

Loogilise programmi täitmise juhtimine

ITI0021

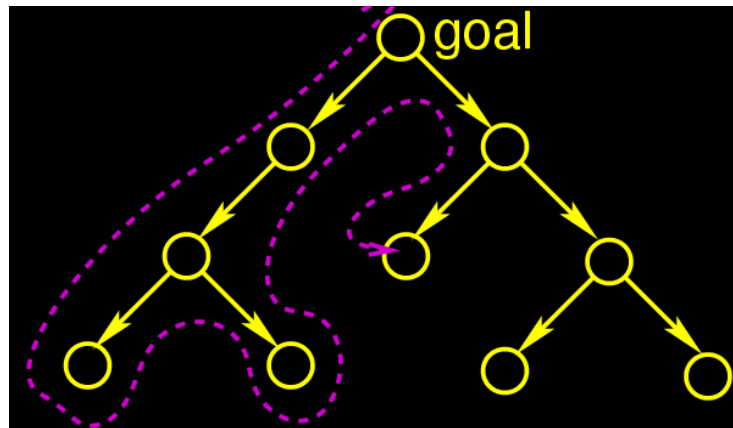
Sügis 2018

Loeng 7

Tagurdamisega otsing

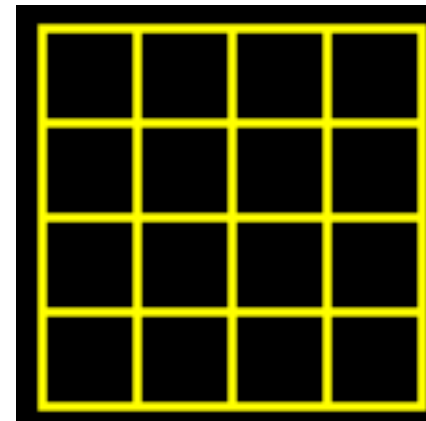
Päringu täitmine on esitatav otsingupuuna, kus

- juurtipp on päring
- iga haru vastab pöördumisele reegli kehast.
- puu lehed on faktid



Otsingu näide: 4 lipu ülesanne

- Olgu 4×4 ruuduga malelaud
- Paigutada lauale 4 lippu nii, et samale reale, veerule ja diagonaalile ei satuks korraga 2 lippu.
- Otsing toimub reegli `solution(Q1, Q2, Q3, Q4)` täitmisel,
- $Q1, Q2, Q3, Q4$ on lippude positsioonid.

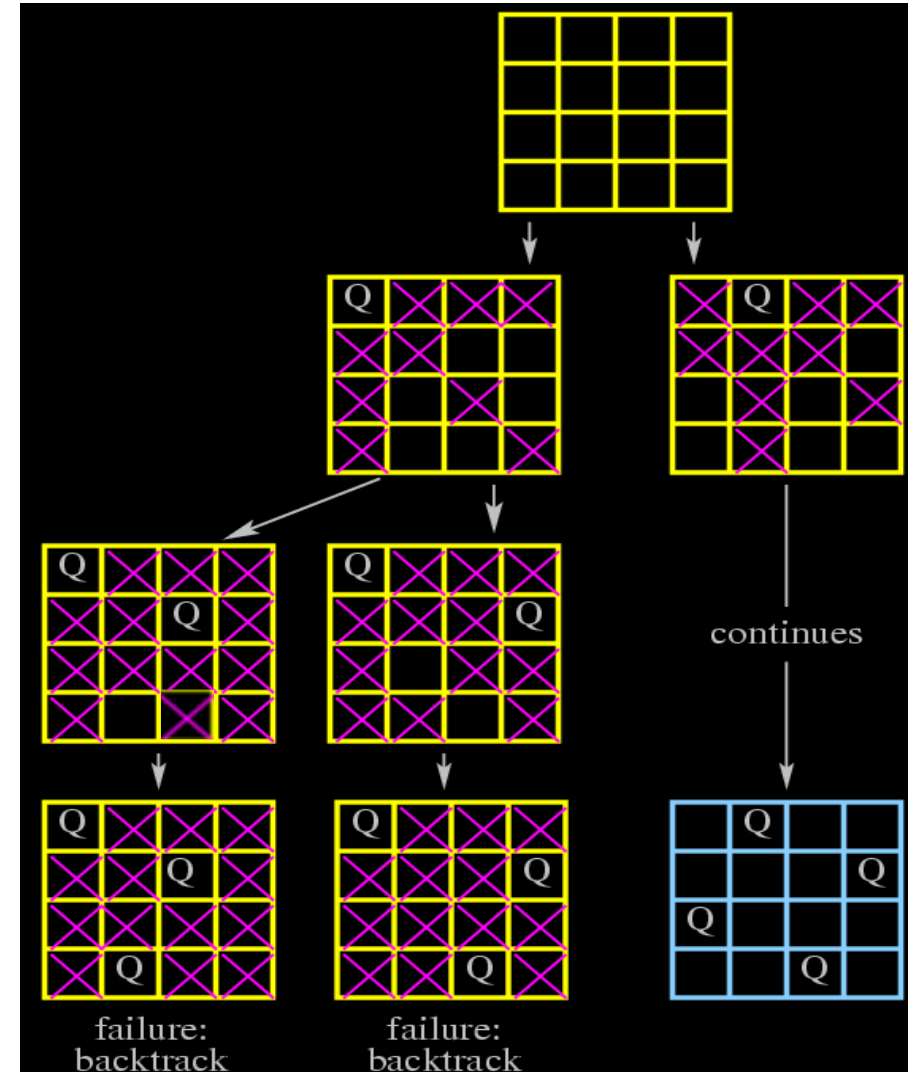


4 lipu paigutamine: otsingu puu

- Otsing **peatub** kui päringu kõik muutujad on väärtustatud.

- **Tagurdamine:**

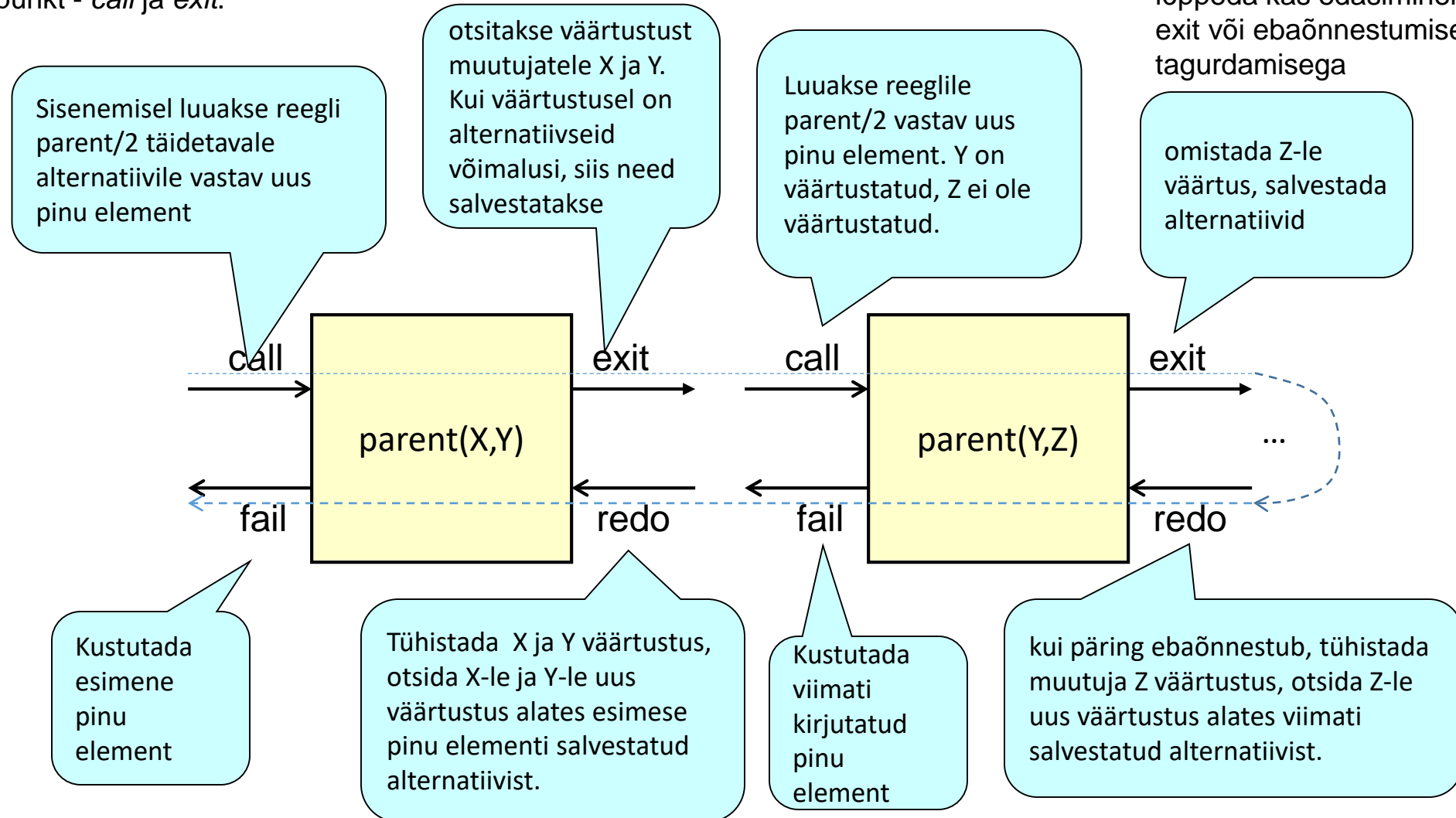
kui jõutakse olekusse, kust edasi ei leidu lahendit, siis võetakse muutujate viimati tehtud väärtustus tagasi ja otsitakse muutujatele uut väärtustust reegli järgmise alternatiivi täitmisega.



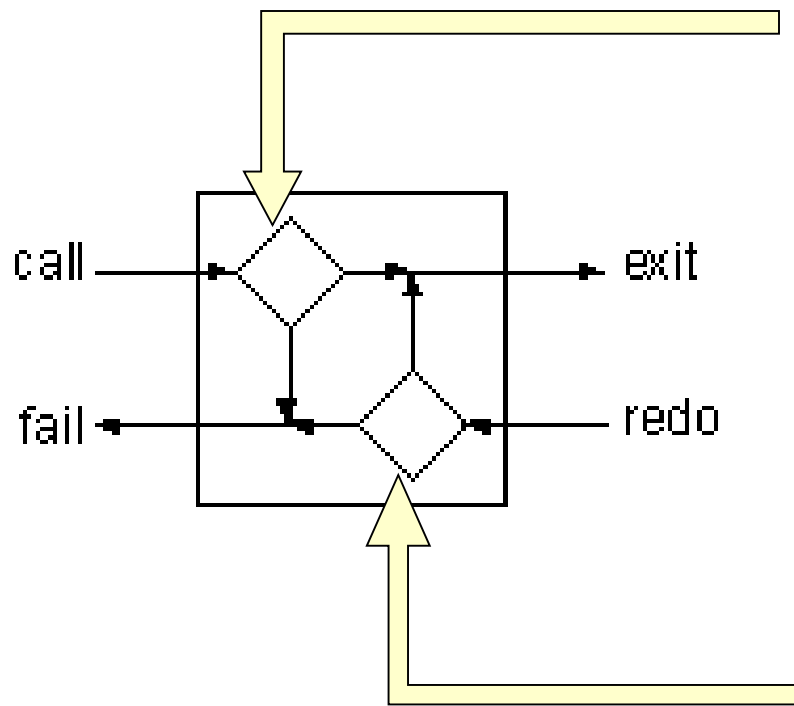
Reegli parent/2 täitmise mudel

Igal koodireal on sisend- ja väljundpunkt - *call* ja *exit*.

Iga koodirea täitmine võib lõppeda kas edasimineku - *exit* või ebaõnnestumisega ja tagurdamisega



Sisemine juhtimisvoog



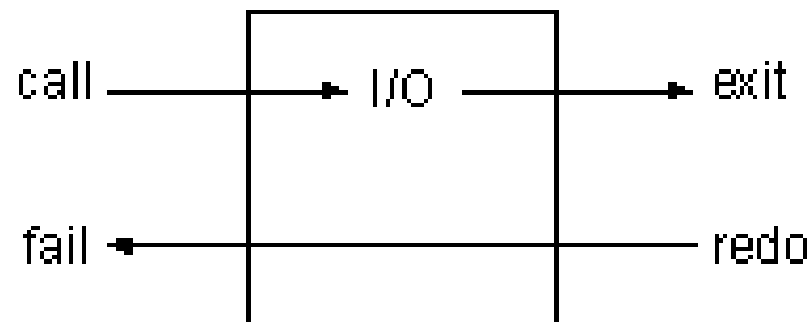
call -- Look for first clause of a predicate that unifies with goal.

If found, bind variables and **exit**, else **fail**.

redo -- From last clause tried, unbind variables bound with this goal and look for next clause that unifies with goal.

If found, bind variables and **exit**, else **fail**.

I/O predikaatide täitmine



Do not change direction of execution.

in CALL, do I/O, out EXIT.

in REDO, out FAIL.

Täitmise juhtimise predikaadid

fail – reverses forward control flow,
in CALL out FAIL
(*no way past fail/0*)

repeat – reverses backward control flow.
in CALL out EXIT
in REDO out EXIT
(*no way back after repeat/0*)

X, Y (and) – succeeds if X and Y
succeed, otherwise fails.
REDO backtracks into Y, and then X.

X; Y (or) – succeeds if X or Y succeed.
REDO backtracks into X, and then Y.

call(X) – treats X as a goal.

not(X) – calls X and succeeds if X fails,
otherwise fails.
FAIL on REDO.

```
parent(michael, diego).  
parent(ana, diego).  
parent(pilar, ana).  
  
male(michael). male(diego).  
  
female(ana). female(pilar).
```

```
?- parent(X, diego), write(X), nl, fail.
```

```
michael
```

```
ana
```

```
false
```

```
?- repeat, write('> '), read(quit).
```

```
> hi.
```

```
> quit.
```

```
true
```

```
?- (male(X) ; female(X)).
```

```
X = michael ;
```

```
X = diego ;
```

```
X = ana ;
```

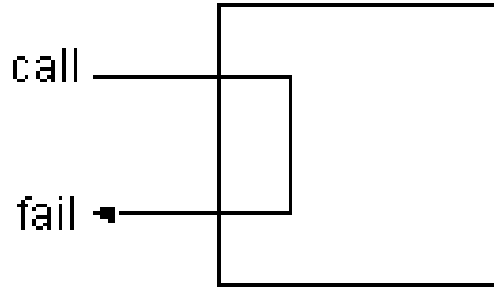
```
X = pilar ;
```

```
false
```

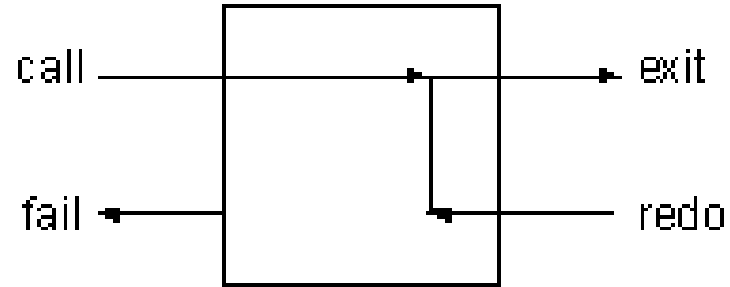
```
?- not(parent(pilar,diego)).
```

```
true
```


Täitmise juhtimise predikaadid



fail -- always fails,
reverses flow of
control from right to
left, to left to right.



repeat -- always succeeds,
reverses flow of control
from left to right, to right
to left.

Cut- ja fail- operaatorite kasutamine.

Küsimus:

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

• Lahendus 1:

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

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

1. alternatiiv kirjeldab erandit
2. alternatiiv esitab üldist reeglit

Cut- ja fail- operaatorid.

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 valida järgmise alternatiivi ja tagastab `true` ka päringule `?- armastab(peeter, madu).`

Cut- ja fail- operaatorid.

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 ja true.

- Lahendus 2:

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

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

CUT-operaator !/0

- 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 on vaja keeles vahendeid
- Tagasivõttu saab juhtida cut-operaatoriga “!”
- cut-operaatoril ei ole argumente

cut-operaatori kasutamise näide

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

- Näide:

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

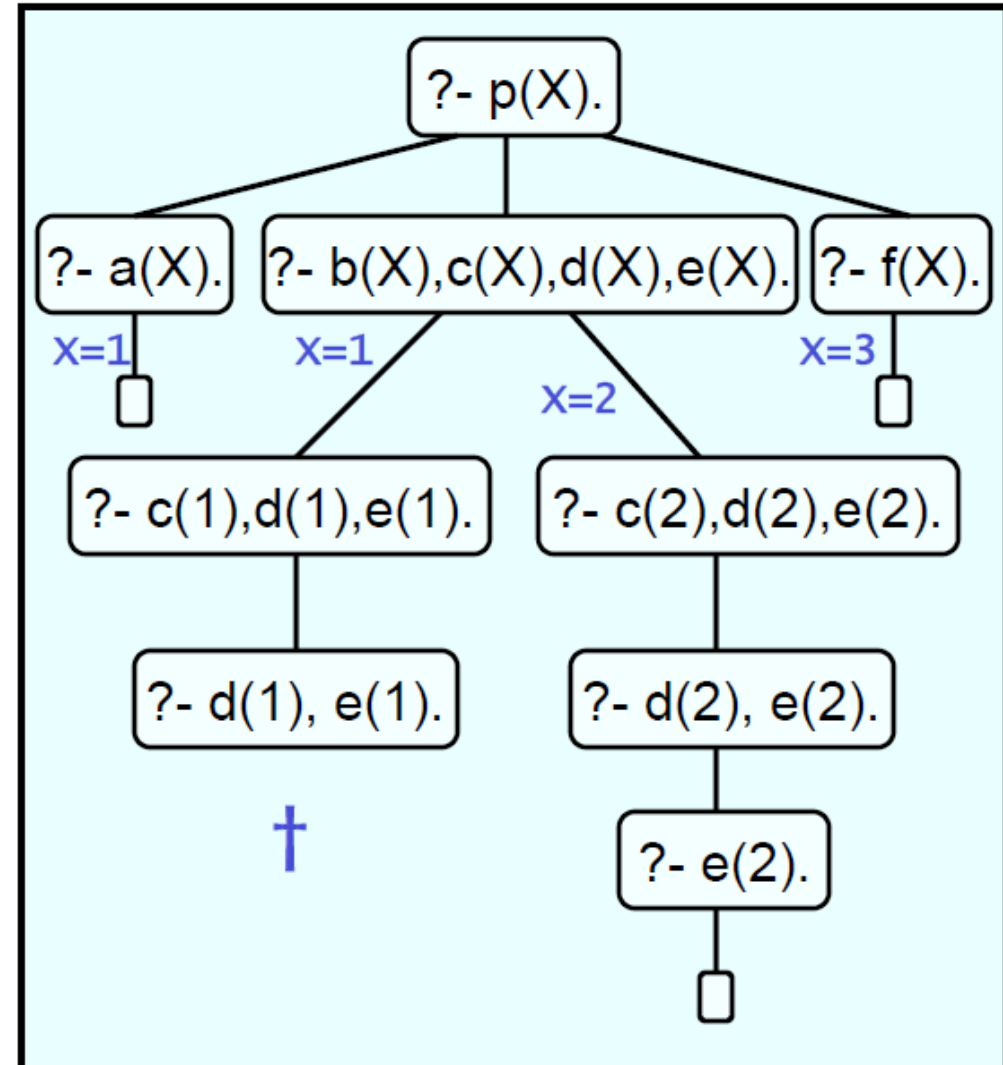
- Cut'i täitmine õnnestub alati
- Cuti kasutamine võimaldab säilitada otsingul lahendi, mis on leitud enne cut-operaatorini jõudmist.

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).
```

Kuidas toimub päringu täitmine?

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



cut-operaatori lisamine

- Lisame cut-operaatori reegli p/1 teise alternatiiv

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

- Päring ?- p(X). tagastab eelnevaga võrreldes erineva tulemuse.

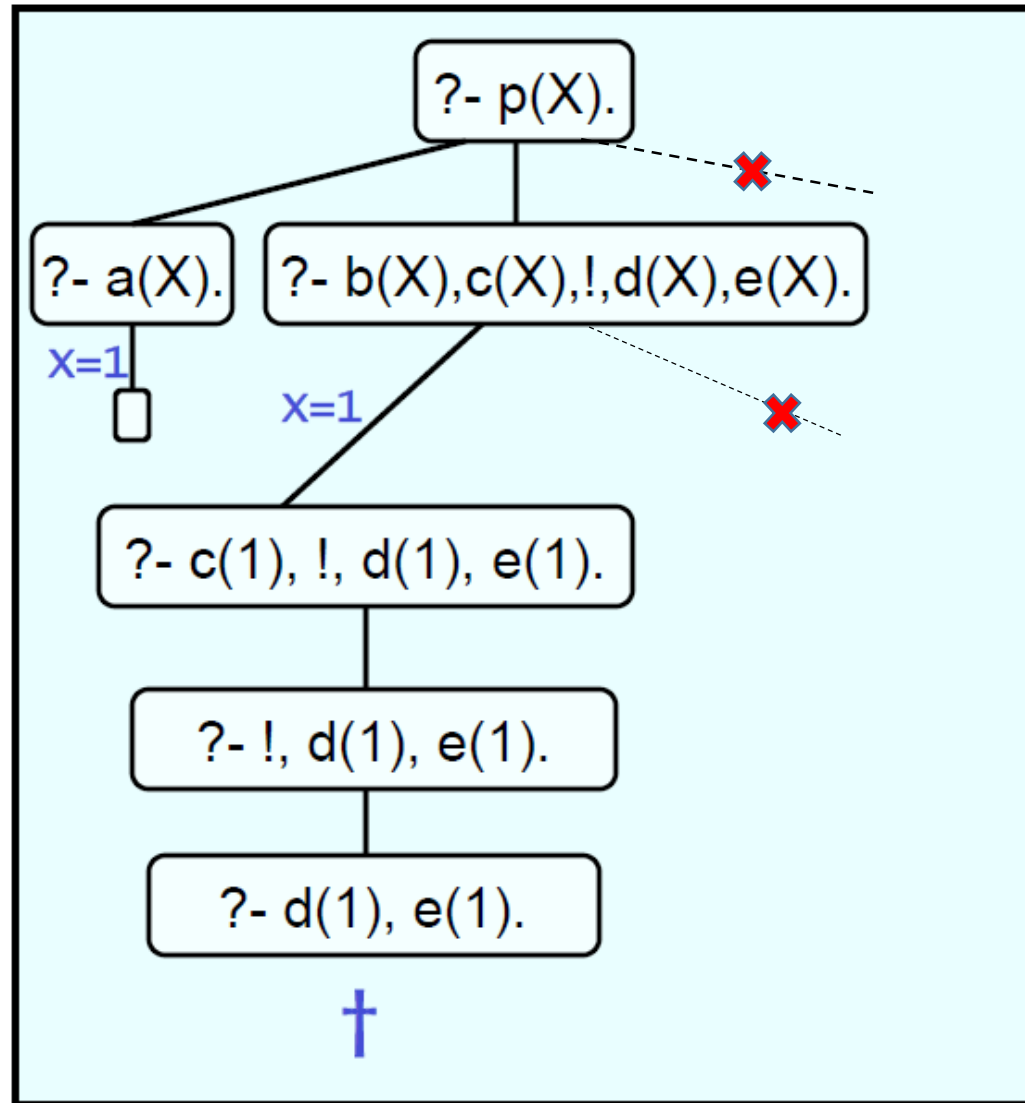
```
?- p(X).  
X=1;  
no
```


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).
```

• Päring

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



Kuidas cut toimib?

- Olgu reegel

q:-

$p_1, \dots, p_m,$

!,

$r_1, \dots, r_n.$

- Tagasivõtu korral
 - toimub alternatiivsete lahenduste otsimine termidele r_1, \dots, r_n , st. termidele, mis on paremal poolt !-operaatorit,
 - termid p_1, \dots, p_m läbitakse tagasivõtu korral ilma neile uut lahendit otsimata,
 - samuti ei otsita uut lahendit reegli q/0 järgmiste alternatiividele.

cut-operaatori kasutusnäiteid

- Vaatame predikaati max/3 , mis tagastab true, kui kolmas argument on kahe esimese argumenti 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
```

Kuidas cut toimib?

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

Reeglis on liiasus:

- Päringu ?- max(3,4,Y). korral unifitseeritakse Y argumendi väärtusega 4.
- Kuid, kui nõuda järgmist lahendit, siis püütakse täita teist alternatiivi, mis antud argumentide väärtuste korral tagastab false.
- Tingimuse $X > Y$ kontrollimine 2. alternatiivi kehas on liiasus.
- cut-operaatori lisamisega vabaneme sellest:

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

Kuidas cut toimib?

```
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

Roheline ja punane cut

- **Roheline 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
- **Punane cut** muudab programmi semantikat.
- Näiteks lihtsustame max/3 reeglit eemaldades tingimuse 2.-st alternatiivist:

```
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 unifitseerimise 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 täisdeklaratiivsed
 - on rasked lugeda
 - võivad tekitada raskesti avastavaid vigu

fail/0

- Predikaat, mis alati tagastab false, kui Prolog proovib seda täita
- Kasulik, kui on vaja suunata Prolog tagasivõttu tegema.

Näide fail ja cut'i kasutamisest: Vincent ja 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).
```

Tahame kodeerida erandi: „Vincent ei armasta suuri Kahuna burgereid“

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

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

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

Eituse kodeerimine *fail* kaudu

- Kodeerime eituse kui *fail*'i järgmise reeglina

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

Näide fail ja cut'i kasutamisest: Vincent ja burgerid (jätk)

- kodeerime Vincenti näite neg predikaadi 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).
```

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