Lab07 – Array/Slice

Esercizio 1 – Call by value/Call by reference

Note introduttive:

a) Quando una variabile `var` viene passata per valore ad una funzione (call by value), la funzione lavora su una copia `copyOfVar` della variabile (e dei dati contenuti in essa). Se la funzione cambia il valore di `copyOfVar`, il valore della variabile `var` non viene modificato.

Es. di chiamata di funzione con una variabile `var` passata per valore: `funcionName(var)`

La signature della funzione sarà del tipo:

```
func funcionName(copyOfVar typeOfVar)
```

b) Quando una variabile `var` viene passata per indirizzo ad una funzione (call by reference), la funzione lavora può modificare il valore della variabile `var`.

Es. di chiamata di funzione con una variabile `var` passata per valore: `funcionName(&var)`

La signature della funzione sarà del tipo:

```
func funcionName(ptrToVar *typeOfVar)
```

Quando il simbolo `*` precede un tipo, il simbolo `*` è un modificatore di tipo.

`ptrToVar` è una variabile di tipo puntatore a `typeOfVar` che contiene l'indirizzo di memoria di associato a `var`.

`*ptrToVar` rappresenta il valore contenuto in `var`.

Quando il simbolo `*` precede una variabile di tipo puntatore a `typeOfVar`, il simbolo `*` è detto operatore di dereferenziazione.

c) La seguente espressione è sempre verificata: `var == *(&var)`

d) Variabili di tipo base (int, float, bool, string,...) sono sempre passate per valore se non specificato altrimenti.

Alla luce delle note introduttive riportate, analizziamo insieme il funzionamento del seguente programma.

```
func main() {

	var a int = 10
	var b float64 = 12.5
	var c byte = 'A'

	fInt(a)
	fFloat(b)
	fByte(c)
	fmt.Printf("%d %f %c\n",a, b, c)

	a = fIntRet(a)
	b = fFloatRet(b)
	c = fByteRet(c)
	fmt.Printf("%d %f %c\n",a, b, c)

	fIntPtr(&a)
	fFloatPtr(&b)
	fBytePtr(&c)
	fmt.Printf("%d %f %c\n",a, b, c)
}

func fInt(x int) {
	x = x + 10
}

func fFloat(x float64) {
	x = x + 10
}

func fByte(x byte) {
	x = x + 10
}

func fIntRet(x int) int {
	return x + 10
}

func fFloatRet(x float64) float64 {
	return x + 10
}

func fByteRet(x byte) byte {
	return x + 10
}

func fIntPtr(x *int) {
	*x += 10
}

func fFloatPtr(x *float64) {
	*x += 10
}

func fBytePtr(x *byte) {
	*x += 10
}
```

Esercizio 2 – Fattoriale

Dato un numero intero `n` non negativo (maggiore o uguale a 0) letto da tastiera, creare e richiamare una funzione che stampi il risultato del fattoriale di tutti i numeri compresi tra 1 e `n` (estremi inclusi).

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go run fattoriale/fattoriale.go
Inserisci un numero:
10
Il fattoriale di 1 è 1
Il fattoriale di 2 è 2
Il fattoriale di 3 è 6
Il fattoriale di 4 è 24
Il fattoriale di 5 è 120
Il fattoriale di 6 è 720
Il fattoriale di 7 è 5040
Il fattoriale di 8 è 40320
Il fattoriale di 9 è 362880
Il fattoriale di 10 è 3628800
```

Esercizio 3 – Cosa stampa il programma?

```go
func main() {

	var a = [...]int{1, 2, 3, 4}

	fmt.Println(a)

	for i:=0; i <len(a); i++ {
		fmt.Println("Indice", i, " - Valore:", a[i])
	}

	fmt.Println()

	for i, v := range a {
		fmt.Println("Indice", i, " - Valore:", v)
	}
}
```

Esercizio 4 – Cosa stampa il programma?

```
func main() {

	a := [...]int{1, 2, 3, 4}

	fmt.Println(a)

	for i:=len(a)-1; i>=0; i-- {
		fmt.Println("Indice", i, " - Valore:", a[i])
	}

	fmt.Println()

	for i, _ := range a {
		fmt.Println("Indice", i, " - Valore:", a[len(a)-1-i])
	}
}
```

Esercizio 5 – Differenze di potenze

Scrivete un programma che inizializzi un array di reali di dimensione 10 (`arr1`) in modo che l'i-esimo elemento dell'array contenga il valore 2^i.

Si dichiari un secondo array di reali di dimensione 10 (`arr2`) in cui memorizzare i valori che si ottengono valutando la differenza $2^{(i+1) \bmod 10} - 2^i$ (i =0,...,9); per farlo si riusino i valori memorizzati nell'array `arr1`.

Si utilizzi il costrutto `for i:=0, i<len(arr1); i++ {...}` per inizializzare `arr1`, ed il costrutto `for i:=range arr2 {...}` per stampare i valori contenuti in `arr2`.

Si dichiari infine un terzo array di reali di dimensioni 10 (`arr3`) che, propriamente inizializzato, permetta di stampare a ritroso i valori contenuti in `arr2` con il costrutto `for _ , v :=range arr3 {...}`.

Esempio di funzionamento

```
$ go run differenze/differenze.go
1 2 4 8 16 32 64 128 256 512
```

```
 1 2 4 8 16 32 64 128 256 -511
-511 256 128 64 32 16 8 4 2 1
```

Esercizio 6 – Fibonacci

Scrivere un programma che calcoli i primi 50 numeri di Fibonacci e li stampi a video.

La successione di Fibonacci è definita in modo ricorsivo:

$$F(n) = \begin{cases} 1 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ F(n-1) + F(n-2) & \text{altrimenti} \end{cases}$$

I numeri della successione possono comunque essere calcolati iterativamente mediante l'utilizzo di un array.

Esempio di funzionamento

```
$ go run fibonacci/fibonacci.go
[1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711
28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141
267914296 433494437 701408733 1134903170 1836311903 2971215073 4807526976
7778742049 12586269025]
```

Esercizio 7 - Analisi di frammenti di codice - Array e slice/Costrutti "for" e "for...range"

Cosa stampa questo codice?

```go
func main() {

  a := [...]int{1, 2, 3, 4, 5, 6}
  b := a
```

```
  c := a[:]
  d := a[2:]

  for i, v := range d {
     fmt.Print(i,": ",v,"\n")
  }
  fmt.Println()

  c[0] = c[1] + c[2]
  d[1] = d[0] + c[0]

  for i, v := range a {
     fmt.Print(i,": ",v,"\n")
  }
  fmt.Println()

  for _, v := range c {
     fmt.Print(v," ")
  }
  fmt.Println()

  for i := range b {
     fmt.Print(i,": ",b[i],"\n")
  }
  fmt.Println()

  for _, v := range b {
     v *= 2
  }
  for i := 0; i<len(b); i++ {
     fmt.Print(i,": ",b[i],"\n")
  }
  fmt.Println()

}
```

Esercizio 8 – Slicing

Scrivete un programma che inizializzi un array di interi di dimensione 10 (`arr1`) in modo che l'i-esimo elemento dell'array contenga il valore 2^i.

Si stampi il contenuto di `arr1`.

Si effettui lo slicing dell'array ottenendo la slice `slice1` contenente gli elementi `arr1[4], arr1[5], arr1[6]`.

Si stampi il contenuto di `slice1`.

Si faccia crescere `slice1` affinché contenga gli elementi `arr1[4], ..., arr1[9]` (reslicing).

Si stampi il contenuto di `slice1`.

Si effettui lo slicing dell'intero array `arr1` ottenendo la slice `slice2`.

Si completi il seguente blocco di codice per ottenere la stampa di tutti gli elementi in `arr1`:

```
for i:=0; i<10; i++ {
            fmt.Print(slice2[0], " ")
            …
}
fmt.Println()
```

Si effettui lo slicing dell'intero array `arr1` ottenendo la slice `slice3`.

Si completi il seguente blocco di codice per ottenere la stampa di tutti gli elementi in `arr1` in ordine inverso:

```
for i:=0; i<10; i++ {
            fmt.Print(slice3[len(slice3)-1], " ")
            …
}
fmt.Println()
```

NOTA

Una slice `s` non può essere modificata per accedere ad elementi di un array che precedono quello contenuto in `s[0]`. L'istruzione `s = s[-1:]` genera un errore in fase di compilazione.

Esempio di funzionamento

```
$ go run slicing/slicing.go
1 2 4 8 16 32 64 128 256 512
16 32 64
16 32 64 128 256 512
1 2 4 8 16 32 64 128 256 512
512 256 128 64 32 16 8 4 2 1
```

Esercizio 9 – Creazione dinamica di array/slice

Scrivete un programma che legga da tastiera un intero `n`, chieda all'utente di inserire `n` interi, legga gli `n` interi memorizzandoli in un array/slice di interi, ed infine stampi gli interi in ordine inverso rispetto a quello di inserimento.

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
>go run stampaRovescio.go
```
4
```
Inserisci i 4 interi:
```
1 4 0 2
```
Numeri stampati in ordine inverso:
2 0 4 1
```

ATTENZIONE

Dal momento che il numero `n` di interi da leggere è specificato a run-time, e memorizzato in una variabile, ad es. `dim`, l'istruzione di creazione dell'array:

```
var arr [dim]int
```

non è lecita perché `dim` non è una costante.

Per questo motivo, non potrete usare direttamente un array, ma dovrete dichiarare

contemporaneamente un array e la corrispondente slice con l'istruzione:

```
var slice1 []int = make([]int, dim)
```

Esercizio 10 - Frazioni

Se a_0 è un numero intero qualsiasi, e $a_1, a_2, \dots, a_n$ sono interi positivi, la notazione $[a_0, a_1, \dots, a_n]$ indica la ***frazione continua***, definita tramite l'espressione:

$$[a_0, a_1, \dots, a_n] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4} \\ \ddots \\ \ddots \\ a_{n-1} + \cfrac{1}{a_n}}}}$$

Ad esempio, $[-1, 5, 2, 4] = -1 + \frac{1}{5+\frac{1}{2+\frac{1}{4}}}$

E allo stesso modo $[-7, 3, 5, 7, 9] = -7 + 1/(3 + 1/(5 + 1/(7 + 1/9))) = -7 + \frac{1}{3+\frac{1}{5+\frac{1}{7+\frac{1}{9}}}}$

Ovviamente, $[a_0, a_1, \dots, a_n]$ è un numero razionale.

Scrivete un programma che legga da tastiera un intero `n`, chieda all'utente di inserire `n` interi $a_0, a_1, \dots, a_n$, legga gli `n` interi memorizzandoli in un array/slice di interi, stampi in output il valore di $[a_0, a_1, \dots, a_n]$.

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
>go run frazioni.go
 Inserisci un numero:
4
 Inserisci i 4 numeri interi:
-1
5
```

```
2
4
Frazione continua: -0.8163265306122449
```

Esercizio 11 - Passaggio di array e slice a funzioni

Note introduttive:

a) Gli array sono sempre passati per valore se non specificato altrimenti.

b) Le slice sono sempre passate per indirizzo.

Alla luce delle note introduttive riportate, analizziamo insieme il funzionamento del seguente programma.

```
func main() {

  var a = [6]int{1, 2, 3, 4, 5, 6}
  var b []int

  stampaArray(a)
  modificaArray(a)
  stampaArray(a)

  fmt.Println("==============")

  b = a[:]
  stampaSlice(b)
  modificaSlice(b)
  stampaSlice(b)
  stampaArray(a)
}

func modificaArray(a [6]int) {
  a[0] = 10
}

func modificaSlice(a []int) {
  a[0] = 10
}

func stampaArray(a [6]int) {
  for _, v := range a {
```

```
      fmt.Print(v, " ")
   }
   fmt.Println()
}

func stampaSlice(a []int) {
   for _, v := range a {
      fmt.Print(v, " ")
   }
   fmt.Println()
}
```

Esercizio 12– Fibonacci (rivisitato)

a) A partire dalla soluzione dell'esercizio 2, scrivere un programma che calcoli i primi `n` numeri di Fibonacci e li stampi a video, dove `n` è un numero intero specificato in input dall'utente. In particolare, strutturate il codice in modo tale che la funzione `main` chiami una funzione che ha come paramento `n` e restituisca una slice (`sl`) in cui sono memorizzati i numeri di Fibonacci fino all'`n`-esimo.

b) Si definisca poi un'ulteriore funzione che accetta come argomenti una slice di interi (`sl1`) ed un valore booleano. La funzione restituisce una slice contenente tutti gli interi pari presenti in `sl1` (nello stesso ordine relativo in cui sono memorizzati in `sl1`) se il valore booleano è uguale a `true`, una slice contenente tutti gli interi dispari presenti in `sl1` (nello stesso ordine relativo in cui sono memorizzati in `sl1`) altrimenti. Si utilizzi la funzione appena definita per stampare a video tutti i numeri dispari presenti tra i primi `n` numeri di Fibonacci (dal più piccolo al più grande).

c) Si consideri infine la sequenza di numeri `S` ottenuta concatenando la sottosequenza dei numeri dispari presenti tra i primi `n` numeri di Fibonacci (dal più piccolo al più grande) e la sottosequenza dei numeri pari presenti tra i primi `n` numeri di Fibonacci (dal più piccolo al più grande). Si stampino a video tutti i numeri in `S` associati ad un indice di sequenza `i` t.c. (`i % 7`)`=2` oppure (`i % 7`)`=4`. Il primo numero della sequenza ha indice `0`. Si risolva il punto c) senza utilizzare la funzioni `append`/`copy`.

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go run fibonacciRivisitato/fibonacciRivisitato.go
Inserisci n:
20
Primi 20 numeri di Fibonacci: [1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765]
Numeri dispari nei primi 20 numeri di Fibonacci: [1 1 3 5 13 21 55 89 233 377 987 1597 4181 6765]
Sottosequenza: [3 13 377 1597 34 610]
```

Esercizio 13 - Analisi di frammenti di codice - Array bidimensionali

Cosa stampa questo codice?

```go
func main() {
  a := [3][2]int{{1, 2}, {3, 4}, {5, 6}}

  a[1][1] = 10
  a[1][0] = a[1][1] + a[2][1]

  for i, riga := range a {
     fmt.Print(i,": ")
     for _, cella := range riga {
        fmt.Print(cella, " ")
     }
     fmt.Println()
  }
}
```

Esercizio 14 – Minimo in array bidimensionali

Scrivete un programma che inizializzi un array bidimensionale di reali di dimensione 5x5 (`arr`) in modo che l'elemento dell'array `arr[i][j]` sull'i-esima riga, `0 <= i <= 4`, e la j-esima colonna, `0 <= j <= 4`, contenga il valore `rand.Intn((i*j+1)*100)`.

In particolare, al fine di ottenere numeri casuali diversi ad ogni esecuzione, si ricordi di utilizzare

l'istruzione `rand.Seed(int64(time.Now().Nanosecond()))`

Si stampi la matrice quadrata corrispondente all'array bidimensionale.

Si trovi il valore minimo (`min`) presente nella matrice.

Si stampi l'intera riga dei valori in cui è contenuto `min`.

Si stampi l'intera colonna dei valori in cui è contenuto `min`.

Esempio di funzionamento

```
$ go run minimo/minimo.go
28 8 15 15 65
47 171 197 300 271
44 13 179 609 267
94 358 65 243 532
98 419 15 71 1064
Minimo: 8
Riga minimo:
28 8 15 15 65
Colonna minimo:
8
171
13
358
419
```

Esercizio 15 - Creazione dinamica di array/slice bidimensionali – La tavola pitagorica

Scrivere un programma che:

- memorizza in un array bidimensionale una tavola pitagorica `n x n`, dove `n` è un numero intero specificato in input dall'utente;
- stampa la tavola pitagorica definita.

SUGGERIMENTO

Utilizzate un array/slice bidimensionale. Per farlo, ispiratevi al seguente programma che crea dinamicamente un array bidimensionale (`matrix`) con `m` righe e `m` colonne.

```
package main
import "fmt"
func main() {
	m := 3
	matrix := make([][]int, m)
	/* matrix è un array/slice bi-dimensionale con lunghezza/capacità pari a  m (righe) */
	for i := 0; i < m; i++ {
		matrix[i] = make([]int, m)
		/* matrix[i] è un array/slice con lunghezza/capacità pari a m (colonne) */
	}
	for i := 0; i < len(matrix); i++ {
		for j := 0; j < len(matrix[i]); j++ {
			fmt.Print(matrix[i][j], " ")
		}
		fmt.Println()
	}
}
```

L'output del programma è il seguente:

```
0 0 0
0 0 0
0 0 0
```

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go run tavolaPitagorica/tavola.go
Inserisci la dimensione n:
```

4

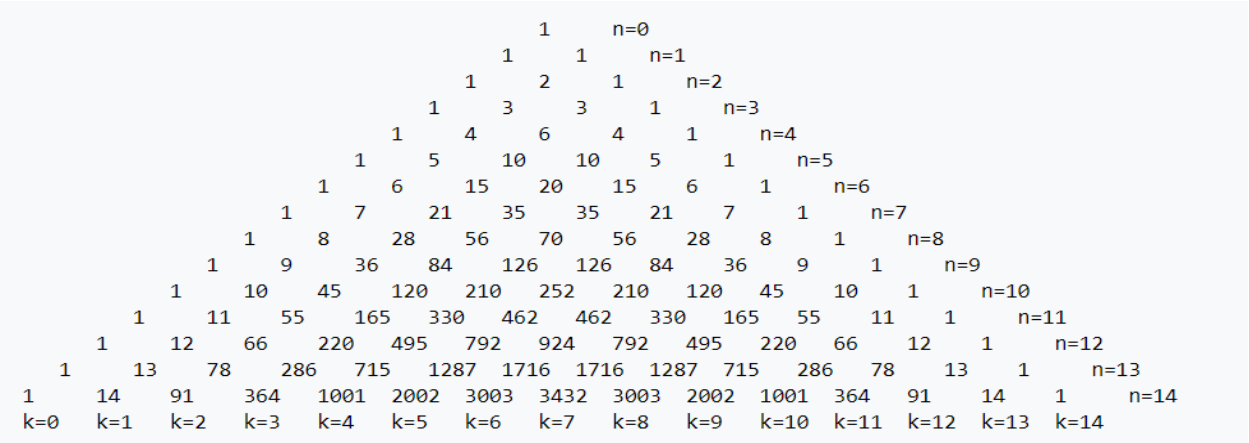
```
Tavola pitagorica 4 x 4
```

14

```
      1     2     3     4
1     1     2     3     4
2     2     4     6     8
3     3     6     9    12
4     4     8    12    16
```

Esercizio 16 - Il triangolo di Tartaglia

Il triangolo di Tartaglia è un sistema per calcolare i coefficienti binomiali $\binom{n}{k} = \frac{n!}{k!(n-k)!}$



Scrivete un programma che scriva un triangolo di Tartaglia con un numero di righe $\mathbf{h} > \mathbf{0}$ specificato dall'utente.

Numerando le righe con l'indice ***n* = 0,…,*h-1*** (come mostrato sopra), e indicizzando gli elementi della generica riga *n*-esima riga con l'indice ***k = 0,…,n*** ($e_{n,k}$ **è l'elemento alla riga n e in posizione k)** potete notare che:

- la generica riga *n*-esima contiene ***n+1*** numeri;
- il primo e l'ultimo elemento di ogni riga ($e_{n,0}$ e $e_{n,n}$), sono pari a 1.
- ogni elemento $e_{n,k}$ nella riga *n*-esima, $n \geq 0$, e in posizione ***k***-esima, $k \geq 1$ del triangolo è pari a $e_{n,k} = e_{n-1,k-1} + e_{n-1,k}$, ovvero alla somma degli elementi nella riga precedente che sono in posizione ***k-1*** e ***k***.

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go run tartaglia/tartaglia.go
Inserisci un valore per h:
5
     1
    1 1
   1 2 1
  1 3 3 1
1 4 6 4 1
```

SUGGERIMENTO

Utilizzate un array/slice bidimensionale. Per farlo, ispiratevi al seguente programma che crea dinamicamente un array bidimensionale (`matrix`) con `m` righe e, relativamente alla riga `i`-esima (con `0 <= i <= m-1`), `i+1` colonne.

```go
package main
import "fmt"
func main() {
	m := 4
	/* m è il numero righe */
	m_dispari := (m % 2) == 1 /* info utilizzata per formattare l'output */
	matrix := make([][]int, m)
	/* matrix è un array/slice bi-dimensionale con lunghezza/capacità pari a  m (righe) */
	for i := 0; i < m; i++ {
		n := i + 1
		/* n è il numero di colonne della riga i-esima, con 0 <= i <= m-1 */
		matrix[i] = make([]int, n)
		/* matrix[i] è un array/slice con lunghezza/capacità pari a n (colonne) */
		for j := 0; j < n; j++ {
			matrix[i][j] = '*'
		}
	}
	for i := 0; i < len(matrix); i++ {
		for k := 0; k < (len(matrix) - len(matrix[i])) / 2; k++ {
			fmt.Print("  ")
		}
		if ((i % 2 == 1) && m_dispari) || ((i % 2 == 0) && !m_dispari){
			fmt.Print(" ")
		}
```

```
                for j := 0; j < len(matrix[i]); j++ {
                        fmt.Print(string(matrix[i][j]), " ")
                }
                fmt.Println()
        }
}
```

L’output del programma è il seguente:

```
    *
   * *
  * * *
* * * *
```

Appendice - Esempio A – Argomenti da linea di comando

Create un file dal nome `inputLineaComando.go` e scriveteci il seguente codice sorgente:

```
package main

import (
  "fmt"
  "os"
)

func main() {
  fmt.Println(os.Args)
  for i, a := range os.Args {
    fmt.Printf("%d-esimo argomento da linea di comando: %[2]v %[2]T \n", i, a)
  }
}
```

Notate che la scrittura `[2]` dopo il simbolo di `%` dice alla funzione `Printf` che va usato il secondo tra gli argomenti da stampare (ovvero `a`), mentre il codice di formattazione `%T` stampa il tipo dell'argomento a cui il codice è associato (in questo caso la variabile `a`).

Il programma innanzitutto stampa la slice `os.Args` (tramite l'istruzione `fmt.Println(os.Args))`, poi con un ciclo `for...range...` stampa ogni valore di `os.Args` seguito dal suo tipo.

Dopo aver formattato il file contenente il programma:

```
>go fmt inputLineaComando.go
```

eseguiamolo dando in input da linea di comando 1 valore intero, 1 stringa, 1 valore float, 1 valore booleano, 1 valore intero positivo:

```
>go run inputLineaComando.go -3 Pippo 15.46734 true 7
```

Ecco cosa succede:

```
0-esimo argomento da linea di comando: /tmp/go-build495812646/b001/exe/
inputLineaComando - tipo: string
1-esimo argomento da linea di comando: -3 - tipo: string
2-esimo argomento da linea di comando: Pippo - tipo: string
3-esimo argomento da linea di comando: 15.46734 - tipo: string
4-esimo argomento da linea di comando: true - tipo: string
5-esimo argomento da linea di comando: 7 - tipo: string
```

`os.Args` è una slice contenente stringhe che riportano: il nome del programma (con estensione `.exe` se eseguito sotto Windows), tutti gli input forniti da linea di comando.

IN PRATICA: Passare argomenti da linea di comando significa lanciare il programma con la solita istruzione e poi far seguire tutti gli input necessari `(arg1, arg2, arg3...)` passandoli come argomenti del comando:

```
>go run mioProgramma.go arg1 arg2 arg3...
```

Il programma trova tutti gli input salvati sotto forma di stringhe nella slice `os.Args[1:]` (NOTATE che `os.Args[0]` non va considerato poiché contiene il nome del programma stesso!!).

ATTENZIONE: Essendo in formato stringa gli input vanno eventualmente convertiti in valori numerici con le funzioni del `package strconv` (si veda l'Esempio 2).

Appendice - Esempio 2 - Conversione di input da linea di comando

Scrivete un programma che riceve in input da linea di comando valori di tipo diverso (valori interi, reali, booleani, stringhe). Per ogni input letto, il programma usa le funzioni `ParseBool(...)`, `ParseFloat(...)`, `ParseInt(...)`, `ParseUint(...)` del `package strconv` per convertire gli input da stringa in variabili di tipo appropriato e quindi stamparne il valore o eventualmente l'errore se la conversione non è stata possibile.

Esempio di funzionamento

Nel seguente esempio, lancio il programma il cui codice è salvato nel file `conversioniConStrconv.go`.

Gli input da linea di comando sono:

```
true 23 5.6 -367547141794764786
```

ovvero:

- un valore booleano;
- un valore intero;
- un valore reale (che può essere salvato sia in una variabile di tipo float32, sia in una di tipo float64);
- un valore intero negativo (che può essere salvato in una variabile di tipo int64, float32, float64 ma non in variabili di tipo int32 o uint - per problemi di overflow).

Dal momento che l'output è piuttosto lungo, per poterlo analizzare meglio uso la **redirezione di output su file** (mettendo in coda al comando il simbolo maggiore ">" seguito dal nome del file out.txt). In questo modo, al termine dell'esecuzione del programma trovo nella directory del file eseguibile un file out.txt che contiene l'output del programma.

```
>go run conversioniConStrconv.go true 23 5.6 -367547141794764786 > out.txt
```

Ecco il contenuto del file `out.txt`:

```
*******
input 1-esimo in formato stringa = true
--------------
CONVERTO IN VALORE BOOLEANO
valore booleano =  true
---------------
CONVERTO IN VALORI INT
NON e' int (errore: strconv.ParseInt: parsing "true": invalid syntax)
NON e' int32 (errore: strconv.ParseInt: parsing "true": invalid syntax)
NON e' int64 (errore: strconv.ParseInt: parsing "true": invalid syntax)
---------------
CONVERTO IN VALORI UINT
NON e' uint (errore: strconv.ParseUint: parsing "true": invalid syntax)
NON e' uint8 (errore: strconv.ParseUint: parsing "true": invalid syntax)
NON e' uint16 (errore: strconv.ParseUint: parsing "true": invalid syntax)
NON e' uint32 (errore: strconv.ParseUint: parsing "true": invalid syntax)
NON e' uint64 (errore: strconv.ParseUint: parsing "true": invalid syntax)
---------------
CONVERTO IN VALORI FLOAT
NON e' float32 (errore: strconv.ParseFloat: parsing "true": invalid syntax)
NON e' float64 (errore: strconv.ParseFloat: parsing "true": invalid syntax)
*******
input 2-esimo in formato stringa = 23
---------------
CONVERTO IN VALORE BOOLEANO
NON e' booleano (errore: strconv.ParseBool: parsing "23": invalid syntax)
---------------
CONVERTO IN VALORI INT
valore int =  23
valore int32 =  23
valore int64 =  23
---------------
CONVERTO IN VALORI UINT
valore uint =  23
valore uint8 =  23
valore uint16 =  23
valore uint32 =  23
```

```
valore uint64 =  23
---------------
CONVERTO IN VALORI FLOAT
valore float32 = 23
valore float64 = 23
*******
input 3-esimo in formato stringa = 5.6
---------------
CONVERTO IN VALORE BOOLEANO
NON e' booleano (errore: strconv.ParseBool: parsing "5.6": invalid syntax)
---------------
CONVERTO IN VALORI INT
NON e' int (errore: strconv.ParseInt: parsing "5.6": invalid syntax)
NON e' int32 (errore: strconv.ParseInt: parsing "5.6": invalid syntax)
NON e' int64 (errore: strconv.ParseInt: parsing "5.6": invalid syntax)
---------------
CONVERTO IN VALORI UINT
NON e' uint (errore: strconv.ParseUint: parsing "5.6": invalid syntax)
NON e' uint8 (errore: strconv.ParseUint: parsing "5.6": invalid syntax)
NON e' uint16 (errore: strconv.ParseUint: parsing "5.6": invalid syntax)
NON e' uint32 (errore: strconv.ParseUint: parsing "5.6": invalid syntax)
NON e' uint64 (errore: strconv.ParseUint: parsing "5.6": invalid syntax)
---------------
CONVERTO IN VALORI FLOAT
valore float32 = 5.599999904632568
valore float64 = 5.6
*******
input 4-esimo in formato stringa = -367547141794764786
---------------
CONVERTO IN VALORE BOOLEANO
NON e' booleano (errore: strconv.ParseBool: parsing "-367547141794764786": invalid syntax)
---------------
CONVERTO IN VALORI INT
valore int =  -367547141794764786
NON e' int32 (errore: strconv.ParseInt: parsing "-367547141794764786": value out of range)
valore int64 =  -367547141794764786
---------------
CONVERTO IN VALORI UINT
NON e' uint (errore: strconv.ParseUint: parsing "5.6": invalid syntax)
NON e' uint8 (errore: strconv.ParseUint: parsing "-367547141794764786": invalid syntax)
NON e' uint16 (errore: strconv.ParseUint: parsing "-367547141794764786": invalid syntax)
NON e' uint32 (errore: strconv.ParseUint: parsing "-367547141794764786": invalid syntax)
NON e' uint64 (errore: strconv.ParseUint: parsing "-367547141794764786": invalid syntax)
---------------
CONVERTO IN VALORI FLOAT
valore float32 = -3.6754715211464704e+17
```

```
valore float64 = -3.675471417947648e+17
```

Per capire come si possono ottenere le conversioni sopra elencate, controllate la documentazione delle funzioni `ParseBool, ParseFloat, ParseInt` e `ParseUint` del package `strconv`:

func ParseBool(str string) (bool, error)
func ParseFloat(s string, bitSize int) (float64, error)
func ParseInt(s string, base int, bitSize int) (i int64, err error)
func ParseUint(s string, base int, bitSize int) (uint64, error)

Tenete comunque presente che ognuna di esse restituisce il valore convertito (se una conversione è possibile!!) e un valore di tipo **`error`**. Se la conversione è andata a buon fine**,** il valore della variabile di tipo **`error`** è pari a `nil` (il valore zero del tipo **`error`**); se la conversione non è andata a buon fine**,** questo valore può essere stampato e riporta l'errore. Ad esempio, il seguente programma cerca di convertire un input da linea di comando in un valore booleano. Se ci riesce lo notifica all'utente e stampa il valore booleano; altrimenti stampa l'errore ottenuto.

```go
package main
import "fmt"
import "strconv"
import "os"
func main() {
        var err error
        var bb bool
        bb, err = strconv.ParseBool(os.Args[1])
        if err==nil{
                fmt.Printf("VALORE BOOLEANO %v\n", bb)
        }else{
                fmt.Printf("NON E' UN VALORE BOOLEANO %v\n", err)
        }
}
```

Se non riuscite a risolvere l'esercizio, guardate il codice riportato nella pagina seguente.

Codice del programma conversioniConStrconv.go

```
package main
import "fmt"
import "strconv"
import "os"
func main() {

   for i, v := range os.Args {
         if i>0{
                fmt.Print("*******\ninput ",i,"-esimo in formato stringa = ",v,"\n")

                fmt.Println("---------------\nCONVERTO IN VALORE BOOLEANO")
                bb, err := strconv.ParseBool(v)
                if err == nil {
                        fmt.Println("valore booleano =", bb)
                } else {
                        fmt.Println("NON e' booleano (errore:",err,")")
                }

                fmt.Println("---------------\nCONVERTO IN VALORI INT ")
                ii, err := strconv.ParseInt(v, 10, 0)
                if err == nil {
                        fmt.Println("valore int =", int(ii))
                } else {
                        fmt.Println("NON e' int (errore:",err,")")
                }
                ii32, err := strconv.ParseInt(v, 10, 32)
                if err == nil {
                        fmt.Println("valore int32 =", int32(ii32))
                } else {
                        fmt.Println("NON e' int32 (errore:",err,")")
                }
                ii64, err := strconv.ParseInt(v, 10, 64)
                if err == nil {
                        fmt.Println("valore int64 =", int64(ii64))
                } else {
                        fmt.Println("NON e' int64 (errore:",err,")")
                }

                fmt.Println("---------------\nCONVERTO IN VALORI UINT ")
                uii, err := strconv.ParseUint(v, 10, 0)
                if err == nil {
                        fmt.Println("valore uint =", uint(uii))
                } else {
                        fmt.Println("NON e' uint (errore:",err,")")
                }
                uii8, err := strconv.ParseUint(v, 10, 8)
                if err == nil {
                        fmt.Println("valore uint8 =", uint8(uii8))
                } else {
                        fmt.Println("NON e' uint8 (errore:",err,")")
                }
                uii16, err := strconv.ParseUint(v, 10, 16)
                if err == nil {
                        fmt.Println("valore uint16 =", uint16(uii16))
                } else {
                        fmt.Println("NON e' uint16 (errore:",err,")")
                }
                uii32, err := strconv.ParseUint(v, 10, 32)
                if err == nil {
                        fmt.Println("valore uint32 =", uint32(uii32))
                } else {
                        fmt.Println("NON e' uint32 (errore:",err,")")
```

```
                }
                uii64, err := strconv.ParseUint(v, 10, 64)
                if err == nil {
                        fmt.Println("valore uint64 =", uint64(uii64))
                } else {
                        fmt.Println("NON e' uint64 (errore:",err,")")
                }

                fmt.Println("---------------\nCONVERTO IN VALORI FLOAT ")
                ff32, err := strconv.ParseFloat(v, 32)
                if err == nil {
                        fmt.Println("valore float32 =", float32(ff32))
                } else {
                        fmt.Println("NON e' float32 (errore:",err,")")
                }
                ff64, err := strconv.ParseFloat(v, 64)
                if err == nil {
                        fmt.Println("valore float64 =", float64(ff64))
                } else {
                        fmt.Println("NON e' float64 (errore:",err,")")
                }

        }
    }
}
```