PROGRAMMAZIONE I (A,B) - a.a. 2016-17
Quarto Appello – 11 Luglio 2017

Esercizio 1

Si scriva una funzione **C** che, dato un array a di dimensione dim_a e un array b di dimensione dim_b, restituisce il valore di verità della seguente formula:

$$\forall i \in [0, dim_a).\exists j \in [0, dim_b).((a[i] = b[j]) \wedge (\forall k \in [0, dim_b).((k \neq j) \Rightarrow (b[j] \neq b[k]))))$$

SOLUZIONE Seguendo pedissequamente la formula logica si può costruire una soluzione che prevede tre cicli (uno per ogni quantificatore $\forall$ o $\exists$). Per chiarezza, si definisce una funzione ausiliaria che realizza il controllo dell'ultimo $\forall$ contenente l'implicazione.

```
int implicazione(int c[], int dim_c, int j)
{
  int k=0;
  int ok=1;
  while (k<dim_c && ok)
  {
    if ((k!=j) && (c[j]==c[k])) ok=0;
    k++;
  }
  return ok;
}

int check(int a[], int dim_a, int b[], int dim_b)
{
  int i=0;
  int ok=1;
  while (i<dim_a && ok)
  {
    int j=0;
    ok=0;
    while (j<dim_b && !ok)
    {
      ok = (a[i]==b[j] && implicazione(b, dim_b, j));
      j++;
    }
    i++;
  }
  return ok;
}
```

Il testo in realtà chiede di controllare che ogni elemento di a sia presente una sola volta in b. Gli elementi di b che non sono presenti in a possono invece essere ripetuti. E' conveniente dare una soluzione modulare, prevedendo una funzione ausiliaria che verifica se un valore è presente in un array esattamente una volta.

```
int esattamente_uno(int c[], int dim_c, int x)
{
  int i;
  int cont=0;
  for (i=0; i<dim_c; i++)
    if (c[i]==x) cont++;
  return cont==1;
}

int check(int a[], int dim_a, int b[], int dim_b)
{
  int i=0;
  int ok=1;
  while (i<dim_a && ok) {
    ok=esattamente_uno(b, dim_b, a[i]);
    i++;
  }
  return ok;
}
```

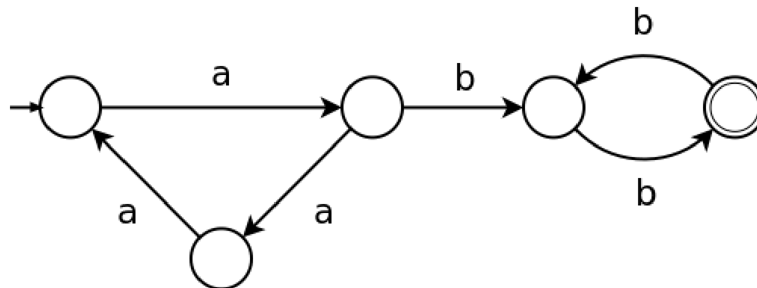
Esercizio 2

Dato il seguente linguaggio sull'alfabeto $\Lambda = \{a, b\}$

$$\mathcal{L} = \{a^n b^{2m} \mid n > 0 \wedge n\%3 = 1 \wedge m > 0\}$$

si verifichi se il linguaggio è regolare o meno (fornendo una opportuna dimostrazione) e si definisca una grammatica che lo genera.

SOLUZIONE Il linguaggio è regolare, in quanto è possibile definire un automa a stati finiti che lo accetta. Un automa che accetta il linguaggio $\mathcal{L}$ è, ad esempio, il seguente:



La grammatica può essere definita semplicemente imitando le transizioni dell'automa come segue:

```
S -> aT
T -> aU | bV
U -> aS
V -> b | bZ
Z -> bV
```

In alternativa, è immediato notare che esistono due categorie sintattiche distinte nel linguaggio, dato che gli esponenti di 'a' e di 'b' non sono fra loro correlati. Si ottiene dunque in maniera semplice la seguente grammatica:

```
S -> aA
A -> aaaA | B
B -> bbB | bb
```

Esercizio 3

Si suppongano predefiniti i tipi

```
struct el {int info; struct el *next;};
typedef struct el ElementoDiLista;
typedef ElementoDiLista* ListaDiElementi;
```

Si scriva in **C** una procedura che, presa una lista, elimina da essa tutti gli elementi compresi tra il primo elemento che contiene il valore 0 e l'ultimo elemento che contiene il valore 0. Tali due elementi contenenti 0 non devono essere eliminati. Se la lista contiene meno di due elementi contenenti il valore 0, nessuna eliminazione dovrà essere eseguita.

SOLUZIONE E' conveniente dare una soluzione modulare definendo due funzioni ausiliarie che restituiscono, rispettivamente, i puntatori al primo e all'ultimo elemento contenenti zero presenti nella lista. Se i due puntatori ottenuti dalle funzioni ausiliare sono diversi tra loro ed entrambi diversi da NULL, allora la lista contiene almeno due elementi contenti zero. In tal caso si procederà con le eliminiazioni.

```
ListaDiElementi trova_primo_zero (ListaDiElementi l) {
  int trovato = 0;
  while (l != NULL && !trovato) {
    if (l->info==0) trovato = 1;
    l = l->next;
  }
  return l;
}

ListaDiElementi trova_ultimo_zero (ListaDiElementi l) {
  ListaDiElementi zero = NULL;
  while (l != NULL) {
    if (l->info==0) zero=l;
    l = l->next;
  }
  return zero;
}

void elimina_tra_zeri(ListaDiElementi l) {
  ListaDiElementi prec = NULL;
  ListaDiElementi corr = trova_primo_zero(l);
  ListaDiElementi last = trova_ultimo_zero(l);

  if (corr!=NULL && last!=NULL && corr!=last) {
    prec = corr;
    corr = corr->next;
    while (corr != last) {
      prec->next = corr->next;
      free(corr);
      corr = prec->next;
    }
```

```
  }
}
```

Esercizio 4

Si definisca in CAML una funzione

`contafrequente : int list -> int`

che, data una lista `lis` di interi, calcola il numero di occorrenze dell'elemento che occorre il maggior numero di volte nella lista. La funzione può essere liberamente definita facendo uso o meno della ricorsione esplicita.

SOLUZIONE Una soluzione ricorsiva che utilizza la funzione ausiliaria `conta`.

```
let rec conta lis n =
  match lis with
    [] -> 0
    x::xs when (x!=n) -> conta xs n
    x::xs when (x=n) -> (conta xs n) + 1;;

let rec contafrequente lis =
  match lis with
    [] -> 0
    x::xs -> let n = (conta xs x)+1
             in
             let m = (contafrequente xs)
             in
             if (n>m) then n else m;;
```

Una soluzione che non usa la ricorsione esplicita, basata sulla stessa funzione `conta` definita nella soluzione precedente.

```
let contafrequente lis =
  let f x (l,n) = let m = (conta l x) + 1
                  in
                  if (m > n) then (x::l , m) else (x::l , n)
  in
  snd (foldr f ([],0) lis);;
```

Una soluzione ricorsiva con accumulatore, di nuovo basata sulla funzione `conta` gi vista.

```
let contafrequente lis =
  let rec contafrequente_acc l n =
    match l with
      [] -> n
      x::xs -> let m = (conta x xs) +1
               in
               let max = if (m>n) then m else n
               in
               contafrequente_acc xs max
  in
    contafrequente_acc lis 0;;
```