Lab10 – Funzioni ricorsive/Strutture ricorsive

Esercizio 1 - Analisi di codice

```
func main() {
  conteggio(10)
}

func conteggio(n uint) {
  if n == 0 {
     fmt.Println("Partenza!")
  } else {
     fmt.Println(n)
     conteggio(n - 1)
  }
}
```

Esercizio 2 - Analisi di codice

```
func main() {
 conteggio(10)
 fmt.Println("Partenza!")

 fmt.Println()

 conteggioAlternativo(10)
 fmt.Println("Partenza!")
}

func conteggio(n uint) {
 if n != 0 {
  fmt.Println(n)
  conteggio(n - 1)
 }
}
```

```go
func conteggioAlternativo(n uint) {
 if n != 0 {
  conteggioAlternativo(n - 1)
  fmt.Println(n)
 }
}
```

Esercizio 3 - Analisi di codice

```go
func main() {
  fmt.Println("La somma dei numeri da 0 a 10 è", somma(10))
}
func somma(n uint) uint {
  if n == 0 {
     return 0
  } else {
     return n + somma(n-1)
  }
}
```

Esercizio 4 - Somma ricorsiva

Scrivere un programma che legga una sequenza di numeri interi da riga di comando e ne effettui la somma utilizzando una funzione ricorsiva.

Esempio di funzionamento

```
$ go run somma/somma.go 1 2 3 4 5
La somma dei valori inseriti è 15
```

Esercizio 5 - Divisione intera

Creare una funzione ricorsiva `divisione(a, b int)` che effettui la divisione tra due numeri interi

letti da riga di comando e restituisca quoziente e resto.

Esempio di funzionamento

```
$ go run divisioneIntera/divisione.go 4 3
4 diviso 3 fa 1 con resto 1

 $ go run divisioneIntera/divisione.go 10 4
10 diviso 4 fa 2 con resto 2
```

Esercizio 6 - Frazione continua

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:

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

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.

Scrivere un programma che chieda all'utente di inserire una sequenza di numeri interi (uno per riga); la sequenza non ha dimensione definita. La fase di inserimento dei numeri termina alla lettura di una riga vuota. Una volta terminata la fase di inserimento, il programma stampa in output il valore di $[a_0, a_1, \dots, a_n]$ calcolato con una funzione ricorsiva**.**

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

```
$go run frazioni.go

Inserisci un numero:
-1
Inserisci un numero:
5
Inserisci un numero:
2
Inserisci un numero:
4
Inserisci un numero:

Inserimento terminato.
Frazione continua: -0.8163265306122449
```

Esercizio 7 - Moltiplicazione vettori

Scrivere un programma che legga da riga di comando un intero `n`, e legga successivamente due sequenze A e B di n numeri interi. Il programma deve calcolare in modo ricorsivo

a1 * b1 + a2 * b2 + a3 * b3 + … + an * bn

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

```
$ go run moltiplicazioneVettori/moltiplicazioneVettori.go 5
A: 1 2 3 4 5
B: 3 4 5 6 1
A = [1 2 3 4 5]
B = [3 4 5 6 1]
A*B = 55
```

Esercizio 8 - Stampa ricorsiva

Scrivere un programma che legga un testo su più righe e stampi il testo, riga dopo riga, utilizzando una

funzione ricorsiva.

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

```
$ go run stampaRicorsiva/stampa.go
Inserisci del testo:
Ciao, come stai?
Io bene, tu?
Io pure, grazie.
Testo inserito:
Riga: Ciao, come stai?
Riga: Io bene, tu?
Riga: Io pure, grazie.
```

Esercizio 9 - Stampa ricorsiva invertita

Scrivere un programma che legga un testo su più righe e lo stampi invertendo l'ordine delle righe lette, utilizzando una funzione ricorsiva.

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

```
$ go run stampaInvertita/stampa.go
Inserisci del testo:
Ciao, come stai?
Io bene, tu?
Io pure, grazie.
Testo invertito:
Io pure, grazie.
Io bene, tu?
Ciao, come stai?
```

Esercizio 10 - Testo invertito

Scrivere un programma che legga un testo su più righe e stampi il testo al contrario, dall'ultimo carattere letto fino al primo, utilizzando una o più funzioni ricorsive.

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

```
$ go run testoInvertito/testoInvertito.go
Ciao, come stai?
Io bene, tu?
Io pure, grazie.
Testo invertito:
.eizarg ,erup oI
?ut ,eneb oI
?iats emoc ,oaiC
```

Esercizio 11 - Analisi di codice

```go
type Persona struct {
  nome, cognome string
  genitore1, genitore2 *Persona
}

func main() {

  padre := new(Persona)
  padre.nome = "Jerry"
  (*padre).cognome = "Smith"
  padre.genitore1 = nil // non necessario
  padre.genitore2 = nil // non necessario

  madre := &Persona{nome: "Beth", cognome: "Smith"}

  figlio := Persona{"Morty", "Smith", padre, madre }

  fmt.Println(figlio)
  stampaPersona(figlio)
}

func stampaPersona(p Persona) {
  fmt.Printf("%s %s", p.nome, p.cognome)
  if p.genitore1 != nil {
     fmt.Printf(" - Genitore: %s %s", p.genitore1.nome, p.genitore1.cognome)
  }
  if p.genitore2 != nil {
     fmt.Printf(" - Genitore: %s %s", p.genitore2.nome, p.genitore2.cognome)
  }
  fmt.Println()
}
```

Esercizio 12 - Lista concatenata semplice

Con riferimento al codice visto a lezione per la gestione di liste concatenate semplici di stringhe, implementare un programma per la gestione di liste concatenate semplici di interi. In particolare:

1. Definire la struttura ricorsiva `Nodo` per la gestione di una lista concatenata semplici di interi.
2. Definire una funzione `Vuota(primo *Nodo) bool` che restituisca TRUE se è vuota la lista il cui primo nodo è `*primo` e FALSE altrimenti.
3. Definire una funzione `Aggiungi(primo *Nodo, valore int) *Nodo` che aggiunga un valore alla lista il cui primo nodo è `*primo`, mantenendo ordinati in senso non decrescente i valori memorizzati all'interno della lista.
4. Definire una funzione `Stampa(primo *Nodo)` che stampi i valori memorizzati nella lista (il cui primo nodo è `*primo`) dal primo fino all'ultimo.
5. Definire una funzione `Lunghezza(primo *Nodo) int` che restituisca il numero di valori memorizzati nella lista il cui primo nodo è `*primo`.
6. Definire una funzione `Trova(primo *Nodo, valore int) int` che restituisca l'indice di posizione nella lista (il cui primo nodo è `*primo`) del nodo relativo a `valore` (-1 se `valore` non è presente nella lista o se la lista è vuota); l'indice di posizione del primo nodo della lista è uguale a 0; se `valore` compare più volte nella lista, la funzione restituisca l'indice di posizione del nodo relativo alla prima occorrenza di `valore` nella lista.
7. Definire una funzione `Elimina(primo *Nodo, posizione int) *Nodo` che elimina il nodo in posizione `posizione` (`0 <= posizione < Lunghezza(primo)`) nella lista il cui primo nodo è `*primo`. La funzione restituisce il puntatore al nuovo nodo che risulta essere, dopo l'eventuale eliminazione, il primo della lista. Il valore di `posizione` deve essere tale che `0 <= posizione < Lunghezza(primo)`, altrimenti l'esecuzione della funzione non ha effetto sulla lista.
8. Definire una funzione `EliminaValore(primo *Nodo, valore int) *Nodo` che elimina il nodo nella posizione corrispondente alla prima occorenza del valore `valore` nella lista il cui primo nodo è `*primo`. La funzione restituisce il puntatore al nuovo nodo che risulta essere, dopo l'evenutuale eliminazione, il primo della lista. Se `valore` non è presente nella lista, l'esecuzione della funzione non ha effetto sulla lista.

9. Definire la funzione `main()` come segue:

```
func main() {

  var lista *Nodo

  for _, v := range os.Args[1:] {
    if n, err := strconv.Atoi(v); err == nil {
      lista = Aggiungi(lista, n)
    }
  }

  fmt.Println("Lughezza della lista:", Lunghezza(lista))
  Stampa(lista)
  fmt.Println("Posizione del valore 5:", Trova(lista, 5))
  fmt.Println("Posizione del valore 4:", Trova(lista, 4))

  fmt.Println("Elimino nodo in posizione -1:")
  lista = Elimina(lista, -1)
  Stampa(lista)
  fmt.Println("Elimino nodo in posizione 9:")
  lista = Elimina(lista, 9)
  Stampa(lista)
  fmt.Println("Elimino nodo in posizione 2:")
  lista = Elimina(lista, 2)
  Stampa(lista)
  fmt.Println("Elimino nodo in posizione 3:")
  lista = Elimina(lista, 3)
  Stampa(lista)
  fmt.Println("Elimino nodo in posizione 0:")
  lista = Elimina(lista, 0)
  Stampa(lista)

  fmt.Println("Elimino valore 5:")
  lista = EliminaValore(lista, 5)
  Stampa(lista)
  fmt.Println("Elimino valore 6:")
  lista = EliminaValore(lista, 6)
  Stampa(lista)

  fmt.Println("Lughezza della lista:", Lunghezza(lista))
}
```

Esempio di funzionamento

```
$ go run lista.go 2 6 1 8 4
Lughezza della lista: 5
Valori nella lista: 1 2 4 6 8
Posizione del valore 5: -1
Posizione del valore 4: 2
Elimino nodo in posizione -1:
Valori nella lista: 1 2 4 6 8
Elimino nodo in posizione 9:
Valori nella lista: 1 2 4 6 8
Elimino nodo in posizione 2:
Valori nella lista: 1 2 6 8
Elimino nodo in posizione 3:
Valori nella lista: 1 2 6
Elimino nodo in posizione 0:
Valori nella lista: 2 6
```

```
Elimino valore 5:
Valori nella lista: 2 6
Elimino valore 6:
Valori nella lista: 2
Lughezza della lista: 1
```

Esercizio 13 - Coda

Una coda è una struttura dati dinamica simile alla lista concatenata semplice che permette di gestire l'inserimento e l'estrazione dei valori in modo tale che i primi valori inseriti nella coda siano anche i primi ad essere estratti (*First In First Out*). L'implementazione di una coda è del tutto simile a quella di una lista concatenata semplice.

Implementare un programma per la gestione di code di interi. In particolare:

1. Definire la struttura ricorsiva `Elemento` per la gestione di una coda.
2. Definire una funzione `CodaVuota(primo *Elemento) bool` che restituisca TRUE se è vuota la coda il cui primo elemento che è stato inserito è `*primo` e FALSE altrimenti.
3. Definire una funzione `AggiungiACoda(primo *Elemento, valore int) *Elemento` che aggiunga un valore in fondo alla coda il cui primo elemento che è stato inserito è `*primo`.
4. Definire una funzione `StampaCoda(primo *Elemento)` che stampi i valori memorizzati nella coda (il cui primo elemento che è stato inserito è `*primo`) dal primo valore inserito fino all'ultimo.
5. Una funzione `EstraiDaCoda(primo *Elemento) (int, *Elemento)` che estragga dalla coda il cui primo elemento che è stato inserito è `*primo`. La funzione restituisce il primo valore inserito ed il puntatore al nuovo elemento che risulta essere, dopo l'estrazione, il primo inserito nella coda.
6. Usare queste funzioni per inizializzare una coda contenente dei numeri interi letti da riga di comando, stampare il suo contenuto, estrarre tutti i valori fino allo svuotamento della coda, e infine stampare di nuovo il contenuto della coda.

Esempio di funzionamento

```
$ go run coda.go 10 12 ciao 3 4
Elementi della coda: 10 12 3 4
Estraggo dalla coda: 10
Elementi della coda: 12 3 4
Estraggo dalla coda: 12
Elementi della coda: 3 4
Estraggo dalla coda: 3
Elementi della coda: 4
Estraggo dalla coda: 4
Elementi della coda:
```

SUGGERIMENTO

Utilizzate il seguente main per richiamare le funzioni da implementare.

```
func main() {
  var coda *Elemento

  for _, v := range os.Args[1:] {
    if n, err := strconv.Atoi(v); err == nil {
      coda = AggiungiACoda(coda, n)
    }
  }

  StampaCoda(coda)

  for !CodaVuota(coda) {
    var n int
    n, coda = EstraiDaCoda(coda)
    fmt.Println("Estraggo dalla coda:", n)
    StampaCoda(coda)
  }

}
```

Esercizio 14 - Pila

Una pila è una struttura dati dinamica simile alla lista concatenata semplice che permette di gestire l'inserimento e l'estrazione dei valori in modo tale che gli ultimi valori inseriti nella pila siano i primi ad essere estratti (*Last In First Out*). L'implementazione di una pila è del tutto simile a quella di una lista concatenata semplice.

Con riferimento al codice visto a lezione per la gestione di liste concatenate semplici di stringhe, implementare un programma per la gestione di pile di interi. In particolare:

1. Definire la struttura ricorsiva `Elemento` per la gestione di una pila.
2. Definire una funzione `PilaVuota(ultimo *Elemento) bool` che restituisca TRUE se è vuota la pila il cui ultimo elemento che è stato inserito è `*ultimo` e FALSE altrimenti.
3. Definire una funzione `Push(ultimo *Elemento, valore int) *Elemento` che aggiunga un valore in cima alla pila il cui ultimo elemento che è stato inserito è `*ultimo`.
4. Definire una funzione `StampaPila(ultimo *Elemento)` che stampi i valori memorizzati nella pila (il cui ultimo elemento che è stato inserito è `*ultimo`) dall'ultimo valore inserito fino al primo.
5. Una funzione `Pop(ultimo *Elemento) (int, *Elemento)` che estragga dalla pila il cui ultimo elemento che è stato inserito è `*ultimo`. La funzione restituisce l'ultimo valore inserito ed il puntatore al nuovo elemento che risulta essere, dopo l'estrazione, l'ultimo inserito nella pila.
6. Usare queste funzioni per inizializzare una pila contenente dei numeri interi letti da riga di comando, stampare il suo contenuto, estrarre tutti i valori fino allo svuotamento della pila, e infine stampare di nuovo il contenuto della pila.

Esempio di funzionamento

```
$ go run pila.go 1 2 3
Elementi della pila: 3 2 1
Estraggo dalla pila: 3
Elementi della pila: 2 1
Estraggo dalla pila: 2
Elementi della pila: 1
Estraggo dalla pila: 1
Elementi della pila:
```

SUGGERIMENTO

Utilizzate il seguente main per richiamare le funzioni da implementare.

```
func main() {
  var pila *Elemento

  for _, v := range os.Args[1:] {
    if n, err := strconv.Atoi(v); err == nil {
      pila = Push(pila, n)
    }
  }

  StampaPila(pila)

  for !PilaVuota(pila) {
    var n int
    n, pila = Pop(pila)
    fmt.Println("Estraggo dalla pila:", n)
    StampaPila(pila)
  }

}
```