UNIVERSITÀ DEGLI STUDI DI MILANO

DIPARTIMENTO DI INFORMATICA

Tutorato di programmazione

Piano Lauree Scientifiche - Progetto di Informatica

Scheda 3: implementazione di *plan*

In questa scheda sono proposti degli esercizi di scrittura di programmi. In tutti gli esercizi è possibile riconoscere un compito (*goal*) tra quelli che abbiamo descritto tra i compiti ricorrenti negli appunti "*Goal*, *plan* e ruoli delle variabili". Prima di iniziare a scrivere i programmi, leggete bene le specifiche (e le eventuali indicazioni aggiuntive), riflettete su qual è il compito fondamentale da svolgere e individuate quale *piano* è utile implementare. Tenete gli appunti a portata di mano: non dovete inventarvi niente di nuovo, ma solo *adattare* i piani ai vari casi particolari!

Autori:
Violetta Lonati (coordinatrice del gruppo), Anna Morpurgo e Umberto Costantini

Hanno contribuito alla revisione del materiale gli studenti tutor:
Alessandro Clerici Lorenzini, Alexandru David, Andrea Zubenko, Davide Cernuto, Francesco Bertolotti, Leonardo Albani, Luca Tansini, Margherita Pindaro, Umberto Costantini, Vasile Ciobanu, Vittorio Peccenati.

Si ringraziano per i numerosi spunti:
Carlo Bellettini, Paolo Boldi, Mattia Monga, Massimo Santini e Sebastiano Vigna.

Nota sull'utilizzo di questo materiale:
L'utilizzo del materiale contenuto in questa scheda è consentito agli studenti iscritti al progetto di tutorato; ne è vietata qualsiasi altra utilizzazione, ivi inclusa la diffusione su qualsiasi canale, in assenza di previa autorizzazione scritta degli autori.

Nota
In questa scheda sono proposti degli esercizi di scrittura di programmi.
In tutti gli esercizi è possibile riconoscere un compito (*goal*) tra quelli che abbiamo descritto tra i compiti ricorrenti negli appunti "*Goal*, *plan* e ruoli delle variabili".
Prima di iniziare a scrivere i programmi, leggete bene le specifiche (e le eventuali indicazioni aggiuntive), riflettete su qual è il compito fondamentale da svolgere e individuate quale *piano* è utile implementare.
Tenete gli appunti a portata di mano: non dovete inventarvi niente di nuovo, ma solo *adattare* i piani ai vari casi particolari!

1 Funzioni che implementano *plan* semplici su interi

Scrivete le funzioni richieste. Per testarle potete utilizzare il main riportato qui sotto, sostituendo a *nomeFunzione* la funzione realizzata.

```go
package main

import "fmt"

func main() {
  const N = 10
  numeri := make([]int, N)

  for i := 0; i < N; i++ {
    fmt.Scan(&numeri[i])
  }

  fmt.Println(nomeFunzione(numeri)) // per gli es. 5.1 e 5.3
  // per l'es. 5.2 invocate invece semplicemente la funzione cosi':
  // pariDispari (numeri)
}
```

1.1 Strano prodotto

Specifiche: Scrivere una funzione `stranoProdotto(numeri []int) int` che, data come parametro una slice di interi, trovi quelli che sono maggiori di 7 e multipli di 4 e ne restituisca il prodotto. Ad esempio, se la slice contiene i numeri 12, 3, 4, 8, 9, 2, la funzione dovrà restituire il numero 96 (pari al prodotto di 12 per 8).

1.2 Pari dispari

Specifiche: Scrivere una funzione `pariDispari(numeri []int)` che, data come parametro una slice di interi, stampi, per ciascun numero, se è pari o dispari.

1.3 Più piccolo dispari

Specifiche: Scrivere una funzione `minDispari(numeri []int) int` che, data una slice di interi, restituisca il più piccolo numero dispari (la slice può contenere sia numeri positivi che negativi).

2 Funzioni che implementano *plan* semplici su stringhe

Scrivete le funzioni richieste. Per testarle potete utilizzare il main riportato qui sotto, sostituendo a *nomeFunzione* la funzione realizzata.

```go
package main

import "fmt"

func main() {
  const N = 10
  parole := make([]string, N)

  for i := 0; i < N; i++ {
    fmt.Scan(&parole[i])
  }

  fmt.Println(nomeFunzione(parole))
}
```

2.1 Quante con 'a'

Specifiche: Scrivere una funzione `quanteConA(parole []string) int` che, data una slice di stringhe, restituisca quante stringhe contengono il carattere 'a'.

2.2 Prima con 'a'

Specifiche: Scrivere una funzione `primaConA(parole []string) string` che, data una slice di stringhe, restituisca la prima parola che contentiene il carattere 'a', o la stringa vuota.

2.3 Parola più corta

Specifiche: Scrivere una funzione `piuCorta(parole []string) int` che, data una slice di stringhe, restituisca la lunghezza della stringa più corta in termini di byte.

Nota
Il prossimo è un esercizio di *debugging*.
A volte i programmi che scriviamo non fanno quello che vorremmo. Per capire la causa del comportamento indesiderato di un programma bisogna analizzare il programma; una tecnica utile in questi casi è quella di simulare l'esecuzione del programma su dei casi specifici, tenendo traccia e annotando come cambiano i valori delle variabili. La tracciatura fornisce informazioni utili a capire cosa c'è che non va e questo è il primo passaggio per poter capire come intervenire per correggere il programma.
In questo caso, il programma è basato sul piano per la *ricerca del valore estremo*.

3 Numero vicino

Specifiche: Scrivere una funzione che, passati come argomenti una slice di interi compresi tra 0 e 20 e un intero *target*, restituisca il valore più vicino a *target*.

Considerate la seguente funzione *numeroVicino*. La funzione utilizza il *plan per la ricerca del valore estremo*, come è corretto che sia, ma l'inizializzazione è gestita in modo errato.

```
1
2  func numeroVicino(numeri []int, target int) int {
3    var closest int
4    for i := 0; i < len(numeri); i++ {
5      if math.Abs(float64(target-numeri[i])) < math.Abs(float64(target-closest)) {
6        closest = numeri[i]
7      }
8    }
9    return closest
10 }
```

1. Considerate ad esempio il caso in cui la slice è [`17 12 8 19`] e *target* vale 4 e tracciate l'esecuzione della funzione.

`numeri[i]`				
`closest`				

2. Descrivete la situazione in cui la funzione non produce il risultato voluto.
3. Correggete la funzione e salvatela con il nome `vicino.go`.

Nota
Nei prossimi esercizi, oltre ad identificare e implementare un piano, ci sarà anche da considerare come gestire l'input. Nei primi esercizi, oltre alle specifiche, troverete scritte delle osservazioni che possono guidarvi nella scrittura del programma.

4 Supera 100

Specifiche: Scrivere un programma `supera100.go` che legge da standard input una sequenza di interi positivi terminata da -1 e stampa il primo numero che supera 100, se presente; altrimenti stampa "nessun numero maggiore di 100".

Per quanto riguarda l'input, osservate che: i dati arrivano da input standard; si tratta di una serie di dati che possono essere elaborati in modo uniforme; non sempre è necessario leggere tutta la sequenza di interi in input, infatti non appena il numero inserito è maggiore di 100 è possibile (e opportuno) terminare la lettura.

5 Conto Corrente

Specifiche: Scrivere un programma che legge da riga di comando un intero che rappresenta il saldo di un conto corrente. Il programma legge poi da standard input una serie di numeri interi che rappresentano spese da addebitare sul conto e stampa il saldo finale. La lettura si interrompe quando il saldo è <=0.

Per quanto riguarda l'input, osservate che: i dati arrivano da standard input; si tratta di una serie di dati che possono essere elaborati in modo uniforme; non è necessario tenere in memoria tutti i numeri, ma solo l'ultimo numero letto per addebitarlo come spesa sul conto corrente.

6 Andamento

Specifiche: Data da standard input una serie di interi positivi terminata da 0, stampare '+' ogni volta che il nuovo valore è maggiore o uguale al precedente e '-' altrimenti.

Considerate il seguente programma `andamento_bug.go`. Il programma utilizza il *plan per l'elaborazione su valori adiacenti*, come è corretto che sia, ma lo implementa in modo errato.

```
1  func main() {
2    var previous, current int
3
4    fmt.Scan(&current)
5
6    for current != 0 {
7      fmt.Scan(&current)
8      if current >= previous {
9        fmt.Println("+")
10     } else {
11       fmt.Println("-")
12     }
13     current = previous
14   }
15 }
```

1. Senza eseguire il programma al computer, tracciatene l'esecuzione quando *sequenza* è uguale a `14 9 6 11 9`

   ATTENZIONE: IL PROGRAMMA SI FERMA CON 9, LEGGE SOLO DUE VALORI, PER COME è SCRITTO ORA

`previous`					
`current`					

2. Quali errori sono stati fatti nel realizzare il *plan*? Spiegate e correggeteli. Salvate il nuovo programma con il nome `andamento.go`.

7 Differenze

Specifiche: Scrivere un programma `differenze.go` che legge una serie di valori da standard input e, a partire dal secondo numero inserito, stampa le differenze col numero precedente, cioè la differenza tra il secondo e il primo, tra il terzo e il secondo, e così via. Il programma termina con input = 0.

Esempio di funzionamento

```
3
6
Differenza: 3
-1
Differenza: -7
4
Differenza: 5
0
```

Prima di iniziare a scrivere il programma `differenze.go` leggete attentamente le specifiche e le osservazioni che trovate di seguito. Poi progettate e scrivete il programma. Per quanto riguarda l'input, osservate che:

i dati arrivano da input standard; si tratta di una serie di dati che possono essere elaborati in modo uniforme; non è necessario tenere in memoria tutti i numeri, ma solo l'ultimo e il penultimo numero letto, per operarvi la differenza.

Per quanto riguarda *goal* e *plan*, osservate che:

1. il *plan* necessario in questo caso è quello per l'*elaborazione su valori adiacenti*
2. in particolare i valori adiacenti da considerare sono due, il numero appena letto e quello precedente, dovendo calcolare la loro differenza
3. sono quindi necessarie due variabili, con ruolo di *inseguitore e inseguito*, in cui salvare rispettivamente sottraendo e minuendo.

Nota
Nei prossimi due esercizi invece delle osservazioni ci saranno delle domande che vi guideranno nella fase iniziale di progettazione e impostazione dei programmi.
Prestate attenzione alle domande poste, in modo da poter essere in grado, in contesti meno guidati, di farvele da soli!

8 Quante vocali? Tante vocali!

8.1 Quante vocali?

Specifiche: Scrivere una funzione `numVocali` che, data una stringa come parametro, restituisca quante vocali contiene. Ad esempio se la parola è `albero`, la funzione restituisce `3`.

Prima di iniziare a scrivere la funzione, rispondete alle seguenti domande:

1. Quale dei *plan* per l'iterazione serve in questo caso?
2. Quali variabili occorrono per implementare questo *plan*?
3. Come e dove vanno inizializzate e come e dove vanno poi aggiornate?

8.2 Tante vocali!

Specifiche: Scrivere un programma dotato della funzione `numVocali` definita sopra e di una funzione `main` che legge da riga di comando una sequenza di parole e stampa la parola che contiene più vocali. Ad esempio se in input ci sono le parole {albero, foglia, cespuglio, aiuola, verde}, il programma stampa "aiuola".

Prima di iniziare a scrivere il programma, rispondete alle seguenti domande:

4. Quale dei *plan* per l'iterazione deve implementare la funzione `main`?
5. Quali variabili occorrono per implementare questo *plan*?
6. Come e dove vanno inizializzate e come e dove vanno poi aggiornate?

9 MinuMaiu

Specifiche: Scrivere un programma che legge da linea di comando una sequenza di stringhe di rune formate da sole lettere (non va fatto nessun controllo in proposito) e stabilisce se tutte le stringhe sono formate da lettere minuscole e maiuscole alternate oppure no. Le stringhe valide rispettano l'alternanza e possono cominciare sia con una lettera minuscola che con una lettera maiuscola (es: `SoS, sAlVaTeMi`). Ad esempio, se la sequenza passata da riga di comando è `sOno stRAnA`, il programma stampa `sequenza non valida`; se legge invece `iO sOnO gIuStA`, stampa `sequenza valida`.

Per quanto riguarda l'input, osservate che: i dati arrivano da riga di comando, quindi non c'è da fare una valutazione riguardo alla necessità di averli tutti in memoria per poterli elaborare, vengono già salvati automaticamente nell'array `os.Args`; i dati letti sono stringhe, mentre i dati da elaborare sono le rune delle stringhe, e quindi vanno estratte da queste.

Per quanto riguarda *goal* e *plan*, in questo caso di *elaborazione su valori adiacenti* occorre stabilire se una sequenza è valida o no nel senso che rispetta un'alternanza o meno. Osservate che per stabilire se una sequenza è valida occorre scorrerla tutta; basta invece fermarsi alla prima violazione dell'alternanza per stabilire che la sequenza non è valida.

Suggerimenti: i metodi `isUpper` e `isLower` del pacchetto `unicode` permettono di verificare se una lettera è minuscola o maiuscola, rispettivamente.

Prima di iniziare a scrivere il programma `minumaiu.go` leggete attentamente le specifiche e le osservazioni. Poi progettate e scrivete il programma.