

PROLOG un loģiskā programmēšana

Dr. sc. ing. Pāvels Rusakovs

Mg. sc. ing. Pāvels Semenčuks

Literatūras saraksts:

1. *Bratko Ivan*. Prolog Programming for Artificial Intelligence. Third edition.

Addison-Wesley, Pearson Education Limited, 2001.

Братко Иван. Алгоритмы искусственного интеллекта на языке PROLOG. Третье издание.

Москва, Издательский дом “Вильямс”, 2004.

-
2. *Luger George F*. Artificial Intelligence. Structures and Strategies for Complex Problem Solving. Fourth Edition.

Addison-Wesley, Pearson Education Limited, 2002.

Люгер Д.Ф. Искусственный интеллект. Стратегии и методы решения сложных проблем. Четвертое издание.

Москва, Издательский дом "Вильямс", 2003.

3. *Адаменко А.Н., Кучуков А.М.* Логическое программирование и Visual Prolog. Санкт-Петербург, “БХВ-Петербург”, 2003.

4. *Yin Khin Maung, Solomon David W.* Using Turbo PROLOG. Que, 1987.

Ин Ц., Соломон Д. Использование Турбо-Пролога. Москва, "Мир", 1990.

5. *Covington Michael A., Nute Donald, Vellino Andre.* Prolog Programming in Depth. Prentice Hall, 1996.

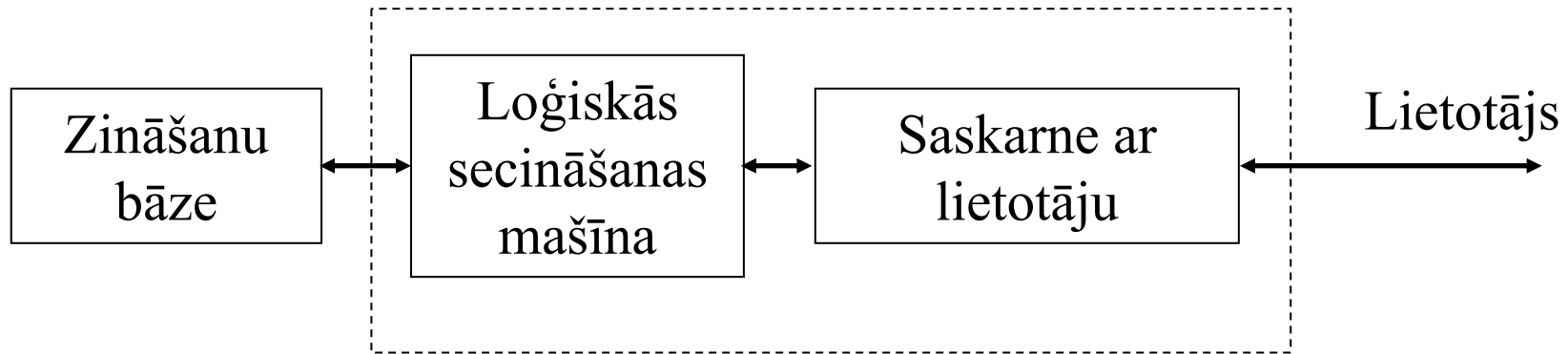
6. *Clocksin W.F., Mellish C.S.* Programming in Prolog: Using the ISO Standard. Springer, 2003.

7. *Sterling Leon, Shapiro Ehud*. The Art of Prolog.
The MIT Press, 1994.

Стерлинг Леон, Шапиро Эхуд. Искусство программирования на языке Пролог. Москва, “Мир”, 1990.

8. *Pratte Robert*. Logic Programming with Perl and Prolog.
O'Reilly, 2005.

Ekspertsistēmas struktūra



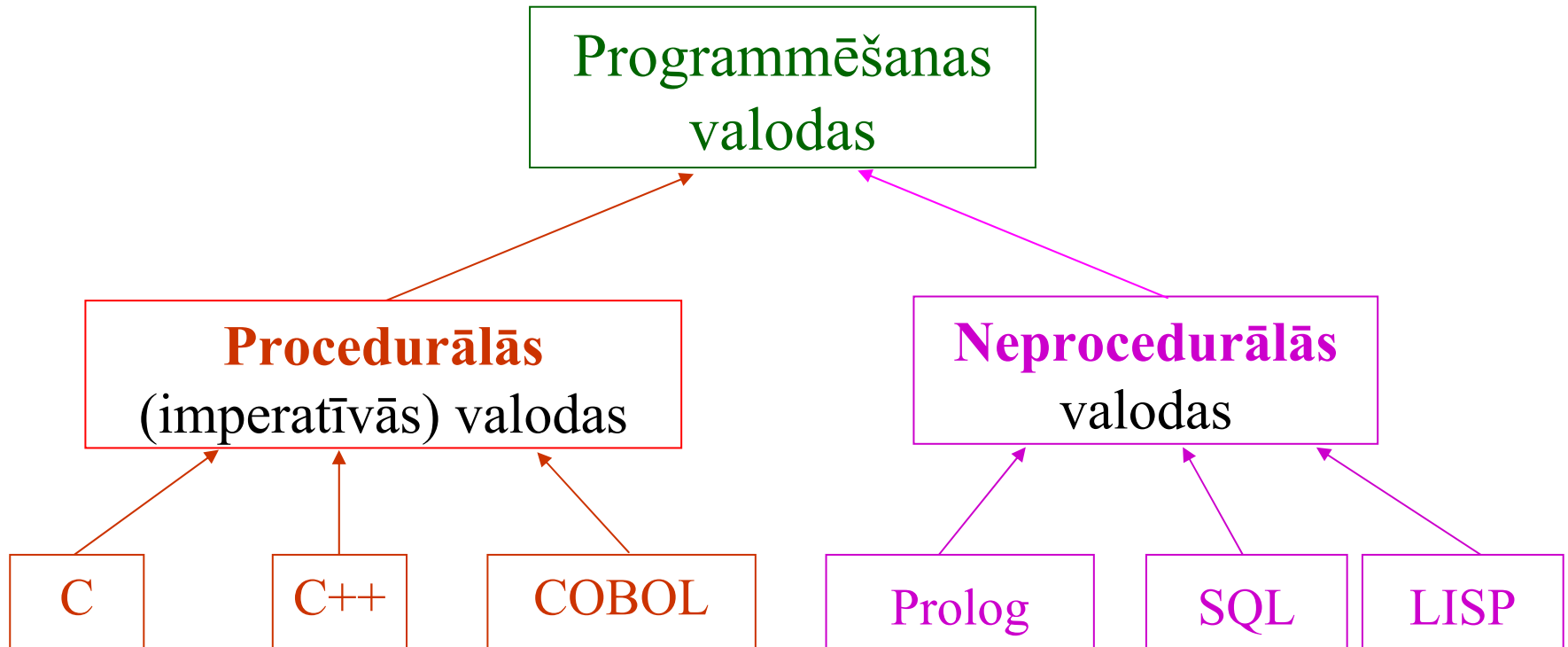
Čaula

Zināšanu attēlošanas piemērs: likumi **If** – **Then** (**Ja** – **Tad**)

Prasības:

1. Modularitāte.
2. Inkrementālas uzaudzēšanas iespēja.
3. Modifikācijas ērtība.

Valodas *Prolog* veidošana



Neprocedurālās valodas izmantošana (SQL)

```
SELECT Employees.*  
FROM Employees  
WHERE Name = „Uldis”
```

Mēs nezinām, **KĀ** bija iegūts rezultāts.

Cikls **for**?

Cikls **while**?

Cikls **do-while**?

PROLOG pamats: *FOPC (First Order Predicate Calculus)*.

Pirmās kārtas predikātu aprēķini.

PROLOG - programmai ir divi jēgas līmeni:

Deklaratīvā jēga.

Kas būs par programmas rezultātu, ko darīt?

Procedūrālā jēga.

Kā šis rezultāts bija iegūts, kā to darīt?

Programmas deklaratīvos aspektus bieži krietni vieglāk saprast, nekā procedurālās detaļas.

Loģiskā programma sastāv no *teikumiem*.

Tajā ir *galīgais* teikumu daudzums

Teikumu tipi:

1. *Fakti*. Apgalvojumi, kuri vienmēr patiesi.
2. *Likumi*. Apgalvojumu patiesums ir atkarīgs no dažiem nosacījumiem.
3. *Jautājumi*. Iespēja uzzināt par patiesīgiem/nepatiesīgiem apgalvojumiem.

Procedūra – likumu kopa, kas raksturo vienu un to pašu attiecību.

Loģiskās programmas teikums izskatās tā:

$A \leftarrow B_1, B_2, \dots, B_n ; n \geq 0$

1. Lai $n=0$. Tad A ir *fakts*.

Deklaratīva interpretācija: A ir patiess.

Procedurālā interpretācija: mērķis A ir sasniegts.

2. Lai $n=1$. Tad A ir *iteratīvais* teikums.

3. Lai daļas A nav.

$B_1, B_2, \dots, B_n ? n \geq 0$

Tad ir noformulētais *jautājums*.

Piezīme: šajā gadījumā var runāt par *konjunktīvu* jautājumu.

Faktu piemēri

```
int_num(1) .
```

```
Int_num(2) .
```

```
INT_num(3) .
```

Likuma piemērs

```
odd(X) :- Y = X mod 2, Y <> 0.
```

Jautājuma piemērs

```
?- odd(2)    %No
```

```
?- odd(3)    %Yes
```

```
?- odd(-3)   %Yes
```

Jēdzieni `int_num(...)`, `odd(...)` ir *termi*.

Termi bez mainīgajiem ir *pamat* termi.

Termi ar mainīgajiem ir *nepamat* termi.

Teikuma sastāvdaļas:

1. Galva.
2. Ķermenis

Faktiem ir tikai galva.

Likumiem ir galva un netukšs ķermenis.

Jautājumiem ir tikai ķermenis.

Jautājums sistēmai ir *mērķu secība*.

Vienkāršs jautājums sastāv no viena mērķa.

Pamatā ir *mērķa sasniegšana*.

Sistēma uztver faktus un likumus kā *aksiomu kopu*.

Lietotāja jautājums ir *teorēma*.

Programma mēģina *pierādīt* šo teorēmu – parādīt, ka teorēmu var *loģiski iegūt* no aksiomām.

Citiem vārdiem sakot, sasniegt mērķi – tas nozīmē parādīt, ka mērķis loģiski seko no programmas faktiem un likumiem.

Teorēmas pierādīšanas piemērs:

Ir divas **aksiomas**:

1. *Visi cilvēki ir mirstīgi.*
2. *Sokrāts - cilvēks.*

Teorēma:

Sokrāts ir mirstīgs

Pirmās aksiomas formalizācija:

$mirstigs(X) :- cilveks(X).$

Predikātu noformēšana:

```
%Visi cilvēki - mirstīgie  
mirstigs(X) :- cilveks(X) . %likums  
  
%Sokrāts - cilvēks  
cilveks(sokrats) . %fakts
```

Jautājums sistēmai:

```
?-mirstigs(sokrats) . %jautājums
```

Sistēmas atbilde:

```
Yes (Jā)
```

Jautājumi līdzīgi pieprasījumiem kādai datubāzei.

Prolog automātiski izpilda:

- dažādu variantu pētīšanu;
 - atgriešanās;
 - loģisko secinājumu.
-

Lai ir teikums:

$C :- A, B.$

Teikumam ir deklaratīvā un procedurālā jēga

Iespējamās *deklaratīvas* interpretācijas:

C ir patiess, ja **A** un **B** ir patiesi.

No **A** un **B** seko **C**.

Iespējamā *procedurālā* interpretācija:

Lai sasniegtu mērķi **C**, no sākuma jāsasniedz **A**, bet pēc tam – **B**.

Loģisko operāciju apzīmējumi:

and => , % (komats)

or => ; % (semikols)

Lai cilvēks ir students *vai* pasniedzējs.

`student(janis) .`

`teacher(juris) .`

`human(X) :- student(X) ; teacher(X) .`

Jautājums:

Goal: `human(X)`

Rezultāti:

`X=janis`

`X=juris`

2 Solutions

To pašu rezultātu var dabūt citas deklarācijas gadījumā:

`human(X) :- student(X) .`

`human(X) :- teacher(X) .`

Inicializētie un neinicializētie mainīgie

Ir *divi* mainīgie X un Y.

goal

X=2, Y=X+1, write(Y) . %3

Ir *viens* mainīgais X.

goal

X=2, X=X+1, write(X) . %nav rezultāta

Secinājums: operators = nodrošina kā piešķiri, tā arī vienādības pārbaudi.

Tas atkarīgs no mainīgā inicializācijas.

Prolog datu tipi:

1. **symbol** % prolog, "prolog", "Prolog"
2. **string** % "prolog", "Prolog"
3. **char** % 'p', 'P'
4. **integer** % 2
5. **real** % 2.5
6. **file** % file = f

X, **_x**, **_X** – mainīgais (pirmais simbols *augšējā* reģistrā vai pasvītrotā)

x – konstante (pirmais simbols *apakšējā* reģistrā)

Prolog programmas struktūra:

1. *Populāras* daļas:

domains % datu tipu definēšana
predicates % predikātu deklarēšana
clauses % faktu un likumu definēšana

2. *Iespējamās* daļas lielos projektos:

global domains
global predicates

3. *Dinamiskie fakti*:

database

4. *Programmas palaišana pakešu režīmā*:

goal

Programmas daļu secība:

domains

global domains

database

predicates

global predicates

goal

clauses

Bez **goal** būs nepieciešama *jautājumu ievade*.

Ar **goal** norāda *mērķu secību*.

Elementāra programma (*viens* predikāts):

predicates

H_W

goal

H_W.

clauses

H_W :- write("Hello, World !").

Elementāra programma (*divi* predikāti):

predicates

H W

goal

H, W.

clauses

H :- write("Hello, ").

W :- write(" World !").

Piemērs ar Sokrātu:

domains

predicates

`cilveks(symbol)`

`mirstigs(symbol)`

clauses

`cilveks(sokrats) .`

`mirstigs(X) :- cilveks(X) .`

Piezīme: ieteicams vienmēr izmantot daļu **domains**.

```
== Dialog ==  
Goal: mirstigs(sokrats)  
Yes  
Goal:
```


Veselu skaitļu ievade un kvadrātu izskaitļošana

domains

c = char

predicates

Dialog

Test(c)

clauses

Test('N') :- nl, write("Good Bye !").

Test(_) :- Dialog.

Dialog :- nl, write("Enter number: "),
readint(X), Y=X*X,
write("Square: ", Y), nl,
write("Continue (Y/N) ?"),
readchar(C), Test(C).

Eksistē *universālie* fakti:

`mul(X, 0, 0) . %X*0 = 0`

`mul(0, X, 0) . %0*X = 0`

Apgalvojums ir spēkā *visiem* X .

Iespējamā problēma: mainīgo X izmanto tikai vienu reizi. Rezultāts: **kompilācijas kļūda**.

Problēmas risinājums *Turbo Prolog* vidē:

Options → Compiler directives →
Variable used once warning **Off**.

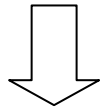
Universālais problēmas risinājums:

`mul(_, 0, 0) .`

`mul(0, _, 0) .`

Anonīmie mainīgie: vai **kādam** ir bērni?

```
has_child(X) :- parent(X, Y) .
```



```
has_child(X) :- parent(X, _) .
```

Tagad Y vietā ir *anonīmais mainīgais* `_`.

Var eksistēt vairāki anonīmie mainīgie:

```
some_has_child :- parent(_, _)
```

Ekvivalents:

```
some_has_children :- parent(X, Y)
```

```
some_has_children :- parent(X, X)
```

Mainīgā vārda leksiskais diapazons: viens teikums.

Ja kāda mainīgā vārds X eksistē divos teikumos, runa ir par diviem dažādiem X .

predicates

```
neg(integer, integer)  
square(integer, integer)
```

clauses

```
neg(X, Y) :- Y = -X.  
square(X, Y) :- Y = X*X.
```

Predikātos $\text{neg}(X, Y)$ un $\text{square}(X, Y)$ ir pilnīgi neatkarīgie mainīgie X un Y .

Lai ir ievadīts pozitīvais skaitlis N .

Izvadīt skaitļus $0, 1, \dots, N-1$.

Risinājumā izmanto rekursiju un steku.

predicates

$n(\text{integer})$

clauses

$n(0) .$

$n(X) :- Y = X - 1, n(Y), \text{write}(Y, " ").$

Rezultāti.

Goal: $n(3)$

0 1 2 Yes

Goal: $n(0)$

Yes

Datubāze “vecāki”:

domains

persona=**symbol**

predicates

parent(persona, persona)

clauses

parent(janis, uldis).

parent(janis, martins).

parent(ilze, uldis).

parent(ilze, martins).

parent(uldis, "Peteris").

parent("Peteris", linda).

Piezīme: “*Peteris*” un *peteris* nav viens un tāds pats cilvēks.

Informācija par *visiem vecākiem*:

Goal: parent(X,)

Rezultāti:

X=janis

X=janis

X=ilze

X=ilze

X=uldis

X=Peteris

6 Solutions

Piezīme: tas ir *vienkāršs* jautājums (sastāv no viena mērķa).

Informācija par *visiem bērniem*:

Goal: parent(, X)

Rezultāti:

X=uldis

X=martins

X=uldis

X=martins

X=Peteris

X=linda

6 Solutions

Informācija par *Jāņa bērniem*:

Goal: parent(janis, X)

X=uldis

X=martins

2 Solutions

Informācija par *Martina vecākiem*:

Goal: parent(X, martins)

X=janis

X=ilze

2 Solutions

Pārbaude: vai *Jānis ir Martina tēvs*?

Goal: parent(janis, martins)

Yes

Informācija par *Lindas vecāku vecākiem* (vectēvu un vecmāmiņu)

Kurš ir viens no Lindas vecākiem ? Lai viņš ir kāds **Y**.

Kurš ir viens no **Y** vecākiem ? Lai viņš ir kāds **X**.

Goal: parent(Y, linda), parent(X, Y)

Y=Peteris, X=uldis

1 Solution

Ja izmainīt mērķu secību, rezultāts neizmainīsies:

Goal: parent(X, Y), parent(Y, linda)

X=uldis, Y=Peteris

1 Solution

Var izveidot attiecību `parent_of_parent`:

predicates

...

`parent_of_parent(symbol, symbol)`

clauses

...

`parent_of_parent(X, Z) :- parent(X, Y),
parent(Y, Z).`

Goal: `parent_of_parent(X, linda)`

`X=uldis`

1 Solution

Problēma: kā atrast *visus* priekštečus (senčus)?

Predikāta `ancestor(X, Y)` izveidošana:

1. Visiem **X** un **Z** :

X ir **Z** priekštecis, ja **X** ir viens no **Z** vecākiem.

```
ancestor(X, Z) :- parent(X, Z) .
```

2. Rekursija:

```
ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z) .
```

Goal: `ancestor(X, linda)`

`X=Peteris`

`X=janis`

`X=ilze`

`X=uldis`

4 Solutions

Lai ir divas papildu attiecības dzimuma norādīšanai:
male(X) un *female(X)*.

predicates

...

`male(symbol)`

`female(symbol)`

clauses

...

`male(janis) .`

`male(uldis) .`

`male(martins) .`

`male("Peteris") .`

`female(ilze) .`

`female(linda) .`

Loģiska attiecība “*māte*”:

Visiem **X** un **Y**: **X** ir **Y** māte,

Ja **X** ir viens no **Y** vecākiem,

Un **X** ir sieviete

```
mother(persona, persona)
```

```
...
```

```
mother(X, Y) :- parent(X, Y), female(X).
```

Goal: mother(X, uldis)

X=ilze

1 Solution

Loģiska attiecība “*brālis*”:

Visiem X un Y : X ir Y brālis,

Ja X_{am} un Y_{am} viens no vecākiem ir kopīgs,

Un X ir vīrietis.

```
brother(X, Y) :- parent(Z, X), parent(Z, Y),  
    male(X). %neveiksmīga realizācija
```

```
brother(X, Y) :- parent(Z1, X), parent(Z2, Y),  
    Z1=Z2, male(X). %arī neveiksmīga realizācija
```

Rezultāti abos gadījumos sakrīt:

Goal: brother(X, martins)

X=uldis

X=martins

X=uldis

X=martins

4 Solutions

```
brother(X, Y) :- parent(Z, X), parent(Z, Y),  
    X<>Y, male(X).
```

Goal: brother(X, martins)

X=uldis

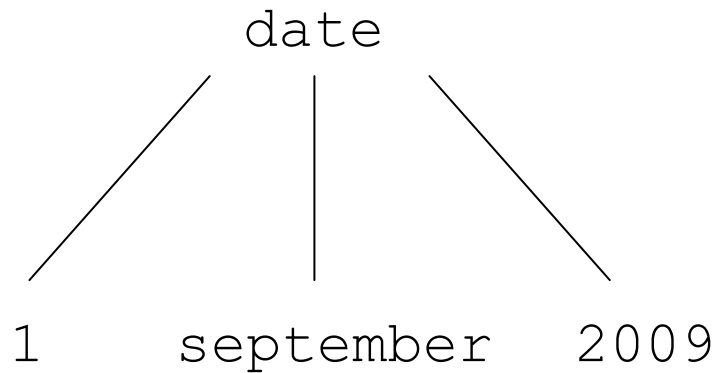
X=uldis

2 Solutions

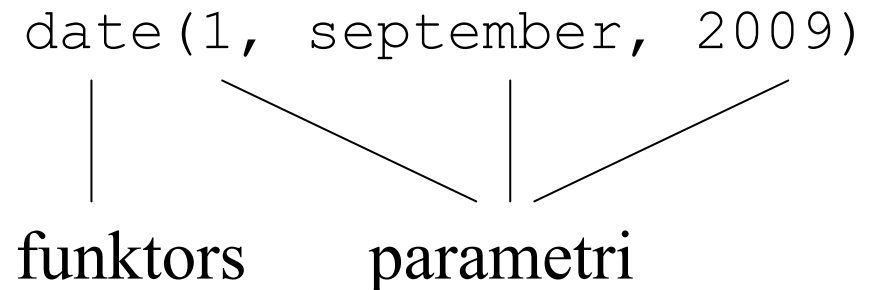
Struktūras

`date(1, september, 2009)`

Struktūrai ir divas interpretācijas:

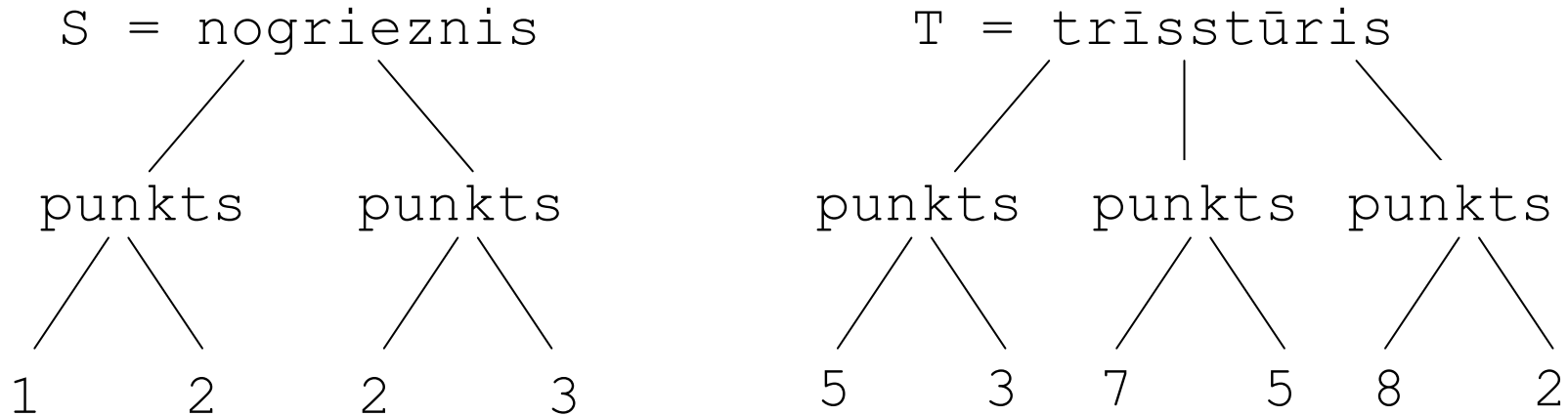


Koks



Prolog - interpretācija

Struktūras izmantošana ģeometriskos objektos:



domains

p = point(**integer**, **integer**)

predicates

seq(p, p)

triangle(p, p, p)

clauses

seq(point(1, 2), point(2, 3)).

triangle(point(5, 3), point(7, 5), point(8, 2)).

Informācija par nogriežņa *virsotnēm*

Goal: seq(X, Y)

X=point(1,2), Y=point(2, 3)

1 Solution

Informācija par pirmās virsotnes *koordinātēm*

Goal: seq(P1,), P1 = point(X, Y)

P1=point(1,2), X=1, Y=2

1 Solution

Piezīme: **nepareizi** būs:

~~**Goal:** seq(P1,), point(X, Y)=P1~~

Vertikāla nogriežņa noteikšana: virsotņu abscisas sakrīt.

domains

s = seq(p, p)

predicates

ver(s)

clauses

ver(seq(point(X, Y1), point(X, Y2))).

Goal: ver(seq(point(3, 1), point(3, 3)))

Yes

Goal: ver(seq(point(2, 1), point(3, 3)))

No

Goal: ver(seq(point(3, 1), point(3, 1)))

Yes %bet tas ir punkts

Labāk būs:

ver(seq(point(X, Y1), point(X, Y2))) :- Y1<>Y2.

Saraksti

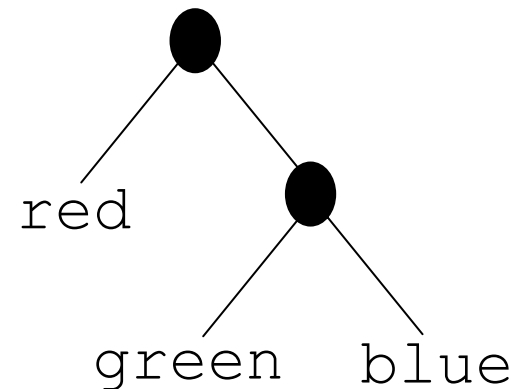
Saraksts ir vienkāršā datu struktūra, kuru plaši izmanto *neskaitliskajā* programmēšanā.

Saraksts ir *elementu secība*.

Lai ir kineskopa pamatkrāsas: red, green, blue. Saraksts:
`[red, green, blue]`

PROLOG

iekšēja interpretācija:



Ir divi gadījumi:

1. *Tukšais* saraksts.
2. *Netukšais* saraksts.

Otrajā gadījumā sarakstu bieži uztver kā struktūru no divām daļām:

1. *Pirmais* elements (-ti). Saraksta *galva*.
2. *Pārēja* saraksta daļa. Saraksta *aste*.

Saraksta aste – obligāti *saraksts*. Piemērs ar krāsām:

```
[red, green, blue]
```

```
[red | [green, blue]]
```

```
[red, green | [blue]]
```

```
[red, green, blue|[]]
```

Saraksta izmantošana programmā:

domains

list=**symbol***

predicates

L(list)

clauses

L([red, green, blue]).

Goal: L(X)

X=["red","green","blue"]

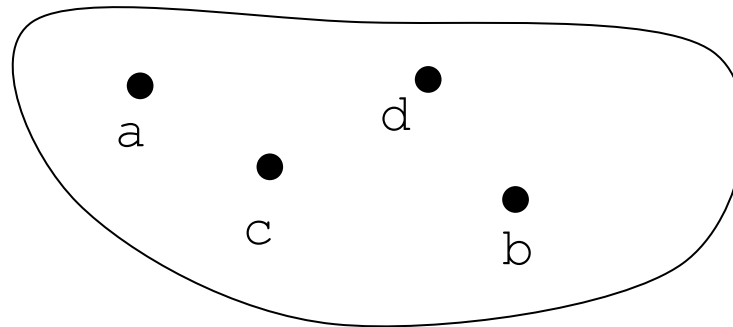
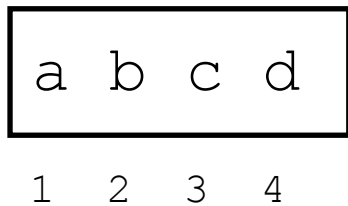
1 Solution

Šis rezultāts sakrīt *visiem četriem* iepriekšējiem saraksta deklarācijas variantiem.

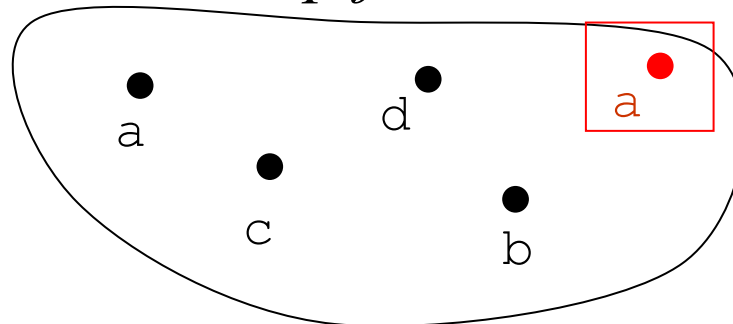
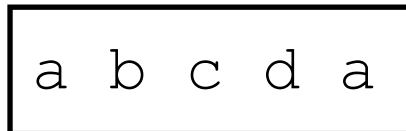
Sarakstus bieži izmanto kopu pārstāvēšanai.

Atšķirības starp *sarakstiem* un *kopām*:

1. Sarakstā elementiem ir noteiktā secība.



2. Sarakstā var eksistēt *elementu kopijas*.



Saraksta izvade abos virzienos

domains

```
list = symbol*
```

predicates

```
pr_frw(list)
```

```
pr_bck(list)
```

clauses

```
pr_frw([X|T]) :- write(X, " "), pr_frw(T).
```

```
pr_frw([]).
```

```
pr_bck([X|T]) :- pr_bck(T), write(X, " ").
```

```
pr_bck([]).
```

Goal: pr_frw([a, b, c])

a b c Yes

Goal: pr_bck([a, b, c])

c b a Yes

Faktu saraksts

domains

h=human (**symbol**)

list=h*

predicates

L(list)

pr(list)

clauses

L([human(uldis), human(ivars)]).

pr([]).

pr([H|T]) :- H=human(X), write(X, " "),
pr(T).

goal

L(X), pr(X).

Rezultāts:

uldis ivars

1. Elementa *piederība* sarakstam

domains

```
list=symbol*
```

predicates

```
member(symbol, list)
```

clauses

```
member(X, [X|T]) .
```

```
member(X, [H|T]) :-member(X, T) .
```

Goal: member(red, [red, green, blue])

Yes

Goal: member(green, [red, green, blue])

Yes

Goal: member(orange, [red, green, blue])

No

2. Jauna elementa *pievienošana*.

Pievienošanas vieta *nav noteikta*.

Jaunu elementu visvieglāk pievienot *saraksta sākumā*.

Attiecīgais predikāts ir *fakts*.

domains

list=**symbol***

predicates

append(symbol, list, list)

clauses

append(X, L, [X|L]).

Goal: append(red, [green, blue], L)

L=["red", "green", "blue"]

3. Elementa *izslēgšana*.

Lai **X** ir izslēdzamais elements. Ir divi gadījumi:

1. **X** ir saraksta galva. Rezultāts: saraksta aste.
 2. **X** atrodas saraksta astē. Tad jāizslēdz **X** no saraksta astes.
-

domains

```
list=symbol*
```

predicates

```
delete(symbol, list, list)
```

clauses

```
delete(X, [X|T], T).
```

```
delete(X, [Y|T], [Y|T1]) :- delete(X, T, T1).
```

Goal: delete(red, [red, green, blue], L)

L=["green", "blue"]

Goal: delete(orange, [red, green, blue], L)

No Solution

Elementa *piederības pārbaude* ar izslēgšanas palīdzību.

Elements **X** pieder sarakstam **L**, ja elementu var izslēgt no saraksta.

```
member(X, L) :- delete(X, L, _).
```

Elementa *ielikšana* ar izslēgšanas palīdzību.

```
insert(X, List, BiggerList) :-  
    delete(X, BiggerList, List).
```

Goal: insert(a, [b, c], X)

X=["a", "b", "c"]

X=["b", "a", "c"]

X=["b", "c", "a"]

3 Solutions

Elementa dzēšana *visā sarakstā*.

1. Elements X ir dzēšamais elements.

`delete(X, [X|T], T1) :- delete(X, T, T1).`

Procedurālā pieeja: rekursīvi izdzēst X no $[X | T]$.

Deklaratīvā pieeja: X dzēšana no $[X | T]$ noved pie $T1$, ja X dzēšana no T noved pie $T1$.

2. Elements X nav dzēšamais elements.

`delete(X, [Y|T], [Y|T1]) :- X<>Y,
delete(X, T, T1).`

Procedurālā pieeja: rezultāts ir saraksts ar galvu Y un asti ar rekursīvi izdzēstu elementu.

3. No tukšā saraksta neko nevar izdzēst.

```
delete(_, [], []).
```

Goal: delete(a, [a,b,a,a,b,a], L)

L=["b", "b"]

1 Solution

Lai 2. likumā *nav* papildu pārbaudes $X \neq Y$.

Goal: delete(a, [a,b,a,b], L)

L=["b", "b"]

L=["b", "a", "b"]

L=["a", "b", "b"]

L=["a", "b", "a", "b"]

4 Solutions

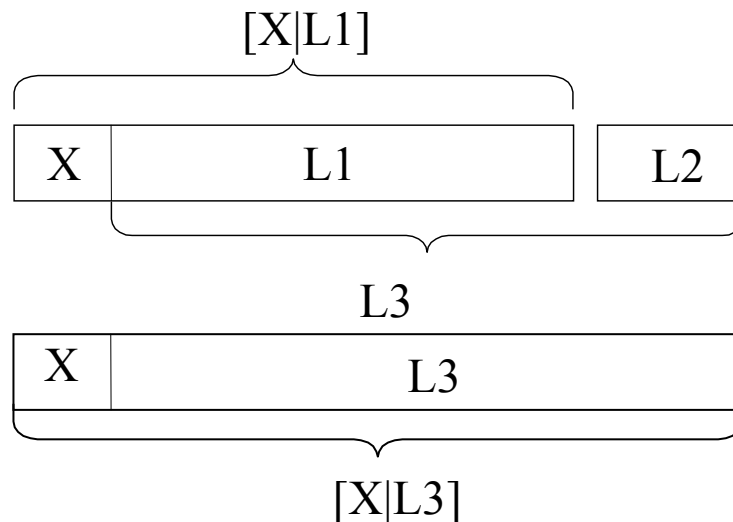
Sarakstu konkatēnācija

`concat_lists(L1, L2, L3)`

1. Ja pirmais saraksts ir *tukšs* saraksts, otrais un trešais saraksti sakrīt.

`concat_lists([], L, L) .`

2. Pirmām parametram ir *galva* un *aste*: `[X | L1]`.



```
concat_lists([X|L1], L2, [X|L3]) :-  
    concat_lists(L1, L2, L3).
```

Rezultāts:

domains

list = **symbol***

predicates

concat_lists(list, list, list)

clauses

concat_lists([], L, L).

concat_lists([X|L1], L2, [X|L3]) :-
 concat_lists(L1, L2, L3).

Goal: concat_lists([a], [b, c], B)
B=["a", "b", "c"]

Saraksta *dekompozīcija* ar konkatēnācijas palīdzību:

Goal: `concat_lists(X, Y, [a, b, c])`

`X=[], Y=["a", "b", "c"]`

`X=["a"], Y=["b", "c"]`

`X=["a", "b"], Y=["c"]`

`X=["a", "b", "c"], Y=[]`

4 Solutions

Elementu meklēšana: atrast visus mēnešus *pirms* un *pēc* norādītā mēneša (maija).

Goal: `concat_lists(Before, [may|After], [january, february, ..., december])`

`Before=["january", "february", "march", "april"]`

`After=["june", "july", ..., "december"]`

Elementu meklēšana: atrast *divus konkrētus* mēnešus *pirms* un *pēc* norādītā mēneša (maija).

Goal: `concat_lists(_,
[Month_before, may, Month_after|_],
[january, february, ..., december])`

`Month_before=april, Month_after=june`

1 Solution

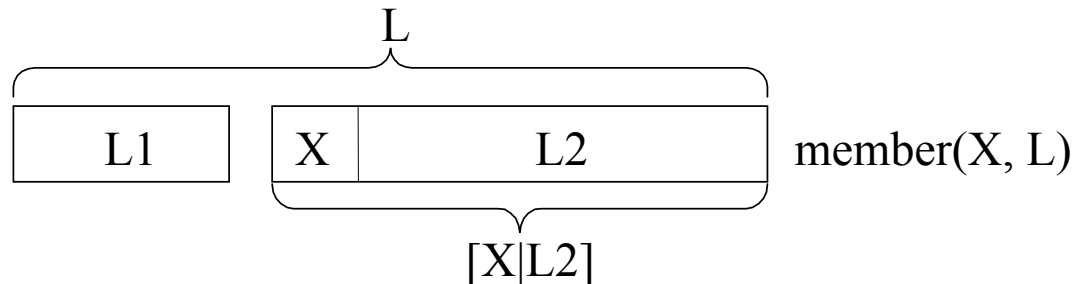
Elementu izslēgšana pēc noteiktas kombinācijas (`x`, `x`)

Goal: `concat_lists(L, [x, x|_],
[a, x, b, x, x, c, d])`

`L=["a", "x", "b"]`

1 Solution

Piederības pārbaude ar konkatēnācijas palīdzību



X pieder sarakstam L , ja sarakstu L var sadalīt divos sarakstos tā, lai X būtu otra saraksta galva.

```
member(X, L) :- concat_lists(L1, [X|L2], L).
```

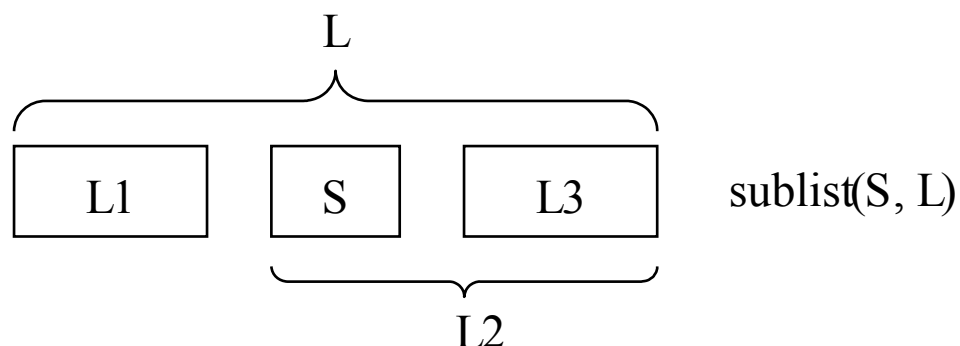
Var izmantot anonīmos mainīgos:

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

Sakārtotais apakšsaraksts

```
sublist([b, c], [a, b, c, d]) -> true
```

```
sublist([b, d], [a, b, c, d]) -> false
```



1. L var sadalīt divos sarakstos: L1 un L2

2. L2 var sadalīt divos sarakstos: S un L3

```
sublist(S, L) :- concat_lists(L1, L2, L),
                  concat_lists(S, L3, L2).
```

Visu apakšsarakstu meklēšana:

Goal: `sublist(S, [a, b, c])`

`S = []`

`S = ["a"]`

`S = ["a", "b"]`

`S = ["a", "b", "c"]`

`S = []`

`S = ["b"]`

`S = ["b", "c"]`

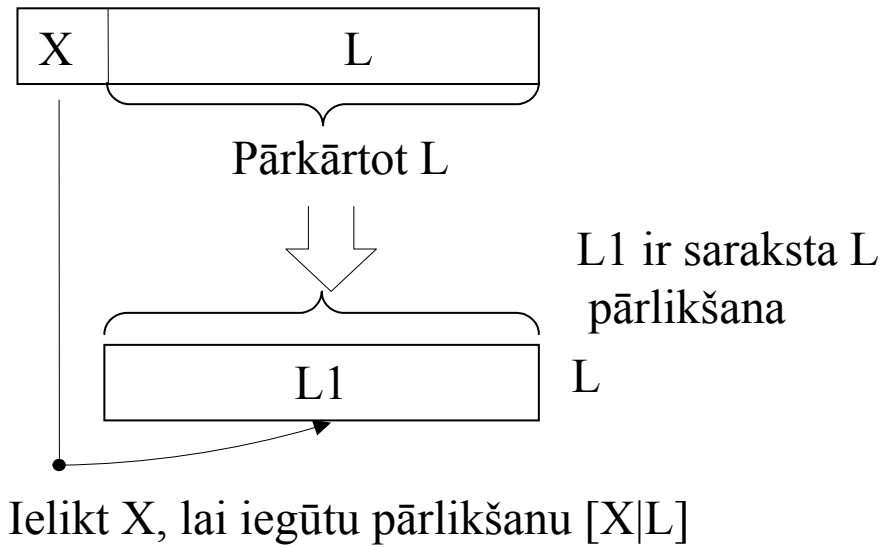
`S = []`

`S = ["c"]`

`S = []`

10 Solutions

Pārlikšanas



```
permutate([], []).
```

```
permutate([X|L], P) :- permutate(L, L1),
    insert(X, L1, P).
```

```
insert(X, List, BiggerList) :-
    delete(X, BiggerList, List). %no konspekta
```


Goal: `permutate([a, b, c], X)`

`X=["a", "b", "c"]`

`X=["b", "a", "c"]`

`X=["b", "c", "a"]`

`X=["a", "c", "b"]`

`X=["c", "a", "b"]`

`X=["c", "b", "a"]`

6 Solutions

Pārlikšanas var arī izpildīt ar *dzēšanas* palīdzību:

`permutate([], []).`

`permutate(L, [X|P]) :- delete(X, L, L1),
permutate(L1, P). %delete - no konspekta`

Unikālā elementa pievienošana:

```
append_unique(X, L, L1)
```

1. Ja X pieder sarakstam L , tad $L1 = L$.
 2. Pretējā gadījumā $L1$ ir saraksts L ar pievienotu elementu X .
-

Visvienkāršāk pievienot elementu X saraksta L sākumā.

Viena no predikāta realizācijām ir `append` no konspekta.

```
append_unique(X, L, L) :- member(X, L), !.
```

```
append_unique(X, L, [X|L]).
```

Atciršanas

Programmētājs var regulēt aprēķināšanas procesu programmā.

1. Izvietot predikātus *konkrētajā secībā*.
2. Pielietot atciršanas, lai ierobežotu *automātisko pārmeklēšanu*.

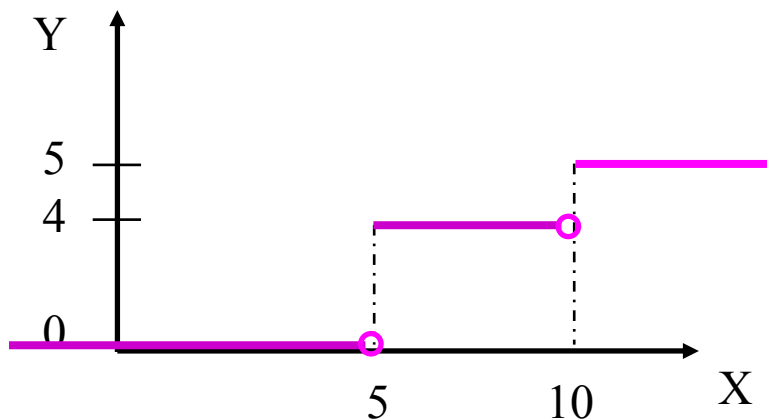
PROLOG sistēmas pārmeklēšana atbrīvo programmētāja no lietišķās programmēšanas.

Neierobežotā pārmeklēšana var samazināt programmas efektivitāti.

Lai ir trīs saistītie likumi:

1. likums: **Ja** $X < 5$
 Tad $Y = 0$
2. likums: **Ja** $5 \leq X < 10$
 Tad $Y = 4$
3. likums: **Ja** $X \geq 10$
 Tad $Y = 5$

Likumu grafiskā interpretācija:



Likumu programmēšana PROLOG valodā (**1.** versija):

$f(X, 0) \text{ :- } X < 5.$

$f(X, 4) \text{ :- } 5 \leq X, X < 10.$

$f(X, 5) \text{ :- } X \geq 10.$

Rezultāti:

Goal: $f(4, Y)$

$Y=0$

1 Solution

Goal: $f(6, Y)$

$Y=4$

1 Solution

Ir redzams, ka visi trīs likumi ir *savstarpēji izslēdzošie*.

Ja tika atrasta kāda atbilde, pārējas alternatīvas *var neapstrādāt*.

Likumu programmēšana PROLOG valodā (2. versija):

$f(X, 0) :- X < 5, !.$

$f(X, 4) :- 5 \leq X, X < 10, !.$

$f(X, 5) :- X \geq 10, !.$

Tagad programma strādās *efektīvāk* – nevajag apstrādāt “bezzēdzīgas” alternatīvas.

Ja *izmest* jebkuru atciršanu, rezultāts *neizmainīsies*.

Ir vēl viena iespēja palielināt programmas ātrdarbību.

Lai ir jautājums:

$f(15, Y)$.

1. Mēģinājums pielietot 1. likumu. $15 < 5$, neveiksme.

Notiek atgriešanās un mēģinājums pielietot 2. likumu, jo atciršanas punkts netika sasniegts.

2. Mēģinājums pielietot 2. likumu. $5 \leq 15$ ir veiksmē, bet $15 < 10$ neveiksme.

Notiek atgriešanās un mēģinājums pielietot 3. likumu (atciršanas punkts atkal nav sasniegts).

3. Mēģinājums pielietot 3. likumu. $10 \leq 15$ - veiksmē.

Problēma: vienu un to pašu nosacījumu pārbauda *divas reizes*.

Ir izskaidrots, ka $X < 5$ nav patiess. Nākošais mērķis: $5 \leq X$ ($5 \leq 15$) obligāti tiks sasniegts.

Otrais apgalvojums ir pirmā apgalvojuma *noliegums*.

Līdzīga situācija ir ar mērķiem $X < 10$ un $X \geq 10$.

Tagad ir likumi **if-then-else**:

Ja $X < 5$

Tad $Y = 0$

Citādi Ja $X < 10$

Tad $Y = 4$

Citādi $Y = 5$

Likumu programmēšana PROLOG valodā (**3.** versija):

```
f(X, 0) :- X < 5, !.
f(X, 4) :- X < 10, !.
f(X, 5).
```

Programma tagad strādā *maksimāli ātri*.

Nav iespējams *atteikties* no atciršanām.

Nav iespējams *izmainīt* teikumu kārtību.

Lai ir mērķi:

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- A, B, C.
```

Pēc atciršanas sasniegšanas alternatīva $C :- V$ netiks
apstrādāta.

Atciršanu formālā interpretācija:

$H :- B_1, B_2, \dots, B_m, !, \dots, B_n.$

Lai teikums aktivizēties, kad kāds mērķis G tiks saskaņots ar H .

Atciršanas sasniegšanas brīdī mērķiem B_1, \dots, B_m jau ir veiksmē.

Iegūtais risinājums tiks “*iesaldēts*”.

Nav iespējas saskaņot G ar kādu citu teikumu.

Piederības pārbaudes optimizācija

```
member(X, [X|T]) :- !.
```

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

Maksimālās vērtības izrēķināšana

```
max(X, Y, Max)
```

Variants *bez* atciršanām:

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

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

Variants *ar* atciršanām (abas alternatīvas ir savstarpēji izslēdzošas):

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

```
max(X, Y, Y).
```

Selektīva operatora `switch` imitēšana ar atciršanu palīdzību:
elementāras aritmētiskas operācijas ar diviem skaitļiem.

predicates

```
cycle  
do(integer, integer, char, real)
```

clauses

```
cycle :- write("A = "), readint(A),  
         write("B = "), readint(B),  
         write("Operation ?"), readchar(C),  
         do(A, B, C, Rez), nl,  
         write("Rezult: ", Rez), nl,  
         write("C - continue"), readchar(X),  
         upper_lower(Y, X), Y = 'C', nl, cycle.  
cycle :- true. %Var arī vienkārši: cycle
```

Selektīva operatora `switch` imitēšana (turpinājums)

```
do(A, B, '+', Rez) :- Rez = A + B, !.  
do(A, B, '-', Rez) :- Rez = A - B, !.  
do(A, B, '*', Rez) :- Rez = A * B, !.  
do(A, B, '/', Rez) :- Rez = A / B, !.  
do(_ , _ , _ , Rez) :- nl,  
    write("Illegal operation !"), Rez = 0.0.
```

goal

```
cycle, write("\nGood Bye !").
```

Piezīme: ja nebūs otrās `cycle` realizācijas, ziņojums "Good Bye" *netiks izdrukāts*.

Objektu *klasifikācijas* uzdevums

Lai ir datu bāze par kāda turnīra šahu partijām.

Katrs dalībnieks spēlēja *vismaz vienu* partiju.

Spēļu rezultāti pārstāvēti programmā *kā fakti*:

uzvareja(janis, uldis) .

uzvareja(juris, peteris) .

uzvareja(aldis, juris) .

Jādefinē attiecība klase (<dalībnieks>, <kategorija>)

Uzvarētājs - uzvarēja *visās* partijās.

Profesionāls - *kādās* partijās uzvarēja,
bet *kādās* zaudēja.

Sportists - zaudēja *visās* partijās.

Mūsu piemērā:

Uzvarētājs: *janis, aldis.*

Profesionāls: *juris.*

Sportists: *uldis, peteris.*

Profesionāla definīcija:

X ir profesionāls, ja

Eksistē kāds **Y**, tāds ka **X** uzvarēja **Y**

Un eksistē kāds **Z**, tāds, ka **Z** uzvarēja **X**.

Uzvarētāja definīcija:

X ir uzvarētājs, ja

X uzvarēja kāda **Y**

Un **X** nebija uzvarēts.

Problēma: *uzvarētāja* definīcija satur noliegumu **nē**.

Līdzīga situācija ir ar *sportista* definīciju.

Ne visas *Prolog* versijas satur noliegumu.

Nolieguma problēmu var apiet, izmantojot attieksmi “*citādi*”.

Ja **X** uzvarēja kāda dalībnieka un **X** kādreiz bija uzvarēts

Tad **X** ir profesionāls

Citādi

Ja **X** uzvarēja kāda dalībnieka

Tad **X** ir uzvarētājs

Citādi

Ja **X** kādreiz bija uzvarēts

Tad **X** ir sportists

Prolog risinājums:

```
klase(X, professionals) :-  
    uzvareja(X, _), uzvareja(_, X), !.  
  
klase(X, uzvaretais) :- uzvareja(X, _), !.  
  
klase(X, sportists) :- uzvareja(_, X).
```

Turbo Prolog atbalsta noliegumu – ir speciāls predikāts **not**, kurš nepilnīgi atbilst Būla loģikai.

Vairākās citās *Prolog* versijās var patstāvīgi uzrakstīt nolieguma predikātu.

Noliegums kā neveiksme

Lai ir izteiksme: “Ilzei patīk visi dzīvnieki, izņemot žurkas”.

Ja X ir žurka

Tad izteiksme “Ilzei patīk X” nav patiesa.

Citādi Ja X ir dzīvnieks, tad Ilzei patīk X.

```
patik(ilze, X) :- zurka(X), !, fail.
```

```
patik(ilze, X) :- dzivnieks(X).
```

Programmas kodu var saīsināt

```
patik(ilze, X) :- zurka(X), !, fail; dzivnieks(X).
```

X un Y nesakritības pārbaude

```
atskiras(X, X) :- !, fail.  
atskiras(X, Y) .
```

Var apvienot abas realizācijas:

```
atskiras(X, Y) :- X=Y, !, fail; true.
```

Var patstāvīgi definēt unāru predikātu **not**.

Ja Mērķis ir veiksmīgs

Tad not (Mērķis) nav veiksmīgs

Citādi not (Mērķis) ir veiksmīgs

Predikāta realizācija

```
not(P) :- P, !, fail; true.
```

Klasifikācijas uzdevuma realizācija *bez atciršanās*

```
klase(X, professionals) :- uzvareja(X, _),  
    uzvareja(_, X).
```

```
klase(X, uzvaretais) :- uzvareja(X, _),  
    not(uzvareja(_, X)).
```

```
klase(X, sportists) :- uzvareja(_, X),  
    not(uzvareja(X, _)).
```

Atciršanu galvenā doma: skaidri pateikt Prolog-sistēmai – nevajag strādāt ar pārējām alternatīvām, tās tik un tā būs neveiksmīgas.

Atciršanas atļauj viegli programmēt likumus:

Ja -> Tad -> Citādi.

Atciršanu izmantošanas problēma

Ja programmā ir atciršanas, tad izmaiņas teikumu kārtībā var *ietekmēt deklaratīvo jēgu*.

Programmas ar izmainītu teikumu kārtību rezultāti var atšķirties no sākotnējas programmas.

Lai ir divi teikumi:

$p :- a, b.$

$p :- c.$

Teikumiem atbilst formula:

$p \iff (a \ \& \ b) \cup c$

Lai vienā teikumā ir atciršana

$p :- a, !, b.$

$p :- c.$

Teikuma interpretācija:

$p \iff (a \ \& \ b) \cup (\sim a \ \& \ c)$

Izmainīsim teikumu kārtību

$p :- c.$

$p :- a, !, b.$

Teikuma interpretācija:

$p \iff c \cup (a \ \& \ b)$

Atciršanu klasifikācija

1. *Zaļās atciršanas.*

Nemaina programmas deklarātīvo stilu.

Lasot programmu, tādas atciršanas var “ignorēt”.

2. *Sarkanās atciršanas.*

Ietekmē programmas deklarātīvo stilu.

Apgrūtina programmas saprašanu.

Atciršanu bieži izmanto kombinācijā ar speciālo mērķi **fail**.

Ieteicams aizvietot šo kombināciju ar predikātu **not**.

Noliegums, salīdzinājumā ar kombināciju “**!**, **fail**”, ir augstāka līmeņa jēdziens.

“Noslēgtas pasaules” koncepcija

Lai ir predikāts:

`not (cilveks (janis))`

Sistēma var atbildēt “jā”, bet tas vēl nenozīmē, ka *janis* nav cilvēks.

Sistēma ne mēģina tieši pierādīt šī mērķa patiesumu.

Sistēma mēģina pierādīt apgalvojumu `cilveks (janis) .`
Tikai ja tas nav iespējams, sistēma uzskata, ka mērķis `not` ir veiksmīgs.

Programmai *trūkst* informācijas, lai pierādītu apgalvojumu, ka Jānis ir cilvēks.

Atciršanas sarakstu kārtošanā: *burbuļa kārtošana*

domains

```
list = integer*
```

predicates

```
bubble(list, list)
```

```
transpose(list, list)
```

```
greater(integer, integer)
```

clauses

```
bubble(Start, Result) :-
```

```
    transpose(Start, New), !,
```

```
    bubble(New, Result).
```

```
bubble(Result, Result).
```

Burbuļa kārtošana (turpinājums)

```
transpose([X, Y | Tail], [Y, X | Tail]) :-  
    greater(X, Y).  
  
transpose([Z|Tail], [Z|Tail1]) :-  
    transpose(Tail, Tail1).  
  
greater(X, Y) :- X>Y.
```

Programmas izpildes piemērs

Goal: bubble([1, 4, 3, 2], L)

L=[1, 2, 3, 4]

1 Solution

Atciršanas sarakstu kārtošanā: *ielikšanu metode*

domains

```
list = integer*
```

predicates

```
insert_sort(list, list)
```

```
insert(integer, list, list)
```

```
greater(integer, integer)
```

clauses

```
insert_sort([], []).
```

```
insert_sort([X|T], Sorted_List) :-
```

```
    insert_sort(T, Sorted_T),
```

```
    insert(X, Sorted_T, Sorted_List).
```

Ielikšanu metode (turpinājums)

```
insert(X, [Y|Sorted_List],  
      [Y|Sorted_List1]) :- greater(X, Y),  
      !, insert(X, Sorted_List, Sorted_List1).  
insert(X, Sorted_List, [X|Sorted_List]).  
greater(X, Y) :- X>Y.
```

Programmas izpildes piemērs

Goal: insert_sort([1, 4, 2, 3], L)

L=[1, 2, 3, 4]

1 Solution

Saraksta lietošana faktos: *papildus jautājumi*

Lai ir informācija par pieaugušiem cilvēkiem formātā:

```
person(<vārds>, <uzvārds>, <bērnu saraksts>).
```

Bērnus apraksta formātā:

```
person(<vārds>, <uzvārds>).
```

Nepieciešams uzrakstīt predikātu, kas meklē cilvēkus *ar kopīgiem bērniem*:

```
family(<vārds1>, <vārds2>).
```

domains

```
p = person(symbol, symbol)
```

```
list=p*
```

predicates

```
person(symbol, symbol, list)
```

```
family(symbol, symbol)
```

clauses

```
person(uldis, strods,  
      [person(janis, strods),  
       person(liene, strode)]).
```

```
person(ilze, strode,  
      [person(janis, strods),  
       person(liene, strode)]).
```

```
family(X, Y) :- person(X,   , Z),  
                person(Y,   , Z), X<>Y.
```

Goal: family(X, Y)

X=uldis, Y=ilze

X=ilze, Y=uldis

2 Solutions

Faktu apstrāde un saraksta veidošana

Lai ir informācija par darbinieku algu:

```
alga(janis, 400.0) .  
alga(ivars, 500.0) .  
alga(ivans, 550.0) .  
alga(sergejs, 450.0) .
```

Uzdevums: atrast vidēju darbinieku algu.

Predikāts findall:

```
findall(<Mainīgais>,  
        <Predikāta izteiksme>, <Saraksts>)
```

Predikāta izmantošana problēmas risināšanai:

```
findall(Nauda, alga(_, Nauda),  
        <Algu saraksts>).
```

Summāras algas meklēšana:

```
sum_alga([], 0, 0).  
sum_alga([H|T], Sum, N) :-  
    sum_alga(T, Sum1, N1), Sum = Sum1 + H,  
    N = N1 + 1.
```

Vidējas algas meklēšana:

```
vid_alga :-  
    findall(Nauda, alga(_, Nauda), Algas),  
    sum_alga(Algas, SumAlga, Number),  
    AvgAlga = SumAlga/Number.
```


Teksta rindiņas

"ABC" vai "\65\66\67"

1. Teksta rindiņas garums:

```
str_len(String, Length)
```

predicates

s1 s2 s3

clauses

```
s1 :- L=4, str_len("Today", L),  
      write(L). %No  
s2 :- L=5, str_len("Today", L),  
      write(L). %5  
s3 :- str_len("Today", L),  
      write(L). %5
```

2. Teksta rindiņu konkatenācija:

```
concat(S1, S2, OutputS) .
```

```
S = ", World !",
```

```
concat("Hello", S, NewS), write(NewS) .
```

3. Rindiņas dalīšana divās daļās:

```
frontstr(N, SrcS, SubS1, SubS2)
```



Pirmie **N** simboli

Pārējie simboli

```
frontstr(6, "Turbo Prolog", S1, S2),
```

```
write(S1), write(S2) %Turbo Prolog
```

4. Pirmā simbola iegūšana:

`frontchar(String, Char, RestOfString)`

Pirmais simbols

Pārējie simboli

Daži predikāta izpildes rezultāti

`frontchar(S, 'P', "rolog")` %S=Prolog

`frontchar("Prolog", C, R)` %C=P, R=rolog

`frontchar("Prolog", C, "rolog")` %C=P

`frontchar("Prolog", C, "Turbo")`

%No Solution

`frontchar("Prolog", 'P', "Prolog")` %No

`frontchar("Prolog", 'P', "rolog")` %Yes

Teksta rindas pārveidošana simbolu sarakstā

domains

ch_l = **char***

predicates

str_to_chrs(**string**, ch_l)

clauses

str_to_chrs("", []).

str_to_chrs(Str, [H|T]) :-
 frontchar(Str, H, Str_1),
 str_to_chrs(Str_1, T).

Rezultāts

Goal: str_to_chrs("abc", S)

S=['a','b','c']

1 Solution

Datu tipu pārveidošana

1. `upper_lower(X, Y)`
 2. `str_char(X, Y)`
 3. `str_int(X, Y)`
 4. `str_real(X, Y)`
 5. `char_int(X, Y)`
-

Predikātu izmantošanas pamatprincipi.

predicates

`c1 c2`

clauses

```
c1 :- upper_lower(X, "ab"), write(X).%AB  
c2 :- upper_lower("AB", X), write(X).%ab
```

Predikātu izmantošanas pamatprincipi (turpinājums)

```
str_int("13", M)      %M=13
```

```
str_int("13", 13)     %Yes
```

```
str_int("14", 13)     %No
```

```
str_int(X, 13)        %X=13
```

Prolog speciālās rindas pārbaude (identifikators)

```
isname(String)
```

```
isname("Circle1")     %Yes
```

```
isname("1Circle")     %No
```

```
isname("Circle 1")    %No
```

Atomu iegūšana

```
fronttoken(String, Token, RestOfString)
```

Daži predikāta izpildes rezultāti

```
fronttoken("Turbo Prolog", X, Y)
```

```
%X="Turbo", Y=" Prolog"
```

```
fronttoken("User007", X, Y)
```

```
X=User007, Y=
```

```
fronttoken(Str, "Default", "Directory")
```

```
%Str=DefaultDirectory
```

```
fronttoken(Str, "$Default", "$Directory")
```

```
%Str=$Default$Directory
```

```
fronttoken("$Command", Token, Rest)
```

```
%Token=$, Rest=Command
```

Teksta rindas pārveidošana atomu sarakstā (fragments)

```
str_to_tokens(Str, [H|T]) :-  
    fronttoken(Str, H, Str1), !,  
    str_to_tokens(Str1, T).  
str_to_tokens(_, []).
```

Lai ir faktu saraksts:

```
cilveks(janis).  
cilveks(ivans).  
cilveks(uldis).  
cilveks(sergejs).
```

Uzdevums: iegūt faktu sarakstu

```
[cilveks(janis), cilveks(ivans),  
  cilveks(uldis), cilveks(sergejs)]
```


Sarežģītākais uzdevums: izveidot sarakstu, ņemot vērā *datu tipu*.

```
transform("janis ivans uldis", F)  
F=[s("janis"),s("ivans"),s("uldis")]
```

1 Solution

```
transform("janis 12", F)  
F=[s("janis"),n(12)]
```

1 Solution

Analizējamie datu tipi

s – string (teksta rinda)

n – number (skaitlis)

c – character (simbols)

Programma informācijas pārveidošanai

domains

```
tkn_type = n(integer) ; c(char) ;  
          s(string)  
tkn_list = tkn_type*
```

predicates

```
transform(string, tkn_list)  
choose_token(string, tkn_type)
```

clauses

```
transform("", []).  
transform(Str, [Tkns_H|Tkns_T]) :-  
    fronttoken(Str, Type, StrRest),  
    choose_token(Type, Tkns_H),  
    transform(StrRest, Tkns_T).
```

```
choose_token(S, n(N)) :- str_int(S, N).
choose_token(S, c(C)) :- str_char(S, C).
choose_token(S, s(S)) :- isname(S).
```

Piezīme: dažreiz programmai būs *vairāk nekā viens* rezultāts.

Goal: transform("A1 B", F)

F=[s("A1"),c('B')]

F=[s("A1"),s("B")]

2 Solutions

Lai viennozīmīgi interpretētu informāciju, jāizmanto atciršanas.

```
choose_token(S, n(N)) :- ..., !.
```

```
choose_token(S, c(C)) :- ..., !.
```

Failu apstrāde

Pamatprincips: *pāradresēšana*.

Vienu un to pašu fizikālo iekārtu var *pēc kārtas* sasaistīt ar *vairākām* loģiskām iekārtām.

Piemērs: vienu un to pašu diska iekārtu var saistīt ar vārdiem **A:** un **B:**

Prolog nodrošina darbu ar standarta iekārtām:

- monitors
- tastatūra
- printeris
- komunikāciju ports

Darbs ar failiem C un C++ valodā

```
#include <stdio>
```

```
FILE *F, *G;
```

```
void main(void) {  
    F = fopen("data1.txt", "w");  
    G = fopen("data2.txt", "w");  
  
    fputs("Hello!", F);  
    fputs("Hello!", G);  
    puts("Hello!");  
  
    fclose(F);  
    fclose(G);  
}
```

C un *C++* programmās funkcijai `fputs()` ir papildus parametrs: faila vārds.

Ir arī funkcija `puts()` informācijas izvadei uz ekrānu.

Pascal valodā ir procedūra `WriteLn()` ar iespējamo papildus parametru – faila vārdu.

Prolog valodā ievades/izvades funkcijās **nav** papildus parametrus.

Informācijas pāradresēšana:

`readdevice()` – lasīšana

`writedevice()` – ierakstīšana

Faila deklarēšana *Prolog* valodā

domains

file = data

Atšķirībā no citiem datu tipiem, vārdu **file** raksta *pa kreisi* no vienādības simbola

Vairāku failu deklarēšana: vārdu **file** var izmantot *tikai vienu reizi*.

file = data1;data2;data3

~~**Neder** variants:~~

~~**file** = data1~~

~~**file** = data2;data3~~

Daži papildus predikāti

1. Faila *eksistēšanas* pārbaude.

```
existfile(<faila vārds>)
```

```
existfile("data.txt")  %Yes vai No
```

2. Faila *dzēšana*.

```
deletefile (<faila vārds>)
```

```
deletefile("data.txt")
```

Ja faila nav – izpildes kļūda.

3. Faila *pārdēvēšana*.

```
renamefile (<vecais vārds>,  
            <jaunais vārds>)
```

```
renamefile("old_data.txt",  
            "new_data.txt")
```

Ja faila nav – izpildes kļūda.

4. *Piekļuves ceļa* norādīšana (iegūšana).

```
disk(<Ceļš>)
```

```
X= "C:/work", disk(X) %nomainīt ceļu
```

```
disk(X), write(X)      %pašreizējais ceļš
```

5. Faila *izvēle* no izvēlnes.

```
dir (<Ceļš>, <Faila paplašinājums>,  
    <Faila vārds>)
```

```
dir ("", "*.txt", X)
```

Rezultāts: izvēlne no visiem failiem ar paplašinājumu *.txt.

Failu meklēšana notiek pašreizējā katalogā.

Pēc faila izvēles:

```
X=C:\WORK\DATA.TXT
```

Ja bija nospiests Esc, rezultāts: No Solution.

Failu apstrādes vispārinātie principi

1. Faila *atvēršana* noteiktajā darba režīmā.
2. *Pāradresēšana* lasīšanai/ierakstīšanai.
3. Lasīšana/ierakstīšana.
4. Jebkuru *citu* predikātu izmantošana.
5. Failu *aizvēršana*.

Piezīme: pirms faila apstrādes ieteicams pārbaudīt arī *faila eksistēšanu*.

Ierakstīšana uz failu

domains

file=f

predicates

out_str

clauses

out_str:-

openwrite(f, "data.txt"),

writedevise(f),

write("This is the test string !"),

closefile(f).

Lasīšana no faila

domains

file=f

predicates

in_str

clauses

in_str :-

openread(f, "data.txt"),

readdevice(f),

readln(X),

write(X),

closefile(f).

Dažas standarta iekārtas

1. keyboard. Tastatūra. Ievades iekārta pēc noklusēšanas.

readdevice (keyboard) %pāradresēšana

2. screen. Ekrāns. Izvades iekārta pēc noklusēšanas.

writedevīce (screen) %pāradresēšana

Neder varianti:

~~readdevice (screen) %klūda~~

~~writedevīce (keyboard) %klūda~~

3. printer. Printeris.

Faila *modifikācija*

domains

file=f

predicates

modi_str

clauses

modi_str:-

```
openmodify(f, "data.txt"),  
writedevise(f),  
write("Here"),  
closefile(f).
```

Rezultāts: failā būs teksta rinda

Here is the test string !

Jaunas informācijas pievienošana

domains

file=f

predicates

app_str

clauses

app_str:-

openappend(f, "data.txt"),

writedevise(f),

write("New String."),

closefile(f).

Rezultāts: failā būs teksta rinda

Here is the test string !New String.

Režīmi *openappend()* un *openmodify()* nodrošina kā ierakstīšanu, tā arī lasīšanu.

Uzdevums: pievienot informāciju tukšajām failam, nolasīt to un izvadīt ekrānā.

domains

file=f

goal

```
openappend(f, "data.txt"),  
writedevise(f), write("Prolog"),  
readdevice(f), filepos(f, 0, 0),  
readln(X), writedevise(screen),  
write(X), closefile(f).
```

Prolog

Informācijas ievade no tastatūras

Uzdevums: nolasīt tekstu no tastatūras un izveidot uz diska teksta failu. **END** – pabeigt teksta ievadi.

Domēnu un predikātu aprakstīšana

domains

file=f

predicates

readin(string)

input

goal

input.

Informācijas ievade no tastatūras (turpinājums)

clauses

```
input :- nl, nl,  
        write("File name -> "), nl, nl,  
        readln(Fname), openwrite(f, Fname),  
        write("Text ->"), nl, writedevise(f),  
        readln(Data), readin(Data),  
        closefile(f).  
readin("END") :- !.  
readin(Data) :-  
    concat(Data, "\13\10", Data_CrLf),  
    write(Data_CrLf), readln(NewData),  
    writedevise(f), readin(NewData).
```

Tiešas piekļuves faili. Rādītāja pozicionēšana.

```
filepos (<Loģiskais vārds>,  
        <Pozīcija failā>, <Nobīdes veids>)
```

Nobīdes vērtība	Nobīdes veids
0	No faila sākuma
1	No pašreizējās pozīcijas
2	No faila beigām

Nolasīt *vienpadsmito* simbolu no faila satura:

```
filepos (data, 10, 0)
```

Simbola lasīšana no faila

Faila data.txt saturs: abcdefgh. Rezultāts: d.

domains

file=f

predicates

get_char(**integer**)

goal

get_char(3) .

clauses

```
get_char(N) :-  
    openmodify(f, "data.txt"),  
    filepos(f, N, 0), readdevice(f),  
    readchar(C), write(C), closefile(f) .
```

Teksta rindiņu lasīšana no faila

Faila data.txt saturs: informācija par darbinieku algām

Igors Ivanovs	500
Jānis Strods	550
Pēteris Celms	450

Pēdējā simbola pozīcija: 23.

Divi simboli CR un LF aizņem divus baidus.

Rezultāts: vienas teksta rindas izmērs ir 25 baiti.

Uzdevums: nolasīt informāciju par darbinieku.

Predikāta parametrs: teksta rindiņas izmērs.

Informācijas lasīšanas programma (sākums)

domains

file=f

predicates

read_record(**integer**)

clauses

```
read_record(RecSize) :-  
    openread(f, "data.txt"),  
    write("Record number: "), nl,  
    readint(RNum),  
    Index = (RNum-1)*RecSize,  
    ...
```

Informācijas lasīšanas programma (turpinājums)

```
... ,  
readdevice(f) ,  
filepos(f, Index, 0) , readln(Data) ,  
write(Data) , nl , closefile(f) .
```

Programmas izpildes rezultāts:

Goal: read_record(25)

Record number:

2

Jānis Strods 550

Yes

Dinamiskās datu bāzes

Ir iespēja *pievienot/dzēst* faktus.

Tādus faktus deklarē sekcijā `database`, nevis sekcijā `predicates`.

Faktus, kā parasti, uzskaita sekcijā `clauses`.

Predikātu saraksts:

1. `assert.`
2. `asserta.`
3. `assertz.`
4. `retract.`
5. `retractall.`

Lai ir datu bāze ar informāciju par darbiniekiem

database

```
d(symbol, symbol, real)
```

clauses

```
d(janis, strods, 600.00).
```

```
d(igors, sergejevs, 550.00).
```

```
d(ivars, jansons, 650.00).
```

Informācijas iegūšana

Goal: d(X, Y, Z)

X=janis, Y=strods, Z=600

X=igors, Y=sergejevs, Z=550

X=ivars, Y=jansons, Z=650

3 Solutions

Jauna ieraksta pievienošana

Goal:

```
assert(d(dmitrijs, andrejevs, 700.00))
```

Yes

Rezultāti:

Goal: d(X, Y, Z)

X=janis, Y=strods, Z=600

X=igors, Y=sergejevs, Z=550

X=ivars, Y=jansons, Z=650

X=dmitrijs, Y=andrejevs, Z=700

4 Solutions

Jaunais ieraksts ir *pēdējais* ieraksts.

Ja tiktu izpildīts predikāts `assertz`, rezultāts būtu tāds pats.

Ja tiktu izpildīts predikāts `asserta`, fakts tiktu ielikts kā *pirmais* fakts.

Goal: `asserta(d(uldis, sarts, 800.00))`

Yes

Rezultāts:

Goal: `d(X, _, _)`

`X=uldis`

...

Ja atkārtot kādu predikātu, tiks ielikts *papildus* fakts.

Faktu unikalitāte *netiks* kontrolēta.

Faktu dzēšana:

Goal: `retract(d(janis, strods, 600.00)).`

Yes

Dzēšanas rezultāts:

Goal: `d(X, _, _)`

...

Informācijas par darbinieku "janis strods" vairs nav.

Atkārtotā predikāta izpilde:

Goal: `retract(d(janis, strods, 600.00)).`

No

Ja fakta nav, rezultāts vienmēr būs No.

Faktu *saglabāšana* failā

```
save(<faila vārds>)
```

Lai ir datubāze ar informāciju par firmu

database

```
darbinieks(symbol, symbol)
```

predicates

```
strukturvieniba(symbol)
```

goal

```
assert(darbinieks(sergejevs, ivans)),  
       save("data.txt").
```

clauses

```
darbinieks(strods, janis).  
strukturvieniba("Programmētāji").
```

Rezultāts (fails `data.txt`):

```
darbinieks("strods", "janis")
```

```
darbinieks("sergejevs", "ivans")
```

Informācija par struktūrvienībām *netika saglabāta*.

Faktu *lasīšana* no faila

```
consult(<faila vārds>)
```

Teksta fails izveidots ar predikāta `save()` palīdzību, vai parastajā teksta redaktorā.

Fakti tiks pievienoti pie jau eksistējošiem faktiem.

```
Lai ir fails data.txt ar informāciju par darbiniekiem  
darbinieks("juris", "strods")  
darbinieks("igors", "ivanovs")
```

Uzdevums: nolasīt visu informāciju no faila, izvadīt ekrānā, pievienot jaunu faktu un saglabāt rezultātu uz diska.

database

```
darbinieks(symbol, symbol)
```

predicates

```
visi_darbinieki
```

clauses

```
visi_darbinieki:- darbinieks(X, Y),  
    write(X, " ", Y), nl, fail.  
visi_darbinieki:-true.
```


Informācijas apstrāde

goal

```
consult("data.txt"), visi_darbinieki,  
write("Vārds -> "), readln(Vards),  
write("Uzvārds -> "), readln(Uzvars),  
assert(darbinieks(Vards, Uzvars)),  
save("data.txt"), retractall(_),  
write("Darbs pabeigts !").
```

Fails uz diska pēc izmaiņām (bija ievadīti vārds uldis un uzvārds egle).

```
darbinieks("juris", "strods")  
darbinieks("igors", "ivanovs")  
[darbinieks("uldis", "egle")]
```

Predikāts `retractall()` dzēš *visus* dinamiskus faktus.

Programmā var eksistēt vairākas database daļas

database

```
prezidents(symbol, symbol)
```

database - darbinieki

```
darbinieks(symbol, symbol)
```

database - klienti

```
klients(symbol, symbol)
```

clauses

```
darbinieks(strods, janis).
```

```
klients(ivans, sergejevs).
```

```
prezidents(juris, petrovs).
```

Uzdevums: izdzēst visus klientus, *informējot* par dzēšanas rezultātiem.

Goal: `retract(klients(X, Y))`

`X=ivans, Y=sergejevs`

1 Solution

Var izdzēst visus klientus, *neinformējot* par dzēšanas rezultātiem.

Goal: `retract(klients(____))`

Yes

1 Solution

Piezīme: abos gadījumos pēc mēģinājuma atkārtot dzēšanu dabūsim atbildi `No Solution`.

Lai datubāzē klienti ir *vairāki* predikātu veidi.

Piemēram, klienti var būt ne tikai *fiziskās*, bet arī *juridiskās* personas.

database - klienti

klients(symbol, symbol)

organizacija(symbol)

...

clauses

...

klients(ivans, sergejevs) .

...

organizacija(dzintars) .

Uzdevums: izdzēst visu informāciju no daļas `klienti`.

Var secīgi izdzēst visus faktus `klients()` un `organizacija()`.

Problēma: jādomā par visiem faktu vārdiem.

Labākais risinājums

Goal: `retractall(_ , klienti)`

Yes

Dzēšanas pārbaude

Goal: `klients(X, Y)`

No Solution

Goal: `organizacija(X)`

No Solution

Predikāta `retractall()` pielietošanas rezultāts: vienmēr
Yes.

Lai dzēšanas operācija bija atkārtota:

Goal: `retractall(____, klienti)`
Yes

Predikātu `retractall()` var pielietot, lai izdzēstu
konkrētos faktus.

Goal: `retractall(klients(____, ____), klienti)`
Yes

Goal: `retractall(klients(____, ____))`
Yes

Predikātu `save()` var pielietot, lai saglabātu faktus no *konkrētās* datu sekcijas.

Goal: `save("klienti.txt", klienti)`

Yes

Līdzīgajā stilā var pielietot predikātu `consult()`.

Goal: `consult("klienti.txt", klienti)`

Yes


Sekcijas vārdu var arī norādīt predikātos `assert()` un `retract()`. Bet nekāda *papildus* efekta nebūs.

`assert(klients(ivars, strazds), klienti).`

`assert(klients(ivars, strazds)).`

Predikātus `assert()` un `retract()` var izmantot *globālo mainīgo* imitēšanai.

Piemērs: stundu skaitītājs. Stundas: 0, 1, ..., 23.



Programmas sākums:

database

laiks(**integer**)

predicates

nak_stunda

tests(**integer**, **integer**)

clauses

laiks(22) .

Programmas beigas:

```
nak_stunda :- laiks(Vecais),  
              tests(Vecais, Jaunais),  
              retract(laiks(_)),  
              assert(laiks(Jaunais)).  
tests(X, Y) :- X < 23, Y = X + 1, !.  
tests(_, 0).
```

Predikātu `nak_stunda` var optimizēt:

```
nak_stunda :- retract(laiks(Vecais)),  
              tests(Vecais, Jaunais),  
              assert(laiks(Jaunais)).
```

Var vienlaicīgi *iegūt* vērtību un *izdzēst* faktu.

Uzdevums: nodrošināt iespēju izmainīt tīkla lietotāja statusu.

database

```
users(symbol, symbol)
```

predicates

```
change
```

clauses

```
users(user001, registered).  
users(user002, registered).  
users(user003, not_registered).  
change :- write("User ?"),  
          readln(User), write("Status ?"),  
          readln(Status),  
          retract(users(User, _)),  
          assert(users(User, Status)).
```

Informācijas sistēma tūrisma atbalstīšanai (prototips)

Predikātu saraksts (informācijas sistēmas *konceptija*)

1. Starp vairākām pilsētām eksistē *tiešie* avio reisi.

`avio(<pilsēta>, <pilsēta>)`

Šis fakts nozīmē, ka ir reisi *abos virzienos*.

2. Cilvēks - tūrists vienmēr atrodas konkrētajā (*aktuālajā*) pilsētā.

`akt_pilseta(<pilsēta>)`

Ceļojuma procesā šī pilsēta *mainīsies*.

3. Katrā pilsētā ir kādi vēsturiskie *pieminekļi*.

`pieminekļis (<pilsēta>)`

4. Sistēma pārbauda *pārlidojuma iespēju*.

`var_lidot (<pilsēta>)`

5. Pārlidojumu *realizē* likums

`lidot (<pilsēta>)`

6. Informāciju par aktuālo pilsētu *koriģē* likums

`lidojums (<pilsēta>)`

7. Katrā pilsētā tūristam ir informācija par *vietējiem pieminekļiem* un par *reisiem* uz citām pilsētām.

informacija (<pilsēta>)

8. Likums informacija savā darbā izmanto likumus.

pieminekli (<pilsēta>) un saites (<pilsēta>)

Ceļojums sāksies no Rīgas.

Informācijas sistēmas *kods*

database

```
akt_pilseta(symbol)
```

predicates

```
avio(symbol, symbol) .
```

```
var_lidot(symbol) .
```

```
lidojums(symbol) .
```

```
lidot(symbol) .
```

```
pieminekklis(symbol, symbol) .
```

```
pieminekli(symbol) .
```

```
saites(symbol)
```

```
informacija(symbol) .
```

Informācijas sistēmas *kods* (turpinājums)**clauses**

```
akt_pilseta („Rīga”) .  
  
avio („Rīga”, „Londona”) .  
avio („Rīga”, „Maskava”) .  
  
avio („Londona”, „Maskava”) .  
avio („Londona”, „Ņujorka”) .  
  
avio („Maskava”, „Ņujorka”) .  
avio („Maskava”, „Vladivostoka”) .  
  
avio („Vladivostoka”, „Ņujorka”) .
```

Informācijas sistēmas *kods* (turpinājums)

```
var_lidot(X) :- akt_pilseta(Y),  
    avio(X, Y), !.  
var_lidot(X) :- akt_pilseta(Y),  
    avio(Y, X), !.  
var_lidot(_) :-  
    write("Lidojums nav iespējams !"),  
    nl, fail.  
  
lidot(X) :- var_lidot(X), lidojums(X).  
  
lidojums(X) :-  
    retract(akt_pilseta(_)),  
    assert(akt_pilseta(X)),  
    informacija(X).
```


Informācijas sistēmas *kods* (turpinājums)

```
piemineklis („Rīga”, „Doma baznīca”).
piemineklis („Rīga”,
    „Brīvības piemineklis”).
piemineklis („Maskava”, „Kremlis”).
piemineklis („Londona”, „Torņa tilts”).
piemineklis („Ņujorka”,
    „Brīvības monuments”).
...
pieminekli(X) :- piemineklis(X, Y),
    write("    ", Y), nl, fail.
pieminekli(_).
```

Informācijas sistēmas *kods* (turpinājums)

```
saites(X) :- avio(X, Y),  
    write("    ", Y), nl, fail.  
saites(X) :- avio(Y, X),  
    write("    ", Y), nl, fail.  
saites(_).  
  
informacija(X) :-  
    akt_pilseta(X), nl,  
    write("Aktuālā pilsēta:", X), nl,  
    write("Pieminekļi: "), nl,  
    pieminekli(X),  
    write("Saites ar citām pilsētām:"),  
    nl, saites(X).
```

Informācijas sistēmas *izmantošanas* piemēri:

Lidojums uz Maskavu

Goal: lidot ("Maskava")

Aktuālā pilsēta: Maskava

Pieminekļi:

Kremlis

Sarkanais laukums

...

Saites ar citām pilsētām:

Ņujorka

Vladivostoka

Rīga

Londona

Informācijas sistēmas *izmantošanas* piemēri (turpinājums)

Reisi no Maskavas:

Goal: `saites ("Maskava")`

`Nuĵorka`

`Vladivostoka`

`Rīga`

`Londona`

Lidojums uz Londonu:

Goal: `lidot ("Londona")`

`Aktuālā pilsēta: Londona`

`Pieminekli:`

`Torņa tilts`

`...`

Mēģinājums lidot uz Vladivostoku:

Goal: lidot("Vladivostoka")

Lidojums nav iespējams !

Lidot uz Rīgu:

Goal: lidot("Rīga")

Aktuālā pilsēta: Rīga

Pieminekli:

Doma baznīca

Brīvības piemineklis

Saites ar citām pilsētām:

Londona

Maskava

Bināru skaitļu ģenerēšana

Iegūt visus binārus skaitļus diapazonā 0000 – 1111.

Imperatīvās programmēšanas valodas: vairāki ieliktie cikli.

Var

```
A, B, C, D : Integer;
```

```
...
```

```
For A := 0 To 1 Do
```

```
    For B := 0 To 1 Do
```

```
        For C := 0 To 1 Do
```

```
            For D := 0 To 1 Do
```

```
                WriteLn(A, B, C, D);
```

Problēma: ieliktu ciklu daudzums tomēr ierobežots.

Problēmas risinājums PROLOG valodā

predicates

```
digit(integer)
```

```
bin_numeric
```

clauses

```
digit(0).
```

```
digit(1).
```

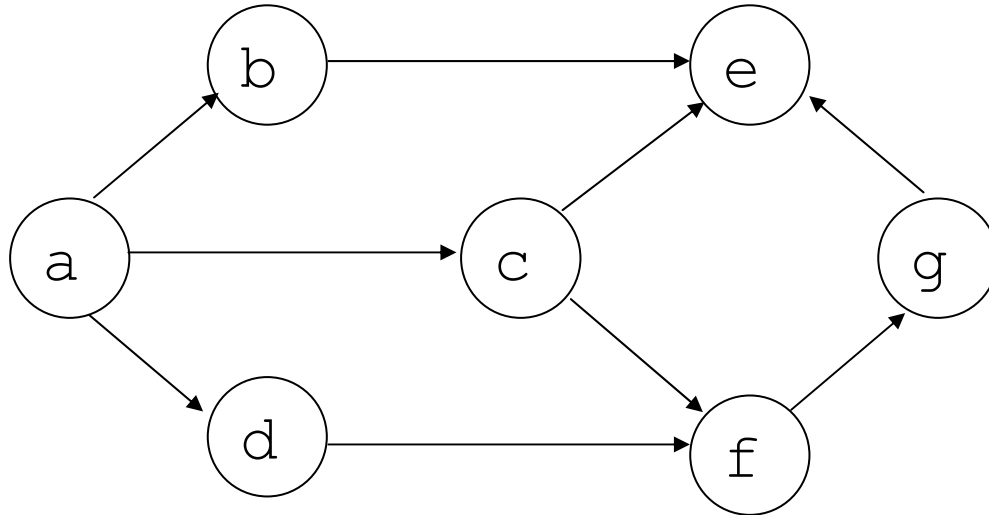
```
bin_numeric :- digit(A), digit(B),  
               digit(C), digit(D),  
               write(A, B, C, D), nl, fail.
```

Piezīme: šis algoritms *nav atkarīgs* no datu tipa.

Var ģenerēt secības, izmantojot, piemēram, *teksta rindas*.

Imperatīvajās programmēšanas valodās var izmantot `enum`.

Ceļu meklēšana orientētājā grafā *bez cikliem*



Programmas sākums

predicates

edge (**symbol**, **symbol**)

path (**symbol**, **symbol**)

rout (**symbol**, **symbol**)

Programmas turpinājums

clauses

```
edge(a,b) . edge(a,c) . edge(a,d) .  
edge(b,e) . edge(c,e) . edge(c,f) .  
edge(d,f) . edge(f,g) . edge(g,e) .  
path(X,X) :- write(X) .  
path(X,Z) :- edge(Y,Z) , path(X,Y) ,  
               write(Z) .  
rout(X,Z) :- path(X,Z) , nl, fail.
```

Visi ceļi no virsotnes e uz virsotni a

Goal: rout(e, a)

No

Visi ceļi no virsotnes a uz virsotni e

Goal: rout (a, e)

abe

ace

acfge

adfge

Visi ceļi no virsotnes d uz citām virsotnēm

Goal: rout (d, V)

d

df

dfg

dfge

Lai ir pievienots vēl viens fakts:

edge (b, a)

Rezultāts:

abababababa...

Ceļu meklēšana orientētājā grafā *ar cikliem*.

domains

list=**symbol***

predicates

edge (**symbol**, **symbol**)

rout (**symbol**, **symbol**)

path_organizer (**symbol**, **symbol**, list)

member (**symbol**, list)

Ceļu meklēšana orientētājā grafā *ar cikliem* (turpinājums)**clauses**

```
... %fakti edge(X, Y)
path_organizer(X,X,_) :- write(X) .
path_organizer(X,Z,T) :- edge(Y,Z) ,
    not(member(Y,T)) ,
    path_organizer(X,Y,[Y|T]) ,
    write(Z) .

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

rout(X,Z) :- path_organizer(X,Z,[Z]) ,
    nl, fail.
```

Tagad datu faktu bāzē ir divi fakti:

edge (b, a) .

edge (e, b) .

Visi ceļi no virsotnes a uz virsotni e

Goal: rout (a, e)

abe

ace

acfge

adfge

Visi ceļi uz virsotni d

Goal: rout (X, d)

d

ad

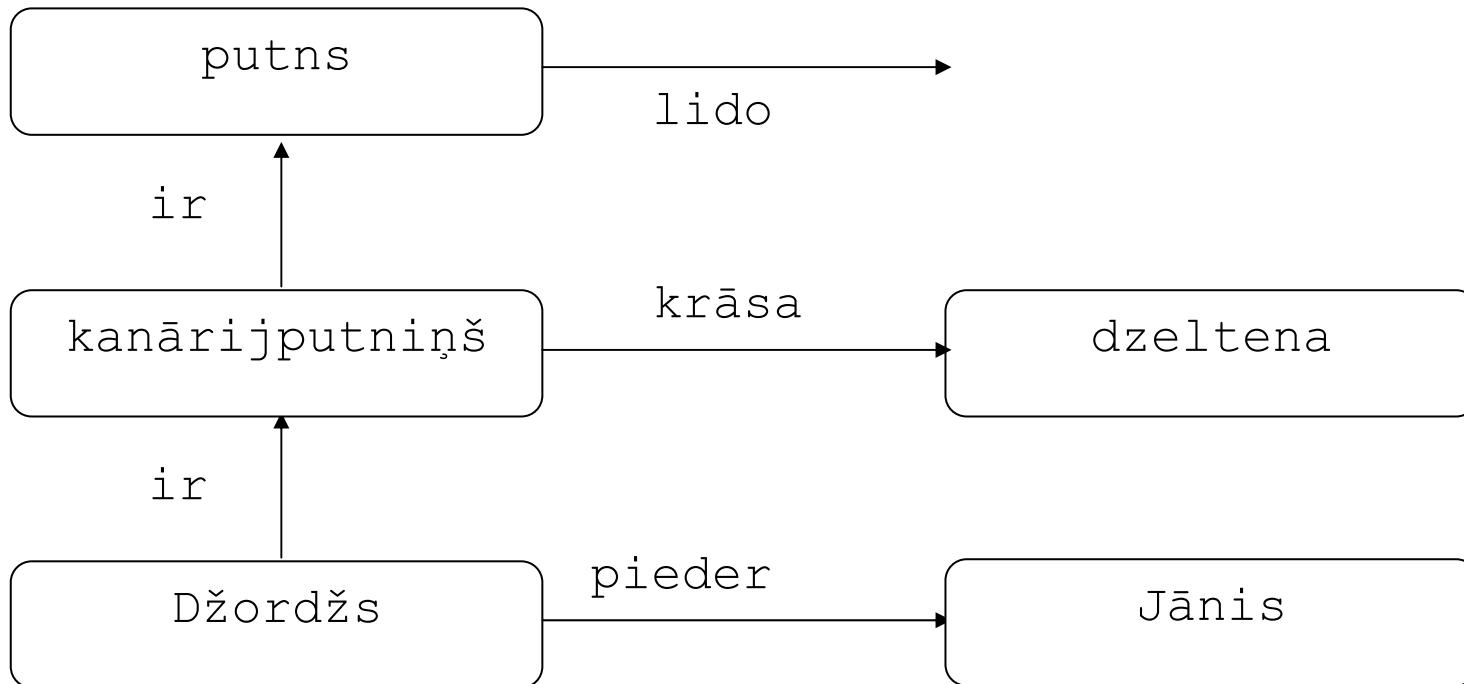
bad

PROLOG un *semantiskie tīkli*

Būtības (būtību klases) ir *mezgli*.

Attiecības starp būtībām ir *loki*.

Semantiskā tīkla piemērs



Semantiskā tīkla realizācija PROLOG valodā
Predikātu *deklarācijas* un daži fakti.

predicates

```
ir(symbol, symbol)
```

```
pieder(symbol, symbol)
```

```
krasa(symbol, symbol)
```

```
lido(symbol)
```

clauses

```
ir(„kanārijputniņš”, putns).
```

```
ir(„Džordžs”, „kanārijputniņš”).
```

```
pieder(„Džordžs”, „Jānis”).
```

Semantiskā tīkla realizācija PROLOG valodā. Likumi:

```
krasa („kanārijputniņš”, dzeltena) .  
krasa(X, Y) :- ir(X, Z), krasa(Z, Y) .  
lido(putns) .  
lido(X) :- ir(X, Y), lido(Y) .
```

Pārbaude: kurš var lidot?

Goal: lido („Džordžs”) %Yes

Goal: lido („kanārijputniņš”) %Yes

Goal: lido („Jānis”) %No

Goal: lido(X)

X=putns

X=kanārijputniņš

X=Džordžs

3 Solutions

Ārējas datubāzes un termu ķēdītes

Ārējā datubāze ir Prolog *termu* kolekcija.

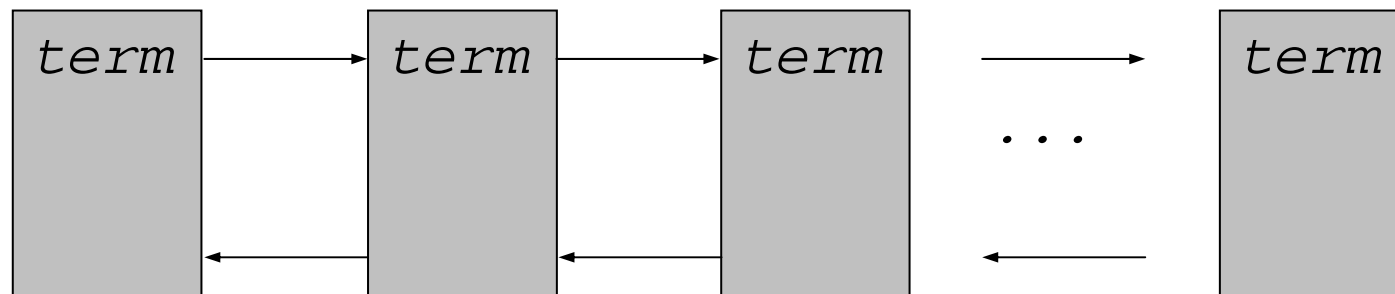
Termi saglabāti *ķēdītēs* (chains).

Ķēdīte satur jebkuru *termu daudzumu*.

Ārējā datubāze satur jebkuru *ķēdīšu daudzumu*.

Katrai ķēdītei ir *vārds* (teksta rinda).

Ķēdīte PROLOG valodā (divkāršsaistītais saraksts)



Daži svarīgie predikāti

Piezīme: predikātu parametru statuss:

i – input

o – output

i/o – input/output

```
1. db_create(db_selector <Dbase>,  
            string <Name>, place <Place>)  
            (i, i, i)
```

Izveidot *jaunu* tukšu bāzi ar vārdu Name.

Norādes uz to realizē izmantojot Dbase.

Datubāzes izvietošana: Place. Vērtības:

in_memory - iekšējā atmiņā, *in_file* - failā, *in_ems* -
EMS atmiņā (paplašinātā atmiņā)

Piezīme: *Turbo Prolog* neatbalsta vērtību *in_ems*.

2. `db_close (db_selector <Dbase>) (i)`

Aizvērt iepriekš atvērtu datubāzi. Ja datubāzes nav – izpildes kļūda.

Obligāti izmantot pēc darba pabeigšanas, citādi datubāze **netiks atvērta** ar `db_open ()` palīdzību.

3. `db_open (db_selector <Dbase>,
 string <Name>, place <Place>)
 (i, i, i)`

Atvērt jau *eksistējošu* datubāzi.

4. `db_delete (string <Name>,
 place <Place>) (i, i)`

Izdzēst aizvērtu datubāzi (izdzēst failu vai atbrīvot atmiņu).
Rezultāts: vienmēr *Yes*, pat ja datubāze neeksistē.

```
5. chain_insertz(db_selector <Dbase>,
    symbol <Chain>, domain <Domain>,
    term <Term>, ref <Ref>)
    (i, i, i, i, o)
```

Ielikt termu ķēdītē kā *pēdējo* termu.

Chain – ķēdītes *vārds*.

Domain – terma *domēns*.

Ref – *norāde* uz termu datubāzē.

```
6. chain_inserta(db_selector <Dbase>,
    symbol <Chain>, domain <Domain>,
    term <Term>, ref <Ref>)
    (i, i, i, i, o)
```

Ielikt termu ķēdītē kā *pirmo* termu.

```
7. chain_insertafter(db_selector <Dbase>,
    domain <Domain>, ref <Ref>,
    term <Term>, ref <NewRef>)
    (i, i, i, i, o)
```

Ielikt termu ķēdītē pēc *noteiktā* terma.

```
8. chain_terms(db_selector <Dbase>,
    symbol <Chain>, domain <Domain>,
    term <Term>, ref <Ref>)
    (i, i, i, i/o, o)
```

Iegūt termu no ķēdītes/*ierakstīt* termu ķēdītē

Vēlāk no iegūtā terma var dabūt visu iekšēju informāciju.

Ārējas datubāzes veidošana

Lai ir informācija par *cilvēkiem*. Katrs cilvēks var strādāt noteiktajā vietā (vietās) *kā darbinieks*.

Cilvēki

Numurs	Vārds	Uzvārds

1

Darbinieki

Numurs	Darba vieta

N

1. iespēja.

a. Ir *divi* faili.

b. Katrā failā ir *viena* termu ķēdīte.

Domēnu un predikātu deklarācija

domains

```
db_selector = cilveeki ; darbinieki  
c = cilveeks(integer, symbol, symbol)  
d = darbinieks(integer, symbol)
```

predicates

```
starts  
informacija
```

clauses

Datubāzes veidošana un termu ielikšana

starts :-

%failu veidošana

```
db_create(cilveeki, "cilv.bin",  
         in_file),
```

```
db_create(darbinieki, "darbin.bin",  
         in_file),
```

%kēdītes veidošana (cilvēki)

```
chain_inserta(cilveeki, chain, c,  
             cilveeks(1, juris, strods),   ),
```

```
chain_insertz(cilveeki, chain, c,  
             cilveeks(2, sergejs, ivanovs),   ),
```


Datubāzes veidošana un termu ielikšana (turpinājums)

```
%kēdītes veidošana (darbinieki)
chain_insertz(darbinieki, chain, d,
             darbinieks(2, "VEF"), _),
chain_inserta(darbinieki, chain, d,
             darbinieks(1, "VEF"), _),
chain_inserta(darbinieki, chain, d,
             darbinieks(1, "Dzintars"), _),
informaacija,
%failu aizvēršana
db_close(cilveeki),
db_close(darbinieki).
```

Informācijas izvade ekrānā

```
informaacija :-  
    write("* Numurs * Vārds: *  
        Uzvārds * Darba vieta *"), nl, !,  
    chain_terms(cilveeki, chain, c,  
        cilveeks(Num, Vards, Uzvars), _),  
    chain_terms(darbinieki, chain, d,  
        darbinieks(Num, Firma), _),  
    writef("* %-8* %-8* %-10* %-12 *",  
        Num, Vards, Uzvars, Firma),  
    nl, fail.  
informaacija.  
makewindow(1, 2, 2, "Rezults", 0, 0,  
    25, 80), starts.
```

Iegūtie rezultāti

```
...
* 1 * juris      * strods      * Dzintars      *
* 1 * juris      * strods      * VEF            *
* 2 * sergejs    * ivanovs     * VEF            *
```

Piezīme: faili „cilv.bin” un „darbin.bin” satur informāciju binārajā formā.

Šo informāciju grūti saprast, atverot failu parastajā teksta redaktorā.

Tagad failus var atvērt ar `db_open()` palīdzību.

2. iespēja.

a. Ir *viens* fails.

b. Failā ir *divas* termu ķēdītes.

Domēnu un predikātu deklarācija

domains

db_selector = clv_drb

c_d = cilveeks(**integer**, **symbol**, **symbol**);
darbinieks(**integer**, **symbol**)

predicates

starts

informaacija

clauses

Datubāzes veidošana un termu ielikšana

```
starts :-
```

```
    %faila veidošana
```

```
    db_create(clv_drb, "cil_darb.bin",  
              in_file),
```

```
    %kādītes veidošana (ciltvēki)
```

```
    chain_inserta(clv_drb, chain1, c_d,  
                  cilveeks(1, juris, strods), _),  
    chain_insertz(clv_drb, chain1, c_d,  
                  cilveeks(2, sergejs, ivanovs), _),
```

Datubāzes veidošana un termu ielikšana (turpinājums)

```
%kēdītes veidošana (darbinieki)
chain_insertz(clv_drb, chain2, c_d,
  darbinieks(2, "VEF"), _),
chain_inserta(clv_drb, chain2, c_d,
  darbinieks(1, "VEF"), _),
chain_inserta(clv_drb, chain2, c_d,
  darbinieks(1, "Dzintars"), _),

%faila aizvēršana
db_close(clv_drb).
```

Informācijas izvade uz ekrānu

```
informaacija :-  
    chain_terms(clv_drb, chain1, c_d,  
        cilveeks(Num, Vards, Uzvars), _),  
    chain_terms(clv_drb, chain2, c_d,  
        darbinieks(Num, Firma), _),  
    writef("%-8*  %-8*  %-10*  %-12", Num,  
        Vards, Uzvars, Firma), nl, fail.  
informaacija:-true.
```

Rezultāti sakrīt ar iepriekšējiem rezultātiem.

3. *iespēja*.

a. Ir *viens* fails.

b. Failā ir *viena* termu ķēdīte.

Iepriekšējā piemērā aizvietot visus `chain1` un `chain2` uz `chain`.

Iegūsim *tos pašus* rezultātus.

Daži svarīgie predikāti (turpinājums)

9. `chain_delete(db_selector <Dbase>, symbol <Chain>) (i, i)`

Izdzēst visus terminus ķēdītē.

Ja ķēdīte eksistē, tad vienmēr `true` (pat ja ķēdīte tukšā). Ja ķēdītes nav, tad `fail`.


```
10. chain_first (db_selector <Dbase>,  
    symbol <Chain>, ref <FirstRef>)  
    (i, i, o)
```

Atsauce uz *pirmo* termu ķēdītē. Ja ķēdīte tukša, vai neeksistē, tad fail.

```
11. chain_last (db_selector <Dbase>,  
    symbol <Chain>, ref <FirstRef>)  
    (i, i, o)
```

Atsauce uz *pēdējo* termu ķēdītē. Ja ķēdīte tukša, vai neeksistē, tad fail.

```
12. chain_next(db_selector <Dbase>,  
    ref <Ref>, ref <NextRef>)  
    (i, i, o)
```

Atsauce uz *nākošu* termu ķēdītē.

Ja nākošā terma nav, tad fail.

```
13. chain_prev(db_selector <Dbase>,  
    symbol <Chain>, ref <FirstRef>)  
    (i, i, o)
```

Atsauce uz *iepriekšēju* termu ķēdītē.

Ja nākošā terma nav, tad fail.

```
14. ref_term(db_selector <Dbase>,  
            domain <domain>, ref <Ref>,  
            term <Term>) (i, i, i, o)
```

Iegūt termu pēc atsaucēs.

Lai ir datubāze par cilvēkiem:

```
cilveks(1, juris, strods)  
cilveks(2, sergejs, ivanovs)  
cilveks(3, aigars, egle)  
cilveks(4, ivans, petrovs)
```

Datubāze bija veidota ar `chain_inserta`
(`chain_insertz`) palīdzību.

Uzdevums: *izvadīt informāciju uz ekrānu.*

Saraksta apstrāde – pārvietošana uz priekšu

domains

```
db_selector = cilveeki
```

```
c = cilveeks(integer, symbol, symbol)
```

predicates

```
start
```

```
lasiit(ref)
```

clauses

```
start :-
```

```
    db_open(cilveeki, "cilv.bin",  
            in_file),
```

```
    chain_first(cilveeki, chain, Start),  
    lasiit(Start), db_close(cilveeki).
```

Saraksta apstrāde – pārvietošana uz priekšu (turpinājums)

```
lasiit(Ref) :-  
    ref_term(cilveeki, c, Ref, Term),  
    write(Term), nl, fail.  
  
lasiit(Ref) :-  
    chain_next(cilveeki, Ref, Next), !,  
    lasiit(Next).  
  
lasiit(_).
```

goal

```
start.
```

Rezultātu fragments:

```
cilveks(1, "juris", "strods")  
cilveks(2, "sergejs", "ivanovs")  
...
```

Lai ir nepieciešama terma *informācijas apstrāde*

```
lasiit(Ref) :-
```

```
    ref_term(cilveeki, c, Ref, Term),
```

```
    Term = cilveeks(Num, Vards, Uzvars),
```

```
    writef("% . % %", Num, Vards, Uzvars),  
    nl, fail.
```

Lai ir nepieciešams apstrādāt sarakstu *pretējā* virzienā.

1. chain_first() => chain_last()

2. chain_next() => chain_prev()

Apstrādes pamatprincipi *neizmainīsies*.

Daži svarīgie predikāti (turpinājums)

```
15. term_delete(db_selector <Dbase>,  
    symbol <Chain>, ref <Ref>)  
        (i, i, i)
```

Izdzēst termu no ķēdītes. Ja atsauce nepareiza - izpildes kļūda.

```
16. term_replace(db_selector <Dbase>,  
    domain <Domain>, ref <Ref>,  
    term <Term>) (i, i, i, i)
```

Aizvietot vienu termu ar otru (pēc atsauces)

```
chain_first(cilveeki, chain, Ref),  
    term_replace(cilveeki, c, Ref,  
        cilveeks(5, dmitrijs, vasiljevs))
```

B+ koki

B+ koks ir speciālā datu struktūra lielu *datu apjomu* *kārtošanai*.

B+ koku var uztvert kā *datubāzes indeksu*. Dažreiz to salīdzina ar *rādītāju*.

Koks būs saglabāts *ārējā datubāzē*, kopā ar datiem.

Koka elementi - vērtību pāri: *atslēgas rinda* un attiecīga *datubāzes atsauce*.

Datubāzes radīšanas procesā no sākuma pievieno *jaunu ierakstu*, bet pēc tam - *ieraksta atslēgu*.

Atslēgas grupētas *pa lappusēm*. Runa ir par *iekšējiem mezgliem*.


```
1. bt_create(db_selector <Dbase>,  
            string <BtreeName>,  
            integer <BtreeSel>,  
            integer <KeyLen>, integer <Order>)  
    (i, i, o, i, i).
```

KeyLen: atslēgas izmērs.

Order: atslēgu daudzums uz lappusēm.

Katrai lappusei ir minimums *Order* atslēgas un maksimums $2 * Order$ atslēgas. Piemērs: ja *Order* ir 4, tad $Min=4$, $Max=8$.

Piezīme: nav iespējams izmainīt *KeyLen* un *Order* vēlāk.

```
bt_create(cilveki, "clv_koks",  
        CilvKoks, 5, 4)
```

Izpildes kļūda, ja koks ar tādu vārdu jau eksistē, vai datubāze nav atvērta.

```
2. bt_close(db_selector <Dbase>,  
            integer <BtreeSel>)  
    (i, i)
```

Aizvērt B+ koku. Kļūda, ja datubāze nav atvērta, vai rādītājs nav pareizs.

```
3. key_insert(db_selector <Dbase>,  
              integer <BtreeSel>, string <Key>,  
              ref <Ref>)  
    (i, i, i, i)
```

Jaunās atslēgas ielikšana.

Ref - attiecīgā atsauce uz termu datubāzē.

Koka veidošana

domains

```
db_selector = cilveki
iedziv =
    cilveks(integer, symbol, symbol)
db_create(cilveki, "cilveki.bin", in_file),
bt_create(cilveki, "clv_koks",
    CilvKoks, 5, 3),
chain_inserta(cilveki, chain1, iedziv,
    cilveks(1, juris, strods), Ref1),
key_insert(cilveki, CilvKoks, "1", Ref1),
..., bt_close(cilveki, CilvKoks).
```

```
4. bt_open(db_selector <Dbase>,
           string <BtreeName>,
           integer <BtreeSel>)
    (i, i, o)
```

Atvērt B+ koku.

```
5. key_search(db_selector <Dbase>,
              integer <BtreeSel>,
              string <Key>, ref <Ref>)
    (i, i, i, o), (i, i, i, i)
```

Atslēgas meklēšana.

```
6. key_current(db_selector <Dbase>,
               integer <BtreeSel>,
               string <Key>, ref <Ref>)
    (i, i, o, o),
```

Informācija par pašreizējo atslēgu.

Informācijas meklēšana:

predicates

```
meklet(db_selector, integer, string)
```

```
...
```

```
meklet(Dati, Koks, Key) :-
```

```
    key_search(Dati, Koks, Key, _), !,
```

```
    key_current(Dati, Koks, NewKey, Ref),
```

```
    write("\nAtslēga: ", NewKey,
```

```
        ", Atsauce: ", Ref), nl,
```

```
    ref_term(Dati, iedziv, Ref, T),
```

```
    write(T).
```

```
meklet(_, _, _) :-
```

```
    write("\nAtslēga nav atrasta !").
```

7. `key_first(db_selector <Dbase>,
integer <BtreeSel>, ref <Ref>)
(i, i, o)`

Atrast *pirmo* atslēgu B+ kokā.

8. `key_last(db_selector <Dbase>,
integer <BtreeSel>, ref <Ref>)
(i, i, o)`

Atrast *pēdējo* atslēgu B+ kokā.

9. `key_next(db_selector <Dbase>,
integer <BtreeSel>, ref <NextRef>)
(i, i, o)`

Pāriet uz *nākošo* atslēgu B+ kokā.

10. `key_prev(db_selector <Dbase>,
integer <BtreeSel>, ref <PrevRef>)
(i, i, o)`

Pāriet uz *iepriekšējo* atslēgu B+ kokā.

Matemātiskas funkcijas

```
1. domain abs(domain <vērtība>) %Modulis  
X = abs(-2), write(X) %2
```

Piezīme: neder

```
write(abs(-2)) %kompilācijas kļūda  
X = abs(-2.5), write(X) %2.5
```

```
2. real sin(real <vērtība>)  
% Sinuss. Arguments: radiāni  
X = sin(0), write(X) %0
```

```
3. real cos(real <vērtība>) %Kosinuss
```

```
4. real tan(real <vērtība>) %Tangenss
```

```
5. real arctan(real <vērtība>) %Arktangenss
```

6. real **log**(real <vērtība>)

%Decimālais logaritms

X=log(100) %2

7. real **ln**(real <vērtība>)

%Naturālais logaritms

8. real **exp**(real <vērtība>).

%Eksponenta

X = ln(exp(1)) %X=1

9. domain **round**(domain <vērtība>)

X = round(1.6) %X=2

X = round(1.3) %X=1

10. domain **trunc**(domain <vērtība>)

X = trunc(1.6) %X=1

X = trunc(1.3) %X=1

Gadījumskaitļu ģenerēšana

1. `random(real <R>)`

%Gadījumskaitlis diapazonā $0 \leq R < 1$

`random(X)` %X=0.83649497967

2. `random(integer <MaxInt>, integer <R>)`

%Gadījumskaitlis diapazonā $0 \leq R < \text{MaxInt}$

`random(10, X)` %X=9

3. `randominit(integer <vērtība>).`

%Inicializēt nejaušo skaitļu ģeneratoru
(**Visual Prolog**).

Skaņu signāli

`sound(<laiks: 1/100 sek.>, <frekvence>)`

`sound(100, 300)` %1 sekunde, 300 Hz

Faktu daudzuma ierobežošana

predicates

determ president(symbol, symbol)

clauses

president(ivans, petrovs).

president(uldis, strods).

Programma tiks palaista. Kļūda notiks pēc jautājuma:

president(X, Y)

Dinamiskā fakta gadījumā tiks apstrādāts tikai *pirmais* fakts:

database

determ president(symbol, symbol)

president(X, Y)

X=ivans, Y=petrovs

1 Solution

Pēc noklusējuma: `nondeterm`. To var norādīt patstāvīgi:

predicates

nondeterm darbinieks (`symbol`, `symbol`)

Konstantes

constants

`pi = 3.14`

goal

`write(pi) . %3.14`

Citas iespējas *deklarēt* konstanti:

`PI = 3.14` vai `Pi = 3.14`

Bet *programmā* obligāti jābūt:

`write(pi) %visi simboli apakšējā reģistrā`

Papildus predikāti:

1. % pašreizējais laiks

`time`(H, M, Sec, MSec)

2. % pašreizējais datums

`date`(Y, M, D)

3. % DOS komandas izpilde

`system`(string <komandas rinda>)

Predikāts vienmēr atgriež Yes.

`system("")` %izeja uz DOS

`system("copy dati.txt copy.txt")`

%rezultāts Yes

`system("copyyy dati.txt copy.txt")`

%rezultāts arī Yes

Logu saskarnes pamati

1. Loga *veidošana*

```
makewindow(  
    integer <WindowNo>,      %loga numurs  
    integer <ScrAttr>,       %loga krāsa  
    integer <FrameAttr>,     %rāmīša krāsa  
    string <FrameHeader>,   %loga virsraksts  
    integer <Row>,           %rinda  
    integer <Column>,        %kolona  
    integer <Height>,        %augstums  
    integer <Width>          %platums
```

Logs uz visu ekrānu:

```
(Row, Column, Height, Width) =  
    (0, 0, 25, 80).
```

2. Loga *eksistēšanas* pārbaude

`existwindow (<WinNo>)`

3. *Ātra* pāreja starp logiem

`gotowindow (<WinNo>)`

4. Pāreja starp logiem ar *satura saglabāšanu*

`shiftwindow (<WinNo>)`

5. Pašreizējā loga *attīrīšana*

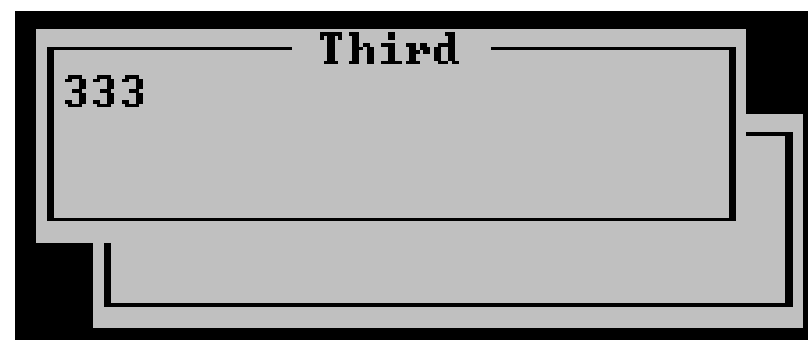
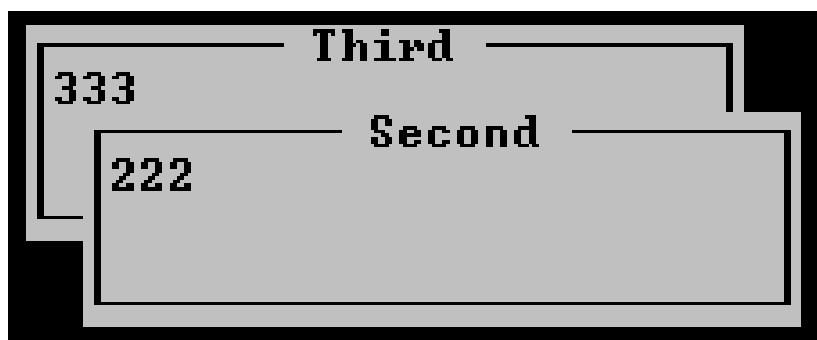
`clearwindow`

6. Pašreizējā loga *dzēšana*

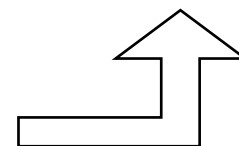
`removewindow`

Divas pārejas starp logiem: piemērs

```
makewindow(1, 0, 1, " First ", 0, 0, 25, 80),  
makewindow(2, 112, 112, " Second ", 8, 7, 5, 25),  
write("222"),  
makewindow(3, 112, 112, " Third ", 6, 5, 5, 25),  
write("333"),  
shiftwindow(2), readchar(_).
```



Lai pāreja notiek ar `gotowindow` palīdzību:



Informācija par *aktuālo* logu

`shiftwindow(X) %2`

Piezīme: **n**eder variants `gotowindow(X)`.

Rezultāts: izpildes kļūda.

7. *Kursora* pozicionēšana

`cursor(<Row>, <Col>)`

Pāriet uz 5 kolonām uz priekšu un izvadīt X:

`cursor(R, C), C1 = C + 5, cursor(R, C1),
write(X).`

8. *Norādītā* loga dzēšana

`removewindow(<WinNo>, <RefreshBehind>)`

Vērtības 0/1 (neatjaunināt/atjaunināt apakšējo logu).

Grafisko iespēju lietošana

1. Grafiskā režīma *draivera iegūšana*

`detectgraph(integer <GrDrv>, integer <GrMd>)`

2. Grafiskā režīma *inicializācija*

`initgraph(integer <GrDrv>, integer <GrMd>,
integer <NewDrv>, integer <NewMd>, string
<PathToDriver>)`

3. Nepieciešams norādīt ceļu pie faila `egavga.bgi`.

4. Grafiskās funkcijas ir līdzīgas Pascal un C funkcijām.

`setbkcolor(...), rectangle(...), bar(...)`

5. *Izeja* no grafiskā režīma

`closegraph`

Grafiskās programmas *karkass*

Programmā izmantota *izņēmumu apstrādes* imitācija

predicates

```
graphstart
```

```
myerror(integer)
```

clauses

```
myerror(E) :- cursor(0, 0),  
    writef("Error : %.", E).
```

```
graphstart :-  
    detectgraph(Gdrv, Gmd),  
    writef("Driver #%, mode #%\n",  
        Gdrv, Gmd), readchar(_),  
    initgraph(Gdrv, Gmd, _, _, "" ),
```

```
getmaxx(X), X1=X div 2,  
getmaxy(Y), Y1=Y div 2,  
setbkcolor(0), setfillstyle(1, 6),  
bar(0, 0, X1, Y1),  
rectangle(10, 10, 70, 40),  
readchar(_), closegraph.
```

goal

```
trap(graphstart, X, myerror(X)).
```

Fails egavga.bgi atrodas aktuālajā katalogā.

SWI-Prolog lietošana. Edinburgas stils.

1. Programmas pirmteksts (`colors.pl`).

```
color(red) .  
color(green) .  
color(blue) .
```

Piezīme: nav daļu **predicates** un **clauses**.

2. Pirmteksta nolasīšana: `File`→`Consult...`

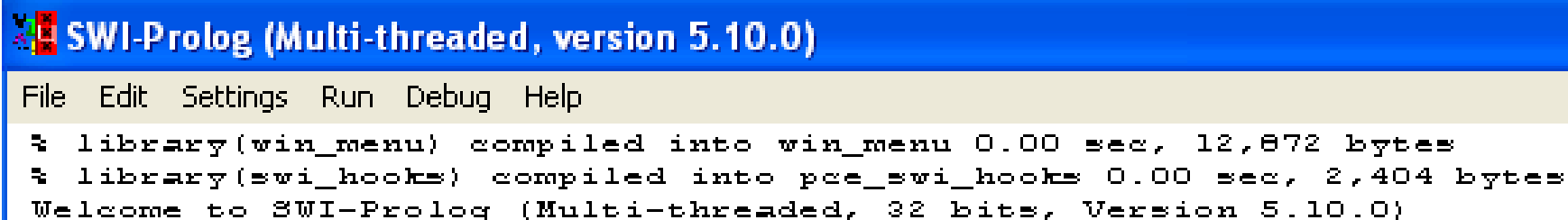
3. Jautājums sistēmai: `?- color(X) .`

4. Atbildes (pēc katras atbildes nospiesta *atstarpe*):

```
X = red ;  
X = green ;  
X = blue .
```

Piezīme: nospiest *Enter* nozīmē *pārtraukt* informācijas izvadi.

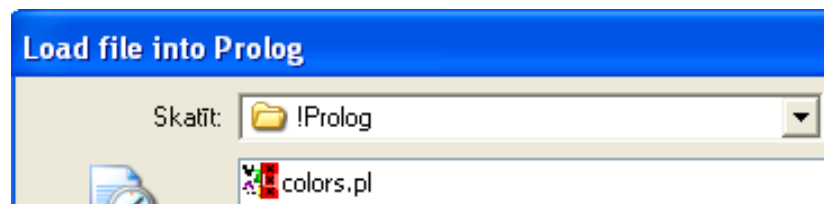
1. SWI-Prolog vide



```

SWI-Prolog (Multi-threaded, version 5.10.0)
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.00 sec, 12,872 bytes
% library(swi_hooks) compiled into pce_swi_hooks 0.00 sec, 2,404 bytes
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.10.0)
    
```

2. Programmas pirmteksta nolasīšana



3. Nolasīšanas rezultāts

```

1 ?-
% c:/!Prolog/colors.pl compiled 0.00 sec, 1,308 bytes
    
```

4. Jautājums sistēmai

```

1 ?- color(X) .
    
```

5. Sistēmas atbilde (1). Nospiesta atstarpe.

```
X = red
```

6. Sistēmas atbilde (2). Nospiesta atstarpe.

```
X = red ;  
X = green
```

7. Sistēmas atbilde (3).

```
X = red ;  
X = green ;  
X = blue .
```

Piezīme: nospiest *Enter* – pabeigt alternatīvo atbilžu izvadi.

Lai pēc atbildes 'X = green' tika nospiests *Enter*.

```
X = red ;  
X = green .
```

8. Jautājumu *atkārtošana* Prolog komandrindā: ↑ un ↓.
Var atkārtot *ne tikai* pēdējo jautājumu.

9. Var atkārtoti ielādēt *modificēto* failu.
Lai failā ir jauns fakts: `color(cyan)`.

Komanda: `File` → `Reload modified files`

`Reload modified files`

```
% c:/!prolog/colors.pl compiled 0.00 sec, 544 bytes
% Scanning references for 1 possibly undefined predicates
```

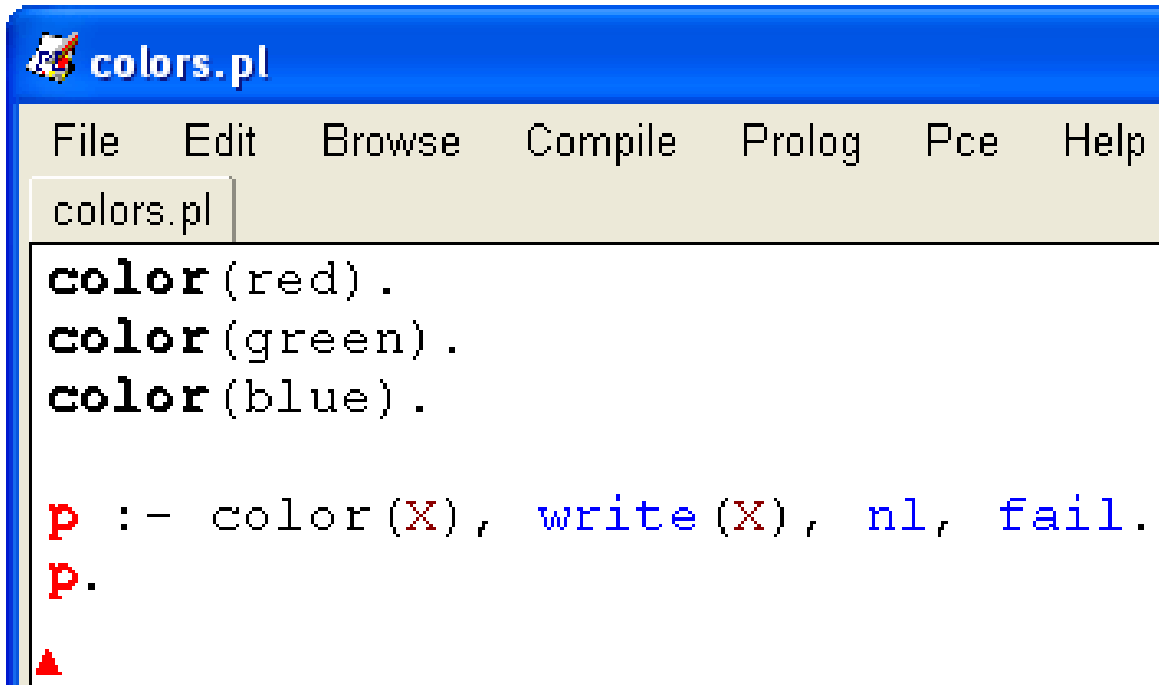
Tagad sistēmai būs viena papildu atbilde: `'X = cyan'`.

10. Lai programmā ir predikāts:
`p :- color(X), write(X), nl.`

Pēc `p` izpildes redzēsīm informāciju *par visām* krāsām.

11.Var izmantot *iebūvēto* programmas redaktoru.

File → Edit... vai File → New...



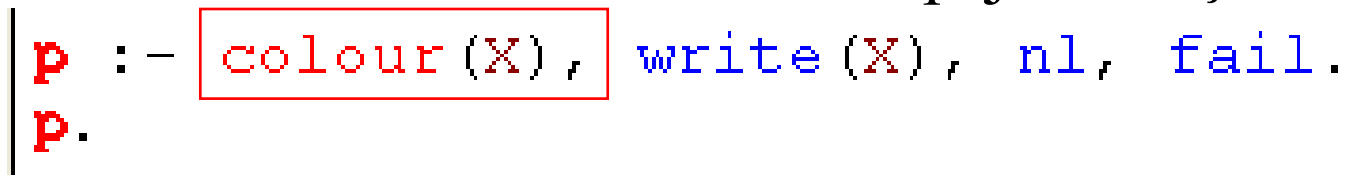
```

color(red) .
color(green) .
color(blue) .

p :- color(X), write(X), nl, fail.
p.

```

Redaktors automātiski iezīme iespējamās kļūdas.



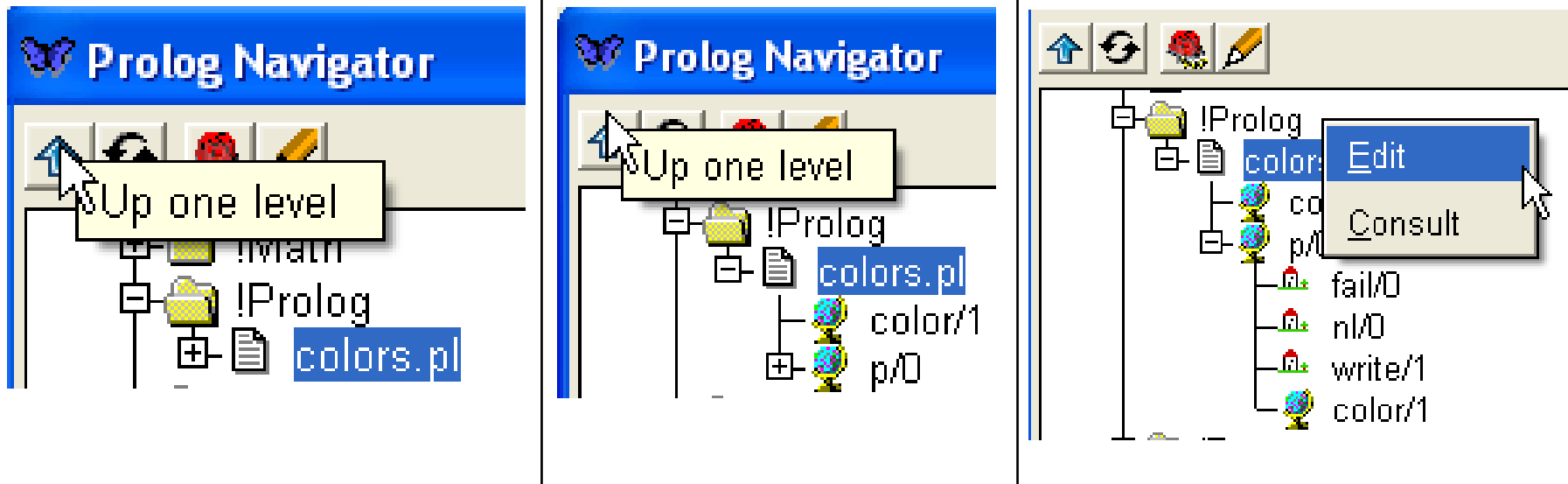
```

p :- colour(X), write(X), nl, fail.
p.

```


12. Jau *eksistējošās* programmas atvēršana:

File → Navigator..., pēc tam peles LP.



13. Programmas *kompilācija*:

Compile → Compile buffer

Pēc tam – *pārslēgšana* uz SWI-Prolog vidi (Alt + Tab).

```
% d: / !Prolog / colors.pl compiled 0.00 sec, 604 bytes
```

```
?- p.
```

Izsaukt Help-failu:

```
?- help.
```

Iegūt informāciju par kādu valodas mehānismu vai predikātu:

```
?- help(!) . %Cut.Discard choice points...
```

```
?- help(->) . %:Condition...
```

```
?- help(false) . %Same as fail,...
```

```
?- help(assert) . %...
```

Iegūt informāciju par kādu *jēdzienu*:

```
?- explain(+).
```

```
% "+" is a prefix... operator of priority...
```

```
?- !!. %atkārtot iepriekšējo jautājumu
```

```
?- !nr. %atkārtot konkrēto jautājumu
```

```
?- h. %jautājumu un to numuru saraksts
```

Termu *identiskuma* pārbaude:

```
?- color(red) == color(red) .      %true
?- color(red) == color(green) .    %false
?- color(red) == human(red) .      %false
```

Rezultāts ir **true**, ja divi termi *identiski*:

1. Termiem ir vienāda *struktūra*.
2. Termiem ir vienādas *komponentes*.

Var pārbaudīt termu *neidentiskumu*:

```
?- color(red) \== color(red) .      %false
?- color(red) \== color(green) .    %true
```

Faktu piešķire:

```
?- color(red)=X, write(X) . %X=color(red)
?- X=color(red), write(X) . %X=color(red)
```

Elementārie *aritmētiskie* aprēķini

```
p:- X=1+2, Y=X+4, write(X), write(" "),
    write(Y).
```

```
?- p.
```

Rezultāts: 1+2 [32] 1+2+4

Komentāri:

1. **N**eder variants: `write(X, " ", Y)`. Predikātā `write(...)` *ir tikai viens* izvadāmais parametrs.
2. **N**enotiek saskaitīšana. `1 + 2` ir terms; `+` ir funktors.
Aritmētikā nepieciešams izmantot operāciju **is**.

```
p:- X is 1+2, Y is X+4, ... %X=3, Y=7
```

3. Tika izvadīts *simbola kods*. Simbola izvade:

```
p:- ..., put(' '), ... %3 7
```

Var arī izmantot `write(' ')`, bet tad *jābūt* apostrofi.

Teksta rindiņu un simbolu *izvade* un *apstrāde*

```
?- write("abc") .    %[97,98,99]
```

```
?- write('abc') .   %abc
```

```
?- put("a") .       %a
```

```
?- put('a') .       %a
```

```
?- put(97) .         %a
```

```
?- put('ab') .      %klūda
```

Predikāts `put(...)` izvada ekrānā *tikai vienu* simbolu.

```
?- concat('Pro', 'Log', S) .
```

```
%S = 'ProLog'
```

Ne visi Turbo Prolog predikāti funkcionē citās versijās.

```
?- frontstr(1, "abc", First, Last) .
```

```
Undefined procedure: frontstr/4 (DWIM
could not correct goal)
```

Simbola *koda* iegūšana

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

Skaitļa *ciparu saraksta* iegūšana

```
?- number_chars(2011, X) .
%X=['2', '0', '1', '1']
```

Līdzīgo efektu var iegūt, strādājot ar jebkuru *atomu*

```
?- atom_chars(maijs, X) .
%X=[m, ā, i, j, s]
?- atom_chars(2011, X) .
%X=['2', '0', '1', '1']
```

Atoma *simbolu kodu* iegūšana

```
?- atom_codes(maijs, X) .
%X=[109, 97, 105, 106, 115]
```

Līdzīgi izmanto predikātu `number_codes(..., ...)`.

Nosacījuma operatora `if - then - else` implementēšana.

```
check_pos(X) :- X>0 -> write('Positive. ');  
                write('NOT Positive.').
```

Predikāta `check_pos(...)` izsaukuma rezultāti:

```
?- check_pos(1).    %Positive.
```

```
?- check_pos(-1).  %NOT Positive.
```

Piezīme: lai operatora `->` vietā ir komats.

```
?- check_pos(1).
```

%Sākumā `Positive`, tālāk:

a. Pēc atstarpes nospiešanas: `NOT Positive`.

b. Pēc Enter nospiešanas: pabeigšana.

```
?- check_pos(-1).  %NOT Positive.
```

Var izpildīt *precīzāku* skaitļa pārbaudi.

Tiks pārbaudīti trīs gadījumi: <0 , 0 , >0 .

```
?- check_pos(X) :-  
    X>0 -> write('Positive. ');  
    X<0 -> write('Negative. ');  
    write('Zero.').
```

Piezīme: to pašu rezultātu var nodrošināt Turbo Prolog stilā:

```
?- check_pos(X) :-  
    X>0, !, write('Positive. ');  
    X<0, !, write('Negative. ');  
    write('Zero.').
```


Teksta rindiņu apstrāde

```
?- string_length('Prolog', X) . %X=6
```

Piezīme: **neder** `str_len(...)` no Turbo Prolog.

```
?- substring('Prolog', 2, 3, X) %X="rol"
```

Piezīme: 2 ir sākotnējā pozīcija, bet 3 ir simbolu daudzums.

```
?- string_concat('Pro', 'log', X) .  
%X="Prolog"
```

Piezīme: var arī izmantot `concat(...)` no Turbo Prolog.

```
?- string_to_list('Prolog', L) .  
%L = [80, 114, 111, 108, 111, 103]
```

```
?- string_to_atom('prolog', X) . %X = prolog
```

Piezīme: apostrofu nav, jo pirmais simbols ir apakšējā reģistrā.

Piešķire un *neinicializētais* mainīgais

```
?- X is 2, Y = X, write(Y) . %2
```

```
?- X is 2, X = Y, write(Y) . %2
```

Abos gadījumos notiek mainīgā Y konkretizācija.

Aritmētiskais salīdzināšanas operators `==` nekad *nekonkretizē* mainīgo.

```
?- X is 2, X == Y, write(Y) .
```

Arguments are not sufficiently instantiated

Līdzīgā kļūda būtu operatora `is` izmantošanas gadījumā.

```
?- X is 2, Y is 2, X == Y, write(Y) . %2
```

```
?- X is 2, Y is 3, X == Y, write(Y) .
```

Pēdējā gadījumā ekrānā *nekas* netiks izvadīts.

Nevienādības pārbaude: `=\=`

```
?- X is 2, Y is 3, X =\= Y, write(Y) . %3
```

Aritmētiskā vienādības pārbaude

?- X is sin(0), X = 0. %false

?- X is sin(0), X == 0. %true

?- X is sin(0), X \= 0. %false

?- 0 is sin(0). %false

?- sin(0) is 0. %false

To pašu rezultātu iegūsim, ja is vietā būs piešķires operators.

?- sin(0) = 0. %false

...

Savukārt:

?- [1, 2] = [1, 2]. %true

?- [1, 2] is [1, 2]. %klūda

?- [1, 2] == [1, 2]. %klūda

Termu *prioritātes* augošajā secībā

1. Mainīgie.
2. Skaitļi.
3. Atomi.
4. Teksta rindiņas.
5. Sarežģītie termi.

Termu prioritāšu *salīdzināšana*: @<, @>, @=<, @>=.

```
?- X @> 2.           %false
?- 'abc' @> 2.        %true
?- f(X) @> 'abc'.    %true
?- 2 @< 3.           %true
?- 2 @= 3.           %klūda - tāda operatora nav
```

Izmanto arī `compare(O, T1, T2)` ar $O=\{<, >, =\}$.

```
?- compare(=, 2, 2). %true
?- compare(=, 2, 3). %false
```

Mainīgo *inicializācijas* pārbaude

1. `var (X)` – mainīgais *nav* inicializēts.
2. `nonvar (X)` – mainīgais *ir* inicializēts.

Saskaitīšanu var izpildīt, tikai ja *abi operandi* inicializēti.

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

Ja X vai Y nav inicializēti, izteiksmes rezultāts ir `false`.

Lai mainīgie X un Y **ne**inicializēti. Tad Z jābūt 0.

?- `var (X), var (Y), Z is 0. %0`

Masīvu radīšana un elementu iegūšana

1. `functor (<masīva mainīgais>,
 <masīva fakts>, <elementu daudzums>)`
2. `arg (<masīva indekss>,
 <masīva mainīgais>, <elementa vērtība>)`

Masīva radīšanas un daļējās inicializācijas piemērs

```
?- functor(Arr, f, 5), %f(_G292,...,_G296)
   arg(1, Arr, 10), %f(10,_G293,...,_G296)
   arg(2, Arr, 20).
%Arr = f(10,20,_G294,_G295,_G296)
```

Elementa iegūšana pēc indeksa

```
?- arg(2, Arr, X) %20
```

Piezīme: izmainīt inicializētu masīva elementu nevar.

Masīvs var saturēt *nehomogēnu* informāciju:

```
?- functor(Arr, f, 5),
   arg(1, Arr, human(ivars)),
   arg(2, Arr, 20), arg(3, Arr, color(red)),
%f(human(ivars),20,color(red),_G295,_G296)
```

Neeksistējošā elementa adresēšanas rezultāts: false.

```
arg(10, Arr, human(ivars)) %false
```

Informācijas ievade

```
?- readln(X), readln(Y), Z is X+Y,
write(Z).
```

Piezīme: predikāts `readint(_)` neeksistē.

```
%X=1 Y=2 → Z=3
```

```
%X=a Y=1 → Z=98
```

```
%X=1 Y=a → Z=98
```

```
%X=a Y=b → Z=195
```

Termu lasīšana: predikāts `read(_)`.

```
?- read(X), read(Y), Z is X+Y, write(Z).
```

Piezīme: pēc katra terma jābūt punkts.

```
%X=100. Y=25. → Z=125
```

Varianti ar simboliem nestrādā: jābūt konkatēnācija.

```
?- read(X), read(Y), concat(X, Y, Z),
write(Z).
```

```
%X=ab. Y=cde. → Z=abcde
```

Simbola izvade:

```
?- put_char("a") . %true
```

Simbola izvade pēc koda:

```
?- put_code(97) . %a
```

Piezīme: abos gadījumos var norādīt kā *simbolu*, tā arī *kodu*.

```
?- put_char(48), put_code(48),  
   put_code('0'), put_char('0') . %0000
```

Simbola ievade:

```
?- get_char(X) . %a → a
```

Simbola koda ievade:

```
?- get_code(X) . %a → 97
```

%līdzīgi strādā get() un get0().

Piezīme: visos gadījumos var izmantot *papildus* pirmo parametru – plūsmu (kopā būs *divi* parametri).

Termu *tipu* pārbaude

1. `number(X)` – skaitlis.
2. `integer(X)` – vesels skaitlis.
3. `float(X)` – reālais skaitlis.

?- `read(X), read(Y), number(X),
number(Y), Z is X + Y, write(Z).`

`%X=100. Y=25. → Z=125`

`%X=12.5. Y=10.1. → Z=22.6`

`%X=a. Y=b. → false`

?- `read(X), read(Y), integer(X),
integer(Y), Z is X + Y, write(Z).`

`%X=100. Y=25. → Z=125`

`%X=12.5. Y=10.1. → false`

4. compound(X) – struktūra.

```
?- read(X), compound(X),
   write('Structure '), human(Name)=X,
   write(Name).
```

```
%X=human(ivars). → Structure ivars
```

```
%X=animal(cat). → Structure
```

```
%X=1+1. → Structure
```

```
%X=cat. → false
```

5. atom(X) – atoms.

6. atomic(X) – atoms vai skaitlis.

```
?- atom(a). %true           ?- atomic(a). %true
```

```
?- atom(1). %false         ?- atomic(1). %true
```

```
?- atom(bin(0)). %false
```

```
?- atomic(bin(0)). %false
```

7. `is_list(X)` – saraksta pārbaude.

```
?- is_list([1, 2]). %true
?- is_list([1]). %true
?- is_list(1). %false
?- is_list("abc") %true
?- is_list('abc') %false
```

8. `ground(f(X))` – neinizializēto mainīgo pārbaude termā

```
?- ground(human(X, Y)) %false
?- ground(human(uldis, Y)) %false
?- ground(human(uldis, strods)) %true
?- X=uldis, Y=strods, ground(human(X, Y))
%true
```

Faila *lasīšana*

1. `see(F)` – ievade no citas plūsmas.
2. `seeing(F)` – aktuālās plūsmas saglabāšana.
3. `seen` – aktuālās plūsmas aizvēršana.
4. `end_of_file` – faila beigas.

Uzdevums: nolasīt visu informāciju no faila un izvadīt ekrānā.

```
rd_f(F) :- seeing(Curr), see(F),  
           read(S), row(S),  
           seen, see(Curr).  
row(end_of_file) :- !.  
row(S) :- write(S), nl, read(S1),  
          row(S1).  
?- rd_f('C:\\P\\d.txt').
```

Piezīme: jebkura faila rindiņa beidzas ar *punktu*.

1.	'One' .		1	One
22.	'Two' .	→	22	Two
333.	'Three' .		333	Three

Cita iespēja iegūt *to pašu* rezultātu: lasīšana netiks dublēta.

```
rd_f(F) :- seeing(Curr), see(F),
           repeat, read(S), row(S),
           seen, see(Curr).
```

```
row(end_of_file) :- !.
row(S) :- write(S), nl, fail.
```

Pārslēgšana uz ekrānu: vērtība `user`.

```
see(user).
```

Ierakstīšana failā

1. `tell(F)` – izvade citā plūsmā.
2. `telling(F)` – aktuālās plūsmas saglabāšana.
3. `told` – aktuālās plūsmas aizvēršana.

Uzdevums: nolasīt informāciju no tastatūras un izvadīt failā.
Ievade beidzas pēc rindiņas `END`.

```
wr_f(F) :- telling(Curr), tell(F),  
           read(S), row(S),  
           told, tell(Curr).  
row(end_of_file) :- !.  
row(S) :- write(S), nl, read(S1),  
          row(S1).
```

Pārslēgšana uz ekrānu: vērtība `user`.

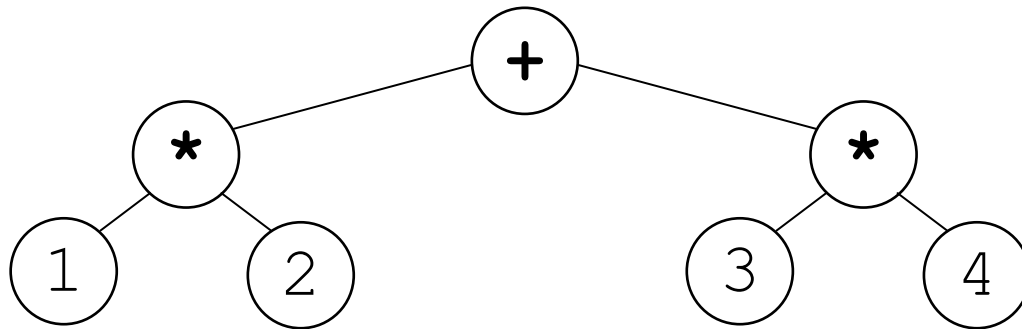
Operatoru formas lietošana

?- X = 2*3. %X=2*3

?- X is 2*3. %X=6

?- X is *(2,3) . %X=6

Infiksa operāciju gadījumā operatoru var lietot kā *funktoru*.
Aritmētisko izteiksmi var interpretēt kā *koku*.



?- X is 1*2 + 3*4. %X=14

?- X is + *(1,2) + *(3,4) . %X=14

?- X is + (* (1,2) + * (3,4)) . %X=14

Var *patstāvīgi* deklarēt jaunas operācijas.

Tādiem nolūkiem lieto speciālos teikumus: *direktīvas*.

Direktīvas op parametri: *prioritāte, operācijas tips, operācija*.

Piemērs: var noformēt operāciju `pieder(kas, kam)`.

?- `op(600, xfx, pieder)`.

?- `assert(pieder(audi, uldis))`.

?- `assert(volvo pieder ivars)`.

1. piezīme: programmā *nav* daļas **database**.

2. piezīme: `pieder` tika izmantots *operatora* formā.

Pieprasījums sistēmai:

?- `X pieder Y`. vai ?- `pieder(X, Y)`.

Rezultāti:

`X=audi, Y=uldis ; X=volvo, Y=ivars.`

Operāciju grupas:

1. *Infiksa* operācijas: **xx** **fy** **yfx**.

2. *Prefiksa* operācijas: **fx** **fy**.

?- op(600, fx, human).

?- assert(human uldis).

3. *Postfiksa* operācijas: **xf** **yf**.

?- op(600, xf, human).

?- assert(uldis human).

f ir operācijas zīme; **x** un **y** ir operandi.

x gadījumā operanda prioritāte ir *mazāk* par operācijas prioritāti; **y** gadījumā operanda prioritāte ir *mazāk vai vienāds*.

Tas atļauj pareizi izpildīt secīgas operācijas ar vienu prioritāti.

Piemērs: programmēšanas valodu zināšana.

Ir ieplānots noformulēt teikumu:

Ivars knows C++ and Java and C#.

?- op(600, **xfx**, knows) .

?- op(500, **xfy**, and) .

?- assert('Ivars' knows 'C++' and 'Java'
and 'C#') .

Informācija par cilvēku un *visām* programmēšanas valodām:

?- knows(X, Y) .

%X='Ivars', Y='C++' and 'Java' and 'C#'

Informācija par cilvēku un *katru* programmēšanas valodu:

?- knows(X, and(A, and(B, C))) .

%X='Ivars', A='C++', B='Java', C='C#'

Eksistējošo operatoru prioritāšu nomaiņa

?- X is 2*3+4. %X=10.

Lai reizināšanas operatora prioritāte ir 2, bet saskaitīšanas: 1.

?- op(2, **xfy**, *).

?- op(1, **xfy**, +).

?- X is 2*3+4. %X=14.

Dažu operatoru *prioritātes*:

op(200, **fy**, -).

op(200, **xfy**, ^). ?- X is 2^3. %X=8.

op(200, **xfy**, **). ?- X is 2**3. %X=8.

op(400, **yfx**, [*, /, //, mod]).

?- X is 5/2 %X=2.5 ?- X is 5//2. %X=2

op(1200, **xfx**, [:-, ?-]).

Unifikācijas operācija

```
?- X=1, =(Y, X) .           %X = 1, Y = 1
?- Y=1, =(Y, X) .           %Y = 1, X = 1.
?- X=1, Y=1, =(Y, X) .      %X = 1, Y = 1
?- X=1, Y=2, =(Y, X) .      %false
```

Predikāts not (...)

```
?- assert(human(ivars)) .
?- not(human(uldis)) .      %true
?- not(human(ivars)) .      %false
?- \+(human(ivars)) .       %false
```

Operators `\+` ir operatora `not` ekvivalents.

Sarakstu veidošana *faktu apstrādes* procesā

Lai ir krāsu deklarācijas RGB modelī:

```
color(blue, 1) .      color(green, 2) .
color(cyan, 3) .      color(red, 4) .
color(magenta, 5) .    color(yellow, 6) .
```

Iegūt krāsu numuru sarakstu:

```
?- findall(X, color( , X), L) .
```

Iegūt *krāsu numuru* sarakstu, kur numuri *pārsniedz* kādu iepriekš norādīto vērtību:

```
?- findall(X, (color( , X), X>3), L) .
```

Iegūt *krāsu nosaukumu* sarakstu, kur numuri *pārsniedz* kādu iepriekš norādīto vērtību:

```
?- findall(X, (color(X, Y), Y>3), L) .
%[red,magenta,yellow]
```

Iegūt *krāsu* secību, *sakārtoto* alfabētiskajā kārtībā:

```
?- setof(C, X^color(C, X), L).
% [blue, cyan, green, magenta, red, yellow]
```

Piezīme: operators \wedge norāda, ka X vērtība mūs neinteresē.

Bez operatora \wedge iegūsim sešus sarakstus no viena elementa.

```
?- setof(C, color(C, X), L).
% [blue] [green] ... [yellow]
```

Saraksta veidošanas procesā dublikāti *netiks* iekļauti sarakstā.

Lai ir fakti ar elementu dublikātiem (divas krāsas *red*).

Iegūt sarakstu ar dublikātiem:

```
?- bagof(C, X^color(C, X), L).
```

Atšķirība starp `findall` un `bagof`: nosacījuma neizpildes rezultātā iegūsim `[]` un **false** attiecīgi.

Saraksta deklarēšana: *papildu* iespējas.

```
list(. (red, . (green, . (blue, []))) ) .
```

Galvas un astes nolasīšana:

```
?- list([H|T]) .
```

Rezultāts:

```
H = red, T = [green, blue] .
```

Saraksts ir vienāds sarakstam [red, green, blue].

```
list1(. (red, . (green, . (blue, []))) ) .
```

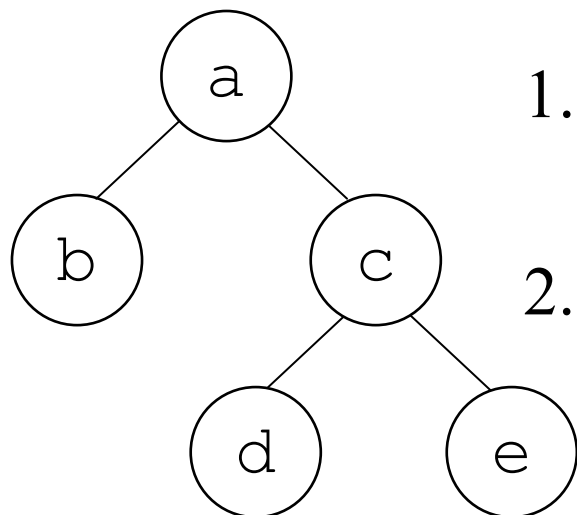
```
list2([red, green, blue]) .
```

```
p :- list1(A), list2(B), A=B,  
    write('OK') .
```

Predikāta p palaišanas rezultāts:

```
OK true.
```

Bināro koku pārstāvēšana un apstrāde ar Prolog palīdzību



1. Binārā koka pārstāvēšanai lieto funktoru `tree(Left, Root, Right)`.
2. Tukšā koka aprakstīšanai izmanto kādu simbolu (piemēram, `nil`).

Virsotnes `X` meklēšana binārajā kokā.

`X` atrodas kokā `Tree`, ja:

1. `X` ir `Tree` *sakne*, vai
2. `X` atrodas *kreisajā* apakškokā `Left`, vai
3. `X` atrodas *labajā* apakškokā `Right`.

Meklēšanas predikāts:

```
in(X, tree(_, X, _)) .  
in(X, tree(Left, _, _)) :- in(X, Left) .  
in(X, tree(_, _, Right)) :- in(X, Right) .
```

Koka deklarēšana un elementa meklēšana:

```
T = tree(  
    tree(nil, b, nil), a,  
    tree(  
        tree(nil, d, nil), c,  
        tree(nil, e, nil))  
), in(e, T). %true
```

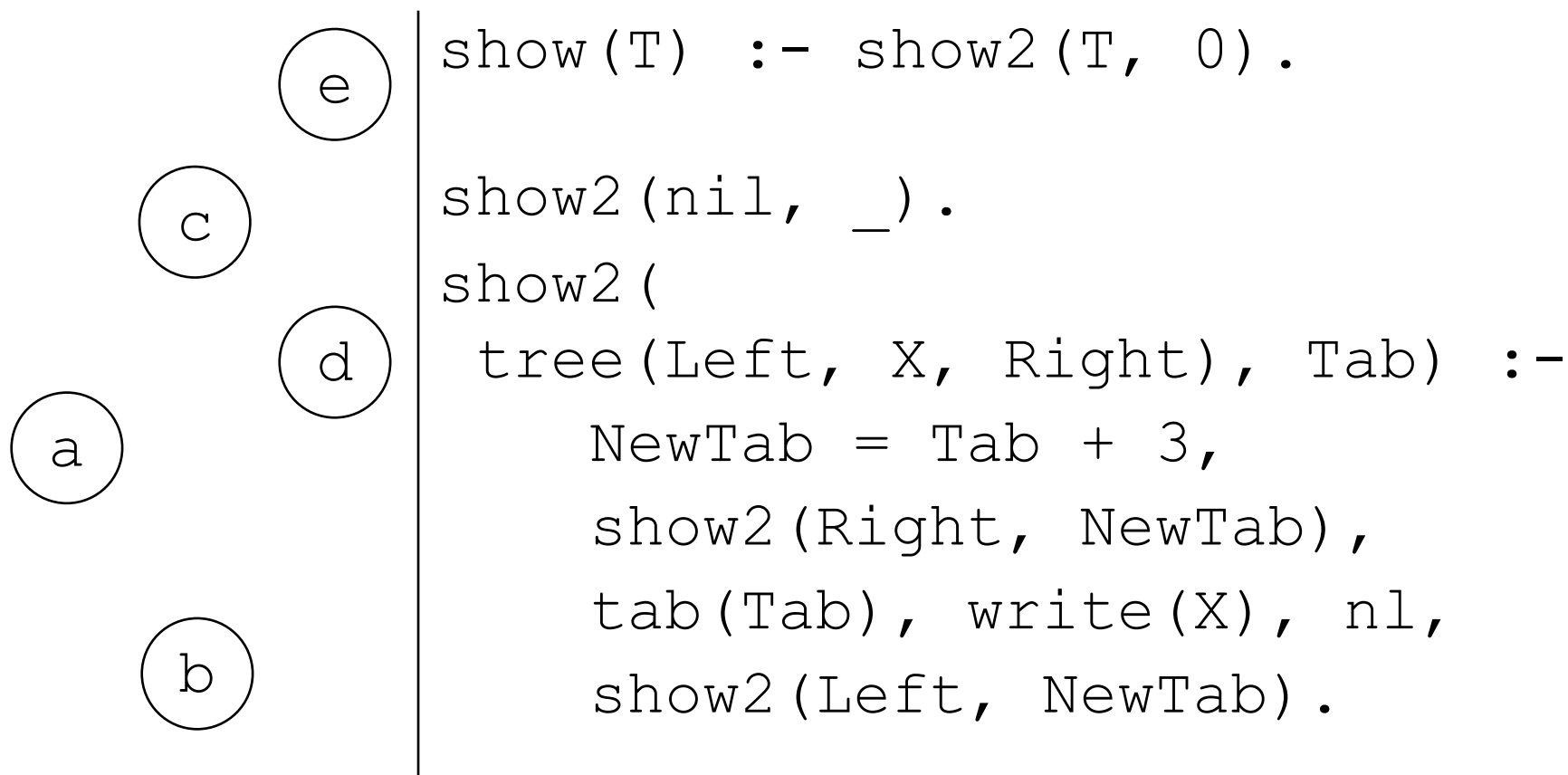
Piezīme: virsotne nil *nekad* netiks atrasta.

```
..., in(nil, T). %false
```

Tāds meklēšanas algoritms *nepaaugstina ražīgumu*.

Faktiski, situācija ir līdzīga *saraksta* apstrādei.

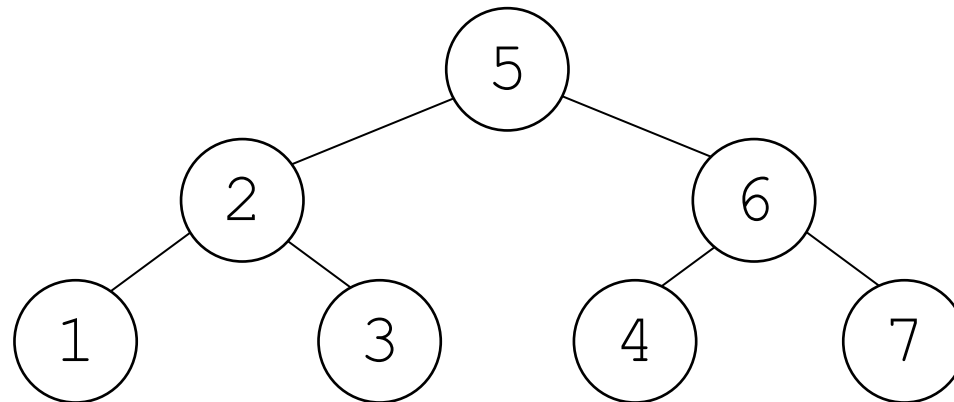
Saraksta izvade ekrānā:



Kārtotā binārā koka deklarēšana

Jebkurai virsotnei X izpildās divas prasības:

1. Visi mezgli *kreisajā* apakškokā ir *mazāk* par X.
2. Visi mezgli *labajā* apakškokā ir *lielāk* par X.



```
T = tree(  
    tree(tree(nil, 1, nil), 2,  
        tree(nil, 3, nil)), 5,  
    tree(tree(nil, 4, nil), 6,  
        tree(nil, 7, nil)))
```

Elementa *meklēšana* kārtotajā binārajā kokā

Meklēšana vienmēr notiek *tikai vienā* apakškokā

Pašā sākumā netiks apskatīts vismaz viens apakškoks.

1. Ja X ir sakne $Root$, tad X ir atrasts.
2. Citādi: ja X ir mazāk par $Root$, tad meklēt kreisajā apakškokā.
2. Citādi: ja X ir lielāk par $Root$, tad meklēt labajā apakškokā.

```
in(X, tree(_, X, _)).
in(X, tree(Left, Root, _)) :-
    Root > X, in(X, Left).
in(X, tree(_, Root, Right)) :-
    Root < X, in(X, Right).
```

Ierakstāmās datubāzes (“recorded database”)

Termi un termu ķēdītes saistīti ar atslēgām.

Tas strādā *ātrāk*, nekā assert/retract.

1. recorda(↓atslēga, ↓terms) %pirmais
2. recordz(↓atslēga, ↓terms) %pēdējais
3. recorded(↓atslēga, ↑vērtība) %lasīšana

Datubāzes veidošana

```
?- recorda(1, human(uldis)),
   recorda(2, human(ivars)),
   recordz(2, human(aldis)).
```

Informācijas iegūšana

```
?- recorded(1, X), write(X).
%human(uldis)
```

?- recorded(2, X), write(X).

%human(ivars) atstarpe human(aldis)

Piezīme: šajā gadījumā atslēga 2 saistīta ar termu ķēdīti.

Visiem *trim* augstāk minētajiem predikātiem ir arī *cita forma*.

1. recorda(↓atslēga, ↓terms, ↑norāde)
2. recordz(↓atslēga, ↓terms, ↑norāde)
3. recorded(↓atslēga, ↑vērtība, ↑norāde)

Termu *dzēšana* un *aizstāšana*:

4. erase(↓norāde) %dzēšana
5. flag(↓atslēga, ↑vecais, ↓jaunais)

Izdzēst ierakstu ar atslēgu 1:

recorded(1, _, Ref), erase(Ref).

Norādes var arī izmantot darbā ar predikātiem `assert(...)`.

1. `assert(↓terms, ↑norāde)`
2. `asserta(↓terms, ↑norāde)`
3. `assertz(↓terms, ↑norāde)`

Termu *pievienošana*:

```
?- assert(human(uldis), RefU),
    assert(human(ivars), RefI),
    assert(human(aldis), RefA).
```

Predikātu var izdzēst *parastajā* stilā:

```
?- retract(human(uldis)).
```

Predikātu var arī izdzēst *pēc norādes*:

```
?- assert(..., RefU), ..., erase(RefU).
```

Piezīme: norāde ir *vesels skaitlis*. Piemēram:

```
RefU = <clause>(01262A00)
```