# 14. Termu apstrādes procedūras

## *Nodaļas saturs*



## *Nodaļā ievietotie attēli, programmu piemēri un tabulas*



## *14.1. Atomu izveidošana un dekompozīcija*

Dažreiz ir nepieciešams simbolu virkni, kas nolasīta no faila vai konsoles kā simbolu kodu virkne, pārveidot par atomu.

Šim nolūkam der iebūvētais predikāts *name/2*:

```
?- name(hallo,L).

L = [104,97,108,108,111]

yes
?- name(N,[65,66,67]).

N = 'ABC'

yes
```

Izmantojot šo predikātu, kā arī jau iepriekš definēto predikātu *rev/2* sarakstu apgriešanai, mēs varam iegūt predikātu atoma apgriešanai *revatom/2* (kopā ar saistītajiem predikātiem *rev/2* un *conc/3*):

```
conc([],Z,Z).
conc([X|Y],W,[X|Z]):-conc(Y,W,Z).
rev([],[]).
rev([A|X],Z):-rev(X,Y), conc(Y,[A],Z).
revatom(A,B):-name(A,X), rev(X,Y), name(B,Y).
```

Predikāts darbībā:

```
?- revatom(sula,A).
```

```
A = alus

yes
```

## 14.2. Termu tipa noskaidrošana

Liela daļa predikātu strādā "dažādos virzienos", t.i., nav noteikts, kuras ir ievades un kuras izvades pozīcijas. Tomēr dažreiz ir nepieciešama papildus pārbaude uz terma tipu, lai korekti uzbūvētu predikātu, piemēram, vai terms ir skaitlis, atoms vai nekonkretizēts mainīgais.

**Tab. 14-1. Prolog predikāti terma tipa pārbaudīšanai.**

| Predikāts | Komentārs |
|---|---|
| `atom/1` | Terms aizpildīts ar atomu (vai nu tas ir atoms, vai arī mainīgais, kas konkretizēts ar atomu). |
| `integer/1` | Terms aizpildīts ar veselu skaitli. |
| `float/1` | Terms aizpildīts ar skaitli ar peldošo komatu. |
| `number/1` | Terms aizpildīts ar skaitli. |
| `atomic/1` | Terms aizpildīts ar atomu vai skaitli. |
| `var/1` | Terms ir nekonkretizēts mainīgais. |
| `nonvar/1` | Terms nav mainīgais vai arī ir konkretizēts. |
| `compound/1` | Terms aizpildīts ar sarakstu vai struktūru ar apjomu, lielāku par 0. |
| `callable/1` | Terms aizpildīts ar atomu, sarakstu vai struktūru ar apjomu, lielāku par 0. |

Piemērs nepieciešamībai pēc šādas pārbaudes ir predikāts, kas noskaidro, vai dotais elements ietilpst sarakstā. Nākošajā piemērā parādīti divi predikāta realizācijas varianti:

```
contains1([A|_],A).
contains1([_|C],A):-contains1(C,A).
contains2([A|_],A):-!.
contains2([_|C],A):-contains2(C,A).
```

Predikāts ir izmantojams 2 veidos:

- padodot 2. pozīcijā vērtību,
- padodot 2. pozīcijā nekonkretizētu mainīgo.

Predikāts *contains1/2* ne gluži perfekti strādā, kad otrajā pozīcijā ir konkrēta vērtība (lieki prasa lietotājam, vai turpināt, un atgriež vairākus rezultātus, ja sarakstā ir vairāki šādi elementi):

```
?- contains1([a,b,c,b],b).

true ? ;

true ? ;

no
```

Savukārt otrs variants, kam šī problēma ir novērta, nemāk izdot rezultātā visus saraksta elementus, ja otrajā pozīcijā padots mainīgais (atgriež tikai pirmo):

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

A = a

yes
```

Izmantojot tipa noteikšanas predikātu, var uzbūvēt apkopojošo predikātu, kas pēc vajadzības izsauc vienu vai otru:

```
contains(X,A):-var(A), !, contains1(X,A).
contains(X,A):-contains2(X,A).
```

Šīs predikāts abos gadījumos strādā korekti:

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

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

A = a ? a

A = b

A = c

A = b

no
```

## 14.3. Termu apstrāde

Ierakstīšana programmā vai vaicājumā nav vienīgais veids, kā dabūt termu Prolog sistēmā. Arī programmas darbības laikā termus var būvēt no sastāvdaļām vai arī dalīt elementos.

**Tab. 14-2. Prolog predikāti termu apstrādei.**

| Predikāts | Komentārs |
|---|---|
| `(=..)/2` | Saliktā terma pārveidošana par elementu sarakstu un atpakaļ |
| `functor/3` | |
| `arg/3` | |

### 14.3.1. Terma izveidošana un dekompozīcija ar (=..)

*T* =.. *L* ir patiess, ja L ir saraksts, kura pirmais elements ir terma T funktors, bet pārējie – šī terma argumenti:

```
?- parent(guntis,rita) =.. L.

L = [parent,guntis,rita]

yes
?- T =.. [female,rita].
```

```
T = female(rita)

yes
```

Nākošajā piemērā terms (predikāta izsaukums) ne tikai tiek uzkonstruēts, bet arī izsaukts un izpildīts:

```
?- L=[conc,[a,b],[c,d],X], T =.. L, T.

L = [conc,[a,b],[c,d],[a,b,c,d]]
T = conc([a,b],[c,d],[a,b,c,d])
X = [a,b,c,d]

yes
```

### 14.3.2. Terma funktora un argumentu noskaidrošana ar functor/3 un arg/3

*functor(T,F,N)* ir patiess, ja F ir terma T galvenais funktors, bet N – argumentu skaits:

```
?- functor(parent(guntis,rita),F,N).

F = parent
N = 2

yes
```

*arg(N,T,A)* ir patiess, ja A ir N-tais arguments termā T:

```
?- arg(2,parent(guntis,rita),A).

A = rita

yes
```

*functor/3* un *arg/3* kopā var veikt arī pilnu terma izveidošanu un dekompozīciju:

```
?- functor(T,conc,3), arg(1,T,[a,b]),
   arg(2,T,[c,d]), arg(3,T,X), T.

T = conc([a,b],[c,d],[a,b,c,d])
X = [a,b,c,d]

yes
```

## *14.4. Vienādību veidi*

Ja iepriekš tika apskatīti vairāki vienādību un nevienādību veidi. Šajā nodaļā dots visu vienādību veidu kopsavilkums.

**Tab. 14-3. Vienādību veidu kopsavilkums.**

| Vienādība | Nevienādība | Komentārs |
|---|---|---|
| `=` | `\=` | Vienādība uz savietošanos |
| `is` | | Aritmētiskā piešķiršana |
| `=:=` | `=\=` | Aritmētiskā vienādība |
| `==` | `\==` | Pārbaude uz identiskumu (“burtiskā” vienādība) |

Tālāk sekojošie piemēri ilustrēs šo vienādību/nevienādību atšķirības.

### 14.4.1. Skaitliskās vienādības atšķirības no savietojamības

Skaitliskā vienādība/nevienādība (=:=) nodrošina abu pušu izteiksmju rezultātus, turpretī vienādība uz savietojamību (=) neko nerēķina un savieto/salīdzina nerēķinot:

```
?- 1+2 = 2+1.

no
?- 1+2 =:= 2+1.

yes
```

Tai pat laikā skaitliskā salīdzināšana “pieprasa”, lai abās pusēs būtu izteiksmes, kuras var izpildīt (mainīgie neder), citādi iestājas konkretizācijas kļūda (*instatntiation error*):

```
?- A = 2+1.

A = 2+1

yes
?- A =:= 2+1.
uncaught exception: error(instantiation_error,(=:=)/2)
```

### 14.4.2. Skaitliskās vienādības atšķirības no skaitliskās piešķiršanas

Skaitliskā piešķiršana pieprasa izrēķināmu skaitlisku izteiksmi tikai labajā pusē, vēl vairāk, kreisajā pusē tāda nedrīkst būt, tur jābūt gatavai vērtībai:

```
?- A =:= 2+1.
uncaught exception: error(instantiation_error,(=:=)/2)
?- A is 2+1.

A = 3

yes
?- 2+1 is 2+1.

no
?- 3 is 2+1.

yes
```

### 14.4.3. Identiskuma atšķirība no savietojamības

Identiskums pieprasa ne tikai vienādu vērtību, bet arī vienādu struktūru, pat mainīgo nosaukumus (ja tie nav konkretizēti):

```
?- A = a.

A = a

yes
?- A == a.

no
?- A == A.

yes
?- [a,b,c] == [a,b,c].

yes
?- X=a, Y=a, X == Y.

X = a
Y = a

yes
?- X == Y.

no
```

### 14.4.4. Skaitliskās vienādības atšķirība no identiskuma

Skaitliskā vienādība (arī visas nevienādības) nodrošina skaitliskās izteiksmes izrēķināšanu, bet pārbaude uz identiskumu to nedara.

```
?- 1+2 == 1+2.

yes
?- 1+2 == 3.

no
?- 1+2 =:= 3.

yes
```

No skaitliskās salīdzināšanas pozīcijām salīdzināšana un identiskumu un savietojamību atšķiras "uz vienu pusi".

## *14.5. Vingrinājumi*

**Vingrinājums #1.**

Definēt predikātu *palindrome/1*, kas atpazīst padoto atomu kā simetrisku vārdu jeb palindromu (t.i., kas lasās vienādi no abām pusēm):

```
?- palindrome(aka).

yes
?- palindrome(akas).

no
```

**Vingrinājums #2.**

Izveidot predikātu *summa/3*, kas rēķina pirmo divu pozīciju summu. Ja predikātam nav padoti vismaz 2 skaitļi, tā izpilde beidzas neveiksmīgi. Ja iztrūkst kāds no pirmajiem diviem elementiem, attiecīgi tiek veikta atņemšanas operācija:

```
?- sum(1,2,A).

A = 3

yes
?- sum(2,A,5).

A = 3

yes
?- sum(A,4,7).

A = 3

yes
?- sum(1,3,4).

yes
?- sum(A,B,2).

no
?- sum(a,b,c).

no
```