Списъци (Part II)

|  |
| --- |
|  |

**Обръщане на списък**

Разгледахме два начина за това. При първия се обръщаме към триаргументен предикат last, в който, за разлика от предишния last, който писахме, държим и целия списък без последния елемент. Тъй като в **prolog** цялата работа, която трябва да се свърши, е да се дефинират читави правила, то обърнат списък дефинираме като такъв, дето главата му е последния елемент на оригиналния списък, а опашката - обърнатия оригинален списък, без последния си елемент. Да, и това е някак достатъчно - деклариране на очевидности. Остана само декларацията на самия помощен предикат last. И естествено - базовите случаи - половината задача. : )

%my\_reverse(<начален списък>, <обърнат списък>).

my\_reverse([], []).

my\_reverse([Head|Tail], Result) :- my\_last(List, Z, Head), my\_reverse(Tail, Z).

%my\_last(<целия списък>, <всичко преди последния елемент>, <последния елемент>).

my\_last([H], [], H).

my\_last([Head|Tail], [Head|L], Result) :- my\_last(Tail, L, Result).

**Забележка** : глупавите имена на предикати (започващи с "my\_") тук са написани с цел да напомнят, че при разните реализации на **prolog** интерпретатор/компилатор има вградени повечето основни предикати и не щат много-много да се предефинират. Нататък ще пропускам да подчертавам.

Вторият начин, който обсъдихме, е чрез стек. Ама неявен такъв - слагаме един слой потребителски интерфейс (кххм!).

%reverse2(<обърнат списък>, <начален списък>).

reverse2(Result, List) :- rev(Result, [], List).

%rev(<обърнат списък>, <стек>, <начален списък>).

rev(Result, Result, []).

rev(Result, Stack, [Head|Tail]) :- rev(Result, [Head|Stack], Tail).

**Quick sort**

Няма какво да ви обяснявам бързото сортиране. Знаете по-добре от мен идеята : ).

%split (<елементът Х>, <начален списък>, <списък с елементи по-малки от Х>, <списък елементи по-големи от Х>).

split (X, [], [], []).

split(X, [Head|Tail], [Head|L], B) :- Head =< X, split(X, Tail, L, B).

split(X, [Head|Tail], A, [Head|L]) :- Head > X, split(X, Tail, A, L).

%qsort(<начален списък>, <сортиран списък>).

qsort([], []).

qsort([Head|Tail], Result) :- split(Head, Tail, A, B), qsort(A, As), qsort(B, Bs), append(As, [Head|Bs], Result).

**Подмножество**

Отново два начина - хитър и не-хитър. Хитрите начини са онези с предиката not, понеже често е много по-лесно да кажем какво резултатът ни не е (за жените какво НЕ искаме е ежедневие, тъй че ако употребата на not е разрешена на контролното, ще има неочакван обрат при оценките, нях-нях). Базов случай - празното множество е подмножество всекиму.

%subset(<предполагаемо подмножество>, <множество>).

subset([], Y).

subset([H|T], Y) :- member(H, Y), subset(T, Y).

А хитрият начин колко да е хитър - на един ред. Тук ползата му не е така осезаема. Идея - ако има някакъв лош елемент А от Х, който да не е елемент на У, то Х не е подмножество на У.

subset(X, Y) :- not(member(A, X), not(member(A, Y))).

Забележка: Тук ни трябва едно- и двуаргументен предикат not. Не всички реализации предлагат такъв. Ако няма, напишете си. Ето идея:

not(P) :- call(P), !, fail. % тук може и само P вместо call(P)

not(P).

Или на един ред така:

not(P) :- (P -> fail; true).

Тук казваме ако Р е вярно, значи се проваляме; в противен случай — yes.

И едно дребно следствие от тези предикати - случаят две множества да съвпадат вече се пише много лесно. Асистентът искаше да ни прецака да го мислим наново, обаче Зори се досети (сефте!), че ако двете множества са си взаимно подмножества, съвпадат. Тоест:

equal(X, Y) :- subset(X, Y), subset (Y, X).

**Binary tree sort**

В уикипедия е само Tree sort, но се подразбира същото. Наричайте го, както смятате за правилно. Идеята му е да се конструира двоично дърво от елементите, които искаме да сортираме, след което да се обходи подходящо (ЛКД, предполагам), като едновременно с това се конструира списък. By pure magic така конструиран списък в края се оказва сортирания първоначален. Та имаме да свършим няколко неща - да напишем как добавяме елемент към дървото, как конструираме дървото, как го обхождаме и как обгръщаме всичко това, за да се получи въпросното сортиране на един ред. По ред на споменаване:

%add(<елемент>, <дърво>, <резултат>).

add(X, [], [X, [], []]).

add(X, [Top, Left, Right], [Top, Lx, Right]) :- X < Top, add(X, Left, Lx).

add(X, [Top, Left, Right], [Top, Left, Rx]) :- X >= T, add(X, Right, Rx).

Дърво представяме чрез списък от корен, ляво поддърво и дясно поддърво. Писахме top вместо root, защото асистентът използва еднобуквени променливи и щеше да има две големи 'R'. Лошо няма.

%maketree(<списък>, <дърво>).

maketree([], []).

maketree([Head|Hs], Tree) :- maketree(Hs, X), add(Head, X, Tree).

Обхождане на дърво и конструиране на списък:

%ltr(<дърво>, <списък>).

ltr([], []).

ltr([Top, Left, Right], S) :- ltr(Left, S1), ltr(Right, S2), append(S1, [Top|S2], S).

Сортирането, което нямаше да бъде възможно без спонсорството на и благодарение на… всичко написано дотук:

%tsort(<сортиран списък>, <начален списък>).

tsort(Result, List) :- maketree(List, Tree), ltr(Tree, Result).

**Selection sort**

%ssort(<сортиран списък>, <начален списък>).

ssort([], []).

ssort([Min|Tail], List) :- min(Min, List), remove(Min, List, R), ssort(Tail, R).

Не, честно, какво да му коментираме?!