Fondamenti di Informatica A
Soluzioni Laboratorio 2
Funzioni e ricorsione

Danilo Pianini
danilo.pianini@unibo.it
Mirko Viroli
mirko.viroli@unibo.it

Alma Mater Studiorum—Università di Bologna

18 marzo 2013

Ricorsione

- `int f(int n){ return n==0?  1 :  n*f(n-1); }`
- Realizza il fattoriale.
- Fino a `f(12)`, incluso. Poi gli interi non sono abbastanza grandi da contenere il risultato (tant'è che `f(13)/f(12)` dà come risultato 4, e non 13.
- Utilizzando il tipo `long`, si riesce a migliorare: `long f(long n){ return n==0?  1 :  n*f(n-1); }`. Si arriva a computare correttamente `f(20)`
- Non tail: l'ultima operazione da effettuare non è la chiamata ricorsiva ma la somma.

Ricorsione

Si analizzi la funzione `f(int, int)`:

```
int f(int x, int y) {
   return f(x, y, x, y);
}

int f(int x, int y, int z, int w) {
   return z==w ? z :
      z>w ? f(x, y, z, w+y) :
         f(x, y, z+x, w);
}
```

- Realizza il minimo comune multiplo
- Ricorsione tail: in ogni caso, l'ultima operazione effettuata è la chiamata ricorsiva.

Modifica di funzioni

```
int sum(int[] array){
   return array.length > 4 ? 0 : sum(array, 0);
}
int sum(int[] array,int pos){
   return (pos==array.length) ? 0 :
      array[pos]+sum(array,pos+1);
}

boolean test_sum(){
   return sum(new int[]{10,20,30})==60 &&
      sum(new int[]{10,20,30,40,50})==0 &&
      sum(new int[]{10})==10 &&
      sum(new int[]{})==0 &&
      sum(new int[]{-10,-20,-30})==-60;
}
```

Modifica di funzioni

Si consideri `sum(int[])`, nella versione non modificata.

- Vengono sommati solo gli elementi in posizione pari, ma si rischia di andare fuori dall'array.
- Sostituendo == con >= si risolve il problema dell'uscita dall'array, poiché non appena l'indice esce dall'intervallo consentito si ritorna 0 (che viene sommato all'indietro con i valori precedenti).

Costruzione di funzioni a partire da funzioni esistenti

```
int sum_odd(int[] array){
   return sum_odd(array, 1);
}

int sum_odd(int[] array, int pos){
   return (pos>=array.length) ? 0 :
      array[pos] + sum_odd(array, pos + 2);
}

boolean test_sum_odd(){
   return sum_odd(new int[]{10,20,30})==20 &&
      sum_odd(new int[]{10,20,30,40,50})==60 &&
      sum_odd(new int[]{10,20,30,40})==60 &&
      sum_odd(new int[]{10,20})==20 &&
      sum_odd(new int[]{10})==0 &&
      sum_odd(new int[]{})==0 &&
      sum_odd(new int[]{-10,-20,-30})==-20;
}
```

Costruzione di funzioni a partire da funzioni esistenti

```
int sum_abs(int[] array) {
   return sum_abs(array, 0);
}

int sum_abs(int[] array, int pos) {
   return (pos==array.length) ? 0 :
      (array[pos] < 0 ? -array[pos] : array[pos])
      + sum_abs(array, pos + 1);
}

boolean test_sum_abs() {
   return sum_abs(new int[]{10,20,30})==60 &&
      sum_abs(new int[]{10,20,-30,-40,50})==150 &&
      sum_abs(new int[]{-10,20,30,40})==100 &&
      sum_abs(new int[]{-10,-20})==30 &&
      sum_abs(new int[]{10})==10 &&
      sum_abs(new int[]{})==0;
}
```

Costruzione di funzioni a partire da funzioni esistenti: riconoscitori EBNF

```
boolean ric123(int[] a) {
   return a.length > 0 && a[0] == 1 && ric23(a, 1);
}
boolean ric23(int[] a, int pos) {
   return pos == a.length ||
      ((a[pos] == 2 || a[pos] ==3 ) ? ric23(a, pos + 1) :
      false);
}
boolean test_ric123(){
   return ric123(new int[]{1}) &&
      ric123(new int[]{1,2}) &&
      ric123(new int[]{1,2,3}) &&
      ric123(new int[]{1,3,3,2,3,2}) &&
      !ric123(new int[]{}) &&
      !ric123(new int[]{1,2,3,4});
}
```

Costruzione di funzioni a partire da funzioni esistenti: riconoscitori EBNF

```
boolean ric1234(int[] a) {
   return ric12(a, 0);
}
boolean ric12(int[] a, int pos) {
   return pos == a.length ? false :
      (a[pos] == 2 || a[pos] == 1)? ric12(a, pos + 1) :
      ric34(a, pos);
}
boolean ric34(int[] a, int pos) {
   return pos + 2 == a.length && a[pos] == 3 && a[pos+1] == 4;
}
boolean test_ric1234(){
   return ric1234(new int[]{3,4}) &&
      ric1234(new int[]{1,2,3,4}) &&
      ric1234(new int[]{1,2,2,2,1,3,4}) &&
      !ric1234(new int[]{}) &&
      !ric1234(new int[]{1,2,3,4,1}) &&
      !ric1234(new int[]{1,2,3,4,2});
}
```

Esercizi aggiuntivi

Da questo punto, vengono proposti esercizi aggiuntivi. Sono consigliati per coloro che hanno completato **tutti** i precedenti con successo e desiderano esercitarsi ulteriormente.

Ricorsione

Si analizzi la funzione `f(int, int)`:

```
int f(int x) { return x == 0 ? 0 : f2(x-1); }
int f2(int x) { return x == 1 ? 1 : f(x-1); }
```

- Con `f(5)`, si ha uno stack overflow: le due funzioni si richiamano l'un l'altra all'infinito, riempiendo l'area di memoria riservata a contenere lo stack.

Progettare nuove funzioni

```
int length(int[] array) {
   return array.length;
}

boolean test_length() {
   return length(new int[]{10,20,30})==3 &&
      length(new int[]{10,20,-30,-40,50})==5 &&
      length(new int[]{-10,20,30,40})==4 &&
      length(new int[]{-10,-20})==2 &&
      length(new int[]{10})==1 &&
      length(new int[]{})==0;
}
```

Progettare nuove funzioni

```
int evens(int[] array) {
   return evens(array, 0, 0);
}

int evens(int[] array, int pos, int buf) {
   return pos == array.length ? buf :
      evens(array, pos + 1, buf + (array[pos] % 2 == 0 ? 1
}

boolean test_evens() {
   return evens(new int[]{12,22,32})==3 &&
      evens(new int[]{1,2,-3,-4,5})==2 &&
      evens(new int[]{-1,2,3,4})==2 &&
      evens(new int[]{-1,-2})==1 &&
      evens(new int[]{10})==1 &&
      evens(new int[]{})==0;
}
```

Progettare nuove funzioni

```
boolean arrayEquals(int[] a, int[] b) {
return a.length != b.length ? false : arrayEquals(a, b, 0);
}

boolean arrayEquals(int[] a, int[] b, int pos) {
   return pos == a.length ? true :
      a[pos] != b[pos] ? false :
      arrayEquals(a, b, pos + 1);
}

boolean test_arrayEquals() {
   return arrayEquals(new int[]{}, new int[]{}) &&
   arrayEquals(new int[]{1,2,3}, new int[]{1,2,3}) &&
   arrayEquals(new int[]{1,0,0,87}, new int[]{1,0,0,87})&&
   !arrayEquals(new int[]{}, new int[]{0}) &&
   !arrayEquals(new int[]{0,1,2}, new int[]{2,1,0}) &&
   !arrayEquals(new int[]{0,0,0,0}, new int[]{0,0,0,1});
}
```

Progettare nuove funzioni: riconoscitori EBNF

```
/* Note that 1{1}1 = 11{1} */
boolean ric111(int[] a) {
   return a.length > 1 && a[0] == 1 && a[1] == 1 ?
      ric111(a, 1) : false;
}

boolean ric111(int[] a, int pos) {
   return pos == a.length ||
      (a[pos] == 1 && ric111(a, pos+1));
}
```

Progettare nuove funzioni: riconoscitori EBNF

```
boolean ric12(int[] a) {
   return a.length == 0 || ric12(a, 0);
}

boolean ric12(int[] a, int pos) {
   return pos == a.length ? true :
      (a[pos] == 1 || a[pos] == 2) && ric12(a, pos + 1);
}
```

Progettare nuove funzioni: riconoscitori EBNF

```
boolean ric123(int[] a) {
   return a.length > 1 && a[0] == 1 && ric23(a, 1);
}
boolean ric23(int[] a, int pos) {
   return pos == a.length ? false : a[pos] == 2 ?
      ric23(a, pos + 1) : ric3(a, pos);
}
boolean ric3(int[] a, int pos) {
   return pos == a.length -1 && a[pos] == 3;
}
```

Progettare nuove funzioni: riconoscitori EBNF

```
boolean ric1234(int[] a) {
   return a.length == 0 || ric12(a, 0);
}
boolean ric12(int[] a, int pos) {
   return pos == a.length ||
      ((a[pos] == 1 || a[pos] == 2) ? ric12(a, pos + 1) :
      ric34(a, pos));
}
boolean ric34(int[] a, int pos) {
   return pos == a.length ||
      ((a[pos] == 3 || a[pos] == 4) ? ric34(a, pos + 1) :
      false);
}
```