Lua (autore: Vittorio Albertoni)

Premessa

Lua è stato creato nel 1993 da Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes, membri del Tecgraf/PUC-RIO presso la Pontifícia Università Cattolica di Rio de Janeiro.

Da sempre software libero, attualmente è rilasciato sotto licenza MIT.

Non si sa perché si chiami Lua, che nel portoghese di Rio significa Luna.

La sua creazione è dovuta soprattutto alle forti limitazioni che fino al 1992 esistevano in Brasile relativamente all'importazione di hardware e software.

E' un linguaggio abbastanza facile da apprendere, almeno per le cose di base, di pochissimo ingombro, interpretato ma velocissimo.

Con tutti questi superlativi non si comprende come mai quasi nessun comune mortale ne abbia mai sentito parlare.

In realtà ha alcuni difetti rispetto ad altri linguaggi più conosciuti ed utilizzati.

Innanzi tutto non ha la ricchezza di librerie che può vantare, per esempio, un Python, nato un paio di anni prima, in territorio in cui circolava liberamente il software, e che quando è nato Lua aveva già una vasta ed affermata community che creava librerie su librerie.

Come linguaggio in sé ha qualche difetto che lo rende in certo modo incompleto rispetto ai linguaggi più noti. Per esempio, pur avendo modo di surrogare la classica programmazione per oggetti, non conosce le classi e non supporta l'ereditarietà, inoltre non gestisce le eccezioni.

Rimangono le doti di potenza, leggerezza e velocità, legate all'attitudine di funzionare negli ambienti più disparati (tutte le versioni di Unix e Windows, mainframe IBM, dispositivi mobili con sistema operativo Android, iOS, BREW, Symbian, Windows Phone, etc.).

Queste doti rendono il linguaggio adatto per sistemi embedded, cioè per essere utilizzato per determinati compiti all'interno di progetti sviluppati con altri linguaggi.

Aggiungere Lua a un'applicazione non ne aumenta di molto il peso in termini di byte. Ad esempio, il codice sorgente e la documentazione dell'ultima versione di Lua, stanno in circa 20.000 righe di codice C che pesano 245K compressi e 960K non compressi.

Per questi motivi Lua, senza che lo si sappia, è stato utilizzato in molte applicazioni industriali (come Adobe Lightroom e Photoshop) ed è attualmente il linguaggio di scripting più utilizzato al mondo per lo sviluppo di videogiochi.

In questo manualetto, da buoni dilettanti che non possono esagerare con l'embedding, cominceremo a conoscere Lua come linguaggio all purpose per creare autonomi script di utilità o, semplicemente, per divertirci con il coding fine a sé stesso.

Con un accenno descrittivo, sul finale, ad un paio di esempi importanti di embedding.

Indice

1 Installazione **3**

2 Come funziona **3**

3 Tipi **4**
3.1 Tipi base 4
3.1.1 Numeri 4
3.1.2 Stringhe 4
3.1.3 Valori booleani 4
3.2 Tipi oggetto 4
3.2.1 Funzioni 5
3.2.2 Tabelle 5

4 Variabili **5**

5 Operatori **5**
5.1 Operatori aritmetici 5
5.2 Operatori di confronto 5
5.3 Operatori logici 6
5.4 Operatori per stringhe 6

6 Funzioni **6**

7 Tabelle **6**

8 Manipolazione di stringhe e tabelle **7**
8.1 Stringhe 7
8.2 Tabelle 7

9 Matematica **8**

10 Interattività con l'utente **8**

11 Input e output su file **9**

12 Strutture di controllo **9**
12.1 Esecuzione condizionale 9
12.2 Ripetizione 10

13 Esempi di script **11**

14 Moduli **13**

15 LuaTeX **14**

16 Solar2D **16**

1 Installazione

Lua si trova all'indirizzo *https://www.lua.org/*.

Qui troviamo la descrizione del linguaggio, la documentazione (fondamentale il Reference Manual in inglese) e il tarball del codice sorgente dell'interprete.

La più recente versione disponibile nel momento in cui scrivo (Marzo 2023) è la 5.4.4 rilasciata il 13/1/2022.

Per l'installazione, senza scaricare nulla dal sito, possiamo procedere nel modo seguente.

Linux

Scriviamo i seguenti comandi a terminale:

```
curl -R -O http://www.lua.org/ftp/lua-5.4.4.tar.gz
tar zxf lua-5.4.4.tar.gz
cd lua-5.4.4
make linux test
make install
```

Versioni non aggiornatissime del linguaggio si possono trovare nel repository della distro e possono essere installate con il gestore del software.

Mac

Scriviamo i seguenti comandi a terminale:

```
curl -R -O http://www.lua.org/ftp/lua-5.4.4.tar.gz
tar zxf lua-5.4.4.tar.gz
cd lua-5.4.4
make macosx test
make install
```

Windows

All'indirizzo *https://luabinaries.sourceforge.net/download.html* troviamo gli zip degli eseguibili per sistemi Windows a 32 bit e a 64 bit.

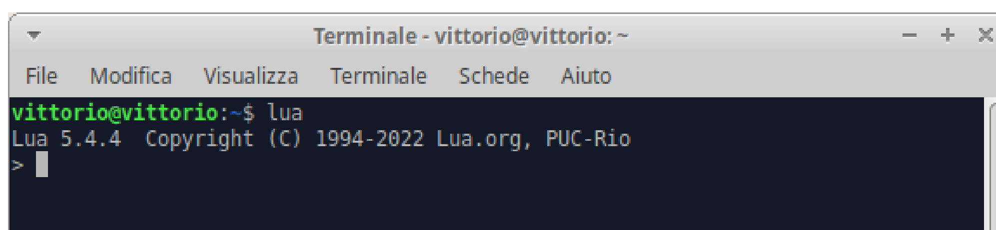
Probabilmente la versione disponibile non sarà aggiornatissima con l'ultimo rilascio del sorgente.

Il file zip va scompattato in una cartella dedicata nella directory C:\WINDOWS e qui troviamo l'eseguibile `luaXX.exe`, dove XX corrisponde alla versione.

2 Come funziona

Lua è un linguaggio interpretato e l'interprete si lancia con il comando `lua` a terminale, se lavoriamo su Linux o Mac, o con il comando `luaXX` nel prompt dei comandi se lavoriamo su Windows (previo inserimento nella variabile di sistema PATH del percorso alla directory dove si trova il relativo file).

Con questi comandi il terminale o il prompt dei comandi di Windows diventa una REPL (Read Eval Print Loop) di Lua. Su un mio computer equipaggiato Linux appare così



Se scriviamo sulla riga del prompt (quella che inizia con il simbolo >) un'istruzione nel linguaggio Lua e premiamo Invio, nella riga successiva vediamo il risultato dell'istruzione stessa.

Se le istruzioni sono molte e concatenate tra loro meglio scriverle in una pagina di editor di testo, creando così un programma, che per i linguaggi interpretati come Lua si chiama script, e salvare in un file con estensione `.lua`.
Con il comando a terminale `lua` seguito dal nome di questo file si eseguono le istruzioni che esso contiene.

Possiamo utilizzare qualsiasi editor di testo, purché non sia un word processor.

Se abbiamo a disposizione Geany per la programmazione va benissimo. Esso riconosce le istruzioni Lua e le colora in modo da evitarci errori di scrittura. Inoltre possiamo eseguire lo script dall'editor stesso.

Esiste tuttavia un editor creato apposta per Lua che, oltre a ciò che fa Geany, ci offre la code completion, cioè ci suggerisce i comandi dopo che ne abbiamo scritte alcune lettere. Si chiama ZeroBrane Studio e lo troviamo all'indirizzo *https://studio.zerobrane.com/*.
Andiamo alla scheda SCARICA cliccando sulla relativa linguetta e poi clicchiamo sulla scritta TAKE ME TO THE DOWNLOAD PAGE THIS TIME.
Dalla pagina che si apre possiamo scaricare gli installer per Linux, Mac e Windows gratuitamente (essendo ringraziati nel caso avessimo pagato e, in caso contrario, essendo invitati, senza obbligo, a fare un'offerta).

3 Tipi

Lua è un linguaggio a tipizzazione dinamica e non richiede che il tipo di dato da elaborare sia preventivamente indicato dall'utente: a seconda della sintassi con cui è espresso il dato da elaborare Lua assegna al dato stesso il tipo.

I tipi di Lua sono tipi base e tipi oggetto.

3.1 Tipi base

I principali tipi base, in tutto simili a quelli che troviamo in ogni linguaggio di programmazione, sono i seguenti.

3.1.1 Numeri

Per i numeri, siano essi interi, siano essi in virgola mobile, abbiamo il tipo `number`.

Per default Lua tratta i numeri con precisione a 64 bit, cioè possiamo avere numeri precisi fino alle 19 cifre, virgola compresa nel caso di decimali.

3.1.2 Stringhe

La stringa (`string`) è una sequenza immutabile di caratteri racchiusa tra apici singoli (') o doppi (").

3.1.3 Valori booleani

I valori booleani (`boolean`) sono `true` (vero) e `false` (falso).

3.2 Tipi oggetto

I tipi oggetto sono elementi del linguaggio che generano valori e che, per riferimento, possono essere assegnati a variabili. Sono sostanzialmente due.

3.2.1 Funzioni

La funzione (`function`) è un frammento di codice che ha un nome e che, prendendo o meno delle variabili in ingresso, esegue una o più operazioni e restituisce uno o più valori in uscita.

3.2.2 Tabelle

La tabella (`table`) è l'unica struttura dati, del tutto particolare, del linguaggio Lua. Essa consiste in un array associativo che può essere indicizzato con numeri, con lettere o con stringhe.

4 Variabili

La variabile è una locazione di memoria destinata a contenere un valore che possiamo manipolare con istruzioni contenute nello script. In Lua una variabile può contenere un valore di tipo base ma anche funzioni e tabelle, che abbiamo appena definito tipi oggetto.

Il valore a una variabile si assegna con l'operatore di assegnazione `=`.

Con l'assegnazione di un valore la variabile viene creata, senza bisogno che sia stata preventivamente dichiarata, ed assume il tipo del valore assegnato.

Per default la variabile creata è una variabile globale, cioè può essere utilizzata e manipolata in tutto lo script.

Per creare una variabile locale, cioè che possa essere utilizzata soltanto all'interno di un blocco di istruzioni, occorre crearla con la parola chiave `local`.

`a = 15.75` crea la variabile globale `a` assegnandole il valore 15.75
`local nome = 'Pippo'` crea la variabile locale `nome` assegnandole il valore Pippo

La variabile acquisisce il tipo del valore assegnatole.

Esiste la funzione `type()` per verificarlo.
Date le variabili esemplificate sopra,
`type(a)` fornisce il risultato `number`,
`type(nome)` fornisce il risultato `string`.

5 Operatori

Gli operatori collegano tra loro operandi in espressioni che forniscono un risultato.

5.1 Operatori aritmetici

In ordine di precedenza di esecuzione sono i seguenti:
^ per l'elevamento a potenza,
* per la moltiplicazione,
/ per la divisione,
% per il modulo (resto della divisione intera),
+ per la somma,
- per la sottrazione.

5.2 Operatori di confronto

Servono per confrontare due valori e il risultato che restituiscono è un valore booleano.
== uguale,
~= non uguale,
< minore,
<= minore o uguale,
> maggiore,
>= maggiore o uguale.

5.3 Operatori logici

Forniscono come risultato un valore booleano e sono i seguenti:
`and` per l'AND logico,
`or` per l'OR logico,
`not` per il non.

5.4 Operatori per stringhe

`..` operatore di concatenamento.

Per esempio
`'Vittorio ' .. 'Albertoni'` ritorna `Vittorio Albertoni`

6 Funzioni

La funzione si crea con la seguente sintassi

```
function <nome>(<eventuali_parametri>)
   <istruzioni>
end
```

dove
<nome> è l'identificativo della funzione,
<eventuali_parametri> sono gli eventuali argomenti da passare alla funzione,
<istruzioni> indicano cosa deve fare la funzione.

La funzione si utilizza richiamandone l'identificativo seguito da due parentesi tonde all'interno delle quali si indicano gli eventuali argomenti richiesti (nel caso separati da virgola).

Creata la funzione

```
function saluta()
   print('Ciao')
end
```

con
`saluta()` otteniamo la scritta `Ciao`.

Creata la funzione

```
function saluta(chi)
   saluto = 'Ciao '..chi
   print(saluto)
end
```

con
`saluta('Pippo')` otteniamo la scritta `Ciao Pippo`

Come abbiamo visto nei Capitoli 3 e 4 possiamo assegnare la funzione ad una variabile e richiamarla attraverso quella variabile.

Possiamo, per esempio assegnare l'ultima funzione creata sopra alla variabile s con l'istruzione
`s = saluta`

e poi, con
`s('Vittorio')` ottenere la scritta `Ciao Vittorio`.

7 Tabelle

La tabella si crea con l'istruzione
<identificatore> `= {}`
che crea una tabella vuota con il nome dell'identificatore.

Con
`t = {}` creiamo una tabella vuota denominata `t`.

La tabella si popola con la sintassi
<identificatore>[<indice>] = <valore>
dove
<identificatore> è il nome dato alla tabella,
<indice> è un numero, una lettera o una stringa,
<valore> è un'entità di uno dei tipi che abbiamo visto nel Capitolo 3.

Una volta creata la tabella vuota `t`,
con `t[1] = 10` inseriamo il numero 10 nella posizione indicizzata con `1`,
con `t['a'] = 'Ciao'` inseriamo la stringa Ciao nella posizione indicizzata con `a`,
con `t['salve'] = saluta` inseriamo la funzione che abbiamo creato nel Capitolo 6 nella posizione indicizzata con la stringa `salve`.

Ciò che abbiamo scritto nella parte sinistra di queste istruzioni crea una variabile. Nel caso specifico abbiamo creato le variabili `t[1]`, `t['a']` e `t['salve']`.

Possiamo verificarne il tipo con
`type(t[1])`, che ritorna `number`,
`type(t['a'])`, che ritorna `string` e
`type(t['salve'])`, che ritorna `function`.

Con l'struzione `t['salve']()` otteniamo lo stesso risultato che otterremmo con `saluta()` in quanto la variabile `t['salve']` contiene la funzione `saluta()`.

E' possibile costruire la tabella inserendo direttamente i campi con la sintassi
t = {<valore>, <valore>, <valore>, ...}
nel qual caso l'indicizzazione avviene numerando i campi con una successione di numeri interi a partire da 1.

La tabella creata sopra, contenente i valori 10, 'Ciao' e saluta si potrebbe creare così
`t = {10, 'Ciao', saluta}`
e il valore 10 avrebbe indice 1, il valore 'Ciao' avrebbe indice 2 e la funzione saluta avrebbe indice 3.

Ricorrendo al manuale troviamo altri modi e semplificazioni per alimentare e indicizzare le tabelle ma per noi principianti è meglio che ci fermiamo qui.

8 Manipolazione di stringhe e tabelle

I tipi `string` e `table` sono le cose più originali del linguaggio Lua, che ci offre numerose funzioni per manipolare le variabili che contengono valori di questi tipi.

Qui richiamo quelle che ritengo di più frequente utilizzo. Per riferimento completo rimando ai capitoli 6.4 e 6.6 del Reference Manual di Lua.

8.1 Stringhe

Le funzioni più comuni per manipolare stringhe sono le seguenti:
`string.len(s)` restituisce la lunghezza della stringa `s`,
`string.rep(s,n)` restituisce una stringa contenente la stringa `s` ripetuta `n` volte,
`string.upper(s)` restituisce la stringa `s` in lettere maiuscole,
`string.lower(s)` restituisce la stringa `s` in lettere minuscole,
`string.sub(s,a,b)` restituisce una stringa iniziando con il carattere di indice `a` e finendo con il carattere di indice `b` della stringa `s`

8.2 Tabelle

Le funzioni più comuni per manipolare tabelle sono le seguenti:
`table.concat(t)` restituisce una stringa formata da tutti gli elementi di `t`,
`table.concat(t, " ")` restituisce una stringa formata da tutti gli elementi di `t`, separati da uno spazio,

`table.concat(t, ", ")` restituisce una stringa formata da tutti gli elementi di `t`, separati da una virgola,
`table.insert(t, `<valore>`)` inserisce un <valore> alla fine della tabella `t`,
`table.insert(t, `<indice>`, `<valore>`)` inserisce un <valore> nella posizione <indice> della tabella `t`,
`table.remove(t, `<indice>`)` rimuove l'elemento corrispondente a <indice> dalla tabella `t`.

9 Matematica

Lua ha una libreria di funzioni matematiche. Le principali e di uso più comune sono le seguenti:
`math.abs(x)` ritorna il valore assoluto di `x`,
`math.ceil(x)` ritorna il più piccolo intero più grande o uguale a `x`,
`math.exp(x)` ritorna il valore di e^x,
`math.floor(x)` ritorna il più grande intero inferiore o uguale a `x`,
`math.log(x)` ritorna il logaritmo naturale di `x`,
`math.log(x, b)` ritorna il logaritmo di `x` in base `b`,
`math.pi` ritorna il valore di π,
`math.random()` genera un numero pseudocasuale compreso tra `0` e `1`,
`math.sqrt(x)` ritorna la radice quadrata di `x`.

Vi sono poi le funzioni trigonometriche `math.sin(x)`, `math.cos(x)` e `math.tan(x)` che ritornano, rispettivamente, seno, coseno e tangente di `x` espresso in radianti.
E le relative inverse `math.asin(x)`, `math.acos(x)` e `math.atan(x)`.

Per convertire da radianti a gradi e viceversa abbiamo le funzioni
`math.deg(x)` converte l'angolo `x` da radianti a gradi,
`math.rad(x)` converte l'angolo `x` da gradi a radianti.

Abbiamo, infine, la funzione
`math.type(x)` che, per `x` valore numerico o identificativo di variabile contenente un valore numerico, cioè di tipo `number`, ce ne indica il sottotipo `integer` o `float`.

10 Interattività con l'utente

L'interfacciamento con l'utente avviene attraverso la REPL su terminale utilizzando tastiera e schermo.

Per l'input abbiamo a disposizione la funzione
`io.read()`
che legge come stringa quanto introduciamo da tastiera.
Il fatto che il valore sia letto come stringa non ci deve preoccupare se l'input serve per fare calcoli: Lua converte automaticamente il valore nel tipo adatto agli operatori che si utilizzano.

Per l'output abbiamo la funzione
`print()`
che utilizzata senza argomenti produce una riga bianca e che all'interno delle parentesi accetta come argomento ciò che deve essere scritto, espresso o in forma scalare, come stringa o numero, o come identificativo di una variabile contenente il valore da scrivere o come espressione matematica per scriverne il risultato.

Esempi:
`a = io.read()` legge ciò che scriviamo sulla tastiera e lo assegna alla variabile a di tipo string.
Se scriviamo un numero, ad esempio 34, esso viene letto come stringa.
Ma se facciamo a + 2 otteniamo tranquillamente il risultato 36.
`print(7.65)` scrive 7.65,
`print('Ciao')` scrive Ciao,
`print(5 * 6)` scrive 30,
`print(A)`, se la variabile A contiene il valore 12, scrive 12.

11 Input e output su file

Per lavorare su un file dobbiamo aprirlo ed assegnarlo ad una variabile.
La variabile può avere un nome qualsiasi, a me piace, in questo caso e senza sprecare troppa fantasia, chiamarla `f`, iniziale di file.
La funzione per aprire il file è questa, con la seguente sintassi:
`io.open(`<nome_file>, <modo>`)`
dove
<nome_file> è una stringa che contiene il percorso al e il nome del file,
<modo> dichiara con un carattere il motivo per cui apriamo il file:
`"w"` per scriverci sopra cancellandone il precedente contenuto,
`"a"` per scriverci sopra in aggiunta al precedente contenuto,
`"r"` per leggere il contenuto.
Con l'istruzione

```
f = io.open("/home/vittorio/Documenti/prova", "w")
```

assegno alla variabile `f` il file `prova` che si trova nella cartella Documenti della mia home in un sistema Linux e lo predispongo per scriverci dentro. Se il file prova non c'è viene automaticamente creato.

La variabile `f` (o come abbiamo preferito chiamarla), che contiene il file, è dotata di metodi per scrivere sul file e leggere dal file con questa sintassi:

`f:write(`<valore>`)`
scrive nel file <valore> senza andare a capo;
<valore> può essere una stringa, un numero, un'espressione matematica che genera il risultato da scrivere, il nome di una variabile che contiene ciò che si vuole scrivere;
possiamo scrivere più elementi separandoli con la virgola;
per andare a capo, se abbiamo scritto una stringa la concludiamo con `\n`;
negli altri casi inseriamo la stringa `"\n"` dove vogliamo andare a capo.
Questo metodo, se usato in modo `"w"` sostituisce quanto scriviamo a ciò che c'era prima;
se usato in modo `"a"` aggiunge quanto scriviamo a ciò che c'era prima.

`f:close()`
inserisce effettivamente quanto abbiamo scritto nel file e lo chiude.

`f:read()`
se il file è stato aperto in modo `"r"` legge la prima riga del file e va sulla successiva;
per leggere la successiva occorre ripetere il comando.
Per leggere tutto il file occorre inserire tra le parentesi tonde la stringa `"*all"`.

12 Strutture di controllo

Come ogni altro linguaggio di programmazione, Lua ha dei comandi per condizionare l'esecuzione di certe istruzioni al verificarsi di determinate condizioni oppure per la ripetizione dell'esecuzione di una o più istruzioni.

12.1 Esecuzione condizionale

if

L'istruzione `if`, chiamata istruzione di esecuzione condizionale (in inglese if è il nostro se), ci dà modo di assoggettare l'esecuzione di un blocco di istruzioni al verificarsi di una determinata condizione: se la condizione è vera viene eseguito il blocco di istruzioni, altrimenti si prosegue l'esecuzione del programma saltando il blocco stesso.

La sintassi è la seguente

```
if <condizione> then
<istruzioni>
end
```

dove <condizione> è una qualsiasi espressione che relaziona due valori attraverso operatori di confronto: se la condizione si verifica vengono eseguite la o le istruzioni indicate prima della parola chiave `end`, altrimenti si passa oltre.

L'istruzione `if` si presta anche all'esecuzione condizionale a due rami. Per ottenere questo dobbiamo abbinarla all'istruzione `else` con questa sintassi

```
if <condizione> then
<istruzioni>
else
<istruzioni>
end
```

In questo caso se la condizione si verifica vengono eseguite le istruzioni contenute nel primo blocco, prima della parola chiave `else`, altrimenti vengono eseguite quelle contenute nel secondo blocco dopo `else` (altrimenti). In ogni caso proseguendo poi nell'esecuzione del resto del programma.

Possiamo infine gestire l'esecuzione condizionale a più rami abbinando a `if` l'istruzione `elseif` (tutto attaccato) con questa sintassi

```
if <condizione> then
<istruzioni>
elseif <condizione> then
<istruzioni>
elseif <condizione> then
<istruzioni>
...
else
<istruzioni>
end
```

12.2 Ripetizione

for

Si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni per un numero definito di volte.

La sintassi per l'uso di questa istruzione è la seguente:

```
for i = 1, n do
<istruzioni>
end
```

La situazione più semplice è quella che utilizza una sorta di contatore, una variabile creata al volo tradizionalmente indicata con `i`, che assuma un certo numero di valori:

```
for i = 1, 5 do
println("Ciao")
end
```

scrive `Ciao` cinque volte.

while

Si usa per ripetere istruzioni fino a quando si verifica una certa condizione.

La sintassi è:

```
while <condizione> do
<istruzioni>
end
```

Se la condizione è espressa attraverso l'uso di un contatore otteniamo gli stessi risultati che otteniamo con l'istruzione `for` vista prima

```
i = 1
while i <= 5 do
print "Ciao"
i = i + 1
end
```

scrive cinque volte la parola `Ciao`.

repeat until

L'istruzione repeat...until (ripeti...finché) permette di eseguire un ciclo finché una condizione rimane vera. La sintassi è:

```
repeat
<istruzioni>
until <condizione>
```

Per scrivere cinque volte la parola Ciao con questa istruzione facciamo così

```
N = 5
i = 0
repeat
print('Ciao')
i = i + 1
until i == N
```

Rispetto al ciclo for, dove la condizione è controllata all'inizio del ciclo, nel ciclo repeat...until la condizione è controllata alla fine del ciclo. Ciò spiega perché, nel caso del ciclo for il contatore i inizia da 1 e nel caso del ciclo repeat...until inizia da 0.

13 Esempi di script

Le istruzioni di Lua possono essere eseguite nella REPL e, scritta l'istruzione e premuto INVIO, ne vediamo il risultato nella stessa REPL.

Nel caso di istruzioni complesse, come la costruzione di una funzione o di un controllo di ciclo, possiamo utilizzare più righe: premendo INVIO, fino a quando l'istruzione non è completata, la nuova riga su cui scrivere assume il simbolo di prompt >> e soltanto quando Lua capirà che l'istruzione è completa la eseguirà, ritornando al prompt >.

Questo modo di procedere si addice comunque più alla sperimentazione ed allo studio che non alla vera e propria programmazione.

Tra l'altro tutto ciò che facciamo in questo modo va irrimediabilmente perduto una volta usciti dalla REPL.

Se scriviamo le nostre istruzioni in un file di testo e lo salviamo con estensione `.lua` creiamo un programma che possiamo utilizzare anche successivamente. Dal momento che questo programma non viene compilato in linguaggio macchina ma viene eseguito da un interprete esso, più propriamente, si chiama script e lo si esegue digitando a terminale il nome del file, con relativa estensione, preceduto dal comando `lua`.

Per la scrittura di questo file di testo non dobbiamo seguire regole particolari, salvo quella di andare a capo con INVIO alla fine di ogni istruzione o di ogni parte di istruzione complessa, tenendo presente che Lua è case sensitive, cioè riconosce e distingue lettere maiuscole e minuscole. La variabile `nome` è diversa dalla variabile `Nome`.

Se nel testo vogliamo inserire commenti e scritte che non debbano influire sull'esecuzione dello script dobbiamo precederle con una doppia lineetta (`--`).

Questo piccolo script

```
print('Come ti chiami?')
nome = io.read()
print('Ciao, '..nome.."!")
```

chiede il nome all'utente per salutarlo.

Quest'altro chiede due numeri per sommarli e fornire il risultato

```
-- script che somma due numeri
print('Primo numero') -- primo addendo
a = io.read()
print('Altro numero') -- secondo addendo
b = io.read()
c = a + b
print("La somma di "..a.." e "..b.." è:  "..c)
```

Ho inserito commenti per descrivere che cosa fa il programma e il significato delle richieste dei dati.

Il banalissimo esempio serve a dimostrare la grande elasticità di Lua nel trattare i tipi di dato.
Le variabili `a` e `b`, create e inizializzate leggendole dalla tastiera con `io.read()`, sono di tipo `string`.
Quando le utilizzo con l'operatore `+`, destinato a dati numerici, per creare e alimentare la variabile `c` esse vengono, per l'occasione, convertite al tipo `number` e la variabile `c` acquisisce questo tipo.
Questa stessa variabile `c`, quando nell'ultima riga viene inserita nella descrizione del risultato, essendo preceduta dall'operatore `..` destinato a concatenare stringhe, si converte, per l'occasione, al tipo `string` e tutto funziona senza la necessità di intervenire con operazioni di casting.

Non conosco altro linguaggio di programmazione in cui ciò possa avvenire.

Ora uno script un tantino più laborioso, destinato a risolvere equazioni di secondo grado del tipo

$$ax^2 + bx + c = 0$$

```
print("Coefficiente di x^2")
a = io.read()
print("Coefficiente di x")
b = io.read()
print("Termine noto")
c = io.read()
delta = b^2 - 4*a*c
if delta < 0 then
    print("Non esistono radici reali")
elseif delta == 0 then
    print("Unica radice:  "..-b/(2*a))
else
    r1 = (-b+math.sqrt(delta))/(2*a)
    r2 = (-b-math.sqrt(delta))/(2*a)
    print("Prima radice:  "..r1)
    print("Seconda radice:  "..r2)
end
```

Le indentature non sono richieste ma servono a leggere meglio lo script.

Se lavoriamo su Linux possiamo inserire, come prima riga dello script, l'indirizzo dell'interprete con la seguente sintassi
`#!/usr/local/bin/lua`
In questo modo possiamo rendere eseguibile il file che contiene lo script e lanciarlo come tale[1].

[1] Rammento che per rendere eseguibile un file basta cliccare destro su di esso nel gestore dei file, scegliere PROPRIETÀ, aprire la scheda PERMESSI e selezionare l'opzione PERMETTI DI ESEGUIRE IL FILE COME UN PROGRAMMA. Possiamo anche farlo da terminale con il comando `chmod 555` seguito dal nome del file.

14 Moduli

Tutto ciò che abbiamo visto finora possiamo farlo con la semplice installazione di base dell'interprete Lua e, per come l'ho presentato, rappresenta il modo più facile di usare il linguaggio per fare le cose meno complicate.

Se leggiamo il manuale, alquanto ostico e privo di qualsiasi esempio che faciliti la comprensione di quanto leggiamo, vediamo che, sempre con la semplice installazione di base, possiamo fare anche cose più complicate da professionisti, come utilizzare le tabelle per surrogare quanto facciamo con linguaggi orientati agli oggetti, come utilizzare il linguaggio insieme al linguaggio C, ecc.

Quando le complicazioni diventano tali che l'installazione di base non basta il mondo di Lua, per la verità ben lontano dalle dimensioni e dalla ricchezza di contenuti, per esempio, del mondo Python, ci offre una raccolta di librerie, che, in Lua, si usano chiamare Moduli.

Noi stessi potremmo costruire moduli, con gli strumenti che ci offre l'installazione di base, e tenerli a disposizione per quando servono ma ritengo che queste cose da professionisti esulino dall'interesse dei principianti dilettanti cui mi rivolgo.

I moduli prodotti dalla community di Lua si trovano catalogati all'indirizzo
https://luarocks.org/
dove possiamo fare ricerche per categoria trovando descrizioni e documentazione non sempre alla portata di dilettanti.

Per installare un modulo sul nostro computer in modo da poterlo utilizzare in arricchimento del nostro Lua dobbiamo disporre del software LuaRocks, che possiamo installare in questo modo:

. sistema Linux
    si trova sicuramente nel repository della nostra distro
. sistema Mac
    con il comando a terminale `brew install luarocks`
. sistema Windows
    dall'indirizzo *http://luarocks.github.io/luarocks/releases/*

A quest'ultimo indirizzo troviamo anche i tarball per l'installazione su Linux nel caso nel repository della nostra distro non ci sia.

Una volta installato questo software e individuato il modulo che ci interessa installare, possiamo procedere all'installazione con il comando a terminale
`luarocks install` <nome_modulo>
dove nome modulo va inserito come tale ed è quello che ci suggerisce la ricerca che abbiamo fatto per trovarlo.

A questo punto è bene verificare se e dove l'installazione sia avvenuta e lo facciamo con il comando a terminale
`luarocks show` <nome_modulo>
e ci verrà mostrato il luogo del file system in cui è avvenuta l'installazione.

Ora apriamo la REPL di Lua e vi inseriamo il comando
`package.path`
che ci mostra i luoghi del file system nei quali l'interprete Lua che abbiamo installato va a cercare i moduli e molto probabilmente ci accorgiamo che il luogo dove è stato installato il modulo non è tra quelli in cui il nostro Lua lo cerca.

Purtroppo nel mondo Lua c'è un po' di confusione.

Se vogliamo poter utilizzare il modulo, aprendo il gestore dei file come superutenti, dobbiamo trasferire quanto installato in posizione adeguata.

A questo punto possiamo utilizzare il modulo richiamandolo in Lua con
`require` <nome_modulo>
dove <nome_modulo>, questa volta, va scritto come stringa tra apici singoli o doppi.

Tenuto conto di quanto precede e del fatto che la documentazione sui moduli Lua è piuttosto ostica e, in molti casi, scritta proprio per addetti ai lavori, devo concludere col dire che

la facilità del linguaggio Lua finisce non appena vogliamo uscire da ciò che possiamo fare con l'installazione di base.

15 LuaTeX

Ho detto in premessa che Lua, data la sua leggerezza, si presta all'embedding con altri linguaggi.

Un caso interessante è quello della versione estesa del software `pdftex` che utilizza Lua come linguaggio incorporato, dando origine al software `luatex`, che consente di arricchire la potenza dei motori TeX e LaTeX per la produzione di file in formato PDF[2].

All'indirizzo *https://www.pragma-ade.com/general/manuals/luatex.pdf* troviamo il manuale completo di LuaTeX e chi voglia approfondire vi trova quanto gli serve.

Nell'economia di questo manualetto per principianti mi limito a richiamare le cose più semplici e di più immediata utilità.

Innanzi tutto dobbiamo installare il software.
All'indirizzo *https://www.luatex.org/index.html* abbiamo la possibilità di farlo, ma, se già abbiamo installato il software per LaTeX probabilmente già lo abbiamo ed è comunque molto più semplice installarlo con lo stesso sistema con cui abbiamo installato LaTeX. Nel caso di texlive il package si chiama **texlive-luatex**.

I comandi per generare il file PDF sono:
. `luatex` <nome_file>`.tex` se il file è stato prodotto in codice Plain Tex,
. `lualatex` <nome_file>`.tex` se il file è stato prodotto in codice Latex,
. se abbiamo prodotto il testo con l'editor LyX lo dobbiamo esportare scegliendo nel menu la modalità `PDF(LuaTex)`.

In alcuni casi non dobbiamo nemmeno usare codice Lua per ottenere il risultato.

Nel caso, per esempio, che ci interessi produrre un file PDF da codice LaTeX con un carattere diverso da quelli standard di LaTeX, purché il carattere che vogliamo utilizzare sia installato sul nostro sistema, basta che inseriamo nel codice LaTeX, nelle righe iniziali, i comandi

```
\usepackage{fontspec}
\setmainfont{<nome_carattere>}
```

e il documento che scriveremo verrà riprodotto nel carattere scelto.

Il file `esempio_1.tex` contenente questo codice LaTeX

```
\documentclass{article}
\usepackage{fontspec}
\setmainfont{olivier}
\begin{document}
Ciao, Vittorio!
\end{document}
```

compilato con il comando

```
lualatex esempio_1.tex
```

produce un file PDF con il testo

Ciao, Vittorio!

Ma la potenza dell'embedding si manifesta anche diversamente, dandoci modo di inserire nel testo che scriviamo degli spezzoni di codice Lua destinati a produrre risultati che verranno evidenziati nel testo stesso.
Per ottenere questo utilizziamo il comando

```
\directlua{<codice_lua>}
```

dove <codice_lua> può essere scritto anche su più righe e verrà eseguito durante la compilazione (istruzioni collocate su una sola riga si separano con punto e virgola).

---

[2] Di TeX e LaTeX ho parlato nel mio blog all'indirizzo *https://www.vittal.it* a più riprese:
. nell'allegato pdf_lyx all'articolo «Scrivi e pubblica veri libri» del novembre 2017,
. nell'allegato musica_latex all'articolo «Latex e la musica» del marzo 2022,
. nell'allegato plain_tex all'articolo «Plain Tex» del giugno 2022.

La funzione Lua per scrivere il risultato in questo contesto è `tex.print()`.

Il file esempio_2.tex contenente questo codice Plain TEX

```
Il valore di $\pi$ \'e all'incirca \directlua{tex.print(math.pi)}
\bye
```

compilato con il comando

```
luatex esempio_2.tex
```

produce il testo

Il valore di π è all'incirca 3.1415926535898.

Il file esempio_3 contenente questo codice Plain TEX

```
L'area di un cerchio di raggio 23,5 \'e
\directlua
{
area = 23.5^2 * math.pi
tex.print(area)
}


Fine del documento.
\bye
```

compilato con il comando

```
luatex esempio_3.tex
```

produce il testo

L'area di un cerchio di raggio 23,5 è 1734.944542945
Fine del documento.

Allo stesso risultato perveniamo con il seguente codice LATEX dell'esempio 4:

```
\documentclass{article}
\begin{document}
L'area di un cerchio di raggio 23,5 è
\directlua
{
area = 23.5^2 * math.pi
tex.print(area)
}


Fine del documento.
\end{document}
```

compilato con il comando

```
lualatex esempio_4.tex
```

Per chi usa l'editor LYX con questo input

Il valore di π è all'incirca `\directlua{tex.print(math.pi)}`.
L'area di un cerchio di raggio 3 è `\directlua{area = 3^2*math.pi;tex.print(area)}`.

esportando con `PDF(LuaTex)` si ottiene il seguente testo su file PDF:

Il valore di π è all'incirca 3.1415926535898.
L'area di un cerchio di raggio 3 è 28.274333882308.

La parte colorata nei rettangoli inserisce il codice Lua come fosse codice TEX (menu di LYX INSERISCI ▷ CODICE TEX oppure pulsante TEX nella barra degli strumenti).

Uno potrebbe commentare che ci sono mille modi di fare i conti fuori da TEX e LATEX per poi inserire i risultati prodotti con questi conti nei testi prodotti da TEX e LATEX. Occorre tuttavia riconoscere che questo embedding, oltre che farci ammirare una bella soluzione informatica, ci rende tutto più comodo, soprattutto se i calcoli sono complicati.

16 Solar2D

Altro bell'esempio di embedding quello della Corona Labs Inc., che ha scelto il linguaggio Lua per lo sviluppo di videogiochi con il framework Corona SDK.

Da quando l'azienda ha chiuso i battenti, nel 2020, i suoi prodotti sono diventati open-source e Corona SDK è diventato Solar2D, software libero a pieno titolo con licenza MIT.

Non è proprio cosa da dilettanti: per progettare cose avvincenti nel campo dei videogiochi occorre essere bravi professionisti.

Segnalo comunque l'esistenza di questo favoloso software con il quale possiamo produrre giochi per sistemi mobile iOS, Android, Amazon Kindle, Windows Phone, tvOS e Android TV, e applicazioni desktop per Linux, Mac e Windows.

Troviamo Solar2D all'indirizzo *https://solar2d.com/*.

Nella home page abbiamo una descrizione del software e dei suoi pregi.

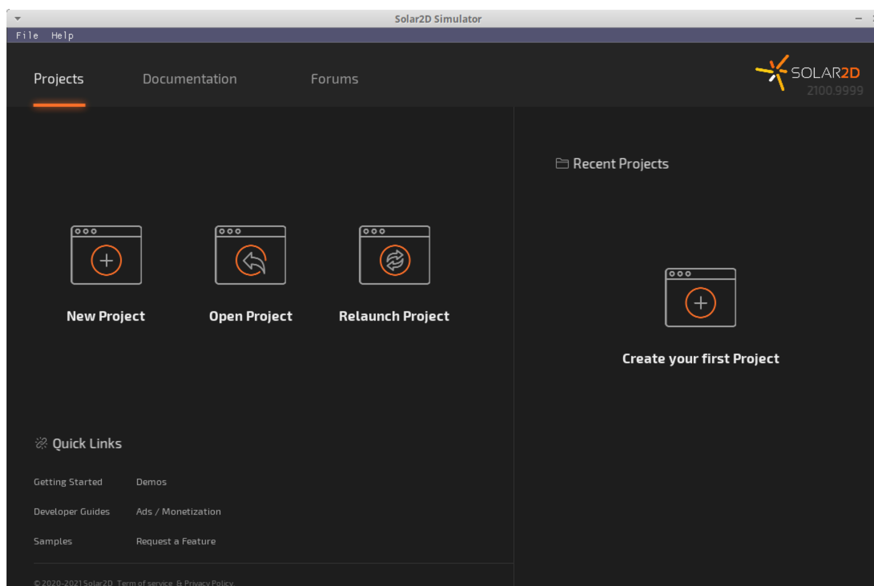
Cliccando sul pulsante DOWNLOADS andiamo nella pagina da cui possiamo scaricare gli installer per Mac, Windows e per Linux su snap[3]. Per Windows ci viene proposto di scaricare l'installer `.msi` e per Mac il file `.dmg`.

Per la documentazione clicchiamo su DOCUMENTATION e veniamo reindirizzati su
https://docs.coronalabs.com/
dove troviamo tutto ciò che c'è da sapere su Solar2D, compresi ottimi tutorial per chi si voglia cimentare.

Per lavorare bene con Solar2D è opportuno installare anche l'editor per Lua ZeroBrane Studio di cui ho parlato nel Capitolo 2.
Questo perché, collegando questo editor al progetto nel modo che vedremo poi, esso ci fornirà la code completion non soltanto per il linguaggio Lua ma anche per l'utilizzo dei numerosissimi oggetti che ci offre Solar2D per creare i nostri programmi di gioco utilizzando questo linguaggio (l'API del motore Corona sottostante a Solar2D ha più di 1000 chiamate): a questo fine, da menu PROJECT ▷ LUA INTERPRETER scegliamo l'opzione CORONA.

Un piccolo banalissimo esempio per capirci.

Apriamo Solar2D e ci troviamo di fronte questa finestra



Se clicchiamo sulla scritta GETTING STARTED abbiamo modo di accedere ad un tutorial in lingua inglese che ci mostra come fare le prime cose un po' più difficili di quella che faremo adesso solo per dimostrare come si usano i tools che abbiamo a disposizione.

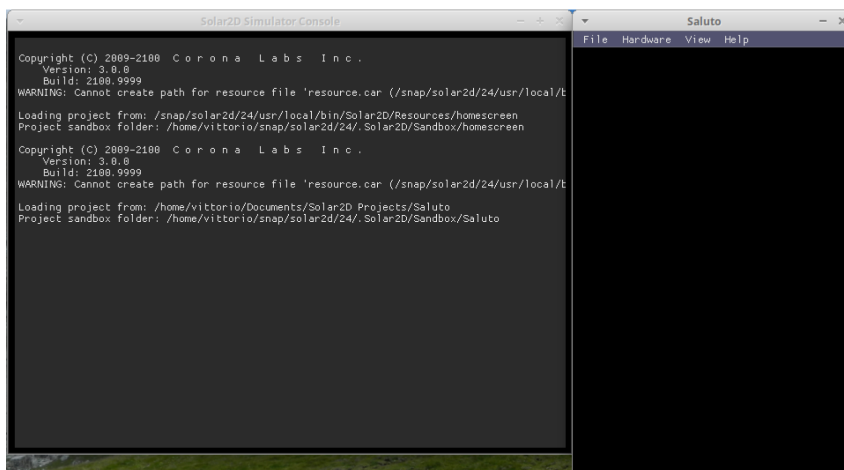
[3] A chi non conosca snap consiglio la lettura dell'allegato pdf «mondo_linux» all'articolo «Le miniere del software libero» pubblicato nell'aprile 2021 sul mio blog all'indirizzo *https://www.vittal.it*

Se è la prima volta che usiamo Solar2D clicchiamo sull'icona, sulla destra, con scritto CREATE YOUR FIRST PROJECT. Se già abbiamo usato Solar2D ed abbiamo ancora archiviati i progetti fatti, qui ne troviamo l'elenco e possiamo riaprirli ed avremo a disposizione l'icona sulla sinistra, con scritto NEW PROJECT per aprire un nuovo progetto.

Si apre così la finestra New Project, nella quale scriviamo, innanzi tutto, il nome che intendiamo dare al nostro progetto: per questo piccolo esercizio scriviamo `Saluto` e lasciamo tutto il resto nell'impostazione di default, secondo la quale il nostro progetto è Blank, cioè parte da zero, e produce una app per schermo di telefono con risoluzione 320x480 (potremmo scegliere uno schermo tablet con risoluzione 768x1024 o riservarci di indicare dimensioni particolari).

In questa finestra vengono proposti luoghi in cui salvare il nostro lavoro. Accettiamoli come proposti e ricordiamocene o cambiamoli a nostro piacimento.

Dato OK, sul nostro schermo compare quanto segue

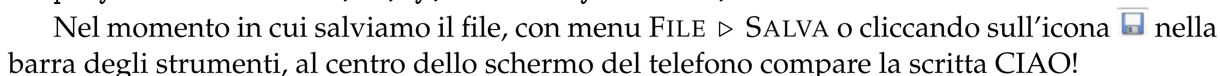


Sulla sinistra abbiamo la console dove ci verrà dato conto di quanto faremo e ci verranno segnalati eventuali errori e sulla destra abbiamo lo schermo di telefono che abbiamo scelto.

Ora apriamo l'editor ZeroBrane Studio e lo colleghiamo al progetto aprendo il menu FILE che abbiamo sulla riproduzione dello schermo del telefono e scegliendo OPEN IN EDITOR.
Il file aperto in ZeroBrane Studio assume così il nome `main.lua` ed è il file guida di tutto il progetto, nel quale, con un mix di linguaggio Lua e di richiami alle API di Corona, programmeremo la nostra applicazione.

Per programmare il nostro piccolo saluto scriviamo nel file `main.lua` quanto segue:

```
local saluto = "CIAO!"
x = display.contentWidth/2
y = display.contentHeight/2
display.newText(saluto, x, y, native.systemFont, 30)
```

Nel momento in cui salviamo il file, con menu FILE ▷ SALVA o cliccando sull'icona  nella barra degli strumenti, al centro dello schermo del telefono compare la scritta CIAO!

Se ora apriamo il menu FILE sopra la finestra del progetto a forma di telefono e scegliamo BUILD, essendo sul sistema Linux come lo sono io, possiamo compilare il nostro progetto come applicazione Linux da eseguire sul PC in uso oppure, con una procedura un tantino più complicata, come app Android. Anche in questo caso ci viene proposto un luogo in cui salvare l'eseguibile e possiamo accettarlo mandandolo a memoria o indicarlo a nostro piacimento.

Una volta compreso il meccanismo dell'uso dei tools, come spero sia avvenuto leggendo quanto precede, possiamo lanciarci a costruire cose più complesse. Ma a questo scopo dobbiamo conoscere le API Corona, e non è cosa da poco, ma, soprattutto, possedere l'inventiva necessaria per creare giochi avvincenti. Sicuramente Solar2D e Lua ci consentono di farlo al massimo livello; il resto dipende da noi.

Se cerchiamo su YouTube `Corona SDK+Lua` possiamo vedere alcuni interessanti e ben fatti tutorial, in lingua italiana, sull'argomento. Purtroppo sono alquanto datati, sia per quanto riguarda Corona SDK che ora è Solar2D, ma praticamente funziona allo stesso modo, sia per quanto riguarda la sintassi del linguaggio Lua che, nel tempo, si è semplificata.