11. Dažādi predikāti sarakstu apstrādei

Nodaļas saturs

Nodaļā ievietotie attēli, programmu piemēri un tabulas

11.1. Patvaļīga elementa izmešana no saraksta

Predikāts elementa izmešanai *removeelement/3* var strādāt gan uz konkrēta elementa izmešanu, patvaļīga elementa izmešana, vai pat patvaļīga elementa ievietošanu.

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

X = [a,c] ? a

no
?- removeelement([a,b,c],A,X).

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

A = b
X = [a,c]

A = c
X = [a,b]

no
?- removeelement(X,x,[a,b,c]).

X = [x,a,b,c] ? a

X = [a,x,b,c]

X = [a,b,x,c]

X = [a,b,c,x]

no
```

Un tā definīcija joprojām ir ļoti īsa:

```
removeelement([A|X],A,X).
```

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

Predikāta realizācijas būtība ir – vai nu izmest pirmo elementu, vai nu atlikt izmešanu uz nākamajiem.

11.2. Elementu skaitīšana sarakstā

Vienkāršākais uzdevums, kas izmanto aritmētiskas operācijas saraksta apstrādē, ir saraksta garums. Predikāts *listlen/2*:

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

N = 3

yes
```

listlen/2 definīcija:

```
listlen([],0).
listlen([_|Z],N):-listlen(Z,M), N is M+1.
```

Nākošais predikāts ir saraksta n-tā elementa iegūšana (*nthelement/3*):

```
?- nthelement([a,b,c,d],3,X).

X = c

yes
?- nthelement([a,b,c,d],N,b).

N = 2

yes
?- nthelement([a,b,c,d],4,d).

yes
```

Predikāts tiek definēts pēc līdzības ar saraksta garuma noskaidrošanu:

```
nthelement([A|_],1,A):-!.
nthelement([_|Z],N,A):-nthelement(Z,M,A), N is M+1.
```

Predikāta *nthelement/3* pirmā noteikuma beigās tiek lietots (!) operators. Novāciet to, un pārbaudiet, kā strādā predikāts uz iepriekšējiem 3 vaicājumiem.

Pārbaude, vai elements ir pēdējais sarakstā, ir realizējama, neizmantojot skaitīšanu:

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

A = c

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

no
```

Tas ir definējams šādi:

```
lastelement([A],A):-!.
lastelement([_|Z],A):-lastelement(Z,A).
```

11.3. Saraksta elementu sajaukšana

Šīs sadaļas predikāti ir vieni no interesantākajiem sarakstu apstrādes predikātiem, kas vēl lielākā mērā nekā predikāts *conc/3* demonstrē Prolog iespējas ļoti īsā tekstā pierakstīt apjomīgus algoritmus.

Permutācijas ir kombinatorikas jēdziens, kas nozīmē visus iespējamos veidus, kā sajaukt vienas kopas (masīva, saraksta) elementus. n dažādu elementu kopumu iespējams sajaukt n! dažādos veidos (P(n)=n!), piemēram, 3 elementiem tie ir 6 dažādi veidi. GNU-Prolog sistēmā ir iebūvētais predikāts *permutation/2*, tomēr tālāk tiks dots funkcionāli identisks predikāts *permut/2*:

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

X = [a,b,c] ? a

X = [a,c,b]

X = [b,a,c]

X = [b,c,a]

X = [c,a,b]

X = [c,b,a]

no
```

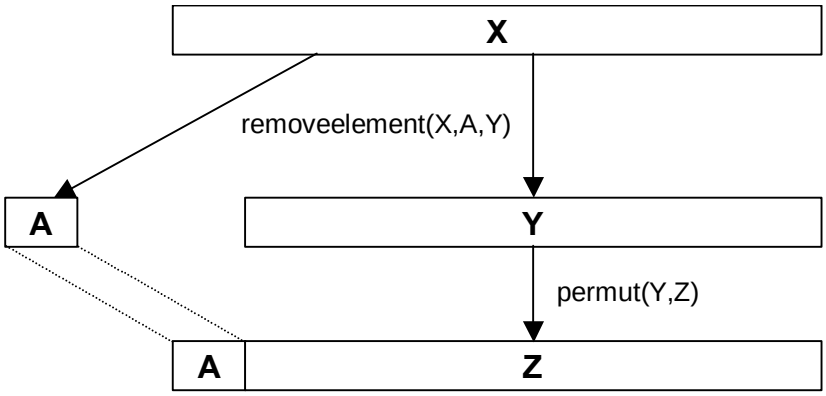
Viens no šī predikāta realizācijas variantiem ir šāds (izmantojot iepriekš definēto predikātu *removeelement/3*):

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

Lai saprastu, kā strādā šis predikāts, jāatceras, ka *removeelement/3* veica patvaļīga elementa izmešanu no saraksta, pārskaitot visus iespējamos izmešanas variantus pie attiecīga vaicājuma formulējuma.

Viens solis procesā (t.i., viena elementi samainīšana) ir izņemt no saraksta patvaļīgu elementu A, bet pēc tam pielikt izņemto elementu jauniegūtajam sarakstam priekšā (tas būtu [A|Y]). Tomēr otrā noteikuma galvā ir [A|Z] nevis [A|Y], jo sarakstā jāveic N šādas samainīšanas, ko nodrošina rekursīvais izsaukums *permut(Y,Z)*.

Att. 11-1. Predikāta *permut/2* noteikuma #2 grafiska reprezentācija.



Kombinācijas ir visas tās dažādās kopas ar lielumu n, kuras var izvēlēties no kopas ar lielumu m (n≤m). Piemēram, ja jāizveido 5 cilvēku komanda (n=5) kādā sporta spēlē, bet ir 10 iespējamie kandidāti (m=10), tad kombinācijas ir visu iespējamo variantu skaits, kā 5 cilvēkus (neņemot vērā to secību) var izvēlēties no 10 iespējamiem. Kombināciju skaits ir $C(m,n)=\frac{m!}{n!(m-n)!}$. C(10,5)=252. Bet, kā redzams nākošajā piemērā – C(4,2)=6.

```
?- combin([a,b,c,d],2,Y).

Y = [c,d] ? a

Y = [b,d]

Y = [b,c]

Y = [a,d]

Y = [a,c]

Y = [a,b]

yes
```

Kombināciju predikāta realizācija jau ir sarežģītāka:

```
combin(_,0,[]):-!.
combin(X,N,X):-listlen(X,N),!.
combin([_|X],N,Y):-combin(X,N,Y).
combin([A|X],N,[A|Y]):-M is N-1, combin(X,M,Y).
```

Pirmais noteikums nozīmē, ka apakškopa garumā nulle ir tukša kopa. Otrais noteikums funkcionāli neietekmē predikāta darbību, bet samazina izpildē tērētos soļus.

Pamatalgoritmu nodrošina trešais un ceturtais noteikums: lai izvēlētos apakškopu, tiek pārstaigāta galvenā kopa pa vienam elementam, un šis viens elements (A) vai nu tiek pielikts apakškopai Y ([A|Y], 4. noteikums), vai nu netiek (3. noteikums).

Dotajā realizācijā apakškopas elementu secība atbilst secībai "lielajā" kopā.

Trešais jēdziens kombinatorikā (apvieno permutācijas un kombinācijas) ir **variācijas** – tās ir visas virknes garumā n, kuras izvēlētas no kopas garumā M elementiem (tātad, salīdzinot ar

kombinācijām, šeit ir svarīga arī secība iegūtajās apakškopās). Variāciju skaits ir $V(m,n)=\frac{m!}{(m-n)!}$. Nākošajā piemērā – V(4,2)=12.

```
?- variat([a,b,c,d],2,A).

A = [c,d] ? a

A = [d,c]

A = [b,d]

A = [d,b]

A = [b,c]

A = [c,b]

A = [a,d]

A = [d,a]

A = [a,c]

A = [c,a]

A = [a,b]

A = [b,a]

no
```

Ja definēti predikāti kombināciju un permutāciju noskaidrošanai, tad variācijas noskaidrot ir ļoti vienkārši:

```
variat(X,N,Z):-combin(X,N,Y), permut(Y,Z).
```

11.4. Apakšvirknes un apakškopas noskaidrošana

Apakšvirkne ir noteikts fragments sākotnējās virknes ietvaros (predikāti *subsequence/2* un *subsequence2/2*):

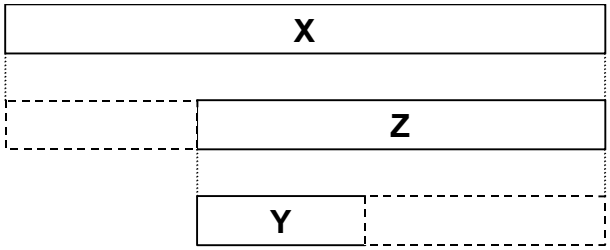
```
?- subsequence2([a,b,c],X).

X = [] ? a

X = [a]

X = [a,b]

X = [a,b,c]

X = [b]
```

```
X = [b,c]

X = [c]

no
```

Ērts veids to realizēt ir, izmantojot predikātu *conc/3*:

```
subsequence(X,Y):-conc(_,Z,X), conc(Y,_,Z).
```

Att. 11-2. Apakšvirknes predikāta *subsequence/2* realizācija ar konkatenāciju.



Problēma konkrētajā realizācijā ir tāda, ka predikāts atgriež vairākus variantus ar tukšo apakšvirkni [], taču to var viegli risināt, speciāli apstrādājot tukšā saraksta gadījumu (izpildes piemērs izmanto tieši šo, otro variantu):

```
subsequence2(_,[]).
subsequence2(X,Y):-conc(_,Z,X), conc(Y,_,Z), Y\=[].
```

Tukšā saraksta atgriešana ir izcelta kā atsevišķs teikums, savukārt galvenais algoritms "nepieņem" tukšus sarakstus.

Apakškopa, salīdzinot ar apakšvirkni, "nepieprasa", lai visi elementi būtu pēc kārtas esoši, tādējādi dotajā piemērā ar [a,bc] visām esošajām apakšvirknēm vēl nāk klāt [a,c]:

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

X = [] ? a

X = [c]

X = [b]

X = [b,c]

X = [a]

X = [a,c]

X = [a,b]

X = [a,b,c]

yes
```

Realizācijas ideja balstās uz predikāta *combin/3* bāzes:

```
subset([],[]).
```

```
subset([_|X],Y):-subset(X,Y).
subset([A|X],[A|Y]):-subset(X,Y).
```

Atšķirība no kombinācijām ir tikai, tāda, ka šeit netiek skaitīts apakškopas garums, jo apakškopa drīkst būt ar jebkādu garumu.

Apakškopas rēķina arī GNU-Prolog iebūvētais predikāts *sublist/2*, tikai ar pretēju argumentu secību (pirmajā pozīcijā ir apakškopa).

Att. 11-3. Nodaļā apskatīto sarakstu apstrādes predikātu kopsavilkums (*list2.pl*).

```
removeelement([A|X],A,X).
removeelement([B|X],A,[B|Y]):-removeelement(X,A,Y).
listlen([],0).
listlen([_|Z],N):-listlen(Z,M), N is M+1.
nthelement([A|_],1,A):-!.
nthelement([_|Z],N,A):-nthelement(Z,M,A), N is M+1.
lastelement([A],A):-!.
lastelement([_|Z],A):-lastelement(Z,A).
permut([],[]).
permut(X,[A|Z]):-removeelement(X,A,Y), permut(Y,Z).
combin(_,0,[]):-!.
combin(X,N,X):-listlen(X,N),!.
combin([_|X],N,Y):-combin(X,N,Y).
combin([A|X],N,[A|Y]):-M is N-1, combin(X,M,Y).
variat(X,N,Z):-combin(X,N,Y), permut(Y,Z).
subsequence(X,Y):-conc(_,Z,X), conc(Y,_,Z).
subsequence2(_,[]).
subsequence2(X,Y):-conc(_,Z,X), conc(Y,_,Z), Y\=[].
subset([],[]).
subset([_|X],Y):-subset(X,Y).
subset([A|X],[A|Y]):-subset(X,Y).
```

11.5. Vingrinājumi

Vingrinājums #1.

Definēt predikātu *removeelement2/3*, kas pēc konkrēta elementa izmešanas uzreiz atbild *yes* (bez liekas pārprasīšanas):

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

X = [a,c]

yes
% Vecais variants strādāja šādi:
?- removeelement([a,b,c],b,X).

X = [a,c] ? a

no
```

Vingrinājums #2.

Definēt predikātu *lastelement2/2*, kas strādā identiski predikātam *lastelement/2*, bet definīcijā izmanto predikātu *conc/3* un definīcijā ir tikai viens noteikums:

```
?- lastelement2([a,b,c],A).

A = c

yes
```

Vingrinājums #3.

Definēt predikātu *variat2/3*, kas strādā identiski predikātam *variat/3*, bet definīcijā neizmanto predikātus *permut/2* un *combin/3*. Ieteikums: idejiski balstīt realizāciju uz *combin/3* bāzes, pievienojot *removelement/3* (pretējā – elementa patvaļīgas iespraušanas – lomā).

Vingrinājums #4.

Definēt predikātu *evenlength/1*, kas pasaka, vai padotā saraksta garums ir pāra skaitlis:

```
?- evenlength([2,e]).

yes
?- evenlength([2,e,f,g,h]).

no
```

Vingrinājums #5.

Definēt predikātu *oddlength/1*, kas pasaka, vai padotā saraksta garums ir nepāra skaitlis:

```
?- oddlength([2,e]).

no
?- oddlength([2,e,f,g,h]).

yes
```

Vingrinājums #6.

Definēt predikātu *shift/1*, kas pārveido sarakstu, pārbīdot elementus pa kreisi (resp., pārvietojot pirmo elementu uz beigām):

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

X = [b,c,a]

yes
```

Vingrinājums #7.

Definēt predikātu *linearize/2*, kas pārveido sarakstu, visus iekļauto sarakstu elementus iznesot galvenajā sarakstā:

```
?- linearize([a,b,[c,d,[e,[f]],[[]]],[],[[[g]]],h],X).
```

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

yes
```

Vingrinājums #8.

Definēt predikātu *rev/2*, kas apgriež sarakstu pretējā secībā (GNU-Prolog ir atbilstošs iebūvētais predikāts *reverse/2*):

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

X = [d,c,b,a]

yes
```