15. Programmas teikumu vadība un citi specifiski mehānismi

Nodaļas saturs

15.1. Programmas ielādēšana

Programmas ielādēšana notiek ar predikātu *consult/1*, kura vienīgais parametrs ir fails vai failu saraksts:

```
?- consult('terms.pl').
```

vai

```
?- consult([list,list2]).
```

Faila nosaukumā drīkst izlaist paplašinājumu ".pl".

Ja Prolog interaktīvā vide nav palaista dotajā direktorijā, jāuzrāda pilns faila nosaukums:

```
?- consult('c:\\mydirectory\\terms.pl').
```

Ja programmā (t.i., failā) ir veiktas izmaiņas, vēlreiz šim failam vai failiem jāpielieto predikāts *consult/1* (citās sistēmās šim nolūkam ir predikāts *reconsult/1*).

Ja ielādējot programmu, tiks sastaptas kļūdas, tās kopā ar atrašanās vietu tiks paziņotas.

GNU-Prolog piedāvā vienkāršotu programmas ielādēšanu – izsaucot sarakstu ar failu nosaukumiem:

```
?- [terms].
```

15.2. Programmas teikumu dinamiska vadība

15.2.1. Teikumu pievienošana ar assert.../1

Jaunus teikumus programmai var pievienot arī dinamiski – izmantojot predikātus *asserta/1* un *assertz/1*:

Vārds "assert" angļu valodā nozīmē apgalvot, bet burti "a" un "z" ir attiecīgi alfabēta pirmais un pēdējais burts, no kā izriet arī šo predikātu definīcijas:

asserta(C) pievieno teikumu *C* Prolog sistēmas iekšējai datu bāzei **pirms pirmā** teikuma, kas attiecas uz doto predikātu.

assertz(C) pievieno teikumu *C* Prolog sistēmas iekšējai datu bāzei **pēc pēdējā** teikuma, kas attiecas uz doto predikātu.

Kā jau iepriekš ir ticis pieminēts, faktu un noteikumu secībai (atšķirībā no matemātiskās loģikas vai gramatikām) Prolog sistēmā ir nozīme.

```
?- asserta(object(first)).

yes
?- assertz(object(after)).

yes
?- asserta(object(before)).

yes
?- object(A).

A = before ? a

A = first

A = after

yes
```

15.2.2. Statiskas un dinamiskas procedūras

GNU-Prolog neļauj dinamiski pievienot jaunu teikumu, ja sistēmā jau eksistē dotā procedūra (predikāts ar noteiktu argumentu skaitu), kas ielādēta no faila (t.i., **statiska procedūra**).

Tādējādi, ja mums ir programma *clauses.pl*:

```
object(first).
```

un mēs esam viņu ielādējuši sistēmā:

```
?- consult(clauses).
compiling C:\...\clauses.pl for byte code...
C:\...\clauses.pl compiled, 0 lines read - 315 bytes written, 15
    ms

yes
```

tad nākošos teikumus GNU-Prolog dinamiski neļauj pievienot:

```
?- assertz(object(after)).
uncaught exception: error(permission_error
    (modify,static_procedure,object/1),assertz/1)
```

Tomēr, izmantojot lasīšanu no faila, varam failā esošo programmu ielādēt dinamiski. Seko triviāls piemērs, kas no faila *clauses.pl* ielādē tikai pirmo teikumu:

```
?- see('clauses.pl'), read(C), asserta(C), seen.
```

```
C = object(first)

yes
?- assertz(object(after)).

yes
?- asserta(object(before)).

yes
?- object(A).

A = before ? a

A = first

A = after

yes
```

read/1 ir termu līmeņa nolasīšanas procedūra, kas nolasa visu struktūru līdz punktam, t.i. visu teikumu.

Ir viegli uzkonstruēt predikātu *consultdynamic/1*, kas nolasa programmas failu un dinamiski to ielādē atmiņā:

```
consultdynamic(F):-see(F), readassert, seen.
readassert:-read(C), (C=end_of_file -> !; assertz(C),
    readassert).
```

Šis predikāts nolasa no faila F visus teikumus, katru no viņiem pievieno iekšējās datu bāzes beigās ar *assertz/1* un beidz darbu, kad sastaptas faila beigas.

15.2.3. Teikumu izmešana ar retract/1 un retractall/1

retract(C) izmet no datu bāzes **pirmo teikumu**, kas ir savietojams ar C.

```
?- assertz(object(viens)),assertz(object(divi)),object(X).

X = viens ? a

X = divi

yes
?- retract(object(viens)),object(X).

X = divi

yes
?- asserta(object(viens)),object(X).

X = viens ? a

X = divi
```

```
yes
?- retract(object(_)),object(X).

X = divi ? a

no
?- retract(object(_)),object(X).

no
```

retractall(H) izmet no datu bāzes **visus teikumus**, kuru **galvas** ir savietojamas ar H.

```
?- assertz(object(viens)),assertz(object(divi)),object(X).

X = viens ? a

X = divi

yes
?- retractall(object(_)),object(X).

no
```

15.2.4. Visas procedūras izmešana ar abolish/1

abolish(P) izmet no datu bāzes **procedūru**, kuru identificē predikāta indikators P (ietver predikāta nosaukumu un argumentu skaitu):

```
?- assertz(object(viens)),assertz(object(divi)),object(X).

X = viens ? a

X = divi

yes
?- abolish(object/1),object(X).
uncaught exception:
    error(existence_error(procedure,object/1),top_level/0)
```

abolish/1 atšķirība no *retractall/1* ir ne tikai sintaksē – pēc *abolish/1*, izsaucot attiecīgo procedūru, iestājas eksistences kļūda, bet pēc *retractall/1*, kaut arī izmesti visi teikumi, atbilde ir vienkārši *no*.

15.3. Predikāts repeat/0 cikla modelēšanai bez rekursijas

Nodaļā 15.2.2 parādītai predikāts *consultdynamic/1* veic termu nolasīšanu no faila, cikla veidošanai izmantojot rekursiju predikātā *readassert*, kā jau tas valodā Prolog "pierasts":

```
consultdynamic(F):-see(F), readassert, seen.
readassert:-read(C), (C=end_of_file -> !; assertz(C),
    readassert).
```

Tomēr šajā uzdevumā ar rekursiju pēc būtības nav nekāda sakara, bet rekursijas dziļums izpildes laikā palielināsies līdz teikumu skaitam failā.

Predikāts *repeat/0* dod iespēju ar Prolog stila līdzekļiem izvairīties no rekursijas lietošanas.

repeat/0 ģenerē bezgalīgu atpakaļkāpšanās izvēļu virkni, ko var apstādināt ar atšķelšanas predikātu (!)/0, bet nodrošināt izpildes nonākšanu līdz tam – ar papildus *fail/0* predikāta pielietošanu (fragments no programmas faila *clauses2.pl*):

```
consultdynamic2(F):-see(F), readassert2, seen.
readassert2:-repeat, read(C),
                  (C=end_of_file -> !; assertz(C), fail).
```

Predikāts *consultdynamic2/1* strādā funkcionāli identiski *consultdynamic/1*.

GNU-Prolog cikla modelēšanai pieejams arī predikāts *for/3*.

15.4. Visu risinājumu iegūšana

Predikāta izsaukumam var būt vairāki risinājumi, un interaktīvajā vidē tos ir iespēja iegūt, ievadot semikolu (;) nākošajam risinājumam vai burtu a visiem atlikušajiem. Prolog dod iespēju visus risinājumus (vai to fragmentus) apkopot vienā kopējā struktūrā. Šim nolūkam pieejami 3 dažādi predikāti: *findall/3*, *bagof/3* un *setof/3*.

Predikātu ilustrēšanai tiks izmantota procedūra *contains/2* no programmas *list.pl*.

bagof(T,P,L) atgriež sarakstā *L* visus risinājumus pēc šablona *T*, kas iegūti, izpildot predikātu *P*:

```
?- bagof(A,contains([a,b,a,c],A),L).

L = [a,b,a,c]

yes
```

setof(T,P,L) strādā tieši tāpat, bet izvada sakārtotu sarakstu un neiekļauj dublikātus:

```
?- setof(A,contains([a,b,a,c],A),L).

L = [a,b,c]

yes
```

Ja šablons T neaptver visus mainīgos, kas ietilpst predikāta izsaukumā P, tad *bagof/3* un *setof/3* atgriež sarakstus pa porcijām un porciju skaitu nosaka šablonā neiekļauto mainīgo iespējamo konkretizāciju skaits:

```
?- bagof(A,conc(A,B,[a,b,a,c]),L).

B = []
L = [[a,b,a,c]] ? a

B = [a,b,a,c]
L = [[]]

B = [a,c]
L = [[a,b]]
```

```
B = [b,a,c]
L = [[a]]

B = [c]
L = [[a,b,a]]

yes
```

findall(T,P,L) no *bagof/3* atšķiras ar to, ka vienmēr atgriež visu risinājumu sarakstu vienā porcijā:

```
?- findall(A,conc(A,B,[a,b,a,c]),L).

L = [[],[a],[a,b],[a,b,a],[a,b,a,c]]

yes
```

15.5. Atdalītājsimbolu izmatošana

Dažādu atdalītājsimbolu lietošana termos nodrošina labāku programmas lasāmību un ir kā vienkāršota alternatīva saliktajiem termiem un fiksēta garuma sarakstiem.

Atdalītājsimbolu izmantošana tiks apskatīta uz piemēru pamata.

Aritmētiskā piešķiršana nodrošina izteiksmes izpildīšanu labajā pusē:

```
?- A is 1+2.

A = 3

yes
```

Savietojošā salīdzināšana neveic rēķināšanu, tomēr visa labā puse netiek uzskatīta par vienu atomu:

```
?- A=1+2, B=1 + 2, A=B.

A = 1+2
B = 1+2

yes
```

Labajā pusē ir 3 pēc kārtas novietoti termi – divi veseli skaitļi un viens atoms:

```
?- atom(1+2).

no
?- integer(1),integer(2),atom(+).

yes
```

Tas izskaidrojams ar to, ka atoms var sastāvēt gan no burtiem, gan cipariem, gan speciālajiem simboliem, bet vienlaicīgi tikai (ja vērtība nav ievietota vienkāršajās pēdiņās):

- no burtiem vai cipariem, kas sākas ar mazo burtu,
- no speciālajiem simboliem:

```
?- atom(++),atom(a123),atom(+-*/).

yes
```

Tādējādi pāreja no cipariem un burtiem uz speciālajiem simboliem un atpakaļ, t.sk. arī tukšumi, nozīmē iepriekšējā terma (skaitļa, atoma) beigas un potenciāli jauna terma sākumu. Šo īpašību var izmantot datu formatēšanā un vizualizācijā:

```
?- A=B*C, parent(B,C).

A = a*b
B = a
C = b ? a

A = a*c
B = a
C = c

A = c*d
B = c
C = d

A = c*e
B = c
C = e

yes
?- findall(A/B,conc(A,B,[a,b,c]),L).

L = [[]/[a,b,c],[a]/[b,c],[a,b]/[c],[a,b,c]/[]]

yes
```

Ar atdalītājsimboliem atdalītus termus var apstrādāt līdzīgi kā sarakstus, tikai, kā redzams, no labās puses:

```
?- X=a/b/c/d,X=A/B.

A = a/b/c
B = d
X = a/b/c/d

yes
```

15.6. Vingrinājumi

Vingrinājums #1.

Dota programma (*clauses3.pl*):

```
parent(a,b).
```

```
parent(a,c).
parent(c,d).
parent(c,e).
```

Definēt predikātu *findallparents/1*, kas atgriež sarakstu ar visiem iesaistītajiem vecākiem šajā programmā:

```
?- findallparents(M).

M = [a,c]

yes
```

Ieteikums. Predikāta definēšanā izmantot *setof/3* un *findall/3*, kaut arī, atsevišķi ņemti, šie predikāti neder, jo atkārto vecākus tik reižu, cik noteikumos tie ir sastapti:

```
?- findall(A,parent(A,_),L).

L = [a,a,c,c]

yes
```

Vingrinājums #1.

Definēt predikātu *listtoseq/2*, kas pārveido sarakstu par ar atdalītājsimbolu "/" atdalītu elementu virkni un otrādi:

```
?- listtoseq(X,a/b/c/d).

X = [a,b,c,d]

yes
?- listtoseq([a,b,c,d],a/b/c/d).

yes
?- listtoseq([a,b,c,d],X).

X = a/b/c/d

yes
```