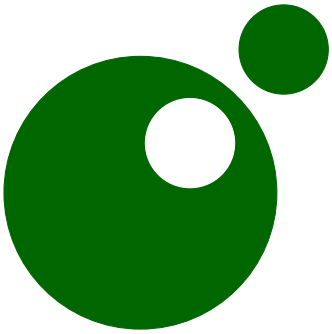
Roberto Giacomelli

Guida al linguaggio Lua per LuaTEX



2021/04/30 — v0.4.8

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
Copyright © 2021, Roberto Giacomelli

Questa documentazione è soggetta alla licenza LPPL LaTeX Project Public License, versione 1.3 o successive ed è curata dall'autore.

Guide's *biblatex* entry:

```
@manual{guit:guidalua,
  title       ={Guida al linguaggio Lua per Lua\TeX},
  author      ={Giacomelli, Roberto},
  date        ={2021-04-30},
  version     ={v0.4.8},
  pagetotal   ={156},
  langid      ={italian},
  url         ={https://www.guitex.org/home/images/doc/GuideGuIT/guidalua.pdf},
  urldate     ={2021-04-01},
  organization={GuIT, Gruppo Utilizzatori Italiani di \TeX},
  series      ={Guide Tematiche del GuIT},
}
```

Presentazione

Motivazione

Questa guida tematica è dedicata alla programmazione in Lua all'interno dei motori di composizione del sistema TEX.

Con Lua è possibile compiere sia elaborazioni generiche come l'interrogazione di basi di dati che elaborazioni tipografiche interagendo con il compositore interno. Con Lua rispetto a TEX, è più semplice ed efficiente effettuare calcoli numerici o avvalersi di avanzate librerie esterne.

La nuova generazione di compositori amplia così notevolmente gli scenari applicativi. Se da un lato è auspicabile che queste potenzialità diventino disponibili per gli utenti finali per mezzo di moduli e pacchetti, dall'altro è utile fornire dettagli ed esempi per implementare proprie soluzioni o per poter scrivere nuovi moduli condividendone lo sviluppo con la community.

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, di bilanciamento tra Lua e TEX, eccetera. Per questo, ho pensato di contribuire con questa guida cercando di presentare il quadro della crescente complessità del sistema.

Piano della guida

La guida è divisa in tre parti: la prima offre una panoramica rapida in forma di *tutorial* per iniziare subito con Lua seguendo i passi di un ipotetico utente alle prese con problemi compositivi (parte I), la seconda tratta delle basi del linguaggio Lua (parte II) e contiene numerosi esercizi, e infine la terza illustra esempi applicativi con l'uso delle librerie di composizione cn i nodi (parte III).

Tra gli argomenti ci sono:

- differenza tra motore e formato di composizione → capitolo 2,
- basi del linguaggio Lua → dal capitolo 4,
- tecniche di programmazione e di rappresentazione dei dati → dal capitolo 13,
- la tecnologia dei nodi in Lua → capitolo 14.

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.

Contribuire e collaborare

Spero che i lettori vorranno contribuire al testo inviando proprie soluzioni o nuovi contributi piccoli o grandi. Lo si può fare attraverso lo strumento che preferite, scrivendomi un messaggio di posta elettronica all'indirizzo `giaconet.mailbox@gmail.com`, oppure utilizzando il `repository git` dei sorgenti della guida, eseguendo un Pull Request o aprendo una discussione premendo il pulsante Issues.

Altre risorse

La risorsa principale per imparare Lua, a cui rimando per tutti gli approfondimenti, è certamente il PIL [Ier16], acronimo del titolo del libro *Programming In Lua* di Roberto Ierusalimschy il principale Autore di Lua. Questo testo non solo è completo e autorevole ma è anche ben scritto e composto[1].

La seconda importante risorsa su Lua si trova in rete all'indirizzo `www.lua.org/manual/5.3/` ed è il *reference* del linguaggio [IFC21] con le specifiche di sintassi, metametodi, funzioni di libreria, C API, eccetera.

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

Quanto a LuaTEX il riferimento è il suo manuale [Lua21] che, come quasi tutta la documentazione nel sistema TEX, può essere visualizzato a video con il comando da terminale:

```
$ texdoc luatex
```

Note di lettura

Nei listati compilabili riportati nella guida compare alla prima linea la *riga magica*, un commento utile per dare istruzioni all'editor sul compilatore da usare, ma che qui aiuterà il lettore a stabilire il contesto del codice. La sintassi delle righe magiche dipende dall'editor, in questa guida useremo le regole di TeXworks.

Se presente nel progetto, alla seconda riga dei listati si troverà invece il nome del file che il lettore potrà scaricare ed eseguire per i propri esperimenti dal `repository git`.

Nella parte II ho cercato di non dare per scontati i concetti fondamentali della programmazione. Ovviamente il lettore già preparato procederà più velocemente nel prendere dimestichezza con Lua. Ho invece escluso dalla guida TEX, per esempio non spiegando come si definisce una macro utente o come si lavora con il formato LATEX3. Rimando senza indugio alla copiosa documentazione disponibile a cominciare da quella scaricabile dal sito GUIT.

LuaTEX è un programma molto complesso, uno strumento eccezionalmente vasto. Le applicazioni illustrate nella guida non possono che derivare dalle mie conoscenze. Non metto in dubbio che ci siano modi migliori, più sicuri, più efficienti, più accurati, più completi di riscriverne il codice.

Auguro buona e proficua lettura nell'esplorare l'avamposto lunare.

Indice

Parte I

Tutorial

Let's start with Lua 1

Per dare l'idea di come si possa usare Lua all'interno del sistema di composizione TEX, questa prima parte della guida è in forma di *tutorial* cioè di brevi resoconti dei progressi compiuti da un ipotetico utente LuaLATEX indaffarato nel risolvere alcuni problemi con i suoi documenti: fare calcoli con una calcolatrice o comporre una tabella ripetitiva.

Nel margine di pagina il lettore troverà i rimandi di approfondimento. I tutorial infatti non hanno l'obiettivo di spiegare la programmazione Lua ma di introdurne l'utilità.

1.1 La calcolatrice

Una calcolatrice, una macro `\expr` che accetti un'espressione numerica e ne stampi il risultato. Sarebbe davvero utile non dover più calcolare a parte il risultato e riportarlo nel sorgente del documento LATEX con un copia incolla o peggio a mano.

Tentiamo qualcosa di molto semplice con Lua, assegnare l'espressione a una variabile per poi stamparla nel documento:

```
% !TeX program = LuaLaTeX
% filename: app-tut/E0-001-expr.tex
\documentclass{article}
\newcommand\expr[1]{\directlua{
    local result = #1
    tex.print(tostring(result))
}}
\begin{document}
Finalmente una calcolatrice:
\( 1.24 (7.45 + 11.21) = \expr{1.24*(7.45 + 11.21)}\)
\end{document}
```

Lua in TEX § 2.1.3

Variabili locali § 2.4, § 4.2.1

`tex.print()` § 2.3

compilando con LuaLaTeX il risultato è:

Finalmente una calcolatrice: $1.24(7.45 + 11.21) = 23.1384$

Un buon inizio. Nel sorgente all'interno della macro `\directlua` il primo argomento è stato sostituito con l'espressione che viene poi valutata da Lua. Nessun pacchetto aggiuntivo caricato, qualsiasi espressione numerica è lecita, e questo solo e soltanto usando Lua incluso in LuaTeX.

Funziona anche con le stringhe, a patto di delimitarne il valore, e con le espressioni booleane. Proviamo:

```
\( 56.9 > 78.42 \) è \texttt{\expr{ 56.9 > 78.42 }}
```

$56.9 > 78.42$ è `false`

1.1.1 Espressioni booleane personalizzate

E se si volessero sostituire le rappresentazioni testuali dei valori vero e falso? Ecco la modifica:

Ciclo `if` § 6.1

Tipo `boolean` § 4.3

Operatore ternario § 7.1

```
\newcommand\expr[1]{\directlua{
    local result = #1
    if type(result) == "boolean" then
        result = result and "vero" or "falso"
    end
    tex.print(tostring(result))
}}
```

Un semplice test ci conforterà sulla correttezza del codice e si funziona:

`\expr{100 == 100 and 7 > 3}`	vero
`\expr{-10 < -100}`	falso

1.1.2 Arrotondamento numerico

Vorrei poter regolare l'arrotondamento del risultato numerico della calcolatrice ricorrendo a un argomento opzionale separato dall'espressione da una virgola:

string.format() § 10.3.1

```
\newcommand\expr[1]{\directlua{
    local result, dec = #1
    if type(result) == "boolean" then
        result = result and "vero" or "falso"
    elseif type(result) == "number" and dec then
        local perc = string.char(37)
        local fmt1 = perc..perc.."0."..perc.."df"
        local fmt2 = string.format(fmt1, dec)
        result = string.format(fmt2, result)
    end
    tex.print(tostring(result))
}}
```

Stavolta il codice perde un po' di chiarezza perché non è possibile usare direttamente il carattere percento `%` che verrebbe interpretato come inizio di un commento nel costruire le stringhe di formato. Ovviamente questo non succederebbe se il codice fosse in un file separato o se fosse racchiuso in un ambiente *luacode* dell'omonimo pacchetto LaTeX [Pég12].

Mettiamo alla prova la modifica alla macro `\expr`:

```
\(\sqrt{2} + \sqrt{3} \approx \expr{ 2^0.5 + 3^0.5, 2}\)
```

$\sqrt{2} + \sqrt{3} \approx 3.15$

1.1.3 Funzioni matematiche

Potremo trovare una sintassi un po' più chiara per indicare il numero di cifre a cui arrontondare il risultato, tuttavia c'è di mezzo un problema più urgente: poter usare funzioni matematiche come seno e coseno. Se scrivessimo `\expr{sin(1)^2 + cos(1)^2}` non otterremo il valore unitario ma un errore. Dovremo infatti usare la scomoda notazione `math.`⟨*funzione*⟩/⟨*costante*⟩ come in `\expr{math.cos(math.pi)}` invece della più naturale `\expr{cos(pi)}`. Ma ci vuole poco a riassegnare le funzioni a nomi locali per far si che l'identità trigonometrica precedente sia un'espressione valida:

```
\newcommand\expr[1]{\directlua{
    local cos = math.cos
```

```
    local sin = math.sin
    local result, dec = #1
    if type(result) == "boolean" then
        result = result and "vero" or "falso"
    elseif type(result) == "number" and dec then
        local perc = string.char(37)
        local fmt1 = perc..perc.."0."..perc.."df"
        local fmt2 = string.format(fmt1, dec)
        result = string.format(fmt2, result)
    end
    tex.print(tostring(result))
}}
```

Una prova della calcolatrice potenziata con le funzioni trigonometriche ci dirà se tutto funziona ancora bene:

```
\(\sin^2(1/2) + \cos^2(1/2) =\expr{sin(0.5)^2 + cos(0.5)^2, 8}\).
```

$\sin^2(1/2) + \cos^2(1/2) = 1.00000000$.

e per un'espressione booleana:

```
A \(1/3\) l'identità è \emph{\expr{sin(1/3)^2 + cos(1/3)^2 == 1}}
```

A 1/3 l'identità è *vero*

Finora ogni nuova funzionalità aggiunta alla calcolatrice non ha presentato difficoltà. Possiamo inserire o meno il risultato in ambiente matematico, arrotondarlo al numero di decimali desiderato e usare funzioni matematiche dell'efficiente libreria in virgola mobile di Lua, un linguaggio che si sta dimostrando semplice da usare e molto efficace.

1.1.4 Costanti numeriche

Continuamo con un nuovo passo: aggiungere costanti numeriche definite dall'utente, una sorta di *funzione di memoria* della calcolatrice. Per inserire variabili letterali in un'espressione abbiamo bisogno che il loro valore

numerico sia inizializzato ma non possiamo ricorrere alla stessa tecnica con cui abbiamo risolto l'inserimento delle funzioni trigonometriche.

Non è possibile infatti codificare variabili locali senza conoscerne il nome, perché è un dato fornito dall'utente. Servirebbe una sorta di metaprogrammazione come con le macro dei linguaggi compilati. Leggendo più a fondo la documentazione di Lua, si scopre che è possibile intervenire sull'ambiente delle variabili globali `_ENV`[1] di un *chunk*, e anzi, a ben vedere il problema di rendere visibili simboli di costanti è lo stesso che quello di rendere disponibili nell'espressione le funzioni matematiche con nomi abbreviati. Facciamo un tentativo ripartendo con il codice da zero:

Tipo `table` § 5

Iteratore `pairs()` § 11.2

```
1   \directlua{
2   calclib = {}
3   for name, object in pairs(math) do
4       calclib[name] = object
5   end
6   }
7   \newcommand\expr[1]{\directlua{
8   do
9       local _ENV = calclib
10      ans = #1
11  end
12  tex.print(tostring(calclib.ans))
13  }}
```

Stiamo sfruttando una tecnica piuttosto interessante: all'interno di un blocco viene riassegnata localmente la variabile `_ENV` a `calclib`, una tabella in cui vi abbiamo riversato tutte le funzioni e le costanti matematiche della libreria `math` di Lua. Alla riga 10, tutti quei nomi saranno visibili come variabili globali proprio quando l'espressione deve essere valutata.

Non solo, come effetto collaterale, il risultato dell'ultimo calcolo sarà disponibile nella successiva espressione memorizzato nella variabile `ans` come succede con altri tool matematici! Proviamolo:

[1] Per ulteriori informazioni sull'*environment* di Lua rimando al PIL dove si trova un intero capitolo ad esso dedicato.

`\expr{pi/4}\\`	0.78539816339745
`\expr{cos(ans)}\\`	0.70710678118655
`\expr{acos(ans)}`	0.78539816339745

Molto bene. Non ci resta che aggiungere con la stessa tecnica la memorizzazione delle costanti attivando l'argomento opzionale della macro `\expr`. Tra le parentesi quadre potremo fornire all'espressione una lista di costanti nel formato chiave/valore con separatore il carattere virgola.

Memorizzeremo le costanti indicate dall'utente solamente se il loro nome non è già stato utilizzato o se non è un un nome di funzione. Inoltre, specificando una stringa solo come chiave tra le opzioni, potremo implementare la memorizzazione del risultato dell'espressione stessa, così che sia poi riutilizzabile:

```
\directlua{
calclib = {}
for name, object in pairs(math) do
    calclib[name] = object
end
}
\newcommand\expr[2][]{\directlua{
do
    local error, pairs, assert, type = error, pairs, assert, type
    local _ENV = calclib
    local opt = {#1}
    local mem = opt[1]; opt[1] = nil
    for c, val in pairs(opt) do
        if _ENV[c] then
            error("Duplicated key '"..c.."' in constant name")
        else
            _ENV[c] = val
        end
    end
    ans = #2
    if mem then
        assert(type(mem) == "string")
        if _ENV[mem] then
            error("Duplicated key '"..mem.."' in memory index")
        else
            _ENV[mem] = ans
```

```
            end
        end
    end
    tex.print(tostring(calclib.ans))
    }}
```

Eccone un esempio:

```
\( b = \expr[b=10, h=20]{b} \), % oppure \expr["b", h=20]{10}
\( h = \expr{h} \),
\( M = \expr[m = 1000]{m}\), % oppure \expr["m"]{1000}
\( \sigma = M/W_\mathrm{x} = \expr[w=(b*h^2)/6]{m/w}\).
```

$b = 10$, $h = 20$, $M = 1000$, $\sigma = M/W_{\mathrm{x}} = 1.5$.

Poiché anche i valori assegnati alle costanti sono valutati da Lua *dopo* la modifica dell'environment, anche per le costanti nelle opzioni della macro `\expr` è possibile assegnare espressioni usando tutte le funzioni matematiche e tutte le costanti precedentemente definite. Da questo punto in poi, possiamo presentare il valore di W_{x} scrivendo nel sorgente `\expr{w}` che da 666.66666666667.

1.1.5 Conclusioni

Tutte le principali funzionalità della calcolatrice sono state implementate in Lua, certo non tutte. Per esempio potremo far eseguire calcoli coinvolgendo anche registri contatori o dimensionali di TEX, per esempio per fornire coordinate in un disegno, oppure considerare che constanti dai nomi speciali come `M1`, `M2` eccetera si comportino come registri di memoria della calcolatrice e quindi che possano essere sovrascritti o possano funzionare da accumulatori.

1.2 La tabella dei pesi

Dopo la calcolatrice si presenta un altro problema compositivo: una tabella che riporta per vari diametri, area e peso della barra d'acciaio di lunghezza unitaria. I diametri variano da 6 a 32 millimetri con passo 2.

L'idea è definire una sorta di iteratore a due componenti. Per esempio, se volessimo una tabella con due colonne, la prima con gli interi da 1 a 10 e

la seconda con i rispettivi quadrati, dovremo poter definire solo la regola di costruzione delle righe. Un secondo componente si occuperà di applicarla le volte necessarie.

Il primo componente è una funzione generatrice, input del secondo componente, che può essere qualsiasi purché sia definita sempre con due argomenti: il primo è il contatore di riga e il secondo è l'array di riga. Nel caso d'esempio si dovrà memorizzare nell'array in posizione 1 il contatore stesso e in posizione 2 il suo quadrato:

```
1  local function row_func(counter, row)
2      row[1] = counter
3      row[2] = counter^2
4  end
```

A ben vedere potremo fare a meno del secondo parametro `row` se restituissimo direttamente un nuovo array di riga, tuttavia in questo modo il codice risulta più efficiente.

L'idea iniziale si concretizza nell'attribuire alla funzione che calcola la generica riga il ruolo di *regola di definizione* dell'intera tabella. Il ruolo del secondo componente è assunto dalla classe `Row`. Essa ha il compito di applicare la regola qualsiasi essa sia, riga dopo riga, senza conoscerla. Tra il primo e il secondo componente vi è quindi una netta separazione e questo è indice di buona progettazione.

Esaminiamo la costruzione della tabella d'esempio in LuaLaTeX nel listato 1.1. Il metodo `new()` della classe `Row` accetta proprio una funzione come primo argomento e il valore totale delle righe come secondo argomento. In Lua le funzioni sono valori come tutti gli altri.

All'interno dell'ambiente *tabular* c'è solo codice Lua: costruito l'oggetto `row` un ciclo `while` esegue l'iterazione con il metodo `next()`. Come si può verificare dall'implementazione con il paradigma a oggetti che segue, è proprio `next()` a chiamare a ogni passo la funzione di generazione:

Metametodo `__index` § 12.6

Costruttore § 12.7

```
1  Row = {}
2  Row.__index = Row
3  -- costruttore
4  function Row:new(fn_next, start, stop, step)
5      if not stop then
6          start, stop = 1, start
7      end
8      local o = {
```

Listing 1.1: Listato parziale per la tabella a due colonne

```
% !TeX program = LuaLaTeX
% filename: app-tut/E0-003-tab.tex
\documentclass{article}
% preambolo non riportato
\begin{document}
\begin{tabular}{rr}\directlua{
local row = Row:new(
    function (c, r) r[1]=c; r[2]=c^2 end, 10
)
local par = string.char(92)..string.char(92)
while row:next() do
    tex.print(row[1].."&"..row[2]..par)
end
}\end{tabular}
\end{document}
```

1	1.0
2	4.0
3	9.0
4	16.0
5	25.0
6	36.0
7	49.0
8	64.0
9	81.0
10	100.0

```
        fn_next = fn_next,
        start = start,
        stop = stop,
        step = step or 1
    }
    setmetatable(o, self)
    return o
end
-- iteratore
function Row:next()
    local var = self.var
    if not var then
        var = self.start
    else
        var = var + self.step
    end
    if var <= self.stop then
        self.var = var
        local fn = self.fn_next
        fn(var, self)
        return true
    end
```

```
end
```

Siamo in fase iniziale e questo giustifica l'assenza di controlli sui dati di input e la gestione degli errori, parte essenziale di ogni programma. Ci si potrà preoccupare in seguito in una fase di consolidamento, di errori e altri dettagli.

Per esempio, stiamo trascurando le conseguenze possibili del fatto che il codice Lua è all'interno del sorgente TEX e che per questo ci potrebbero essere effetti non desiderati dovuti all'espansione dell'argomento della primitiva `\directlua`. Un esempio? Riceveremmo un errore con il blocco della compilazione se nel preambolo caricassimo il pacchetto *`polyglossia`* con l'opzione *babelshorthands* per la lingua italiana.

Nel codice abbiamo infatti usato il doppio apice come delimitatore dei valori letterali delle stringhe, carattere che diviene attivo con quelle impostazioni. Per fortuna ci sono altri modi più sicuri di delimitare i valori letterali delle stringhe in Lua in questi casi, proprie del linguaggio.

1.2.1 Dati tabellari

La funzione generatrice è generale potendo costruire qualsiasi tipo di vista tabellare sui dati, sia essi ricavati con il calcolo come nel caso precedente oppure già definiti in memoria, per esempio come una lista di nomi di file con relativa dimensione in byte:

```
local data = {
    {"files.txt",  4710},
    {"lib.lua"  ,   330},
    {"parse.lua", 50995},
    {"path.txt" ,  2150},
}
```

Come dovremo definire il primo componente, la funzione `row_func()`, per costruire la tabella a due colonne nome e dimensione file? La risposta è nel listato 1.2. Nel codice incontriamo una ridondanza perché dobbiamo specificare il numero di righe nel costruttore `new()` quando questo dato è invece derivabile dai dati.

In Lua è normale poter chiamare una funzione variando il tipo dei suoi argomenti Se passassimo al metodo `new()` direttamente la tabella dati anziché la funzione generatrice, il codice sarebbe in grado di rilevare la

Listing 1.2: Generazione della tabella da un set di dati

```
% !TeX program = LuaLaTeX
% filename: app-tut/E0-004-tab.tex
\documentclass{article}
% preambolo non riportato
\begin{document}
\begin{tabular}{lr}\directlua{
local data = {
    {"files.txt",  4710},
    {"lib.lua"  ,   330},
    {"parse.lua", 50995},
    {"path.txt" ,  2150},
}
local function row_func(counter, row)
    row[1] = data[counter][1]
    row[2] = data[counter][2]
end
local row = Row:new(row_func, 4)
local p = string.char(92); p = p..p
while row:next() do
    tex.print(row[1].."&"..row[2]..p)
end
}\end{tabular}
\end{document}
```

files.txt	4710
lib.lua	330
parse.lua	50995
path.txt	2150

differenza e costruire sia la funzione `row_func()` che il numero totale di righe.

In effetti nel codice del file `E0-004-tab.tex` si trova un'implementazione che fa proprio questo. Questa forma di polimorfismo interno del costruttore può essere convertita in una forma esterna a beneficio dell'interfaccia che esprime in modo diretto all'utente l'adattibilità della classe, nonché della chiarezza del codice, implementando più di un costruttore, uno per ciscun tipo di dato in ingresso con cui è possibile costruire la classe.

Per esempio già i nomi dei costruttori `from_func()` o `from_file()` definiscono esplicitamente il tipo di sorgente dati da cui costruire l'oggetto. Aggiungere un nuovo tipo di sorgente per i dati in forma tabellare significa aggiungere un nuovo costruttore. Il codice dei costruttori già implementati

non ne risente mentre invece se il polimorfismo fosse interno, non potremo evitare di modificare l'unico costruttore.

1.2.2 Template di riga

Proseguiamo adesso con il migliorare il modo in cui generare il codice di riga nell'ambiente `tabular`. Stiamo infatti usando la concatenazione di stringhe, un modo non molto efficiente e nemmeno comodo. Potrebbe essere più conveniente specificare una sorta di template con segnaposto come la stringa seguente per un'ipotetica tabella a due colonne:

```
template = [[\textbf{<1>} & <2>\\]]
```

Il numero tra parentesi acute `<>` indica l'indice di riga corrispondente a quello definito dalla funzione `row_func()`.

Per implementare il template di riga, basta aggiungere alla classe `Row` un iteratore che a ogni passo generi la stringa di codice della riga di tabella:

`string.gsub()` § 10.3.5
Pattern § 10.3.2
Capture § 10.3.3
Stateless iterator § 11.5

```
function Row:iter_template(tmpl)
    local iter_fn = function(row, i)
        if not i then
            i = row.start
        else
            i = i + row.step
        end
        if i <= self.stop then
            self.fn_next(i, self)
            local s = tmpl:gsub("<(%d+)>", function (s)
                local n = tonumber(s)
                return row[n]
            end)
            return i, s
        end
    end
    return iter_fn, self, nil
end
```

Al di la di considerazioni di efficienza legate all'uso della funzione di libreria `gsub()`, l'iteratore in effetti funziona come dimostra il seguente codice per LuaLaTeX estratto dal file `app-tut/E0-005-tab.tex` parte della

guida. Abbiamo inserito la macro `\noexpand` per bloccare l'espansione delle control sequence[2]:

```
\begin{tabular}{lr}
\directlua{
local tmpl = [[\noexpand\textbf{<1>} & <2>\noexpand\\]]
for _, s in row:iter_template(tmpl) do
   tex.print(s)
end
}
\end{tabular}
```

Torniamo alla nostra tabella dei pesi. La funzione generatrice e il template di riga saranno le seguenti:

```
local function row_func(diam, row)
    row[1] = diam
    local area = math.pi * (diam/20)^2
    local fmt = string.char(37)..'0.3f'
    row[2] = fmt:format(area)
    row[3] = fmt:format(0.785*area)
end
row = Row:new(row_func, 6, 32, 2)
tmpl = [[\noexpand\textbf{<1>} & <2> & <3>\noexpand\\]]
```

e il risultato è:

6	0.283	0.222
8	0.503	0.395
10	0.785	0.617
12	1.131	0.888
14	1.539	1.208
16	2.011	1.578
18	2.545	1.998
20	3.142	2.466
22	3.801	2.984
24	4.524	3.551
26	5.309	4.168
28	6.158	4.834
30	7.069	5.549
32	8.042	6.313

[2]Certo mi ostino ancora a non utilizzare il pacchetto *luacode*.

Miglioriamo ora il codice della funzione generatrice aggiungendo il metodo `insert()` alla classe `Row`, a tre argomenti: il numero di colonna `col`, il valore da inserire nella cella `val` e infine il valore opzionale di arrotondamento numerico `prec`. Eccone una sua implementazione molto semplice:

```
function Row:insert(col, val, prec)
    if prec then
        local p = string.char(37)
        local fmt = string.format(p..p.."0."..p.."df", prec)
        val = string.format(fmt, val)
    end
    self[col] = val
    return self
end
```

Il nuovo metodo restituisce l'oggetto stesso così che possiamo concatenare più inserimenti di cella. Ecco come la funzione di generazione può semplificarsi:

```
local function row_func(diam, row)
    local area = math.pi * (diam/20)^2
    row:insert(1, diam)
       :insert(2, area, 3)
       :insert(3, 0.785*area, 3)
end
```

Sono scomparse le acrobazie per il formato numerico a favore della compattezza. Un ulteriore miglioramento ci consente di evitare di dover controllare l'espansione quando inseriamo il testo del template di riga grazie al comando `\detokenize`.

Introduciamo in proposito una nuova macro `\printrow` che ha come argomento il template che rappresenta il modello della generica riga della tabella:

```
\newcommand{\printrow}[1]{\directlua{
local tmpl = [=[\detokenize{#1}]=]
for _, s in row:iter_template(tmpl) do
   tex.print(s)
end
}}
```

Per non introdurre un secondo argomento, nell'istanziare l'oggetto della classe `Row` dovremo solo ricordarci di chiamare la variabile `row`, lo stesso nome usato nella definizione di `\printrow`. Mettiamo subito al lavoro la nuova macro:

```
\begin{tabular}{lrr}
\printrow{\textbf{<1>} & <2> & <3>\\}
\end{tabular}
```

Molto semplice: si definisce prima la funzione generatrice e con essa si costruisce l'oggetto `Row`, poi si scrive il codice LaTeX dando alla macro `\printrow` il template con i segnaposto.

Molto importante è far corrispondere i numeri di cella dei segnaposto del template con i valori che la funzione di riga inserisce nella varie posizioni.

1.2.3 Composizione finale

L'ultimo passo è migliorare l'aspetto della tabella. Con il pacchetto *`booktabs`* aggiungiamo un'intestazione e un filetto ogni tre righe per facilitare la lettura dei dati. Dobbiamo così modificare la funzione di riga per derminare se il numero di riga è multiplo di tre — senza usare l'operatore modulo `%` di Lua perché non ci troviamo in un file esterno:

```lua
local function fn(diam, row)
    local area = math.pi * (diam/20)^2
    local peso = 0.785*area
    local c = row.counter
    local midrule = ""
    if c - 3*math.floor(c/3) == 0
    and not (diam == row.stop) then
        midrule = string.char(92)..."midrule"
    end
    row:insert(1, diam)
       :insert(2, area, 3)
       :insert(3, peso, 6)
       :insert(0, midrule)
end
```

Introduciamo anche il pacchetto *`siunitx`* [Wri20] utilissimo per comporre numeri, unità di misura e tabelle, con questo ambiente *`tabular`* ridisegnato:

```
\begin{tabular}{
    c
    S[table-format=4.3]
    S[table-format=1.3]
    S[table-format=1.6]
}
\toprule
\diameter & {Sviluppo} & {Sezione} & {Peso}\\
\small\si{mm} & {\small\si{cm^2/m}} & {\small\si{cm^2}} &
    {\small\si{daN/m}}\\
\midrule
\printrow{\(\mathbf{<1>}\) & <4> & <2> & <3>\\<0>}
\bottomrule
\end{tabular}
```

Il progetto completato si trova nel file `app-tut/E0-006-tab.tex`, dove ho aggiunto alla tabella la colonna con il calcolo della superficie laterale delle barre.

1.2.4 Conclusioni

La nostra classe `Row` ci permette di costruire tabelle iterative in Lua in modo del tutto generale, compiendo calcoli numerici e ogni sorta di possibili elaborazioni. Molti altri affinamenti sono possibili come il caricamento di dati esterni oppure l'uso di pipeline di operatori all'interno dei segnaposto dei template. Anche questo tutorial si chiude perciò con una lista di nuove idee da implementare con Lua.

La tabella 1.1 mostra il risultato finale.

Tabella 1.1 La tabella dei pesi generata automaticamente definendone la regola di generazione e il template di riga.

∅ mm	Sviluppo cm^2/m	Sezione cm^2	Peso daN/m
6	188,496	0,283	0,221 954
8	251,327	0,503	0,394 584
10	314,159	0,785	0,616 538
12	376,991	1,131	0,887 814
14	439,823	1,539	1,208 414
16	502,655	2,011	1,578 336
18	565,487	2,545	1,997 582
20	628,319	3,142	2,466 150
22	691,150	3,801	2,984 042
24	753,982	4,524	3,551 256
26	816,814	5,309	4,167 794
28	879,646	6,158	4,833 654
30	942,478	7,069	5,548 838
32	1005,310	8,042	6,313 345

Sul sistema TEX e Lua 2

Questo capitolo fornisce informazioni sulla differenza tra motore di composizione e formato, e sull'esecuzione di codice Lua all'interno di un sorgente TEX.

2.1 Motori di composizione e formati

Un sorgente TEX contiene testo e macro. Il testo formerà i capoversi, i titoli e il resto del documento, mentre le macro ne stabiliranno l'aspetto e la struttura. I *motori di composizione* costruiscono il documento eseguendo macro dette *primitive*, sono cioè implementate in modo nativo per svolgere compiti elementari, e macro definite con esse per svolgere sequenze ricorrenti.

Le macro sequenze dette di alto livello, possono costituire una vera e propria libreria il cui codice è generalmente scritto da veri esperti, che prende il nome di *formato*, come LATEX o ConTEXt, perché per esse ne è stata ricercata la coerenza interna e sono state stabilite nuove regole di sintassi comune.

Per esempio, nel formato LATEX è presente il concetto di *ambiente* con la coppia di macro di delimitazione `\begin{}` e `\end{}`, oppure quello di suddivisione del sorgente in preambolo e in corpo del documento.

Con le macro primitive o con quelle del formato, l'utente può implementare funzionalità specifiche oppure caricare macro definite in pacchetti o moduli che integrano o estendono quelle del formato.

I motori di composizione hanno l'abilità nella fase iniziale della compilazione di caricare il formato in forma precompilata. Se si avvia un qualsiasi programma di composizione della famiglia TEX verrà caricato di default il formato più semplice chiamato *plain*. Se si vuole invece utilizzarne uno

diverso, per esempio il più diffuso LATEX, occorre specificarne il nome al compilatore con l'opzione `--fmt` nel comando al terminale.

Tuttavia, data l'importanza per gli utenti dei formati di alto livello, sono stati predisposti appositi comandi scorciatoia. Per esempio il programma pdflatex rimanda all'effettivo motore di composizione pdftex con l'istruzione di caricare il formato LATEX, e così anche per lualatex con luatex.

Per essere chiari, i due comandi seguenti sono del tutto equivalenti, nel primo si invoca direttamente il motore di composizione mentre nel secondo lo si fa con una scorciatoia:

```
$ pdftex --fmt=pdflatex mydoc
$ pdflatex mydoc
```

Quando il nome del formato corrisponde al nome del programma scorciatoia, non li si deve confondere: il programma latex utilizza ancora il compositore pdftex con il formato LATEX con uscita in DVI[1].

I file binari dei formati sono usualmente generati o rigenerati in automatico dalle utility della distribuzione nel momento dell'installazione o quando l'utente esegue un aggiornamento del sistema che contiene modifiche alle macro del formato, perciò è molto raro che ci si trovi nella situazione di doverne generare uno.

Riassumendo, i motori di composizione sono programmi di composizione tipografica mentre i formati sono insiemi coerenti di macro basate sulle primitive del compositore. I nomi dei programmi disponibili nel sistema TEX possono confondere se non si conosce questa importante distinzione: alcuni di essi sono infatti comandi scorciatoia per eseguire un motore di composizione con un particolare formato.

2.1.1 Compositori Lua-powered

LuaTEX è un programma che elabora un file di testo contenente codice TEX per comporne il corrispondente file PDF, l'uscita di default, quindi è un motore di composizione. Nella famiglia TEX ci sono almeno altri due compositori dotati dell'interprete Lua, LuaHBTEX e LuaJITTEX.

Tutti e tre questi compositori possono eseguire il formato LATEX. Come detto in apertura di capitolo, il programma lualatex esegue un compositore

[1] DVI sta per Device Indipendent e oggi è in disuso a favore del formato PDF. Tuttavia è utile far notare che un programma compositore può fornire anche più formati di uscita.

che carica il formato LaTeX. Dalla TeX Live 2020 questo compositore è luahbtex. Per rendercene conto basta scrivere in un terminale il nome del programma con l'opzione `--version`:

```
> lualatex --version
This is LuaHBTeX, Version 1.13.0 (TeX Live 2021)

Execute  'luahbtex --credits'  for credits and version details.
...
```

Per compilare un sorgente LaTeX con LuaLaTeX, con la TeX Live 2021 i due comandi equivalenti sono:

```
$ luahbtex --fmt=lualatex mydoc
$ lualatex mydoc
```

Per ulteriore informazione, luahbtex è il motore di composizione luatex in cui è stato sostituito il componente per il calcolo della forma dei font con il modulo HarfBuzz [Esf21], mentre luajittex è un'altra variante di luatex in cui l'interprete Lua è stato sostituito con LuaJIT [Pal21] un'implementazione indipendente più veloce dell'interprete ufficiale che sfrutta le tecniche di compilazione denominate *Just In Time*.

In generale un sorgente TeX che contiene codice Lua viene correttamente compilato da qualsiasi dei tre compositori grazie al mantenimento della compatibilità.

2.1.2 Codice Lua in LuaTeX

Per illustrare l'esecuzione di codice Lua all'interno di un sorgente LuaTeX, consideriamo la stampa nell'output di console di un testo semplice. Il codice Lua va inserito come argomento della primitiva `\directlua`:

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

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

```
$ luahbtex primo
```

e per il sistema operativo Linux e la distribuzione TeX Live 2021, l'output in console è:

```
This is LuaHBTeX, Version 1.13.0 (TeX Live 2021)
restricted system commands enabled.
(./primo.texHello World!
)
warning  (pdf backend): no pages of output.
Transcript written on t.log.
```

Il testo uscirà tra gli altri messaggi di output senza che sia stato prodotto un file PDF. Ciò significa che `\directlua` è una macro espandibile con risultato vuoto.

2.1.3 Codice Lua in LuaLATEX

Con LuaLATEX si ottiene lo stesso risultato ma con il sorgente scritto nella sintassi LATEX, ovvero:

```
% !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 `\directlua` ne *espande* l'argomento e passa a Lua il controllo che esegue immediatamente il codice per poi restituirlo di nuovo a TEX al termine dell'esecuzione.

2.2 La primitiva `directlua`

Abbiamo ora tutte le informazioni per svolgere alcuni esercizi e applicazioni. Non rimane che ricordare che 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 ed eseguito come blocco, può quindi contenere macro o argomenti macro con un testo di sostituzione;
2. le variabili locali hanno validità solo all'interno del gruppo/blocco mentre quelle globali saranno valide anche in quelli di successive `\directlua`;
3. l'espansione di `\directlua` è vuota;

Per documentarsi nel dettaglio esiste un'ottima risorsa su Overleaf, la popolare piattaforma di compilazione e gestione progetti LaTeX online, questo indirizzo web.

2.3 Passaggio di dati

La comunicazione dati bidirezionale tra TeX e Lua può avvenire con la tecnologia dei nodi, come vedremo, certamente quella più avanzata e complessa ma non è l'unica: i dati possono arrivare a Lua tramite l'espansione, mentre nella direzione opposta è possibile scrivere del testo nella lista di input del compositore con le funzioni della famiglia `tex.print()`.

Interrompete la lettura della guida per provare a scrivere la funzione Lua `fact()` che calcola il fattoriale di un intero. La useremo per dimostrare come avviene questo scambio di dati. L'idea è definire un nuovo comando che stampi il fattoriale del numero argomento:

```
1  \newcommand{\fattoriale}[1]{\directlua{
2      local n = #1
3      tex.print(tostring(fact(n)))
4  }}
```

Quando TeX incontra una macro utente con un segnaposto[2] lo sostituisce con i dati corrispondenti a un singolo token oppure a un gruppo di token tra parentesi graffe. Per questo quando nel sorgente scriviamo `\fattoriale{5}`, dopo la sostituzione il codice Lua effettivamente eseguito sarà:

```
1  local n = 5
2  tex.print(tostring(fact(n)))
```

[2] Fate riferimento ha una buona guida per LaTeX per saperne di più sulla definizione di macro utente.

La funzione `tex.print()` inserisce l'argomento stringa nell'input d'ingresso come se fossero stati letti dal sorgente. Quando a termine l'esecuzione del blocco di codice Lua della macro `\fattoriale` i successivi caratteri che si troverà a elaborare il compositore saranno le cifre 1, 2, e 0 e l'espansione della macro sarà stata vuota.

Il listato del sorgente completo dell'esercizio proposto da confrontare con la vostra personale soluzione è:

```
% !TeX program = LuaLaTeX
% filename app-tut/E001-fatt.tex
\documentclass{article}
\newcommand{\fattoriale}[1]{\directlua{
    local n = assert(tonumber(#1))
    local res = 1
    for i = 1, n do
        res = res * i
    end
    tex.print(tostring(res))
}}
\begin{document}
\( 5! = \fattoriale{5} \)
\end{document}
```

2.4 Globale o locale

Prendendo spunto ancora dall'esempio precedente, soffermiamoci sul comportamento nei diversi blocchi `\directlua` delle definizioni locali e globali: come descritto dalle specifiche di Lua, tutto quello che è locale a un blocco non è più disponibile al di fuori di esso. Nel contesto di LuaTEX il codice contenuto in una macro `\directlua` forma un blocco.

Per questo motivo se separassimo il codice che calcola il fattoriale dalla definizione della macro utente `\fattoriale`, per non ricevere un errore di chiamata di un valore `nil` dal secondo blocco dovremo definire la funzione come globale come in:

```
% !TeX program = LuaLaTeX
% filename app-tut/E1-002-fatt.tex
\documentclass{article}
\directlua{
function fact(n)
```

```
    local res = 1
    for i = 1, n do
        res = res * i
    end
    return res
end
}

\newcommand{\fattoriale}[1]{\directlua{
    local n = assert(tonumber(#1))
    tex.print(tostring(fact(n)))
}}

\begin{document}
\( 5! = \fattoriale{5} \)
\end{document}
```

Le definizioni globali tuttavia possono determinare errori dovuti alla collisione dei nomi definiti in altre parti del sorgente essendo l'ambiente appunto globale e quindi unico. Spesso, come per gli esempi della guida, non è un problema ma è buona norma definire una tabella di *namespace* dove memorizzare le funzioni limitando la collisione dei nomi solamente al nome del riferimento alla tabella.

Questa buona prassi diviene *obbligatoria* quando stiamo scrivendo codice applicativo in contesti di utilizzo reale.

Come ulteriore esempio, nel listato[3] che segue è mostrato come si possa formattare il numero del fattoriale con il pacchetto `siunitx`:

```
% !TeX program = LuaLaTeX
% filename: app-tut/E1-003-fatt.tex
\documentclass{article}
\usepackage{siunitx}
% tabella di namespace
\directlua{
assert(not app)
app = {}
function app.fact(n)
    local res = 1
    for i = 1, n do
```

[3]Questo listato è interessante perché migliorabile sia per eliminare le ripetizioni di codice a cui è costretto l'utente sia nell'efficienza di esecuzione.

```
            res = res * i
        end
        return res
    end
    }
    % user command
    \newcommand{\fattoriale}[1]{\num{\directlua{
        local n = assert(tonumber(#1))
        tex.print(tostring(app.fact(n)))
    }}}
    \begin{document}
    \noindent\( 10! = \fattoriale{10} \)\\
    \( 11! = \fattoriale{11} \)\\
    \( 12! = \fattoriale{12} \)\\
    \( 13! = \fattoriale{13} \)\\
    \( 14! = \fattoriale{14} \)\\
    \( 15! = \fattoriale{15} \)\\
    \( 16! = \fattoriale{16} \)\\
    \( 17! = \fattoriale{17} \)\\
    \( 18! = \fattoriale{18} \)
    \end{document}
```

2.5 Espansione di macro

La comunicazione di un dato tra TEX e Lua può avvenire anche per espansione di macro. Come esempio minimo consideriamo il seguente sorgente LuaTEX che stampa l'ora di inizio della compilazione avvalendosi del contatore `\time` che indica i minuti trascorsi dall'inizio del giorno:

```
% !TeX program = LuaTeX
% filename: app-tut/E1-004-time.tex
\directlua{
    local time = \the\time
    local h = math.floor(time/60)
    local m = time - h*60
    local percent = string.char(37)
    local fmt_hhmm = percent.."02d:".. percent.."02d"
    tex.print(string.format(fmt_hhmm, h, m))
}
\bye
```

Al termine dell'espansione l'istruzione di assegnazione della variabile numerica `time` sarà l'effettiva e corretta sintassi Lua. Per il formato LuaLaTeX lo stesso file potrebbe essere:

```
% !TeX program = LuaLaTeX
% filename: app-tut/E1-005-time.tex
\documentclass{article}
\begin{document}
\directlua{
    local time = \the\time
    local h = math.floor(time/60)
    local m = time - h*60
    local percent = string.char(37)
    local fmt_hhmm = percent.."02d:".. percent.."02d"
    tex.print(string.format(fmt_hhmm, h, m))
}
\end{document}
```

Nel capitolo 13 vedremo che anche con la libreria standard di Lua è possibile ottenere la data e l'ora corrente.

2.6 Caratteri speciali

Alcuni simboli hanno un diverso significato per TeX e per Lua, per esempio il carattere cancelletto, lo stesso backslash o il simbolo di percento. LuaTeX non si occupa direttamente di modificare il significato dei simboli che si sovrappongono.

Le descrizioni dei conseguenti errori possono essere criptici ma ci sono almeno quattro diverse soluzioni:

- scrivere il codice Lua in file esterni,
- utilizzare codice TeX per gestire l'espansione dei simboli o i codici di categoria,
- usare i codici ASCII per creare stringhe che contengono i simboli, con la funzione `string.char()`,
- utilizzare il pacchetto *luacode*.

Come preferenza personale non utilizzo l'ottimo pacchetto *luacode*, anche per evitare una dipendenza in più. Tuttavia non appena il codice Lua cresce in numero di linee o diventa un componente di un progetto reale,

quasi sempre si devono utilizzare file esterni per una più agevole gestione del progetto.

Rimando alla documentazione del pacchetto *luacode* per i dettagli sulla collisione dei simboli. Vedrete che esso offre due comandi e due ambienti per poter far inserire codice Lua in un sorgente LATEX con varie impostazioni di caratteri.

Nel codice Lua si possono usare i commenti in stile TEX con il simbolo del percento perché il processo d'espansione li elimina *prima* di passare il codice all'interprete, ma non si possono usare i commenti in stile Lua con il doppio trattino a meno che non siano all'ultima linea. Infatti, per la stessa espansione tutto il codice Lua nella macro `\directlua` è inviato all'interprete come un unica linea di codice perciò il commento dopo un doppio trattino si estenderebbe non solo a fine riga ma a tutto il codice che segue.

Per fortuna la grammatica Lua a differenza di Python, consente la libera scrittura e i rientri del codice altrimenti questo meccanismo non potrebbe funzionare.

2.7 Le librerie disponibili in LuaTEX

Agli usuali moduli della libreria standard di Lua, sono stati aggiunti in LuaTEX, così come negli altri motori di composizione estesi, speciali funzionalità dedicate al controllo dello stato interno e alla creazione di elementi tipografici che chiameremo librerie interne, e funzionalità generiche che chiameremo librerie esterne.

Tutte queste librerie formano un nutrito elenco come riporta dettagliatamente il manuale dell'utente. Tra quelle interne più note ricordiamo:

- `node`, gestione dei nodi tipografici;
- `token`, scansione dei token per la raccolta di argomenti, creazione di macro da codice, e molto altro;
- `tex`, funzioni di ingresso dell'input del compositore, accesso a registri e parametri interi;
- `harfbuzz`, informazioni sui font e i singoli glifi (solo per il compositore luahbtex);
- `mplib`, METAPOST per la grafica vettoriale.

Tra le librerie esterne troviamo:

- `lfs`, Lua File System per visitare l'albero delle directory e ottenere informazioni sui file;
- `socket`, connessioni di rete con protocolli SMTP, HTTP e FTP;
- `luazip`, compressione e decompressione di file e cartelle;
- `lpeg`, Parsing Expression Grammar, dati da testo strutturato.

Alcune di queste librerie richiedono l'autorizzazione esplicita dell'utente all'esecuzione per mezzo di opzioni di compilazione.

2.8 LaTeX con pdflatex o lualatex

I sorgenti LaTeX possono essere compilati sia con il tradizionale motore `pdftex` che con il compositore `luahbtex` con minime modifiche. Vediamo quali sono affiancando i preamboli minimi nei due casi:

```
% !TeX program = pdfLaTeX
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage[italian]{babel}
```

```
% !TeX program = LuaLaTeX
\documentclass{article}
\usepackage{fontspec}
\usepackage[italian]{babel}
```

Nel preambolo di un sorgente scritto per `pdflatex` si caricano tre pacchetti fondamentali *`fontenc`* per la codifica dei font, *`inputenc`* per quella del sorgente, e *`babel`* per le lingue.

Oggi la stragrande maggioranza degli utenti, e se non siete tra loro fatelo subito, codificano i file sorgenti con UTF-8. Il testo in Unicode infatti elimina ogni incoveniente quando si condividono i file tra diversi sistemi operativi[4].

Per LuaTeX è obbligatoria la codifica UTF-8 perciò la prima modifica al preambolo è semplicemente l'eliminazione del pacchetto *`inputenc`*. Se i vostri file non fossero codificati UTF-8, si può utilizzare un tool come `iconv` come spiegato nella guida tematica menzionata in nota.

Per i font non è più necessario il pacchetto *`fontenc`*. Per selezionarne nel documento si usa il pacchetto *`fontspec`* e le sue macro `\setmainfont`, `\setmonofont` eccetera. I font matematici vanno invece gestiti con il pacchetto *`unicode-math`* e con la macro `\setmathfont`.

[4]Per informazioni dettagliate sulle codifiche rimando alla guida tematica GuIT Introduzione alle codifiche in entrata e uscita di Claudio Beccari e Tommaso Gordini [BG17].

L'immediato vantaggio di passare da pdfLATEX a LuaLATEX è proprio nel poter comporre il documento con i font Open Type. Per la mia esperienza la compatibilità con i numerosi pacchetti LATEX è ottima mentre è possibile utilizzare i pacchetti scritti in Lua ormai numerosi.

Ulteriori informazioni si possono trovare nel documento `lualatex-doc`, al solito disponibile a video con l'utility `texdoc`, dove viene considerato anche il compositore XƎLATEX.

Parte II

Fondamenti del linguaggio Lua

Come eseguire gli esercizi 3

È certamente fondamentale eseguire noi stessi esempi ed esercizi di programmazione allo scopo di acquisire la padronanza di Lua. Nella guida ne trovate alcuni alla fine di ciascun capitolo della parte seconda.

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.

Rispetto all'interprete lua standard texlua non ha la modalità interattiva REPL[1] con cui si digita una linea di codice alla volta in un prompt interattivo, modalità molto utile per fare prove velocemente.

Il codice dunque, andrà memorizzato in un file con estensione `.lua`. Come esempio elementare, digitiamo questa unica riga di codice in un file di testo chiamato `primo.lua`:

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

apriamo una finestra di terminale[2] e lanciamo il comando:

```
$ texlua primo.lua
```

Ora che sappiamo come eseguire il codice Lua, concentriamoci con i prossimi capitoli sulle basi del linguaggio. Torneremo nella terza parte della guida su ulteriori modalità di esecuzione anche per il codice Lua interno a sorgenti TEX.

[1] Read–eval–print loop.

[2] Maggiori dettagli per diversi sistemi operativi sulla linea di comando possono essere trovati nella guida tematica dedicata *Guida alla console* scaricabile dal sito GUIT.

Assegnazione e tipi predefiniti 4

4.1 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. In portoghese "Lua" si pronuncia LOO-ah e significa "Luna"!

4.2 L'assegnazione

Ci occupiamo ora di uno degli elementi di base dei linguaggi informatici: l'istruzione di *assegnazione*. 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:

```
1   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 assegnazione omette il tipo di dato non essendone prevista una dichiarazione esplicita. In altre parole, i dati hanno un tipo,

determinabile con la funzione `type()`, ma ciò non ha rilevanza semantica ed è solo in fase di esecuzione che l'interprete determina il tipo di dato e in funzione di questo ne memorizza il valore in memoria.

Altro concetto importante di Lua è che le variabili sono tutte globali a meno che non le si dichiari locali al blocco di codice.

4.2.1 Locale o globale?

Una proprietà dell'assegnazione è che se non diversamente specificato Lua crea 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.

4.2.2 Assegnazioni multiple

In Lua possono essere assegnate più variabili alla volta nella stessa istruzione con una sintassi in realtà più complessa di quella presentata fino a ora. A sinistra del simbolo di uguale è ammessa una lista di variabili separate da virgole e a destra una lista di espressioni sempre separate da virgole che, una volta valutate, saranno assegnate in ordine.

Se si premette la parola chiave `local` tutte le variabili saranno locali. La sintassi generale dell'assegmento multiplo è la seguente:

```
1   [local] var_1, var_2, var_3, ... = expr_1, expr_2, expr_3, ...
```

perfettamente equivalente a:

```
1   [local] var_1 = expr_1
2   [local] var_2 = expr_2
3   [local] var_3 = expr_3
```

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

Così

```
1  local a, b = 0.45 + 0.23, "text" -- a = 0.68; b = "text"
```

è equivalente a:

```
1  local a = 0.45 + 0.23
2  local b = "text"
```

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

```
1  local a, b, c = 0.45, "text"  -- c vale nil
2  local x, y = "op", "qw", "lo" -- "lo" è un dato ignorato
```

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:

```
1  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  local pi = 3.14159
2  local r = 10.8 -- raggio del cerchio
3  -- grandezze cerchio
4  local diam, circ, area = 2*r, 2*pi*r, pi*r^2
5  -- stampa grandezze
6  print("Diametro:", diam)
7  print("Circonferenza:", circ)
8  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.

4.3 Una manciata di tipi

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

- `number` il tipo numerico[2];
- `table` il tipo tabella → capitolo 5;
- `boolean` il tipo booleano → capitolo 7;
- `string` il tipo stringa → capitolo 8;
- `function` il tipo funzione → capitolo 9.
- `nil` il tipo nullo → sezione 4.3.1;

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.

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

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

4.4 Gli identificatori

I nomi che possiamo dare a variabili e funzioni, sono stringhe di lettere maiuscole o minuscole, numeri e trattini bassi `'_'` purché non comincino

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

con una cifra e non corrispondano alle *keyword*, le parole riservate del linguaggio.

In Lua gli identificatori sono *case sensitive*: le stesse lettere formano identificatori distinti se esse sono diverse nel maiuscolo o minuscolo, come per `vaR` e `Var`.

Si è quindi liberi di utilizzare nomi qualsiasi, tuttavia è conveniente aderire a qualche *convenzione* che aggiunga al nome un significato di categoria. Per esempio, gli identificatori che iniziano con una lettera maiuscola si possono riservare agli oggetti, come nel capitolo 12, oppure quelli che iniziano con un solo trattino basso, l'underscore `'_'`, pur essendo identificatori come tutti gli altri, si possono riservare per dati privati come campi di tabella o funzioni ausiliarie.

Infatti nel capitolo 11 degli iteratori si fa uso dell'identificatore `'_'` per le variabili di ciclo non utilizzate. Qualsiasi sia la convenzione adottata per i nomi, l'importante è rispettarne sempre le regole per evitare errori o un listato meno chiaro.

Sono invece comunque da evitare i nomi che iniziano con un doppio trattino basso, che potrebbero collidere con quelli dei metametodi (vedi il capitolo 12) e quelli che pur iniziando con un singolo trattino basso hanno poi lettere tutte in maiuscolo come per esempio `_VERSION`, perché potrebbero collidere con i nomi predefiniti di Lua.

4.5 Il Garbage Collector

I dati non più utili come quelli di cui non esiste più un *riferimento* a essi durante l'esecuzione, per esempio perché la 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 e sopratutto dai deleteri errori di programmazione che si possono facilmente compiere nel farlo, al prezzo di una piccola diminuzione delle prestazioni in fase di esecuzione.

Al termine del programma tutte le risorse in memoria vengono automaticamente liberate.

4.6 Esercizi

esercizio 1 Usando due sole istruzioni, scrivere il codice Lua che crea due variabili `x` e `y` di valore 12.34 e assegni alle altre due variabili `sum` e `prod` rispettivamente la somma e il prodotto delle prime due. Si stampino 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 5

In questo capitolo parleremo della *tabella*, l'unico tipo strutturato predefinito di Lua e il suo costruttore. 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, sarà un dizionario.

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

Per creare una tabella si usa il *costruttore*. Come vedremo tra poco, la sua forma più semplice è quella che corrisponde a una tabella vuota e consiste in una coppia di parentesi graffe:

```
1   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:

```
1   local t = {}      -- tabella vuota
2   t["key"] = "val"  -- nuovo elemento chiave/valore
3   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:

```
1   local t = {}
2   t["key"] = 123
3   t[123] = "key"
```

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

5.1 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. La variabile a cui viene assegnato l'oggetto contiene l'informazione della posizione in memoria della tabella. Per rendersene conto è sufficiente copiare una variabile a una tabella in una seconda variabile e verificare che non si tratta di una sua copia:

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

Con la funzione `assert()` si può dichiarare l'equivalenza logica tra due espressioni oppure imporre la verità di un'espressione prima di assegnarla a una variabile. Se l'argomento è vero essa si comporta in modo neutro restituendo l'argomento stesso, ma se è falso, essa termina l'esecuzione del programma stampando la descrizione d'errore opzionalmente fornita dal secondo argomento stringa.

Come vedremo nel capitolo 7, un'espressione è vera se vale `true` oppure non è `nil`, e falsa se vale `false` oppure è `nil`,

I tipi come i numeri invece non sono oggetti e assegnare una variabile che li contiene a un'altra comporta la copia del dato, come si può verificare con un codice simile al precedente.

Il fatto che la tabella sia 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 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 rappresentato in un'altra parte della memoria.

5.2 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 dei dati bibliografici di BibTeX, uno dei programmi storici del sistema TeX usato per la gestione delle bibliografie nei documenti LaTeX, il costruttore di Lua può creare tabelle da una sequenza di chiavi/valori inserite tra parentesi graffe:

```
1   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:

```
1   -- codice equivalente
2   local t = {}
3   t["a"] = 123
4   t["b"] = 456
5   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:

```
1   -- chiavi numeriche nel costruttore?
2   local t_error = { 20 = 123 }
3   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:

```
1   local t = { 30, 8, 500 }
2   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 valore associato a

una chiave stringa si ottiene scrivendo la chiave dopo il nome del riferimento della tabella, separato dal carattere `.`:

```
1  local t = { chiave = "123" }
2  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:

```
1  local chiave = "ok"
2  local t = { ok = "123"} -- t.ok == t[chiave]
3
4  -- attenzione!
5  local k = "ok"
6  print( t.k ) --> stampa nil: "k" non è definita in t
7  print( t[k]) --> stampa "123"
8  -- t[k] == t["ok"] == t.ok
9  -- t.k è 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.

5.3 Esercizi

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

```
1  local t = {
2      paese = "Italia",
3      lingua = "italiano",
```

```
4        codice = "IT",
5        regioni = 20,
6        province = 110,
7        comuni = 8047,
8        capitale = {"Roma", "RM", abitanti = 2753000},
9    }
```

esercizio 4 Ricreare la stessa tabella `t` dell'esercizio precedente ma *senza* usare il costrutture.

esercizio 5 Dato il triangolo di vertici di coordinate $A = (-1, 8)$, $B = (5, 3)$, e $C = (1, -3)$, creare con il costruttore una tabella di tabelle array che rappresenti il triangolo e con essa calcolare le coordinate del baricentro.

esercizio 6 Rappresentare con una tabella il triangolo dell'esercizio precedente utilizzando come chiavi i nomi assegnati dei vertici. Dalla tabella ricavare poi una seconda tabella che rappresenti il triangolo aventi i vertici nei punti medi dei lati.

esercizio 7 Istanziare una tabella dizionario con il costruttore a rappresentare i propri dati anagrafici. Stampare poi su più linee il proprio indirizzo postale.

Costrutti di base 6

6.1 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`:

```lua
-- filename: code/e1-001.lua
-- costruttore tabella (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` incrementiamo un contatore solo se 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 se 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 `<`, `<=`.

6.2 Operatore 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:

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

e in questo caso cosa verrà stampato?

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

e ancora in questo caso con il costruttore?

```
1  local t1 = {nil, nil,  nil, nil, nil, nil, nil, 8}
2  print(#t1)
3  local t2 = {nil, nil,  nil, nil, nil, nil, nil, 8, nil}
4  print(#t2)
```

L'operatore `#` restituisce la lunghezza di una tabella array come uno dei sui *bordi*. Un bordo è la posizione di un valore non `nil` seguito da un valore `nil`, oppure zero se la posizione `1` è `nil`.

Questo comportamento riflette la particolare e sofisticata implementazione della tabella di Lua. L'operatore lunghezza `#` può restituire uno dei

qualsiasi indici interi corrispondenti a un bordo della tabella, in funzione di come è stata creata o anche della presenza di chiavi non intere.

Se la tabella è una *sequenza*, cioè se non ci sono buchi di valori tra le posizioni intere partendo da 1 fino a n, allora la tabella ha un solo bordo che vale n. Se la tabella è vuota il suo bordo vale zero.

L'operatore `#` viene calcolato molto velocemente anche per tabelle array grandi, ed è usato per espressioni idiomatiche come quella usatissima per l'inserimento di un nuovo dato in coda a una sequenza:

```lua
local t = {101, 102, 103, 104, 105} -- una sequenza
t[#t + 1] = 106
print(#t, t[6])
```

```
> 6     106
```

6.3 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 `%`:

```lua
-- filename: code/e1-002.lua
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.

6.4 Intermezzo

In Lua non è obbligatorio inserire un carattere delimitatore sintattico ma è facoltativo il segno `;`. 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 segno `;`. 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.

6.5 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:

```
1  local digit = {}
2  local n = 123321
3
4  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
```

6.6 `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.

```lua
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.

6.7 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 6.3 per includerne la molteplicità.

esercizio 3 Calcolare il determinante della matrice corrispondente alla seguente tabella, che contiene tre tabelle/array con tre numeri in sequenza.

```lua
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.

```lua
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 7

Il tipo `boolean` può assumere i valori `true` oppure `false`.

A qualsiasi espressione Lua assegna un valore booleano se richiesto, per esempio in un condizionale `if` o in un ciclo `while`. In Lua un'espressione è vera se essa corrisponde al valore booleano `true` oppure a un valore che non è `nil`, altrimenti è falsa.

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.

```
1   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.

7.1 Operatore ternario

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`:

```
1   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:

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

Il massimo tra due numeri è un'espressione condizionale:

```
1   local x, y = 45.69, 564.3
2   local max
3   if x > y then
```

```
4        max = x
5    else
6        max = y
7    end
```

ma con gli operatori logici è tutto più Lua:

```
1    local x, y = 45.69, 564.3
2    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:

```
1    print(not 5)       --> 'false'
2    print(not not 5)   --> 'true'
3    print(not true)    --> 'false'
4    print(not false)   --> 'true'
5    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`:

```
1    local t = {} -- una tabella vuota
2    if not t.prezzo then -- t.prezzo è nil
3        print("assente")
4    else
5        print("presente")
6    end
7
8    t.prezzo = 12.00
9    if not t.prezzo then
10       print("assente")
11   else
12       print("presente")
13   end
```

7.2 Esercizi

esercizio 1 Prevedere il risultato delle seguenti espressioni Lua:

```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?

```lua
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 8

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 (`\\`):

```lua
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.

```
1   local long_text = [[
2   Questo è un testo multiriga
3   dove i caratteri di escape non contano
4   come \n o \" o \' o \\.
5
6   Inoltre, se il testo come in questo caso
7   comincia con un ritorno a capo allora questo
8   carattere \n sarà ignorato.
9   ]]
10  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:

```
1   local long_text = [=[
2   Questo è il codice Lua da stampare:
3
4   local tab = {10, 20, 30}
5   local idx = {3, 2, 1}
6   print(tab[idx[1]]) -- ops due parentesi quadre
7
```

```
8   local long_text = [[ -- questo non potremo farlo...
9       Testo lungo...
10  ]]
11
12  Tutto chiaro?
13  In Lua le stringhe letterali nel codice
14  possono essere proprio letterali
15  senza caratteri di escape e senza
16  preoccupazioni sulla presenza di gruppi
17  di delimitazione di chiusura...
18  ]=]
19
20  print(long_text)
```

8.1 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:

```
1   -- questo è un commento di riga
2
3   --[[
4   questo è un commento
5   multiriga
6   ]]
7
8   --[=[
9   e anche questo è un commento
10  multiriga
11  ]=]
```

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
```

8.2 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 più essere modificate. 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 al posto del valore originale.

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

```
local s = ""  -- stringa vuota
for i = 1, 100 do
    s = s .. "**"
end
print(#s) -- l'operatore # funziona anche per le stringhe!
--           contandone i byte
```

Ma cosa succede in dettaglio? Perché questo codice non è efficiente? A 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` di Java. In Lua la soluzione è una funzione della libreria `table` che, anticipando rispetto alle nostre chiaccherate è `table.concat()`:

```
1  local t = {}
2  for i = 1, 100 do
3      t[#t + 1] = "**"
4  end
5  local s = table.concat(t)
6  print(#s) --> stampa 200
```

Le singole stringhe `"**"` sono state memorizzati come elementi di una tabella in sequenza. La funzione `table.concat()` li concatena in una sola stringa senza usare concatenazioni parziali quindi in modo efficiente. Nel caso specifico avremo dovuto usare la funzione `string.rep()` considerando `table.concat()` nel caso generale.

8.3 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?

```
1  local s = "'"..'"'.."ok"..[["']]
2  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 9

Le funzioni sono il principale mezzo di astrazione e organizzazione del codice.

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

Come primo esempio, consideriamo 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. Viceversa, se gli argomenti sono inferiori a quelli previsti allora a quelli mancanti verrà assegnato il valore `nil`.

Le stesse regole valgono anche per i dati di uscita quando la funzione è usata come espressione in un'istruzione di assegnazione. Dopo l'istruzione `return` si può scrivere una lista delle espressioni separate da virgole che saranno assegnate alle corrispondenti variabili.

Per esempio, potremo modificare la funzione precedente per restituire anche 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
```

9.1 Funzioni: valori di prima classe, I

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

A ben vedere in Lua tutte le funzioni sono memorizzate in variabili e di per se non hanno un nome. Il modo di definizione è quindi la *sintassi anonima*:

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

Questa importante proprietà si riassume dicendo che in Lua le funzioni sono valori di *prima classe*: sono assegnate a variabili e non hanno un nome esattamente come non lo hanno gli altri tipi come numeri e stringhe.

Per comodità il linguaggio ammette anche la sintassi classica di definizione:

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

L'interprete Lua la tradurrà automaticamente nel codice equivalente in sintassi anonima, una caratteristica nascosta detta *zucchero sintattico*:

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

9.2 Funzioni: valori di prima classe, II

Un esempio di funzione con un argomento funzione è il seguente, dove si vuole eseguire per un certo numero di volte consecutivamente la funzione argomento:

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

local function print_five()
    print(5)
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).

Essendo le funzioni valori di prima classe è possibile riassegnare variabili per cambiarne il significato, come per esempio con la funzione predefinita `print()`:

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

print(12)
```

```
> Argomento funzione -> 12
```

9.3 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 foo()
    -- body
end
local t = {}
t.foo = foo

-- secondo caso
local t = {}
t.foo = function ()
```

```
11        -- body
12    end
13
14    -- terzo caso con più funzioni
15    local t = {
16        foo = function ()
17            -- body
18        end,
19        hoo = function ()
20            -- body
21        end,
22    }
```

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

9.4 Numero di argomenti variabile

Una funzione può ricevere un numero variabile di argomenti rappresentati da `...` tre caratteri punto consecutivi[1]. 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:

```
1    -- per un massimo di 5 argomenti
2    local function add(...)
3        local n1, n2, n3, n4, n5 = ...
4        return n1 + n2 + (n3 or 0) + (n4 or 0) + (n5 or 0)
5    end
6
7    -- con tutti gli argomenti
8    local function add_many(...)
9        local t = {...} -- collecting args in a table
10       local sum = 0
11       for i = 1, #t do
12           sum = sum + t[i]
13       end
```

[1]Questa caratteristica è chiamata *variadic function*.

```
14        return sum
15    end
16
17    print(add(40, 20))
18    print(add_many(45, 48, 54, 56, -58, 20))
19    print(add(14, 15, 6), add_all(-89, 45))
```

Nell'ultima riga di codice si può notare che anche la funzione predefinita `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:

```
1    local function add_and_multiply(molt, ...)
2        local t = {...}
3        local sum = 0
4        for i = 1, #t do
5            sum = sum + t[i]
6        end
7
8        return molt * sum
9    end
10
11   print(add_and_multiply(10, 45.23, 48, 9.36, -8, -56.3))
```

La 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 `#` determina la lunghezza della sequenza in base ai bordi cioè alle posizioni dei valori non nulli seguite da valori `nil` (maggiori dettagli alla sezione §6.2).

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 esempio:

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

9.5 Omettere le parentesi

In Lua esiste la sintassi di chiamata a funzione semplificata che consiste nella possibilità di ommettere le parentesi tonde `()`. È ammessa solo se alla funzione si passa un unico argomento di tipo stringa o di tipo tabella:

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

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

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

9.6 Closure

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

Questa proprietà di Lua amplia il concetto di funzione rendendo possibile l'accesso dall'interno di essa a 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 ()
```

```
4            i = i + 1 -- accesso alla closure
5            return i
6        end
7    end
8
9    local c1 = new_counter()
10   print(c1()) --> 1
11   print(c1()) --> 2
12   print(c1()) --> 3
13   print(c1()) --> 4
14   print(c1()) --> 5
15
16   local c2 = new_counter()
17   print(c2()) --> 1
18   print(c2()) --> 2
19   print(c2()) --> 3
20
21   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 la funzione `table.sort()` della libreria di Lua che introdurremo nel prossimo capitolo, per applicare l'algoritmo di ordinamento alla tabella primo argomento in base al criterio stabilito con la funzione passata come secondo argomento in sintassi anonima.

```
1    local function sort_by_value(tab)
2        local val = {
3            [1994] = 12.5,
4            [1996] = 10.2,
5            [1998] = 10.9,
6            [2000] =  8.9,
7            [2002] = 12.9,
```

```
8       }
9       table.sort(tab,
10          function (a, b) -- accesso alla closure
11              return val[a] > val[b]
12          end
13      )
14  end
15
16  local years = {1994, 1996, 1998, 2000, 2002}
17  sort_by_value(years)
18  for i = 1, #years do
19      print(years[i])
20  end
```

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

9.7 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 `"cen"`.

Esercizio 5 Scrivere una funzione `ordinates()` 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 $n \geq 2$ in cui suddividere equamente l'intervallo, e restituisca i valori in sequenza che la funzione f assume nei punti d'ascissa così definiti, in una tabella/array con indice iniziale 1.

Verificare poi che le due tabelle calcolate da `ordinates()` con le funzioni di libreria `math.sin` e `math.cos`, nello stesso intervallo 0, $\pi/2$ con $n = 101$ contengano lo stesso valore nella posizione di indice 51.

La libreria standard di Lua 10

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 per la 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.

10.1 Libreria matematica

Nella libreria memorizzata nella tabella `math` ci sono le funzioni trigonometriche `sin()`, `cos()`, `tan()`, `asin()` eccetera — che 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
```

Accanto a funzioni matematiche vere e proprie troviamo la funzione `math.type()` che esamina l'argomento e restituisce `"integer"` o `"float"` a seconda se esso è un intero o un numero in virgola mobile, oppure `nil` se non si tratta di un numero. Questa funzione è utile per eseguire il controllo di validità dei parametri in ingresso.

La funzione `math.tointeger()` è, come la precedente, utile per verificare se l'argomento è un intero. Essa infatti tenta la conversione e nel caso non sia possibile il risultato sarà `nil`.

L'argomento può essere anche un numero o una stringa, e in questo è simile alla funzione `tonumber()` ma solo per i valori interi:

```
print(tonumber(123), math.tointeger(123))
print(tonumber(-89), math.tointeger(-89))
print(tonumber(40.5 * 2), math.tointeger(40.5 * 2))
print(tonumber(40.5 * 3), math.tointeger(40.5 * 3))
print(tonumber("89"), math.tointeger("89"))
print(tonumber("89.6"), math.tointeger("89.6"))
```

```
> 123   123
> -89   -89
> 81.0  81
> 121.5 nil
> 89    89
> 89.6  nil
```

10.2 Libreria tabelle

Per le tabelle la libreria standard fornisce funzionalità attraverso la tabella `table`. Tra le funzioni più importanti troviamo `table.concat()` che concatena tutti i valori nelle posizioni sequenza in una stringa. Per esempio:

```
1   -- table.concat(list [, sep [, i [, j]]])
2   assert(table.concat{1,2,3,4,5,6} == "123456")
```

Ci sono anche utili argomenti opzionali: un secondo argomento stringa che sarà usato come separatore dei vari elementi, un terzo argomento intero che è l'indice dal quale gli elementi saranno raccolti e un quarto argomento come indice finale.

Questa funzione è molto importante nelle applicazioni perché consente di concatenare stringhe in un modo efficiente quando il loro numero è maggiore di una decina[1].

Altre funzioni molto utili sono `table.insert()` con la sua controparte `table.remove()`, e `table.sort()` per riordinare gli elementi della sequenza di una tabella secondo criteri definibili di cui un esempio si trova nella sezione 9.6.

10.3 Libreria stringhe

La libreria per le stringhe è memorizzata nella tabella `string` ed è una delle più utili per questo l'approfondiremo più in dettaglio rispetto alle altre. 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.

10.3.1 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

[1] L'ordine di grandezza del numero delle stringhe da concatenare per cui conviene l'uso di `table.concat()` dovrebbe essere una decina ma non ho fatto benchmark in proposito.

digit mentre `"%f"` indica il segnaposto per un numero decimale con `f` che sta per float.

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

```lua
-- "%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 carattere `%` e lo specificatore di tipo, per esempio per indicare il numero delle cifre decimali di arrotondamento.

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*.

10.3.2 Pattern I

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 un pattern è `string.match()`. Essa restituisce la prima corrispondenza trovata in una stringa data come primo argomento che corrispondente al pattern dato come secondo argomento.

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

```
1  -- semplice pattern in azione
2  local s = [[
3  le prime tre cifre decimali di \pi = 3,141592654 sono]]
4  local pattern = "%d%d%d"
5  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 ripetizione;

Esempio:

```lua
-- corrispondenza 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
```

10.3.3 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)"`:

```lua
-- 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:

```lua
-- 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
```

10.3.4 Pattern II

Mentre il carattere percento unito a una lettera forma una delle classi carattere elencate alla sezione 10.3.2, la stessa lettera scritta in maiuscolo rappresenta la classe carattere negazione.

La classe `"%D"` rappresenta un carattere che non sia una cifra decimale, così come `"%S"` corrisponde al carattere che non ha corrispondenza con `"%s"`. Per esempio, per catturare una prima sequenza di eventuali caratteri spazio in una stringa si può usare il pattern di ricerca `"(%s*)%S?"`.

Due ulteriori caratteri speciali `^` per l'inizio riga e `$` per la fine riga, svolgono il ruolo di ancoraggio del pattern. Possono essere usati entrambi per esempio per eliminare da una stringa i caratteri spazio iniziali o finali:

```
1  local pattern = "^%s*(.-)%s*$"
2  print("|"..string.match("ok", pattern).."|" )      --> "|ok|"
3  print("|"..string.match(" ok", pattern).."|" )     --> "|ok|"
4  print("|"..string.match(" ok ", pattern).."|" )    --> "|ok|"
5  print("|"..string.match(" ok ok ", pattern).."|" ) --> "|ok ok|"
```

Passiamo ora ai *char-set*. Si tratta di classi definite dall'utente racchiudendo tra parentesi quadre caratteri o classi di carattere già definiti. Per esempio, per estrarre da una stringa una sequenza di 0 e 1 possiamo definire una nuova classe carattere che corrisponde o alla prima o alla seconda cifra con `[01]`.

Per esempio, per trovare corrispondenze di *identificatori*, cioè sequenze di caratteri alfanumerici che comprendono il trattino basso e che non iniziano con una cifra, potremo usare il pattern `"^[_%a][_%w]*"`:

```
1  local pattern = "^[_%a][_%w]*"
2  print(string.match("1var", pattern))    --> nil
3  print(string.match("_", pattern))       --> "_"
4  print(string.match("var", pattern))     --> "var"
5  print(string.match("_var", pattern))    --> "_var"
6  print(string.match("1var", pattern))    --> nil
7  print(string.match("var1_=5", pattern)) --> "var1_"
8  print(string.match("0", pattern))       --> nil
```

Ulteriore esempio è la ricerca di corrispondenze per numeri interi con segno:

```
1  local pattern = "[+-]?%d+"
2  print(string.match("56", pattern))   --> "56"
```

```
3   print(string.match("+56", pattern))  --> "+56"
4   print(string.match("-56", pattern))  --> "-56"
5   print(string.match("-+56", pattern)) --> "+56"
6   print(string.match("0000", pattern)) --> "0000"
```

In un pattern, tra parentesi quadre è ammesso l'uso del carattere trattino per definire intervalli. Il pattern `"[012345]"` che corrisponde a una classe carattere per le prime sei cifre decimali, può essere definito anche con `"[0-5]"`.

10.3.5 La funzione `string.gsub()`

Abbiamo appena cominciato a scoprire le funzionalità dedicate al testo disponibili nella libreria standard di Lua. Diamo solo un altro sguardo presentando la funzione `string.gsub()`. Il suo nome sta per *global substitution*, ovvero la sostituzione di tutte le parti di un testo che corrispondono a uno schema.

Per individuare le parti da sostituire è naturale pensare di utilizzare un pattern e che sia possibile utilizzare le capture nel testo di sostituzione, per esempio:

```
1   local s = "The house is black."
2   print(string.gsub(s, "black", "red"))
3   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, il terzo è il testo di sostituzione per ciscuna corrispondenza, ma può anche essere una tabella dove le chiavi corrispondenti al pattern saranno sostituite con i rispettivi valori, oppure una funzione che dal singolo frammento di testo estratto elabora la stringa di sostituzione.

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

```
1   local s = "Cose da fare oggi 5, cosa da fare domani 2"
2   print(string.gsub(s, "%d+", function(n)
3       return tonumber(n)*12
4   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.

Infine, le funzioni della libreria `string` possono essere chiamate su variabili stringa usando la *colon notation* che introdurremo alla sezione 12.3, grazie ai metametodi:

```
1  string.match(s, pattern) -- equivalente a:
2  s:match(pattern)
```

10.3.6 La funzione `string.gmatch()`

Con la funzione `string.gmatch()` è possibile iterare tutte le corrispondenze di un pattern in una stringa in un ciclo generic for. Gli argomenti da fornire sono la stringa e il pattern che può contenere delle capture. Vediamo un primo esempio:

```
1  local s = "trova le vocali"
2  for vowel in string.gmatch(s, "[aeiou]") do
3      print(vowel)
4  end
```

```
> o
> a
> e
> o
> a
> i
```

Se invece il pattern contiene delle capture, ad ogni iterazione altrettante variabili di ciclo assumeranno i valori stringa delle catture:

```
1  local s = "da A=(1,9) a B=(-9,5) passando per C=(0,3)"
2  local pattern = "([A-Z])=%(([+-]?%d+),([+-]?%d+)%)"
3  for p, x, y in string.gmatch(s, pattern) do
4      print(p, tonumber(x), tonumber(y))
5  end
```

```
> A     1       9
> B     -9      5
> C     0       3
```

10.3.7 Altre funzioni utili

Nominiamo per prima la funzione `string.find()`. Essa restituisce la posizione della corrispondenza nella stringa di ricerca, nella forma della coppia d'indici di inizio e di fine della sequenza. La ricerca può cominciare da una posizione qualsiasi in virtù del terzo argomento opzionale, e se il pattern contiene una capture la corrispondenza verrà restituita come ulteriore terzo risultato:

```
1  -- string.find(s, pattern [, init [, plain]])
2  local s = "Un intero 123 e un secondo 456!"
3  local i, j = string.find(s, "123") -- i = 11, j = 13
4  print(string.find(s, "(%d+)", j+1))
```

```
> 28    30      456
```

Conviene citare altre due funzioni utili per esempio per poter usare il simbolo di percentuale nel codice Lua interno a sorgenti TEX. Sono `string.byte()` e `string.char()`: la prima da una lista di interi positivi calcola la stringa codificata con essi e la seconda calcola i codici dei caratteri della stringa argomento. Per esempio:

```
1  -- string.byte(s [, i [, j]])
2  -- string.char(···)
3  local s = "LuaTeX"
4  local t = {string.byte(s, 1, -1)}
5  assert(table.concat(t, ", ") == "76, 117, 97, 84, 101, 88")
6  assert(string.char(76, 117, 97, 84, 101, 88) == s)
```

10.4 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?

```
1  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 estrarre 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 ad altrettante capture nel pattern di ricerca.

Iteratori 11

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 ed elegante.

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

```
1  for <lista variabili> in iterator_function() do
2  -- codice
3  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 rispettivamente 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.

11.1 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 comincia dalla posizione 1 e termina fino a quando il valore diventa `nil`:

```
1  -- una tabella array
2  local t = {45, 56, 89, 12, 0, 2, -98}
3
4  -- iterazione tabella come array
5  for i, v in ipairs(t) do
6      print(i, v)
```

```
7   end
```

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

```
1   -- una tabella array
2   local t = {45, 56, 89, 12, 0, 2, -98}
3   do
4       local i, v = 1, t[1]
5       while v do
6           print(i, v)
7           i = i + 1
8           v = t[i]
9       end
10  end
```

Se non interessa il valore dell'indice è buona norma dare al nome di variabile corridspondente un segno di underscore _ che in Lua è un identificatore valido. Per esempio:

```
1   -- una tabella array
2   local t = {45, 56, 89, 12, 0, 2, -98}
3   local sum = 0
4   for _, elem in ipairs(t) do
5       sum = sum + elem
6   end
7   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()` che tratteremo nella prossima sezione.

11.2 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()` al prezzo di perdere l'ordinamento:

```
1   -- produzione tabella con salto
2   local t = {45, 56, 89}
```

```
3   local i = 100 + #t -- 100 holes
4   for j, v in ipairs{12, 0, 2, -98} do
5       t[i+j] = v
6   end
7   print("ipairs() table iteration test")
8   for index, elem in ipairs(t) do
9       print(string.format("t[%3d] = %d", index, elem))
10  end
11  print()
12  print("pairs() table iteration test")
13  for key, val in pairs(t) do
14      print(string.format("t[%3d] = %d", key, val))
15  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] = 12
> t[105] = 0
> t[106] = 2
> t[107] = -98
```

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

11.3 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 9.6. Proviamo a scrivere il codice per iterare una tabella:

```
1   -- costructor
```

```
2   local t = {45, 87, 98, 10, 16}
3
4   function iter(t)
5       local i = 0
6       return function ()
7           i = i + 1
8           return t[i]
9       end
10  end
11
12  -- utilizzo
13  local iter_fn = iter(t)
14  while true do
15      local val = iter_fn()
16      if val == nil then
17          break
18      end
19      print(val)
20  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_fn()` 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:

```
1   -- costructor
2   local t = {45, 87, 98, 10, 16}
3
4   function iter(t)
5       local i = 0
6       return function ()
7           i = i + 1
8           return t[i]
9       end
10  end
```

```lua

-- 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:

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

11.4 L'esempio dei numeri pari

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

Potremo fare così: dato il numero iniziale per prima cosa potremo calcolare il primo numero pari dell'intervallo usando l'operatore modulo `%` e poi creare la funzione di iterazione in sintassi anonima che prima incrementa di 2 la variabile del ciclo — ed ecco perché dovremo inizialmente sottrarle la stessa quantità — e poi, se questa è inferiore all'estremo superiore dell'intervallo, restituisce indice e numero pari corrente. Ecco il codice completo:

```lua
-- iteratore dei numeri pari compresi
-- nell'intervallo [first, last]
function evenNum(first, last)
    -- valori iniziali delle variabili di ciclo
    local val = first + first % 2 - 2
    local i = 0
    return function ()
```

```
8           i = i + 1
9           val = val + 2
10          if val<=last then
11              return val, i -- due variabili di ciclo
12          end
13      end
14  end
15  -- iterazione con due variabili di ciclo
16  for val, i in evenNum(13,20) do
17      print(string.format("iterazione %d -> %d", i, val))
18  end
```

```
> iterazione 1 -> 14
> iterazione 2 -> 16
> iterazione 3 -> 18
> iterazione 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 nella sezione 4.2.

Naturalmente, l'implementazione data di `evenNum()` è solo una fra quelle possibili, 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.

11.5 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, implementando uno *stateless iterator*.

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.

```
1  for <var list> in <iter_fn>, <state>, <ctl_var> do
2      ...
3  end
```

La funzione d'iterazione verrà chiamata a ogni ciclo con due argomenti: lo stato invariante e la variabile di controllo e ci si aspetta che restituisca uno o più dati di ciclo. Quando questi valori saranno `nil` il ciclo termina.

Nel seguente codice la funzione `evenNum()` provvede a restituire i tre parametri necessari: la funzione `next_even()` come iteratore, lo stato invariante, ovvero il numero a cui la sequenza dovrà fermarsi e la variabile di controllo che è proprio il valore nella sequenza dei numeri pari.

```
1   -- even number stateless iterator
2   local function next_even(last, i)
3      i = i + 2
4      if i<=last then
5         return i
6      end
7   end
8
9   local function evenNum(a, b)
10     return next_even, b, a + a % 2 - 2
11  end
12
13  -- run the 'generic for'
14  for n in evenNum(10, 20) do
15      print(n)
16  end
```

Con gli iteratori abbiamo terminato l'esplorazione di base del linguaggio Lua. Gli otto capitoli dal 4 al 11 sono sufficienti per scrivere programmi utili perché trattano tutti gli argomenti essenziali, il prossimo invece, tratterà il paradigma della programmazione a oggetti in Lua.

11.6 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 interi in sequenza) 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?

esercizio 6 Definire un iteratore stateless attraverso la funzione `Range()` che, funzionando con il generic for, simuli invece il ciclo for numerico a variabile intera, iterando su un intervallo con un dato passo.

Gli argomenti della funzione possono variare da uno a tre secondo questo schema:

```
Range(7)       --> 1, 2, 3, 4, 5, 6, 7
Range(2, 7)    --> 2, 3, 4, 5, 6, 7
Range(2, 7, 2) --> 2, 4, 6
```

Il passo può essere negativo? Quali vantaggi potrebbe avere un iteratore di questo tipo rispetto al ciclo for numerico? Suggerimento: considerare che le funzioni in Lua sono valori di prima classe.

Programmazione a oggetti in Lua 12

In sintesi, il paradigma della *programmazione a oggetti*[1], 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 specializzarne il comportamento.

Per creare un oggetto di una classe si utilizza un metodo speciale chiamato *costruttore*, che valida gli eventuali dati in ingresso e imposta 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 in classi e le loro relazioni di ereditarietà.

Negli ultimi anni, la programmazione a oggetti è stata ripensata 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.

12.1 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

[1] Object Oriented Programming OOP.

di classe, non offre alcun meccanismo per dichiarare come privati campi e metodi, e lascia al programmatore più di un modo per implementare il paradigma.

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

12.2 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:

```
1   -- prima tentativo di implementazione
2   -- di una classe rettangolo
3   Rettangolo = {} -- creazione tabella (oggetto)
4
5   -- creazione dei due campi
6   Rettangolo.b = 12
7   Rettangolo.h = 7
8
9   -- un primo metodo assegnato direttamente
10  -- ad un campo della tabella
11  function Rettangolo.area ()
12      -- accesso alla variabile 'Rettangolo'
13      return Rettangolo.b * Rettangolo.h
14  end
15
16  -- primo test
17  print(Rettangolo.area()) --> stampa 84, OK
18  print(Rettangolo.h)      --> 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:

```
1   -- ancora la prima implementazione
2   Rettangolo = {b = 12, h = 7}
3
4   -- un metodo dell'oggetto
5   function Rettangolo.area ()
```

```
6        -- accesso alla variabile 'Rettangolo' attenzione!
7        local l = Rettangolo.b
8        local a = Rettangolo.h
9        return l * a
10   end
11
12   -- secondo test
13   r = Rettangolo   -- creiamo un secondo riferimento
14   Rettangolo = nil -- distruggiamo il riferimento originale
15
16   print(r.b)       --> stampa 12, OK
17   print(r.area())  --> errore!
18   -- attempt to index a nil value (global 'Rettangolo')
```

Il problema sta nel fatto che nel metodo `area()` compare il particolare riferimento alla tabella `Rettangolo`. La soluzione non può che essere l'introduzione del riferimento dell'oggetto come parametro esplicito nel metodo stesso, ed è la stessa utilizzata anche dagli altri linguaggi di programmazione che supportano gli oggetti. Vedremo tra poco come, allo stesso modo dei linguaggi OOP, anche in Lua si possa nascondere il riferimento con la colon notation.

Secondo quest'idea dovremo riscrivere il metodo `area()` in questo modo (in Lua il riferimento implicito all'oggetto deve chiamarsi `self` pertanto abituiamoci fin dall'inizio a questa convenzione):

```
1   -- seconda implementazione
2   Rettangolo = {b=12, h=7}
3
4   -- il metodo diviene indipendente dal particolare
5   -- riferimento all'oggetto:
6   function Rettangolo.area(self)
7      return self.b * self.h
8   end
9
10  -- e ora il test
11  myrect = Rettangolo
12  Rettangolo = nil -- distruggiamo il riferimento
13
14  print(myrect.b)            --> stampa 12, OK
15  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. Questo punto è importante: le chiavi nella tabella dell'oggetto possono contenere sia funzioni/metodi sia valori/campi perciò sovrascrivere una chiave nell'intenzione di introdurre un nuovo campo/metodo porta a errori.

12.3 Colon notation

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:

```
1  -- forma classica
2  myrect.area(myrect)
3
4  -- forma implicita, 'self' è lo stesso riferimento di 'myrect'
5  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 aggiungerà implicitamente un primo parametro chiamandolo `'self'` con il riferimento alla tabella stessa, insomma puro zucchero sintattico.

12.4 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 in Lua si utilizza la tabella, l'unica struttura dati predefinita nel linguaggio, assieme a particolari funzionalità dette *metatabelle* e *metametodi*.

Il salto definitivo nella programmazione a oggetti consiste nel poter costruire una *classe* senza ogni volta assemblare 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 oggetti che condividono una stessa aritmetica.

Metatabelle e metametodi quindi, possono rendere il codice intuitivo e compatto, sono quindi una funzionalità indipendente dalla programmazione a oggetti.

I nomi delle funzioni di una metatabella vengono detti *metametodi* e iniziano tutti con un doppio trattino basso. Per esempio nel caso della somma sarà richiesta la funzione `__add()` nella metatabella associata al primo addendo o se non esiste a quella del secondo addendo.

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

12.5 Il metametodo `__tostring()`

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:

```
1   -- un numero complesso
2   local complex = {real = 4, imag = -9}
3   print(complex) --> stampa: 'table: 0x9eb65a8'
4
5   -- un qualcosa di più utile: metametodo __tostring()
6   -- in sintassi anonima
7   local mt = {}
8   mt.__tostring = function (c)
9      local fmt = "(%0.2f, %0.2f)"
10     return string.format(fmt, c.real or 0, c.imag or 0)
11  end
12
13  -- assegnazione della metatabella mt a complex
14  setmetatable(complex, mt)
15
16  -- riprovo la stampa
```

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

12.6 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 la chiave 'a' ma non '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 (t, key)
        return 'Attenzione: campo "'..key..'" inesistente!'
    end
}
-- assegniamo 'mt' come metatabella di 't'
setmetatable(t, mt)

-- adesso riproviamo
print(t.a)  --> stampa 'Campo A'
print(t.b)  --> stampa 'Attenzione: campo "b" 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 = {b=10, h=10}

-- un metodo:
function Rettangolo:area()
  return self.b * self.h
end

-- creazione metametodo
Rettangolo.__index = Rettangolo

```

```
12  -- un nuovo oggetto Rettangolo
13  local r = {}
14  setmetatable(r, Rettangolo)
15
16  print(r.b)      --> stampa 10, Ok
17  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 `b`, 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 da essa per essere accessibili da `r`. Il campo `b` 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: `b` e `h`. 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 in `r`.

Questa costruzione funziona ma può essere migliorata con l'introduzione del costruttore come vedremo meglio in seguito. L'oggetto `Rettangolo` apparirà sempre più concettualmente simile a una classe.

12.7 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 e una volta effettuato il controllo di validità degli argomenti.

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

```
1  -- nuova classe Rettangolo (campi con valore di default)
2  local Rettangolo = {b = 1, h = 1}
3
```

```
4   -- metametodo
5   Rettangolo.__index = Rettangolo
6
7   -- metodo di classe
8   function Rettangolo:area()
9       return self.b * self.h
10  end
11
12  -- costruttore di classe
13  function Rettangolo:new(o)
14      -- creazione nuova tabella
15      -- se non ne viene fornita una
16      o = o or {}
17      -- controllo campi
18      if o.b and o.b < 0 then
19          error("campo larghezza negativo")
20      end
21      if o.h and o.h < 0 then
22          error("campo altezza negativo")
23      end
24      -- assegnazione metatabella
25      setmetatable(o, self)
26      -- restituzione riferimento oggetto
27      return o
28  end
29
30  -- codice utente ------------------
31  local r1 = Rettangolo:new{b=12, h=2}
32  print(r1.b)       --> stampa  12, Ok
33  print(r1:area())  --> stampa  24, Ok
34
35  local r2 = Rettangolo:new{h=5}
36  print(r2.b)       --> stampa 1, Ok
37  print(r2:area())  --> stampa 5, Ok
```

Il costruttore `new()` accetta una tabella come argomento, altrimenti ne crea una vuota, controlla gli eventuali parametri geometrici, assegna la metatabella e restituisce l'oggetto. Alla funzione arriva il riferimento implicito a `Rettangolo` grazie alla colon notation, per cui `self` è un riferimento alla stessa tabella del di quello della variabile `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.

Essendo il costruttore una normale funzione Lua, i suoi argomenti possono essere più di uno e di diverso tipo, mentre la classe può averne anche più di uno con nomi differenti.

12.8 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 una funzione che modifichi il valore del raggio aggiungendovi una misura:

```lua
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 = {radius = r}
    setmetatable(o, self)
    return o
end
```

```
23
24  -- codice utente ----------------------
25  local o = Cerchio:new()
26  print(o)  --> stampa 'Sono un cerchio di raggio 0.00'
27
28  o:addToRadius(12.342)
29
30  print(o)  --> stampa 'Sono un cerchio di raggio 12.34'
31  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 è chiamato 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.

12.9 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 metodi o 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:

```lua
-- 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
end

-- method overriding
function Schermista:print()
    local fmt = "'%s' weapon->'%s' rank->%d"
```

```
39        print(
40            string.format(
41                fmt,
42                self.name,
43                self.weapon,
44                self.rank
45            )
46        )
47    end
48
49    -- test
50    local s = Sportivo:new{name="Gianni"}
51    s:print() --> stampa 'Gianni' OK
52
53    -- il metodo costruttore new() è quella della classe base!
54    local f = Schermista:new{
55        name="Tiger",
56        weapon="Foil"
57    }
58    f:add_to_rank(45)
59
60    f:print() --> stampa 'Tiger' weapon->'Foil' rank->45
61
62    -- chiamata a un metodo della classe base
63    f:set_name("Amedeo")
64    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.

12.10 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 III

Applicazioni Lua in LuaTEX

Un registro delle compilazioni 13

Siamo giunti al primo capitolo di taglio applicativo. Ci occuperemo di creare in Lua un sistema di registrazione delle compilazioni che si concretizza in un file posizionato nella directory principale del progetto di documento.

Non è possibile reperire dati generali sulla compilazione eseguendo codice durante la compilazione stessa. Dovremo farlo con un tool esterno. Solo in questo modo è possibile ottenere il tempo di compilazione totale o accertarsi del nome del file sorgente. Tuttavia la registrazione in fase di compilazione è utile per imparare con Lua a lavorare con i file e sperimentare un processo di sviluppo iterativo molto più rapido che con TEX.

Chiameremo questo registro `history.log` verso cui invieremo una linea di testo informativa per ciscuna compilazione che il compositore completi correttamente.

13.1 Scrivere sul registro

Iniziamo a implementare la gestione automatica del registro dalla funzionalità chiave: la scrittura su file. Ci rivolgeremo alla libreria standard di Lua, modulo `io`[1]:

```
-- write a line of text in a file
local function append(filename, line)
    local f = io.open(filename, "a+")
    f:write(line.."\n")
    f:close()
end
```

[1] La completa documentazione della libreria `io` si trova alla pagina https://www.lua.org/manual/5.3/manual.html#6.8.

La funzione `io.open()` restituisce un riferimento a un oggetto che rappresenta un canale di input/output del sistema operativo verso un file i cui metodi vanno chiamati in colon notation (vedi alla sezione 12.2). `open()` accetta il nome e la modalità di apertura del file. Lo specificatore `"a+"` indica di aprire il file in *append* senza distruggerne il contenuto preesistente. In caso il file non esista ne verrà creato uno nuovo perciò in ogni caso otterremo il file aperto in scrittura.

La funzione `append()` può essere provata in un sorgente plain LuaTEX compilabile come in questo:

```
% !TeX program = LuaTeX
% filename: app-reg/01.tex
\directlua{
register = {}
function register.append(filename, line)
    local f = io.open(filename, "a+")
    local nl = string.char(10)
    f:write(line..nl)
    f:close()
end
}

Testo

\directlua{register.append("history.log", "new run")}
\bye
```

13.2 Dati di compilazione

Costruite le fondamenta stabiliamo quali dati inserire nel registro: l'utente, mano a mano che il lavoro sul documento procede, eseguirà senza dubbio delle compilazioni intermedie perciò registreremo le seguenti informazioni di base:

1. il nome del file sorgente,
2. la dimensione del file sorgente,
3. data e ora della compilazione,
4. il tempo di esecuzione della composizione,
5. il nome del motore di composizione.

Per formare la linea di registro, al posto della concatenazione di stringhe useremo quella di una tabella di stringhe, più efficiente. Riempiremo la tabella un dato alla volta in un blocco `\directlua` posizionato immediatamente prima del termine del sorgente. Come prima istruzione invece, inseriremo il blocco di codice Lua che definirà alcuni parametri come il nome del file del registro e il separatore di campo testuale, e la funzione `append()`:

```
% !TeX program = LuaTeX
% filename: app-reg/02.tex
\directlua{
register = {
    start = os.clock(),
    sep = ", ",
    filename = "history.log",
}
function register:append(tline)
    local f = io.open(self.filename, "a+")
    local sep = self.sep
    f:write(table.concat(tline, sep))
    f:write(string.char(10))
    f:close()
end
}

Testo

\directlua{
local jobname = tex.jobname..".tex"
register:append{
    jobname,                      % 1
    lfs.attributes(jobname).size, % 2
    os.date(),                    % 3
    os.clock() - register.start,  % 4
    status.luatex_engine,         % 5
}}
\bye
```

13.2.1 La variabile jobname

Il nome del file sorgente non è detto che sia contenuto nella variabile `jobname` che possiamo trovare anche nella macro omonima oppure nel campo omonimo della tabella `tex`.

Il campo o la macro conterrà il nome del sorgente soltanto se l'utente non ha valorizzato l'opzione `--jobname` nel comando di compilazione con un altro nome o se non è stato modificato il campo `tex.jobname`.

Infatti, se provassimo a compilare il listato `02.tex` con il seguente comando

```
$ luatex --jobname=abc 02
```

otterremo un errore che blocca la compilazione. Nel secondo `\directlua` la variabile locale `jobname` conterrebbe infatti la stringa `abc.tex` che è un file che non esiste. Di conseguenza la funzione `lfs.attributes()` restituisce `nil` che ovviamente non può essere indicizzato con la chiave `size`.

La tabella `lfs` contiene la libreria Lua File System disponibile per i sistemi operativi più diffusi, con alcune funzionalità sui file che non troviamo di base in Lua perché l'interprete ha regole stringenti di portabilità. La troviamo in LuaTEX già compilata staticamente. La documentazione si trova all'indirizzo web https://keplerproject.github.io/luafilesystem/manual.html.

Se la funzione `lfs.attributes()` non restituisce la tabella con i dati del file significa dunque che nel comando di compilazione è stata attivata l'opzione `--jobname` per assegnare un diverso nome al file PDF di uscita. Modifichiamo dunque il codice eseguito in chiusura in questo senso:

```lua
local tline = {}
local jobname = tex.jobname
local attr = lfs.attributes(jobname..".tex")
if attr then
    tline[1] = jobname
    tline[2] = attr.size
else
    tline[1] = "unknow"
    tline[2] = "unknow"
end
tline[3] = os.date()
tline[4] = os.clock() - register.start
```

```
tline[5] = status.luatex_engine
register:append(tline)
```

Anche l'estensione `.tex` è un problema perché se fosse diversa ancora una volta il codice non funzionerebbe, per esempio se il file sorgente avesse estensione `.TEX`.

13.2.2 Gli altri dati

Come sappiamo dal capitolo 10 la tabella `os` appartiene alla libreria standard di Lua. La documentazione delle funzioni `os.date()` e `os.clock()` si trova quindi nel reference ufficiale del linguaggio alla pagina https://www.lua.org/manual/5.3/manual.html#6.9.

Infine, maggiori informazioni sulla tabella `status` da cui leggiamo il nome del motore di composizione, possono essere reperite nel manuale di LuaTEX.

Quanto alla misura del tempo di compilazione che registriamo, più precisamente si tratta del tempo che trascorre tra i due momenti in cui eseguiamo la funzione `os.clock()` perciò non comprende il tempo iniziale di avvio e caricamento del formato, né i tempi finali di chiusura come la composizione del materiale non ancora emesso sulla pagina.

13.3 Creare il modulo e il pacchetto

Per consentire l'utilizzo del registro da qualsiasi sorgente occorre spostare il codice Lua in un file esterno di *modulo* e scrivere un secondo file di *pacchetto* per fornire un'interfaccia utente conforme al formato, per esempio per LaTeX.

Questa separazione non è solo utile al processo di sviluppo, facilitando l'accesso alle funzionalità da parte degli utenti indipendentemente dal formato.

Allo stato attuale dello sviluppo del codice, il modulo Lua è il seguente:

```lua
-- filename: app-reg/mod-history.lua
local register = {
    start = os.clock(),
    sep = ", ",
    filename = "history.log",
```

```
}
-- output function
function register:append(tline)
    local f = io.open(self.filename, "a+")
    local sep = self.sep
    f:write(table.concat(tline, sep))
    f:write(string.char(10))
    f:close()
end
-- main user fucntion
function register:log()
    local jobname = tex.jobname
    local jn_attr = lfs.attributes(jobname..".tex")
    local tline = {}
    if jn_attr then
        tline[1] = jobname..".tex"
        tline[2] = jn_attr.size
    else
        tline[1] = "unknow"
        tline[2] = "unknow"
    end
    tline[3] = os.date()
    tline[4] = os.clock() - self.start
    tline[5] = status.luatex_engine
    self:append(tline)
end
return register
```

Tutte le funzionalità sono incluse in un unica tabella Lua chiamata `register`. Dall'esterno, caricando il modulo con la funzione `require()` potremo assegnarne il riferimento a una variabile locale o globale a seconda delle necessità.

L'uso della colon notation per chiamare le funzioni non fa parte di un'implementazione a oggetti ma è solo un semplice modo per poter disporre di un riferimento alla tabella contenitore con la chiave `self`. Ciò rende pulito il codice ed elimina la necessità di creare una closure con un piccolo incremento delle prestazioni per includere il riferimento alla tabella stessa.

Il sorgente in plain LuaTEX si semplifica in questo:

```
% !TeX program = LuaTeX
% filename: app-reg/03.tex
```

```
\directlua{register = require "mod-history"}

Testo

\directlua{register:log()}
\bye
```

Per compilarlo correttamente il file Lua del modulo chiamato `mod-history.lua` deve essere nel percorso di ricerca. La soluzione più semplice è copiarlo in locale nella directory del sorgente.

A questo punto è facile creare un pacchetto per LaTeX. Non ci sarà alcuna macro utente ma un aggiunta al codice di chiusura del documento con la macro `\AtEndDocument`. Ecco il codice:

```
% filename: app-reg/historylog.sty
\NeedsTeXFormat{LaTeX2e}[2000/06/01]
\ProvidesPackage{historylog}[2020/06/07 Compile task logger]
% load the Lua module
\directlua{
hl_register = require "mod-history"
}
% register an infoline at the end of the history.log file
\newcommand{\hl@logrun}{\directlua{hl_register:log()}}
\AtEndDocument{\hl@logrun}
\endinput
```

Basta caricare il pacchetto e tutte le compilazioni del sorgente verranno registrate come per questo file sorgente per LuaLaTeX. Compilatelo più volte e controllate poi il file locale `history.log`:

```
% !TeX program = LuaLaTeX
% filename: app-reg/doc.tex
\documentclass{article}
\usepackage{fontspec}
\setmainfont{Libertinus Serif}
\usepackage[italian]{babel}
\usepackage{historylog}
\begin{document}
Testo
\end{document}
```

Il file di registro conterrà informazioni simili a queste:

```
03.tex, 141, Sat Apr 10 17:31:35 2021, 0.00073999999999996, luatex
03.tex, 141, Sat Apr 10 17:31:38 2021, 0.00083600000000006, luatex
03.tex, 141, Sat Apr 10 17:31:40 2021, 0.00073299999999998, luatex
```

13.4 Sviluppo

Abbiamo costruito da zero un modulo Lua per inserire in un registro alcune informazioni sulle compilazioni dei sorgenti. Le funzionalità sono minime ma si tratta di un buon punto di partenza per dare risposta a nuove domande: quale struttura dovremo dare al modulo in modo che sia semplice aggiungere nuove funzionalità o modificare la lista delle informazioni registrabili.

A poco a poco prende forma e si precisa la struttura di un *framework* grazie all'attività di progettazione in passi successivi. Ciò che era molto meno evidente con lo sviluppo dei pacchetti LaTeX è invece chiaro con i moduli Lua: la necessità di ricercare soluzioni con l'organizzazione del codice.

13.4.1 Dati per descrivere dati

Nel registro ogni linea contiene informazioni ma nel codice non ne esiste una rappresentazione di alto livello, un oggetto che si possa aggiungere o eliminare, o stampare in un diverso formato.

I concetti espressi sono due: dare la possibilità di scegliere le informazioni da registrare e il loro formato, e rendere indipendente il registro dalle informazioni. Ciò si risolve con l'introduzione di oggetti `Info`, insieme di:

- un identificativo,
- l'informazione testuale da registrare,
- una serie di possibili regole di formato.

Il costruttore di tabelle è una struttura in grado di rappresentare ogni tipo di dato e la seguente è la diretta descrizione dei componenti di `Info`:

```
1   {
2       id = "EngineName",
3       info = status.luatex_engine,
4       fmt = {
5           upper = function (engine_name)
6               return engine_name:upper()
```

```
7            end,
8        }
9    }
```

In questo codice l'informazione registrabile è il nome del compositore. I nomi dei campi dovranno essere sempre gli stessi così che il componente del registro possa elaborarli. Intenderemo il valore associato al campo `info` di tipo stringa o di tipo funzione. In quest'ultimo caso la funzione dovrà attendersi l'oggetto stesso come argomento e dovrà restituire un valore stringa.

Per esempio, nell'oggetto `ElapsedTime` che misura gran parte del tempo di compilazione, il campo `info` è una funzione definita secondo le regole appena enunciate:

```
1    {
2        id = "ElapsedTime",
3        info = function (t)
4            return tostring(os.clock() - t._start)
5        end,
6        _start = os.clock(),
7        fmt = {
8            second = function (t) ... end,
9        }
10   }
```

13.4.2 Nuova implementazione del registro

Una volta stabilito il tipo di dato non rimane che implementare il codice che gestisce gli oggetti `Info`. Le operazioni fondamentali sono:

- installazione di un oggetto `Info` nel registro, metodo `install()`;
- aggiungere un oggetto a una lista di registrazione specificando una eventuale funzione di formato da applicare, metodo `add_to_log()`;
- rimuovere un oggetto `Info` dalla lista di uscita verso il registro, metodo `remove_from_log()`;
- generare una lista di uscita di registrazione o caricarne una predefinita, metodo costruttore con sintassi di chiamata di funzione;
- impostare un formato tra quelli disponibili di un oggetto `Info` nella lista di uscita, metodo `set_format()`;
- ottenere il testo formattato pronto per la registrazione, metodo `log`.

Non spiegherò tutti i dettagli per non eccedere con la descrizione del codice, tratterò solo i punti salienti lasciando libero il lettore di consultare il codice completo nel file `app-reg/lib-logger.lua`.

Di base, la possibilità di selezionare l'insieme di informazioni da registrare comporta il dover distinguere la lista di oggetti disponbili che chiameremo `_iArchive` da quella degli oggetti che saranno inseriti nell'output di nome `_logList`. La nuova implementazione del registro inizia con il definire le due liste interne di oggetti `Info`:

```
1  local register = {
2      _iArchive = {},
3      _logList = {},
4  }
```

Stiamo usando la convenzione che i nomi dei campi privati inizino tutti con un carattere di trattino basso.

Esaminiamo il metodo `install()` chen come stabilito dall'interfaccia pubblica, ha il compito di verificare la struttura attesa di un generico oggetto `Info` e in caso di successo di memorizzarla nell'archivio interno:

```
1  -- check and install a new `Info` object
2  function register:install(itab)
3      local id = assert(itab.id) -- mandatory string field `id`
4      assert(type(id) == "string")
5      -- have we an `id` already saved?
6      local arch = self._iArchive
7      assert(not arch[id], "id '"..id.."' is not new")
8      -- check `info` field
9      assert(itab.info, "`info` field not found for '"..id.."'")
10     -- check `fmt` field
11     local fmt = itab.fmt
12     if fmt then
13         assert(type(fmt) == "table")
14         local flag = false
15         for k, f in pairs(fmt) do
16             flag = true
17             assert(type(k) == "string")
18             assert(type(f) == "function")
19         end
20         assert(flag, "Empty format function table")
21     end
22     arch[id] = itab
```

```
        return self
    end
```

L'archivio d'installazione è una tabella dizionario indicizzata con il valore del campo `id` presente nella tabella dell'oggetto `Info` stesso. In alternativa, avrei potuto rendere esterno il nome dell'oggetto passandolo come parametro esplicito al metodo `install()`.

Come forma di verifica dei dati abbiamo usato la funzione `assert()`, e non una più sofisticata gestione degli errori poiché contiamo sul fatto che chi utilizza la libreria non abbia esigenze d'impostazione frequenti ma intervenga di tanto in tanto nel modificare le registrazioni delle compilazioni.

Più complessa è la struttura che definisce la lista delle informazioni da registrare. Il motivo è che il codice deve essere eseguito in due distinti momenti: al caricamento del pacchetto e alla fine del documento, altrimenti non potremo registrare l'intervallo di tempo trascorso che in difetto corrisponde al tempo di compilazione.

Quando il codice è caricato, l'oggetto `ElapsedTime` è installato assieme a tutti gli altri tra i predefiniti della libreria, e in conseguenza in quel momento viene letto l'orologio di sistema.

Alla fine del processo di compilazione, le informazioni verranno calcolate e, se presente in output l'oggetto `ElapsedTime`, rileggerà l'orologio misurando il tempo trascorso.

Nulla vieta di implementare altre strategie come quella di rendere la lettura dell'orologio al caricamento della libreria, un dato del registro stesso e poi dare accesso a questo tipo di parametri comuni a tutti gli oggetti `Info`. Sono scelte di progettazione molto importanti che divengono chiare durante lo sviluppo a cui si può dare risposta con criteri di semplicità ma che certamente sono il campo dell'ingegneria del software.

Nell'ambito di questa seconda iterazione di sviluppo, tornando cioè alla strategia di oggetti `Info` autonomi cioè in grado di fornire l'informazione senza dover accedere a parametri del registro, descriviamo la forma della lista di selezione.

Per ciascuna informazione, i dati utili sono il nome identificativo e i dati di eventuali funzioni di formato composti dal nome della funzione seguito da possibili parametri. Uno schema aiuta a disegnare e implementare la lista:

```
{
```

```
2       { <id_info>, <id_fmtFunc>, <param>, <param>, ...},
3       { <id_info>, <id_fmtFunc>, <param>, <param>, ...},
4       ...
5   }
```

Per esempio, se volessimo registrare in prima posizione il nome del compositore utilizzato per la compilazione, oggetto `EngineName`, con un testo tutto in maiuscolo, formato `upper`, all'indice 1 della lista `_logList` ci dovrebbe essere la tabella:

```
1   {
2       {"EngineName", "upper"},
3       ...
4   }
```

Il limite di questa struttura dati è che viene considerata un'unica eventuale funzione formato, anche se con eventuali parametri secondari. Non è possibile applicare quindi una funzione formato su un dato già formattato, limitazione eliminabile modificando la struttura dati.

Cominciamo con il metodo `remove_from_log()`. Serve a eliminare un'informazione dalla lista:

```
1   function register:remove_from_log(id_info)
2       assert(type(id_info) == "string")
3       local t = self._logList
4       local i = assert(
5           self:_find_index(t, id_info), id_info.." not found"
6       )
7       table.remove(t, i)
8       return self
9   end
```

Il metodo si basa su una funzione di utilità `_find_index()` che scorre la lista e restituisce, se esiste, la posizione di un oggetto `Info`:

```
1   function register:_find_index(t, id_name)
2       local i = 1
3       local res
4       while t[i] do
5           if t[i][1] == id_name then
6               res = i
7               break
8           end
```

```
9            i = i + 1
10       end
11       return res
12   end
```

Per evitare la creazione di una closure la funzione ausiliaria è stata definita come membro della libreria, tuttavia la si può sostituire con un dizionario identificatore/posizione da memorizzare nella tabella del registro e aggiornare ogni volta che si modifica la lista.

Esaminiamo ora il codice del metodo `add_to_log()` che aggiunge un'informazione tra quelle da registrare in uscita inserendola nella posizione giusta nella lista:

```
1   function register:add_to_log(info, index)
2       local ll = self._logList
3       local len = #ll
4       local arch = self._iArchive
5       local is_pos
6       local log_entry = {} -- final data
7       if type(info) == "string" then
8           is_pos = self:_find_index(ll, info)
9           log_entry[1] = info
10      else
11          assert(type(info) == "table")
12          local id = info.id
13          if id then -- info is a `Info` object
14              self:install(info) -- check and install
15              log_entry[1] = id
16          else
17              -- [index 1] expected a `Info` tab or a string
18              local o = info[1]
19              local id
20              if type(o) == "string" then
21                  is_pos = self:_find_index(ll, o)
22                  log_entry[1] = o
23                  id = o
24              else
25                  assert(type(o) == "table")
26                  self:install(o) -- check and install
27                  log_entry[1] = o.id
28                  id = o.id
29              end
```

```
30                -- format function?
31                local id_fmt = info[2] -- name of the format function
32                if id_fmt then
33                    assert(type(id_fmt) == "string")
34                    -- check if the fmt function exists
35                    local fmt_repo = arch[id].fmt
36                    if fmt_repo then
37                        assert(fmt_repo[id_fmt])
38                    else
39                        error(
40                            "No format function available for '"..
41                            id.."' Info"
42                        )
43                    end
44                    log_entry[2] = id_fmt
45                    local i = 3
46                    while info[i] do
47                        log_entry[i] = info[i]
48                        i = i + 1
49                    end
50                end
51            end
52        end
53        local id = log_entry[1]
54        assert(arch[id], "`Info` id '"..id.."' not found")
55        if is_pos == nil then -- insert or append
56            if index then
57                assert(type(index) == "number")
58                assert(index >= 1 and index <= len + 1)
59                table.insert(ll, index, log_entry)
60            else
61                ll[#ll + 1] = log_entry
62            end
63        else -- update an existing log entry
64            if index then
65                assert(type(index) == "number")
66                assert(index >= 1 and index <= len)
67                if not (index == is_pos) then
68                    local o = table.remove(ll, is_pos)
69                    table.insert(ll, index, o)
70                    is_pos = index
```

```
71                 end
72             end
73             local id_fmt = log_entry[2]
74             local fmt_repo = arch[id].fmt
75             if fmt_repo then
76                 if id_fmt then
77                     assert(
78                         fmt_repo[id_fmt],
79                         "format funcion '"..id_fmt.."' not found"
80                     )
81                     local entry = ll[is_pos]
82                     entry[2] = id_fmt
83                     local i = 3
84                     while log_entry[i] do -- copy parameters
85                        entry[i] = log_entry[i]
86                        i = i + 1
87                     end
88                     entry[i] = nil -- safe breaking the param list
89                 else
90                     -- rise eventually a warning message
91                     -- "unuseful insertion call"
92                 end
93             else
94                 error(
95                     "Info '"..id..
96                     "' doesn't have any format function"
97                 )
98             end
99         end
100        return self
101    end
```

Il metodo è piuttosto articolato perché prevede un elevato grado di polimorfismo degli argomenti. Con esso possiamo non solo inserire in uscita un'informazione specificandone o meno la posizione, ma anche installarne e inserirne una nuova dandone la definizione, e indicare o meno una funzione di formato con o senza parametri.

In fondo, si installa un nuovo oggetto `Info` solo per utilizzarlo e di questo la libreria tiene conto permettendo all'utente di chiamare un solo metodo. In altre parole, sarebbe assurdo che l'utente installi un tipo di informazione senza poi farne uso.

Il codice di `add_to_log()` è diviso in due parti, nella prima viene esaminato il dato di input e nella seconda viene modificata la lista di informazioni di registro. Vediamone alcuni esempi d'uso.

Per spostare nella posizione 1 con il formato `upper` l'informazione `EngineName` si chiama:

```
reg:add_to_log({"EngineName", "upper"}, 1)
```

e per aggiungere un testo di commento in fondo alla lista creando un nuovo oggetto `Info`:

```
reg:add_to_log{
    id = "CommentText",
    info = "Compilazione di fase due"
}
```

e ancora per inserire in posizione 2 un nuovo tipo di informazione con la larghezza di pagina in punti a tre decimali:

```
local pw = {
    id = "PageWidth",
    info = function () return tex.pagewidth end,
    fmt = {
        point = function (pw, n)
            n = n or 2
            if n == 0 then
                return tostring(p)
            else
                assert(type(n), "number")
                assert(n>0)
                local f1 = "%%0.%df pt"
                local f2 = f1:format(n)
                return f2:format(pw/65536)
            end
        end
    },
}
reg:add_to_log({pw, "point", 3}, 2)
```

Esaminiamo infine il metodo `log()`. Non fa altro che leggere la lista delle informazioni e per ciascuna di esse richiamarne la definizione ottenendo il valore stringa da registrare su cui eventualmente applicare la funzione di formato. Il risultato non è la stringa di testo da memorizzare nel file di

registro, ma ancora un tipo strutturato, cioè la lista delle coppie identificatore oggetto e informazione. Infatti, come vedremo nelle prossime sezioni il componente che gestisce gli oggetti `Info` non si occupa di interagire con file esterni.

```
function register:log()
    local tline = {}
    local arch = self._iArchive
    for pos, info in ipairs(self._logList) do
        local id = info[1]
        local iobj = assert(arch[id])
        local datalog = assert(iobj.info)
        if type(datalog) == "function" then
            datalog = assert(datalog(iobj))
        end
        datalog = assert(tostring(datalog))
        local id_fmt = info[2]
        if id_fmt then
            assert(type(id_fmt) == "string")
            local fn_fmt = assert(
                iobj.fmt[id_fmt], id_fmt.." field not found"
            )
            assert(type(fn_fmt)=="function")
            datalog = fn_fmt(datalog, table.unpack(info, 3))
        end
        tline[#tline + 1] = {id, datalog}
    end
    return tline
end
```

13.4.3 Inizializzare la libreria

Il registro può essere inizializzato o con la lista utente di informazioni in uscita, oppure se questa non è fornita con una lista di default, tuttavia l'utente potrebbe dimenticare di inizializzare la lista. Per questo motivo, è possibile imporre l'inizializzazione nascondendo dietro a una chiamata di funzione il riferimento del modulo del registro.

Con il metametodo `__call()` e le closure di Lua si fa in modo che l'oggetto restituito al caricamento della libreria non sia il modulo atteso `register` ma un oggetto vuoto con l'unica via della chiamata di inizializzazione:

```
\directlua{
local lib_init = require "lib-logger"
local register = lib_init("EngineName", "ElapsedTime", ...)
}
```

Se l'utente non fornisce alcuna lista sarà restituito il riferimento al modulo con la lista di default, altrimenti con la lista utente, come nell'esempio sopra.

Questo comportamento si affida alla tecnica delle metatabelle ed è possibile anche l'alternativa diretta che non esamineremo. Dovreste poter riconoscere il punto nel codice che richiede la costruzione della closure:

```
local mt = {
    __call = function (_, ilist)
        return register:_define_or_default(ilist)
    end
}

local lib = {}
setmetatable(lib, mt)
return lib
```

La funzione ausiliria, sempre dal nome che inizia con un trattino basso a indicare un membro privato, è la seguente che sfrutta il metodo `add_to_log()` perciò l'utente è libero di creare qualsiasi composizione di informazioni, sia con oggetti `Info` predefiniti sia con quelli di propria definizione, con o senza funzioni di formato:

```
function register:_define_or_default(i_list)
    i_list = i_list or self._default_log_entry_list
    assert(type(i_list) == "table")
    for _, info in ipairs(i_list) do
        self:add_to_log(info)
    end
    return self
end
```

13.4.4 Metodi di uscita

Dal registro le informazioni possono essere inviate al file esterno ma nulla vieta di creare un modulo indipendente che implementi diversi formati

e destinazioni. Per questo nel file `lib-logger.lua` si trova il modulo `loglib` che mette a disposizione una o più modalità di output.

I metodi del modulo si attendono in ingresso la lista delle coppie identificatore e informazione, sempre lo stesso tipo di dato, per poi creare una nuova linea di testo in un file nel formato CSV[2], l'unico effettivamente implementato in questa seconda iterazione di sviluppo, oppure inviare i dati a un database, o spedirli in un messaggio di posta elettronica.

13.4.5 Il punto di vista dell'utente

Per usare la libreria l'utente carica il relativo file, inizializza il registro con la lista di default o con una personalizzata, e seleziona la modalità di output. Solamente in caso di nuovi oggetti `Info` l'utente dovrebbe scrivere l'opportuno codice Lua.

Si possono fare esperimenti scaricando il pacchetto di seconda generazione `app-reg/logger.sty` e il file di prova `app-reg/doc2.tex`, secondo le istruzioni della sezione Note di lettura del capitolo di apertura della guida.

Una terza iterazione di sviluppo, potrebbe porsi lo scopo di minimizzare il numero di righe di codice necessarie a implementare un nuovo oggetto `Info`, consentendo di evitare ripetizioni. Su questa idea, il punto chiave potrebbe essere l'inserire gli oggetti `Info` in una gerarchia di classi, implementando in una classe base funzioni di formato che i singoli oggetti ereditano automaticamente.

13.5 Conclusioni

Dalla progettazione della libreria del registro abbiamo imparato due cose: la prima è l'importanza di realizzare gli schemi delle strutture dati perché ciò aiuta sia nel definirle sia nell'implementarne i metodi. In Lua infatti possiamo contare solo sulla tabella.

La seconda, è che lo sviluppo genera non solo risposte ma anche nuove domande, e che perciò la complessità del problema può essere affrontata in successive iterazioni dove una nuova struttura del codice prende il posto di quella dello stadio precedente, fino a raggiungere il framework che risponde ai migliori criteri.

[2] Comma Separated Value.

Tartaglia 14

In questo capitolo utilizzeremo le funzionalità di creazione dei nodi per realizzare il triangolo di Tartaglia. I nodi sono oggetti tipografici pronti da posizionare sulla pagina che vengono prodotti da TEX in uno degli ultimi momenti del processo. LuaTEX è in grado di creare i nodi anche per altra via, tramite la creazione di oggetti in Lua.

I nodi sono di vario tipo, per esempio dimensioni elastiche o glifi, e vengono assemblati in liste anche molto complesse perché strutturate in scatole verticali od orizzontali. Infine, senza entrare troppo nel dettaglio tecnico, i nodi non rientrano nella gestione automatica della memoria operata dal Garbage Collector di Lua, mentre occorre programmare correttamente i riferimenti al nodo seguente e precedente se si vuole creale una lista.

Per questi motivi conviene manipolare le liste dei nodi attraverso le funzioni che si trovano nella tabella `node`, anziché operare su di essi direttamente.

Con i nodi ogni dettaglio deve essere costruito. Si opera come un tipografo che lavora con strumenti elementari, assemblando un pezzo alla volta.

14.1 Costruzione del triangolo di Tartaglia

Il Triangolo di Tartaglia[1] fino all'ottavo livello è qui rappresentato:

[1] https://it.wikipedia.org/wiki/Triangolo_di_Tartaglia.

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1

Ogni nuovo livello è costruito sul precedente sommando i due interi che sovrastano un dato elemento in modo che il primo e l'ultimo numero siano sempre 1. La proprietà più nota del triangolo è che il livello n è formato dai coefficienti binomiali $(a+b)^n$.

Procediamo con il codice. Chiudete la guida e cercate una vostra implementazione in LuaTEX stampando i numeri dei livelli fino all'ottavo su una stessa linea separandoli con uno spazio. Confrontate poi la soluzione fornita nel prossimo listato.

Al solito stiamo procedendo per gradi. Otteniamo prima il codice che produce i numeri del triangolo e poi il codice che costruisce la lista dei nodi da inserire in una scatola che lo compone sulla pagina.

Ecco la mia versione, che utilizza una sola tabella che cresce livello dopo livello:

```
% !TeX program = LuaTeX
% filename: app-tartaglia/01.tex
\directlua{
local nl = string.char(92)..."par"
local t = {}
for row = 0, 8 do
    t[row+1] = 1
    for e = row, 2, -1 do
       t[e] = t[e] + t[e-1]
    end
    tex.print(table.concat(t, " "))
    tex.print(nl)
end
}
\bye
```

14.2 Nodi

I numeri del triangolo vanno posizionati in punti ben precisi. Otterremo la disposizione geometrica regolando le distanze tra i gruppi di cifre in modo che il centro del testo che rappresenta un numero sia sull'asse verticale opportuno per il livello, e disponendo una scatola sull'altra per l'insieme dei livelli. I passi che svolgeremo in plain LuaTEX sono i seguenti:

1. comporre una cifra,
2. comporre un numero in una lista,
3. comporre più numeri in linea congiungendo le liste con un nodo spazio,
4. assemblare una scatola sull'altra.

14.2.1 Un numero

Per costruire un nodo di tipo glifo, un singolo elemento nella collezione di un font, si utilizza la funzione `node.new()`. Poi è obbligatorio valorizzare almeno il codice del carattere per il campo `char` e il numero del font per il campo `font`.

Ottenuto l'oggetto nodo possiamo comporlo sulla pagina con la funzione `node.write()`. Per esempio se volessimo stampare un 8:

```
% !TeX program = LuaTeX
% filename: app-tartaglia/02.tex
\leavevmode\directlua{
local g = node.new("glyph")
g.char = 8 + string.byte("0")
g.font = font.current()
node.write(g)
}
\bye
```

La macro `\leavevmode` è importante perché è vietato inserire un oggetto glifo in modo verticale, ed è questa la modalità in cui si trova TEX all'inizio.

14.2.2 Dal numero alla lista

Dal numero, con l'operatore modulo a 10 è possibile ricavare in un ciclo le cifre componenti la rappresentazione decimale a partire da quella meno significativa. Con la singola cifra si crea il glifo e lo si concatena in

una lista tramite la funzione `node.insert_before()` che funziona anche per aggiungere un elemento in testa.

Curando il caso particolare dello zero, questo è quello che fa la funzione `digit()` nel seguente sorgente compilabile:

```
% !TeX program = LuaTeX
% filename: app-tartaglia/03.tex
\begingroup
\catcode`\%=12
\gdef\percentchar{%}
\endgroup
\leavevmode\directlua{
local function digit(n)
    assert(type(n) == "number")
    local f = font.current()
    local ascii_0 = string.byte("0")
    if n == 0 then
        local g = node.new("glyph")
        g.char = ascii_0
        g.font = f
        return g
    end
    local list
    while n > 0 do
        local digit = n \percentchar 10
        n = (n - digit)/10
        local g = node.new("glyph")
        g.char = digit + ascii_0
        g.font = f
        list = node.insert_before(list, list, g)
    end
    return list
end
local list = digit(12090)
node.write(list)
}
\bye
```

Il metodo `node.write()` accetta un nodo e non una lista. Ma se il nodo argomento ha un riferimento a un nodo nel campo `next`, verrà composta tutta la catena. Questi riferimenti sono stati inseriti per noi da `node.insert_before()`.

Dunque la lista costruita è la sequenza di glifi delle cifre del numero 12090. Dobbiamo ricordarci che il testo composto che ne risulta è tipografia minimale, perché la lista non è stata modificata per inserire legature, kerning, o punti di cesura a fin di riga. Con i nodi siamo noi gli artigiani digitali.

14.2.3 Numeri e spazi

Un nodo *glue* distanzia il nodo precedente da quello successivo, in orizzontale. Può essere elastico in estensione o riduzione, oppure rigido. Per il nostro scopo dovremo calcolare la distanza rigida tra i nodi in modo tale che i centri dei due numeri successivi a un livello del triangolo distino sempre lo stesso valore.

Occorre quindi misurare la larghezza del numero composto. La cosa più semplice è inserire la lista dei nodi glifo in una scatola orizzontale per poi misurarla con la funzione `node.dimensions()` che ne restituisce larghezza, altezza sulla lina base e profondità dalla linea base.

La dimensione tra due scatole dovrà essere la differenza tra la distanza assiale con la semisomma delle larghezze delle scatole adiacenti.

Il passo successivo è quindi aggiungere la funzione `pack_level()` per costruire la scatola orizzontale contentente la lista di un livello intero del triangolo di Tartaglia a partire dalla tabella di interi.

Una scatola orizzontale di una lista si costruisce passando il nodo capolista alla funzione `node.hpack()`. nel codice ho modificato la funzione `digits()` affinché restituisca due paramentri: il nodo della scatola orizzontale che contiene la lista dei nodi glifi e la larghezza della scatola stessa.

Il sorgente compilabile diventa questo:

```
% !TeX program = LuaTeX
% filename: app-tartaglia/04.tex
\begingroup
\catcode`\%=12
\gdef\percentchar{%}
\endgroup
\leavevmode\directlua{
local function pack_digits(n)
    assert(type(n) == "number")
    assert(not (n < 0))
```

```
    local f = font.current()
    local ascii_0 = string.byte("0")
    if n == 0 then
        local g = node.new("glyph")
        g.char = ascii_0
        g.font = f
        local hbox = node.hpack(g)
        local w = node.dimensions(hbox)
        return hbox, w
    end
    local list
    while n > 0 do
        local digit = n \percentchar 10
        n = (n - digit)/10
        local g = node.new("glyph")
        g.char = digit + ascii_0
        g.font = f
        list = node.insert_before(list, list, g)
    end
    local hbox = node.hpack(list)
    local w = node.dimensions(hbox)
    return hbox, w
end

local function pack_level(t)
    local a = tex.sp "24pt"
    local w1
    local head, last
    for _, n in ipairs(t) do
        local hbox, w2 = pack_digits(n)
        if w1 then
            local g = node.new("glue")
            g.width = a - (w1+w2)/2
            w1 = w2
            head, last = node.insert_after(head, last, g)
            head, last = node.insert_after(head, last, hbox)
        else
            w1 = w2
            head, last = node.insert_after(nil, nil, hbox)
        end
    end
```

```
$ luatex 04
This is LuaTeX, Version 1.13.2 (TeX Live 2021)
restricted system commands enabled.
(./04.tex
├─HLIST width: 5pt, height: 6.44pt
│ ╙─head:
│   └─GLYPH char: 0, width: 5pt, height: 6.44pt
├─GLUE width: 14pt
├─HLIST width: 15pt, height: 6.44pt
│ ╙─head:
│   ├─GLYPH char: 8, width: 5pt, height: 6.44pt
│   ├─GLYPH char: 8, width: 5pt, height: 6.44pt
│   └─GLYPH char: 8, width: 5pt, height: 6.44pt
├─GLUE width: 11.5pt
├─HLIST width: 10pt, height: 6.44pt
│ ╙─head:
│   ├─GLYPH char: 8, width: 5pt, height: 6.44pt
│   └─GLYPH char: 8, width: 5pt, height: 6.44pt
├─GLUE width: 11.5pt
├─HLIST width: 15pt, height: 6.44pt
│ ╙─head:
│   ├─GLYPH char: 8, width: 5pt, height: 6.44pt
│   ├─GLYPH char: 8, width: 5pt, height: 6.44pt
│   └─GLYPH char: 0, width: 5pt, height: 6.44pt
├─GLUE width: 11.5pt
└─HLIST width: 10pt, height: 6.44pt
  ╙─head:
    ├─GLYPH char: 8, width: 5pt, height: 6.44pt
    └─GLYPH char: 8, width: 5pt, height: 6.44pt
```

Figura 14.1 Rappresentazione della lista dei nodi contenuta nella variabile `list` del listato `04.tex` modificato per l'utilizzo del pacchetto `nodetree`. Si può notare che i nodi glue tranne il primo hanno lunghezza di 11,5pt perchè le cifre adiacenti sono sempre di due e tre cifre, il che porta a calcolare sempre la stessa distanza per mantenere quella assiale di 24pt con cifre larghe 5pt.

```
52        return head
53    end
54
55    local list = pack_level{0, 888, 88, 880, 88}
56    node.write(list)
57    }
58    \bye
```

Se esageriamo con la grandezza dei numeri allora si sovrapporrano. Questo succede se la lunghezza elastica è negativa poiché la distanza assiale

di 24pt (vedi la variabile locale `'a'`) è troppo piccola. A questo livello del codice, l'utente deve controllare che non ci siano sovrapposizioni specie all'ultima riga del triangolo dove si trovano i numeri più grandi.

14.2.4 Intermezzo: debug di una lista di nodi

Per visualizzare i nodi di una lista per scopi di debug è possibile usare il pacchetto *nodetree* [Fri20] di Josef Friedrich. Uno dei modi è caricare la libreria omonima e usare la funzione `nodetree.print()` per stampare a terminale la rappresentazione dell'albero contenuta nel nodo passato come argomento:

```
local list = ...
local nodetree = require "nodetree"
nodetree.print(list)
```

La documentazione del pacchetto fornisce i dettagli delle opzioni, per esempio per regolare la quantità d'informazioni tramite il parametro *verbosity* o per impostare stili di stampa. L'output può essere dirottato verso un file, specie quando comprende numerose linee di testo.

Con questa tecnica la stampa della rappresentazione dell'albero dei nodi della lista di numeri spaziati ottenuta alla sezione precedente con il codice del file `04.tex`, e contenuta nella variabile `list`, è riportata nella figura 14.1.

Con `nodetree` si possono incontrare problemi se il terminale non è compatibile con i codici di colori e con alcuni caratteri grafici usati per collegare i nodi.

14.2.5 Sovrapposizione scatole

Il passo finale è quello di sovrapporre le scatole orizzontali a formare il triangolo. Basterà impacchettare le scatole in una scatola verticale con la funzione `node.vpack()` dopo aver costruito la lista di scatole e spazi verticali.

Dobbiamo prima modificare la funzione `pack_level()` perché restituisca una scatola orizzontale per il materiale di un intero livello. Fino a ora la lista poteva anche essere una sequenza di scatole e nodi glue perché la immettevamo in modo orizzontale. Adesso invece immettiamo le righe del

triangolo in ambiente verticale impacchettando la lista delle righe separate con un nodo di lunghezza con la funzione `node.vpack()`.

Queste sono le nuove funzioni: `next_level()` calcola la riga del triangolo rispetto a quella precedente, e `tartaglia()` genera il triangolo fino al livello specificato in una scatola verticale:

```lua
-- filename: app-tartaglia/05.tex
local function next_level(t, row)
    t[row+1] = 1
    for e = row, 2, -1 do
        t[e] = t[e] + t[e-1]
    end
end
local function tartaglia(level)
    assert(type(level)=="number")
    local il = tex.sp "8.5pt"
    local head, last
    local t = {}
    for l = 0, level do
        next_level(t, l)
        local hbox = pack_level(t)
        if head then
            local g = node.new("glue")
            g.width = il
            head, last = node.insert_after(head, last, g)
            head, last = node.insert_after(head, last, hbox)
```

1								
1	1							
1	2	1						
1	3	3	1					
1	4	6	4	1				
1	5	10	10	5	1			
1	6	15	20	15	6	1		
1	7	21	35	35	21	7	1	
1	8	28	56	70	56	28	8	1

Figura 14.2 Triangolo di Tartaglia allineato a sinistra, ottenuto con il listato `05.tex`.

```
21            else
22                head, last = node.insert_after(head, last, hbox)
23            end
24        end
25        local vbox = node.vpack(head)
26        return vbox
27    end
```

Il risultato è quello della figura 14.2.

14.2.6 Opzione allineamento

Per allineare al centro o a destra le linee possiamo introdurre dei nodi lunghezza nella scatola orizzontale della singola riga. Conviene inserire questi nodi con la funzione `pack_level()` perché se lo facessimo all'esterno dovremo poi reimpacchettare la lista in un'ulteriore scatola orizzontale per poterle poi sovrapporre in ambiente verticale.

A questo scopo aggiungeremo il parametro `align`. Per dimostrare quanto si riveli utile la dinamicità del linguaggio Lua, considereremo tre diversi possibili gruppi di valori di tipo diverso per il parametro:

- `align` vale `nil`, per esempio perché nella chiamata di funzione principale il valore non è stato assegnato: l'allineamento assume il valore di default di triangolo centrato;
- `align` è una stringa, allora potrà valere `left`, `center` o `right`;
- `align` è un numero come frazione di spazio che deve rimanere a sinistra del primo elemento in alto. Quindi 0 è la stessa cosa dell'allineamento `left`, 1/2 di `center` e 1 di `right`. Sono possibili valori negativi o maggiori di 1.

Il trucco per implementare facilmente l'aggiunta delle lunghezze di allineamento davanti e in coda alla lista degli elementi di un livello del triangolo è quello di conoscere quanto vale lo spazio w da distribuire opportunamente.

Al livello r, se a è la distanza assiale tra numeri adiacenti e r_{tot} è il numero totale dei livelli, allora w vale:

$$w = k_{\text{left}} + k_{\text{right}} = a\,(r - r_{\text{tot}})$$

L'esattezza matematica dell'espressione è dovuta al fatto che il numero in testa e in coda per ogni riga del triangolo è sempre 1.

Il listato completo della funzione `pack_level()` è il seguente:

```
-- filename: app-tartaglia/06.tex
local function pack_level(t, diff_level, k_left, k_right)
    local a = tex.sp "24pt"
    local w1
    local head, last
    if diff_level == 0 then
        k_left, k_right = nil, nil
    end
    if k_left then
        head = node.new("glue")
        head.width = a*diff_level*k_left
        last = head
    end
    for _, n in ipairs(t) do
        local hbox, w2 = pack_digits(n)
        if w1 then
            local g = node.new("glue")
            g.width = a - (w1+w2)/2
            w1 = w2
            head, last = node.insert_after(head, last, g)
            head, last = node.insert_after(head, last, hbox)
        else
            w1 = w2
            head, last = node.insert_after(head, last, hbox)
        end
    end
    if k_right then
        local g = node.new("glue")
        g.width = a*diff_level*k_right
        head, last = node.insert_after(head, last, g)
    end
    return node.hpack(head)
end
```

La funzione tiene conto delle situazioni in cui non è necessario inserire il distanziamento di allineamento su un lato, cioè quando la lunghezza vale zero oppure quando la linea da impacchettare è l'ultima, riga che non ha mai necessità di essere traslata.

Tuttavia, non viene fatto affidamento sulla *direzione di composizione* per allineare a destra o a sinistra e viene inserito sempre lo spazio. Se

l'allineamento fosse a sinistra le scatole sarebbero allineate a sinistra dal compositore che dispone gli oggetti in modo orizzontale da sinistra a destra. Ma se la direzione fosse impostata al contrario l'effetto sarebbe l'opposto.

Per questo nel codice viene inserita la lunghezza a destra nonostante l'allineamento a sinistra. I parametri k_{left} e k_{right} sono definiti dalla funzione principale `tartaglia()` a seconda del parametro `align`. Il listato è il seguente:

```
-- filename: app-tartaglia/06.tex
local function tartaglia(level, align)
    assert(type(level)=="number")
    local k_left, k_right; if align then
        if type(align) == "string" then
            if align == "center" then
                k_left, k_right = 0.5, 0.5
            elseif align == "right" then
                k_right = 1
            elseif align == "left" then
                k_left = 1
            end
        elseif type(align) == "number" then
            if align == 0 then
                k_right = 1
            elseif align == 1 then
                k_left = 1
            else
                k_left, k_right = align, 1 - align
            end
        else
            error("Unexpected alignment value")
        end
    else
        k_left, k_right = 0.5, 0.5
    end
    local il = tex.sp "8.5pt"
    local head, last
    local t = {}
    for l = 0, level do
        next_level(t, l)
        local hbox = pack_level(t, level - l, k_left, k_right)
        if head then
```

```
            local g = node.new("glue")
            g.width = il
            head, last = node.insert_after(head, last, g)
            head, last = node.insert_after(head, last, hbox)
        else
            head, last = node.insert_after(head, last, hbox)
        end
    end
    local vbox = node.vpack(head)
    return vbox
end
```

Domanda: se avessimo avuto numeri diversi da 1 come primo e ultimo elemento, se ritenuto necessario, quali modifiche occorrebbe considerare nel codice?

14.2.7 Verifica grafica degli allineamenti con TikZ

Per controllare visivamente gli allineamenti verticali nel triangolo di Tartaglia è possibile sovrapporre linee verticali sottili di passo a al disegno. Realizzare questo disegno è in realtà molto semplice poiché una volto costruito il nodo del contenitore, la scatola può essere assegnata direttamente a uno dei registri tramite indicizzazione della tabella `tex.box`:

```
% !TeX program = LuaTeX
\newbox\tartbox % nuovo registro
\directlua{
    ... definizioni come prima
    tex.box.tartbox = tartaglia(8)
}
\box\tartbox
\bye
```

La macro `\box` è una primitiva di TEX. Quello che fa è comporre sulla pagina il contenuto del box indicato dalla control sequence che lo segue e poi svuotarlo.

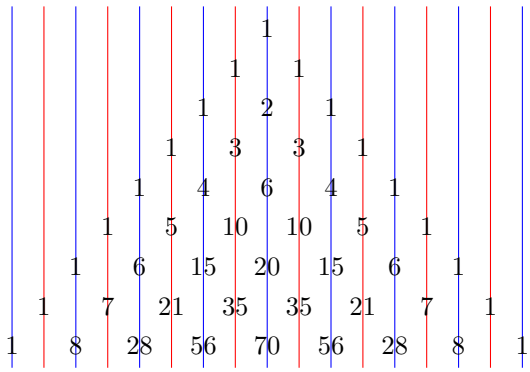
A questo punto è facile separare la costruzione del triangolo dal suo impiego, e un esempio è proprio far espandere la scatola in una macro `\node` del pacchetto grafico TikZ:

```
% !TeX program = LuaTeX
% filename: app-tartaglia/07.tex
```

```
\input tikz.tex
\newbox\tartbox
% ...
\directlua{
    ... definizioni come prima
    tex.box.tartbox = tartaglia(8)
}
\tikzpicture
\foreach \x in {-96,-72,...,96} {
\draw[blue] (\x pt,68pt) -- (\x pt,-68pt);
}
\foreach \x in {-84,-60,...,84} {
\draw[red] (\x pt,68pt) -- (\x pt,-68pt);
}
\node at (0, 0) {\box\tartbox};
\endtikzpicture
\bye
```

Le rette rosse e quelle blue distano 24pt una dall'altra. Sono posizionate a partire dall'ascissa zero poiché TikZ inserirà la scatola usando il suo punto centrale nell'origine del sistema di riferimento.

Le linee rosse corrispondono alle posizioni dei numeri sui livelli dispari, e quelle blu a quelle dei livelli pari. Il risultato è:



14.2.8 Regolazione automatica della distanza

Sappiamo che la distanza a tra i centri di due numeri consecutivi su una linea del triangolo è fissa. Se non fosse abbastanza grande i due numeri

si sovrapporrebbero e a quel punto l'utente dovrebbe reimpostarne il valore nel sorgente e ricompilare.

Possiamo invece rendere l'operazione automatica e in diversi modi. Per esempio, potremo intendere che il triangolo venga costruito in base a una distanza minima tra un numero e l'altro, oppure impostando la distanza a come fissa per incrementarla in caso di sovrapposizioni.

Una prima soluzione è costruire la scatola con il triangolo solo alla fine, salvare cioè le scatole orizzontali dei numeri in una tabella e nel frattempo calcolarne il valore minimo di a; costruire poi la scatola contenitore del triangolo distanziando opportunamente i nodi.

Una seconda strada è quella di impacchettare il triangolo come fatto fino a ora negli esempi, per poi eventualmente scorrere la lista dei nodi per incrementare la distanza tra i centri. Questa seconda strategia è quella che seguirò per mostrare come una lista di nodi già costruita possa essere utilmente modificata.

14.2.9 Modificare la distanza

Useremo la funzione `pack_level()` per creare la lista di un livello del triangolo di Tartaglia già scritta in precedenza, e una nuova funzione `add_distance()` per modificare la distanza tra i centri.

Alcune informazioni utili tratte dal manuale di LuaTEX che è bene richiamare: la lista contenuta in un nodo scatola orizzontale o verticale inizia con il nodo contenuto nel campo `head`. Nella lista il nodo successivo può essere ricavato leggendo il campo `next` del nodo attuale. Il campo numerico `id` indica il tipo di nodo, per esempio `12` individua un nodo `glue` e `0` un nodo `hbox`.

Aggiungere una distanza fissa è molto semplice. Sappiamo che il nodo `hbox` è l'alternanza tra scatole orizzontali e nodi `glue`, perciò se `hbox` è il mnome di variabile che contiene la scatola, il riferimento `hbox.head.next` punta al primo nodo distanza. In un ciclo `while` scorrere i soli nodi `glue` significa saltare un nodo e quindi preparare il prossimo riferimento con il campo `glue.next.next`:

```
1   -- filename app-tartaglia/08.tex
2   local function add_distance(hbox, d)
3       assert(hbox and hbox.id == 0)
4       local glue = hbox.head.next
```

```
    while glue do
        assert(glue.id == 12)
        glue.width = glue.width + d
        glue = glue.next.next
    end
end
```

Nel triangolo dobbiamo tener in conto tuttavia degli eventuali nodi di spaziatura iniziale e finale per l'allineamento orizzontale del triangolo. In questi spazi l'incremento della distanza è proporzionale alla differenza tra il numero dei livelli totali con il numero di quello corrente. Dovremo adattare la funzione `add_distance()` per modificare le lunghezze non solo dei nodi intermedi tra un numero e l'altro, ma anche per gli eventuali spazi di allineamento citati, in questo modo:

```
-- filename app-tartaglia/09.tex
local
function add_distance(hbox, d, k_left, k_right, level, totlevel)
    assert(hbox and hbox.id == 0)
    local glue = hbox.head
    local tdist = d*(totlevel - level)
    if k_left then
        assert(glue.id == 12)
        glue.width = glue.width + tdist*k_left
        glue = glue.next
    end
    glue = glue.next
    for _= 1, level do
        assert(glue.id == 12)
        glue.width = glue.width + d
        glue = glue.next.next
    end
    if k_right then
        assert(glue.id == 12)
        glue.width = glue.width +  tdist*k_right
    end
end
```

Non si utilizza il ciclo `while` ma un ciclo `for` che itera tante volte quanti sono gli spazi nel livello, ovvero il numero del livello a cominciare da 1 perché il livello 0 non ha spazi intermedi. In questo modo è più semplice

per il codice gestire il puntatore al nodo invece che passare da un nodo al successivo.

La versione finale contenuta nel file `app-tartaglia/09.tex`, conta 160 linee di codice Lua in grado di generare il triangolo di Tartaglia con il numero di livelli richiesti, diverse opzioni di allineamento orizzontale, e con la capacità di mantenere la distanza assiale tra i numeri di 24pt più l'eventuale distanza perché ci siano almeno 3pt tra due numeri consecutivi.

Domanda: se e come cambiereste il codice per considerare la simmetria dei numeri sulla riga di uno stesso livello del triangolo di Tartaglia?

14.3 Riepilogo

La tecnologia dei nodi consente di comporre oggetti tipografici di complessità arbitraria. Seguendo vari passi di sviluppo, in questo capitolo abbiamo costruito con essa il triangolo di Tartaglia, un esempio applicativo interessante proprio per poter implementare nuove funzionalità come quella di rendere variabile l'allineamento o lasciare che sia il codice ad aggiustare la distanza tra i numeri se necessario.

Rimane da esplorare la gestione in Lua delle opzioni e dei parametri perché l'utente possa modificare l'aspetto del triangolo. Considero questo tema separato da quello dei nodi di LuaTEX. Per questo motivo l'esercitazione può dirsi conclusa.

Note finali

I miei ringraziamenti vanno a Claudio Beccari per aver scritto la classe *guidatematica.cls* con cui è stata composta la guida e per avermi dato risposte come sempre precise e complete alle mie domande.

Ringrazio Gianluca Pignalberi che mi ha proposto un uso avanzato del pacchetto *siunitx* nella composizione della tabella proposta nel secondo tutorial.

Ci sono sempre buone occasioni per imparare e speriamo che i maestri non manchino mai.

Bibliografia

[BG17] Claudio Beccari e Tommaso Gordini. *Introduzione alle codifiche in entrata e in uscita*. Guide Tematiche del GuIT. GuIT. 26 Ott. 2017. 34 pp. URL: https://www.guitex.org/home/images/doc/GuideGuIT/introcodifiche.pdf (visitato il 01/04/2021).

[Esf21] Behdad Esfahbod. *HarfBuzz text shaping library*. The FreeType Project. 2021. URL: https://harfbuzz.github.io/ (visitato il 01/04/2021).

[Fri20] Josef Friedrich. *The `nodetree` Package*. v2.2. Leggibile con il comando `texdoc nodetree`. 23 Ott. 2020. 44 pp. URL: https://ctan.org/pkg/nodetree (visitato il 28/04/2021).

[Ier16] Roberto Ierusalimschy. *Programming in Lua, Fourth Edition*. 4ª ed. Lua.Org, 2016. ISBN: 8590379868.

[IFC21] Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes. *Lua 5.3 Reference Manual*. 1 Apr. 2021. URL: https://www.lua.org/manual/5.3/.

[Lua21] Development Team LuaTeX. *LuaTEX Reference Manual*. v1.13. Leggibile con il comando `texdoc luatex`. 14 Feb. 2021. 318 pp. URL: https://ctan.org/pkg/luatex (visitato il 01/04/2021).

[Pal21] Mike Pall. *The LuaJIT Project*. 2021. URL: https://luajit.org/ (visitato il 01/04/2021).

[Pég12] Manuel Pégourié-Gonnard. *The `luacode` Package*. v1.2a. Leggibile con il comando `texdoc luacode`. 23 Gen. 2012. 10 pp. URL: https://ctan.org/pkg/luacode (visitato il 01/04/2021).

[Wri20] Joseph Wright. *siunitx — A comprehensive (SI) units package*. v2.8c. Leggibile con il comando `texdoc siunitx`. 25 Feb. 2020. 96 pp. URL: https://ctan.org/pkg/siunitx (visitato il 01/04/2021).

Indice Analitico

Simboli

A

B

C

D

E

F