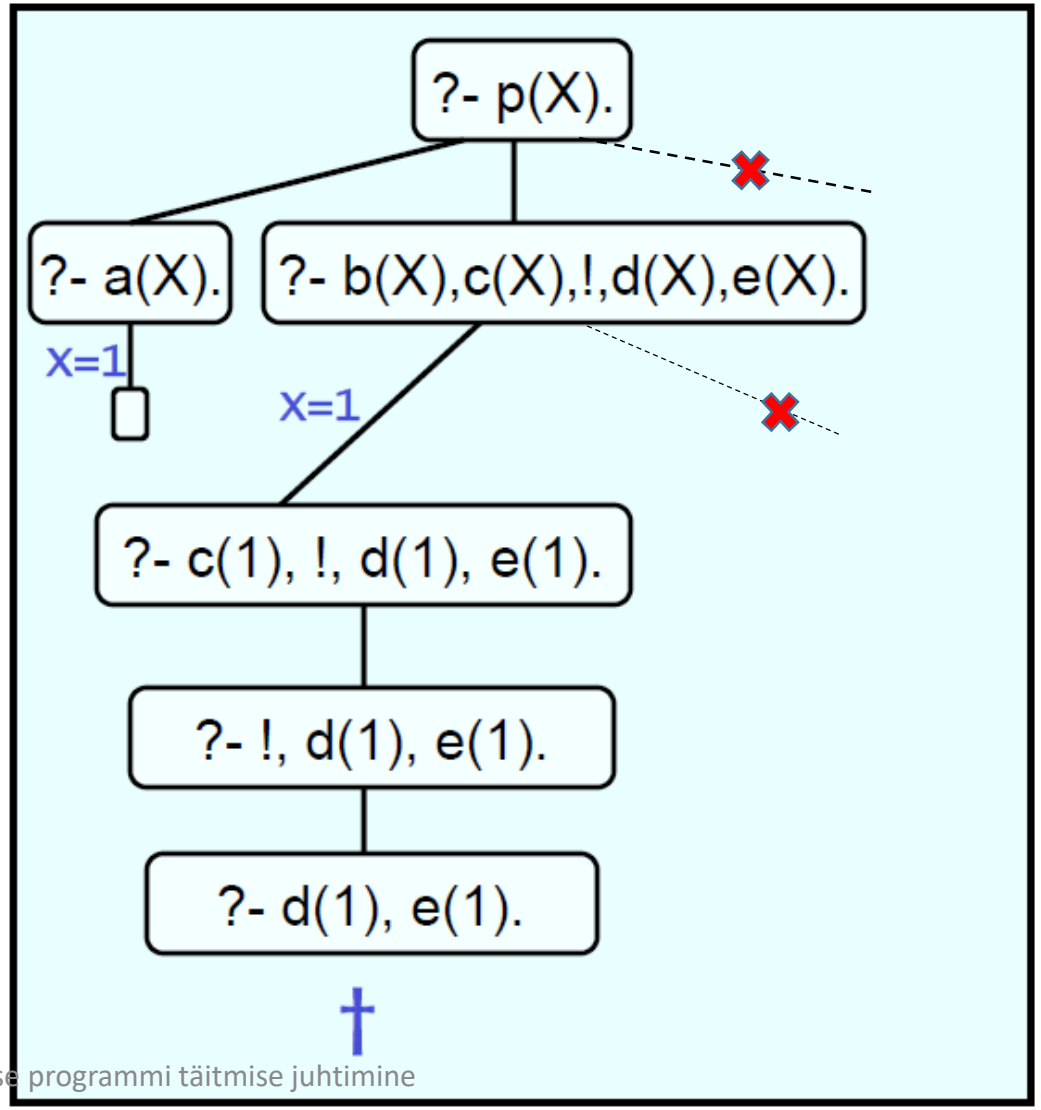
Näide reeglist, kus esineb cut-operaator

```
p(X):- a(X).
p(X):- b(X), c(X),!, d(X), e(X).
p(X):- f(X).
a(1).
b(1). b(2).
c(1). c(2).
d(2).
e(2).
f(3).
```

Seda osa tagurdamisel enam ei täideta!

- Päring

```
?- p(X).
X=1;
no
```



Kuidas kasutada cut-i õigesti?

- Vaatame predikaati `max/3` , mis tagastab *`true`*, kui kolmas argument on kahe esimese argumendi maksimum.

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

```
?- max(2,3,3).
yes

?- max(7,3,7).
yes
```

```
?- max(2,3,2).
no

?- max(2,3,5).
no
```

- Näib, et reegel toimib õigesti, ainult, et siin on liiasus – teise alternatiivi tingimuse võiks ära jätta.

Kuidas kasutada cut-i õigesti?

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

- Päringu `?- max(3,4,Y).` korral unifitseeritakse Y argumendi väärtusega 4.
- Kuid, kui küsida järgmist lahendit, siis püütakse täita teist alternatiivi, mis antud argumentide väärtuste korral tagastab `false`.
- Tõepoolest, tingimuse `X > Y` kontrollimine teise alternatiivi kehas on liiasus.
- cut-operaatori lisamisega vabaneme sellest:

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

Kuidas kasutada cut-i õigesti?

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

- Kui esimese alternatiivi tingimus on tõene, siis cut lõikab otsingupuust teise alternatiivi
- kui esimese alternatiivi tingimus ei kehti, siis täidetakse teine alternatiiv

Millal kasutada cut-i: roheline ja punane cut

- **Rohelise cut'i** kasutamine ei muuda programmi semantikat.
- Reeglis `max/3` on roheline cut:
  – uus kood annab täpselt sama tulemuse, kui ilma cut'ta versioon
  – kuid on efektiivsem – programm lõpetab ilma vääraid alternatiive läbimata.
- **Punane cut** muudab programmi semantikat.
- Näiteks lihtsustame `max/3` reeglit eemaldades teisest alternatiivist tingimuse „`X > Y`“:

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

Roheline ja punane cut

- Kuidas see mõjutab täitmist?

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

```
?- max(200,300,X).
X=300
yes
```

```
?- max(400,300,X).
X=400
yes
```

```
?- max(200,300,200).
yes
```

uups!

- Toome sisse väljundparameetri unifitseerimise Y=Z peale cut-i:

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

```
?- max(200,300,200).
no
```

Punase cut-ga kaasnevad ohud

- Punase cut-ga programmid
  - ei ole täis-deklaratiivsed
  - on rasked lugeda
  - võivad tekitada raskesti avastavaid vigu

fail/0

- Predikaat, mis alati tagastab `false`, kui Prolog proovib seda täita
- Vajalik, kui otsingut tuleb sundida tagurdama.

Näide `fail` ja cut'i kasutamisest

Tahame kirjutada reegli „Vincent armastab burgereid v.a. suured Kahuna burgerid“

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

```
?- enjoys(vincent,a).
yes
```

```
?- enjoys(vincent,b).
no
```

```
?- enjoys(vincent,c).
yes
```

Eituse kodeerimine *`fail`*'i kaudu

- Kodeerime eituse *fail*'i kaudu järgmise reegli abil

```
neg(Goal):- Goal, !, fail.
neg(Goal).
```

Näide fail ja cut’i kasutamisest

- kodeerime Vincenti näite uuesti predikaadi `neg/1` abil

```
enjoys(vincent,X):- burger(X),
                    neg(bigKahunaBurger(X)).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

```
?- enjoys(vincent,X).
X=a;
X=c;
X=d;
no
```

Standardpredikaat \+

„\+“ on süsteemi predikaat, mis realiseerib predikaadi „neg“.

```
enjoys(vincent,X):-  burger(X),
                     \+ bigKahunaBurger(X).
```

```
?- enjoys(vincent,X).
X=a;
X=c;
X=d;
no
```

„\+“ on kohatundlik:

```
enjoys(vincent,X):- \+ bigKahunaBurger(X),
                       burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

```
?- enjoys(vincent,X).
no
```