Prolog

Raccolta di prove realizzate in Prolog e commentate da me.

`utilities-lists.pl`

Il file `utilities-lists.pl` contiene tutti i vari predicati che realizzano funzioni basilari con le liste, tra le quali:

- append
- reverse
- contains
- sublist
- palindrome
- count occurrences
- flattening
- sort

N.B. Molti predicati sono già presenti di base in Prolog, ma ne viene mostrata la realizzazione.

Ripasso

Associatività

Prefix	Postfix	Infix	Associativity
fx	xf	xfx	none
	yf	yfx	left
fy		xfy	right

Operatori aritmetici

+	Somma due elementi	fy
-	Sottrae due elementi	fy
*	Moltiplica due elementi	yfx
/	Divide due elementi	yfx
//	Divisione tra interi	yfx
mod	Modulo	yfx
**	Potenza	xfx

Predicati aritmetici

`abs(Exp)`	Valore assoluto

`atan(Exp)`	Arcotangente
`cos(Exp)`	Coseno
`exp(Exp)`	`e^Exp`
`log(Exp)`	Logaritmo naturale di `Expr`
`sin(Exp)`	Seno
`sqrt(Exp)`	Radice quadrata di `Expr`
`tan(Exp)`	Tangente
`sign(Exp)`	Assume valore 1 o -1 in base al segno di `Expr`
`float(Exp)`	Trasforma un numero da intero a float

`truncate(Exp)` Rimuove la parte decimale di un numero `floor(Exp)` Arrotonda per difetto `round(Exp)` Se `>= .5` arrotonda per eccesso, per difetto altrimenti `ceiling(Exp)` Arrotonda per eccesso

Operatori relazionali

`<`	Minore	xfx
`>`	Maggiore	xfx
`=<`	Minore o uguale	xfx
`>=`	Maggiore o uguale	xfy
`\+ Expr`	Ha successo se `Expr` fallisce. Sostituisce `not`	fy
`\==`	Disuguaglianza logica	xfx
`=:=`	Uguaglianza aritmetica (`3 =:= 3.0` è vera)	xfx
`=\=`	Disuguaglianza aritmetica	xfx
`\=`	Non unificazione. Equivalente a `not(X = Y)`	xfx

`is`

```
<number or variable> is expr
```

`is` valuta le espressioni aritmetiche alla sua destra, prima di assegnare il risultato dell'espressione ai termini alla sua sinistra.

Esempio:

```
?- X is 2 + 4.
X = 6
```

Osservazioni generali

- Lista di liste: va scomposto il problema in un sottopredicato che si occupi delle liste singole
- Quando devo effettuare un conteggio e **non ho una variabile inizializzata**, occorre prima effettuare una chiamata ricorsiva al predicato e poi incrementare il conteggio. L'inizializzazione avviene sempre nel caso base!

Esempio:

```
% Count occurrences
count([], _, 0).
count([E|T], E, C) :- count(T, E, C1),
                      C is C1 + 1.
count([_|T], E, C) :- count(T, E, C).
```

- Quando invece ho un valore già inizializzato, le operazioni con quel valore vanno fatte prima della chiamata ricorsiva:

```
% NB inserisco il cut, poiché Prolog esegue i check left most e a sinistra
si ha una variabile/indifferenza.
% Se non venisse inserito il cut, Prolog cercherebbe di fare match anche
con la seconda clausola.
buildList(_, [], 0) :- !.
buildList(X, [X|T], N) :- N1 is N - 1, buildList(X, T, N1).
```

- Se ho da restituire una lista contenenti coppie di liste e devo iterare sulla lista risultato, per avere una lista di coppie posso usare `[[H1,H2] | T3]]` NB non dimenticarsi `[]` che racchiude la coppia!
- Se devo verificare l'uguaglianza tra due elementi in testa a due liste, per poi aggiungere l'elemento alla lista risultato basta fare:

```
prova([H|T1],[H|T2], [H|T]) :- prova(T1,T2,T).
```

- Se una variabile non viene utilizzata, usare il simbolo di indifferenza _
- Negli esercizi con query yes/no, se si vuole ottenere no, ricordarsi del NF (Negation as Failure) + logica positiva di Prolog: se si vuole fallimento, nessuna condizione deve essere vera, quindi non si devono prevedere condizioni alternative.
- Se si vuole prendere (o anche scartare) il primo e il secondo elemento di una lista in Prolog, basta fare:

```
extract([H1,H2|T]
```

- Se in un caso base si ha un predicato che considera un elemento e una lista vuota, e si vuole restituire una lista con quell'elemento, occorre fare:

```
prova(N, [], [N]).
```

In questo esempio si mettono le parentesi quadre, in quanto si vuole distinguere il terzo argomento (una lista) da un elemento (`N`).

Cut

- `!` (cut) utilizzabile per *if mutuamente esclusivi*
- Il cut può essere necessario anche nella condizione base, per motivi di efficienza, o quando si è raggiunta una condizione prevista dal caso base.

Esempio 1:

```
% Caso base, (cut necessario per evitare di finire nel secondo caso, una
volta raggiunta
% la condizione base, ovvero S > E).
% NB la lista non è inizializzata, pertanto inizialmente sarà in ogni caso
vuota!
nuovaLista([], S, E) :- S > E, !.
nuovaLista([S|T], S, E) :- S1 is S + 1, nuovaLista(T, S1, E).
```

Esempio 2 (`member`):

```
% Per motivi di efficienza aggiungo !, dato che appena trovo l'elemento
fermo la mia ricerca.
% Non ha senso che continui la ricerca, se il mio scopo è di trovare che
esista almeno un'occorrenza di X.
member([X|_], X) :- !.
member([_|T], X) :- member(T, X).
```

- Il cut è necessario nel caso base anche qualora il termine più a sinistra **non sia una costante** (`[]`, numero fisso), ma una variabile o una indifferenza (`_`): in questo modo, qualora il caso base sia soddisfatto non si esplora la clausola successiva.

Caso particolare di inizializzazione

Si definisca un predicato `stessaSomma(L,V)` che data una lista di liste non vuota L, controlla che le somme di tutti gli elementi per ciascuna delle (sotto-)liste contenute in L siano uguali a V. Ad esempio per il goal:

```
?-stessaSomma([[6,7,2], [1,5,9], [8,3,4]], V).
Yes V=15
```

perché la somma di tutti gli elementi di ciascuna sotto-lista è pari a 15.

Altri esempi:

```
?-stessaSomma([[6,4,1,2], [1,12], [8,5]], V).
Yes V=13
```

```
?-stessaSomma([[6,7], [1,12], [8,5]], 13).
Yes
```

```
?-stessaSomma([],V).
No
```

Soluzione

```
stessaSomma([H], V) :- !, sum(H,V).
stessaSomma([H|T], V) :- sum(H,V), stessaSomma(T,V).

sum([], 0).
sum([H], H) :- !.
sum([H|T], V) :- sum(T, V1),  V is V1 + H.
```

Osservazioni

- `sum`: il secondo caso è stato introdotto per motivi di efficienza: si risparmia un' iterazione, quindi se si hanno più liste si risparmiano `N` iterazioni (con `N` sottoliste della lista) -`stessaSomma`: caso particolare di inizializzazione. Nell'esercizio si ha:

```
?-stessaSomma([],V).
No
```

Se fosse stato `yes, V=0`, l'esercizio si poteva risolvere in questo modo:

```
stessaSomma([], 0).
stessaSomma([H|T], V) :- sum(H,V), stessaSomma(T,V).

sum([], 0).
sum([H], H) :- !.
sum([H|T], V) :- sum(T, V1),  V is V1 + H.
```

Poiché era richiesta inizializzazione nel caso in cui la lista di liste fosse vuota. In questo caso, quindi, era necessario *non avere il caso base con la lista vuota*, quindi bisognava ragionare in un altro modo, ovvero:

> *Non possiamo avere il caso lista vuota, poiché V in quel caso non può essere inizializzato, quindi il risultato della query deve essere false*

$\Longrightarrow$ La lista non deve mai essere vuota, ma deve contenere almeno un elemento.

```
stessaSomma([H], V) :- !, sum(H,V).
```