Primi esperimenti con `node` di LuaTEX

Roberto Giacomelli

Sommario

Il nuovo motore di composizione LuaTEX sta per uscire in una release stabile. Questo lavoro è un'introduzione passo passo su alcuni degli elementi interni, come i nodi `rule`, `glue` e `glyph`.

Inoltre sull'idea del contesto reale, viene sviluppato un progetto pilota per riprodurre i codici a barre nella specifica Code 39, allo scopo di svelare le potenzialità delle nuove funzioni e mettere alla prova il linguaggio di programmazione Lua.

Abstract

The new LuaTEX typesetting engine is going to be released in a stable version, ready for powerful applications. This paper is a step-by-step introduction of several LuaTEX internals such as the `rule`, `glue`, and `glyph` nodes.

Moreover, a pilot project is developed as a real design problem, to reproduce barcodes in the Code 39 specification, in such a way to reveal the new features and to dive into the Lua programming language.

1 Introduzione

Il nuovo motore di composizione LuaTEX offre una semplice modalità di scambio dati tra il codice Lua eseguito durante la compilazione del sorgente e il compositore TEX. In dettaglio, la funzione `tex.print()` consente d'inserire nel flusso di token qualsiasi elemento si voglia per farlo poi elaborare ulteriormente da TEX come se lo avessimo digitato manualmente nel sorgente.

Come esempio minimo compilabile di questa *comunicazione*, scriviamo nella macro primitiva espandibile `\directlua` il seguente codice Lua per immettere un testo in maiuscolo:

```
\directlua{
    local str = string.upper("casa")
    tex.print(str)
}
\bye
```

La prima cosa da notare è che questo sorgente non è scritto secondo il linguaggio di un formato utente come LATEX o ConTEXt, ma nella sintassi diretta del motore di composizione LuaTEX. Esattamente come per TEX, in esso non esiste un preambolo e nemmeno la nozione di ambiente. L'unica macro richiesta è `\bye`[1] perché necessaria all'uscita del programma che altrimenti entrerebbe in attesa al raggiungimento della fine del file.

1. Questa macro non è primitiva ma è definita in Plain tramite l'istruzione `\end`.

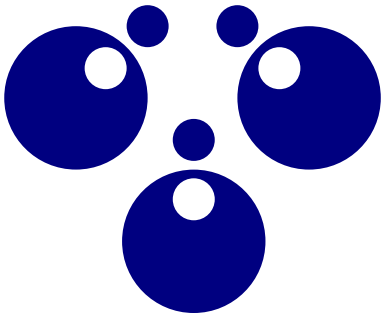

Figura 1: Il logo del progetto LuaTEX (logo by Taco Hoeackwater & Hans Hagen, used by permission).

Inserendo il codice in un file di nome `test.tex` è possibile compilarlo per verificarne il risultato con il comando da terminale:

```
$ luatex test
```

Effettivamente ha funzionato: il codice argomento della macro `\directlua` è stato eseguito dall'interprete Lua interno a LuaTEX e i token della parola CASA sono stati inviati a TEX, che ne ha poi composto l'unico capoverso del documento.

Questa modalità consente facilmente di sfruttare il linguaggio di programmazione Lua per elaborare informazioni e farle poi comporre da TEX. Ciò è davvero notevole se pensiamo che con Lua è possibile connettersi a database o compiere calcoli in virgola mobile, tutti compiti praticamente impossibili con il motore tradizionale.

Ma è naturale chiedersi se esiste un modo più diretto interno a Lua con cui svolgere *direttamente* compiti di composizione. La risposta è sì, grazie alla libreria `node` di LuaTEX; illustrarne piacevolmente alcune caratteristiche di base è lo scopo di questo lavoro.

1.1 Struttura dell'articolo

L'articolo si compone di due parti: la prima è un'introduzione alla libreria `node` con esempi di codici semplici e compilabili al fine di invogliare la sperimentazione diretta da parte del lettore; la seconda è un approfondimento applicativo che mostra i possibili percorsi di un progetto ipotetico per la stampa di codici a barre, con il duplice obiettivo di mostrare le nuove potenzialità di LuaTEX e alcune tecniche di programmazione implementabili in Lua.

1.2 Il linguaggio Lua

Ci sono alcune guide sintetiche disponibili in rete, ma per imparare a programmare in Lua il riferimento più autorevole è certamente il libro scritto proprio dallo sviluppatore principale di questo linguaggio dal titolo "Programming in Lua" (Ierusalimschy, 2013), meglio noto semplicemente come PIL.

I concetti fondamentali da acquisire sono:

- caratteristiche del sistema dinamico dei tipi;
- regole di binding dell'assegnamento;
- basi del tipo *tabella*, l'unico dato strutturato predefinito, e del suo costruttore;
- sintassi dei costrutti di controllo;
- iteratori;
- funzioni come tipi di prima classe.

Studiare il linguaggio invece di tentare d'indovinarne la sintassi vi darà solide basi facendovi risparmiare molto tempo.

1.3 Il motore LuaTEX

Semplificando, possiamo pensare a LuaTEX come il motore di composizione TEX a cui è stato aggiunto un interprete Lua. In qualsiasi momento della compilazione è possibile far eseguire codice Lua che al termine ripasserà il controllo a TEX.

Ma Lua in TEX sarebbe del tutto inutile se non ci fosse un modo per scambiare dati tra i due livelli d'esecuzione. In effetti potremmo far elaborare i dati in modo indipendente e sequenziale in modo che l'uscita del programma Lua, o di qualsiasi altro linguaggio, sia l'ingresso dati per il compositore.

Un report, per esempio, potrebbe essere costruito con un programma che prima interroghi la base dati e poi scriva il file di testo pronto per essere compilato con TEX[2].

Come vedremo in seguito, lo scambio dati da Lua a TEX non solo è possibile, ma la tecnologia dei nodi costituisce una funzionalità notevole e a quanto pare unica tra tutti i programmi tipografici, anche se ancora *work in progress* e a portata di crash.

2 I nodi

In termini generali, il funzionamento del motore di composizione TEX si compone di quattro livelli in cascata:

Input processor: traduzione del flusso di caratteri d'ingresso nella lista di token;

Expansion processor: sostituzione ricorsiva dei token espandibili con i token componenti;

Execution processor: esecuzione dei token non ulteriormente espandibili e invio alle liste principali orizzontali verticale e matematica;

Visual processor: creazione dei capoversi, degli allineamenti, delle pagine, composizione della matematica e scrittura del file `dvi`.

I dati d'uscita di un livello diventano quelli d'ingresso del successivo fino a che i nostri sorgenti non sono trasformati in documenti di alta qualità tipografica.

In LuaTEX il codice Lua è in grado d'inserire dati in due dei processi descritti: nella forma di token nel processo d'espansione e nella forma di nodi nel processo di esecuzione.

Abbiamo già incontrato nell'introduzione la funzione `tex.print()` all'interno di una macro `\directlua`. In esecuzione, questa funzione rilascia l'argomento come informazione tradotta in token. Ciò avviene per esempio con il sorgente seguente:

```
\def\centra#1{\directlua{
   local s = string.upper("#1")
   local m = "\string\\centerline{"..s..."}"
   tex.print(m)
}}
\centra{casa}
\bye
```

un modo complicato di scrivere il codice TEX `\centerline{CASA}` attraverso l'espansione della primitiva `\directlua` poi ulteriormente espanso al passaggio successivo dal processo di espansione.

La presenza della macro `\string` è solo un modo per produrre il testo corretto dopo la lettura del contenuto della macro `\directlua`. Ciò naturalmente non sarebbe necessario se il codice Lua fosse richiamato da un file esterno. Questa lettura permette di passare dati a Lua proprio come si vede nell'esempio dove il comando `\centra` attende un argomento che sarà sostituito all'interno del codice Lua alla riga 2 prima dell'esecuzione.

Più in dettaglio, tutti i caratteri passati alle funzioni `tex.print()` e simili vengono mano a mano inseriti in un'area di memoria e rilasciati nel flusso d'espansione come risultato dell'espansione della macro `\directlua`.

Con la libreria `node` di LuaTEX invece inseriamo non caratteri in fase di espansione, ma elementi tipografici completi direttamente nelle liste orizzontale, verticale o matematica del compositore al terzo livello di elaborazione, già pronti per essere disposti sulla pagina nel processo visuale.

2. L'idea è stata ampiamente sviluppata nell'articolo apparso sul numero scorso di ArsTEXnica (Giacomelli e Pignalberi, 2015). In questo lavoro è anche possibile reperire informazioni utili sul linguaggio Lua, in particolare sul formato *data description* cioé sull'idea di usare il costruttore di tabelle per creare un archivio dati testuale interpretabile direttamente da Lua.

2.1 Nodi `rule`

Riferendoci al manuale della versione beta 0.80[3] di LuaTEX, ovvero la versione inclusa in TeX Live 2015, alla libreria `node` viene dedicato il capitolo 4, che elenca le funzioni disponibili, e il capitolo 8, che elenca i tipi di nodi con i singoli parametri. Tra i vari tipi di nodi troviamo il tipo `rule` che ci permetterà di fare i primi semplici esperimenti.

Un nodo `rule` rappresenta un rettangolo pieno che corrisponde esattamente a quelli prodotti con la macro `\rule` di LATEX. Per costruire un filetto di 10pt di larghezza e 5pt di altezza, con la macro `\rule` scriviamo:

```
\rule{10pt}{5pt}
```

e il risultato è .

Con la libreria `node` è possibile ottenere la stessa cosa programmando in Lua. Il codice è abbastanza semplice: prima si crea un nuovo oggetto `node` di tipo `rule` passando semplicemente la stringa del nome alla funzione `node.new()`, poi si impostano le dimensioni in punti scalati con assegnazione diretta ai campi dimensionali e infine si immette l'oggetto composto nel flusso di uscita con la funzione `write()`:

```
A rule 10pt width and 5pt height:
\directlua{
local r = node.new("rule")

r.width  = 10*65536
r.height =  5*65536

node.write(r)
}.
\bye
```

Compilando questo sorgente si ottiene il risultato atteso perché, senza esserne consapevoli, abbiamo scritto il nodo quando TEX si trovava nel modo orizzontale all'interno di un capoverso. Se non ci si trovasse in `hmode`, come quando siamo all'inizio di un nuovo capoverso, accadrebbe l'inatteso: per verificarlo basta inserire nel codice precedente una riga vuota tra la prima e la seconda.

Una volta inseriti i nodi, per assicurarci che si trovino in modo orizzontale possiamo inserire la macro `\leavevmode` prima del codice Lua oppure impacchettare il nodo all'interno di un nodo che rappresenta un box, come vedremo in seguito nel dettaglio.

2.2 I punti scalati

Prima di passare al prossimo oggetto `node`, soffermiamoci un momento sulle unità di misura dimensionali dei *punti scalati* `sp`. TEX opera internamente in aritmetica intera e converte in punti scalati le misure espresse in una delle altre otto unità di misura ammesse, per esempio dai punti `pt`, dai pollici `in`, dai millimetri `mm` o dai centimetri `cm`.

Tabella 1: Fattori interi di conversione tra le unità di misura delle lunghezze accettate dal motore di composizione TEX e l'unità fondamentale dei punti scalati, ricavati con la funzione `tex.sp()` di LuaTEX.

Simbolo	Unità di misura	Fattore
`cm`	centimetro	1 864 679
`mm`	millimetro	186 467
`in`	pollice	4 736 286
`pt`	punto tipografico	65 536
`pc`	pica	786 432
`dd`	punto Didot	70 124
`cc`	cicero	841 489
`bp`	punto grande	65 781
`sp`	punto scalato	1

Un punto scalato `sp` equivale esattamente a una parte su 65 536 di un punto tipografico `pt` — una distanza veramente molto piccola — ed è l'unità fondamentale di TEX, quindi anche di LuaTEX. Perciò quando si lavora con la libreria `node` le dimensioni devono essere moltiplicate per i fattori riportati nella tabella 1 per la conversioni necessarie verso i punti scalati.

Nella libreria `tex` di LuaTEX esiste la funzione `sp()` che accetta una stringa nel formato ⟨*numero*⟩⟨*unità*⟩. Nel codice precedente avremo potuto quindi scrivere:

```
r.width  = tex.sp "10pt"
r.height = tex.sp " 5pt"
```

Questa funzione è interessante perché rende il codice più comprensibile e può essere impiegata per convertire un dato introdotto direttamente dall'utente, per esempio quando egli impiega una macro che richiede di specificare una lunghezza. Naturalmente l'uso diretto di fattori di conversione è più efficiente e lascia allo sviluppatore qualche libertà in più rispetto alla precisione di conversione.

2.3 Nodi `glue`

Occupiamoci ora di un secondo tipo di nodo che rappresenta uno spazio tra due elementi: il tipo `glue`. Paradossalmente lo spazio vuoto in TEX è molto più complicato di quello che si potrebbe a prima vista immaginare. Non è infatti solo una dimensione rigida, ma ammette una variazione dinamica in funzione delle esigenze compositive come una sorta di molla. Questa *elasticità* è definita da due misure, una per il massimo accorciamento chiamata `shrink` e una per il massimo allungamento chiamata `stretch`.

Per creare un nodo `glue` lo schema è identico al precedente:

1. creazione del nodo con la funzione di libreria `node.new(`⟨*type name*⟩`)`;

3. Ad oggi, la versione più recente di LuaTEX, la 0.89.0 datata 4 febbraio 2016, non è disponibile ovviamente in TEXLive ma nei repository di codice del progetto, oppure installando ConTEXt minimal `http://wiki.contextgarden.net/ConTeXt_Standalone`.

2. impostazione di parametri interni dell'oggetto;
3. scrittura in uscita dell'oggetto verso il motore di composizione.

Nella versione attuale di LuaTEX le proprietà elastiche di uno spazio si impostano attraverso un ulteriore oggetto nodo chiamato `glue_spec`, cosa che nel futuro potrebbe cambiare con l'unione dei due tipi. Un esempio di codice è il seguente, nel quale si lascia uno spazio di 18pt — rigido — tra due gruppi di lettere:

```
ABC\directlua{
local spec18pt = node.new("glue_spec")
spec18pt.width = 18*65536

local space = node.new("glue")
space.spec = spec18pt

node.write(space)
}DEF
\bye
```

2.4 Interazione con la lista TEX

Prendendo spunto dal codice precedente, se voleste inserire due spazi consecutivi identici attraverso i nodi, cosa scrivereste?

Prima di proseguire nella lettura pensateci un momento e magari provate a compilare un vostro esempio. La mia prima soluzione — sbagliata — per risolvere questo problema è stata quella di duplicare la chiamata della funzione `write()` con lo stesso oggetto nodo, ritrovandomi così con il programma bloccato:

```
ABC\directlua{
local space = node.new("glue")
space.spec = node.new("glue_spec")
space.spec.width = 18*65536

% errore!
node.write(space)
node.write(space)
}DEF
\bye
```

Per capire cosa è avvenuto occorre ricordarsi che TEX processa gli elementi tipografici inserendoli in una lista, una struttura dati dove ogni elemento mantiene un riferimento al successivo.

Come recita il manuale di LuaTEX, la funzione `write()` — peraltro ancora sperimentale — copia in coda alla lista di TEX solamente il riferimento al nodo e non l'intero oggetto. Quando TEX elabora il primo nodo non riesce quindi a scorrere la lista perché il nodo successivo è ancora un riferimento allo stesso nodo.

Questo ci insegna che con la libreria `node` è molto facile corrompere la lista principale usata da TEX per comporre gli elementi tipografici. Con le versioni stabili di LuaTEX i messaggi d'errore e la documentazione potranno aiutare gli sviluppatori di applicazioni a divenire esperti evitando questo genere di errori.

2.5 Collegare insieme i nodi

A questo punto non vi sorprenderà sapere che anche i nodi di LuaTEX sono in realtà potenziali elementi di una lista. Più precisamente, invece di elaborare i nodi in modo indipendente uno dall'altro, li si collega in una lista e la si invia a TEX indicando solamente il primo nodo.

Per provare come possa apparire il codice diamoci un nuovo problema: disegnare dieci quadrati di 10pt di lato a distanza di 10pt uno dall'altro. Sapendo che ogni oggetto nodo ha un campo `next` in cui memorizzare il riferimento al nodo successivo, provate a prendere carta e matita per scrivere la vostra soluzione a questo esercizio.

Chiudete però le pagine della rivista perché tra un attimo riporterò il codice scritto da me — incurante dei pericoli — con cui potrete confrontare le vostre soluzioni per ottenere questo simpatico disegno:

```
\leavevmode\directlua{
local tenpt = 10 * 65536
local firstElem = node.new("rule")
firstElem.width = tenpt
firstElem.height = tenpt

local lastElem = firstElem

for _ = 2,10 do
    local s = node.new("glue")
    s.spec = node.new("glue_spec")
    s.spec.width = tenpt
    lastElem.next = s
    lastElem = s

    local q = node.new("rule")
    q.width = tenpt
    q.height = tenpt
    lastElem.next = q
    lastElem = q
end

node.write(firstElem)
}
\bye
```

Quello che ho fatto è impostare manualmente i campi `next` dei nodi per *appendere* uno alla volta i successivi alla lista, mantenendo un riferimento all'ultimo e al primo nodo da passare alla funzione `write()`. Tuttavia, è certamente più sicuro formare la lista con le funzionalità interne di LuaTEX sia per maggiore compatibilità con le versioni future sia perché i nodi, avendo anche il campo `prev`, dovrebbero formare liste doppiamente concatenate.

Ci viene in aiuto la funzione `insert_after()` della libreria `node`, in grado di aggiungere un nuovo elemento a una lista in base alla posizione di uno dei suoi nodi. Gli argomenti previsti da questa funzione sono il riferimento al primo nodo della

lista, il riferimento al nodo dopo il quale verrà inserito il nuovo elemento e il riferimento al nuovo nodo. I parametri restituiti sono due: il riferimento al primo nodo della lista e quello al nuovo nodo aggiunto.

Questa funzione — assieme a `insert_before()` — può anche svolgere il ruolo di *costruttore* della lista, perché quando il riferimento al nodo di testa assume il valore `nil`, valore assunto anche dal secondo parametro, varrà il caso di *lista vuota*.

Volendo usare queste funzioni, si dovrà però fare attenzione a che il riferimento al nodo subito dopo il quale si vuole inserire quello nuovo appartenga alla stessa lista individuata dal nodo primo argomento. Il codice può essere riscritto così:

```
\leavevmode\directlua{
local tenpt = 10 * 65536
local elem = node.new("rule")
elem.width = tenpt
elem.height = tenpt

% nuova lista
local head, last = node.insert_after(
    nil, nil, elem
)

for _ = 2,10 do
    local s = node.new("glue")
    s.spec = node.new("glue_spec")
    s.spec.width = tenpt
    head, last = node.insert_after(
        head, last, s
    )

    local q = node.new("rule")
    q.width = tenpt
    q.height = tenpt
    head, last = node.insert_after(
        head, last, q
    )
end

node.write(head)
}
\bye
```

La versione finale che darò del codice d'esempio è ancora più compatta. Essa fa uso della funzione `node.copy()`, che duplica il nodo argomento compresi gli eventuali sotto-nodi e fa a meno del riferimento all'ultimo nodo della lista[4] che in precedenza avevo chiamato `last`, inserendo ogni volta i nodi quadrato e spazio con ordine inverso subito dopo il nodo di testa:

```
\leavevmode\directlua{
local tenpt = 10 * 65536

% square node
local q = node.new("rule")
q.width = tenpt
q.height = tenpt

% space node
local s = node.new("glue")
s.spec = node.new("glue_spec")
s.spec.width = tenpt

% a new list
local head = node.insert_after(nil, nil, q)

for _ = 2,10 do
    head = node.insert_after(
        head, head, node.copy(q)
    )
    head = node.insert_after(
        head, head, node.copy(s)
    )
end

node.free(s)
node.write(head)
}
\bye
```

Se il numero dei quadrati fosse una potenza di 2 si potrebbe a ogni ciclo appendere alla lista una copia di se stessa tramite la funzione `node.copy_list()`. Lascio al lettore altri interessanti esperimenti dandogli appuntamento per la discussione sul forum del GUIT.

2.6 Nodi in memoria

Il sistema dei tipi di Lua è estensibile, ciò significa che esiste un meccanismo per aggiungere nuovi tipi chiamati *userdata* implementandoli in linguaggio C. Ebbene, i nodi sono oggetti userdata compatibili a quelli prodotti da TEX per le sue liste.

In Lua l'oggetto nodo è un riferimento, diremmo un puntatore, all'oggetto userdata in memoria. Una volta inserito all'interno delle liste svuotate dal Visual processor, il codice Lua non può più eliminarlo dalla memoria automaticamente tramite il Garbage Collector[5] perché potrebbe eliminare un oggetto non ancora processato. In altre parole, Lua non ha e non può avere il controllo sull'operato di TEX, almeno con la tecnologia sviluppata con l'attuale LuaTEX.

Per questo motivo i nodi sono stati esclusi dal processo automatico del Garbage Collector di Lua, perciò la gestione della memoria deve essere fatta manualmente dallo sviluppatore.

In definitiva, le regole da seguire si basano sul fatto che non è possibile per Lua conoscere lo stato

4. Se non siete a vostro agio con la semantica che Lua riserva alla corrispondenza degli argomenti di input e output delle funzioni consultate una guida o il libro Ierusalimschy (2013).

5. Il Garbage Collector è un processo automatico attivo in fase di esecuzione che libera la memoria non più indirizzata da almeno un riferimento. Questo succede per esempio quando assegnamo il valore `nil` all'unica variabile Lua che indirizza una tabella, perché da quel momento essa non può più essere utilizzata nel programma e può quindi essere eliminata.

di un oggetto in memoria che è stato ormai incluso nelle liste di TEX con `node.write()`, il ché significa che non possiamo sapere se l'oggetto è ancora utilizzato da TEX perché si trova nella lista in attesa di essere inviato al processo di visualizzazione, né sapere se TEX stesso ha già provveduto a liberare la memoria dell'oggetto perché ormai già processato.

Forse non vi sarà passata inosservata una delle ultime righe di codice dell'esempio precedente, quella dove viene chiamata la funzione `node.free()`. Farlo è corretto in questo caso perché quel nodo `s` è sempre e solo stato copiato, non fa parte di nessuna lista e non è mai stato inviato al motore di composizione.

2.7 Nodi `glyph`

Uno dei tipi di nodi che ci si aspetterebbe di trovare è quello che rappresenta i caratteri. Finalmente possiamo ora dedicarci al tipo chiamato `glyph`. Tra i campi dell'oggetto, quelli fondamentali sono due: `char` con la codifica del carattere e `font`.

Sapendo questo, proviamo a scrivere il codice Lua per stampare tutte le lettere maiuscole dell'alfabeto latino. Al solito il consiglio è di non proseguire nella lettura prima di aver provato in proprio con i suggerimenti di usare la codifica ASCII e la funzione `font.current()` per ottenere il font da usare. Ecco una possibile soluzione:

```
\leavevmode\directlua{
local alpha, last
for i=1,26 do
    local c = node.new("glyph")
    c.char = 64+i
    c.font = font.current()
    alpha, last = node.insert_after(
        alpha, last, c
    )
end

node.write(alpha)
}\bye
```

Sì, funziona, perché la codifica ASCII pur non essendo quella di LuaTEX che opera in Unicode, è inclusa nella trasformazione multi-byte UTF-8 di quest'ultima fino alla posizione numero 127.

A questo punto dovremo parlare di font, legature e tanti altri temi legati alla composizione del testo, ma a mio parere sono argomenti da affrontare solo dopo aver compreso il funzionamento di base dei nodi, che del resto si legano a tecniche avanzate come gli *attributi* e le funzioni di *callback*. Proseguiamo dunque con il prossimo tipo di nodo che è infatti fondamentale per le applicazioni tipografiche: la scatola.

2.8 Nodi scatole

Due funzioni della libreria `node` consentono di costruire scatole con al loro interno elementi impacchettati in orizzontale, funzione `hpack()`, o in verticale, funzione `vpack()`. Esse accettano una lista di nodi tramite il riferimento al nodo di testa e restituiscono il riferimento al nodo del box corrispondente.

È dunque indispensabile conoscere cosa sono i box in TEX, quali i dettagli dei modi di composizione, delle dimensioni e degli allineamenti; del resto questo è stato vero fin dall'inizio. Possiamo riassumere il tutto con la frase *non c'è Lua senza TEX*... almeno quando si parla di LuaTEX.

Altra conseguenza implicita dell'idea di contenitore è l'annidamento. Possiamo cioè inserire un box dentro l'altro impacchettando oggetti fino a raggiungere il risultato tipografico desiderato. Ne mostrerò esempi e applicazioni già nella prossima sezione.

Studiamo questo esempio:

```
\directlua{
local function glyph(c)
    local n = node.new("glyph")
    n.char = c
    n.font = font.current()
    return n
end

local function boxed(s)
    local h, l
    for c in string.utfvalues(s) do
        h, l = node.insert_after(
            h, l, glyph(c)
        )
    end
    return node.hpack(h)
end

local b1 = boxed "ABC"
local b2 = boxed "123456789"
local b3 = boxed "DEF"
b1.next = b2
b2.next = b3

local b = node.vpack(b1)
tex.box[0] = b
}
% stampo e svuoto il registro 0
\box0
\bye
```

un sorgente compilabile dove è prodotto un box verticale da una lista di nodi con tre scatole orizzontali contenenti a sua volta altrettante liste di glifi, con il seguente risultato:

ABC
123456789
DEF

Nel codice troviamo anche la modalità con cui si possono usare i registri box direttamente da TEX: è sufficiente assegnare a un registro il nodo iniziale della lista tramite indicizzazione diretta dell'array `tex.box`. Possiamo dunque usare i comandi TEX previsti per la composizione, come `\box` che stampa e libera il contenuto del registro dal numero che

lo segue — con a disposizione ben 65 536 registri — e `\copy` che stampa il contenuto ma lascia intatto il registro.

Il fatto che la tabella/array `box` della libreria `tex`, quindi un oggetto Lua, sia così intimamente legato ai registri TEX, come abbiamo appena visto, è un chiaro indice del funzionamento interno dei nodi.

È anche possibile utilizzare i registri allocati da TEX tramite una control sequence per evitare che un pacchetto collida con un altro nell'indirizzarli per numero. Si tratta di rispettare una convenzione dei nomi già in uso nei pacchetti LATEX — una sorta di *namespace* basato sul simbolo `@` — per rendere molto meno probabile la sovrapposizione di macro e registri.

Occorre prima allocare il registro con la primitiva `\newbox` e poi assegnarlo con la funzione `tex.setbox()` in Lua. Vediamone un esempio rimandando al manuale di LuaTEX per i dettagli della sintassi:

```
\newbox\mypackMybox

\directlua{
local function glyph(c)
    local n = node.new("glyph")
    n.char = c
    n.font = font.current()
    return n
end

local function boxed(s)
    local h, l
    for c in string.utfvalues(s) do
        h, l = node.insert_after(
            h, l, glyph(c)
        )
    end
    return node.hpack(h)
end

local b = boxed "ABC"
tex.setbox("mypackMybox", b)
}

\copy\mypackMybox
\copy\mypackMybox
\box\mypackMybox
\bye
```

Accanto alla funzione `setbox()` esiste anche `getbox()` che accetta l'indentificatore del registro box, sia esso il numero intero o la stringa della control sequence, per restituire il nodo interno al contenitore. Ciò lascia supporre, e ne lascio la verifica al lettore, che sia possibile modificare il contenuto di un registro creato in precedenza con le macro di TEX.

Prima di passare oltre consideriamo questo frammento di codice TEX:

```
\hbox to 64pt {A\hfil A}
```

Esso compone una scatola di larghezza fissata a 64 pt contenente una grandezza elastica il cui effetto è per così dire di *spingere* le due lettere verso l'esterno fino alle pareti del contenitore. Più è elevato il valore di stretching dello spazio elastico e maggiore sarà l'effetto. Possiamo fare lo stesso con i nodi in Lua?

La risposta è sì. Occorre fornire alla funzione `hpack()` non solo il nodo che rappresenta la lista, ma altri due argomenti che finora vi ho tenuto nascosti. Il primo è una dimensione e il secondo è una stringa che può essere `"exactly"` — nel qual caso la dimensione sarà interpretata come la lunghezza del box — oppure `"additional"`, con la lunghezza che sarà la capacità di allungarsi del box stesso.

Proviamo a ricreare gli stessi nodi prodotti da TEX con la breve riga del codice precedente:

```
\directlua{
% glyphs
local a = node.new "glyph"
a.char, a.font = 65, font.current()
local a1 = node.copy(a)

% skip
local skip = node.new "glue"
skip.spec = node.new "glue_spec"
skip.spec.stretch = tex.sp "60pt"

% list it together
a.next, skip.next = skip, a1

% packed it in a hbox
local box, b = node.hpack(
    a, tex.sp "64pt", "exactly"
)
tex.print("box badness:", b)
tex.box[0] = box
}

$|$\copy0$|$
$|$\copy0$|$

$|$\hbox to 64pt{A\hfil A}$|$
$|$\hbox to 64pt{A\hfil A}$|$
\bye
```

La verifica ha successo anche nel mostrare che la funzione `hpack()` fornisce un secondo parametro che misura la qualità della spaziatura del box chiamata *badness*, un intero compreso tra 0 e 10 000 — che nel nostro caso vale 54.

3 Applicazione barcode

Ci proponiamo di utilizzare la tecnologia dei nodi di LuaTEX per stampare il barcode di una stringa testuale secondo il formato Code 39. Si tratta di un'esercitazione interessante perché implica un certo sforzo di progettazione e che illustrerò passo passo nelle prossime sezioni cominciando al solito da una definizione generale.

Il *barcode* è un disegno di rettangoli pieni che non si sovrappongono costruito seguendo regole di codifica e decodifica di un testo e diffuse da enti o industrie.

Una scacchiera risponde ai requisiti grafici ma non è un barcode, non essendo conforme a nessuno dei formati noti e riconosciuti dai lettori ottici o a luce laser. Se lo fosse, al termine della scansione otterrei la sequenza dei caratteri rappresentati. Lo è invece la figura che segue, trattandosi del QR Code che scansionato con i vostri smartphone vi porta al sito del GUIT:

Il più semplice formato monodimensionale Code 39 Wikipedia risale al 1974 a opera del Dottor David Allais e di Ray Stevens allora alla Intermec, un produttore di scanner americano. Esso consiste in una sequenza di barre della stessa altezza e di spessore sottile o spesso che chiameremo rispettivamente `b` e `B`, distanziate l'una dall'altra da spazi sottili o spessi che chiameremo rispettivamente `w` e `W`. Un esempio è il seguente che codifica lo stesso testo del precedente barcode:

Questi strani disegni geometrici, ormai talmente diffusi da non attirare più la nostra attenzione, nascondono ragionamenti matematici piuttosto complessi allo scopo di rendere la scansione del codice più affidabile possibile. Con considerazioni come queste il Code 39 assegna a un insieme di 43 caratteri — 10 cifre numeriche, 26 lettere maiuscole e 7 segni d'interpunzione — un'univoca rappresentazione grafica con 5 barre distanziate da 4 spazi a due diversi spessori, con 3 elementi spessi e 6 sottili.

Un simbolo particolare a codifica asimmetrica, l'asterisco, apre e chiude la sequenza delle barre del testo codificato; lo si può verificare confrontando i caratteri stampati sotto al barcode precedente.

3.1 Il carattere A

L'idea è quella di stampare la rappresentazione della lettera A — codificata con la sequenza `BwbwbWbwB` — con la funzione Lua che chiameremo `bar_char()` che accetti la stringa di codifica del carattere e ne restituisca il primo nodo della lista dei 9 elementi `rule` e `glue`.

Ciò è ragionevole, ma come reperire le misure degli elementi? Le opzioni sono tre: inserire le misure nel corpo della funzione, negli argomenti o in una variabile globale. Per il momento scegliamo la soluzione più semplice, ovvero la prima:

```
\leavevmode\directlua{
local function bar_char(code)
    % 1D code39 barcode
    % bar definitions:
    % b = narrow black bar
    % B = wide black bar
    % w = narrow white bar
    % W = wide white bar
    % module = 0.25mm
    % height = 8mm
    local mod = 0.25 * 186468
    local b, w = mod, mod
    local B, W = 3*mod, 3*mod
    local height = 8 * 186468

    local head, last
    for c in string.gmatch(code, ".") do
        if c == "b" then
            local bar_b = node.new("rule")
            bar_b.width = b
            bar_b.height = height
            head, last = node.insert_after(
                head, last, bar_b
            )
        elseif c == "B" then
            local bar_B = node.new("rule")
            bar_B.width = B
            bar_B.height = height
            head, last = node.insert_after(
                head, last, bar_B
            )
        elseif c == "w" then
            local sp_w = node.new("glue")
            sp_w.spec=node.new("glue_spec")
            sp_w.spec.width = w
            head, last = node.insert_after(
                head, last,  sp_w
            )
        elseif c == "W" then
            local sp_W = node.new("glue")
            sp_W.spec=node.new("glue_spec")
            sp_W.spec.width = W
            head, last = node.insert_after(
                head, last, sp_W
            )
        end
    end

    return head
end

% the Code 39 'A' letter
node.write(bar_char("BwbwbWbwB"))
}
\bye
```

L'esecuzione di questo codice fornisce il disegno di figura 2 con cui possiamo verificare la correttezza di spessore e spaziatura. Il nucleo della funzione generatrice consiste in un'istruzione condizionale a più rami poiché Lua non dispone di un costrutto *switch*, ma in compenso lo si può migliorare in molti modi, per esempio sfruttando il fatto che le funzioni Lua sono tipi di prima classe.

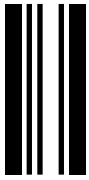

FIGURA 2: Rappresentazione della lettera A nel formato Code 39 ottenuta con il codice riportato nel testo a partire dalla stringa `'BwbwbWbwB'`. Non si tratta ancora di un barcode completo perché mancano i caratteri asterisco di inizio/fine.

Ciò rende possibile associare alle quattro chiavi di formato `b`, `B`, `w` e `W`, le funzioni che generano il nodo corrispondente:

```
local mod = 0.25 * 186468
local b, w = mod, mod
local B, W = 3*mod, 3*mod
local height = 8 * 186468

local Efactory = {
    b = function ()
        local bar = node.new("rule")
        bar.width = b
        bar.height = height
        return bar
    end,
    w = function ()
        local white = node.new("glue")
        white.spec = node.new("glue_spec")
        white.spec.width = w
        return white
    end,
    B = function ()
        local bar = node.new("rule")
        bar.width = B
        bar.height = height
        return bar
    end,
    W = function ()
        local white = node.new("glue")
        white.spec = node.new("glue_spec")
        white.spec.width = W
        return white
    end,
}
```

e riscrivere `bar_char()` così, indicizzando la tabella e lanciando la chiamata della funzione restituita con una coppia di parentesi tonde:

```
local function bar_char(code)
    local head, last
    for c in string.gmatch(code, ".") do
        head, last = node.insert_after(
            head, last, Efactory[c]()
        )
    end

    return head
end
```

Questo codice mostra anche un'altra caratteristica di Lua: le *closure*. Invece di entrare nei dettagli dirò solamente dove nel codice entrano in azione, cioè all'interno delle funzioni factory per la costruzione dei nodi, dove si può notare che le dimensioni sono contenute in variabili che si riferiscono al contesto locale.

3.2 Di nuovo il carattere A

Nei barcode, il testo codificato può essere riportato al di sotto del disegno a barre così da consentire di recuperare manualmente l'informazione in caso di errori di lettura. Ma come aggiungere alle nostre funzioni l'opzione di riportare il simbolo immediatamente al di sotto delle barre mantenendo gli allineamenti?

Una soluzione è quella d'inserire due box all'interno di un box verticale: la scatola che contiene la sequenza di barre/spazi che codifica la lettera A e la scatola che contiene il glifo del simbolo centrato grazie a due spazi elastici di larghezza uguale a quella naturale del box superiore.

Il codice completo che implementa l'idea è il seguente:

```
\directlua{
local module = 0.25 * 186468
local height = 8 * 186468
local bardim = {
    b = module,
    B = 3*module,
    w = module,
    W = 3*module,
}

local Efactory = {
    b = function ()
        local bar = node.new("rule")
        bar.width = bardim.b
        bar.height = height
        return bar
    end,
    w = function ()
        local white = node.new("glue")
        white.spec = node.new("glue_spec")
        white.spec.width = bardim.w
        return white
    end,
    B = function ()
        local bar = node.new("rule")
        bar.width = bardim.B
        bar.height = height
        return bar
    end,
    W = function ()
        local white = node.new("glue")
        white.spec = node.new("glue_spec")
        white.spec.width = bardim.W
        return white
    end,
}

local function bar_char(code)
    local len = 0
    local head, last
    for c in string.gmatch(code, ".") do
```

```
        len = len + bardim[c]
        head, last = node.insert_after(
            head, last, Efactory[c]()
        )
    end
    return head, len
end

% natural width hbox
local A_bars, lenA_bars = bar_char(
    "BwbwbWbwB"
)
local boxA_bars = node.hpack(A_bars)

% setting width hbox
local A_glyph = node.new("glyph")
A_glyph.char = 65
A_glyph.font = font.current()

local head_skip = node.new("glue")
head_skip.spec = node.new("glue_spec")
head_skip.spec.stretch = 6*module

local tail_skip = node.copy(head_skip)
head_skip = node.insert_after(
    head_skip, head_skip, A_glyph
)
head_skip = node.insert_after(
    head_skip, A_glyph, tail_skip
)
local boxA_glyph = node.hpack(
    head_skip, lenA_bars, "exactly"
)
boxA_glyph.height = 10 * 65536
boxA_bars = node.insert_after(
    boxA_bars, boxA_bars, boxA_glyph
)

% vertical hbox assembly
local mainbox = node.vpack(boxA_bars)

node.write(mainbox)
}
\bye
```

La funzione `bar_char()` adesso non restituisce più solamente il nodo della lista barre/spazi, ma anche la loro larghezza naturale che servirà per dimensionare il box sottostante sommando le larghezze dei singoli elementi.

Poiché in ogni codifica di simbolo ci sono sempre 6 spessori sottili e 3 spessi, questa dimensione è pari a $6b+3B$ solo nell'ipotesi che gli spessori delle barre coincidano con quelli degli spazi, perciò la funzione è si più lenta ma è valida più in generale.

L'ultima osservazione che faccio tra le molte altre ancora sviluppabili è che il codice di creazione e assemblaggio dei nodi è piuttosto verboso. Un livello d'astrazione maggiore lo renderebbe compatto e molto più leggibile, facendo però perdere allo sviluppatore la costante consapevolezza di stare manipolando oggetti tipografici a basso livello, per definizione un poco *delicati*.

3.3 Gestione degli errori e dati di formato

La flessibilità d'uso della tabella di Lua ci permette di risolvere in un colpo solo sia la gestione degli errori che la rappresentazione di formato del barcode. Nella validazione del dato in ingresso ci occorrerà infatti l'informazione di esistenza del simbolo Code 39, mentre nel disegnarlo ci servirà conoscerne la sequenza corretta dei nove elementi alternati di barre e spazi.

Poiché indicizzare una tabella Lua[6] restituirà `nil` se la chiave non esiste o al contrario il valore a essa associato, una tabella con le 43 coppie simbolo/rappresentazione contiene l'informazione necessaria e sufficiente sia a validare che a disegnare il barcode.

Un esempio di codice per quest'idea è il seguente:

```
local Code39Symbols = {
["0"] = "bwbWBwBwb",
["1"] = "BwbWbwbwB",
["2"] = "bwBWbwbwB",
-- ...
}

-- validazione del carattere "0"
local char = "0"
local ok = false
if Code39Symbols[char] then
    print("carattere code39 valido")
    ok = true
end

-- stampa codice carattere se ok
if ok then
    local code = Code39Symbols[char]
    bar_char(code)
end
```

3.4 Validazione *algebrica*

Anche se non rigorosamente, con la tabella di Lua possiamo rappresentare un tipo di dato chiamato *tipo algebrico*, una sorta di unione del tipo enumerazione con tipi dati. Esso è molto importante nei linguaggi a tipizzazione statica che non contemplano il tipo nullo come Haskell (Haskell) e Rust (Rust).

In particolare, vorremmo usare una funzione che controlli la validità di tutti i caratteri della stringa in ingresso sulla base della codifica Code 39 e, se la stringa è valida, restituisca il dato stesso altrimenti un opportuno messaggio d'errore:

```
function validateCode39(s)
   local chars = {}
   for c in string.gmatch(s, ".") do
```

6. La tabella di Lua implementa la struttura dati chiamata *dizionario* o anche *array associativo*, l'insieme di coppie chiavi/valori. In essa possono essere memorizzati dati all'indirizzo di chiavi univoche. La sintassi del costruttore di tabelle e l'iteratore `ipairs()` del linguaggio consentono di usare l'unica struttura dati complessa di Lua come se fosse un vettore.

```
        if not Code39Symbols[c] then
            local fmt = "Error: " ..
                "the char '%s' is not a "..
                "valid Code 39 symbol"
            local msg = string.format(fmt,c)
            % primo caso
            return {Err = msg}
        end
        chars[#chars+1] = c
    end
    % secondo caso
    return {Some=chars}
end
```

Chiamando questa funzione otterremo una tabella contenente o la chiave `Err`, a cui sarà associato il messaggio d'errore, oppure la chiave `Some`, a cui sarà associato il vettore dei caratteri del codice. Per conoscere in quale caso ci troviamo potremo scrivere:

```
local code = validateCode39("ABC")

if code.Err then
    error(code.Err)
else
    typesetCode39(code.Some)
end
```

Questa soluzione ha il vantaggio di essere semplice ed espressiva. Ovvio che in Lua non avremo il controllo sui tipi a tempo di compilazione e che potremo chiamare direttamente `typesetCode39()` con la tabella contenente simboli non validi, per esempio con `{Some = {"a", "b", "c"}}`, ma la trovo un'idea elegante e concorde nella filosofia di un linguaggio dinamico come Lua.

3.5 Il codice completo

Dalla discussione fatta finora basata su semplici esempi di codice funzionante, il nostro progetto assumerà la seguente struttura:

- una tabella Lua conterrà le informazioni di codifica Code 39 e le funzioni utente verranno caricate come libreria esterna;
- una funzione pubblica validerà il testo in ingresso da tradurre in barcode;
- una funzione pubblica comporrà il disegno del barcode accettando solamente stringhe di testo già validate e un secondo argomento di opzioni.

Le opzioni previste per il disegno del barcode saranno solamente due: l'altezza del codice a barre — che dovrebbe essere proporzionale alla sua lunghezza — e la stampa dei caratteri codificati.

Il prossimo listato Lua è la libreria del progetto ottenuta assemblando i risultati descritti nelle sezioni precedenti. In un file esterno vengono definite funzioni e dati locali usati dall'interfaccia pubblica attraverso le closure:

```
-- libreria d'esempio per la composizione
-- di barcode base Code39

local lib = {}

local Dim = {
    mod = 0.25 * 186468, -- sp
    height = 8 * 186468, -- sp
}
Dim.b, Dim.w = Dim.mod, Dim.mod
Dim.s = Dim.mod
Dim.B, Dim.W = 3*Dim.mod, 3*Dim.mod

local Code39Symbols = {
["0"]={"b","w","b","W","B","w","B","w","b"},
["1"]={"B","w","b","W","b","w","b","w","B"},
["2"]={"b","w","B","W","b","w","b","w","B"},
["3"]={"B","w","B","W","b","w","b","w","b"},
["4"]={"b","w","b","W","B","w","b","w","B"},
["5"]={"B","w","b","W","B","w","b","w","b"},
["6"]={"b","w","B","W","B","w","b","w","b"},
["7"]={"b","w","b","W","b","w","B","w","B"},
["8"]={"B","w","b","W","b","w","B","w","b"},
["9"]={"b","w","B","W","b","w","B","w","b"},
["A"]={"B","w","b","w","b","W","b","w","B"},
["B"]={"b","w","B","w","b","W","b","w","B"},
["C"]={"B","w","B","w","b","W","b","w","b"},
["D"]={"b","w","b","w","B","W","b","w","B"},
["E"]={"B","w","b","w","B","W","b","w","b"},
["F"]={"b","w","B","w","B","W","b","w","b"},
["G"]={"b","w","b","w","b","W","B","w","B"},
["H"]={"B","w","b","w","b","W","B","w","b"},
["I"]={"b","w","B","w","b","W","B","w","b"},
["J"]={"b","w","b","w","B","W","B","w","b"},
["K"]={"B","w","b","w","b","w","b","W","B"},
["L"]={"b","w","B","w","b","w","b","W","B"},
["M"]={"B","w","B","w","b","w","b","W","b"},
["N"]={"b","w","b","w","B","w","b","W","B"},
["O"]={"B","w","b","w","B","w","b","W","b"},
["P"]={"b","w","B","w","B","w","b","W","b"},
["Q"]={"b","w","b","w","b","w","B","W","B"},
["R"]={"B","w","b","w","b","w","B","W","b"},
["S"]={"b","w","B","w","b","w","B","W","b"},
["T"]={"b","w","b","w","B","w","B","W","b"},
["U"]={"B","W","b","w","b","w","b","w","B"},
["V"]={"b","W","B","w","b","w","b","w","B"},
["W"]={"B","W","B","w","b","w","b","w","b"},
["X"]={"b","W","b","w","B","w","b","w","B"},
["Y"]={"B","W","b","w","B","w","b","w","b"},
["Z"]={"b","W","B","w","B","w","b","w","b"},
["-"]={"b","W","b","w","b","w","B","w","B"},
["."]={"B","W","b","w","b","w","B","w","b"},
[" "]={"b","W","B","w","b","w","B","w","b"},
["$"]={"b","W","b","W","b","W","b","w","b"},
["/"]={"b","W","b","W","b","w","b","W","b"},
["+"]={"b","W","b","w","b","W","b","W","b"},
["%"]={"b","w","b","W","b","W","b","W","b"},
}
local Code39Star = {
       "b","W","b","w","B","w","B","w","b"
}

local Efactory = {
    b = function ()
        local bar = node.new("rule")
```

```
        bar.width = Dim.b
        bar.height = Dim.height
        return bar
    end,
    w = function ()
        local white = node.new("glue")
        white.spec = node.new("glue_spec")
        white.spec.width = Dim.w
        return white
    end,
    B = function ()
        local bar = node.new("rule")
        bar.width = Dim.B
        bar.height = Dim.height
        return bar
    end,
    W = function ()
        local white = node.new("glue")
        white.spec = node.new("glue_spec")
        white.spec.width = Dim.W
        return white
    end,
    s = function ()
        local white = node.new("glue")
        white.spec = node.new("glue_spec")
        white.spec.width = Dim.s
        return white
    end,
}

local function bar_char(tchars, head, last)
    for _, e in ipairs(tchars) do
        head, last = node.insert_after(
            head, last, Efactory[e]()
        )
    end
    return head, last
end

local function hboxbars(tchars)
    local head, last = bar_char(Code39Star)
    for _, c in ipairs(tchars) do
        head, last = bar_char(
            {'s'}, head, last
        )
        head, last = bar_char(
            Code39Symbols[c], head, last
        )
    end
    head, last = bar_char({'s'}, head, last)
    head, last = bar_char(
        Code39Star, head, last
    )
    return node.hpack(head)
end

local function barsdim(tcode)
    local dim = 0
    for _, e in ipairs(tcode) do
        dim = dim + Dim[e]
    end
    return dim
end

local function hboxsymb(tchars)
    -- empty space under '*'
    local head = node.new("glue")
    head.spec = node.new("glue_spec")
    head.spec.width = barsdim(Code39Star)

    local last = head
    for _, c in ipairs(tchars) do
       local w = barsdim(Code39Symbols[c])
       local s0 = Efactory.s()
       local g1 = node.new("glue")
       g1.spec = node.new("glue_spec")
       g1.spec.stretch = w/2
       local g2 = node.copy(g1)
       local glyph = node.new("glyph")
       glyph.char = string.byte(c)
       glyph.font = font.current()
       g1 = node.insert_after(g1, g1, g2)
       g1 = node.insert_after(g1, g1, glyph)
       local hb=node.hpack(g1, w, "exactly")
       head, last = node.insert_after(
           head, last, s0
       )
       head, last = node.insert_after(
           head, last, hb
       )
    end
    local box = node.hpack(head)
    box.height = tex.sp "10pt"
    return box
end


-- public function

function lib.validate(s)
    local chars = {}
    for c in string.gmatch(s, ".") do
        if not Code39Symbols[c] then
            local fmt = "Error: " ..
            "the char '%s' is not a valid"..
            " Code 39 symbol"
            local msg = string.format(fmt,c)
            return {Err = msg}
        end
        chars[#chars+1] = c
    end
    return {Some=chars}
end

function lib.typeset(tcode, topt)
    if tcode.Err then
        error(tcode.Err)
    end
    local symb = false
    local olddim
    if topt then
        -- optional barcode height
        if topt.h then
            olddim = Dim.height
            Dim.height = tex.sp(topt.h)
        end
        -- optional symbol ref
        if topt.symb then
            symb = true
        end
```

```
    end

    local barbox = hboxbars(tcode.Some)
    if symb then
        local symbbox = hboxsymb(tcode.Some)
        barbox = node.insert_after(
            barbox, barbox, symbbox
        )
    end
    local vbox = node.vpack(barbox)
    local res = node.hpack(vbox)
    node.write(res)

    if olddim then
        Dim.height = olddim
    end
end

return lib
```

3.6 Test

È sempre fondamentale verificare la correttezza del programma con una batteria di test il più completa e automatica possibile. Nel caso della libreria LuaTEX, per stampare barcode secondo il formato Code 39, i test sono stati di due tipi: controllo a schermo del disegno del barcode e verifica fisica mediante lettore laser di una serie di barcode stampati su carta.

Questi test hanno dato esito positivo. In particolare il lettore laser, una volta connesso su una porta USB del computer con funzionamento in emulazione di tastiera, ha fornito i codici rappresentati senza errori, come è possibile verificare con il barcode di figura 3.

Esistono tuttavia altri tipi di verifiche, come l'efficacia di utilizzo della sintassi dei comandi e l'efficienza di esecuzione misurata nei tempi in secondi di compilazioni di sorgenti prestabiliti su una macchina di riferimento, nell'intento di valutare grandezze relative e confrontare implementazioni differenti della stessa libreria.

Per il primo aspetto presento il codice compilabile di un sorgente minimo d'esempio. Per farlo funzionare occorre salvare nel file di nome `lib-code39.lua` il codice del listato precedente, nella stessa cartella su disco del file sorgente di prova seguente:

```
% caricamento libreria
\directlua{
code39 = require("lib-code39.lua")
}
```

Figura 3: Il codice a barre con l'indirizzo del sito web dell'associazione GUIT, creato con il codice LuaTEX illustrato nel testo.

```
Test Code 39

% barcode with default option
\directlua{
code = code39.validate("WWW.GUITEX.ORG")
code39.typeset(code)
}

% barcode with specified height and symbol
\directlua{
code39.typeset(code,{h="50pt",symb=true})
}

% local barcode object
\directlua{
local c = code39.validate("ABC123")
code39.typeset(c ,{h='12pt',symb=true})
}
\bye
```

Per il secondo aspetto invece ho pensato di realizzare un documento contenente 10 000 barcode degli stessi dodici caratteri. A ogni esecuzione, la funzione di disegno genera da capo tutta la sequenza di nodi necessaria, perciò si tratta di una prova che sollecita in particolare la generazione dei nodi di LuaTEX senza impegnare invece l'interruzione di riga. Il sorgente di prova è il seguente:

```
% loading library
\directlua{
c39 = require("lib-code39.lua")
}

\leavevmode\directlua{
local code = "ABC123456789"
local c = c39.validateCode39(code)
local opt = {h='12pt',symb=true}

% hugly
for i = 1,10000 do
    c39.typesetCode39(c, opt)
end
}
\bye
```

L'esecuzione è portata a termine in circa 4 s con LuaTEX 0.80, il ché vorrebbe dire circa 2500 barcode per secondo, ma è un dato assoluto che dovrebbe invece essere solamente confrontato con altri ottenuti sulla stessa macchina.

4 Conclusioni

Esplorare la libreria interna di LuaTEX dedicata ai nodi è un ottimo modo per avvicinarsi a questo nuovo motore di composizione che promette di ampliare notevolmente le applicazioni da cui trarre vantaggio per comporre documenti di qualità.

Il linguaggio Lua da usare per programmare i nodi si dimostra un ottimo strumento perché con una sintassi essenziale offre un buon numero di concetti, compreso quello relativo alle funzioni di prima classe del paradigma funzionale, le closure,

il tipo di dati tabella molto generale, efficiente e con un costruttore dalla sintassi intuitiva davvero potente.

Internamente a LuaTEX i nodi sono stati implementati come tipi *userdata* — cioè tipi previsti da Lua per estendere il linguaggio usando codice scritto in C — ma essi non sono gestiti in memoria nel modo usuale e automatico, ovvero con il Garbage Collector, un processo che elimina i dati non più utilizzabili in esecuzione. Questo perché non potremmo evitare che il Garbage Collector possa cancellare un nodo non ancora processato da TEX, che opera in un ambito parallelo prelevando dati dalle proprie liste principali.

Nel programmare con i nodi e con le altre librerie dovremo quindi fare molta attenzione per gestire la memoria correttamente, pena il crash del processo. Altri pericoli vengono dagli errori sulla gestione della lista tipografica TEX in attesa di processare gli elementi da noi prodotti con Lua. Spetta allo sviluppatore Lua mantenere in uno stato coerente le liste principali del compositore.

Nell'esempio applicativo abbiamo incontrato la necessità di *progettare* la struttura del codice valutando pro e contro delle alternative possibili e mostrando alcune tecniche utili con Lua. Le considerazioni d'efficienza d'esecuzione, d'interazione con le altri componenti software, di correttezza d'uso della memoria e la capacità del programma di sviluppo futuro del codice assumono importanza maggiore rispetto alla programmazione TEX/LATEX, richiedendo agli sviluppatori qualche dote in più nel campo dell'ingegneria del software proprio per coglierne tutte le potenzialità.

Rimane ancora molto da esplorare, e tutto lascia intuire applicazioni tipografiche molto difficili se non proprio impossibili da realizzare con altri software, anche se con LuaTEX si dovrà procedere con cautela.

5 Ringraziamenti

È il momento finale dei ringraziamenti che vanno a Luigi Scarso, che mi ha aiutato — praticamente in tempo reale — spiegandomi i dettagli spesso intricati dei nodi di LuaTEX e durante tutto lo sviluppo di questo articolo introduttivo. Tuttavia, gli errori che comunque spero di non aver commesso, sono da imputare solamente a me.

Ringrazio Claudio Beccari per le sue informazioni TEXniche su quando il compositore passa dal modo verticale a quello orizzontale — colmando un'evidente mia lacuna — e su alcune finezze del font design.

Grazie al team di LuaTEX per avermi concesso l'uso del logo del progetto su interessamento di Luigi Scarso — sempre in tempo reale ovviamente.

Ringrazio infine la redazione di ArsTEXnica per l'accurato lavoro di revisione e il lettore che ha svolto con passione gli *esercizi* proposti nel testo.

Riferimenti bibliografici

Giacomelli, R. e Pignalberi, G. (2015). «Generare documenti LATEX con diversi linguaggi di programmazione». *ArsTEXnica*, (20), pp. 40–73. URL http://www.guitex.org/home/it/numero-20-ottobre-2015.

Haskell (2016). «Official web site». https://www.haskell.org/.

Ierusalimschy, R. (2013). *Programming in Lua, Third Edition*. Lua.Org.

Rust (2016). «Official web site». https://www.rust-lang.org/.

The LuaTEX development team (2016). *LuaTEX Reference Manual*, 0.80.0 edizione.

Wikipedia (2016). «Code 39». https://en.wikipedia.org/wiki/Code_39.

▷ Roberto Giacomelli
Carrara
giaconet dot mailbox at gmail dot com