Linguaggi di Programmazione, laboratorio

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a ldurante le esercitazioni in laboratorio. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`. **Grazie mille e buono studio!**

Capitolo 2

Laboratorio

2.1 Laboratorio 1

Esercizio 1. *Esempio base:*

```
lavora_per(bill, google). %questo è un fatto
lavora_per(mario, oracle).
lavora_per(steve, apple).  % con ; scorro le opzioni
lavora_per(jane, google).

collega(X, Y) :- lavora_per(X, Z),
                 lavora_per(Y, Z),
                 X \= Y. /* questa è una regola e con \=
                            evita che X = X sia un risultato
```

si possono avere le seguenti richieste con i seguenti output:

```
?- lavora_per(X, google).
X = bill ;
X = jane.

?- lavora_per(bill, X).
X = google.

?- collega(bill, jane).
true.
```

Esercizio 2. *esercizio 2, logica dei Naturali:*

```
/* definisco i naturali: */
nat(0). % 0 è naturale e lo sono anche i successori di un naturale
nat(s(N)) :- nat(N). /* s(N) indica il successore,
                        in compilatore sarà s(s(0)) etc...*/

/* definisco la somma come n + m = (n+1) + (m-1) se m>0 n + 0 = n */

succ(N, s(N)). % defisco il successore
sum(N, 0, N). % il terzo è il risultato
sum(N, s(M), s) :- sum(N, M, T),
                   succ(T, s).

/* si può anche fare così; */
summ(N, 0, N).
summ(N, s(M), s(T)) :- summ(N, M, T).
summ(N, M, S) :-  summ(s(N), P, S),
                  succ(P, M).
```

si hanno i seguenti output:

```
?- nat(0).
true.

?- nat(s(0)).
true.

?- sum(s(s(0)), 0, X).
X = s(s(0)) ;
false.

?- summ(s(0), 0, X).
X = s(0).
```

Esercizio 3. *Calcolo il fattoriale:*

```
fact(0, 1). % fattoriale di 0 è 1, caso base
fact(N, M) :- N>0,
              N1 is N-1,
              fact(N1, M1),
              M is N * M1. /* ogni volta salva in M1 il
                              risultato parziale, N>0
                               porterà a non fare il caso
                                base*/
```

```
?- fact(3, X).
X = 6 ;
false.
```

Esercizio 4. *Primi esercizi sulle liste:*

```
/* trovo primo elemento della lista */
intesta(X, [X|_]). /* | separa testa e coda (che non
                      interessa e indico con _,
                       senza non si sa che la lista può
                        continuare. Interrogo con, per
                         esempio  intesta(2, [2,3]). */

/* trovo ennesimo elemento*/
nth(0, [X|_], X). % se cerco lo 0, caso base
nth(N, [_|T], X) :- N1 is N-1,
                    nth(N1, T, X). /* se non è all'inizio
                                      cerco nella coda ad
                                       ogni passo la head
                                        aumenta di uno
                                        tengo solo T che è
                                         la coda, ovvero
                                         tutto tranne il
                                          primo*/

/*lista contiene l'elemento X?*/
contains(X, [X|_]). % controllo testa
contains(X,[_|T]) :- contains(X, T). % controllo ricorsivcamente la coda
```

Si hanno i seguenti output:

```
?- intesta(2, [2, 3]).
true.

?- intesta(2, [1, 3]).
false.

?- nth(0, [1,2,3], X).
X = 1 ;
false.

?- nth(2, [1,2,3], X).
X = 3 ;
false.

?- nth(18, X, 1).
X = [_594, _600, _606, _612, _618, _624|...] ;

?- contains(5, [5, 6, 7]).
true ;
false.

?- contains(4, [5, 6, 7]).
false.
```

dove l'ultimo risultato di nth ci dice che non può fare nulla con l'istruzione data, cerca all'infinito una risposta senza trovarla. Il primo di contains da false ad una seconda richiesta di risultato in quanto prova ad usare un'altra regola il compilatore di prolog

Esercizio 5. *Ancora sulle liste:*

```
/* funzione append per concatenare */
append([], L, L). % vuota più L = L
append([H|T], L, [H|X]) :- append(T, L, X).
/* l'inizio della lista finale è H e la fine è la fine
    delle liste concatenate */

/* chiedo se una lista  è ordinata */

sorted([]). % lista vuota ordinata
sorted([_]). % lista di un elemento è ordinata
sorted([X, Y| T]) :- X =< Y,
                     sorted([Y| T]). /* confronto sempre l'elemento col
                     resto della tail */

/* chiedo ultimo elemento */

last([X], X). %lista di un elemento ha come ultimo quell'elemento
last([_|T], X) :- last(T, X). /*controllo ricorsivamente
                                 la coda della lista finché
                                  non ho solo T e posso usare
                                   il caso base */
/* posso anche chieder e una lista che finisca con un certo N
                                 per esempio 4  last(X, 4). */

/* tolgo tutte le occorrenze */

remove_all([], _, []). % la lista vuota non ha nulla da rimuovere */
remove_all([X|T], X, L) :- remove_all(T, X, L).
/* se la testa è quel numero rimuovo tutte
    le occorrenze... ma se la testa \= X non va*/
remove_all([H|T], X, [H|L]) :- H \= X,
                               remove_all(T, X, L).
                               /* se H\=X  faccio il
                                   controllo senza H */

/* somma elementi lista */

somma_lista([], 0).
somma_lista([H|T], X) :-  somma_lista(T, N),
```

```
                                X is H + N.
/* sommo tutte le tail ricorsivamente e poi ci sommo la H */

/* ricordiamo le basi delle code */
coda([_|T], T).

/* duplico lista [1,2]->[1,1,2,2] */

duplico([], []).
duplico([H|T], [H, H|X]) :- duplico(T, X).
/* a priori duplico H riscivendo nel risultato
    poi duplico il tail, dove di volta in volta
     ogni head verrà duplicato. Se inverto e metto
      ( x, [lista]) mi toglie i duplicati*/
```

si hanno i seguenti output:

```
?- contains(5, [5, 6, 7]).
true ;
false.

?- contains(4, [5, 6, 7]).
false.

?- append([4], [5, 6, 7], X).
X = [4, 5, 6, 7].

?- append([1, 2, 3], X, [1, 2, 3]).
X = [].

?- append([1, 2, 3], X, [1, 2, 3, 6]).
X = [6].

?- append(X, Y, [1, 2, 3, 6]).
X = [],
Y = [1, 2, 3, 6] ;
X = [1],
Y = [2, 3, 6] ;
X = [1, 2],
Y = [3, 6] ;
```

```
X = [1, 2, 3],
Y = [6] ;
X = [1, 2, 3, 6],
Y = [] ;
false.

- sorted([1,2,1]).
false.

?- sorted([1, 2, 1]).
false.

?- sorted([1, 2, 1]).
false.

?- last([1, 2, 3], 3).
true.

?- last([1, 2, 3], X).
X = 3.

?- remove_all([1, 2, 1], X, L).
X = 1,
L = [2] ;
false.

?- somma_lista([1, 2, 3, 4], X).
X = 10.

coda([1,2,3], X).
X = [2,3]

?- duplico([1, 2, 3], X).
X = [1, 1, 2, 2, 3, 3].
```

2.2 laboratorio 2

Esercizio 6. *ancora sulle liste*

```
/* trovo min lista*/

% la lista da in automatico false
min([H], H).
min([H | T], H) :- min(T, X),
                   H =< X. % fa solo H
min([H | T], X) :- min(T, X),
                   H > X. % fa tutto il resto
```

```
?- min([3, 2, 2], X).
X = 2 .

?- min([2, 3, 4], X).
X = 2.
```

Esercizio 7. *ancora sulle liste*

```
/* rimuovo valore (una sola copia) da lista salvando in lista */
remove_one([], _, []).
remove_one([H | T], H, T). % se è la testa la rimuovo
remove_one([H | T], X, [H | S]):- remove_one(T, X, S),
                                  X \= H.
 /* la H la tengo e riattacco la nuova S
       e uso il primo casoi base e non il secondo
         usando il \=*/
```

```
?- remove_one([1, 2, 3, 4], 2, L).
L = [1, 3, 4] ;
false.

?- remove_one([1, 1, 2, 3, 4], 1, L).
L = [1, 2, 3, 4] ;
false.
```

Esercizio 8. *ancora sulle liste*

```
/* selection sort */
selection_sort([], []).
```

```
selection_sort(X, [H| T]) :- min(X, H), % minimo in testa
                             remove_one(X, H, X1), %toglie il minimo
                             selection_sort(X1, T). % rifa conl T

/* unisco liste in ordine */
merge([], X, X).
merge(X, [], X).
merge([H1 | T1], [H2 | T2], [H1 | T]) :- H1 =< H2,
                             merge(T1, [H2 | T2], T).
merge([H1 | T1], [H2 | T2], [H1 | T]) :- H2 =< H1,
                             merge([H1 | T1], T2, T).

/* spezzo in due lista */
split_in_two([],[],[]).
split_in_two([X],[X],[]).
split_in_two([H1, H2 | T],[H1 | T1],[H2 | T2]) :- split_in_two(T, T1, T2).

/* mergesort */
/* divide in 2 la lista, fa il mergesort delle due e merge dei risultati*/
mergesort([], []).
mergesort([X], [X]).
mergesort(L1, L2) :-  split_in_two(L1, X1, Y1),
                      mergesort(X1, X2),
                      mergesort(Y1, Y2),
                      merge(X2, Y2, L2).
```

```
?- selection_sort([2, 1, 4], L).
L = [1, 2, 4] ;
false.

?- merge([2,4], [1,3,5], X).
X = [1, 2, 3, 4, 5].

?- split_in_two([1,2,3,4],X, Y).
X = [1, 3],
Y = [2, 4].

?- split_in_two([1,2,4],X, Y).
X = [1, 4],
```

```
Y = [2] ;

?- mergesort([2,4,1,7,6,2,12,3], X).
X = [1, 2, 2, 3, 4, 6, 7, 12] .
```

Esercizio 9. *ancora sulle liste*

```
/* da più listre annidate a una */
/* da [[1, 3],[[4]]] */
/* uso definizione di lista e append */
listp([]).
listp([_ | _]). % vero solo se ho lista

append([], X, X).
append([H|T], L, [H|X]) :- append(T, L, X).

flatten([], []).
flatten([H | T], L) :- listp(H),
                       flatten(H, X),
                       flatten(T, Y),
                       append(X, Y, L). /* se testa è lista la appiattisco
                                           e la aggiungo al risultato */
flatten([H | T], [H | X]) :- flatten(T, X). /* altrimenti appiattisco la tail
                                                ricorsivamente tenendo la head*/
```

```
?- flatten([[1,2,3],[[4]]], X).
X = [1, 2, 3, 4]
```