9. Atpakaļkāpšanās mehānisms, tā ietekmēšana un citi kontroles mehānismi

Nodaļas saturs

Nodaļā ievietotie attēli, programmu piemēri un tabulas

9.1. Atpakaļkāpšanās mehānisma darbības principi

Nodaļā par vaicājumu izpildi (7.) bija iespēja izsekot, kā viens vaicājums var atgriezt vairākus rezultātus un kā notiek visu iespējamo variantu koka pārstaigāšana. Atgādinājumam piemērs par brāļu un māsu noskaidrošanu Ritai programmas *family2* ietvaros.

```
sibling(rita,A).

A = valdis ? a

A = pauls

no
```

Izvedums.

```
S1. sibling(rita,A?).
  A18. X=rita, Y?=A?, parent(Z?,rita), parent(Z?,Y?), rita≠Y
    S2. parent(Z?,rita), parent(Z?,Y?), rita≠Y
      A7. guntis=Z, parent(guntis,Y?), rita≠Y
        S3. parent(guntis,Y?), rita≠Y
          A7. rita=Y, rita≠rita, fail
          A8. valdis=Y, rita≠valdis, true
          valdis=s2.Y, s2.Y=s1.A
          A=valdis
          A9. pauls=Y, rita≠pauls, true
          pauls=s2.Y, s2.Y=s1.A
          A=pauls
      no
```

Ja konkrētā solī ir pieņemts negatīvs lēmums (rita≠rita ⇒ *fail*), tas nenozīmē, ka izpilde kopumā beidzas neveiksmīgi, bet gan, ka izpildē jāpaiet vienu hierarhisko līmeni uz augšu (uz nākošo **izvēles punktu** – *choice point*) un tur jāmeklē nākošais variants.

Šo principu nākošā varianta meklēšanā sauc par **atpakaļkāpšanās mehānismu** (*backtracking*). To var salīdzināt ar pavediena izmantošanu nezināma labirinta pārstaigāšanā – sastopot strupceļu, jādodas pa "diegu" atpakaļ, līdz būs iespējama jauna izvēle ieiet nepārstaigātā atzarojumā.

Atšķirība ar labirinta izstaigāšanu Prolog programmas izpildei ir tāda, ka Prolog programmas izpildes koks var dažreiz teorētiski būt ļoti liels vai pat bezgalīgs, tāpēc būtiski ir ievērot meklēšanas stratēģiju – uz kuru pusi pa priekšu meklēt.

Prolog programmā tas izpaužas šādi:

- ja vienu predikātu definē vairāki teikumi, ir svarīga šo teikumu secība,
- ir svarīga predikātu izsaukumu secība programmas noteikumu ķermenī.

Vistuvākais piemērs tam ir šāds – ja predikāts ir definēts ar kādu rekursīvu noteikumu, tad:

- šis noteikums nav vienīgais,
- pirmais no predikātu definējošiem noteikumiem nav rekursīvs.

Ja tas tā nav, tad programmas izpilde var iecikloties un beigties ar steka pārpildīšanos.

Tas pats attiecas arī uz rekursīva noteikuma ķermeni – rekursīvais izsaukums tajā nekad nebūs pirmais.

9.2. Pārlasāmo variantu skaita ierobežošana ar (!)/0

9.2.1. Darbības princips

Dažreiz programmētājam, pārzinot problēmas specifiku, ir papildus zināšanas par iespējamo variantu koka pārstaigāšanu, un tās var nodot Prolog sistēmai, lai padarītu efektīvāku tās darbu, izvairoties no lieku variantu pārskatīšanas.

Lūk, piemērs, uz programmas *family2* bāzes, kas noskaidro, vai dotā persona ir kādam māte:

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

Šīs programmas izpildes divi varianti:

```
?- mother(A).

A = minna ? ;

A = minna ? ;

A = liesma ? ;

A = liesma ? ;

no
?- mother(minna).

true ? ;
```

```
    yes
```

Pirmajā variantā sistēma katru no mātēm piemin tik reizes, cik tai ir bērnu. Otrajā variantā divas reizes tiek dota apstiprinoša atbilde – jo Minnai ir 2 bērni.

Lai saprastu, kas šeit būtu uzlabojams, otrajam variantam izveidosim izpildes izvedumu (*mother/1* definīciju numurēsim ar A21):

Izvedums (*mother(minna)***).**

```
S1. mother(minna).
  A21. X=minna, female(minna), parent(minna,_)
    S2. female(minna), parent(minna,_)
      A10. female(minna) true
        A1. parent(minna,_) true
        true
        A2. parent(minna,_) true
    yes
```

Pēc noteikuma A1 apskatīšanas uz ekrāna tiek izdrukāts apstiprinošs paziņojums *true*, tomēr pēc tā tiek meklēts nākošais derīgais variants.

Savukārt mums pilnīgi pietiktu, ka pēc tam, kad noskaidrots, ka dota persona ir sieviete (*female/1*) mēs meklētu tikai vienu rezultātu.

Lai atrisinātu šo problēmu, mums jāievieš jauns predikāts – *parent/1*, kas par doto personu pasaka, vai tā kādam vispār ir vecāks.

Šī predikāta definīcija izskatītos apmēram šāda:

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

Tomēr ir vajadzīgs papildus mehānisms, kas nodrošinātu, ka, izpildoties pazīmei par vecāka esamību (*parent/1*) uz doto personu vienreiz, meklēt nākošo iespējamo variantu tiek aizligts. Tāda iespēja ir, un to nodrošina izsaukuma zīmes (!) jeb atšķelšanas (*cut*) predikāts ((!)/0).

Šī predikāta definīcija izskatītos apmēram šāda:

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

Nogriešanas operators jeb nogriešanas predikāts, salīdzinājumā ar labirinta izstaigāšanu, atgādina mezglu pavedienā, tālāk par kuru viena teikuma ietvaros atpakaļ atkāpties nedrīkst. Tad, kad šis teikums būs veiksmīgi izpildīts, tad, atpakaļ gan drīkstēs kāpties to izsaukušais teikums. Tādējādi ar (!) predikātu tiek iezīmēta zona, kurā ir aizliegti izvēles punkti atpakaļkāpšanās procesā. (!) kā predikāts vienmēr atgriež *true*.

Tad, kad mums ir nodefinēts predikāts *parent/1*, izmantojot (!) operatoru, tad varam definēt arī mūs interesējošo predikātu *mother/1*, kas neatkārtos rezultātus:

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

Un šī predikāta izpilde:

```
    ?- mother(A).

    A = minna ? a

    A = liesma

    no
```

```
?- mother(minna).

yes
```

Izvedums (*mother(minna)*), jaunajā variantā.

```
S1. mother(minna).
  A22. X=minna, female(minna), parent(minna)
    S2. female(minna), parent(minna)
      A10. female(minna) true
        A21. X=minna, parent(minna,_), !
          S3. parent(minna,_), !
            A1. X=minna true
              S3. ! (aizliedz lēkt atpakaļ zemāk, kā uz līmeni S2,
                        bet neviena cita female(minna) nav)
                true
    yes
```

Att. 9-1. Programmas (*family4.pl*) fragments.

```
A1.  parent(minna,liesma).
A10. female(minna).
A21. parent(X):-parent(X,_),!.
A22. mother(X):-female(X),parent(X).
```

9.2.2. Eksperimenti ar predikātu (!)

9.2.2.1. Zīmes funkcija signum

Funkcija *signum(·)* darbojas šādi:

$$signum(x) = \left\{ \begin{array}{l} 1; x > 0 \\ 0; x = 0 \\ -1; x < 0 \end{array} \right\}$$

Sistēmā GNU-Prolog tai atbilst iebūvētā funkcija *sign*. Mēģināsim izveidot šim predikātam savu definīciju:

```
signum(X,-1):-X<0.
signum(X,0):-X=0.
signum(X,1):-X>0.
```

Pārbaude:

```
?- signum(-5,Y).

Y = -1 ? ;

no
?- signum(0,Y).

Y = 0 ? ;

no
?- signum(5,Y).
```

```
Y = 1

yes
```

Izpilde parāda, ka negatīva skaitļa vai nulles gadījumā, dodot atbildi, sistēma vēl nedod apstiprinošo *yes*, bet prasa lietotājam turpināt, jo acīmredzami ir sagatavojusies pārskatīt arī pārējos noteikumus, kas definē predikātu *signum/2*.

Programmētājam ir iespēja pateikt programmai, ka, sastopot pozitīvu rezultātu, sistēma drīkst tālāk nemeklēt, kas tiek demonstrēts predikāta *signum2/2* definīcijā, pielietojot atšķelšanas predikātu

```
signum2(X,-1):-X<0,!.
signum2(X,0):-X=0,!.
signum2(X,1):-X>0.
```

Pārbaude:

```
?- signum2(-5,Y).

Y = -1

yes
?- signum2(0,Y).

Y = 0

yes
?- signum2(5,Y).

Y = 1

yes
```

9.2.2.2. Predikāts elementa piederībai sarakstam contains/2

Predikāts *contains/2* pasaka, vai elements ietilpst sarakstā:

```
contains([A|_],A).
contains([_|C],A):-contains(C,A).
```

Pārbaude:

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

true ? ;

no
?- contains([a,b,c,b],b).

true ? a

true
```

```
no
?- contains([a,b,c],X).

X = a ? a

X = b

X = c

no
```

Kā redzams, predikātam ir šādas problēmas – dodot apstiprinošu atbildi uz tiešu pieprasījumu, sistēma uzreiz nezin, ka jābeidz, un gatavojas meklēt nākošo variantu, turklāt, ja sarakstā ir vienādi elementi, tad meklējot šādu elementu, atbilde tiks izdota tik reižu, cik elements sastopams sarakstā.

Arī šeit var palīdzēt atšķelšanas operators:

```
contains2([A|_],A):-!.
contains2([_|C],A):-contains2(C,A).
```

Pārbaude:

```
?- contains2([a,b,c],c).

yes
| ?- contains2([a,b,c,b],b).

yes
```

Tiesa gan, jāsamierinās ar šādu variantu:

```
?- contains2([a,b,c],X).

X = a

yes
```

9.3. Operatori semikols(;) un bultiņa (->)

Operators komats(,), kuru lieto teikuma elementu atdalīšanai noteikumu un vaicājumu ķermenī, atbilst konjunkcijai (&, un) matemātiskajā loģikā. Tomēr Prologā pieejami arī disjunkcijas ($\vee$, vai) un implikācijas ($\rightarrow$, seko) analogi – attiecīgi semikols(:) un bultiņa (->, defise un lielāks).

Ar šo operatoru palīdzību programmu var pierakstīt ar mazāku noteikumu skaitu (katrs jauns noteikums ir kā disjunkts (ar $\vee$ atdalīts elements).

Piemērs ir predikāts maksimālā elementa noskaidrošanai *max/3*:

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

Pārbaude:

```
?- max(1,2,X).

X = 2

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

X = 3

yes
```

Izmantojot operatorus (;) un (->), to var pierakstīt vienā noteikumā, predikāts *max2/3*:

```
max2(X,Y,Z):-X>Y->Z=X;Z=Y.
```

Pārbaude:

```
?- max2(1,2,X).

X = 2

yes
?- max2(3,2,X).

X = 3

yes
```

Bultiņas operatoru var aizvietot ar attiecīgu atšķelšanas predikātu (apgabali starp semikoliem vienkārši aizstāj atsevišķu noteikumu ķermeņus):

```
max3(X,Y,Z):-X>Y,!,Z=X;Z=Y.
```

9.4. Predikāti true/0 un fail/0

Predikāts *true/0* vienmēr veiksmīgi izpildās. Tā izmantošanas jēga parasti ir tikai komplektā ar bultiņas operatoru (->) vai semikola operatoru (;).

Predikāts *fail/0* vienmēr izpildē cieš neveiksmi un tādējādi iedarbina atpakaļkāpšanās mehānismu (*backtracking*).

Jāsaprot, ka *fail/0* nozīmē lokālu neveiksmi, nevis loģisko NOT, un pēc lokālas neveiksmes Prolog sistēma mēģinās atrast nākošo variantu, tādēļ fail/0 tiek bieži lietots komplektā ar (!), lai modelētu loģisko nē.

Labs piemērs ir predikāts "nav vienāds" *notequal/2*:

```
?- notequal(a,b).

yes
?- notequal(x,x).

no
```

Lūk, divi šī predikāta realizācijas varianti:

```
notequal(A,A):-!, fail.
notequal(_,_).
notequal2(A,B):-A=B, !, fail; true.
```

Att. 9-2. Nodaļā apskatīto predikātu kopsavilkums (*backtracking.pl*).

```
signum(X,-1):-X<0.
signum(X,0):-X=0.
signum(X,1):-X>0.
signum2(X,-1):-X<0,!.
signum2(X,0):-X=0,!.
signum2(X,1):-X>0.
contains([A|_],A).
contains([_|C],A):-contains(C,A).
contains2([A|_],A):-!.
contains2([_|C],A):-contains2(C,A).
max(X,Y,X):-X>Y,!.
max(_,Y,Y).
max2(X,Y,Z):-X>Y->Z=X;Z=Y.
max3(X,Y,Z):-X>Y,!,Z=X;Z=Y.
notequal(A,A):-!, fail.
notequal(_,_).
notequal2(A,B):-A=B, !, fail; true.
```

9.5. Vingrinājumi

Vingrinājums #1.

Iepriekš tika apskatīts predikāts *removeelement/3*:

```
removeelement([A|X],A,X).
removeelement([B|X],A,[B|Y]):-removeelement(X,A,Y).
```

Konkrēta izmetamā elementa gadījumā tas strādā šādi:

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

X = [a,c] ? ;

no
```

Izveidot predikātu *removeelement2/3*, kas, sniedzot atbildi, tajā brīdī arī apstājas, nevis sagatavojas nākamās atbildes meklēšanai:

```
?- removeelement2([a,b,c],b,X).

X = [a,c]

yes
```

Vingrinājums #2.

Izmantojot operatorus (->) un (;), definēt predikātu *signum/2* (piemēru sk. augstāk), izmantojot vienu vienīgu noteikumu.

Vingrinājums #3.

Dots predikāts *evenlength/1*, kas pasaka, vai saraksts ir ar garumu pāra skaitlis:

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

Definēt predikātu *oddlength/1*, kas pasaka, vai saraksts ir ar garumu nepāra skaitlis, kā negāciju no predikāta *evenlength/1* (pēc līdzības ar augstāk apskatīto predikātu *notequal/2*).