PROGRAMMAZIONE I (A,B) - a.a. 2017-18
I appello – 17 Gennaio 2018

Esercizio 1

Data la seguente grammatica sull'alfabeto $\Sigma = \{a, b, c\}$

$$S \to cAc$$
$$A \to aAa \mid bAb \mid c$$

si dia il linguaggio generato dalla grammatica e si verifichi se tale linguaggio è regolare o meno.

SOLUZIONE Il linguaggio generato dalla grammatica è il seguente

$$L = \{\ c\alpha c\beta c\ \mid\ \alpha, \beta \in \Sigma^*\ \wedge\ |\alpha|_c = |\beta|_c = 0\ \wedge\ \beta = \alpha \text{ rovesciato}\}$$

oppure, equivalentemente:

$$L = \{\ c\alpha c\beta c\ \mid\ \alpha, \beta \in \{a, b\}^*\ \wedge\ \beta = \alpha \text{ rovesciato}\}$$

Volendo definire formalmente la nozione di stringa rovesciata si sarebbe anche potuto scrivere come segue:

$$L = \{\ c\alpha c\alpha^R c\ \mid\ \alpha, \beta \in \{a, b\}^*\}$$

dove l'operazione R è definita ricorsivamente come $\epsilon^R = \epsilon$ e $(xw)^R = w^R x$ con $x \in \Sigma$ e $w \in \Sigma^*$.

Il linguaggio non è regolare. Per dimostrarlo usando il Pumping Lemma scegliamo, per un generico $n \in IN$, la stringa

$$w = ca^n ca^n c$$

Con questa scelta, considerando tutte le possibili suddivisioni di w in xyz tali che $|xy| \leq n$ e $y \neq \epsilon$ abbiamo due casi:

$$x = \epsilon$$
$$y = ca^s$$
$$z = a^t ca^n c$$

con $0 \leq s < n$ e $s + t = n$, e

$$x = ca^r$$
$$y = a^s$$
$$z = a^t b^n c^n$$

con $0 \leq r < n$, $0 < s < n$ e $r + s + t = n$.

In entrambi i casi, considerando la stringa $xy^i z$ con $i = 0$ otteniamo un risultato che non appartiene al linguaggio L.

Esercizio 2

Si scriva una funzione **C** che, dato un array a di dimensione dim, restituisca il valore di verità della seguente formula:

$$\forall i \in [1, dim).\exists j \in [0, i).\big(a[i] = \sum_{k \in [j,i)} a[k]\ \big)$$

SOLUZIONE Modularmente, seguendo la struttura della formula possiamo definire una funzione per calcolare, dati a, j e i, il valore di

$$\sum_{k \in [j,i)} a[k]$$

come segue:

```
int somma(int a[], int j, int i) {
  int s = 0;
  for (int k=j; k<i; k++)
    s = s + a[k];
  return s;
}
```

A questo punto, la formula proposta dall'esercizio risulta essere equivalente alla seguente:

$$\forall i \in [1, dim). \exists j \in [0, i). \big(a[i] = somma(a, j, i)\ \big)$$

Procediamo ancora modularmente per calcolare, dati a, dim e i, il valore di

$$\exists j \in [0, i). \big(a[i] = somma(a, j, i)\ \big)$$

come segue:

```
int exists(int a[], int dim, int i) {
  int trovato=0;
  int j=0;
  while (j<dim && !trovato)
    if (a[i]==somma(a,j,i))
      trovato=1;
    else
      j++;
  }
  return trovato;
}
```

A questo punto, la formula proposta dall'esercizio risulta essere equivalente alla seguente:

$$\forall i \in [1, dim). exists(a, dim, i)\ \big)$$

che possiamo verificare, dati a e dim, come segue:

```
int check(int a[], int dim) {
  int ok=1;
  int i=1;
  while (i<dim && ok) {
    if (!exists(a,dim,i))
      ok=0;
    else
      i++;
  }
  return ok;
}
```

Una soluzione alternativa, più efficiente, consiste nel non ricalcolare la somma dall'inizio per ogni valore di j, andando a sostituire alle funzioni `somma` e `exists` della soluzione precedente, la seguente funzione `exists` (la `check` rimane uguale):

```
int exists(int a[], int dim, int i) {
  int trovato = 0;
  int s=0;
  int j=(i-1);
  while (j>=0 && !trovato) {
    s += a[j];
    if (a[i]==s)
      trovato=1;
    else
      j--;
  }
  return trovato;
}
```

Un'altra soluzione ancora, non modulare, prevede l'utilizzo di tre cicli annidati:

```
int check (int a[], int dim) {
  int ok=1;
  int i=1;
  while (i<dim && ok) {
    int trovato=0;
    int j=0;
    while (j<i && !trovato) {
      int s=0;
      for (int k=j; k<i; k++) {
        s+=a[k];
      }
      if (a[i]==s)
        trovato = 1;
      else
        j++;
    }
    if (!trovato)
      ok = 0;
    else
      i++;
  }
  return ok;
}
```

Anche questa soluzione può essere resa più efficiente sostituendo i due cicli più interni con un unico ciclo while che calcola la somma decrementando j ad ogni passo.

Esercizio 3

Si definisca in CAML, senza usare la ricorsione esplicita, una funzione

`split_fine : int list -> int list * int list`

che, data una lista `lis` di interi, restituisce la coppia `(lis1,lis2)` tale che

- `lis1` e `lis2` sono due sottoliste (porzioni possibilmente vuote) di `lis`,
- `lis1` concatenato a `lis2` corrisponde esattamente all'intera lista `lis`
- e `lis2` è la sottolista FINALE più lunga possibile i cui elementi sono disposti in ordine strettamente crescente.

SOLUZIONE Una possibile soluzione è la seguente:

```
let split_fine lis =
  let f x (lis1,lis2) =
    match lis1,lis2 with
         [],[] -> ([],[x])
      |  [],y::ys -> if x < y then ([],x::lis2)
                              else ([x],lis2)
      |  y::ys,l -> (x::lis1,lis2)
  in
    foldr f ([],[]) lis;;
```

L'idea è che si inseriscono gli elementi che si incontrano (partendo dal fondo della lista) nella seconda lista della coppia. Si procede così fintanto che si incontrano valori sempre strettamente minori. Appena si incontra un valore che non è strettamente minore di quelli già incontrati, si inizia a riempire la prima lista della coppia. Da quel momento (ossia, non appena la prima lista diventa non vuota) si continua sempre a inserire nella prima lista della coppia.

Esercizio 4

Si definisca in CAML una funzione ricorsiva in modo esplicito

`split_inizio : int list -> int list * int list`

che, data una lista `lis` di interi, restituisce la coppia `(lis1,lis2)` tale che

- `lis1` e `lis2` sono due sottoliste (porzioni possibilmente vuote) di `lis`,
- `lis1` concatenato a `lis2` corrisponde esattamente all'intera lista `lis`
- e `lis1` è la sottolista INIZIALE più lunga possibile i cui elementi sono disposti in ordine strettamente crescente.

SOLUZIONE Una possibile soluzione è la seguente:

```
let rec split_inizio lis =
  match lis with
      [] -> ([],[])
    | x::[] -> ([x],[])
    | x::y::xs -> if x<y then let (lis1,lis2) = split_inizio (y::xs)
                              in (x::lis1,lis2)
                         else
                              ([x],y::xs);;
```

Usando degli accumulatori è possibile definire anche la seguente soluzione, che inizia mettendo l'intera lista come secondo elemento della coppia e poi sposta un elemento alla volta nel primo elemento della coppia interrompendosi quando raggiunge un valore che non è maggiore del precedente:

```
let split_inizio lis =
  let rec split_accum lis1 lis2 =
    match lis2 with
         [] -> (lis1,[])
      |  x::[] -> (lis1@[x],[])
      |  x::y::xs -> if x < y then split_accum (lis1@[x]) (y::xs)
                              else (lis1@[x],y::xs)
  in
     split_accum [] lis;;
```