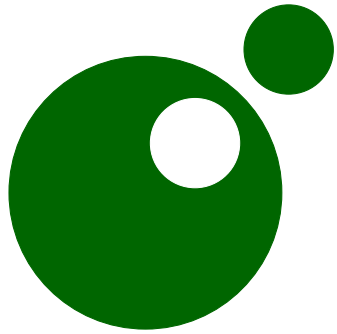
Roberto Giacomelli

Guida al linguaggio Lua per LuaTeX



2020/05/22 — v0.2

Associati anche tu al GUIT

Fai click per associarti

L'associazione per la diffusione di TEX in Italia, riconosciuta ufficialmente in ambito internazionale, si sostiene *unicamente* con le quote sociali.

Se anche tu trovi che questa guida tematica gratuita ti sia stata utile, il mezzo principale per ringraziare gli autori è diventare socio.

Divenendo soci si ricevono gratuitamente:

- l'abbonamento alla rivista ArsTEXnica;
- il DVD TEX Collection;
- un eventuale oggetto legato alle attività del GUIT.

L'adesione al GUIT prevede un quota associativa compresa tra 12,00 € e 70,00 € a seconda della tipologia di adesione prescelta e ha validità per l'anno solare in corso.

Guida al linguaggio Lua per LuaTEX

Questo documento è curato da Roberto Giacomelli.

Indice

Introduzione

Presentazione della guida

Questa guida tematica è dedicata alla programmazione in Lua all'interno dei motori di composizione del sistema TEX. Quel che occorre conoscere è l'ambito in cui ci troviamo per poter avvalerci di funzionalità ben più semplici da codificare che non con gli strumenti tradizionali messi a disposizione dal solo TEX o dalle macro del formato.

Elaborare dati, effettuare calcoli numerici, eseguire compiti in pratica impossibili con il compositore tradizionale come connettersi a database relazionali o avvalersi di avanzati sistemi esterni, sono funzionalità possibili con la nuova generazione di compositori, in scenari applicativi notevolmente ampliati.

Se da un lato è auspicabile che queste potenzialità diventino disponibili per gli utenti finali per mezzo di moduli e pacchetti tali da minimizzare la necessità di programmare, dall'altro è utile fornire dettagli ed esempi per implementare proprie soluzioni e collaborare condividendo idee di sviluppo.

Sono certamente molte le cose da conoscere: un nuovo linguaggio molto diverso da TEX, numerosi dettagli sul funzionamento interno dei compositori Lua-powered, nuovi problemi di organizzazione del codice. Per questo, ho pensato di contribuire con questa guida tentando di illustrare la cresciuta complessità del sistema.

Tra gli argomenti della guida ci sono:

- la differenza tra motore e formato di composizione,
- tecniche di programmazione e di rappresentazione dei dati,
- l'interazione con lo stato interno del motore di composizione.

A mio parere se si vuole puntare sullo sviluppo razionale di progetti documentali non bisogna mai dimenticare TEX, anzi occorre conoscerlo

più a fondo per riuscire ad equilibrarne le componenti in azione per la costruzione di documenti di qualità.

Auguro perciò ai lettori Happy LuaTeXing!

Lua, proprio un bel nome

Lua è un linguaggio semplice ma non banale. Il suo ambito di applicazione è quello dei linguaggi di scripting: text processing, manutenzione del sistema, elaborazioni su file dati, eccetera e lo si può anche trovare come linguaggio embedded di programmi complessi come i videogiochi o altri applicativi che danno la possibilità di essere programmati con esso dall'utente.

Lua è stato ideato da un gruppo di programmatori esperti dell'Università Cattolica di Rio de Janeiro in Brasile. "Lua" si pronuncia LOO-ah e significa "Luna" in portoghese!

Piano della guida

La guida è divisa in due parti: la prima tratta delle basi del linguaggio Lua e la seconda le applica in esempi concreti con l'uso delle librerie interne del compositore.

La risorsa principale per imparare Lua a cui si rimanda per tutti gli approfondimenti è certamente il PIL acronimo del titolo del libro *Programming In Lua* di Roberto Ierusalimschy, principale Autore di Lua. Questo testo non solo è completo e autorevole ma è anche ben scritto e composto[1].

Quanto a LuaTEX il riferimento è il suo manuale che, come quasi tutta la documentazione nel sistema TEX, può essere visualizzato a video con il comando da terminale:

```
$ texdoc luatex
```

Contribuire alla guida

Spero che i lettori vorranno contribuire al testo inviando le propri soluzioni o nuovi contributi anche piccoli. Lo si può fare attraverso lo

[1] Tra l'altro il libro ufficiale su Lua viene composto in LATEX e commercializzato per contribuire allo sviluppo del linguaggio stesso.

strumento che preferite, scrivendomi un messaggio email, oppure utilizzando il repository `git` dei sorgenti.

Origine della guida

Per illustrare i concetti del linguaggio ho preso spunto da un breve corso su Lua che scrissi qualche tempo fa per il blog Lubit Linux di Luigi Iannoccaro che mi propose di realizzare un progetto di divulgazione su Lua. Luigi ha acconsentito all'utilizzo di quegli appunti per produrre questa guida tematica.

Come eseguire gli esercizi

Per iniziare è certamente molto utile eseguire noi stessi esempi ed esercizi di programmazione. Nella guida ne trovate alcuni alla fine di ciascun capitolo della parte prima.

Questa sezione vi introduce brevemente al programma texlua che già trovate compreso in ogni recente distribuzione TEX. Si tratta dell'interprete Lua controparte di `luatex` nell'esecuzione del codice Lua come suggerisce il nome.

Rispetto all'interprete lua standard esso non comprende la modalità interativa detta REPL[2] con cui si digita una linea di codice alla volta senza dover creare un file per fare semplici prove.

Il codice andrà memorizzato in un file con estensione `.lua`, in questo modo: in un file `primo.lua` digitiamo questa unica riga di codice:

```
print("Hello World!")
```

apriamo una finestra di terminale[3] e lanciamo il comando:

```
$ texlua primo.lua
```

Ora che sappiamo come eseguire codice Lua, concentriamoci con i prossimi capitoli sulle basi del linguaggio. Torneremo nella seconda parte della guida su ulteriori modalità di esecuzione e a conoscere importanti dettagli sull'esecuzione di Lua all'interno dei motori di composizione.

[2] Read–eval–print loop.

[3] Maggiori dettagli per diversi sistemi operativi sulla linea di comando possono essere trovati nella guida tematica dedicata scaricabile dal sito GuIT.

Parte I

Fondamenti

Assegnazione e tipi predefiniti 1

1.1 L'assegnamento

Ci occupiamo ora di uno degli elementi di base dei linguaggi informatici: l'istruzione di *assegnamento*. Con questa operazione viene introdotto un *simbolo* nel programma associandolo a un valore che apparterrà a uno dei possibili *tipi* di dato.

La sintassi di Lua non sorprende: a sinistra compare il nome della variabile e a destra l'espressione che fornirà il valore da assegnare al simbolo. Il carattere di '`=`' funge da separatore:

```
a = 123
```

Durante l'esecuzione di questo codice, Lua determina dinamicamente il tipo del valore letterale '`123`' — un numero — creandolo in memoria col nome di '`a`'.

L'istruzione di assegnamento omette il tipo di dato non essendone prevista una dichiarazione esplicita. In altre parole, i dati hanno un tipo, ma ciò entra in gioco solamente a tempo di esecuzione.

Altro concetto importante di Lua è che le variabili sono tutte globali a meno che non si dichiari il contrario.

1.1.1 Locale o globale?

Una proprietà dell'assegnamento è che se non diversamente specificato Lua istanzia i simboli nell'ambiente globale del codice in esecuzione. Se si desidera creare una variabile locale rispetto al blocco di codice in cui è definita, occorre premettere alla definizione la parola chiave `local`.

Le variabili locali evitano alcuni errori di programmazione e in Lua rendono il codice più veloce. Le useremo *sempre* quando un simbolo appartiene in modo semantico a un blocco, per esempio al corpo di una funzione[1].

Se si crea una variabile locale con lo stesso nome di una variabile globale quest'ultima viene *oscurata* e il suo valore sarà protetto da modifiche fino a che il blocco in cui è definita la variabile locale non termina.

1.1.2 Assegnazioni multiple

In Lua possono essere assegnate più variabili alla volta nella stessa istruzione. Questo significa che l'assegnamento è in realtà più complesso di quello presentato fino a ora perché è possibile scrivere una lista di variabili separate da virgole che assumeranno i valori corrispondenti della lista di espressioni, sempre separate da virgole che compare dopo il segno di uguale:

```
local a, b = 0.45 + 0.23, "text"
```

Quando il numero delle variabili non corrispondono a quello delle espressioni, Lua assegnerà automaticamente valori `nil` o ignorerà le espressioni in eccesso. Per esempio:

```
local a, b, c = 0.45, "text"    -- 'c' vale nil
print(a, b, c)
local x, y = "op", "qw", "lo"   -- "lo" è un dato ignorato
print(x, y)
```

Nell'assegnazione Lua prima valuta le espressioni a destra e solo successivamente crea le rispettive variabili secondo l'ordine della lista. Perciò per scambiare il valore di due variabili, operazione chiamata *switch*, è possibile scrivere semplicemente:

```
x, y = y, x
```

Un ulteriore esempio di assegnazione multipla è il seguente, a dimostrazione che le espressioni della lista a destra vengono prima valutate e solo dopo assegnate alle corrispondenti variabili nella lista di sinistra:

[1] Da notare che in sessione interattiva, ovvero in modo REPL dell'interprete Lua, ogni riga è un blocco quindi le variabili locali non sopravvivono alla riga successiva. Perciò in questa modalità useremo solo le variabili globali.

```
local pi = 3.14159
local r = 10.8 -- raggio del cerchio
-- grandezze cerchio
local diam, circ, area = 2*r, 2*pi*r, pi*r^2
-- stampa grandezze
print("Diametro:", diam)
print("Circonferenza:", circ)
print("Area:", area)
```

```
> Diametro:     21.6
> Circonferenza:        67.858344
> Area: 366.4350576
```

Le assegnazioni multiple sono interessanti ma sembra non siano così importanti, possiamo infatti ricorrere ad assegnazioni singole. Diverranno invece molto utili con le funzioni e con gli iteratori di cui ci occuperemo in seguito.

1.2 Una manciata di tipi

In Lua esistono una manciata di tipi. Essenzialmente, omettendone due, sono solo questi:

- `number` il tipo numerico[2];
- `string` il tipo stringa;
- `boolean` il tipo booleano;
- `table` il tipo tabella;
- `nil` il tipo nullo;
- `function` il tipo funzione.

Il breve elenco suscita due osservazioni: tranne la tabella non esistono tipi strutturati mentre le funzioni hanno il rango di tipo.

Questo fa capire molto bene il carattere di Lua: da un lato l'essenzialità ha ridotto all'indispensabile i tipi predefiniti nel linguaggio, ma dall'altro ha spinto all'inclusione di concetti intelligenti e potenti.

[2]Solamente dalla versione 5.3 di Lua vengono internamente distinti gli interi e i numeri in virgola mobile

1.2.1 Il tipo `nil`

Uno dei concetti più importanti che caratterizzano un linguaggio di programmazione è la presenza o meno del tipo nullo. In Lua esiste e viene chiamato `nil`. Il tipo nullo ha un solo valore possibile, anch'esso chiamato `nil`. Il nome è così sia l'unico valore possibile che il tipo.

Leggere una variabile non istanziata non è un errore perché Lua restituisce semplicemente `nil`, mentre assegnare il valore nullo a una variabile la distrugge:

```
print(z)      --> stampa nil, la variabile 'z' non esiste
local z = 123 --> assegnamento di un tipo numerico
print(z)      --> stampa 123
z = nil       --> distruzione della variabile
```

I dati non più utili come quelli la cui variabile è stata riassegnata a `nil` oppure quelli locali nel momento in cui escono di scopo, vengono automaticamente eliminati dal *garbage collector* di Lua. Questo componente solleva l'utente dalla gestione diretta della memoria al prezzo di una piccola diminuzione delle prestazioni.

Certamente se è un garbage collector a liberare dietro le quinte la memoria non più utilizzata, il programma non conterrà gli errori tipici della gestione manuale, ma sarà un po' più lento a runtime.

1.3 Esercizi

esercizio 1 Scrivere il codice Lua che instanzi due variabili `x` e `y` al valore 12.34. Si assegni alle altre due variabili `sum` e `prod` la somma e il loro prodotto delle prime. Si stampi in console i risultati.

esercizio 2 Scrivere il codice Lua che dimostri che modificare una variabile locale non modifica il valore della variabile globale con lo stesso nome. Suggerimento: utilizzare la coppia `do`/`end` per creare un blocco di codice con le proprie variabili locali.

La tabella 2

In questo capitolo parleremo della *tabella*, l'unico tipo strutturato predefinito di Lua. Diamone subito la definizione: la tabella è un *dizionario* cioè l'insieme non ordinato di coppie chiavi/valore e, allo stesso tempo, anche un *array* cioè una sequenza ordinata di valori.

In Lua ne è previsto quindi un uso molteplice: se le chiavi sono numeri interi la tabella sarà un array, se le chiavi sono di altro tipo, per esempio stringhe, avremo un dizionario.

Le chiavi possono essere di tutti i tipi previsti da Lua tranne che `nil`. I valori possono appartenere a qualsiasi tipo. Nulla vieta che in una stessa tabella coesistano chiavi di tipo diverso.

Dal punto di vista sintattico, una tabella di Lua è un oggetto racchiuso tra parentesi graffe e, la più semplice quella vuota, si crea così:

```
local t = {} -- una tabella vuota
```

Per assegnare e ottenere il valore associato a una chiave si utilizzano le parentesi quadre, l'operatore di indicizzazione, ecco un esempio:

```
local t = {}
t["key"] = "val"
print(t["key"]) --> stampa "val"
```

Stando alla definizione che abbiamo dato, una tabella può avere chiavi anche di tipo differente, e infatti è proprio così e ciò vale anche per i valori. In questo esempio una tabella ha chiavi di tipo numerico e di tipo stringa con valori a sua volta di tipo diverso:

```
local t = {}
t["key"] = 123
```

```
t[123] = "key"
print(t["key"]) --> stampa il tipo numerico 123
print(t[123])   --> stampa il tipo stringa "key"
```

2.1 La tabella è un oggetto

Cosa significa che la tabella di Lua è un oggetto? Vuol dire che la tabella è un dato in memoria gestito con un riferimento. In conseguenza, se si copia una variabile a una tabella in una seconda variabile questa farà riferimento ancora alla stessa tabella e non una sua copia:

```
local t = {}
t[1], t[2] = 10, 20
-- copia la tabella o il riferimento?
local other = t
t[1] = t[1] + t[2] -- modifichiamo t
-- l'altra variabile riflette la modifica?
assert(t[1] == other[1])
```

Con la funzione `assert()` si può esprimere l'equivalenza logica tra due espressioni. Essa ritorna l'argomento se questo è `true` oppure se non è `nil`, altrimenti termina l'esecuzione del programma riportando l'errore descritto eventualmente da un secondo argomento testuale.

Il fatto che la tabella è un oggetto è la premessa fondamentale per la programmazione a oggetti in Lua e per scrivere codice più compatto nelle elaborazioni su tabelle dalla struttura molto complessa.

In effetti possiamo annidare in una tabella ulteriori tabelle assegnandole come valore a corrispondenti chiavi, con complessità arbitraria. In altri termini una tabella può rappresentare una struttura ad albero senza limiti teorici. Poiché essa è gestita attraverso un riferimento nell'albero vi saranno solamente i corrispondenti riferimenti mentre il dato effettivo sarà presente in qualche altra parte della memoria.

2.2 Il costruttore e la dot notation

Dunque la tabella è un tipo di dato molto flessibile, è un oggetto, ed è sufficientemente efficiente. Può essere usata in moltissime diverse situazioni ed è ancora più utile grazie all'efficacia del suo *costruttore*.

Ispirato al formato di dati bibliografici di BibTeX, uno dei programmi storici del sistema TEX per la gestione delle bibliografie nei documenti LATEX, il costruttore di Lua può creare tabelle da una sequenza di chiavi/valori inserite tra le parentesi graffe:

```
local t = { a = 123, b = 456, c = "valore"}
```

La chiave appare come il nome di una variabile ma in realtà nel costruttore essa viene interpretata come una chiave di tipo stringa. Così l'esempio precedente è equivalente al seguente codice:

```
-- codice equivalente
local t = {}
t["a"] = 123
t["b"] = 456
t["c"] = "valore"
```

La notazione del costruttore non ammette l'utilizzo diretto di chiavi numeriche. Se occorrono è necessario utilizzare le parentesi quadre per racchiudere il numero che fa da indice:

```
-- chiavi numeriche nel costruttore?
local t_error = { 20 = 123 }
local t_ok = { [20] = 123 }
```

Invece, se nel costruttore omettiamo le chiavi, otteniamo una tabella array con indici interi impliciti in sequenza a partire da 1, contrariamente alla maggior parte dei linguaggi dove l'indice comincia da 0. Ecco un esempio:

```
local t = { 30, 8, 500 }
print(t[1] + t[2] + t[3]) --> stampa 538
```

Non è tutto. L'efficacia sintattica del costruttore è completata dalla *dot notation*, valida solamente per le chiavi di tipo stringa: il campo di una chiave di tipo stringa si indicizza scrivendone la chiave dopo il nome del riferimento della tabella, separato dal carattere `.`:

```
local t = { chiave = "123" }
assert(t.chiave == t["chiave"])
```

Prestate attenzione perché all'inizio si può male interpretare il risultato del costruttore della tabella se unito alla dot notation:

```
local chiave = "ok"
local t = { ok = "123"} -- t.ok == t[chiave]

-- attenzione!
local k = "ok"
print( t.k ) --> stampa nil: "k" non è definita in t
print( t[k]) --> stampa "123"
-- t[k] == t["ok"] == t.ok
-- t.k diverso da t[k] !!!
```

Non confondete il nome di variabile con il nome del campo in dot notation!

Riassumendo, indicizzare una tabella con una variabile restituisce il valore associato alla chiave uguale al valore della variabile, mentre indicizzare in dot notation con il nome uguale a quello della variabile restituisce il valore associato alla chiave corrispondente alla stringa del nome.

2.3 Esercizi

esercizio 1 Scrivere il codice Lua che memorizzi in una tabella i primi 10 numeri primi usando il costruttore.

esercizio 2 Utilizzando la dot notation è possibile utilizzare caratteri spazio nel nome della chiave delle tabelle?

esercizio 3 Scrivere il codice Lua che stampi il valore associato alle chiavi `paese` e `codice`, e il numero medio di comuni per regione, per la seguente tabella. Stampare inoltre il numero di abitanti della capitale.

```
local t = {
    paese = "Italia",
    lingua = "italiano",
    codice = "IT",
    regioni = 20,
    province = 110,
    comuni = 8047,
```

```
    capitale = {"Roma", "RM", abitanti = 2753000},
}
```

Costrutti di base 3

3.1 Il ciclo `for` e il condizionale `if`

Cominciamo con il contare i numeri pari contenuti in una tabella che funziona come un array, ricordandoci che gli indici partono da 1 e non da 0. Rileggete il capitolo precedente come utile riferimento.

Creiamo la tabella con il costruttore in linea e iteriamo con un ciclo `for`:

```
-- costruttore (l'ultima virgola è opzionale)
-- i doppi trattini rappresentano un commento
-- come // lo sono per il C
local t = {
    12, 45, 67, 101,   3,
     2, 89, 36,   7,  99,
    88, 33, 17,  12, 203,
    46,  1, 19,  50, 456,
}

local c = 0 -- contatore
for i = 1, #t do
    if t[i] % 2 == 0 then
        c = c + 1
    end
end
print(c)
```

```
> 8
```

Il corpo del ciclo `for` di Lua è il blocco compreso tra le parole chiave obbligatorie `do` ed `end`. La variabile `i` interna al ciclo assumerà i valori da 1 fino al numero di elementi della tabella, ottenuto con l'operatore lunghezza `#` valido anche per le stringhe.

Per ciascuna iterazione con il costrutto condizionale `if` incrementeremo un contatore solo nel caso in cui l'elemento della tabella è pari. L'`if` ha anch'esso bisogno di definire il blocco di codice e lo fa con le parole chiavi obbligatorie `then` ed `end`, mentre `else` o `elseif` sono rami di codice facoltativi.

Il controllo di parità degli interi si basa sull'operatore modulo, resto della divisione intera `%`. Infatti un numero pari è tale quando il resto della divisione per `2` è zero.

L'operatore di *uguaglianza* è il doppio carattere di uguale `==` e quello di *disuguaglianza* è la coppia dei segni tilde e uguale `~=`. Naturalmente funzionano anche gli operatori di confronto `>`, `>=` e `<`, `<=`.

3.1.1 Operatore di lunghezza

Ma come si comporta l'operatore di lunghezza `#` per le tabelle array con indici non lineari? Per esempio, qual è il risultato del seguente codice:

```
local t = {}
t[1] = 1
t[2] = 2
t[1000] = 3
print(#t)
```

e in questo caso cosa verrà stampato?

```
local t = {}
t[1000] = 123
print(#t)
```

Avrete certamente capito che l'operatore `#` tiene conto solamente degli elementi con indici consecutivi a cominciare da `1` e s'interrompe quando incontra `nil`. Infatti, l'operatore di lunghezza `#` considera per le tabelle il valore `nil` di un indice come termine dell'array. L'operatore è usato molto spesso per inserire progressivamente elementi:

```
local t = {}
for i = 1, 100 do
    t[#t+1] = i*i
end
```

```
print(t[100])
```

3.2 Il ciclo while

Passiamo a scrivere il codice per inserire in una tabella i fattori primi di un numero. Fatelo per esercizio e poi confrontate il codice seguente che utilizza l'operatore modulo `%`:

```
local factors = {}
local n = 123456789

local div = 2
while n > 1 do
    if n % div == 0 then
        factors[#factors + 1] = div
        n = n / div
        while n % div == 0 do
            n = n / div
        end
    end
    div = div + 1
end

for i= 1, #factors do
    print(factors[i])
end
```

```
> 3
> 3607
> 3803
```

Così abbiamo introdotto anche il ciclo `while` perfettamente coerente con la sintassi dei costrutti visti fino a ora: il blocco di codice ripetuto fino a che la condizione è vera, è obbligatoriamente definito da due parole chiave, quella di inizio è `do` e quella di fine è `end`.

Le variabili definite come locali nei blocchi del ciclo `for`, nei rami del condizionale `if` e nel ciclo `while`, non sono visibili all'esterno.

3.3 Intermezzo

In Lua non è obbligatorio inserire un carattere delimitatore sintattico ma è facoltativo il ;. I caratteri spazio, tabulazione e ritorno a capo vengono considerati dalla grammatica come separatori, perciò si è liberi di formattare il codice come si desidera inserendo per esempio più istruzioni sulla stessa linea. Solitamente non si utilizzano i punti e virgola finali, ma se ci sono due assegnazioni sulla stessa linea — stile sconsigliabile perché poco leggibile — li si può separare almeno con un ;. Come sempre una forma stilistica chiara e semplice vi aiuterà a scrivere codice più comprensibile anche a distanza di tempo.

Generalmente è buona norma definire le nuove variabili il più vicino possibile al punto in cui verranno utilizzate per la prima volta, un beneficio per la comprensione ma anche per la correttezza del codice perché può evitare di confondere i nomi e magari di introdurre errori.

3.4 Il ciclo for con il passo

Provate a scrivere il codice Lua che verifica se un numero è *palindromo*, ovvero che gode della proprietà che le cifre decimali sono simmetriche come per esempio avviene per il numero 123321. Confrontate poi questa soluzione:

```
local digit = {}
local n = 123321

local num = n
while num > 0 do
    digit[#digit + 1 ] = num % 10
    num = (num - num % 10) / 10
end

local sym_n, dec = 0, 1
for i=#digit,1,-1 do
    sym_n = sym_n + digit[i]*dec
    dec = dec * 10
end

print(sym_n == n)
```

```
> true
```

La soluzione utilizza una tabella per memorizzare le cifre in ordine inverso del numero da verificare, che vengono poi utilizzate successivamente nel ciclo `for` dall'ultima — la cifra più significativa — fino alla prima per ricalcolare il valore. Se il numero iniziale è palindromo allora il corrispondente numero a cifre invertite è uguale al numero di partenza.

Nel ciclo `for` il terzo parametro opzionale -1 imposta il passo per la variabile `i` che quindi passa dal numero di cifre del numero da controllare (6 nel nostro caso) a 1.

In effetti non è necessaria la tabella:

```
local n = 123321

local num, sym_n, dec = n, 0, 1
while num > 0 do
    sym_n = sym_n + (num % 10)*dec
    dec = 10 * dec
    num = (num - num % 10) / 10
end
print(sym_n == n)
```

```
> true
```

3.5 IF A RAMI MULTIPLI

Il prossimo problema è il seguente: determinare il numero di cifre di un intero. Ancora una volta, confrontate il codice proposto solo dopo aver cercato una vostra soluzione.

```
local n = 786478654
local digits
if n < 10 then
    digits = 1 -- attenzione non 'local digits = 1'
elseif n < 100 then
    digits = 2
elseif n < 1000 then
    digits = 3
elseif n < 10000 then
    digits = 4
elseif n < 100000 then
```

```
    digits = 5
elseif n < 1000000 then
    digits = 6
elseif n < 10000000 then
    digits = 7
elseif n < 100000000 then
    digits = 8
elseif n < 1000000000 then
    digits = 9
elseif n < 10000000000 then
    digits = 10
else -- fermiamoci qui...
    digits = 11
end

print(digits)
```

Questo esempio mostra in azione l'`if` a più rami che in Lua svolge la funzione del costrutto `switch` presente in altri linguaggi, con una nuova parola chiave: `elseif`.

L'esempio è interessante anche per come viene introdotta la variabile `digits`, cioè senza inizializzarla per poi assegnarla nel ramo opportuno dell'`if`. Infatti una variabile interna a un blocco non sopravvive oltre, per questo motivo dichiararla all'interno dell'`if` non è sufficiente.

Come è necessario *non* premettere `local` nelle assegnazioni nei rami del condizionale: in questo caso verrebbe creata una nuova variabile locale al blocco che *oscurerebbe* quella esterna con lo stesso nome. In altre parole, al termine del condizionale `digits` varrebbe ancora `nil`, il valore che assume nel momento della dichiarazione.

3.6 Esercizi

esercizio 1 Contare quanti interi sono divisibili sia per 2 che per 3 nell'intervallo $[1, 10\,000]$. Suggerimento: utilizzare l'operatore modulo `%`, resto della divisione intera tra due operandi.

esercizio 2 Determinare i fattori del numero intero 5 461 683 modificando il codice riportato alla sezione 3.2 per includerne la molteplicità.

esercizio 3 Instanziare la tabella seguente con tre tabelle/array di tre numeri e calcolarne il determinante della matrice corrispondente.

```
local t = {
    { 0,  5, -1},
    { 2, -2,  0},
    {-1,  0,  1},
}
```

esercizio 4 Data la tabella seguente stampare in console il conteggio dei numeri pari e dei numeri dispari contenuti in essa. Verificare che la somma di questi due conteggi sia uguale alla dimensione della tabella.

```
local t = {
    45, 23, 56, 88, 96, 11,
    80, 32, 22, 85, 50, 10,
    32, 75, 10, 66, 55, 30,
    10, 13, 23, 91, 54, 19,
    50, 17, 91, 44, 92, 66,
    71, 25, 19, 80, 17, 21,
    81, 60, 39, 15, 18, 28,
    23, 10, 18, 30, 50, 11,
    50, 88, 28, 66, 13, 54,
    91, 25, 23, 17, 88, 90,
    85, 99, 22, 91, 40, 80,
    56, 62, 81, 71, 33, 30,
    90, 22, 80, 58, 42, 10,
}
```

esercizio 5 Data la tabella precedente, scrivere il codice per costruire una seconda tabella uguale alla prima ma priva di duplicati e senza alterare l'ordine degli interi.

esercizio 6 Data la tabella precedente costruire una tabella le cui chiavi siano i numeri contenuti in essa e i valori siano il corrispondente numero di volte che la chiave stessa compare nella tabella di partenza. Stampare poi in console il numero che si presenta il maggior numero di volte.

Operatori logici 4

In Lua un'espressione è vera se essa corrisponde al valore booleano `true` oppure a un valore che non è `nil`.

Gli operatori logici `and`, `or` e `not` danno luogo ad alcune espressioni idiomatiche di Lua. Cominciamo con `or`: è un operatore logico binario. Se il primo operando è vero lo restituisce altrimenti restituisce il secondo. Per esempio nel seguente codice `a` vale 123.

```
local a = 123 or "mai assegnato"
```

L'operatore `and` — anche questo binario come `or` — restituisce il primo operando se esso è falso altrimenti restituisce il secondo operando.

Con `and` e `or` combinati otteniamo l'operatore ternario del C++ in Lua: Ecco l'espressione in un esempio: se `a` è vera il risultato è `b` altrimenti `c`:

```
local val = (a and b) or c
```

Poiché `and` ha priorità maggiore rispetto a `or` nell'espressione precedente possiamo omettere le parentesi per un codice ancor più idiomatico:

```
local val = a and b or c -- a ? b : c del C++
```

Il massimo tra due numeri è un'espressione condizionale:

```
local x, y = 45.69, 564.3
local max
if x > y then
    max = x
else
    max = y
end
```

ma con gli operatori logici è tutto più Lua:

```
local x, y = 45.69, 564.3
local max = (x > y) and x or y
```

L'operatore logico `not` restituisce `true` se l'operando è `nil` oppure se è `false` e, viceversa, restituisce `false` se l'operando non è `nil` oppure è `true`. Alcuni esempi:

```
print(not 5)        --> 'false'
print(not not 5)    --> 'true'
print(not true)     --> 'false'
print(not false)    --> 'true'
print(not nil)      --> 'true'
```

L'operatore di negazione può essere usato per controllare se una variabile è valida oppure no. Per esempio possiamo controllare se in una tabella esiste il campo `prezzo`:

```
local t = {} -- una tabella vuota
if not t.prezzo then -- t.prezzo è nil
    print("assente")
else
    print("presente")
end

t.prezzo = 12.00
if not t.prezzo then
    print("assente")
else
    print("presente")
end
```

4.1 Esercizi

esercizio 1 Prevedere il risultato delle seguenti espressioni Lua:

```
local a = 1 or 2
```

```
local b = 1 and 2
local c = "text" or 45

local d = not 12 or "ok"
local e = not nil or "ok"
```

esercizio 2 Nel seguente codice, se il valore del primo condizionale è `true` cosa stamperà invece il secondo condizionale?

```
if "stringa" then print "it's not 'nil'" end
if "stringa" == true then
    print("it's 'true'")
else
    print("it's not 'true'")
end
```

esercizio 3 Come distinguere se una variabile contiene il valore `false` o il valore `nil`?

esercizio 4 Usando gli operatori logici di Lua codificare l'espressione che restituisce la stringa `più grande di 100`, `uguale` o `più piccolo di 100` a seconda del valore numerico fornito.

Il tipo stringa 5

In Lua le stringhe rappresentano uno dei tipi di base del linguaggio. Per rappresentare valori stringa letterali ci sono tre diversi delimitatori:

- doppi apici: carattere ;
- apice semplice: carattere ';
- doppie parentesi quadre: delimitatori `[[` e `]]` con o senza un numero corrispondente di caratteri `=`, per esempio `[==[` e `]==]`.

In una stringa delimitata da doppi apici possiamo inserire liberamente apici semplici e viceversa, e caratteri non stampabili come il ritorno a capo (`\n`) e la tabulazione (`\t`), tramite il carattere di escape backslash che quindi va inserito esso stesso come doppio backslash (\\):

```
local s1 = "doppi 'apici'"
local s2 = 'apici semplici e non "doppi"'
local s3 = "prima riga\nseconda riga"
local s4 = "una \\macro"
local s5 = "\"" -- o anche '"'

print(s1)
print(s2)
print(s3)
print(s4)
print(s5)
```

```
> doppi 'apici'
> apici semplici e non "doppi"
> prima riga
> seconda riga
> una \macro
> "
```

In Lua non esiste il tipo carattere quindi gli Autori del linguaggio hanno pensato di utilizzare i delimitatori normalmente destinati a rappresentarne la forma letterale, per consentire all'utente di creare stringhe contenenti i delimitatori stessi, senza utilizzare l'escaping.

Sono comunque ammessi i simboli \ e \' che rappresentano i caratteri corrispondenti, come si vede nella variabile `s5` del codice precedente.

Il terzo tipo di delimitatore per le stringhe è una coppia si parentesi quadre e ha la proprietà di ammettere il ritorno a capo. Si possono così introdurre nel sorgente interi brani di testo nel quale i caratteri di escaping non saranno interpretati.

```
local long_text = [[
Questo è un testo multiriga
dove i caratteri di escape non contano
come \n o \" o \' o \\.

Inoltre, se il testo come in questo caso
comincia con un ritorno a capo allora questo
carattere \n sarà ignorato.
]]
print(long_text)
```

```
> Questo è un testo multiriga
> dove i caratteri di escape non contano
> come \n o \" o \' o \\.
>
> Inoltre, se il testo come in questo caso
> comincia con un ritorno a capo allora questo
> carattere \n sarà ignorato.
>
```

Se per caso nel testo fossero presenti i delimitatori di chiusura è possibile inserire un numero qualsiasi di caratteri = tra le parentesi quadre, purché il numero sia lo stesso per i delimitatori di apertura e chiusura, esempio:

```
local long_text = [=[
Questo è il codice Lua da stampare:

local tab = {10, 20, 30}
local idx = {3, 2, 1}
print(tab[idx[1]]) -- ops due parentesi quadre
```

```
local long_text = [[ -- questo non potremo farlo...
    Testo lungo...
]]

Tutto chiaro?
In Lua le stringhe letterali nel codice
possono essere proprio letterali
senza caratteri di escape e senza
preoccupazioni sulla presenza di gruppi
di delimitazione di chiusura...
]=]

print(long_text)
```

5.1 Commenti multiriga

Questi delimitatori variabili con numero qualsiasi di segni = li troviamo anche nei commenti multiriga di Lua. Abbiamo incontrato fino a ora i commenti di riga che si introducono nel codice con un doppio trattino --.

I commenti multiriga sono comodi quando si vuol escludere dall'esecuzione un intero blocco di righe: iniziano con i doppi trattini seguiti da un delimitatore di stringa multiriga e terminano con la corrispondente chiusura:

```
-- questo è un commento di riga

--[[
questo è un commento
multiriga
]]

--[=[
e anche questo è un commento
multiriga
]=]
```

Normalmente in Lua i commenti multiriga vengono chiusi premettendo i doppi trattini anche al gruppo delimitatore di chiusura. Questo è solo un trucco per riattivare rapidamente il codice eventualmente contenuto nel

commento, basta uno spazio per far trasformare il commento multiriga in uno semplice:

```
--[[ righe di codice non attive
local tab = {}
--]]

-- [[ notare lo spazio dopo i doppi trattini
-- questo codice invece viene eseguito

local tab = {}
--]] -- e questo diventa una normale riga di commento
```

5.2 Concatenazione stringhe e immutabilità

In Lua l'operatore `..` concatena due stringhe, in questo modo:

```
local s1 = "Hello" .. " " .. "world"
local s2 = s1 .. " OK"
s2 = s2 .. "."

print(s1 .. "!")
print(s2)
```

```
> Hello world!
> Hello world OK.
```

Il concetto importante riguardo alle stringhe è se queste siano o no immutabili. Se non lo sono la concatenazione di stringhe non comporta la creazione di una nuova stringa ma la modifica in memoria.

In Lua, come in molti altri linguaggi, le stringhe sono invece immutabili. Ciò significa che una volta create, le stringhe non possono essere modificate e nel codice precedente, l'operazione di concatenare il carattere punto in coda alla stringa `s2`, genera una nuova stringa che è assegnata alla stessa variabile.

Per poche operazioni di concatenazione ciò non è un problema, ma in alcuni casi invece si. Consideriamo il seguente codice apparentemente innocuo:

```
local s = ""

for i = 1, 100 do
    s = s .. "**"
end
print(#s) -- # funziona anche per le stringhe!
```

Ma cosa succede in dettaglio? Perché questo codice non è efficiente? Ad ogni concatenazione viene creata una nuova stringa. La prima volta vengono copiati due byte per dare la stringa `**`. La seconda iterazione la memoria copiata sarà di 4 byte, e alla terza di 6 byte, eccetera.

A ogni iterazione la memoria copiata cresce di due byte con il risultato che per produrre una stringa di 200 asterischi (200 byte) avremo copiato in totale la memoria equivalente a 10100 byte!

In Java e negli altri linguaggi con stringhe immutabili normalmente si corre ai ripari mettendo a disposizione una struttura dati o una funzione che risolve il problema, per esempio un tipo `StringBuffer`. In Lua la soluzione è una funzione della libreria `table` che, anticipando rispetto alle nostre chiaccherate è `table.concat()`:

```
local t = {}
for i = 1, 100 do
    t[#t + 1] = "**"
end

print(#table.concat(t))
```

Nel caso specifico avremo dovuto usare la funzione `string.rep()` anche se `table.concat()` è più generale.

5.3 Esercizi

esercizio 1 Come fare in Lua per creare una stringa letterale contenente sia il carattere apice semplice che doppio?

esercizio 2 Quale sarà il risultato dell'esecuzione del seguente codice?

```
local s = "'"..'"'.."ok"..[["']]
print(s)
```

esercizio 3 Creare la stringa \/.

esercizio 4 Scrivere un programma che a partire dalla stringa * crei e stampi la stringa di 64 asterischi senza utilizzare l'operatore di concatenazione o la funzione `string.rep()`.

esercizio 5 Scrivere un programma che a partire dalla stringa * crei e stampi la stringa di 64 asterischi usando l'operatore di concatenazione il minimo indispensabile di volte.

Funzioni 6

Le funzioni sono il principale mezzo di astrazione e lo strumento base per strutturare il codice.

Coerentemente con il resto del linguaggio la sintassi di una funzione comprende due parole chiave che servono per delimitare il blocco di codice di definizione: `function` ed `end`. Una funzione può restituire dati tramite la parola chiave `return`.

Come primo esempio, vi presento una funzione per calcolare l'ennesimo numero della serie di Fibonacci. Un elemento si ottiene sommando i due precedenti elementi avendo posto uguale a 1 i primi due:

```
function fibonacci(n)
    if n < 2 then
        return 1
    end

    local n1, n2 = 1, 1
    for i = 1, n-1 do
        n1, n2 = n2, n1 + n2 -- assegnazione multipla
    end
    return n1
end

print(fibonacci(10)) --> 55
```

Con le regole dell'assegnazione multipla una funzione può accettare più argomenti. Se gli argomenti passati sono in eccesso rispetto a quelli che essa prevede, quelli in più verranno ignorati. Se viceversa, gli argomenti sono inferiori a quelli previsti allora a quelli mancanti verrà assegnato il valore `nil`.

Ma questo vale anche per i dati di ritorno quando la funzione è usata come espressione in un'istruzione di assegnamento. Basta inserire dopo l'istruzione `return` la lista delle espressioni separate da virgole che saranno valutate e assegnate alle corrispondenti variabili.

Per esempio, potremo modificare la funzione precedente per restituire la somma dei primi n numeri di Fibonacci oltre che l'ennesimo elemento della serie stessa e considerare un valore di default se l'argomento è `nil`:

```
function fibonacci(n)
    n = n or 10 -- the default value is 10
    if n == 1 then
        return 1, 1
    end

    if n == 2 then
        return 1, 2
    end

    local sum = 1
    local n1, n2 = 1, 1
    for i = 1, n-1 do
        n1, n2 = n2, n1 + n2
        sum = sum + n1
    end
    return n1, sum
end

local fib_10, sum_fib_10 = fibonacci()
print(fib_10, sum_fib_10)
```

```
> 55    143
```

6.1 Funzioni: valori di prima classe, I

In Lua le funzioni sono un tipo. Possono essere assegnate a una variabile e possono essere passate come argomento a un'altra funzione, una proprietà che non si trova spesso nei linguaggi di scripting e che offre una nuova flessibilità al codice.

In realtà in Lua tutte le funzioni sono memorizzate in variabili. Per assegnare direttamente una funzione a una variabile esiste la sintassi anonima:

```
add = function (a, b)
    return a + b
end
print(add(45.4564, 161.486))
```

Essendo le funzioni valori di prima classe ne consegue che in Lua le funzioni sono oggetti senza nome esattamente come lo sono gli altri tipi come i numeri e le stringhe. Inoltre, la sintassi classica di definizione:

```
function variable_name (args)
    -- function body
end
```

è solo *zucchero sintattico* perché l'interprete Lua la tradurrà automaticamente nel codice equivalente in sintassi anonima:

```
variable_name = function (args)
    -- function body
end
```

6.2 Funzioni: valori di prima classe, II

Un esempio di funzione con un argomento funzione è il seguente, dove viene eseguito un numero di volte dato, la stessa funzione priva di argomenti:

```
local function print_five()
    print(5)
end

local function do_many(fn, n)
    for i=1, n or 1 do
        fn()
    end
end

do_many(print_five)
```

```
do_many(print_five, 10)
do_many(function () print("---") end, 12)
```

Molto interessante. Nell'ultima riga di codice l'argomento è una funzione definita in sintassi anonima che verrà eseguita 12 volte.

Per prendere confidenza con il concetto di *funzioni come valori di prima classe*, cambiamo il significato della funzione `print()`:

```
local orig_print = print
print = function (n)
    orig_print("Argomento funzione -> "..n)
end

print(12)
```

6.3 Tabelle e funzioni

Se una tabella può contenere chiavi con qualsiasi valore allora può contenere anche funzioni! Le sintassi previste sono queste, esplicitate con il codice riportato di seguito:

- assegnare la variabile di funzione a una chiave di tabella;
- assegnare direttamente la chiave di tabella con la definizione di funzione in sintassi anonima;
- usare il costruttore di tabelle per assegnare funzioni in sintassi anonima.

```
-- primo caso
local function tipo_i()
    -- body
end

local t = {}
t.func_1 = tipo_i

-- secondo caso
local t = {}
t.func_2 = function ()
    -- body
end
```

```
-- terzo caso con più di una funzione
local t = {
    func_3_i = function ()
        -- body
    end,

    func_3_ii = function ()
        -- body
    end,

    func_3_iii = function ()
        -- body
    end,
}
```

Con questo meccanismo una tabella può svolgere il ruolo di *modulo* memorizzando funzioni utili a un certo scopo. In effetti la libreria standard di Lua si presenta all'utente proprio in questo modo.

6.4 Numero di argomenti variabile

Una funzione può ricevere un numero variabile di argomenti rappresentati da tre dot consecutivi `...`. Nel corpo della funzione i tre punti rappresenteranno la lista degli argomenti, dunque possiamo o costruire con essi una tabella oppure effettuare un'assegnazione multipla.

Un esempio è una funzione che restituisce la somma di tutti gli argomenti numerici:

```
-- per un massimo di 3 argomenti
local function add_three(...)
    local n1, n2, n3 = ...
    return (n1 or 0) + (n2 or 0) + (n3 or 0)
end

-- con tutti gli argomenti
local function add_all(...)
    local t = {...} -- collecting args in a table
    local sum = 0
    for i = 1, #t do
        sum = sum + t[i]
```

```
    end
    return sum
end

print(add_three(40, 20))
print(add_all(45, 48, 5456))
print(add_three(14, 15), add_all(-89, 45.6))
```

Per inciso, anche la funzione base `print()` accetta un numero variabile di argomenti. Il meccanismo è ancora più flessibile perché tra i primi argomenti vi possono essere variabili "fisse". Per esempio il primo parametro potrebbe essere un moltiplicatore:

```
local function add_and_multiply(molt, ...)
    local t = {...}
    local sum = 0
    for i = 1, #t do
        sum = sum + t[i]
    end

    return molt * sum
end

print(add_and_multiply(10, 45.23, 48, 9.36, -8, -56.3))
```

Un'altra funzione predefinita `select()` consente di accedere alla lista degli argomenti in dettaglio. Infatti nel codice precedente, se tra gli argomenti compare un valore `nil` avremo problemi ad accedere ai valori successivi perché — come sappiamo già — l'operatore di lunghezza `#` considera il `nil` come valore sentinella di fine array/tabella.

Il selettore prevede un primo parametro fisso seguito da una lista variabile di valori rappresentata dai tre punti `...`. Se questo parametro è un intero allora verrà considerato come indice per restituire l'argomento corrispondente. Se invece il parametro è la stringa `#` allora la funzione restituisce il numero totale di argomenti, inclusi i `nil`.

Il codice seguente preso pari pari dal PIL ne è un'applicazione:

```
for i = 1, select("#", ...) do
    local arg = select(i, ...)
    -- loop body
end
```

6.5 Omettere le parentesi se...

In Lua esiste la sintassi di chiamata a funzione semplificata che consiste nella possibilità di ommettere le parentesi tonde `()`, ammessa solo se:

- alla funzione si passa un unico argomento di tipo stringa;
- alla funzione si passa un unico argomento di tipo tabella.

Per esempio:

```
print "si è possibile anche questo..."

-- e questo:
local function is_empty(t)
    if #t == 0 then
        return true
    else
        return false
    end
end
print(is_empty{})
print(is_empty{1, 2, 3})

-- invece di questo (sempre possibile):
print(is_empty({}))
print(is_empty({1, 2, 3}))
```

6.6 Closure

Chiudiamo il capitolo parlando con uno strano termine forse meglio noto agli sviluppatori dei linguaggi funzionali: la *closure*.

Questa proprietà di Lua amplia il concetto di funzione rendendo possibile l'accesso dall'interno di essa ai dati presenti nel contesto esterno. Ciò è possibile perché alla chiamata di una funzione viene creato uno spazio di memoria del contesto esterno unico e indipendente.

> *Tutte le chiamate a una stessa funzione condivideranno una stessa closure.*

Se questo è vero una funzione potrebbe incrementare un contatore creato al suo interno, e anche qui prendo l'esempio di codice dal PIL:

```
local function new_counter()
    local i = 0 -- variabile nel contesto esterno
    return function ()
        i = i + 1 -- accesso alla closure
        return i
    end
end

local c1 = new_counter()
print(c1()) --> 1
print(c1()) --> 2
print(c1()) --> 3
print(c1()) --> 4
print(c1()) --> 5

local c2 = new_counter()
print(c2()) --> 1
print(c2()) --> 2
print(c2()) --> 3

print(c1()) --> 6
```

Il codice definisce una funzione `new_counter()` che restituisce una funzione che ha accesso indipendente al contesto (la variabile `i`).

Tecnicamente la closure *è* la funzione effettiva mentre invece la funzione non è altro che il prototipo della closure.

Le closure consentono di implementare diverse tecniche utili in modo naturale e concettualmente semplice. Una funzione di ordinamento potrebbe per esempio accettare come parametro una funzione di confronto per stabilire l'ordine tra due elementi tramite l'accesso a una seconda tabella esterna contenente informazioni utili per l'ordinamento stesso.

Nel prossimo esempio mettiamo in pratica l'idea. Il codice utilizza una funzione della libreria di Lua, che introdurremo nel prossimo capitolo, in particolare `table.sort()`, per applicare l'algoritmo di ordinamento alla tabella passata come argomento in base al criterio di ordine stabilito con la funzione passata come secondo argomento in sintassi anonima.

```
local function sort_by_value(tab)
    local val = {
        [1994] = 12.5,
        [1996] = 10.2,
```

```
        [1998] = 10.9,
        [2000] =  8.9,
        [2002] = 12.9,
    }
    table.sort(tab,
        function (a, b)
            return val[a] > val[b]
        end
    )
end

local years = {1994, 1996, 1998, 2000, 2002}
sort_by_value(years)

for i = 1, #years do
    print(years[i])
end
```

```
> 2002
> 1994
> 1998
> 1996
> 2000
```

6.7 Esercizi

esercizio 1 Scrivere una funzione che sulla base della stringa in ingresso `+`, `-`, `*`, `/` restituisca la funzione corrispondente per due operandi.

esercizio 2 Scrivere la funzione che accetti due argomenti numerici e ne restituisca i risultati delle quattro operazioni aritmetiche.

esercizio 3 Scrivere una funzione che restituisca il fattoriale di un numero memorizzandone in una tabella di closure i risultati per evitare di ripetere il calcolo in chiamate successive con pari argomento.

esercizio 4 Scrivere una funzione con un argomento opzionale rispetto al primo parametro numerico che ne restituisca il seno interpretandolo in radianti se l'argomento opzionale è `nil` oppure `rad`, in gradi sessadecimali se `deg` o in gradi centesimali se `grd`.

esercizio 5 Scrivere una funzione che accetti come primo argomento una funzione $f : \mathbb{R} \to \mathbb{R}$ (prende un numero e restituisce un numero), come secondo e terzo argomento i due valori dell'intervallo di calcolo e come quarto argomento il numero di punti in cui suddividere l'intervallo. La funzione dovrà stampare i valori che la funzione argomento assume nei punti d'ascissa così definiti.

La libreria standard di Lua 7

In Lua sono immediatamente disponibili un folto gruppo di funzioni che ne formano la *libreria standard*. Si tratta di una collezione di funzioni utili a svolgere compiti ricorrenti su stringhe, file, tabelle, eccetera, e si trovano precaricate in una serie di tabelle.

L'elenco completo ma in ordine sparso con il nome della tabella/modulo contenitore e la descrizione applicativa è il seguente:

`math`	matematica;
`table`	utilità sulle tabelle;
`string`	ricerca, sostituzione e pattern matching;
`io`	input/output facility, operazioni sui file;
`bit32`	operazioni bitwise (solo in Lua 5.2);
`os`	date e chiamate di sistema;
`coroutine`	creazione e controllo delle coroutine;
`utf8`	utilità codifica Unicode UTF-8 (da Lua 5.3);
`package`	caricamento di librerie esterne;
`debug`	accesso alle variabili e performance assessment.

La pagina web a questo link fornisce tutte le informazioni di dettaglio sulla libreria standard di Lua 5.3.

7.1 Libreria matematica

Nella libreria memorizzata nella tabella `math` ci sono le funzioni trigonometriche `sin()`, `cos()`, `tan()`, `asin()` eccetera — che come di consueto lavorano in radianti — le funzioni esponenziali `exp()`, `log()`, `log10()`, quelle di arrotondamento `ceil()`, `floor()`, e quelle per la generazione pseudocasuale di numeri come `random()`, e `randomseed()`. Oltre a funzioni, la tabella include campi numerici come la costante π.

Un esempio introduttivo è questo dove nella funzione `one()` viene definita una funzione locale:

```
print(math.pi)
print(math.sin( math.pi/2 ))
print(math.cos(0))

-- accorciamo i nomi delle funzioni ;-)
local pi, sin, cos = math.pi, math.sin, math.cos
local function one(a)
    local square = function (x) return x*x end
    return square(sin(a)) + square(cos(a))
end

for i=0, 1, 0.1 do
    print(i, one(i))
end
```

7.2 Libreria stringhe

La libreria per le stringhe è memorizzata nella tabella `string` ed è una delle più utili. Con essa si possono formattare campi e compiere operazioni di ricerca e sostituzione. In effetti, in Lua non è infrequente elaborare grandi porzioni di testo.

7.2.1 Funzione `string.format()`

La funzione più semplice è quella di formattazione `string.format()`. Essa restituisce una stringa prodotta con il formato definito dal primo argomento dei dati forniti dal secondo argomento in poi.

Il formato è esso stesso specificato come una stringa contenente dei segnaposto creati con il simbolo percentuale e uno specificatore di tipo. Per esempio `%d` indica il formato relativo a un numero intero, dove `d` sta per digit mentre `%f` indica il segnaposto per un numero decimale — `f` sta per float.

I campi formato derivano da quelli della funzione classica di libreria `printf()` del C. Di seguito un esempio di codice:

```
-- "%d" means match a digit
local s1 = string.format("%d + %d = %d", 45, 54, 45+54)
```

```
print(s1)
local s2 = string.format("%06d", 456)
print(s2)

-- "%f" means float
local num = 123.456
local s3 = string.format(
    "intero %d e decimale %0.2f",
    math.floor(num),
    num
)
print(s3)

-- "%s" means string
print(string.format("s1='%s', s2='%s'", s1, s2))

print(string.format("%24s", "pippo"))
```

```
> 45 + 54 = 99
> 000456
> intero 123 e decimale 123.46
> s1='45 + 54 = 99', s2='000456'
>                     pippo
```

Come avete potuto notare nel codice, è anche possibile fornire un ulteriore specifica di dettaglio tra il `%` e lo specificatore di tipo, per esempio per gestire il numero delle cifre decimali.

Per elaborare il testo si utilizza di solito una libreria per le espressioni regolari. Lua mette a disposizione alcune funzioni di sostituzione e *pattern matching* meno complete dell'implementazione dello standard POSIX per le espressioni regolari ma molto spesso più semplici da utilizzare.

Esistono due strumenti di base, il primo è il *pattern* e il secondo è la *capture*.

7.2.2 Pattern

Il pattern è una stringa che può contenere campi chiamati *classi* simili a quelli per la funzione di formato visti in precedenza, che stavolta però si riferiscono al singolo carattere, e questa differenza è essenziale.

La funzione di base che accetta pattern è `string.match()` che restituisce la prima corrispondenza trovata in una stringa primo argomento corrispondente al pattern dato come secondo argomento.

Per esempio, possiamo ricercare in un numero di tre cifre intere all'interno di un testo con il pattern `%d%d%d`:

```
-- semplice pattern in azione
local s = [[
le prime tre cifre decimali di \pi = 3,141592654 sono]]
local pattern = "%d%d%d"
print(string.match(s, pattern))
```

```
> 141
```

Le classi carattere possibili sono le seguenti:

`.` un carattere qualsiasi;
`%a` una lettera;
`%c` un carattere di controllo;
`%d` una cifra;
`%l` una lettera minuscola;
`%u` una lettera maiuscola;
`%p` un carattere di interpunzione;
`%s` un carattere spazio;
`%w` un carattere alfanumerico;
`%x` un carattere esadecimale;
`%z` il carattere rappresentato con il codice 0.

Le classi ammettono quattro modificatori per esprimere le ripetizioni dei caratteri:

`+` indica 1 o più ripetizioni;
`*` indica 0 o più ripetizioni;
`-` come `*` ma nella sequenza più breve;
`?` indica 0 o 1 occorrenza;

Esempio:

```
-- occorrenza di un numero intero
-- come una o più cifre consecutive
print(string.match("l'intero 65 interno", "%d+"))
print(string.match("l'intero 0065 interno", "%d+"))
```

```
-- e per estrarre un numero decimale?
-- il punto è una classe così si utilizza
-- la classe '%.' per ricercare il carattere '.'
print(string.match("num = 45.12 :-)", "%d+%.%d+"))
```

```
> 65
> 0065
> 45.12
```

7.3 Capture

Il pattern può essere arricchito non solo per trovare corrispondenze ma anche per restituirne parti componenti. Questa funzionalità viene chiamata *capture* e consiste semplicemente nel racchiudere tra parentesi tonde le classi.

Per esempio per estrarre l'anno di una data nel formato `dd/mm/yyyy` possiamo usare il pattern con la capture seguente `%d%d/%d%d/(%d%d%d%d)`:

```
-- extract only
local s = "This '10/03/2025' is a future date"
print(string.match(s, "%d%d/%d%d/(%d%d%d%d)"))
```

```
> 2025
```

Più capture ci sono nel pattern e altrettanti argomenti multipli di uscita saranno restituiti:

```
-- extract all
local s = "This '10/03/2025' is a future date"
local d, m, y = string.match(
    s,
    "(%d+%d?)/(%d%d)/(%d%d%d%d)"
)
print(d, m, y)
```

```
> 10    03      2025
```

7.3.1 La funzione string.gsub()

Abbiamo appena cominciato a scoprire le funzionalità dedicate al testo disponibili nella libreria standard di Lua precaricata a runtime.

Diamo solo un altro sguardo alla libreria presentando la funzione `string.gsub()`. Il suo nome sta per *global substitution*, ovvero la sostituzione di tutte le occorrenze in un testo.

Intanto per individuare le occorrenze è naturale pensare di utilizzare un pattern e che sia possibile utilizzare le capture nel testo di sostituzione, per esempio:

```
local s = "The house is black."
print(string.gsub(s, "black", "red"))
print(string.gsub(s, "(%a)lac(%a)", "%2lac%1"))
```

```
> The house is red.     1
> The house is klacb.   1
```

Il primo argomento è la stringa da ricercare, il secondo è il pattern e il terzo è il testo di sostituzione dell'occorrenza, ma può anche essere una tabella dove le chiavi corrispondenti al pattern saranno sostituite con i rispettivi valori, oppure anche una funzione che riceverà le catture e calcolerà il testo da sostituire.

Una funzione quindi assai flessibile. Mi viene in mente questo esercizio: moltiplicare per 12 tutti gli interi in una stringa, ed ecco il codice:

```
local s = "Cose da fare oggi 5, cosa da fare domani 2"
print(string.gsub(s, "%d+", function(n)
    return tonumber(n)*12
end))
```

```
> Cose da fare oggi 60, cosa da fare domani 24  2
```

A questo punto degli esempi avrete certamente capito che `gsub()` restituisce anche il numero delle sostituzioni effettuate.

Tutte queste funzioni restituiscono una stringa costruita ex-novo e non modificano la stringa originale di ricerca. In Lua le stringhe sono dati immutabili.

7.4 Esercizi

esercizio 1 Qual è la differenza tra i campi di formato della funzione `string.format()` e le classi dei pattern? Quali le somiglianze?

esercizio 2 Stampare una data nel formato `dd/mm/yyyy` a partire dagli interi contenuti nelle variabili `d`, `m` e `y`.

esercizio 3 Cosa restituisce l'esecuzione della seguente funzione?

```
string.match("num = .123456 :-)", "%d+%.%d+")
```

Quale pattern corrisponde a un numero decimale la cui parte intera può essere omessa?

esercizio 4 Come estrarre dal nome di un file l'estensione?

esercizio 5 Come eliminare da un testo eventuali caratteri spazio iniziali e/o finali?

esercizio 6 Il pattern `(%d+)/(%d+)/(%d+)` è adatto per catturare giorno, mese e anno di una data presente in una stringa nel formato `dd/mm/yyyy`?

esercizio 7 Creare un esempio che utilizzi `string.gsub()` con una funzione in sintassi anonima a due argomenti corrispondenti a due capture nel pattern di ricerca.

Iteratori 8

Gli iteratori offrono un approccio semplice e unificato per scorrere uno alla volta gli elementi di una collezione di dati. Vi dedicheremo un capitolo proprio perché sono molto utili per scrivere codice efficiente, pulito ed elegante.

Il linguaggio Lua prevede il ciclo d'iterazione *generic for* che introduce la nuova parola chiave `in` secondo questa sintassi:

```
for <lista variabili> in iterator_function() do
-- codice
end
```

Le tabelle di Lua sono oggetti che possono essere impiegati per rappresentare degli array oppure dei dizionari. In entrambe i casi Lua mette a disposizione due iteratori predefiniti tramite le funzioni `ipairs()` e `pairs()`.

Queste funzioni restituiscono un iteratore conforme alle specifiche del generic for. Mentre impareremo più tardi a scrivere iteratori personalizzati, dedicheremo le prossime due sezioni a questi importanti iteratori predefiniti per le tabelle.

8.1 Funzione ipairs()

La funzione `ipairs()` restituisce un iteratore che a ogni ciclo genera due valori: l'indice dell'array e il valore corrispondente. L'iterazione comincincia dalla posizione 1 e termina quando il valore della posizione corrente à `nil`:

```
-- una tabella array
local t = {45, 56, 89, 12, 0, 2, -98}
```

```
-- iterazione tabella come array
for i, n in ipairs(t) do
    print(i, n)
end
```

Il ciclo con `ipairs()` è equivalente a questo codice:

```
-- una tabella array
local t = {45, 56, 89, 12, 0, 2, -98}
do
    local i, v = 1, t[1]
    while v do
        print(i, v)
        i = i + 1
        v = t[i]
    end
end
```

Se non interessa il valore dell'indice possiamo convenzionalmente utilizzare per esso il nome di variabile corrispondente a un segno di underscore che in Lua è un identificatore valido:

```
-- una tabella array
local t = {45, 56, 89, 12, 0, 2, -98}

local sum = 0
for _, elem in ipairs(t) do
    sum = sum + elem
end
print(sum)
```

Se non vogliamo incorrere in errori è molto importante ricordarsi che con `ipairs()` verranno restituiti i valori in ordine di posizione da 1 in poi e fino a che non verrà trovato un valore `nil`. Se desiderassimo raggiungere tutte le coppie chiave/valore dovremo far ricorso all'iteratore `pairs()` trattato alla prossima sezione.

8.2 Funzione pairs()

Questa funzione primitiva di Lua considera la tabella come un dizionario pertanto l'iteratore restituirà in un ordine casuale tutte le coppie chiave valore contenute nella tabella stessa.

Una tabella con indici a salti verrà iterata parzialmente da `ipairs()` ma completamente da `pairs()`:

```
-- produzione tabella con salto
local t = {45, 56, 89}
local i = 100 + #t -- 100 holes
for _, v in ipairs({12, 0, 2, -98}) do
    t[i] = v
    i = i + 1
end

print("ipairs() table iteration test")
for index, elem in ipairs(t) do
    print(string.format("t[%3d] = %d", index, elem))
end

print("\npairs() table iteration test")
for key, val in pairs(t) do
    print(string.format("t[%3d] = %d", key, val))
end
```

```
> ipairs() table iteration test
> t[  1] = 45
> t[  2] = 56
> t[  3] = 89
>
> pairs() table iteration test
> t[  1] = 45
> t[  2] = 56
> t[  3] = 89
> t[104] = 0
> t[105] = 2
> t[106] = -98
> t[103] = 12
```

Il comportamento di questi due iteratori può lasciare perplessi ma è coerente con le caratteristiche della tabella di Lua.

8.3 Generic for

Come può essere implementato un iteratore in Lua? Per iterare è necessario mantenere alcune informazioni essenziali chiamate *stato* dell'iteratore. Per esempio l'indice a cui siamo arrivati nell'iterazione di una tabella/array e la tabella stessa.

Perchè non utilizzare la closure per memorizzare lo stato dell'iteratore?

Abbiamo incontrato le closure nella sezione 6.6. Proviamo a scrivere il codice per iterare una tabella:

```
-- costructor
local t = {45, 87, 98, 10, 16}

function iter(t)
    local i = 0
    return function ()
        i = i + 1
        return t[i]
    end
end

-- utilizzo
local iter_test = iter(t)
while true do
    local val = iter_test()
    if val == nil then
        break
    end
    print(val)
end
```

Funziona, molto semplicemente. Non è stato necessario introdurre nessun nuovo elemento al linguaggio. L'iteratore è solamente una questione d'implementazione che tra l'altro in questo caso ricrea l'iteratore `ipairs()` visto poco fa.

Infatti, la funzione `iter_test()` mantiene nella closure lo stato dell'iteratore — l'indice `i` e la tabella `t` — e restituisce uno dopo l'altro gli elementi della tabella. Il ciclo `while` infinito, s'interrompe quando il valore è `nil`.

Tuttavia, data l'importanza degli iteratori, Lua introduce il nuovo costrutto chiamato *generic for* che si aspetta una funzione proprio come la `iter()` del codice precedente. E in effetti funziona:

```
-- costructor
local t = {45, 87, 98, 10, 16}

function iter(t)
    local i = 0
    return function ()
        i = i + 1
        return t[i]
    end
end

-- utilizzo con il generic for
for v in iter(t) do
    print(v)
end
```

Riassumendo, la costruzione di un iteratore in Lua si basa sulla creazione di una funzione che restituisce uno alla volta gli elementi dell'insieme nella sequenza desiderata. Una volta costruito l'iteratore, questo potrà essere impiegato in un ciclo generic for.

Se per esempio si volesse iterare la collezione dei numeri pari compresi nell'intervallo da 1 a 10, avendo a disposizione l'apposito iteratore `evenNum()` che definiremo in seguito, potrei scrivere semplicemente:

```
for n in evenNum(1,10) do
    print(n)
end
```

8.4 L'esempio dei numeri pari

Per definire questo iteratore dobbiamo creare una funzione che restituisce a sua volta una funzione in grado di generare la sequenza dei numeri pari. L'iterazione termina quando giunti all'ultimo elemento, la funzione restituirà il valore nullo ovvero `nil`, cosa che succede in automatico senza dover esplicitare un'istruzione di `return` grazie al funzionamento del generic for.

Potremo fare così: dato il numero iniziale per prima cosa potremo calcolare il numero pari successivo usando la funzione della libreria standard

di Lua `math.ceil()` che fornisce il numero arrotondato al primo intero superiore dell'argomento.

Poi potremo creare la funzione di iterazione in sintassi anonima che prima incrementa di 2 il numero pari precedente — ed ecco perché dovremo inizialmente sottrarre la stessa quantità all'indice — e, se questo è inferiore all'estremo superiore dell'intervallo ritornerà l'indice e il numero pari della sequenza. Ecco il codice completo:

```
-- iteratore dei numeri pari compresi
-- nell'intervallo [first, last]
function evenNum(first, last)
    -- primo numero pari della sequenza
    local val = 2*math.ceil(first/2) - 2
    local i = 0 -- indice
    return function ()
        i = i + 1
        val = val + 2
        if val<=last then
            return val, i -- due variabili di ciclo
        end
    end
end

-- iterazione con due variabili di ciclo
for val, i in evenNum(13,20) do
    print(string.format("[%d] %d", i, val))
end
```

```
> [1] 14
> [2] 16
> [3] 18
> [4] 20
```

In questo esempio, oltre ad approfondire il concetto di iterazione basata sulla closure di Lua, possiamo notare che il generic for effettua correttamente anche l'assegnazione a più variabili di ciclo con le regole viste nel capitolo 1.

Naturalmente, l'implementazione data di `evenNum()` è solo una delle possibili soluzioni, e non è detto che non debbano essere considerate situazioni particolari come quella in cui si passa all'iteratore un solo numero o addirittura nessun argomento.

8.5 Stateless iterator

Una seconda versione del generatore di numeri pari può essere un buon esempio di un iteratore in Lua che non necessita di una closure, per un risultato ancora più efficiente.

Per capire come ciò sia possibile dobbiamo conoscere nel dettaglio come funziona il generic for in Lua; dopo la parola chiave `in` esso si aspetta altri due parametri oltre alla funzione da chiamare a ogni ciclo: una variabile che rappresenta lo stato invariante e la variabile di controllo.

Nel seguente codice la funzione `evenNum()` provvede a restituire i tre parametri necessari: la funzione `nextEven()` come iteratore, lo stato invariante, che per noi è il numero a cui la sequenza dovrà fermarsi e la variabile di controllo che è proprio il valore nella sequenza dei numeri pari, e con ciò abbiamo realizzato un stateless iterator in Lua, ovvero un iteratore che non ha necessità di closure.

La funzione `nextEven()` verrà chiamata a ogni ciclo con, nell'ordine, lo stato invariante e la variabile di controllo, pertanto fate attenzione, dovete mettere in questo stesso ordine gli argomenti nella definizione:

```
-- even numbers stateless iterator
local function nextEven(last, i)
   i = i + 2
   if i<=last then
      return i
   end
end

local function evenNum(a, b)
   local start = 2*math.ceil(a/2)-2
   return nextEven, b, start
end

-- example of the 'generic for' cycle
for n in evenNum(10, 20) do
    print(n)
end
```

Con gli iteratori abbiamo terminato l'esplorazione di base del linguaggio Lua. Questi primi nove capitoli sono sufficienti per scrivere programmi utili

perché trattano tutti gli argomenti essenziali. Il prossimo capitolo tratterà del paradigma della programmazione a oggetti in Lua.

8.6 Esercizi

esercizio 1 Dopo aver definito una tabella con chiavi e valori stampare le singole coppie tramite l'iteratore predefinito `pairs()`.

esercizio 2 Scrivere una funzione che accetta un array (una tabella con indici sequenziali interi) di stringhe e utilizzando la funzione di libreria `string.upper()` restituisca un nuovo array con il testo trasformato in maiuscolo (per esempio da `{abc, def, ghi}` a `{ABC, DEF, GHI}`).

esercizio 3 Scrivere la funzione/closure per l'iteratore che restituisce la sequenza dei quadrati dei numeri naturali a partire da 1 fino a un valore dato.

esercizio 4 Scrivere la versione *stateless* dell'iteratore dell'esercizio precedente.

esercizio 5 Scrivere la versione *stateless* dell'iteratore `ipairs()`. È possibile implementarlo in modo che la funzione d'iterazione restituisca per il ciclo generic for solamente l'elemento della tabella e non anche l'indice?

Programmazione a oggetti in Lua 9

In sintesi, il paradigma della *Object Oriented Programming* OOP, si basa sulla creazione di entità indipendenti chiamate *oggetti*. Ciascun oggetto incorpora sia dati che funzioni, che prendono il nome di *metodi*.

Ogni oggetto è un'istanza che fa parte di una stessa famiglia chiamata *classe*, una sorta di prototipo che rappresenta un "tipo di dati". Le classi possono essere ricavate da altre classi con il meccanismo dell'*ereditarietà* per specializzare il comportamento.

Per instanziare un oggetto di una classe si utilizza un metodo speciale chiamato *costruttore*, che valida gli eventuali dati in ingresso e instanzia in memoria l'oggetto.

In questo capitolo ritroveremo tutti questi concetti del paradigma della programmazione a oggetti dal punto di vista di Lua. Con essi la struttura del problema non è più pensata in termini di funzioni, ma attraverso la rappresentazione dei suoi elementi concettuali attraverso le classi, e le relazioni fra di essi attraverso l'ereditarietà.

Negli ultimi anni, la programmazione a oggetti è stata ridimensionata, tant'è che nei linguaggi di nuova generazione come Go e Rust non è inclusa nel modo classico. Ciò non toglie che essa possa rendere più intuitiva la programmazione in Lua, in special modo per chi sviluppa applicazioni per LuaTEX.

9.1 Il minimalismo di Lua

Il linguaggio Lua non è progettato con gli stessi obiettivi di Java o del C++, i due linguaggi più noti per la programmazione a oggetti, non possiede un controllo preventivo del tipo, non prevede il concetto sintattico di classe, non offre alcun meccanismo per dichiarare come privati campi

e metodi, e lascia al programmatore più di un modo per implementare i dettagli.

Tuttavia Lua offre basandosi sulle tabelle, il pieno supporto ai principi del paradigma a oggetti senza perdere le caratteristiche minimali del linguaggio.

9.2 Una classe Rettangolo

Costruiremo una classe per rappresentare un rettangolo. Si tratta di un ente geometrico definito da due soli parametri *larghezza* e *altezza*, e dotato di proprietà come l'area e il perimetro, che implementeremo come metodi.

Un primo tentativo potrebbe essere questo:

```
-- prima tentativo di implementazione
-- di una classe rettangolo
Rectangle = {} -- creazione tabella (oggetto)

-- creazione dei due campi
Rectangle.width = 12
Rectangle.height = 7

-- un primo metodo assegnato direttamente
-- ad un campo della tabella
function Rectangle.area ()
   -- accesso alla variabile 'Rectangle'
   return Rectangle.larghezza * Rectangle.altezza
end

-- primo test
print(Rectangle.area())    --> stampa 84, OK
print(Rectangle.height)    --> stampa  7, OK
```

Ci accorgiamo presto che questo tentativo è difettoso in quanto non rispetta l'indipendenza degli oggetti rispetto al loro nome. Infatti il prossimo test fallisce:

```
-- ancora la prima implementazione
Rectangle = {width = 12, height = 7}

-- un metodo dell'oggetto
function Rectangle.area ()
```

```
   -- accesso alla variabile 'Rectangle' attenzione!
   local l = Rectangle.larghezza
   local a = Rectangle.altezza
   return l * a
end

-- secondo test
r = Rectangle   -- creiamo un secondo riferimento
Rectangle = nil -- distruggiamo il riferimento originale

print(r.width)     --> stampa 12, OK
print(r.area())    --> errore!
```

Il problema sta nel fatto che nel metodo `area()` compare il particolare riferimento alla tabella `Rectangle` che invece deve poter essere qualunque. La soluzione non può che essere l'introduzione del riferimento dell'oggetto come parametro esplicito nel metodo stesso, ed è la stessa utilizzata — in modo nascosto ma vedremo che è possibile nascondere il riferimento anche in Lua — dagli altri linguaggi di programmazione che supportano gli oggetti.

Secondo quest'idea dovremo riscrivere il metodo `area()` in questo modo (in Lua il riferimento esplicito all'oggetto deve chiamarsi `self` pertanto abituiamoci fin dall'inizio a questa convenzione così da poter generalizzare la validità del codice):

```
-- seconda implementazione
Rettangolo = {larghezza=12, altezza=7}

-- il metodo diviene indipendente dal particolare
-- riferimento all'oggetto:
function Rettangolo.area ( self )
   return self.larghezza * self.altezza
end

-- ed ora il test
myrect = Rettangolo
Rettangolo = nil -- distruggiamo il riferimento

print(myrect.larghezza)    --> stampa 12, OK
print(myrect.area(myrect)) --> stampa 84, OK
                           --      funziona!
```

Fino a ora abbiamo costruito l'oggetto sfruttando le caratteristiche della tabella e la particolarità che consente di assegnare una funzione a una variabile. Da questo momento entra in scena l'operatore `:` — che chiameremo *colon notation*.

L'operatore `:` fa in modo che le seguenti due espressioni siano perfettamente equivalenti anche se le rende differenti dal punto di vista concettuale agli occhi del programmatore:

```
-- forma classica:
myrect.area(myrect)

-- forma implicita
-- e 'self' prende lo stesso riferimento di 'myrect'
myrect:area()
```

Questo operatore è il primo nuovo elemento che Lua introduce per facilitare la programmazione orientata agli oggetti. Se si accede a un metodo memorizzato in una tabella con l'operatore due punti `:` anziché con l'operatore `.` allora l'interprete Lua aggiungerà implicitamente un primo parametro con il riferimento alla tabella stessa a cui assegnerà il nome di `self`.

9.3 Metatabelle

Il linguaggio Lua si fonda sull'essenzialità tanto che supporta la programmazione a oggetti utilizzando quasi esclusivamente le proprie risorse di base senza introdurre nessun nuovo costrutto. In particolare Lua implementa gli oggetti utilizzando la tabella l'unica struttura dati disponibile nel linguaggio, assieme a particolari funzionalità dette *metatabelle* e *metametodi*.

Il salto definitivo nella programmazione OOP consiste nel poter costruire una *classe* senza ogni volta assemblare i campi e metodi, introducendo un qualcosa che faccia da stampo per gli oggetti.

In Lua l'unico meccanismo disponibile per compiere questo ultimo importante passo consiste nelle *metatabelle*. Esse sono normali tabelle contenenti funzioni dai nomi prestabiliti che vengono chiamati quando si verificano particolari eventi come l'esecuzione di un'espressione di somma tra due tabelle con l'operatore `+`. Ogni tabella può essere associata a una metatabella e questo consente di creare degli insiemi di tabelle che condividono una stessa aritmetica.

I nomi di queste funzioni particolari dette *metametodi* iniziano tutti con un doppio trattino basso, per esempio nel caso della somma sarà richiesta la funzione `__add()` della metatabella associata alle due tabelle addendo — e se non esiste verrà generato un errore.

Per assegnare una metatabella si utilizza la funzione `setmetatable()`. Essa ha due argomenti tabella, la prima è l'oggetto e la seconda è la metatabella.

Il metametodo più semplice di tutti è `__tostring()`. Esso viene invocato se una tabella è data come argomento alla funzione `print()` per ottenere il valore stringa da stampare. Se non esiste una metatabella associata con questo metametodo verrà stampato l'indirizzo di memoria della tabella:

```
-- un numero complesso
local complex = {real = 4, imag = -9}
print(complex) --> stampa: 'table: 0x9eb65a8'

-- un qualcosa di più utile: metatabella in sintassi
-- anonima con il metametodo __tostring()
local mt = {}
mt.__tostring = function (c)
   local fmt = "(%0.2f, %0.2f)"
   return string.format(fmt, c.real or 0, c.imag or 0)
end

-- assegnazione della metatabella mt a complex
setmetatable(complex, mt)

-- riprovo la stampa
print(complex)  --> stampa '(4.00, -9.00)'
```

9.4 Il metametodo __index

Il metametodo che interessa la programmazione a oggetti in Lua è `__index`. Esso interviene quando viene chiamato un campo di una tabella che non esiste e che normalmente restituirebbe il valore `nil`. Un esempio di codice chiarirà il meccanismo:

```
-- una tabella con un campo 'a'
-- ma senza un campo 'b'
local t = {a = 'Campo A'}
```

```
print(t.a)  --> stampa 'Campo A'
print(t.b)  --> stampa 'nil'

-- con metatabella e metametodo
local mt = {
    __index = function ()
        return 'Attenzione: campo inesistente!'
    end
}

-- assegniamo 'mt' come metatabella di 't'
setmetatable(t, mt)

-- adesso riproviamo ad accedere al campo b
print(t.b)  --> stampa 'Attenzione: campo inesistente!'
```

Tornando all'oggetto `Rettangolo` riscriviamo il codice creando adesso una tabella che assume il ruolo concettuale di una vera e propria classe:

```
-- una nuova classe Rettangolo (campi):
local Rettangolo = {larghezza=10, altezza=10}

-- un metodo:
function Rettangolo:area()
  return self.larghezza * self.altezza
end

-- creazione metametodo
Rettangolo.__index = Rettangolo

-- un nuovo oggetto Rettangolo
local r = {}
setmetatable(r, Rettangolo)

print( r.larghezza ) --> stampa 10, Ok
print( r:area() )    --> stampa 100, Ok
```

Queste poche righe di codice racchiudono il meccanismo della creazione di una nuova classe in Lua: abbiamo infatti assegnato a una nuova tabella `r` la metatabella con funzione di classe `Rettangolo`. Quando viene richiesta la stampa del campo `larghezza`, poiché tale campo non esiste nella tabella

vuota `r` verrà ricercato il metametodo `__index` nella metatabella associata che è appunto la tabella `Rettangolo`.

A questo punto il metametodo restituisce semplicemente la tabella `Rettangolo` stessa e questo fa sì che tutti i campi e i metodi siano ereditati in essa contenuti siano accessibili da `r`. Il campo `larghezza` e il metodo `area()` del nuovo oggetto `r` sono in realtà quelli definiti nella tabella `Rettangolo`.

Se volessimo creare invece un rettangolo assegnando direttamente la dimensione dei lati dovremo semplicemente crearli in `r` con i nomi previsti dalla classe: `larghezza` e `lunghezza`. Il metodo `area()` sarà ancora caricato dalla tabella `Rettangolo` ma i campi numerici con le nuove misure dei lati saranno quelli interni dell'oggetto `r` e non quelli della metatabella poiché semplicemente esistono.

Questa costruzione funziona ma può essere migliorata con l'introduzione del costrutture come vedremo meglio in seguito. Il linguaggio apparirà sempre più concettualmente simile a una classe.

9.5 Il costruttore

Proponendoci ancora la rappresentazione del concetto di rettangolo, completiamo il quadro introducendo il costruttore della classe. Il lavoro che dovrà svolgere questa speciale funzione sarà quello di inizializzare i campi argomento in una delle tante modalità possibili, una volta effettuato il controllo di validità degli argomenti.

Il codice completo della classe `Rettangolo` è il seguente:

```
-- nuova classe Rettangolo (campi con valore di default)
local Rettangolo = {larghezza = 1, altezza = 1}

-- metametodo
Rettangolo.__index = Rettangolo

-- metodo di classe
function Rettangolo:area()
  return self.larghezza * self.altezza
end

-- costruttore di classe
function Rettangolo:new( o )
```

```
    -- creazione nuova tabella
    -- se non ne viene fornita una
    o = o or {}
    -- controllo campi
    if o.larghezza and o.larghezza < 0 then
        error("campo larghezza negativo")
    end
    if o.altezza and o.altezza < 0 then
        error("campo altezza negativo")
    end

    -- assegnazione metatabella
    setmetatable(o, self)

    -- restituzione riferimento oggetto
    return o
end

-- codice utente ------------------
local r = Rettangolo:new{larghezza=12, altezza=2}

print(r.larghezza)  --> stampa  12, Ok
print(r:area())     --> stampa  24, Ok

local q = Rettangolo:new{larghezza=12}
print(q:area())     --> stampa 12, Ok
```

Il costruttore chiamato `new()` accetta una tabella come argomento, altrimenti ne crea una vuota, controlla gli eventuali parametri geometrici, assegna la metatabella e restituisce l'oggetto.

Il costruttore passa al metodo `new()` il riferimento implicito a `Rettangolo` grazie alla colon notation, per cui `self` punterà a `Rettangolo`.

Quando viene passata una tabella con uno o due campi sulle misure dei lati al costruttore, l'oggetto disporrà delle misure come valori interni effettivi, cioè dei parametri indipendenti che costituiscono il suo stato interno. Lo sviluppatore può fare anche una diversa scelta, quella per esempio di considerare la tabella argomento del costruttore come semplice struttura di chiavi/valori da sottoporre al controllo di validità e poi includere in una nuova tabella con modalità e nomi che riguardano solo l'implementazione interna della classe.

9.6 Questa volta un cerchio

Per capire ancor meglio i dettagli e renderci conto di come funziona il meccanismo automatico delle metatabelle, costruiamo una classe `Cerchio` che annoveri fra i suoi metodi uno che modifichi il valore del raggio aggiungendovi una misura:

```
local Cerchio = {radius=0}
Cerchio.__index = Cerchio

function Cerchio:area()
   return math.pi*self.radius^2
end

function Cerchio:addToRadius(v)
   self.radius = self.radius + v
end

function Cerchio:__tostring()
   local frmt = 'Sono un cerchio di raggio %0.2f.'
   return string.format(frmt, self.radius)
end

-- il costruttore attende l'eventuale valore del raggio
function Cerchio:new(r)
   local o = {}
   if r then
      o.radius = r
   end
   setmetatable(o, self)
   return o
end

-- codice utente ----------------------
local o = Cerchio:new()
print(o)  --> stampa 'Sono un cerchio di raggio 0.00'

o:addToRadius(12.342)

print(o)  --> stampa 'Sono un cerchio di raggio 12.34'
print(o:area())  --> stampa '478.54298786'
```

Nella sezione del codice utente viene dapprima creato un cerchio senza fornire alcun valore per il raggio. Ciò significa che quando stampiamo il valore del raggio con la successiva istruzione otteniamo 0 che è il valore di default del raggio dell'oggetto `Cerchio` per effetto della chiamata a `__index` della metatabella.

Fino a questo momento la tabella dell'oggetto `o` non contiene alcun campo `radius`. Cosa succede allora quando viene lanciato il comando `o:addToRadius(12.342)`?

Il metodo `addToRadius()` contiene una sola espressione. Come da regola viene prima valutata la parte a destra ovvero `self.radius + v`. Il primo termine assume il valore previsto in `Cerchio` — quindi zero — grazie al metametodo, e successivamente il risultato della somma uguale all'argomento `v` è memorizzato nel campo `o.radius` che viene creato effettivamente solo in quel momento.

9.7 Ereditarietà

Il concetto di ereditarietà nella programmazione a oggetti consiste nella possibilità di derivare una classe da un'altra per specializzarne il comportamento.

L'operazione di derivazione incorpora automaticamente nella sottoclasse tutti i campi e i metodi della classe base. Dopodiché si implementano o si modificano i metodi della classe derivata creando una gerarchia di oggetti.

In Lua l'operazione di derivazione consiste molto semplicemente nel creare un oggetto con il costruttore della classe base e modificarne o aggiungerne i metodi o i campi.

Vediamo un esempio semplice dove si rappresenta il concetto generale di una persona che svolge attività sportiva e da questo, il concetto di una persona che svolge uno specifico sport:

```
-- classe base
local Sportivo = {}

-- costructor
function Sportivo:new(t)
    t = t or {}
    setmetatable(t, self)
    self.__index = self
```

```
    return t
end

-- base methods
function Sportivo:set_name(name)
    self.name = name
end

function Sportivo:print()
    print("'"..self.name.."'")
end

-- derivazione
local Schermista = Sportivo:new()

-- specializzazione classe derivata
-- nuovo campo
Schermista.rank = 0

function Schermista:add_to_rank(points)
    self.rank = self.rank + (points or 0)
end

function Schermista:set_weapon(w)
    self.weapon = w or ""
end

-- overriding method
function Schermista:print()
    local fmt = "'%s' weapon->'%s' rank->%d"
    print(
        string.format(
            fmt,
            self.name,
            self.weapon,
            self.rank
        )
    )
end

-- test
```

```
local s = Sportivo:new{name="Gianni"}
s:print() --> stampa 'Gianni' OK

-- il metodo costruttore new() è quella della classe base!
local f = Schermista:new{
    name="Tiger",
    weapon="Foil"
}
f:add_to_rank(45)

f:print() --> stampa 'Tiger' weapon->'Foil' rank->45

-- chiamata a un metodo della classe base
f:set_name("Amedeo")
f:print() --> stampa 'Amedeo' weapon->'Foil' rank->45
```

Continua tutto a funzionare per via della ricerca effettuata dal metametodo `__index` che funziona a ritroso fino alla classe base.

9.8 Esercizi

esercizio 1 Aggiungere alla classe `Rettangolo` riportata nel testo il metametodo `__tostring()` che stampi in console il rettangolo dalle dimensioni corrispondenti ad altezza e larghezza usando i caratteri `+` per gli spigoli e i caratteri `-` e `|` per disegnare i lati. Utilizzare le funzioni di libreria `string.rep()` e `string.format()`.

esercizio 2 Creare una classe corrispondente al concetto di numero complesso e implementare le quattro operazioni aritmetiche tramite metametodi (riferimento matematico qui). Aggiungere anche il metodo `__tostring()` per stampare il numero complesso e poter controllare i risultati di operazioni di test.

esercizio 3 Ideare una classe base e una classe derivata dandone un'implementazione.

Parte II

Applicazioni Lua in LuaTEX

Preparazione del sistema 10

Questo capitolo è ancora introduttivo fornendo informazioni di base per l'esecuzione di codice Lua all'interno dei motori di composizione.

10.0.1 Motori di composizione e formati

LuaTEX è un programma che elabora un file di testo contenente codice TEX per comporne il corrispondente file PDF. Il sistema TEX distribuisce almeno altri due *motori di composizione* dotati dell'interprete Lua, LuaHBTEX e LuajitTEX.

Il codice TEX contiene testo e macro. Il testo formerà i capoversi, i titoli eccetera del documento, mentre le macro ne stabiliscono aspetto e struttura. I motori di composizione dispongono di particolari macro dette *primitive* implementate direttamente in essi, e la possibilità di definire nuove macro per svolgere più facilmente compiti ripetitivi.

Queste nuove macro il cui codice è generalmente scritto da esperti offrono funzionalità di più alto livello molto utili per l'utente. I motori di composizione caricano sempre nella fase iniziale un insieme di macro di alto livello chiamate *formato* perché possono stabilire anche nuove regole di sintassi dei comandi.

Se si avvia un qualsiasi motore di composizione della famiglia TEX verrà caricato il formato più semplice chiamato *plain*. Se si vuole invece utilizzare un diverso formato, per esempio il più diffuso LATEX, occorre specificarne il nome con l'opzione `--fmt` quando si scrive il comando di compilazione al terminale.

Tuttavia data la diffusione tra gli utenti dei formati ad alto livello, sono stati predisposti comandi scorciatoia. Per esempio il programma `pdflatex`, rimanda all'effettivo motore di composizione, il tradizionale `pdftex`, dando istruzione di caricare il formato LATEX.

Riassumendo, i motori di composizione sono programmi tipografici mentre i formati sono insiemi coerenti di macro basate sulle primitive di sistema. I nomi dei programmi disponibili nel sistema TEX possono quindi confondere gli utenti se non conoscono questa importante distinzione: alcuni di essi sono comandi scorciatoia per identificare sia il motore sia il formato e non un motore di composizione a se stante.

Tornando all'ambito Lua, il programma `lualatex` non è un vero e proprio motore di composizione ma il collegamento a `luahbtex` che carica il formato LATEX. Per rendercene conto basta scrivere in un terminale il nome del programma, leggere l'output e premere CTRL + C per chiudere l'esecuzione di prova:

```
> lualatex
This is LuaHBTeX, Version 1.12.0 (TeX Live 2020/W32TeX)
 restricted system commands enabled.
**
```

Per ulteriore informazione, `luahbtex` è il motore di composizione `luatex` a cui è stato sostituito il componente nativo di calcolo della forma dei font con il modulo HarfBuzz.

10.0.2 Lua in LuaTEX

Consideriamo la stampa di un semplice testo nell'output di console usando Lua. Per farlo, in un sorgente LuaTEX il codice va inserito come argomento della primitiva `\directlua` in questo modo:

```
% !TeX program = LuaTeX
\directlua{
    print("Hello World!")
}
\bye
```

Con la compilazione di questo sorgente il testo uscirà tra gli altri messaggi di output senza che sia prodotto un file PDF. Ciò significa che `\directlua` è una macro espandibile con risultato vuoto.

La prima riga di commento è una *riga magica*, comodissima nel dare istruzione allo shell editor su quale motore di composizione e formato

utilizzare per compilare il documento. In questo caso essa è scritta nella sintassi prevista da TeX Works. Per la comprensione del codice le righe magiche sono inutili ma aiutano il lettore a stabilire il contesto di esecuzione, perciò le troverete se pertinente nei listati della guida.

Se il sorgente è memorizzato nel file `primo.tex`, possiamo verificare quanto previsto in un terminale lanciando il comando:

```
$ luatex primo
```

per il sistema operativo Windows e la distribuzione TeX Live 2020, l'output della console è:

```
This is LuaTeX, Version 1.12.0 (TeX Live 2020/W32TeX)
    restricted system commands enabled.
(./primo.texHello World!
)
warning  (pdf backend): no pages of output.
Transcript written on primo.log.
```

10.0.3 Lua in LuaLaTeX

Con LuaLaTeX si ottiene lo stesso risultato ma con il sorgente scritto nella sintassi LaTeX:

```
% !TeX program = LuaLaTeX
\documentclass{article}
\directlua{
    print("Hello World!")
}
\begin{document}
\end{document}
```

e questa volta il comando di compilazione è:

```
$ lualatex primo
```

Avremo potuto inserire la macro all'interno dell'ambiente *document* anziché nel preambolo. Quando TeX incontra la macro `\directlua` ne *espande* l'argomento e passa a Lua il controllo. L'interprete esegue immediatamente il codice restituendo il controllo dell'esecuzione a TeX al termine.

10.0.4 Motori di composizione e formati

LuaTEX è un programma che elabora un file di testo contenente codice TEX per comporne il corrispondente file PDF. Il sistema TEX distribuisce almeno altri due *motori di composizione* dotati dell'interprete Lua, LuaHBTEX e LuajitTEX.

Il codice TEX contiene testo e macro. Il testo formerà i capoversi, i titoli eccetera del documento, mentre le macro ne stabiliscono aspetto e struttura. I motori di composizione dispongono di particolari macro dette *primitive* implementate direttamente in essi, e la possibilità di definire nuove macro per svolgere più facilmente compiti ripetitivi.

Queste nuove macro il cui codice è generalmente scritto da esperti offrono funzionalità di più alto livello molto utili per l'utente. I motori di composizione caricano sempre nella fase iniziale un insieme di macro di alto livello chiamate *formato* perché possono stabilire anche nuove regole di sintassi dei comandi.

Se si avvia un qualsiasi motore di composizione della famiglia TEX verrà caricato il formato più semplice chiamato *plain*. Se si vuole invece utilizzare un diverso formato, per esempio il più diffuso LATEX, occorre specificarne il nome con l'opzione `--fmt` quando si scrive il comando di compilazione al terminale.

Tuttavia data la diffusione tra gli utenti dei formati ad alto livello, sono stati predisposti comandi scorciatoia. Per esempio il programma `pdflatex`, rimanda all'effettivo motore di composizione, il tradizionale `pdftex`, dando istruzione di caricare il formato LATEX.

Riassumendo, i motori di composizione sono programmi tipografici mentre i formati sono insiemi coerenti di macro basate sulle primitive di sistema. I nomi dei programmi disponibili nel sistema TEX possono quindi confondere gli utenti se non conoscono questa importante distinzione: alcuni di essi sono comandi scorciatoia per identificare sia il motore sia il formato e non un motore di composizione a se stante.

Tornando all'ambito Lua, il programma `lualatex` non è un vero e proprio motore di composizione ma il collegamento a `luahbtex` che carica il formato LATEX. Per rendercene conto basta scrivere in un terminale il nome del programma, leggere l'output e premere CTRL + C per chiudere l'esecuzione di prova:

```
> lualatex
This is LuaHBTeX, Version 1.12.0 (TeX Live 2020/W32TeX)
 restricted system commands enabled.
**
```

Per ulteriore informazione, `luahbtex` è il motore di composizione `luatex` a cui è stato sostituito il componente nativo di calcolo della forma dei font con il modulo HarfBuzz.

10.0.5 Lua in LuaTeX

Consideriamo la stampa di un semplice testo nell'output di console usando Lua. Per farlo, in un sorgente LuaTeX il codice va inserito come argomento della primitiva `\directlua` in questo modo:

```
% !TeX program = LuaTeX
\directlua{
    print("Hello World!")
}
\bye
```

Con la compilazione di questo sorgente il testo uscirà tra gli altri messaggi di output senza che sia prodotto un file PDF. Ciò significa che `\directlua` è una macro espandibile con risultato vuoto.

La prima riga di commento è una *riga magica*, comodissima nel dare istruzione allo shell editor su quale motore di composizione e formato utilizzare per compilare il documento. In questo caso essa è scritta nella sintassi prevista da TeX Works. Per la comprensione del codice le righe magiche sono inutili ma aiutano il lettore a stabilire il contesto di esecuzione, perciò le troverete se pertinente nei listati della guida.

Se il sorgente è memorizzato nel file `primo.tex`, possiamo verificare quanto previsto in un terminale lanciando il comando:

```
$ luatex primo
```

per il sistema operativo Windows e la distribuzione TeX Live 2020, l'output della console è:

```
This is LuaTeX, Version 1.12.0 (TeX Live 2020/W32TeX)
    restricted system commands enabled.
(./primo.texHello World!
)
warning  (pdf backend): no pages of output.
Transcript written on primo.log.
```

10.0.6 Lua in LuaLaTeX

Con LuaLaTeX si ottiene lo stesso risultato ma con il sorgente scritto nella sintassi LaTeX:

```
% !TeX program = LuaLaTeX
\documentclass{article}
\directlua{
    print("Hello World!")
}
\begin{document}
\end{document}
```

e questa volta il comando di compilazione è:

```
$ lualatex primo
```

Avremo potuto inserire la macro all'interno dell'ambiente *document* anziché nel preambolo. Quando TeX incontra la macro `\directlua` ne *espande* l'argomento e passa a Lua il controllo. L'interprete esegue immediatamente il codice restituendo il controllo dell'esecuzione a TeX al termine.

Primi passi in LuaTEX 11

In questo capitolo affronteremo il tema dello scambio dati tra TEX e Lua, per poi approfondire le applicazioni nelle pagine successive, un territorio di gran lunga ancora inesplorato.

11.1 La primitiva `\directlua`

Come già accennato nell'introduzione, il principale modo di eseguire codice Lua in LuaTEX è assegnarlo come argomento alla macro `\directlua`. Quello che avviene è stabilito da queste regole:

1. l'argomento di `\directlua` viene espanso — può quindi contenere macro con un testo di sostituzione — ed eseguito come blocco;
2. le variabili locali hanno validità solo all'interno del blocco mentre quelle globali saranno valide anche in quelli di successive `\directlua`;
3. l'espansione di `\directlua` è vuota;

Come esempio minimo consideriamo il seguente sorgente LuaTEX che stampa in console assieme agli altri messaggi emessi dal processo di compilazione l'ora di inizio della compilazione:

```
% !TeX program = LuaTeX
\directlua{
local time = \the\time
local h = math.floor(time/60)
local m = time - h*60
print(h..":"..m)}% stampa '18:16'
\bye
```

Al termine dell'espansione l'istruzione di assegnazione della variabile numerica `time` contiene il minuto trascorso dall'inizio del giorno.

Per il formato LuaLATEX lo stesso file potrebbe essere:

```
% !TeX program = LuaLaTeX
\documentclass{article}
\directlua{
    local time = \the\time
    local h = math.floor(time/60)
    local m = time - h*60
    print(h..":"..m)
}
\begin{document}
\end{document}
```

11.1.1 Registro delle compilazioni

Ammettiamo di voler mantenere un registro

Note finali

Per i ringraziamenti ...