12. Operatoru notācija

Nodaļas saturs

Nodaļā ievietotie attēli, programmu piemēri un tabulas

12.1. Operatoru pieraksts pret funkcionālo pierakstu

Matemātikā un arī programmēšanas valodās (t.sk. Prolog) ir pierasts aritmētiskas izteiksmes pierakstīt, lietojot infikso pieraksta formu – operatora zīme ir pa vidu ("in") argumentiem:

```
?- A is 1 * 2 + 3 * 4.

A = 14

yes
```

Tas ir pretēji funkcionālajai pieraksta formai, kur funkcijas (piemēram, predikāta) nosaukums ir pirms (nevis pa vidu) argumentiem, turklāt argumenti ievietoti iekavās. Izrādās, ka arī aritmētiskas operācijas valodā Prolog ir pierakstāmas funkcionālā formā:

```
?- A is +(*(1,2), *(3,4)).

A = 14

yes
```

Aritmētisku izteiksmju pierakstam nešaubīgi ērtāks un uzskatāmāks pieraksta veids ir operatoru pieraksts. Tāpēc jautājums nav par to, ka mēs varam aritmētiskas izteiksmes pierakstīt savādāk, bet gan par to, kā izsaukt predikātus vaicājumos, izmantojot infiksu formu, kas vairāk līdzinās dabiskajām valodām un tādēļ būtu ērtāka. Piemēram, predikāta izsaukums, kas noskaidro, vai elements ietilpst sarakstā (*contains/2*, sk. 10. nodaļu):

```
?- [a,b,c,d] contains c.

true ? ;

no
```

Tradicionālais variants:

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

```
true ? ;

no
```

Izrādās, ka valodā Prolog tas ir iespējams, un operatoru pieraksts var attiekties gan uz predikātiem (kā šajā gadījumā), gan uz funktoriem (sk. piemērus zemāk), kas krietni paplašina operatoru pieraksta izmantošanas iespējas (operatoriem ir ne vairāk kā 2 argumenti, bet predikātiem tādu var būt vairāk).

Lai ļautu pievienot operatoru pierakstam jaunus elementus (predikātus, funktorus), vispirms jāsaprot valodā Prolog eksistējošā iekšējā iebūvēto operatoru izmantošanas sistēma, ietverot tādus jēdzienus, kā prioritāte un izpildes secība jeb asociativitātes virziens.

12.2. Operatoru pieraksta īpašības

Katru operatoru raksturo divas īpašības – prioritāte un asociativitāte.

Prioritāte nosaka, kuru no blakusesošajiem operatoriem izpildīt pirmo. Valodā Prolog prioritāte parasti (arī GNU-Prolog) tiek noteikta kā skaitlis robežās **0..1200** (mazāks skaitlis nozīmē augstāku prioritāti), piemēram, izteiksmē *1+2*3* vispirms izpildās reizināšana.

Asociativitāte nosaka, kā rīkoties, ja blakus atrodas vienādas prioritātes (parasti – vieni un tie paši) operatori. Asociativitāte infiksajiem operatoriem var būt 3 veidu:

- **kreisā**, piemēram, atņemšanai: *2-3-2* nozīmē *(2-3)-2*,
- **labā**, piemēram, kāpināšanai (**): *2**3**2* nozīmē *2**(3**2)*.
- **nav**, piemēram, augstāk pieminētais "contains": pieraksts *[a,b,c] contains X contains Y* – nav atļauts, jo tam nav semantiskas jēgas.

Valodā Prolog operatora asociativitāti apzīmē 2 vai 3 burtu kombinācija:

Tab. 12-1. Prolog operatoru veidi pēc to asociativitātes.

Vispārējais tips	Veids	Komentārs
infiksais	`yfx`	kreisā asociativitāte, piemēram, -, /
	`xfy`	labā asociativitāte, piemēram, kāpināšana
	`xfx`	nav asociativitātes, piemēram, salīdzināšana =
prefiksais	`fy`	atļauta asociativitāte, piemēram, unārais -
	`fx`	nav asociativitātes, piemēram, ++x valodā C++
postfiksais	`yf`	atļauta asociativitāte
	`xf`	nav asociativitātes, piemēram, x++ valodā C++

Asociativitātes veida apzīmējumu var nojaust pēc atbilstošā komentāra – burts 'f' nozīmē pašu operatoru, bet burti 'x' un 'y' – argumentus, bet tieši burts 'y' – atļauju asociativitātei no attiecīgās puses.

Stingri ņemot 'x' nozīmē, ka attiecīgajam argumentam jābūt ar stingri augtāku prioritāti nekā operatoram, bet 'y' – kā tā drīkst būt vienāda vai augstāka.

Ja arguments ir iekļauts iekavās vai tas ir vienkāršs objekts (t.i., bez struktūras), tad tā prioritāte tiek noteikta kā 0, citādi, argumenta prioritāte vienāda ar tā galvenā operatora prioritāti.

Tab. 12-2. GNU-Prolog iebūvēto operatoru pārskats ar prioritātēm un asociativitātes veidiem.

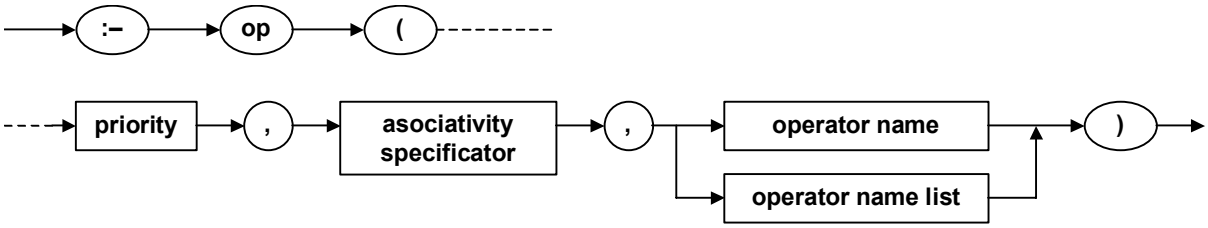
Prioritāte	Veids	Komentārs
1200	xfx	`:-  -->`
1200	fx	`:-`
1100	xfy	`;`
1050	xfy	`->`
1000	xfy	`,`
900	fy	`\+`
700	xfx	`=  \=  =..  ==  \==  @<  @=<  @>  @>=  is  =:=  =\=  <  =<  >  >=`
600	xfy	`:`
500	yfx	`+  -  /\  \/`
400	yfx	`*  /  //  rem  mod  <<  >>`
200	xfy	`**  ^`
200	fy	`+  -  \`

12.3. Unāru un bināru predikātu pārdefinēšana operatoru pierakstā

Tā kā iespējamā pierakstā operatori ir unāri vai bināri (vienvietīgi vai divvietīgi), tad visvieglāk operatoru pierakstu ir piedefinēt šādiem predikātiem.

Lai piedefinētu operatoru pierakstu šādiem, "jau gataviem", predikātiem, vienkārši katram no tiem jāizsauc attiecīga **operatoru notācijas definēšanas direktīva**, kuru realizē predikāts *op/3*.

Att. 12-1. Sintakse: operatoru notācijas definēšanas direktīva (*operator directive*).



Tā kā operatoru direktīva nosaka tikai pieraksta veidu, bet nemaina programmas būtību – nav svarīgi, vai tā atrodas pirms vai pēc attiecīgā predikāta definīcijas.

Tālāk seko programmas fragments, kurā definēti 3 predikāti: *evenlenght/1* – pāra garuma saraksts, *oddlength/1* – nepāra garuma saraksts, *contains/2* – piederības noteikšana sarakstam. Bez tam ir noteikta arī šo predikātu operatoru notācija, attiecīgi –prefiksa, postfiksa un infiksa.

```
:-op(600, xfx, contains).
:-op(500, fx, evenlength).
:-op(500, xf, oddlength).
evenlength([]).
evenlength([_,_|X]):-evenlength(X).
oddlength([_]):-!.
oddlength([_,_|X]):-oddlength(X).
contains([A|_],A).
contains([_|X],A):-contains(X,A).
```

Operatoru direktīvas nodrošina šādu sintaksi predikātu izmantošanā:

```
?- evenlength [a,b,c].

no
?- evenlength [a,b,c,d].

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

yes
?- [a,b,c,d] oddlength.

no
?- [a,b,c,d] contains c.

true ?

yes
?- [a,b,c,d] contains f.

no
```

Konkrētai prioritātes skaitliskajai vērtībai nav nekādās nozīmes, svarīga ir tikai dažādu operatoru prioritāšu relatīvā attiecība.

12.4. Daudzvietīgu predikātu pārdefinēšana operatoru pierakstā

Tā kā tiešā veidā operatoru direktīvu daudzvietīgajiem predikātiem piemērot nevar, tad pirmais darbs ir predikāta pārrakstīšana divu argumentu sintaksē, vairāku iespējamo argumentu apvienošanu veicot ar funktora palīdzību.

Piemērs šeit būs konkatenācijas predikāts *conc/3*:

```
conc([],Z,Z).
conc([X|Y],W,[X|Z]):-conc(Y,W,Z).
```

Un tā pārveidošana par divvietīgu predikātu, izmantojot funktoru *both/2* (ievērojiet, ka jaunais predikāts izmanto veco *conc/3*!):

```
conc(both(X,Y),Z):-conc(X,Y,Z).
```

Pēc tam operatoru direktīvu attiecina gan uz *conc/2*, gan uz *both/2*:

```
:-op(400,xfx,both).
:-op(600,xfx,conc).
```

Tādējādi pilna programma izskatās šādi:

```
:-op(400,xfx,both).
:-op(600,xfx,conc).
conc([],Z,Z).
conc([X|Y],W,[X|Z]):-conc(Y,W,Z).
conc(both(X,Y),Z):-conc(X,Y,Z).
```

Tās darbināšana operatoru notācijā notiek sekojoši:

```
?- [a,b,c] both [d,e,f] conc X.

X = [a,b,c,d,e,f]

yes
```

Funkcionālajā pierakstā tas būtu:

```
?- conc(both([a,b,c],[d,e,f]),X).

X = [a,b,c,d,e,f]

yes
```

Izmantojot funktorus komplektā ar operatoru notāciju mēs varam iegūt vēl vairāk – izveidot n-vietīgu konkatenācijas operāciju, kas darbojas šādi:

```
?- [a] and [b,c] and [d,e,f] and [g,h,i,j] conc X.

X = [a,b,c,d,e,f,g,h,i,j]

yes
```

Funkcionālajā pierakstā tas izskatās šādi:

```
?- conc(and(and(and([a],[b,c]),[d,e,f]),[g,h,i,j]),X).

X = [a,b,c,d,e,f,g,h,i,j]

yes
```

Programmas papildinājums ir nedaudz sarežģītāks, jo predikāta pirmajā pozīcijā jāapstrādā rekursīva struktūra.

```
:-op(400,yfx,and).
:-op(600,xfx,conc).
%...
conc(and(and(X1,X2),Y),Z):-!, conc(and(X1,X2),X), conc(X,Y,Z).
conc(and(X,Y),Z):-conc(X,Y,Z).
```

Pirmais noteikums apstrādā gadījumu, kad pirmais arguments ir vismaz divu līmeņu *and* struktūra, kura vēl ir jāanalizē, bet otrais noteikums – gadījumu, kad *and* struktūra ir vienā līmenī, un tādējādi tās pirmais arguments ir saraksts (nevis cita *and* struktūra).

Att. 12-2. Nodaļā apskatīto predikātu un operatoru direktīvu kopsavilkums (*op.pl*).

```
:-op(400,xfx,both).
:-op(400,yfx,and).
:-op(600,xfx,[conc,contains]).
:-op(500,fx,evenlength).
:-op(500,xf,oddlength).
evenlength([]).
evenlength([_,_|X]):-evenlength(X).
oddlength([_]):-!.
oddlength([_,_|X]):-oddlength(X).
```

```
contains([A|_],A).
contains([_|X],A):-contains(X,A).
conc([],Z,Z).
conc([X|Y],W,[X|Z]):-conc(Y,W,Z).
conc(both(X,Y),Z):-conc(X,Y,Z).
conc(and(and(X1,X2),Y),Z):-!, conc(and(X1,X2),X), conc(X,Y,Z).
conc(and(X,Y),Z):-conc(X,Y,Z).
```

12.5. Vingrinājumi

Vingrinājums #1.

Definēt operatoru pierakstu predikātam *permut/2* (sk. 11. nodaļu):

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

Vingrinājums #2.

Definēt operatoru pierakstu predikātam *removeelement/3* (sk. 11. nodaļu):

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

kas darbotos šādi:

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

X = [a,b] ?

yes
```

Vingrinājums #3.

Vingrinājuma #2 paplašinājums. Definēt operatoru pierakstu predikātam *removeelement/3* tā, lai būtu iespējams ar vienu izsaukumu izņemt no saraksta vairāk nekā vienu elementu:

```
?- [a,b,c,d] remove c remove A removeelement X.

A = a
X = [b,d] ? a

A = b
X = [a,d]

A = d
X = [a,b]

no
```