

# Compilateur Lua - Projet IFT3150

Philippe Caron

28 août 2017

# 1 Introduction

Ce documents décrit le fonctionnement du compilateur **Luna** en détail. **Luna** est conçu pour compiler le langage Lua 5.3 et produit du code assembleur (syntaxe GAS) pour l'architecture de processeur Intel X86-64, pouvant être compilé par gcc sur un ordinateur Linux ou OSX.

## 1.1 Le langage Lua 5.3

Lua est un langage normalement utilisé pour les scripts, il est donc très simple, typé dynamiquement, et la gestion de mémoire est automatique. On pourrait dire que ces caractéristiques *intuitives* pour l'humain le rendent relativement complexe à implémenter du côté machine. Il est normalement implémenté en C et roule sur une machine virtuelle<sup>1</sup>. Étant donné sa proximité avec C, il est possible d'appeler directement des fonction C depuis le code. Évidemment, le but ultime à vouloir compiler ce langage est de produire des fichiers exécutable par le processeur lui-même, il n'est donc pas question de perdre aucune des fonctionnalités originale du langage. C'est pourquoi le code compilé par **Luna** a aussi une collection de mémoire automatique, ainsi que la possibilité d'appeler une fonction C directement<sup>2</sup>.

### 1.1.1 Opérateurs

Tous les opérateurs ont au moins un paramètre et au plus deux. Lors du stage de *preprocessing* ils sont regroupés et parenthésés adéquatement. Tous les opérateurs du langage Lua sont supportés par **Luna**, et certains autres ont été ajoutés. Voici la table complète des opérateurs et de leur priorité (le niveau 0 représentant la priorité la plus haute) :

#### 1.1.1.1 Table des priorité

Priorité	Niveau	Opérateurs
0	Opération sur les bits	<< >> >>>   & ^^ === != ~
1	Puissance	^
2	Opérateur unaire	- not ~ #
3	Multiplicatif	* / % \ //
4	Additif	+ -
5	Concaténation	..
6	Booléen	== ~= <= >= < >
7	Conjonctif	and
8	Disjonctif	or

1. <https://www.lua.org/manual/5.3/manual.html>, 1 - Introduction

2. La conversion de type n'est pas automatique, la fonction C est donc responsable de gérer la conversion de son côté

### 1.1.1.2 Référence

<code>a &lt;&lt; n</code>	<b>bsal/bshl</b>	Décale a de n bits vers la gauche (décalage arithmétique ou logique).
<code>a &gt;&gt; n</code>	<b>bsar</b>	Décale a de n bits vers la droite. Si le MSB est 1, tous les bits décalés depuis le MSB sont mis à 1 (décalage arithmétique, conserve la négativité).
<code>a &gt;&gt;&gt; n</code>	<b>bshr</b>	Décale a de n bits vers la droite. Les bits décalés depuis le MSB sont 0 (décalage logique, ne conserve pas la négativité).
<code>a   b</code>	<b>bor</b>	Effectue un 'OR' bit à bit entre a et b.
<code>a &amp; b</code>	<b>band</b>	Effectue un 'AND' bit à bit entre a et b.
<code>a ~ b</code>	<b>bxor</b>	Effectue un 'XOR' bit à bit entre a et b.
<code>a ^^ b</code>		
<code>a === b</code>	<b>beq</b>	Compare les pointeurs, retourne vrai si ils pointent vers le même endroit.
<code>!=</code>	<b>bneq</b>	Compare les pointeurs, retourne faux si ils pointent vers le même endroit.
<code>^^ ~</code>	<b>bxor</b>	Opérateur 'XOR' sur les bits
<code>^^ ~</code>	<b>bxor</b>	Opérateur 'XOR' sur les bits
<code>^^ ~</code>	<b>bxor</b>	Opérateur 'XOR' sur les bits

## 1.1.2 Types et représentation interne

### 1.1.2.0 Entiers/Adresses

#### 1.1.2.1 Valeurs spéciales

#### 1.1.2.2 Chaînes de caractères

#### 1.1.2.3 Tables

#### 1.1.2.4 Objets

##### 1.1.2.4.0 Fermetures

##### 1.1.2.4.1 Fichiers

**1.1.2.5 Pile**

**1.1.2.6 Nombres à virgule flottante**

**1.1.2.7 Fonctions**

**1.1.3 Fermetures**

**1.1.4 Assignation**

**1.1.5 Boucles**

**1.1.6 Librairies**

**1.2 Collection de déchets**

**1.2.1 Type de GC**

**1.2.2 Performance du GC**

## 2 Préprocesseur

Le but du préprocesseur est de standardiser tous les fichiers de code de manière à ce que le compilateur puisse éventuellement traiter des fichiers suivant un certain patron plus strict de formatage. C'est important puisque rien ne garanti que tous les programmeurs ont la même éthique d'écriture. Dans ce cas-ci particulièrement, le préprocesseur est très utile car le langage Lua a une syntaxe plus permissive que la plupart des autres langages impératifs. Alors que beaucoup exigent le fameux «;» à la fin de chaque ligne, en Lua il n'est nécessaire que pour séparer deux instructions consécutives ambiguës. Il existe également plusieurs forme équivalentes pour exprimer la même chose, ce qui pourrait devenir compliquer pour un compilateur. Finalement, les expressions ne sont pas forcément correctement parenthésées au moment de la compilation, le préprocesseur permet de compléter les parenthèse manquante.

### 2.1 Fonctions du préprocesseur

Afin de faciliter au maximum le travail du compilateur, le préprocesseur accomplit plusieurs tâches diverses, allant du formatage à la précompilation. Les voici :

#### 2.1.1 Détection d'erreur

Il est facile de faire des erreurs en programant, mais pas toujours de les trouver. Le simple fait de chercher une parenthèse mal fermée peut représenter une perte de temps importante. Par exemple dans le segment de code suivant, on voit qu'une parenthèse fermante manque.

```
1 local x = print(test(i)
```

Exemple 1 – Mauvais parenthésage

Un des rôles du préprocesseur va être d'avertir l'utilisateur de son erreur. Si on tente d'exécuter luna sur le code suivant, on obtient ceci :

```
FILE: tmp.lua
Error at line 1: Parenthesis mismatch.
local x = print(test(i)
                      ^
```

Comme on peut le voir, le programme indique une erreur en pointant sur la parenthèse qui n'a pas été fermée. Ce genre d'aide est très utile à l'utilisateur, et lui permet de sauver du temps. De plus en détectant les erreurs plus haut dans le processus de compilation et en les supportant, on réduit le risque d'erreur fatale plus tard dans le processus de compilation, où un code incorrect est très difficile à prévoir et peut facilement faire planter le compilateur.

#### 2.1.2 Effacement des commentaires

Une fonction bête mais essentielle du préprocesseur est d'effacer les commentaires laissés par l'utilisateur. Cette suppression permettra au compilateur de lire le fichier Lua sans avoir à se préoccuper de fragments de code inutiles. Un fragment de code comme celui-ci :

```
1 --Bonjour
2 local x = "tout" .. --[[le]] "monde"
3 --[[Il
4   fait
5   beau]]
6 local y = "dehors"
```

Exemple 2 – Commentaires multiples

en devient un beaucoup plus simple :

```

1
2 local x = ("tout" .. monde)
3 local y = "dehors"

```

### 2.1.3 Indentation

Même si le but ici n'était pas d'écrire un éditeur de texte pour le langage, l'idée de faire une fonction d'indentation semblait très raisonnable. En effet, que ce soit par presse ou par paresse, les programmeurs vont occasionnellement écrire du code mal formaté et il est alors très pratique de pouvoir l'indexer automatiquement. Le préprocesseur va également conserver les choix personnel du programmeur quand à ses «saut de ligne», de sorte qu'un code comme celui-ci :

```

1 if
2     y==x and
3         b==3 and
4     c==4
5 then
6     if y==2 then
7         print("y=2")
8     elseif
9         x==3 then
10        print("x=3")
11    else
12        print(c)
13    end
14 end

```

Exemple 3 – Code mal formaté

va être transformé par le préprocesseur pour produire un code bien indenté :

```

1 if
2     ((y == x) and
3     (b == 3)) and
4     (c == 4))
5 then
6     if (y == 2) then
7         print ("y=2")
8     else if
9         (x == 3) then
10        print ("x=3")
11    else
12        print (c)
13    end
14 end
15 end

```

Bien évidemment, une telle fonction requière une certaine quantité de ressource en comparaison avec une autre qui n'aurait pas autant d'éléments à prendre en charge. Il faut cependant réaliser qu'elle permet de lire des nombres et des opérateurs collés et ensuite de les décoller. Un des avantages immense de cette partie du processus sera évidente ultérieurement, alors que l'écriture d'une fonction de *parsing* deviendra triviale.

### 2.1.4 Parenthésage

Comme dans tous les langages à notation infixe, le compilateur doit s'assurer d'avoir un parenthésage complet avant d'évaluer une expression.

```
1 local x=3+4-2-3*-2^-2+4/5%8
```

Exemple 4 – Parenthésage incomplet

Une expression du genre est plutôt difficile à évaluer, c'est pourquoi le préprocesseur va former les expressions une à une en suivant l'ordre de priorité prescrit par le langage. Le résultat est le suivant :

```
1 local x = (((((3 + 4) - 2) - (3 * (- (2 ^ (- 2)))))) + ((4 / 5) % 8))
```

On constate immédiatement une lisibilité accrue ; il en va de même pour le compilateur. Pour les étapes suivantes on sait qu'il y a 1 ou 2 opérande par opérateur, il n'y aura donc que deux cas à gérer.

## 2.1.5 Précompilation

La ligne obtenue dans la précédente section est en réalité obtenue en exécutant `./luna -npc -s tmp.lua`, c'est à dire en spécifiant au compilateur qu'on ne veut pas précompiler. Sans la `switch -npc`, le résultat est en fait le suivant :

```
1 local x = 6.55
```

Le principe est d'éviter au programme compilé des calculs facilement évitable. Au fur et à mesure que les parenthèses sont formées, le préprocesseur vérifie si celles-ci sont possible à évaluer (qu'elles ne contiennent pas de variable). Si l'évaluation est réussie, il remplace la parenthèse par le résultat de son évaluation.

## 2.1.6 Standardisation

L'objectif final, et non le moindre, du préprocesseur est de limiter le langage à un sous ensemble sur lequel il le compilateur pourra compter. Le langage Lua offre de nombreuses contractions et macros que le préprocesseur transforme en version plus basique.

```
1 tmp = { allo = 9 }
2
3 function tmp.test(a)
4     return a.x:sub(i, i)
5 end
6
7 local function loc(n)
8     return n + 1
9 end
```

Exemple 5 – Utilisation des contractions et des macros

Comme on peut le voir dans l'exemple ci-haut, il est possible de référer aux indices textuels d'une table en utilisant «`table.index`» ; cela revient exactement au même qu'écrire «`table ["index"]`». La seconde forme sera priorisée dans le texte final puisqu'elle est standard et fonctionne avec tous les types d'index. On remarque aussi qu'il est possible de déclarer des fonction sans utiliser explicitement l'opérateur de déclaration «`=`». Cette déclaration en ligne est pratique car agréable à l'oeil, mais ce n'est pas un format conventionnel que le compilateur pourra traiter, il faut donc aussi les développer :

```
1 tmp = { allo = 9 }
2
3 tmp ["test"] = function (a)
4     return a : sub (i , i)
5 end
6 local loc ; loc = function (n)
7     return (n + 1)
8 end
```

Lors de l'expansion de la déclaration d'une fonction locale, un point-virgule s'ajoute au code<sup>3</sup>. Ce point-virgule est en fait essentiel car en Lua les assignations sont faites après l'évaluation de toutes les valeurs. Sans le point-virgule, lorsque la fonction `loc` serait évaluée, le terme «`loc`» n'existerait pas encore et elle ne pourrait pas s'y référer. Par contre le deux-point dans l'expression «`a : sub (i , i)`» n'est pas transformé. Ce choix a été fait pour des raisons d'optimisation. Cette optimisation est particulièrement visible dans les expressions de la forme suivante :

```
1 table.x.y.z:sub(1, 1) ==
2 table ["x"] ["y"] ["z"] ["sub"] (table ["x"] ["y"] ["z"] , 1 , 1)
```

Exemple 6 – Développement sous-optimal

On peut voir que développer le «:» est en fait sous-optimal, dans le second cas, la table «`table`» devra être indexée deux fois, une fois à 4 niveau de profondeur, la seconde à 3. En conservant le «:», on peut implémenter une manière beaucoup plus efficace de réaliser cette opération en rajoutant des instructions au compilateur.

## 2.2 Mécanisme

Le préprocesseur a de nombreuses fonctions à remplir. Ces fonctions sont pour la plupart cependant simplement destinées à faciliter la tâche des compilateurs ultérieurement et à favoriser les chances de succès, c'est-à-dire qu'aucune d'entre elles ne fait réellement partie du processus de compilation. Le tout semblant un peu superflu, on souhaite réduire au maximum le temps que le programme prend pour effectuer cette tâche. La première version du préprocesseur était très simple mais demandait le lire le texte à plusieurs reprises en modifiant une chose à chaque fois. La version présentement utilisée ne fait qu'un seul passage, et exécute chacune de ces fonctions simultanément.

### 2.2.1 Fonction de *parsing*

La reconnaissance de texte en Lua n'est pas la chose la plus évidente. Il faut donc une fonction robuste capable de différencier tous les termes réservés du langage de ceux utilisables pour les variables, de différencier une variable d'un opérateur même s'il n'y a pas d'espace entre les deux, d'un chiffre et ainsi de suite... Pour Luna, cette fonction s'appelle `nexttoken`. Pour un endroit quelconque dans le texte, lorsqu'appelée cette fonction retournera le prochain élément du langage, déjà formaté dans le cas d'un string, ainsi que la nouvelle position du lecteur.

```
1 tmp = "[[Paul profitait du soleil.
2 - "Il n'y avait pas beaucoup de monde sur la plage aujourd'hui", pensait-il.]]
```

Exemple 7 – Lecture d'un string

Dans l'exemple ci-haut position du lecteur est donné par «`°`». Suite à l'appel de la fonction `nexttoken` le programme se retrouve dans cet état :

```
1 tmp = "Paul profitait du soleil.\n\t- \"Il n'y avait pas beaucoup de monde
sur la plage aujourd'hui\", pensait-il." °
```

Le nouveau *string* produit utilise la notation classique avec les guillemets anglais, prenant soin de bien protéger les caractères spéciaux, comme le *newline* et le *tab* ainsi que les guillemets présents avec la barre oblique inversée.

### 2.2.2 Lecture de ligne

À chaque ligne la fonction `nexttoken` est appelée jusqu'à ce que le lecteur rencontre le caractère *newline*, atteigne un délimiteur d'environnement, ou atteigne la fin du fichier. Cependant, si la ligne se termine par

3. <https://www.lua.org/manual/5.3/manual.html>, section 3.4.11



un terme qui en nécessite un autre (comme un opérateur) la ligne ne sera pas considérée comme terminée. Au fur et à mesure que les mots sont lus, ils sont ajoutés à une table d'expressions. C'est cette table qui sera utilisée pour le parenthésage.

### 2.2.2.1 Parenthésage

Le système de parenthésage est plutôt simple, il est réalisé par les fonctions `comb1` et `comb2`, depuis la fonction `scan`. Essentiellement, `scan` les appelle successivement avec une liste d'opérateur à associer (dans l'ordre de priorité du langage). La fonction de combinaison parcourt la table, et lorsqu'elle trouve un opérateur de ses paramètres, elle va tout d'abord tenter de résoudre le calcul posé par cet opérateur grâce à `trysolve`. Dans le cas d'un échec elle va retourner un string qui correspond aux deux opérandes et l'opérateur entre parenthèses.

```

1  expr = { "a", "+", "b", "*", "c", "(", "3", "+", "(", "-", "2", ")", ")", "}"
2  scan(expr):
3  > scan { "3", "+", "(", "-", "2", ")", "}"
4    > scan { "-", "2" }
5      -- niveau unaire
6      > comb1 { {"-", "2"}, "not", "-", "~", "#" }
7        > trysolve ("-", "2", nil)
8          << "-2"
9        << "-2"
10       << "(-2)"
11       -- niveau additif
12       > comb2 { {"+", "(-2)"}, "3", "+", "-" }
13         > trysolve ("+", "(-2)", "3")
14         << "1"
15       << "1"
16     << "(1)"
17     -- niveau multiplicatif
18     > comb2 { {"a", "+", "b", "*", "c (1)"}, "*", "/", "\\", "%" }
19     -- niveau additif
20     > comb2 { {"a", "+", "(b * c (1))"}, "+", "-" }
21     << "(a + (b * c (1)))"

```

Exemple 8 – Trace partielle d'exécution de la fonction `scan`

`scan` s'exécute récursivement lorsque des parenthèses sont déjà présentes. Avant d'appeler les fonctions de combinaison, `scan` s'assure également de combiner les fonctions avec leurs paramètres. Il est impératif que cette fonction produise du code fidèle (sans parenthèse superflue), car en lua les parenthèses de trop sont utilisées pour la troncature de la séquence de retour, ainsi «`(var)`» n'a pas la même signification que «`var`».

### 2.2.2.2 Conversion de la macro «.»

La représentation sous forme de table est le moment idéal pour éliminer les index de table qui utilisent le point. Le préprocesseur peut facilement passer à travers les éléments, lorsqu'il détecte un point, il supprime cet élément ainsi que le précédent de la table et remplace le suivant par précédent ["suivant"].

## 2.2.3 Préprocesseur et parcours des environnements

Afin de préparer le code, le préprocesseur appelle la fonction de lecture de ligne en boucle. Mais parfois celle-ci tombe sur un délimiteur d'environnement. Les fonctions décrites précédemment sont toutes à l'affût des délimiteurs, si l'une d'entre elles détecte un mot réservé qui débute un environnement, le préprocesseur s'appellera lui-même en augmentant de 1 le paramètre d'indexation. Le préprocesseur descend

ainsi récursivement dans les environnements s'assurant par le fait même d'une bonne indexation et d'une structure cohérente.

## 2.3 Récapitulation

Le préprocesseur est utile pour de nombreuses raisons, la principale étant de produire du code lisible pour le compilateur. Cependant il est également pratique pour nettoyer un code mal indenté, détecter les erreurs de l'utilisateur et précompiler le code pour rendre celui-ci le plus court possible.

```
1 function add(v1, v2)
2 --[[
3     Ceci est un exemple de fichier qui
4     sera "preprocessed", similaire au code
5     du compilateur-ir
6 ]]
7 local function mem(v)
8     if true and v then
9         v = "(" .. "%rbx" .. ")"
10    end
11 end
12 return "\tmovq\t" .. v1 .. ", %rax\n" ..
13        "\tmovq\t" .. v2 .. ", %rdx\n" .. -- v2 dans %rdx
14        "\taddq\t" .. "%rax" .. ", " .. "%rdx" .. "\n"
15 end
```

Exemple 9 – Récapitulatif des fonctionnalités

```
1 add = function (v1, v2)
2     local mem ; mem = function (v)
3         if v then
4             v = "(" .. "%rbx" .. ")"
5         end
6     end
7     return ("\tmovq\t" .. (v1 .. (" , %rax\n" ..
8         ("\tmovq\t" .. (v2 .. (" , %rdx\n" ..
9             "\taddq\t%rax, %rdx\n")))))
10 end
```

### **3    Compileur**