Parsing di opzioni in LuaTEX

Roberto Giacomelli

Sommario

In questo lavoro esploreremo come implementare un parser in Lua, il linguaggio di programmazione elegante e facile da imparare incluso in LuaTEX, per una lista di opzioni nel formato ⟨*chiave*⟩=⟨*valore*⟩.

Un simile componente è utile nello sviluppo di pacchetti, perché fornisce un sistema di opzioni progettato per l'efficienza e l'espressività della sintassi.

Abstract

This paper explains how to implement a parser in Lua—the elegant and easy to use programming language included in LuaTEX—for a list of options in the ⟨*key*⟩=⟨*value*⟩ format.

A similar parser is useful in the package development allowing an option system specifically designed for efficiency and syntax expressiveness.

1 Motivazione

Lentamente sto acquisendo una maggiore consapevolezza delle potenzialità del motore di composizione LuaTEX rispetto al tradizionale pdfTEX. Ho scoperto dapprima la tecnologia dei nodi, ovvero tutte quelle funzionalità che fanno capo alla libreria `node` e che consentono di costruire oggetti tipografici nativi programmando in Lua. Ho poi approfondito le questioni, se vogliamo più tecnologiche, del caricamento di librerie esterne non specifiche del sistema TEX e capaci di leggere le informazioni presenti in database anche di livello enterprise, ovvero ciò che rende l'utente in grado di costruire splendidi report preparando un sorgente TEX.

Ogni volta ne ho dato conto con un articolo su ArsTEXnica, la prestigiosa rivista del Gruppo Utilizzatori Italiani di TEX, sia per condividere e ricambiare contributi ricevuti, sia per mettere nero su bianco le cose a mia migliore comprensione e a futuro riferimento.

Oggi, sto gradualmente spostando l'attenzione verso lo sviluppo applicativo, cercando di dare concretezza a un progetto a medio termine. L'idea è impegnativa, ma di grande soddisfazione: creare funzioni avanzate in LuaTEX che anche gli altri utenti possano utilizzare componendo i loro documenti nel formato LATEX o ConTEXt.

Alla pagina web https://github.com/robitex/barracuda trovate la casa virtuale del progetto, che ho chiamato `Barracuda` per assonanza con il termine *barcode* e che implementa il disegno di alcune di questi simboli, che compaiono praticamente sull'etichetta di ogni prodotto.

Uno dei componenti del progetto sarà una classe di documento chiamata `labelreport` per la generazione di etichette, un elemento tipografico molto particolare perché comprende sia la grafica che la gestione dati.

In questo articolo non mi soffermerò sui dettagli del codice, quasi tutto scritto in Lua, ma illustrerò come poter risolvere proprio con questo linguaggio un problema molto importante, perché determina la qualità dell'esperienza utente in termini di semplicità di configurazione tipografica: l'insieme di opzioni ⟨*chiave*⟩=⟨*valore*⟩.

Lo scopo del presente breve articolo sarà quello di illustrare lo sviluppo in Lua di un modulo per la valutazione di questi elenchi di opzioni. Proveremo a sperimentare il setup di opzioni di un comando come il seguente, analogamente a quanto già è in grado di fare il pacchetto per LATEX `xkeyval`, per la definizione geometrica della pagina di etichette:

```
\setupGrid{
    xsep = 12mm,
    ysep = 5mm,
    showgrid,
    name = label21,
    shape = rect(68mm, 38mm)
}
```

Molti pacchetti offrono agli utenti una sintassi di configurazione simile, per esempio `siunitx` per la corretta composizione di numeri e unità di misura del Sistema Internazionale, o `pgf-tikz`, notissimo per la produzione della grafica, solo per citarne un paio. A volte gli Autori implementano il codice di gestione delle opzioni per proprio conto, a volte invece contano sulle librerie LATEX3.

Queste ultime offrono la soluzione migliore perché se tutti usassero L3 per fornire funzionalità di configurazione delle macro dei propri pacchetti, gli utenti si troverebbero a utilizzare un'interfaccia consistente e già nota. Tuttavia, è anche probabile che chi scrive pacchetti in Lua per LuaTEX desideri ricevere i parametri utente come variabili Lua.

La facilità di scrivere codice in Lua rispetto a TEX determina comunque un vantaggio importante per una libreria che dovesse fornire funzioni di lettura di sistemi di opzioni, in particolare perché è più semplice implementare nuove idee, a tutto vantaggio degli utenti.

Cercherò di illustrare alcune idee in Lua con l'obiettivo sperimentale di valutare l'impegno neces-

sario per implementare una libreria per la lettura di valori di configurazione utente secondo una sintassi ⟨*chiave*⟩=⟨*valore*⟩, nonché valutarne la semplicità d'uso dal punto di vista di chi scrive pacchetti.

1.1 Note e requisiti

Per la comprensione del codice riportato nelle prossime sezioni è preferibile che il lettore abbia un minimo di dimestichezza col linguaggio Lua, che si acquista senza particolari difficoltà grazie alla sua semplicità.

Troverete inoltre spesso la macro `\directlua`. Si tratta di una primitiva espandibile di LuaTEX che invoca l'interprete Lua con il codice rappresentato dal proprio argomento. Alcuni articoli apparsi su questa rivista, inoltre, contengono l'illustrazione di base delle caratteristiche di Lua e del tipo di dato tabella, l'unico predefinito a essere strutturato e che implementa allo stesso tempo un array e un dizionario.[1]

Saranno quindi utili riferimenti sia il testo ufficiale su Lua (Ierusalimschy, 2016), sia il manuale di LuaTEX (The LuaTEX development team, 2018). Rimando tuttavia al forum GUIT per le domande che eventualmente fossero ancora rimaste senza risposta.

2 Sviluppo

La sintassi prevista per un'opzione consiste nello specificare un nome seguito dal segno di uguale, seguito a sua volta dal valore. Ogni coppia ⟨*chiave*⟩=⟨*valore*⟩ deve essere separata da una virgola.

Non c'è alcuna necessità di implementare codice di parsing per un tale elenco di opzioni, poiché Lua stesso ne implementa già uno: il costruttore di tabelle. All'ipotetico comando della sezione precedente `\setupGrid` potremo passare opzioni racchiudendo i valori di tipo stringa tra doppi apici e assegnando alle alternative valori del tipo vero (`true`) o falso `false`:

```
\setupGrid{
    xsep = 12,
    ysep = 5,
    showgrid = true,
    name = "label21",
    shape = {rect = {68, 38}}
}
```

Per catturare queste opzioni è sufficiente il codice di questo sorgente minimo:

```
% !TeX program = LuaTeX
\def\setupGrid#1{\directlua{
    local t = {#1}
    % print some options
    local par = string.char(92).."par"
    tex.print("xsep = " .. t.xsep)
    tex.print(par)
    tex.print("name = " .. t.name)
    tex.print(par)
    tex.print("shape.rect[1] = " ..
    t.shape.rect[1])
}}
\setupGrid{
    xsep = 12,
    ysep = 5,
    showgrid = true,
    name = "label21",
    shape = {rect = {68, 38}}
}
\bye
```

Tuttavia, l'utente LaTEX non è abituato a racchiudere testo tra doppi apici; non si possono specificare unità di misura per le dimensioni né esprimere enumerazioni come quella dell'opzione `shape`, con la sintassi del costruttore di tabelle Lua.

Inoltre, definendo particolari sintassi per le opzioni potremo rendere più intuitivo il sistema, se solo fosse disponibile una funzione di parsing corrispondente.

2.1 Un parser byte per byte

Un *parser* è una funzione che ricava dati strutturati dalla lettura di un testo. Per implementarne uno possiamo usare apposite librerie, per esempio LPEG[2] inclusa in LuaTEX oppure affidarci alle espressioni regolari.

Preferisco, tuttavia, usare un algoritmo *byte per byte*, che consiste nel leggere un carattere alla volta dall'input e dedurne i dati, esercitando così un controllo minuzioso sulla sintassi e senza dover utilizzare le funzioni di LPEG, per me poco intuitive. In un secondo momento, poi, è sempre possibile definire grammatiche con questa libreria per un codice più veloce.

Per l'opzione semplice dove esiste la chiave e il valore è un semplice testo, la funzione potrebbe essere:

```
local function parseOption(s) --> key, val
    local isKeyParse = true -- state: key
    local key, val
    for c in s:gmatch(".") do
        if isKeyParse then -- read a key
            if c ~= " " then
                if c == "=" then
                    isKeyParse = false
                else
                    key = (key or "") .. c
                end
            end
        else -- read a value
            val = (val or "") .. c
        end
    end
    return key, val
end

print(parseOption "xsep = 12mm")
```

1. La struttura dati dizionario è detta anche 'array associativo'.

2. http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html.

La funzione utilizza un iteratore sul singolo carattere della stringa in ingresso e il valore booleano di stato `isKeyParse` è vero se stiamo leggendo il nome dell'opzione, falso se invece ne stiamo leggendo il valore. Nel primo caso, il carattere viene concatenato alla stringa `key`, altrimenti alla stringa `val`.

In particolare, quando incontriamo il carattere di uguale (`=`) lo stato `isKeyParse` è impostato su `false`, mentre ignoriamo completamente i caratteri spazio solo se stiamo leggendo il nome dell'opzione.

Passiamo ora alla lista di opzioni modificando il codice in questo modo, inserendo il codice Lua in una definizione di comando tramite la primitiva ormai nota `\directlua`:

```
% !TeX program = LuaTeX
\long\def\setupGrid#1{\directlua{
local function parseOption(s) % -> topt, err
    local isKeyParse = true % state: key parsing
    local topt = {}
    local i = 0 % state: option list number
    for c in s:gmatch(”.”) do
        if c == ”,” then
            if i == 0 then
                return nil, ”Empty first option”
            end
            i = i + 1 % new option
            topt[i] = {}
            isKeyParse = true
        elseif c == ”=” then
            isKeyParse = false
        else
            if isKeyParse then % read a key
                if i == 0 then
                    i = 1
                    topt[i] = {}
                end
                if not (c == ” ”) then
                    local t = topt[i]
                    t.key = (t.key or ””) .. c
                end
            else % read a value
                local t = topt[i]
                t.val = (t.val or ””) .. c
            end
        end
    end
    return topt
end
local s = [===[#1]===]
local topt, err = parseOption(s)
if err then error(err) end
for _, t in ipairs(topt) do
    tex.print(t.key..” = ”..t.val..”, ”)
end
}}

\setupGrid{
    xsep = 12mm,
    ysep = 5mm,
    showgrid = true,
    name = label21,
    shape = rect(68mm 38mm)
}
\bye
```

Si tratta ancora di codice grezzo, per esempio non possiamo usare la virgola per separare i vari valori associati all'enumerazione `rect` in `shape`, ma stampa correttamente i nomi e i valori corrispondenti delle opzioni. Se si compila il sorgente con LuaTEX, lo si può verificare.

2.2 Definizione di opzioni

A questo punto diventa chiaro che abbiamo la necessità di associare ulteriori informazioni al sistema delle opzioni, per esempio, per far sì che il parser possa segnalare nomi di opzioni non valide, o, cosa più importante, implementare sintassi più intuitive.

Con una tabella Lua è possibile descrivere la struttura di un'opzione in modo semplice e completo. Al campo `default` potremo far corrispondere il valore da assegnare all'opzione se l'utente non lo esplicita, e interpretarne l'assenza con la sua obbligatorietà.

Il campo `optype` potrebbe definire il tipo del valore associato all'opzione, per esempio una dimensione, e il campo `fncheck` potrebbe contenere una funzione Lua da applicare al valore specificato dall'utente per il controllo di validità e prima della memorizzazione.

Per l'opzione `xsep` la descrizione precedente si concretizza nel codice:

```
xsep = {
    default = 0.0,
    optype = ”dim”,
    fncheck = function (v)
        return v >= 0
    end,
}
```

Per le enumerazioni, che definirò compiutamente nella prossima sezione, la struttura di definizione potrebbe essere la seguente:

```
shape = {
    optype = ”enum”,
    enum = {
        rect = {
            {arg = ”x”, optype = ”dim”},
            {arg = ”y”, optype = ”dim”},
        },
        roundrect = {
            {arg = ”x”, optype = ”dim”},
            {arg = ”y”, optype = ”dim”},
            {arg = ”r”, optype = ”dim”},
        },
        circle = {
            {arg = ”d”, optype = ”dim”},
        },
    },
}
```

Si noti come in Lua sia possibile assegnare a campi di tabella valori che corrispondono a funzioni

per mezzo della sintassi anonima. Anzi, in Lua le funzioni sono valori di prima classe, perciò non hanno un nome, poiché la sintassi tradizionale è convertita dall'interprete in sintassi anonima per far sì che le definizioni seguenti siano equivalenti:

```
function nome (arg)
    -- body
end

nome = function (arg)
    -- body
end
```

Una volta ottenuto il nome dell'opzione leggendola dalla stringa di input inserita dall'utente, il parser può ottenerne la corrispondente definizione e proseguire con la lettura del valore con la specifica funzione.

2.3 Interazione tra opzioni

Possiamo ora comprendere più chiaramente quello che abbiamo chiamato il 'sistema delle opzioni', composto dai seguenti tre distinti componenti:

1. definizione delle opzioni in base al tipo;
2. parsing del testo con validazione e ritorno dei valori associati;
3. memorizzazione dei valori.

Non rimane quindi che soffermarci sulla terza fase, quella della memorizzazione. Ottenuti i parametri, è compito del pacchetto applicativo rielaborarli in una fase successiva e indipendente da quella del parsing, acquisendoli all'interno del proprio stato.

Questo, oltre alla memorizzazione, può comprendere anche una rielaborazione, per esempio per tenere conto di *interazioni* tra parametri. In una prima fase di implementazione del parsing (l'interazione tra parametri) può essere vantaggiosamente demandata al pacchetto applicativo per mantenere un contesto più semplice. In seguito, potrebbe essere inclusa nel parsing stesso attraverso funzioni da aggiungere alla definizioni delle opzioni.

Un esempio d'interazione tra opzioni è dato dal voler includere l'unità di misura delle dimensioni in modo che l'utente possa omettere quest'informazione nel caso che le misure da inseririre fossero molte. L'ulteriore opzione `unit`, disponibile per esempio in pstricks, tra i parametri della griglia consentirebbe di scrivere il seguente codice utente:

```
\setupGrid{
    unit = mm,
    xsep = 12,
    ysep = 5,
    shape = rect(68, 38)
}
\bye
```

L'interazione potrebbe anche essere più complessa, per esempio quando il valore dato dall'utente a un parametro, che chiameremo principale, determina la validità o meno di quello assegnato a un secondo parametro che chiameremo secondario. Questa dipendenza può essere implementata senza alcun problema in Lua modificando le funzioni `fncheck` nelle definizioni delle opzioni e aggiungendo un campo intero che determini l'*ordine* di chiamata delle stesse funzioni di verifica.

In questo modo, il sistema di opzioni passa da un insieme di parametri indipendenti a un grafo di nodi dipendenti che deve essere risolvibile, ovvero deve esistere un percorso di visita del grafo stesso che connetta tutti i parametri, da quelli principali a quelli secondari.

L'ordine di verifica stabilisce a tutti gli effetti la risoluzione del grafo di opzioni solamente nella procedura di verifica.

2.4 Meta opzioni

Ulteriore idea è quella di consentire all'utente di influenzare il parser di opzioni con specifici parametri. Diventerebbe possibile per esempio specificare il carattere che funge da separatore per le coppie ⟨*chiave*⟩=⟨*valore*⟩, indicare come considerare i caratteri spazio per il valore o applicare un eventuale arrotondamento ai valori numerici.

Questo tipo di regolazione influenza la sintassi con cui il parser legge le opzioni, nella maggioranza dei casi per evitare i casi in cui i caratteri di separazione confliggono con i valori.

Rimane da valutare con ulteriori approfondimenti se queste metaopzioni debbano essere impostate dal pacchetto oppure dall'utente. Infatti, è sempre possibile pensare che il pacchetto costruisca delle opzioni utente che in realtà corrispondono dietro le quinte a una o più metaopzioni per il parser.

3 Sintassi ⟨*chiave*⟩=⟨*valore*⟩

Sono previsti dal sistema diversi tipi di opzioni che l'utente dovrà conoscere e lo sviluppatore implementare. Nelle prossime sezioni proverò a definirne alcuni.

Una regola generale prevede che una data opzione debba obbligatoriamente essere valorizzata dall'utente, a meno che nella definizione non ne esista un valore di default. Quanto al carattere di separazione delle coppie ⟨*chiave*⟩=⟨*valore*⟩, la virgola, si ammette che dopo l'ultima opzione possa essercene una facoltativa.

3.1 Chiavi

L'elemento sintattico delle chiavi è definito semplicemente come un nome, inteso come una o più parole separate da spazi. Dei caratteri spazio possono essere inseriti sia prima che dopo la sequenza di parole, mentre tra le parole possono essere inseriti spazi a piacere, che saranno comunque considerati come uno solo.

Con questa definizione ammettiamo che le chiavi possano contenere spazi, come accade per le opzioni del pacchetto tikz o pgfplots. Così le seguenti sono tutte chiavi valide e diverse una dall'altra:

```
show grid
showgrid
show-grid
show_grid
ShowGrid
SHOWgrid
```

mentre le seguenti rappresentano sempre la stessa chiave:

```
show grid
   show grid
show    grid
```

Nell'ipotetico parser che legge i caratteri uno dopo l'altro, introduciamo delle funzioni Lua specializzate per leggere un dato elemento sintattico, facendo sì che il codice sia ben strutturato. Queste funzioni avranno sempre la stessa firma: gli argomenti saranno l'array con i caratteri da esaminare e l'indice di partenza, mentre il risultato restituito sarà la coppia di valori della posizione raggiunta dell'indice e il valore atteso.

È da notare che, essendo le tabelle Lua degli oggetti, le variabili ne contengono un riferimento, perciò passare tabelle a una funzione è efficiente e quindi lo è anche la soluzione di leggere l'array di caratteri anziché direttamente la stringa.

La funzione che legge dall'array una chiave di opzione in base alle regole stabilite può essere la seguente:

```
local function parseKey(t, i) --> i, key
    local k = {}
    local isInnerChar = false
    local isLastChar = true
    while true do
        if i > #t then
            break
        end
        local c = t[i]
        if c == ”,” or c == ”=” then
            break
        end
        if c == ” ” then
            if isInnerChar and isLastChar then
                 isLastChar = false
                 k[#k + 1] = c
            end
        else
            isLastChar = true
            isInnerChar = true
            k[#k + 1] = c
        end
        i = i + 1
    end
    local key
    local len = #k
    if len > 0 then
        if not isLastChar then
            k[len] = nil
        end
        key = table.concat(k)
    end
    return i, key
end
```

L'indice finale sarà la posizione del carattere da cui occorrerà leggere il valore dell'opzione a seconda del tipo corrispondente.

3.2 Opzioni booleane

L'opzione più semplice è quella che assume un valore vero oppure falso, acceso o spento. Seguendo il nostro sistema di esempio per la definizione della geometria delle etichette, stiamo parlando per esempio dell'opzione `showgrid`. Essa indica se la stampa del disegno della griglia regolare di etichette — che corrisponde ai loro contorni — debba o meno essere eseguita.

La sintassi base prevede che l'utente assegni il valore `true` o `false` all'opzione oppure che ne indichi solamente la chiave per attivarla. In altre parole, assegnare il valore `true` è opzionale.

Una funzione di lettura del valore è la seguente:

```
local function parseBool(t, i) --> index, bool
    local c = t[i]
    local len = #t
    if i > len or c == ”,” then
        return i, true
    end
    if c == ”=” then
        i = i + 1
        while i<= len do
            local b = t[i]
            if b ~= ” ” then
                break
            end
            i = i + 1
        end
        local res
        local concat = table.concat
        if i + 4 <= len and
            concat(t, ””, i, i+4) == ”false”
            then
            i = i + 5
            res = false
        elseif i + 3 <= len and
            concat(t, ””, i, i+3) == ”true”
            then
            i = i + 4
            res = true
        end
        while i <= len do
            local c = t[i]
            if c == ”,” then
                return i + 1,  res
            end
            if c ~= ” ” then
                return i, nil
            end
            i = i + 1
        end
```

```
        return i, res
    end
    return i, nil
end
```

Nel leggere il codice della funzione `parseBool()`, si deve tener presente che l'indice nell'array passato come argomento sarà posizionato su un segno di uguale o su una virgola, oppure alla fine dell'array, per effetto della funzione di parsing delle chiavi. L'idea, infatti, è quella di chiamare la funzione di lettura del valore subito dopo la chiamata della funzione di lettura della chiave, determinando l'avanzamento dell'indice.

3.3 Opzioni numeriche

Di base, qualsiasi parametro intero o frazionario come parti o dimensioni è rappresentato dal tipo numerico. Possiamo quindi intendere che una lunghezza come la distanza `xsep`, che separa orizzontalmente le etichette affiancate, appartenga a questo tipo anche se entrano in gioco le unità di misura.

Una funzione di lettura valore potrebbe essere la seguente:

```
local function parseDim(t, i) --> index, sp
    local c = t[i]
    local len = #t
    if i > len or c ~= "=" then
        return i, nil
    end
    i = i + 1
    local k = i
    while k < len and t[k] ~= "," do
        k = k + 1
    end
    local pos = k
    if t[k] == "," then
        pos = k - 1
    end
    local dim = table.concat(t, "", i, pos)
    return k+1, tex.sp(dim)
end
```

che molto semplicemente ricerca il carattere di separazione '`,`' oppure la fine dell'array determinando l'indice `k`, e usa la funzione LuaTEX `tex.sp()` per tradurre la stringa tra l'indice corrente e l'indice `pos` in punti scalati `sp`, l'unità di misura delle lunghezze interna a TEX (che corrisponde a 1/65536) punti tipografici.

3.4 Opzioni enumerazione

Questo tipo può assumere solamente i valori che appartengono a un dato insieme i cui elementi sono di tipo stringa. Per esempio, l'opzione `shape` può assumere i valori `rect` per rettangolo, `roundrect` per rettangolo con angoli arrotondati e `circle` per la forma circolare.

A un dato valore è anche possibile associare argomenti racchiudendoli tra parentesi tonde. Questi argomenti se previsti sono obbligatori. Così, nell'esempio pratico della forma dell'etichetta, è possibile definire larghezza e altezza del rettangolo o diametro del cerchio.

Assieme alle dimensioni di pagina e alle distanze orizzontali e verticali che separano le etichette, espresse dalle opzioni `xsep` e `ysep` rispettivamente, questi dati sono sufficienti per definire in modo completo l'esatta posizione sulla pagina di ciascuna etichetta, sapendo che la griglia è regolare e sulle righe e sulle colonne ci sono il massimo numero di etichette possibili in posizione simmetrica rispetto agli assi della pagina stessa.

Associare valori a un'enumerazione non è una novità: alcuni moderni linguaggi di programmazione (Rust per esempio) ne sono capaci attribuendovi il nome di tipi algebrici.[3]

A ben vedere le opzioni booleane sono opzioni enumerazione dove i possibili valori sono solamente `true` oppure `false`, se non fosse che la sintassi preveda opzionale specificare un valore `true`.

Mentre si lascia al lettore l'implementazione della funzione `parseEnum()`, perché simile a quello delle funzioni precedenti, nella prossima sezione presenterò un listato completo per la lettura dei valori di un dato sistema di opzioni.

3.5 Limitazioni

Una limitazione che forse potrebbe essere risolta è la defininizione di opzioni tramite espansione di macro. Non è possibile, infatti, inserire al posto di un valore una macro, perché il codice Lua non la espanderebbe a meno di non modificare opportunamente il comando TEX del parser.

4 Il parser principale

Una volta trasformata la stringa d'ingresso in array di caratteri, il codice del parser dovrà svolgere la lettura delle opzioni ⟨*chiave*⟩=⟨*valore*⟩ con i seguenti passi:

1. leggere dall'array la chiave dell'opzione;
2. determinare dalla definizione associata il tipo di dato;
3. leggere dall'array il valore per quel tipo;
4. tornare al passo iniziale con l'indice posizionato sul successivo carattere separatore oppure terminare, se si è raggiunta la fine dell'array.

Innanzi tutto predisponiamo il codice come modulo Lua contenuto in un file: dovremo memorizzare tutte le funzioni in una tabella locale e, come ultima istruzione, restituirne il riferimento. Lo schema è:

```
local lib = {}
lib.defopt = {
    ...
}
```

3. https://www.rust-lang.org/.

```
local function parseKey(t, i) --> index, key
    ...
end

local fnparselib = {}
fnparselib.bool = function (t, i) --> i, bool
    ...
end

fnparselib.dim = function (t, i) --> i, sp
    ...
end

fnparselib.enum = function (t, i) --> i, e
    ...
end

function lib:parseOption(s)
    ..
end
return lib
```

La chiave `defopt` conterrà le definzioni delle opzioni in forma di tabella, segue la dichiarazione della funzione `parseKey()` che vale per le chiavi di tutte le opzioni possibili. Le funzioni di parsing di valori, già introdotte precedentemente tranne per il tipo `enum`, sono invece incluse in una tabella indicizzata con i nomi dei tipi, in modo da facilitare l'implementazione della funzione principale di parsing chiamata `parseOption()` e che riporto di seguito:

```
function lib:parseOption(s) --> topt, err
    local t = {} -- option table
    for c in s:gmatch(".") do
        t[#t + 1] = c
    end
    local i = 1
    local len = #t
    local opt = {}
    local defopt = self.defopt
    while i <= len do
        local key
        i, key = parseKey(t, i)
        if not key then
            error("Error at index " .. i)
        end
        if not defopt[key] then
            error("Key not found " .. key)
        end
        local def = defopt[key]
        local optype = def.optype
        local fnparse = fnparselib[optype]
        local val
        i, val = fnparse(t, i)
        if not val then
            error("Invalid value for ".. key)
        end
        local fncheck = def.fncheck
        if fncheck then
            local ok = fncheck(val)
            if not ok then
                error(
                    "Invalid value for "..
                    key
                )
            end
        end
        opt[key] = val
    end
    return opt
end
```

Si noti nello schema che le funzioni di parsing ausiliarie sono tutte dichiarate come locali. Questo le renderà non visibili all'utente della libreria ma non alle altre funzioni grazie alla tecnologia delle *closure* di Lua.

4.1 Verifica

Proviamo la libreria con il seguente file LuaTEX che stampa i valori raccolti dalle opzioni:

```
% !TeX program = LuaTeX
\directlua{
local parselib = require "parseopt"
local opt = parselib:parseOption([[
    ysep = 1pt  ,
    showgrid, xsep=12pt,]]
)
print()
for k, v in pairs(opt) do
    print(k, v)
end
}
\bye
```

Se compilato, il sorgente stampa in console le righe seguenti:

```
showgrid    true
xsep    786432
ysep    65536
```

Le opzioni possono essere stampate con un diverso ordine poiché la tabella di Lua intesa come dizionario non è ordinata, perciò l'iteratore `pairs()` non garantisce sempre lo stesso ordine anche a parità di contenuto della tabella. Provate a compilare più volte di seguito per constatare questa proprietà.

Se vogliamo usare una macro, potremo scrivere:

```
% !TeX program = LuaTeX
\long\def\setupGrid#1{\directlua{
local parselib = require "parseopt"
local opt = parselib:parseOption([===[#1]===])
print()
for k, v in pairs(opt) do
    print(k, v)
end
}}

\setupGrid{
    ysep = 1pt  ,
    showgrid, xsep=12pt,}
\bye
```

4.2 Famiglie di opzioni

La libreria di parsing delle opzioni ha così raggiunto il livello minimo di implementazione con la struttura e i concetti fino a ora illustrati. Il prossimo passo è l'introduzione di una funzione che memorizzi internamente le definizioni delle opzioni.

A questo scopo, come già nel pacchetto xkeyval, è possibile introdurre sia il concetto di *famiglia*, la raccolta di un gruppo di opzioni, sia quello di *namespace*, un gruppo di famiglie.

In questo modo, il parser può organizzare le opzioni necessarie alla classe di documento o al singolo pacchetto che occupa un proprio namespace.

5 Conclusioni

Creare una chiara e intuitiva sintassi per la definizione di un sistema di opzioni basato sul formato chiave/valore può aiutare sia gli sviluppatori di pacchetti sia gli utenti. In questo lavoro introduttivo ho illustrato l'idea di *tipo* per le opzioni e presentato il codice Lua per implementare una libreria di parsing per LuaTEX generale perché in grado di utilizzare le *definizioni* delle singole opzioni.

Ho anche accennato ai problemi d'interazione tra le opzioni quando vi sono valori la cui correttezza dipende da altri parametri utente. Queste considerazioni e i brevi listati di codice mostrano come il problema di come facilitare l'utente nella configurazione di parametri sia molto più interessante di quello che ci si potrebbe attendere.

Il codice del parser qui presentato si basa sulla scansione sequenziale dei caratteri digitati dall'utente. Pur trattandosi di un problema interessante come esercizio di programmazione utile ad acquisire dimestichezza con Lua, potrebbe essere sostituito da sorgenti predisposti per l'uso di LPEG, abbandonando algoritmi che possiamo definire di basso livello, e senza che cambi l'interfaccia.

I soci GUIT potranno scaricare i file dei sorgenti dal sito dell'associazione con le modalità previste dagli amministratori e potranno sperimentare agevolmente il codice eseguendo gli script con `texlua` e LuaTEX.

Chi è interessato al progetto `Barracuda` che includerà la libreria di parsing di un sistema di opzioni così come accennato in questo lavoro, visiti periodicamente la pagina web `https://github.com/robitex/barracuda`.

6 Ringraziamenti

Ringrazio la redazione della rivista ArsTEXnica per l'opera di revisione che ha migliorato questo lavoro, oltre che, naturalmente, tutti gli altri membri dello staff. Ringrazio anche Luigi Scarso per il suo costante appoggio nello sviluppo di nuovi progetti basati su LuaTEX.

Riferimenti bibliografici

Ierusalimschy, Roberto (2016). *Programming in Lua*. Lua.org, 4ª edizione.

The LuaTEX development team (2018). *LuaTEX Reference Manual*, 1.0.7 edizione.

▷ Roberto Giacomelli
Carrara
`giaconet dot mailbox at gmail dot com`