Contents

G1: Introduzione

Cos'è un computer

- Hardware
- Firmware: si trova nella memoria ROM (Read Only Memory, in realtà scrivibili). Linguaggio macchina eseguito durante il boot.

- Sistema operativo: è un software, mette a disposizione all'esterno delle API (Application Program Interface) che permettono di interfacciarsi con il firmware. I linguaggi di programmazione comunicano con il SO attraverso il binding. Gestisce anche le risorse.
- Applicativi

Hardware

Componenti:

- Scheda/Piastra madre: contiene i componenti, saldati o con cavo, collegati tra loro tramite transistor in silicio
- CPU: fa conti, sposta memoria
- Memoria: può essere sequenziale (es. nastri) o random (es. Random Access Memory)
- Periferiche: collegate alla MOBO interagiscono con l'esterno, di tipo Input o Output
- Memorie di masa: contengono dati permanentemente; dischi rigidi (HD) e memoria a stato solito (SSD) senza nulla meccanico
  - Gli HD sono più grandi e fragili
  - Gli SSD sono più veloci e costosi

Non si lavora su memoria di massa ma su memoria centrale perchè la prima è troppo lenta e si rovinerebbe. I dati vanno spostati da memoria a memoria di massa.

Software

I linguaggi di programmazione permettono di dare istruzioni ai computer con un linguaggio simile al nostro, senza usare il linguaggio macchina.

I linguaggi ad alto livello contengono istruzioni molto semplici, che devono diventare linguaggio macchina. La conversione può essere effettuata da:

- **Compilatore** (nome tecnico traduttore): legge tutto il file e crea un nuovo eseguibile in linguaggio macchina, è più lento ma l'esecuzione del programma è veloce
- **Interprete**: esegue le istruzioni mentre legge il codice. è più più veloce ma l'esecuzione sarà lenta. Il vantaggio è che si può eseguire il codice tante volte al volo senza aspettare i tempi della compilazione.
- **Un mix dei due**: permette di avere una rappresentazione intermedia tra linguaggio macchina e alto livello (es. bytecode in java) creata dal compilatore. Questo livello intermedio verrà poi interpretato.

Rappresentazione della memoria

La memoria è una sequenza di bit (0 o 1). I bit si dividono in ottetti chiamati byte, 1 byte = 8 bit. Per la CPU la memoria è un insieme di byte numerati per identificarne la posizione.

I multiplici binari sono identificati da 2^10n e sono Ki, Mi, Gi e Ti (kilo binario, mega binario etc…). Il rapporto tra byte e quello che rappresenta è irrilevante, il flusso di dati verrà poi elaborato da una periferica, mentre per il PC rimangono dati.

Modello di von neumann

Una macchina è formata da diversi elementi collegati tra loro tramite un bus:

- CPU
  - ALU (Arithmetic Logic Unit): fa i conti
  - Registri: della grandezza dell'architettura
  - Program Register e Program Counter
- Memoria
- Periferiche

La CPU esegue le istruzioni in 3 parti: fetch, decode ed execute. Fetch: guarda dov'è l'istruzione nel program counter e la mette nel program register; decode: la decodifica per capire cosa fare: exec: esegue l'istruzione. Infine, sposta il program counter all'istruzione dopo.

Linguaggio macchina

Esempio linguaggio macchina in architettura Risk-V nella CPU MIPS32 a 32bit (4 byte) con istruzioni che parte dalla posizione di memoria 1000:

```
1000: lw $2, 2000($0) # load word, sposta i dati dalla memoria al registro $2
1004: add $3, $0, $0  # somma $0, $0 e sposta dentro $3. Mette 0 dentro $3
1008: add $4, $3, $0
1012: mult $4, $4, $4 # mette il quadrato di $4 dentro sè stesso
1016: slt $5, %4, $2  # slt = set if less. Controlla se $4 è minore di $2 e mette il risultato (
1020: beq $5, $0, +8  # beq = brench (ramifica) if equal. Altera il program counter, se i primi
1024: addi $3, $3, +1
1028: j 1008         # jump, altera il programm counter
1032: sw $3, 3000($0)
```

Rappresentazione interna dei dati (immagini, audio, etc...)

Ogni cosa è rappresentata da dei bit e bisogna quindi definire uno standard di rappresentazione (es. ASCII per le lettere). Le due operazioni per passare dalla rappresentazione al computer sono **discretizzazione** e **quantizzazione**.

Immagini:

La **discretizzazione** di un'immagine ad esempio consiste nel dividere l'immagine in celle molto piccole, ciascuna delle quali avrà assegnato un colore. La quantizzazione definisce la rappresentazione dei colori (es. RGB).

Suoni

La discretizzazione avviene sull'asse temporale, registrando il segnale audio sulla griglia. Le frequenze del segnale si possono campionare per diversi kHZ al secondo, il campionamento più usato è a 44kHZ.

Un secondo di audio costa 44000 byte x 2 (byte di quantizzazione) x 2 (canali stereo) = 176000 byte.

L'ascolto della musica avviene attraverso il DAC (Digital to Analog Converter), che converte da digitale ad analogico.

Esempio compressione dati:

Immagini e audio non sono tutte equiparabili. Si può ridurre la quantità di dati in un file. Un tipo di fase permette data una lista di byte (es. 0, 0, 2, 2, 2, 100, 100, 2, 2, 2, 2) invece di mandare byte ripetuti di mandarli in formatto conteggio-byte ripetuto (lista di prima –> 2, 0, 3, 2, 2, 100, 4, 2). Questo però assume che i byte siano ripetuti spesso, e andava bene quando i colori erano pochi.

Altri tipi di compressione sono "con perdita di informazione" (lossy) o "senza perdita di informazione" (lossless).

Una compressione lossy manda le ripetizioni ma divise 2, rendendo le sequenze pari giuste ma dispari sbagliate (lista di prima –> 1, 0, 1, 2, 1, 100, 2, 2).

G2: Il linguaggio Go

Ciclo di vita del software

Diverse fasi:

1. Esigenza
2. Studio di fattibilità –> ritorno alle esigenze se impossibile
3. Analisi delle specifiche (es. studio I/O)
4. Progettazione
5. Sviluppo
6. Testing e debugging –> ritorno allo sviluppo se non funziona
7. Rilascio
8. Manutenzione –> ritorno a fase di testing in caso di problemi

Tool go

Go è un linguaggio compilato, può essere eseguito in due modi: compilazione con il comando `go build` ed esecuzione dell'eseguibile creato; esecuzione diretta del codice con `go run`.

Altri tool sono `go doc` che prende una funzione e ne da la documentazione. `go fmt` formatta il file go.

G3: Struttura del linguaggio

Parti del linguaggio

Un linguaggio di programmazione contiene i token che identificano qualsiasi cosa abbia significato, essi si suddividono in:

- Parole chiave: parole usate dal linguaggio unici (es. func, var, package)
- Identificatori: nomi, diversi dalle parole chiave, identificati da sequenze di caratteri e underscore. Devono iniziare con un underscore o una lettera (non numeri). Una lettera in Go è qualsiasi codice dell'unicode
- Spaziatori: hanno lo scopo di rendere il codice più leggibile e di separare i token che appiccicati non avrebbero senso. .

- Letterali: costanti (es. 42, “Hello, world”).
- Operatori: operatori numerici o di altro tipo comef15

Il punto e virgola non è necessario a meno che due istruzioni siano sulla stessa riga.

Con “zucchero sintattico” si definiscono i costrutti sintattici che attraverso un'operazione di “sughering” non cambiano la struttura programma ma soltanto la forma. Es: `x = x + 1` => `x++`.

Alcune parti del codice possono essere commentate, ovvero essere saltate dal compilatore al fine di descrivere il codice. I token per commentare un sorgente sono due: `// testo` commenta una singola riga; `/* ... */` commenta più righe.

Variabili

Go è un linguaggio imperativo: il programma ha uno stato che viene manipolato durante il funzionamento. Lo stato è rappresentato dalle variabili.

Per dichiarare una variabile si utilizza la sintassi `var NOME TIPO`. L'operatore `=` assegna un valore a una variabile; `:=` permette di dichiarare e assegnare un valore contemporaneamente a una variabile senza specificarne il tipo.

```
var x int = 0
x := 0
```

L'assegnamento breve non funziona se l'identificatore a sinsitra è già stato dichiarato. Tuttavia, in caso di assegnamento breve multiplo parallelo basta che una sola variabile a sinistra sia nuova.

```
// Non compila
var x int
x := 2

// Compila
var x int
x, y := 1, 3
```

Il tipo float64

I letterali frazionari rappresentano il tipo float64, numeri decimali a virgola mobile. La sintassi `1.2234en` moltiplica il numero per 10^n. (es. 1.2234e4 => 12234).

Inferenza di tipo

Le operazioni aritmetiche sono valide solo se tutti gli operandi sono dello stesso tipo. Una possibile soluzione se si utilizzano dei etterali nell'espressione è cambiarne il tipo esplicitamente (es. 3 => 3.0 da intero a float). Altrimenti va applicata l'inferenza di tipo, ovvero la conversione del tipo dell'operando con la sintassi `tipo(espressione)`.

I/O formattato

Ogni programma, se non diversamente specificato, riceve dei byte e da in output dei byte rispettivamente da stdin e verso stdout. La funzione Scan() del pacchetto fmt legge dei byte da stdin; le funzioni Print() e Println() scrivono byte su stdout con la differenza che Println() va a capo e mette uno spazio tra gli argomenti passati.

G4: Strutture di selezione e cicli

Costrutto sintattico di selezione

Il costrutto di selezione binaria esegue il corpo solo se è soddisfatta una condizione. In caso contraria esegue cioò che è contenuto nel corpo di else, se presente.

```
if CONDIZIONE {
    CORPO
} else if CONDIZIONE {
    CORPO
} else {
    CORPO
}
```

Dove CONDIZIONE è un'espressione booleana. Le graffe sono obbligatorie anche con una sola istruzione. Dopo else è possibile mettere anche un altro if:.

Algebra di Boole

Ogni condizione logica può essere scritta utilizzando AND (congiunzione logica), OR (disgiunzione logica) e NOT (negazione). AND è distributiva rispetto ad OR.

La **legge di De Morgan** dice che un'espressione negata può essere riscritta portando dentro la negazione e invertendo l'operatore, esempio: `!(a && b) = !a || !b)`.

Per valutare gli operatori logici il Go utilizza una tecnica detta valutazione cortocircuitata. Date due espressioni come `(ESPR) && (ESPR)` e la prima è falsa, essendo entrambe legate da una congiunzione, Go non valuta la seconda ma da direttamente falso. Lo stesso vale per `(ESPR) || (ESPR)` dove se la prima è vera la seconda non viene valutata.

Visibilità di variabili

Una variabile essite soltanto nel suo spazio dei nomi, delimitato solitamente dalle graffe. Limitare la visibilità di una variabile è utile per evitare errori e risparmiare memoria.

```
// delta è visibile all'interno dell'if e dell'else
if delta := b * b - 4 * a * c; delta > 0 {
    ...
} else {
    ...
}

// x è visibile solo nel blocco if
```

```
if espressione {
    x := 2
    ...
} else {
    ...
}
```

Quando una variabie viene dichiarata in un blocco di gerarchia minore essa sarà diversa e appartenente al suo spazio dei nomi. Esempio:

```
func main() {
    n := 2

    if n > 0 {
        var n int
        n = 1
    }
}
```

La variabile n dentro main() è diversa dalla variabile del contesto del blocco if.

Cicli

I cicli ripetono un'espressione per un certo numero di volte date certe condizioni. Il linguaggio Go ha soltanto il ciclo for, di diversi tipi.

For zerario

```
for {
    CORPO
}
```

Ripete all'infinito le istruzioni nel corpo.

For unario

```
for ESPRESSIONE_BOOLEANA {
    CORPO
}
```

Ripete le istruzioni fino a quando l'espressione booleana rimane vera. Il for zerario equivale infatti a `for true`.

For ternario

```
for PRIMA_ISTRUZIONE; ESPRESSIONE_BOOLEANA; ISTRUZIONE {
    ...
}
```

Questa terza forma del for esegue PRIMA_ISTRUZIONE, controlla ESPRESSIONE_BOOLEANA e se vera entra nel corpo. Finito di eseguire il corpo esegue ISTRUZIONE, controlla ESPRESSIONE_BOOLEANA e se vera rientra nel corpo, reiterando per le prossime istruzioni.

G5: Interi

Overflow e imprecisione

Effettuare alcune somme e moltiplicazioni potrebbe portare ad overflow ed imprecisioni.

```
n := 1.0
n -= 0.2
n -= 0.2
n -= 0.2
n -= 0.2
n -= 0.2
```

Non da 0 ma un numero molto piccolo vicino a zero poichè 0.2 non ha una rappresentazione binaria esatta.

Inoltre effettuare operazioni troppo grandi potrebbe portare a overflow (trabocco della variabile intera)

Tipi unsigned int

Go prevede diversi tipi interi:

uint uint8 (byte) uint16 uint32 uint64

In caso di trabocco il risultato viene messo in modulo con 2^k dove `k` è il numero di bit. Questo fenomeno è utile per la rappresentazione in complemento a 2, infatti la somma di 1 e -1 in C2 da 256, che in modulo a 8 da 0.

Il C2 è anche il motivo per cui l'overflow scrive numeri negativi, in quanto il primo bit andrà a 1.

La rappresentazione dei floating point avviene in IEEE754. Poichè un binario con virgola è la somma di 2^{-n} alcuni numeri non sono rappresentbili esattamente, ad esempio 1/3. L'aritmetica in floating point registra gli overflow, segnando a infinito il risultato.

G6 : Stringhe e selezione multiaria

Le stringhe

La prima codifica dei caratteri utilizzata dagli elaboratori era l'ASCII, inventata dagli USA e basata su 8 bit. L'ASCII non comprendeva però gli accenti e tutti i caratteri di altre lingue come il cinese, portando a creare nuovi standard ISO per le varie lingue.

Il problema degli standard ISO tuttavia era che fosse necessario ogni volta specificare il set di caratteri utilizzato.

Nacque così lo standard unicode, che codifica qualunque carattere incluse ad esempio le emoji. Ogni carattere ha una descrizione, le associazioni carattere-descrizione formano un repertorio di caratteri codificati con un certo numero.

Unicode ha tabelle per ogni lingua, i primi 128 caratteri corrispondono esattamente all'ASCII. Unicode non dipende dai glifi, il font. Sono i glifi a dover supportare una parte di unicode.

Le stringhe su Go sono una sequenza di rune, rappresentabili da unicode, dunque sequenze di byte in UTF-8.

Ogni carattere su unicode non è rappresentato da tutti e 32 i bit ma solo da una parte. Bisogna quindi scegliere una codifica, la più usata è UTF-8. UTF-8 è compatibile con il passato poichè i primi 128 caratteri sono esattamente come quelli di ascii a 7 bit. Su UTF-8 è possibile specificare il numero di byte usati nel primo byte, ad esempio se inizia con 1110xxxx significa 3 byte. I byte di continuazione dopo iniziano con 10 per identificare l'inizio della sequenza UTF-8.

Le stringhe su Go sono di tipo "string", mentre i singoli caratteri "rune". "rune" è zucchero sintattico, sta ad indicare un intero da 32 bit. è possibile ricavare i byte delle stringhe indicizzandole (es. s = "ciao", s[0] da il byte rappresentante 'c', byte != rune). Per accedere alle singole rune si utilizza range:

Forma range del for

```
var c string = " tre"

for i, r := range c {
// i = indice bit della stringa dove inizia la runa r, r = runa
}
```

_ blank variable, può sostituire ad esempio la i nel for range.

La concatenazione avviene usando il +. È valido l'operatore +=.

Le stringhe si possono confrontare con ordinamento lessicografico (=alfabetico) come fossero dei numeri.

Slice di stringhe

La sintassi `s[EXPR1:EXPR2]` estrae i byte da una sottostringa con EXPR1 indice incluso e EXPR2 escluso, dove s è una stringa.

Costrutto di selezione multiaria

```
switch EXPR_CONDIZIONE {
    case R1, R2, ..., R3:
        ...
    default:
        ...
}
```

EXPR_CONDIZIONE è opzionale, es:

```
x := 2

switch {
    x > 0:
        Println("Positivo")
    x < 0:
        Println("Negativo")
    x == 0:
        Println("Nullo")
}
```

Usando la parola chiave `fallthrough` alla fine delle istruzioni di un case questo va avanti a considerare anche i casi avanti a cascata. `break` invece interrompe l'esecuzione di uno switch. `continue` riprende il ciclo dall'inizio saltando le istruzioni successive.

G7: Funzioni

Definire una funzione

Una funzione permette di dividere il codice in gruppi. Vengono definite come segue:

```
func NOME_FUNZIONE(PARAMETRI_FORMALI) (TIPI_RITORNO) {
    // istruzioni
}
```

I parametri formali vanno definiti con `NOME_VARIABILE TIPO`. I tipi di ritorno vanno messo tra parentesi se più di uno e separati da una virgola.

Una chiamata a funzione avviene attraverso il suo nome seguiti dalle parentesi tonde con all'interno espressioni che daranno il valore ai parametri, es. `foo(n + 2)`. Le espressioni passate devono essere dello stesso tipo dei parametri formali nella definizione della funzione.

Return

Le funzioni posso ritornare dei valori attraverso l'istruzione `return` che accetta una lista di espressioni separate da virgole. Le espressioni devono essere dello stesso tipo di quelli nella definizione della funzione. Se è indicato un tipo di ritorno ma la funzione non ritorna nulla il compilatore da errore.

`return` senza nessuna espressione a seguire ha tre modalità di utilizzo: - la funzione non prevede tipi di ritorno: l'istruzione `return` esce dalla funzione - la funzione prevede tipi di ritorno: viene restituito il valore di default dei tipi - la funzione prevede tipi di ritorn con nome: la funzione restituisce il valore della variabile in quel momento. Non è obbligatorio ritornare la variabile indicata nei tipi di ritorno.

es. ultimo caso:

```
func foo() (s int) {
    s = 1
    return s
}
```

`foo()` ritorna 1

Comma ok

Il comma ok consiste nel restituire in una funzione un intero o un booleano (o altro) per indicare se la funzione ha eseguito correttamente o meno. esempio:

```
func DecToBin(n int) (string, bool) {
    if n < 0 {
        return 0, false
    }

    // istruzioni

    return bin, true
}
```

G8: Conversioni di stringhe

Pacchetto strconv

strconv è un pacchetto di Go che si occupa di conversioni di stringhe. Due funzioni molto importanti sono Atoi() e Itoa() che convertono rispettivamente un ascii a un intero e un intero in un ascii.

G9: Puntatori

Puntatori

I puntatori sono variabili che contengono la posizione di memoria di altre variabili. Per indicare un tipo puntatore si utilizza la sintassi `*[TIPO A CUI PUNTA]` (es. `var p *int` dichiara un puntatore p a tipi interi).

Un puntatore non iniziallizato contiene il valore `nil` e non punta a nessun'area di memoria.

Operatori sugli indirizzi

Indirizzamento

`&` è chiamato operatore di indirizzamento, applicato a una variabile ha come valore la posizione in memoria della variabile. Notare che l'espressione `q = &(&x)` non funziona perchè & è un operatore che si applica soltanto a una variabile. Esempio:

```
var x int = 5
fmt.Println(&x)
```

stampa la posizione in memoria di x

Deferenziazione

`*` è chiamato operatore di deferenziazione e applicato a un puntatore restituisce il valore della variabile a cui punta. Esempio:

```
var x int = 5
var p *int = &x

*p = 3
fmt.Println(x)
```

stampa x poichè l'istruzione `*p = 3` ne ha cambiato il valore.

L'espressione `*&x` è identica a scrivere `x`. L'espressione `&*p` è identica a scrivere `p`.

Errori comuni

Deferenziare un operatore nullo genera un errore a livello del SO e dunque il compilatore non darà avvisi. Il SO crea uno spazio virtuale per i puntatori, che vanno poi mappati allo spazio fisico.

L'aliasing è un fenomeno per cui ci si puà riferire a una variabile di un programma in più modi diversi. Questo diventa particolarmente vero con i puntatori e può creare problemi.

Passaggio per riferimento

La maggior parte delle funzioni passano i parametri formali per valore. I puntatori se messi come parametri permettono di modificare i valori delle variabili passate all'interno delle funzioni.

Puntatori a puntatori

Si possono creare diversi livelli di puntatori, quindi utilizzare puntatori ad altri puntatori. Ad esempio:

```
var p *int
var q **int
```

q punta a un puntatore che punta ad un intero

Funzione new

La funzione new riserva dello spazio per un tipo e ne ritorna la posizione in memoria. La sintassi è `new(T)` dove T è un tipo.

Con new si possono creare variabili al tempo di esecuzione. Essendo creata a tempo di esecuzione una variabile creata con new non ha nome ma solo il puntatore associato.

G10: Strutture

Type

La parola chiave type permette di creare alias dei tipi di Go. L'istruzione `type pippo = int` imposta pippo come alias del tipo int e rende possibile creare variabili del tipo pippo.

Alternativamente senza mettere `=` (es. `type anno int`) il nuovo tipo si baserà sul tipo base ma non permetterà di effettuarci operazioni.

Creare strutture

Una struttura rappresentà un'entità avente diversi attributi, detti campi. Ogni campo ha nome e tipo. Per creare una struttura si utilizza la parola chiave struct come segue:

```
type NomeStruttura struct {
    nomeCampo1 tipoCampo1
    nomeCampo2 tipoCampo2
}
```

Una nuova struttura può essere allocata con la funzione new(), oppure indicandone i valori in due modi:

```
var t NomeStruttura = NomeStruttura{valoreCampo1, valoreCampo2}
var t NomeStruttura = NomeStruttura{nomeCampo1:valoreCampo1, nomeCampo2:valoreCampo2}
```

Se un campo non è specificato esso assume il valore di default.

Per accedere al valore di un campo si utilizza il selettore, ovvero la notazione: `t.NomeCampo`.

Una struttura può contenere dei campi anonimi a cui è associato solo un tipo. Il campo anonimo può essere un'altra struttura, creando un semplice meccanismo di ereditarietà. Si può accedere ai campi della struttura incorporata senza specificarla, ad esempio:

```
type s1 struct {
    v1 int
    v2 int
}

type s2 struct {
    v3 int
    s1
}

func main() {
    s1 := s1{}
    s1.v1 = 2
}
```

Strutture e puntatori

Alle funzioni che hanno come parametro una struttura è conveniente passarne il puntatore poichè creare una copia della struttura occupa molta memoria.

I puntatori a strutture possono accedere e modificare i campi della struttura senza dereferenziare.

G11: Vettori

Array

Gli array sono una sequenza di elementi omogenei di lunghezza fissa.

La definizione di un array avviene tramite la sintassi `var NOME_ARRAY [NUMERO_ELEMENTI]TIPO` dove NUMERO_ELEMENTI è un'espressione costante intera. Per accedere agli elementi di un'array si usa la notazione `NOME_ARRAY[INDICE]`. L'indice è un'espressione intera che va da 0 a n_elementi - 1. Ad esempio:

```
var a [10]int

a[0] = 1
a[2] = 3

fmt.Println(a[2])
```

stampa `3` in quanto valore di `a` nella posizione due.

Altri modi per assegnare valori a un array sono:

- `var x [10]int = [10]int{1, 2, 3, 4, ..., 10}`
- `x := [...]int{1, 2, 3, 4}`: "..." dice al linguaggio di calcolare automaticamente la dimensione dell'array in base al numero di elementi inseriti tra graffe.

Slice

Le slice sono sequenze di elementi omegenei di lunghenzza variabile, la definizione avviene attraverso `var NOME_SLICE []TIPO`. Si inizializzano nello stesso modo degli array. Al posto di TIPO si può inserire `any` che permette di inserire elemtni di tipo diverso.

La funzione append(SLICE, NUOVI_ELEMENTI) restituisce una nuova slice formata da quella passata con i nuovi elementi, esempio: `a = append(a, 1, 2, 3)`.

La funzione len() restituisce la lunghezza di array e slice.

Sugli slice si può usare il range.

```
for i := range a { // for i := 0; i < len(a); i++
    Println(a[i])
}
```

Le stringhe sono in realtà slice immutabili

Subslicing

Il subsclicing è un operazione che restituisce una sotto-sequenza di una slice con la sintassi `NOME_SLICE[INIZIONE:FINE]` con INIZIO e FINE espressioni intere, FINE esclusa.

Assegnare a uno slice una sotto-slice ne assegna il puntatore a memoria causando aliasing. esempio:

```
a := []int{1, 2, 3, 4, 5, 6}
b := a[2:5]
b[0] = 0
fmt.Println(a[2])
```

stampa 0 poichè b punta alla stessa area di memoria di a.

Implementazione delle slice

Una slice è una struttura di 3 campi:

- Puntatore a un elemento di un array che è il primo elemento della slice (ma potrebbe non essere il primo elemento dell'array)
- Lunghezza della slice
- Capacità: numero massimo di elementi che l'array cui lo slice fa riferimento senza cambiare array

Quando si esegue append() il linguaggio controlla se la capacità è abbastanza per il nuovo elemento, se sì crea una nuova slice che punta allo stesso array col nuovo elemento e lo restituisce. Se la capacità non è abbastanza, Go alloca un nuovo array grande il doppo del precedente dove la prima metà è uguale all'array originale. Lo svantaggio è che se devo aggiungere solo un element che sfonda la capacità di 1 allora alloco il doppio della memoria necessaria.

Il raddoppio avviene in modo esponenziale perchè se cresce in modalità costante la velocità della struttura diventa sempre più lenta più grande è la struttura.

Il subslicing consiste nel spostare il puntatore dello slice precedente e di cambiare la lunghezza (la differenza degli indice) e la capacità (capacità originale - primo indice).

Funzione cap

La funzione cap() ha come argomento un array o una slice e da la capacità del vettore passato.

Funzione make

La funzione make([]TIPO_SLICE, lunghezza, capacità) permette di impostare lunghezza e capacità a una nuova slice. La capacità non è obbligatoria, se non indicata è uguale alla lunghezza.

Stringhe convertibili

Le stringhe si possono convertire come slice di rune (es. `sr := []rune(s)` dove s è una stringa). Una runa sono 4 byte, un carattere ASCII 1 byte quindi va ricordato che la coinversione consuma memoria.

Argomenti da linea di comando

La libreria "os" contiene lo slice Args che contiene gli argomenti passati da riga di comando all'esecuzione del programma.

printf

La funzione printf stampa stringhe formattate. I modificatori (verbi) con il simbolo % indicano come stampare i valori passati.

Ad esempio `printf("Il numero è %f.2", numero)` stampa un float con due cifre decimali.

I modificatori principali sono:

- %f: numeri float
- %v: stampa la variabile come la scriverebbe Go
- %#v: stampa il letterale come verrebbe creato a livello di codice
- %T:
- %s: stringhe, possibile allineare la stringa con un numero davanti
- %%: stampa il carattere '%'

G12: Numeri pseudocasuali

Generatori di numeri pseudocasuali

Generare numeri casuali è molto difficile. I generatori di numeri più comunemente usati in verità generano numeri pseudocasuali.

Generatore congruenziale lineare (LCG)

Un esempio di algoritmo usato da linguaggi come C è l'LCG.

Questo algortimo prende un numero x tra 0 e un certo n, date poi due costanti a e b ritorna x = (ax + b) % n. Di solito $n = 2^{n_b itcpu}$).

G13: Ricorsione

Stack e heap

Lo stack è una struttura dati di tipo LIFO (Last In First Out, come una pila). Viene usata per le allocazioni di memoria di tipo statiche e per le chiamate di funzioni; quando una funzione viene seguita crea un nuovo frame sullo stack, che viene rimosso quando ritorna.

L'heap invece salva la memoria allocata e liebrata durante l'esecuzione del programma. Viene dunque utilizzata per elementi quali slice, mappe e strutture create con new() e make().

Stdin/stdout

os.Stdin os.Stdout os.Stderr

Le funzioni di lettura e scrittura da e per terminale hanno una versione con la F davanti per specificare il file da cui leggere o scrivere; la versione con la S davanti opera sulle stringhe. I file di os di sistema possono essere passati come qualsiasi altro file.

G14: Metodi e interfacce

Metodi

I metodi sono funzioni che agiscono su una variabile (ricevitore) di un certo tipo. Si definiscono come segue:

```
func (nomeRicevitore tipoRicevitore) nomeFunzione(parametri) (tipiRitorno)
```

E si chiamano come segue:

```
ricevitore.nomeFunzione(parametri)
```

Due metodi possono avere lo stesso nome purchè abbiano un ricevitore di tipo diverso. Si possono attaccare funzioni soltanto a tipi nuovi definiti con type.

Se il ricevitore è un puntatore Go converte automaticamente in puntatore la variabile passata.

Leggere più righe da stdin

```
scanner := bufio.NewScanner(os.Stdin)

for scanner.Scan() {
    Println(scanner.Text())
}
```

Interfacce

Un'interfaccia è un'insieme di dichiarazioni di metodi. Ad esempio:

```
interface Pippo {
    pippo(int) int
}
```

In questo caso posso dichiarare un nuovo tipo e associargli l'interfaccia Pippo implementando tutti i metodi dell'interfaccia. In questo modo posso creare un metodo associato all'interfaccia Pippo e chiamarlo su qualunque tipo implementi l'interfaccia.

Go si basa sul ducktipying cioè non è necessario specificare che il tipo implementi l'interfaccia.

Si possono attaccare metodi come String() ai tipi nuovi che formatta in modo diverso il tipo quando si stampa.

G15: I/O su file

Per aprire un file si utilizza la funzione Open() del pacchetto os. file è un tipo che implementa reader e quindi si può essere ovunque venga accettato un tipo reader.

esempio:

```
import (
    . "fmt"
    "strconv"
    "os"
    "bufio"
)

func main() {
    s := os.Args[1]
    file, err := os.Open(s)
    if err != nil {
        Println(err)
        return
    }

    scanner := bufio.NewScanner(file)

    for scanner.Scan() {
        Println(scanner.Text())
    }
}
```

G16 : Tipi funzionali e funzioni anonime

Un tipo funzionale è un tipo che contiene una funzione. La dichiarazione è identica a una funzione ma senza gli identificatori, ad esempio: `var x func(int, string) bool`. I tipi funzionali funzionano come tutti gli altri letterali, ad esempio: `pippo(20, func (x int) int {return x * x}`, in questo caso si chiama funzione anonima.

Un esempio pratico è la funzione Slice(x any, less func(i, j int) bool) che prende la slice da ordinare come x e una funzione di confronto:

```
s := []int{0, 4, -1, 2}
sort.Slice(s, func(i, j int) bool {
    return s[i] > s[j]
})
```

Questo è possibile perchè Go applica la chiusura lessicale: all'interno di una funzione anonima ci si può riferire a qualunque variabile visibile in quel momento al chiamante. Una variabile che ha chiusura lessicale viene salvata in memoria (non nello stack).

defer, panic e recover

La parola chiave `defer` utilizzata dentro una funzione esegue una determinata funzione al termine della funzione dentro cui è chiamata. Gli argomenti sono valutati al momento della chiamata, non all'uscita.

Dentro una funzione differita si può usare `recover()` che restituisce la descrizione di un panic (l'esecuzione del programma termina comunque).

Esempio di recupero recover da un panic per divisone a 0, il programma va avanti e stampa un messaggio d'errore:

```
package main

import . "fmt"

func dividi(x, y int) float64 {
    defer func() {
        if r:= recover(); r != nil {
            Println("Errore:", r)
        }
    }()

    return float64(x / y)
}

func main() {
    Println(dividi(2,0))
    Println("Avanti")
}
```

Funzionamento I/O

Reader e Writer sono due interfacce con metodi Read(p []byte) e Write(p []byte) per leggere e scrive len(p) byte da/verso un file. L'interfaccia Closer col metodo Close() chiude un file.

Il tipo File del pacchetto os implementa le interfacce Reader, Writer e Closer.

Per chiudere più risorse a cascata facilmente si può usare l'esecuzione differita. L'esecuzione differita permette di far eseguire una funzione in uscita a un'altra funzione e avviene attraverso la parola chiave `defer`. Si nota che è possibile per chiusura lessicale stampare un valore passandolo a una funzione ma poi ristamparlo col valore corrente.

G17: Moduli e pacchetti

Un modulo contiene più pacchetti, ogni pacchetto è una divisone sintattica dagli altri pacchetti.

Il comando `go get URL` scarica il pacchetto in URL. Per assegnare un numero di versione si utilizza `git tag`; per fare il push dei tag bisogna usare il comando `git push --tags`.

C1: Il linguaggio C

Hello world:

```
#include <stdio.h>

void main() {
    printf("Hello, world\n");
}
```

Costrutti di controllo di flusso

I costrutti di controllo del flusso sono simili al Go.

Selezione

```
if (EXPR) {
    ...
} else {
    ...
}
```

Esegue se EXPR espressione intera è diversa da 0. In C non esite il tipo bool, il vero è un valore diverso da 0; viceversa falso è il valore intero 0.

for

```
for (int i = 0; i < n; i++) {
    ...
}
```

while

```
while (EXPR) {
    ...
}

do {
    ...
} while (EXPR)
```

Esegue finchè EXPR è diversa da 0.

switch

```
switch (EXPR) {
    case LABEL:
        ...
    default:
        ...
}
```

Viene valutata EXPR, se EXPR == LABEL con LABEL costante allora esegue il codice sottostante e va fino in fondo, anche nei case successivi. Spesso quindi si utilizza un `break` per terminare il case.

C2: Tipi

I tipi fondamentali sono:

`int`: interi da 16 a 64 bit `long`: interi da 32 a 64 bit `long long`: interi da 64 a 128 bit `float`: virgola mobile a 32 bit `double`: virgola mobile a 64 bit `char`: carattere su 8 bit E i relativi unsigned. Dal C99 esistono nuovi tipi con dimensione esatta, introdotti dalla libreria stdint.h, ad esempio: int8_t, int32_t, uint32_t, etc…

La libreria stdbool.h introduce il tipo `bool`, che rappresenta comunque 0 ed 1 ed è dunque zucchero sintattico.

Il tipo string si può implementare come array di char ma non esiste.

include

La direttiva include è eseguita dal preprocessore, ed importa altri file nel codice, di fatto copiandone il contenuto nel file che andrà compilato. L'utilizzo dei simboli maggiore e minore `#include <nome.h>` indica di cercare i file nella directory standard del sistema operativo; l'utilizzo delle virgolette `#include "nome.h"` cerca il file nella directory corrente.

include viene usato principalmente per importare intestazioni delle librerie, i file con estensione `.h` che contengono le definazioni delle funzioni, costanti, etc…

variabili

La dichiarazione di una variabile segue la sintassi `TIPO nomeVar0, nomeVar1 = valVar1, ..., nomeVarN`. Si possono dunque dichiarare più variabili sulla stessa riga e si possono inizializzare separatamente.

Dopo l'assegnamento si possono inserire le lettere:

- `U`: indica che è un unsigned
- `L`: indica che è long
- `LL`: indica che è long long

In C si possono utilizzare variabili non inizializate ma non hanno un valore di default, dunque una variabile appena dichiarate ha il valore già precedentemente contenuto nel frame d'attivazione.

Puntatori

La sintassi dei puntatori e i relativi operatori di indirizzamento `&` e deferenziazione `*` sono simili a Go, con la differenza che di fatto i puntatori sono sempre interi.

Stringhe

Il tipo carattere si dichiara col tipo `char` e può contenere solo caratteri ASCII.

Le stringhe sono implementate come puntatori a caratteri. L'ultimo carattere di una stringa è \0 che ne indica la fine. Si può accedere ai singoli caratteri indicizzando come fosse un'array: di fatto esiste un'aritmetica dei puntatori.

Array

Un'array si dichiara come:

```
TIPO NOME[DIMENSIONE]
```