

Laboratorio 8 - Array, Slice, Riga di comando (2) ...e Test

Tipi di errori - Errori formali

Il programma viola alcune regole del linguaggio. Tipicamente segnalati dal compilatore o dall'interprete.

Esempio:

```
var a itn
fmt.Println(a)
```

Tipi di errori - Errori logici

Il programma è scritto utilizzando il linguaggio in modo corretto ma non produce l'output atteso.

Esempio:

```
for i = 0; i < 10; i-- {
    fmt.Println(i)
}
```

Debugging

Attività di ricerca e correzione degli errori (**bug**) presenti all'interno del programma.

Come cercare i bug?

- utilizzare messaggi di log
- utilizzo di debugger
- **testing**

Testing

Attività che prevede di eseguire il programma (o parti del programma) molteplici volte, sottoponendo ad ogni esecuzione input diversi. Gli input sono pensati appositamente per cercare gli errori.

L'attività di testing **non** può garantire l'assenza di errori.

Livelli di testing

- **unit testing**: test di singole entità del programma che possono essere trattate in modo indipendente (i.e.: funzioni).
- **integration testing**: test sulla cooperazione tra entità diverse ma che interagiscono tra di loro.

- **system testing:** test del funzionamento dell'intero programma all'interno di un ambiente il più simile possibile all'ambiente designato per l'esecuzione del programma stesso.

Tipi di testing

- **funzionali:** verificano che il programma risponda in modo corretto agli input fornendo l'output atteso.
- **non funzionali:** verificano che il programma rispetti delle proprietà collaterali (e.g.: tempi di calcolo, consumo di memoria, ...)

Testing - esempio

Problema

Scrivere un programma che legga da **riga di comando** un numero intero positivo e stampi a video se tale numero è primo. Se il valore inserito non è un numero intero o è un numero intero non positivo, il programma stampa un messaggio d'errore.

Unit di test

Possibili unit di test:

- il valore inserito è un numero intero positivo primo
- il valore inserito è un numero intero positivo non primo
- il valore inserito è 1 (1 non è un numero primo)
- il valore inserito è un numero intero nullo
- il valore inserito è un numero intero negativo
- il valore inserito è un numero reale
- il valore inserito non è un numero

Test in Go

Il package `testing` mette a disposizione degli strumenti per il testing.

```
import "testing"
```

Lo strumento `go test` esegue tutte le funzioni aventi prefisso `Test` presenti nei file con suffisso `_test.go` nella cartella corrente.

```
func TestFunzione(t *testing.T) {
    // codice della unit di test
}
```

Ogni funzione con prefisso `Test` è una unit di test. Una unit di test fallisce se chiama una funzione `t.Error()` o `t.Errorf()`.

```
$ go test
--- FAIL: ...
```

Se nessuna unit fallisce, la fase di testing è superata.

```
$ go test
PASS
```

Esempio Test Pari e dispari

Scrivere un programma che legga da **standard input** un numero intero **n** e stampi a video se **n** è pari o dispari.

Oltre alla funzione `main()`, devono essere definite le seguenti funzioni: * una funzione `Pari(n int) bool` che riceve in input un valore intero **n** e restituisce `true` se **n** è pari, `false` altrimenti; * una funzione `Dispari(n int) bool` che riceve in input un valore intero **n** e restituisce `true` se **n** è dispari, `false` altrimenti.

Verificare il corretto funzionamento delle funzioni utilizzando le unit di test allegate.

Esempio d'esecuzione:

```
$ go run pari_dispari.go
Inserisci un numero: 5
Il numero 5 è dispari
```

```
$ go run pari_dispari.go
Inserisci un numero: 4
Il numero 4 è pari
```

Esercizi Pratici

1) Carte da gioco

Scrivere un programma per la rappresentazione e gestione di carte da gioco “francesi”. A tale scopo definire i seguenti tipi:

- Definire un tipo `CartaDaGioco`, che rappresenti delle carte da gioco “francesi”; Ogni carta avrà due attributi: `seme`, una stringa tra `["quadri", "picche", "cuori", "fiori"]`, e `simbolo`, una stringa fra `["asso", "due", "tre", "quattro", "cinque", "sei", "sette", "fante", "donna", "re"]`.
- Definire un tipo `Mazzo` che rappresenti mazzi di (inizialmente 40) carte. Ogni mazzo avrà un attributo `carte` contenente la collezione di carte da gioco e un attributo `nCarte` intero contenente il numero corrente di carte da gioco nel mazzo

Dati i tipi di cui sopra, oltre alla funzione `main()` definire le seguenti funzioni:

- La funzione `CreaCarta(seme, simbolo string) CartaDaGioco` che prende in ingresso due stringhe rappresentanti il seme e il simbolo e restituisce un tipo `CartaDaGioco`
- La funzione `CreaMazzo() Mazzo` che crea e restituisce un mazzo di 40 carte da gioco
- La funzione `Mischia(m Mazzo)` che riceve in ingresso un mazzo di carte e lo mischia

Suggerimento: E' possibile mischiare il contenuto di una slice nel seguente modo (la soluzione utilizza il pacchetto `math/rand`):

```
for i := range slice {
    j := rand.Intn(i + 1)
    slice[i], slice[j] = slice[j], slice[i]
}
```

- La funzione `Preleva(m Mazzo) (CartaDaGioco, Mazzo, error)` che simula il prelievo di una carta dalla cima di un mazzo; restituisce la tripla `(c,m,nil)`, dove `c` è la carta in cima al mazzo, `m` è un mazzo senza `c`, se il mazzo non è vuoto; restituisce `(c,m,err)`, dove `c` ed `m` sono arbitrari, ed `err` è un valore diverso da `nil`, altrimenti.
- La funzione `Riponi(m Mazzo, c CartaDaGioco) (Mazzo, error)` che simula la posa di una carta in cima ad un mazzo restituisce la coppia `(m,nil)`, dove `m` è un mazzo con l'aggiunta di `c`, se il mazzo non è pieno; restituisce `(m,err)`, con `m` arbitrario ed `err` un valore diverso da `nil`, altrimenti.

Testare le funzioni all'interno del `main()`.

2) Sottostringhe Palindrome

Scrivere un programma che legga da **riga di comando** una stringa composta di caratteri unicode e stampi a schermo tutte le sottostringhe palindrome (che siano uguali lette da destra e da sinistra) della stringa.

Esempio d'esecuzione

```
$ go run esercizio_1.go sottotono
[otto tt tot oto ono]
```

```
$ go run esercizio_1.go banana
[ana anana nan ana]
```

```
$ go run esercizio_1.go ereggere
[ere ereggere regger egge gg ere]
```

```
$ go run esercizio_1.go Vaðlaheiðarvegavinnuverkfærageymsluskúrslyklakippuhringurinn
```

```
[nn pp nn]
```

```
$ go run esercizio_1.go donaudampfschiffahrtselektrizitätenhauptbetriebswerkbauunterbeamt  
[ff fff ff ele izi tät uu ese ll]
```

3) Parentesi bilanciate

Uno dei compiti più importanti di un compilatore è quello di controllare se le parentesi sono ben bilanciate. Ad ogni parentesi aperta deve corrispondere una parentesi chiusa, e le coppie di parentesi devono essere innestate propriamente.

Un esempio di parentesi tonde ben bilanciate è il seguente: `(())()`

Un esempio di parentesi tonde *non* ben bilanciate è il seguente: `()))()`

Notare che in quest'ultimo esempio, anche se ad ogni parentesi aperta corrisponde una parentesi chiusa, le coppie di parentesi non sono propriamente innestate (viene chiusa una parentesi di troppo ed una parentesi rimane aperta senza mai essere chiusa).

Parte 1

Leggere da **riga di comando** una stringa. Se la stringa contiene caratteri diversi da parentesi tonda aperta `(` e parentesi tonda chiusa `)` (che ricordiamo sono caratteri ASCII) terminare l'esecuzione del programma, altrimenti stampare **bilanciata** se la stringa ha parentesi bilanciate o **non bilanciata** altrimenti. A tal fine implementare e usare una funzione `isBalanced(sequence string) bool` che riceve in input una stringa `sequence` composta di sole parentesi (aperte e chiuse) e che restituisce il valore booleano `true` se la stringa `sequence` è ben bilanciata, o `false` altrimenti.

- Nota: poichè le parentesi sono caratteri speciali nell'esecuzione dei programmi da riga di comando, per inserirle bisogna o fare l'escape di ciascuna oppure inserire la stringa tra virgolette.

Parte 2

Stampare tutte le sottostringhe ben bilanciate.

Esempio d'esecuzione:

```
$ go run esercizio_2.go "(( ))()"
bilanciata
---
Sottosequenze bilanciate:
1) (( ))
2) (( ))()
3) ()
4) ()
```

```
$ go run esercizio_2.go "()"
non bilanciata
---
Sottosequenze bilanciate:
1) ()
```

```
$ go run esercizio_2.go "()()"
bilanciata
---
Sottosequenze bilanciate:
1) (()())
2) ()
3) ()()
4) ()
```

```
$ go run esercizio_2.go "())("
non bilanciata
---
Sottosequenze bilanciate:
1) ()
```

```
go run esercizio_2.go "))(("
non bilanciata
---
Sottosequenze bilanciate:
Nessuna.
```

```
go run esercizio_2.go []
L'input fornito non aveva esclusivamente parentesi tonde.
```

4) Sottosequenze crescenti

Scrivere un programma che: * legga da **riga di comando** una stringa **s** costituita da cifre decimali; * stampi a schermo tutte le sottosequenze (anche ripetute) della stringa **s** nelle quali le cifre sono in ordine crescente (si considerino solamente sottosequenze di almeno 2 cifre).

Se la stringa **s** letta da **riga di comando** non è costituita solamente da cifre decimali, il programma restituisce una slice vuota.

Oltre alla funzione **main()**, deve essere definita ed utilizzata almeno la funzione **Sottostringhe(s string) []string**, che riceve in input un valore **string** nel parametro **s**, e restituisce un valore **[]string** che contiene tutte le sottosequenze desiderate

Utilizzare la unit di test fornita per verificare la correttezza della soluzione.

Esempio d'esecuzione:

```
$ go run esercizio_3.go 123456
[12 123 1234 12345 123456 23 234 2345 23456 34 345 3456 45 456 56]

$ go run esercizio_3.go 654321
[]

$ go run esercizio_3.go 123121212
[12 123 23 12 12 12]

$ go run esercizio_3.go acc23
[]

$ go run esercizio_3.go 01010101
[01 01 01 01]

$ go mod init sottostringhe
...

$ go test
PASS
ok      sottostringhe    0.002s
```