Università di Bologna

Corso di Laurea in Informatica
Esercizi risolti sulla
RICORSIONE STRUTTURALE
Version 1 (tutti gli esami passati degli anni solari 2018 e 2019)

1. [**logica2018-2019_risolto**] Considerare la seguente sintassi delle liste di X: $L ::= [] \mid X :: L$ dove $::$ è associativo a destra. Scrivere la funzione ricorsiva f che su una lista L di numeri che restituisca la lista LL di liste **non vuote** di numeri tale che:

   (a) Se $LL = L_1 :: \ldots :: L_n :: []$ allora $L = L_1 @ \ldots @ L_n$ dove @ è la funzione che concatena due liste. In altre parole, LL è fatta da frammenti di L nell'ordine in cui occorono in L.

   (b) ogni L_i è una lista monotona, non crescente o non decrescente, di numeri

   (c) L_1 è monotona **non decrescente**

   (d) le liste L_i sono di **lunghezza massimale**, ovvero L_i e L_{i+1} non possono essere concatenate per ottenere un'unica lista monotona.

   Esempi:

   $$f(1 :: 3 :: 7 :: 6 :: 6 :: 4 :: 8 :: 10 :: []) = (1 :: 3 :: 7 :: []) :: (6 :: 6 :: 4 :: []) :: (8 :: 10 :: []) :: []$$

   $$f(7 :: 4 :: 2 :: 9 :: []) = (7 :: []) :: (4 :: 2 :: []) :: (9 :: []) :: []$$

   È possibile utilizzare funzioni ausiliarie su liste, da definirsi usando la ricorsione strutturale, funzioni ausiliarie su numeri (da non definirsi) e/o passare parametri ausiliari alle funzioni.

   - Primo problema:

     Specifica: data L restituire la lista di liste LL come specificato nell'esercizio.

     Funzione che lo risolve: $f(L)$

     Codice:
     ```
     f([]) = []
     f(N::L) =
       if L ≠ [] and N <= first(L) then
          add_to_first_list(N,f(L))
        else
          (N::[])::g(L)
     ```

Nota: nel caso $N :: L$ la chiamata ricorsiva strutturale $f(L)$ risolve il problema su L. Mi resta da capire cosa farne di N. Se N può fare parte della prima sequenza monotona di L, lo aggiungo ad essa. Altrimenti N deve formare una sequenza singoletto da solo e deve essere seguito dalla lista di sequennze che inizia con una monotona non crescente. Risolvo tale problema con la funzione g. La f e la g sono definite per ricorsione mutua: ognuna richiama l'altra solo su sottotermini immediati dell'input, rispettando i dettami della ricorsione strutturale. Un'altra soluzione possibile è che la $f()$ e la $g()$ siano la stessa funzione che prende in input un parametro aggiuntivo (un booleano) per determinare quale dei due problemi risolvono (prima sequenza non decrescente vs non crescente).

- Secondo problema:

Specifica: data L restituire la lista di liste LL come specificato nell'esercizio ma con la differenza che la lista L_1 deve essere monotona non crescente.

Funzione che lo risolve: $g(L)$

Codice:
```
g([]) = []
g(N::L) =
  if L ≠ [] and N >= first(L) then
     add_to_first_list(N,g(L))
   else
     (N::[])::f(L)
```

- Terzo problema:

Specifica: dato un numero N e una lista di liste di numeri LL, aggiungere N in testa alla prima lista di LL. Se LL non contiene liste, il comportamento non è specificato (= fate un poco come volete). Esempio: `add_to_first_list(3,(1::4::0::[])::(5::[])::[]) = (3::1::4::0::[])::(5::[])::[]`

Funzione che lo risolve: `add_to_first_list(N,LL)`

Codice:
```
add_to_first_list(N,[]) = []
add_to_first_list(N,L::LL) = (N::L)::LL
```

- Quarto problema:

Specifica: data una lista non vuota di numeri, restituirne il primo elemento. Se la lista è vuota il comportamento non è specificato.

Funzione che lo risolve: `first(L)`

Codice:

```
first([]) = 666999
first(N::L) = N
```

2. [**logica2017-2018**] Definizione: $N \Rightarrow P$ è positiva sse N è negativa e P è positiva; $P \Rightarrow N$ è negativa sse P è positiva e N è negativa; $\neg P$ è negativa sse P è positiva; $\neg N$ è positiva sse P è negativa; la variabile proposizionale A è positiva.

   Scrivere una funzione f ricorsiva su F tale che $f(F, tt) = tt$ sse F è positiva.

   Problema generalizzato: scrivere una funzione f ricorsiva su F tale che $f(F, b) = tt$ sse (F è positiva e $b = tt$ oppure F è negativa e $b = ff$).

   ```
   f(F1 ⇒ F2, b) = f(F2,b) && f(F1,not b)
   f(¬F, b) = f(F,not b)
   f(A,b) = (b=tt)
   ```

3. [**logica180917**] Considerare la seguente sintassi per le liste di numeri naturali:

   $$L ::= \epsilon \mid N; L$$

   dove il ; è associativo a destra e N è il tipo dei numeri naturali.

   Scrivere, per induzione strutturale sulla lista di numeri L, una funzione ricorsiva $f(L)$ che calcola il numero di sottosequenze monotone crescenti massimali di L. È possibile utilizzare funzioni ausiliarie definite anch'esse per ricorsione strutturale.

   Esempio: $f(3; 5; 2; 1; 4; 12; 6; 6; \epsilon) = 5$ in quanto le sottosequenze monotone crescenti massimali dell'input sono
   $(3; 5; \epsilon); (2; \epsilon); (1; 4; 12; \epsilon); (6; \epsilon); (6; \epsilon); \epsilon$.

   - Problema 1: $f(L)$ deve calcolare il numero di sottosequenze monotone crescenti massimali di L.

     ```
     f(ε) = 0
     f(N;L) = if smaller(N,L) then f(L) else 1 + f(L)
     ```

   - Problema 2: $smaller(N, L) = true$ sse L è non vuota e N è più piccolo della testa di L.

     ```
     smaller(N,ε) = false
     smaller(N,M;L) = N<M
     ```

4. [**logica180706**] Scrivere una funzione che, data in input una lista L di liste di numeri, restituisca la lista O di tutti e soli i numeri che occorrono in tutte le liste in L.

- Problema 1: data una lista LL di liste di numeri restituire la lista O di tutti e soli i numeri che occorrono in tutte le liste in LL.
```
f([]) = []
f(L::LL) = filter(L,LL)
```
- Problema 2: scrivere una funzione $filter(L, LL)$ per ricorsione strutturale su L che restituisca la lista degli elementi di L che occorrono anche in tutte le liste in LL.
```
f([],LL) = []
f(N::L,LL) = if meml(N,LL) then N::filter(L,LL) else filter(L,LL)
```
- Problema 3: scrivere una funzione $meml(N, LL)$ per ricorsione strutturale su LL che restituisca true sse N è un elemento di tutte le liste in LL.
```
meml(N,[]) = false
meml(N,L::LL) = mem(N,L) && meml(N,LL)
```
- Problema 4: scrivere una funzione $mem(N, L)$ per ricorsione strutturale su L che restituisca true sse N è un elemento della lista L.
```
mem(N,[]) = false
mem(N,M::L) = (N = M) || mem(N,L)
```

5. [**logica180621**] Sia A un qualunque tipo di dato e $\approx$: $A \times A \to \mathbb{B}$ l'implementazione di una relazione di equivalenza su A. Considerare la seguente sintassi per le liste di numeri naturali:

$$L ::= \epsilon \mid B; L$$

dove il ; è associativo a destra e B è un tipo qualunque.

Scrivere, per induzione strutturale su L, lista di elementi di A pensata come un insieme di elementi di A, una funzione $f(L)$ che ritorni la lista delle classi di equivalenza di A a meno di $\approx$. Ovvero, $f(L)$ deve ritornare la lista $L_1; \ldots; L_n; \epsilon$ dove

(a) $\forall i, L_i \neq \epsilon$

(b) $\forall i, \forall j, \forall x \in L_i, \forall y \in L_j, x \approx y = tt \iff (i = j)$

Come visto a lezione potete implementare, sempre per ricorsione strutturale, funzioni ausiliarie e potete passare parametri ulteriori alle vostre funzioni se necessario.

(a) Problema 1: $f(L)$ deve restituire la lista delle classi di equivalenza di elementi di L modulo la relazione `approx`.
`f([]) =` ϵ
`f(N,L) = add_to_class(N,f(L))`

(b) Problema 2: `add_to_class(N,LL)` deve aggiungere `N` alla lista di elementi, contenuta in `L`, che è un sottoinsieme di una classe di equivalenza, o creare una nuova lista/classe se la sua non è ancora presente in `LL`.
`add_to_class(N,[]) = N;`ϵ
`add_to_class(N,L;LL` = if mem(N,L) then (N;L);LL else L;add_to_class(N,LL)

(c) Problema 3: `mem(N,L)` deve ritornare `true` quando `N` fa parte della classe di equivalenza di cui `L` è un sottoinsieme.
`mem(N,[]) = false`
`mem(N,M;L) = e(N,M)`

6. **[logica180528]** Considerare la seguente grammita per alberi binari: $B ::= \epsilon \mid B{*}B$. Scrivere, facendo ricorso alla sola ricorsione strutturale, la funzione $s(B_1, B_2)$ che ritorna *true* sse B_2 è un sottoalbero di B_1. È possibile fare ricorso a funzioni ausiliarie e usare parametri ulteriori.

   Esempi:

   - $s(((\epsilon * \epsilon) * \epsilon) * \epsilon, \epsilon * \epsilon) = true$
   - $s(((\epsilon * \epsilon) * \epsilon) * \epsilon, \epsilon * (\epsilon * \epsilon)) = false$

   (a) Problema 1: $s(B_1, B_2)$ deve restituire *true* sse B_2 è un sottoalbero di B_1. Procediamo per ricorsione strutturale su B_1.
   $s(\epsilon, B_2) = equal(\epsilon, B_2)$
   $s(B_1^1 * B_1^2, B_2) = equal(B_1^1 * B_1^2, B_2)\ or\ s(B_1^1, B_2)\ or\ s(B_1^2, B_2)$

   (b) Problema 2: $equal(B_1, B_2)$ deve restituire *true* sse B_1 e B_2 sono alberi identici.

$equal(\epsilon, B_2) = is\_epsilon(B_2)$
$equal(B_1^1 * B_1^2, B_2) = equal(B_1^1 * B_1^2, B_2)\ or\ s(B_1^1, first(B_2))\ or\ s(B_1^2, second(B_2))$
$is\_epsilon(\epsilon) = true$
$is\_epsilon(B_1 * B_2) = false$
$first(\epsilon) = \perp$
$first(B_1 * B_2) = B_1$
$second(\epsilon) = \perp$
$second(B_1 * B_2) = B_2$

Nota: il codice precedente è verboso. La totalità dei linguaggi di programmazione che supportano pattern-matching vi permettono di scriverlo in forma condensata come segue, matchando entrambi i parametri contemporaneamente:

$equal(\epsilon, \epsilon) = true$
$equal(B_1^1 * B_1^2, B_2^1 * B_2^2) = equ$
$equal(\epsilon, B_2^1 * B_2^2) = false$
$equal(B_1^1 * B_1^2, \epsilon) = false$

7. **[logica180209_fila2]** Considerare la seguente sintassi per le liste di numeri naturali:

$$L ::= [] \mid \mathbb{N} :: L$$

dove il :: è associativo a destra. Scrivere, per induzione strutturale su L, una funzione $g(L_1, L_2)$ che ritorni il booleano tt se la lista L_1 non contiene L_2 come sottolista.

Esempi:
$g(0 :: 1 :: 2 :: 3 :: [], 1 :: 2 :: []) = tt$
$g(0 :: 1 :: 2 :: 3 :: [], 0 :: 2 :: []) = ff$.

L'unica funzione della quale potete assumere l'esistenza è $\cdot = \cdot$ utilizzabile per confrontare due numeri. Potete implementare funzioni ausiliarie, sempre per ricorsione strutturale.

(a) Problema 1: $g(L_1, L_2)$ ritorna tt sse L_1 non contiene L_2 come sottolista.
$g([], L_2) = not\ equal([], L_2)$
$g(N :: L_1, L_2) = not\ equal(N :: L_1, L_2) \&\& g(L_1, L_2)$

(b) Problema 2: $equal(L_1, L_2)$ ritorna tt sse le due liste sono identiche.
$equal([], []) = tt$
$equal(N_1 :: L_1, N_2 :: L_2) = (N_1 = N_2) \&\& equal(L_1, L_2)$
$equal([], N :: L) = ff$
$equal(N :: L, []) = ff$

Come nell'esercizio precedente il codice dell'*equal* può essere riscritto per evitare di fare matching contemporaneamente su entrambi i parametri, usando tre funzioni ausiliarie $is\_empty, head, tail$.

8. **[logica180112]** Considerare la seguente sintassi per le liste di numeri naturali:

$$L ::= \epsilon \mid \mathbb{N}; L$$

dove il ; è associativo a destra. Scrivere, per induzione strutturale su L, una funzione $f(L)$ che ritorni il booleano *tt* se la lista L è palindroma e *ff* altrimenti.

Non potete assumere l'esistenza di nessuna operazione sulle liste (compresa, per esempio, l'uguaglianza). Come visto a lezione potete implementare, sempre per ricorsione strutturale, funzioni ausiliarie.

(a) Problema 1: $f(L)$ ritorna *tt* sse L è palindroma.
$f(L) = equal(L, reverse(L, \epsilon))$

(b) Problema 2: $reverse(L_1, L_2)$ ritorna la lista L_1 scritta da destra a sinistra concatenata alla lista L_2. Esempio: $reverse(1; 2; 3; \epsilon, 4; 5; \epsilon) = 3; 2; 1; 4; 5; 6; \epsilon$
$reverse(\epsilon, L) = L$
$reverse(N; L_1, L_2) = reverse(L_1, N; L_2)$

(c) Problema 3: $equal(L_1, L_2)$ ritorna *tt* sse le due liste sono identiche. Vedi soluzione **[logica180209_fila2]**.