

# Compilateur Lua - Projet IFT3150

Philippe Caron

29 août 2017

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Le langage Lua 5.3 . . . . .	3
1.1.1	Opérateurs . . . . .	3
1.1.2	Types et représentation interne . . . . .	4
1.1.3	Fermetures . . . . .	8
1.1.4	Assignation . . . . .	8
1.1.5	Boucles . . . . .	9
1.1.6	Fonctions variadiques . . . . .	10
1.1.7	Appels terminaux . . . . .	10
1.1.8	Librairies . . . . .	11
1.2	Collection de déchets . . . . .	12
1.2.1	Type de GC . . . . .	12
1.2.2	Performance du GC . . . . .	12
1.2.3	Bugs . . . . .	12
<b>2</b>	<b>Préprocesseur</b>	<b>13</b>
2.1	Fonctions du préprocesseur . . . . .	13
2.1.1	Détection d'erreur . . . . .	13
2.1.2	Effacement des commentaires . . . . .	13
2.1.3	Indentation . . . . .	14
2.1.4	Parenthésage . . . . .	15
2.1.5	Précompilation . . . . .	15
2.1.6	Standardisation . . . . .	16
2.2	Mécanisme . . . . .	18
2.2.1	Fonction de <i>parsing</i> . . . . .	18
2.2.2	Lecture de ligne . . . . .	18
2.2.3	Préprocesseur et parcours des environnements . . . . .	19
2.3	Récapitulation . . . . .	20
<b>3</b>	<b>Compilateur</b>	<b>21</b>
3.1	Jeu d'instruction IR . . . . .	21
3.2	Mécanisme . . . . .	25
3.2.1	Lecture des expressions . . . . .	25
3.2.2	Compilation . . . . .	25

<b>4</b>	<b>Compilateur-IR</b>	<b>26</b>
4.1	Standards x86-64 . . . . .	26
4.1.1	Convention d'appel . . . . .	26
4.1.2	Utilisation des registres . . . . .	26
4.2	Luna et le standard . . . . .	27
4.2.1	Utilisation des registres par le compilateur Luna . . . . .	27
4.2.2	Alignement de la pile . . . . .	28
4.2.3	Initialisation . . . . .	28
4.3	Gestion des registres et de la pile virtuelle . . . . .	29
4.3.1	Fonctionnement de la pile virtuelle et attribution des registres . . . . .	29
4.3.2	Masquage des pointeurs . . . . .	29
4.3.3	Buffer de la pile virtuelle . . . . .	29
4.3.4	Valeurs de retour . . . . .	30
4.3.5	Protection de la pile d'exécution réelle . . . . .	30
<b>5</b>	<b>Compilateur Luna</b>	<b>31</b>
5.1	Modificateurs . . . . .	31
5.2	Commentaires . . . . .	31

# 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) :

Table des priorité

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

Le sens de l'associativité décrit dans quel ordre les expression vont se former si plusieurs d'entre elles du même niveau de priorité se trouvent côte-à-côte. Par exemple :

```
1 x^y^z = (x ^ (y ^ z)) --droite
2 1-2-3 = ((1 - 2) - 3) --gauche
```

Exemple 1 – Exemple du sens de l'associativité

Pour de plus amples détails vous pouvez consulter le jeu d'instruction de la section ??.

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

### 1.1.2 Types et représentation interne

Vu que Lua est typé dynamiquement, chaque variable doit contenir son information de type. On tire profit du fait que les pointeur n'occupent que 6 bytes les plus bas pour stocker l'information de type. En décalant les pointeurs de 3 bits vers la gauche, on obtient suffisamment d'espace pour encore 8 types différents. Pour récupérer les types on fait 'AND' avec 7, et pour récupérer le pointeur 'SAR' avec 3. Le paragraphe suivant décrit les types et leur représentation interne. À des fins de clareté on fera référence à toutes les valeurs qui ont leur 3 LSB libre comme étant étiquetées (T : *tagged*) et toutes celles étant dans leur format *naturel* comme étant non-étiquetées (UT : *untagged*). Voici un tableau de référence pour comprendre les schéma de représentation interne :

Abbréviation	Signification
Ptr	Pointeur
Int	Entier
Float	Nombre à virgule flottante
Byte	Octet
Fct	Fonction
UT	Représentation naturelle
T	Le nombre est décalé de 3 bits vers la gauche et étiqueté
+	La case est occupée par une valeur
*	La case est occupée par un pointeur
#	La case est occupée par une valeur quelconque (peut être un pointeur ou une valeur)
X->B	La case contient la valeur B et est pointée par X
@	Indique l'adresse de la case
[x]	Représente une séquence de x fois le contenu de la parenthèse

#### Entiers/Adresses - 000

Le choix a été pour éviter les comparaisons constantes de représenter tous les nombres en lua sous forme de doubles. Ainsi le type entier est en quelque sorte désuet au point de vue de l'utilisateur, cependant, certaines structures internes utilisent encore sa représentation.

Une valeur contenant ses trois LSB à 0 indique qu'elle ne pointe sur rien et donc que le GC peut l'ignorer. C'est pourquoi toutes les adresses sont des multiples de 8. Le compilateur s'assure d'aligner les adresses de retour et d'appelle, ainsi que de conserver la mémoire et la pile alignée. Ce faisant, on évite les accidents où le GC tenterait de copier une adresse de retour. La reconnaissance par le GC n'est pas la seule utilité de cet alignement. En effet, l'alignement général du stack et de la mémoire sur des multiples de 8 améliore la performance du cache, car aucun *data chunk* ne risque d'être tronqué.

## Valeurs spéciales - 001

Les valeurs spéciales sont des constantes qui vont permettre au compilateur de d'obtenir de l'information sur certains object, au GC de reconnaître des cas particulier, ou simplement de représenter des constantes du langages. Voici la table de valeurs spéciales.

Nom	Valeur (hex)	Utilisation
FALSE	0x00000001	Représente la constante <b>false</b>
TRUE	0x00000009	Représente la constante <b>true</b>
NIL	0x00000011	Représente la constante <b>nil</b>
VOID	0x00000021	Indique que la pile est vide et ne contient plus de valeur
FRAH	0x00000081	(For Ahead) Signale au GC qu'une boucle <b>for</b> se trouve à cet endoroit sur la pile
NEW	0x7FFFFFFE1	Signale au GC que le le tableau est en cours d'initialisation
UNKN	0x7FFFFFFF1	(Unknown) Indique que la longueur d'un tableau n'est plus fiable et doit être recalculée

## Chaînes de caractères - 010

Les chaînes de caractères sont représentées par une case mémoire (8 bytes) contenant la taille de la séquence de byte qui suit (moins le caractère nul) sous la forme d'un entier non-étiqueté. La fin du string contient au moins un caractère nul, mais potentiellement plus pour être aligné avec la mémoire. Exemple :

CASE VIDE@(Longueur + 8 mod 8)	0000000073756E74
[Longueur](Caractères : UT Bytes[])	2072756F696E6F42
String->+(Longueur : UT Int)	000000000000000C

FIGURE 1 – Exemple de la représentation en mémoire du string "Bonjour tous"

## Tables - 011

Les tables en Lua ont la propriété de pouvoir recevoir non seulement des nombre mais aussi des chaînes de caractères comme index. Bien que pratique d'un point de vu programmation, ceci rend leur implémentation plus complexe. Plusieurs option sont disponible pour permettre un tel usage. L'idée évidente serait d'empiler des paires CLEF/VALEUR une par dessus l'autre. Cependant cette technique est difficile car elle requière un grand espace mémoire contigue, l'alternative évidente est donc la liste chaînée. La liste chaînée pose cependant un inconvénient majeur lors du parcours des valeurs de la table, si on veut accéder au 100<sup>ième</sup> élément, il faut faire 101 accès mémoire! Considérant que Lua repose beaucoup sur l'utilisation des tables, et que la majorité des tables sont initialisées à la bonne taille, une combinaison des deux options plus haute a été choisie.

*(Tableau (TB) : UT Ptr)
*(Liste chaînée (LC) : UT Ptr)
Table->+(Longueur : UT Int)

FIGURE 2 – Représentation en mémoire d'une table

NB : À l'intérieur de la table, toutes les valeurs sont non-étiquetées pour favoriser un traitement rapide de l'information.

Chaque pointeur de la liste pointe sur le prochain élément jusqu'à ce qu'il n'y en ait plus, il vaut alors NIL. Si une table ne contient pas de liste chaînée, alors le pointeur de liste vaut NIL. Tous les index non entier ( $\in \mathbb{R} \setminus \mathbb{Z}$ ) et les chaînes de caractères sont dirigés vers la liste chaînée.

*(A : UT Ptr)	*(B : UT Ptr)	*(C : UT Ptr)	NIL
#(VALEUR : T)	#(VALEUR : T)	#(VALEUR : T)	#(VALEUR : T)
LC->#(CLEF : T)	A->#(CLEF : T)	B->#(CLEF : T)	C->#(CLEF : T)

FIGURE 3 – Exemple d'une séquence de chaînage de la liste chaînée

La partie tableau de la liste contient tous les index entiers. Si on tente d'accéder à un index trop élevé ou trop bas en mode «accès», NIL sera retourné. En mode «définition» le tableau sera copié à un autre endroit dans la mémoire et redimensionné pour accommoder le nouvel index. Le GC est responsable de redimensionner une table inutilement trop volumineuse.

CASE VIDE@(Capacité + 2)
[Capacité]NIL
+(Décalage : T Int)
TB->+(Capacité : T Int)

FIGURE 4 – Base du tableau d'index entiers

Ceci donne la possibilité à l'utilisateur de créer une perte de mémoire assez massive en définissant deux index extrêmement éloignés l'un de l'autre, cependant un tel comportement n'aurait pas vraiment de but autre que celui de faire planter le programme. Le bénéfice d'avoir un tableau directement indexable surpasse largement les inconvénients.

## Objets - 100

Le type objet sert pour tous les types qui ne font pas partie des 7 principaux. En décalant à droite, on obtient un pointeur vers une case mémoire qui contient le type de l'objet, le compilateur doit donc faire une étape de plus pour obtenir le type, c'est pourquoi il décrit les objets moins fréquents. On considère que le type pointé en mémoire s'ajoute à 8 pour qu'il n'y ait pas de conflit avec les autres types existants. Par exemple, le type 4->0, est en fait le type 8 (le premier après Fonction (7)).

**Fermetures - 1000** Les fermetures suivent le même principe que la liste chaînée des tables, mais elles sont chaînées dans l'autre sens de sorte que l'objet le plus récent est accessible le plus rapidement. Leur forme est la même que celle des maillons des tables, mais avec une case mémoire en plus pour le type. Vu qu'elles pointent toutes vers la même racine(ENV), ce sont les feuilles qui sont sauvegardées dans chaque fonction.

*(ENV : UT Ptr)   NIL
#(VALEUR : T)
#(CLEF : T)
Fermeture->+(Type : 0)

FIGURE 5 – Format d'une valeur «enfermée»

**Fichiers - 1001** Les fichiers sont principalement gérés par du code C. L'objet fichier est une paire comprenant le type et soit un pointeur vers le fichier ouvert (`io.ftype(file) == "file"`), ou la valeur NIL (`io.ftype(file) == "closed file"`).

*(FILE : UT File Ptr)   NIL
File->+(Type : 1)

FIGURE 6 – Exemple de l'objet «file»

## Piles - 101

Les piles sont très importantes en Lua, une bonne partie du langage repose sur leur existence étant donné la possibilité de retourner un nombre indéfini de valeur. Les piles sont normalement volatiles, c'est-à-dire qu'elles ne sont pas sauvegardées en mémoire. Cependant, dans les fonctions variadiques la pile doit être sauvegardée c'est pourquoi le type 5 existe. Une fois désétiqueté, la valeur pointe sur une région de la mémoire ou de la pile (d'où le nom donné au type) qui peut être décrémentée successivement jusqu'à l'atteinte d'une valeur sentinelle (VOID).

Pile->#(VALEUR : T)
...
VOID

FIGURE 7 – Format d'une pile

## Nombres à virgule flottante (doubles) - 110

Toutes les manipulations numériques effectuées par l'utilisateur se font sur les doubles. Ceci évite de devoir vérifier le type de chaque opérande avant chaque opération, ce qui rendrait le code lourd et beaucoup moins performant. Étant donné leur format, les doubles ne peuvent pas être décalés vers la gauche. On pourrait simplement masquer les trois LSB et ainsi perdre une certaine précision de la partie fractionnaire (3 bits/52). Mais cette option n'est pas idéale : en plus de perdre plus de 5% de la précision, on se retrouve à devoir gérer des cas particuliers (parfois les opérateurs doivent utiliser `sarq` d'autre fois `xorq` pour désétiqueter, etc...). Le plus simple sans compromettre la précision est tout simplement de gérer des pointeurs de double.

Double->+(VALEUR : UT Float)
------------------------------

FIGURE 8 – Représentation d'un double dans la mémoire

## Fonctions - 111

Les fonctions sont représentées par deux cases, l'une contient l'adresse d'appel, tandis que l'autre pointe sur l'environnement de fermeture. Avant d'appeler une fonction, le vieil environnement est sauvegardé sur la pile, et le nouvel environnement est chargé.

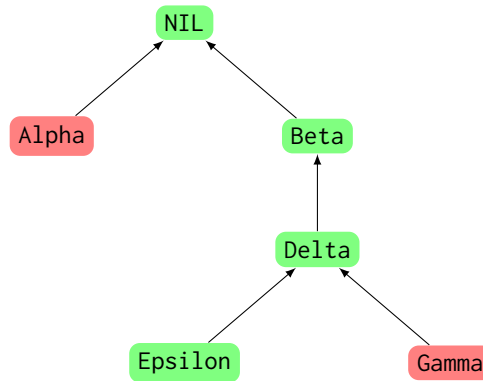
*(ENV : UT Ptr)
Fonction->*(FN : UT Fct Ptr)

FIGURE 9 – Représentation d'un double dans la mémoire

### 1.1.3 Fermetures

#### Principe de fonctionnement

L'environnement de fermeture fonctionne un peu comme un arbre. Au départ du programme, il pointe sur une case vide (NIL), la racine. Si une nouvelle fermeture est créée, elle s'ajoute et pointe sur l'ancien environnement, cela continue ainsi jusqu'à ce que la fonction se termine, à ce moment le vieil environnement est restauré et si rien ne pointe sur la fonction terminée, la feuille est perdue. Dans l'exemple suivant, on assigne la fonction Epsilon à une variable globale, ce qui conserve une référence à son pointeur d'environnement. Les fonctions Alpha et Gamma qui ne sont pas assignées voient leur environnement disparaître à la fin de leur exécution.



#### Déroulement

Une fois que le compilateur détecte qu'une valeur locale devra être utilisée depuis une autre fonction, il «enferme» cette valeur dans l'environnement de fermeture, une fois cela produit, toutes les références ultérieures à cette valeur (même celles de son propre environnement) seront toutes passées par l'environnement de fermeture. C'est pourquoi dans l'exemple suivant, même si `a` est modifié après que la fonction `x` soit définie, le résultat est quand même 10.

```
1 test = function (a)
2     local x ; x = function (b)
3         return (a + b)
4     end
5     a = (a + 2)
6     return x
7 end
8
9 local x = test (3)
10 print (x (5))
```

Exemple 2 – Fermeture

### 1.1.4 Assignment

Le processus d'assignation de Lua est assez complexe malgré son apparente simplicité. Tout d'abord, les déclarations multiples ne sont pas faites variable par variable, comme en C. Elles sont plutôt faites par lot. C'est parce qu'en Lua, toutes les destinations sont évaluées avant les valeurs.

```
1 local x, y, z = 1, 2, 3
2 x, y, z = y, y + x, x + y
```

Exemple 3 – Déclaration multiple



On pourrait s'attendre à ce que  $x$  prenne la valeur de  $y$ , et ainsi donne la valeur de  $2 * y$  à  $y$ , et alors que  $z$  vaille  $3 * y$ . Or le résultat est le même que si toutes les assignation avaient été effectuées simultanément.

Le compilateur utilise la pile pour sauvegarder la position des variables. Au fur et à mesure de l'évaluation, les adresses sont empilées, ensuite il calcule les résultats qui sont empilés eux aussi, puis une fonction de transfert fait correspondre chaque partie valeur de la pile à sa partie destination. Cette opération est beaucoup plus coûteuse qu'une affectation directe, mais elle est fidèle aux spécificités du langage<sup>3</sup>.

### 1.1.5 Boucles

Le compilateur Luna prend en charge toute les boucle définie dans le langage Lua soient **for**, **for ... in**, **while**, **repeat .. until**. Pour la plupart, leur implémentation est très simple. Dans le cas des boucles **while** et **repeat .. until** il s'agit d'un simple saut accompagné d'une comparaison. Dans le cas des boucles **for** c'est un peu plus complexe.

#### Boucle for ... in

Lorsque la boucle **for .. in** est appelée, le compilateur réserve 3 espace sur la pile d'exécution où il placera respectivement la fonction d'itération, puis les deux paramètres d'initialisation. La pile d'exécution ressemble alors à ceci :

...
*(Itérateur : T Fct Ptr)
#(P1 : T)
#(P2 : T)
[n]#(Variables d'itération : T)

FIGURE 10 – Bloc d'activation de la boucle **for .. in**

Les variables d'itérations sont définies par l'utilisateur pour prendre la valeur de retour (ou les valeurs) de la fonction d'itération.

#### Boucle for

La boucle **for** quand à elle est optimisée pour pouvoir «tourner» le plus vite possible, on veut donc stocker les doubles directement pour qu'ils soient facilement accessible et manipulable. Cependant, des double non-étiquetés sur la pile sont dangereux pour le GC, c'est pour quoi il est averti de la présence du **for** grâce au FRAH.

...
FRAH
*(Référence : T Float Ptr)
+(Itérateur : UT Float)
+(Arrêt : UT Float)
+(Incrémentateur : UT Float)

FIGURE 11 – Bloc d'activation de la boucle **for**

3. Cet algorithme de transfert pourrait être amélioré. Si on empile les destinations à l'envers, ils suffit alors de "popper" la destination et y placer la valeur. L'idée derrière la fonction de transfert actuelle était de permettre que des fonctions retournant plusieurs arguments puissent être retournée au sein de la déclaration (et non seulement en dernière position)

Toutes les variables d'itération ne sont pas étiquetées sauf la variable de référence. C'est pour pouvoir référer à l'itérateur dans le code. En Lua, il est impossible de changer la valeur de l'itérateur dans une boucle **for**, un nouveau double est donc synthétisé à chaque tour depuis l'itérateur et placé dans la case de référence.

### 1.1.6 Fonctions variadiques

#### Paramètres manquants

En Lua il est acceptable d'appeler une fonction avec le mauvais nombre de paramètres. Cette situation est problématique car Luna respecte les standard d'appel de C, langage où cette pratique est totalement interdite. Le problème vient du fait que les 6 premiers paramètres sont passés par les registres, mais les paramètres suivants sont sur la pile. Cela ne pose pas problème si il y a trop de paramètres, mais en appelant une fonction qui requière au dessus de 6 paramètres, si on en omet un ou plus, les éléments de la fonction vont tenter de référer à une partie de la pile d'exécution qui ne leur appartient pas. C'est pourquoi avant chaque fonction, la routine `_nil_fill` est appelée. Cette routine va modifier la pile de sorte à ce que les arguments manquants soient référençables (et contiennent la valeur NIL). L'exemple ci-dessous se produirait si on appelait une fonction de 8 arguments avec 6 arguments ou moins :

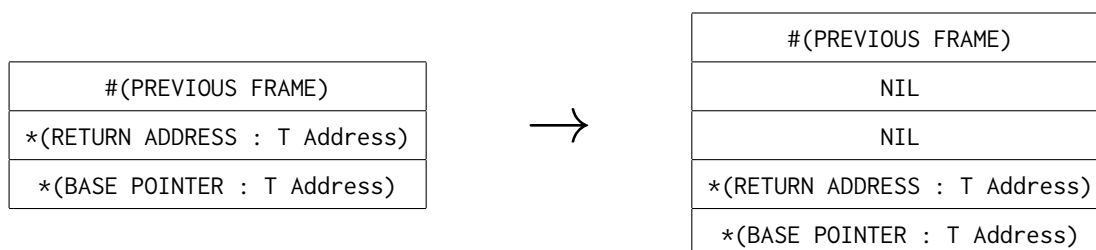


FIGURE 12 – Effet de `_nil_fill` sur la pile

#### Paramètres variables

Si le paramètre variable est inclu, la fonction va ensuite appeler la routine `_var_args`. Cette routine a pour but de récupérer les paramètres en trop et de les placer dans une pile à laquelle le programme peut avoir accès tout au long de la fonction.

### 1.1.7 Appels terminaux

Le compilateur optimise l'appel terminal en conservant le même *frame* pour tous les appels terminaux. En raison de l'imprévisibilité de la taille du *frame* dans une fonction variadique, celles-ci ne peuvent pas bénéficier de l'optimisation terminale.

### 1.1.8 Librairies

Le compilateur **Luna** a accès à une série de librairies où sont définies des variables et fonctions utiles. Quand les librairies sont construites, le compilateur inscrit les variables disponibles ainsi que leur provenance dans un fichier «.lib». Par la suite, chaque fois qu'il compile un fichier, il vérifie si celui-ci accède à une des variables des librairies. Si c'est le cas, il l'ajoute à la liste d'inclusion. De cette manière l'inclusion est automatique et l'utilisateur n'a pas à se tracasser. De plus, le fichier exécutable produit ne contient pas de code mort qui augmenterait inutilement sa taille.

#### **arg**

La librairie **arg** est toute petite, elle permet à un programme Lua d'accéder aux arguments de la ligne de commande et définit ces arguments dans la table **arg**.

#### **io**

Cette librairie est presque complètement implémentée, elle décrit la table **io**. Cette table définit une série de fonction qui permettent entre autre la lecture et l'écriture de fichier, la redirection du **stdin** ou **stdout**.

#### **lua**

Cette librairie est la librairie «de base», presque tout les programmes en ont besoin. C'est elle qui contient les références à **print**, **tostring**, **tonumber**, etc...

#### **os**

La table **os**, définie dans cette librairie contient des fonctions relatives au système d'exploitation. Elles permettent par exemple d'exécuter un programme, ou bien de quitter l'exécution abruptement.

#### **string**

La librairie **string**, contient toutes les fonctions relatives aux chaînes de caractères, comme le formatage, la recherche, ou le prélèvement.

#### **table**

Finalement, **table** est une librairie qui contient des utilités pour faciliter le traitement des tables, comme une fonction de bouclage ou de concaténation.

**\*\*\*Sans compter la librairie **io**, toutes les librairies sont à un stade très jeune de leur écriture.**

## 1.2 Collection de déchets

Le compilateur inclut dans le code son collecteur de déchet, ou *Garbage Collector* (écrit en C). Le but du GC est de réorganiser et compacter la mémoire pour faire de la place aux allocations ultérieures. À chaque fois que le programme effectue une opération qui remplit potentiellement la mémoire (transfert, appel de fonction, retour de fonction, définition de variable, définition de table, etc.) il appelle le GC. Celui-ci vérifie si la mémoire est encore en dessous du seuil acceptable, définie par défaut à 75%. Si c'est le cas, il retourne immédiatement à l'exécution, sinon, il commence son travail. Suite à la relocalisation de la mémoire le GC va évaluer si le programme utilise trop d'espace (la nouvelle plage mémoire se situe en dessous de 25% de la mémoire allouée) ou pas assez (la nouvelle plage mémoire est toujours au dessus de 75% même après la compaction). Si c'est le cas, il va ajuster la taille de la mémoire en conséquence (la diviser par deux, ou la doubler respectivement). De cette manière si un programme a besoin d'une quantité immense de mémoire pour un court laps de temps, il se la verra allouée et le programme ne plantera pas, mais lorsque sa demande va diminuer, le GC diminuera à son tour la taille de la mémoire afin d'alléger le stress que le programme impose à la machine.

### 1.2.1 Type de GC

Au départ, Luna était supposé avoir un GC Stop & Copy standard. Cependant, afin d'optimiser les opérations internes sur les structures (qui sont nombreuses, particulièrement dans le cas des tableaux) les pointeurs internes ne sont pas étiquetés. Ceci signifie que le GC Stop & Copy ne serait pas en mesure de détecter les fragments de structures internes nouvellement copiés dans la mémoire (à moins que celles-ci soient pourvues d'un tag lors de la copie). Pour faire plus simple, le programme fait tout simplement explorer récursivement toutes les structures qu'il rencontre au moment de la copie. Cela occupe plus de mémoire lors du processus de nettoyage, mais permet de sauver beaucoup d'étape pendant l'exécution normale du programme.

### 1.2.2 Performance du GC

La performance du GC n'est pas optimale, premièrement il y a le problème de mémoire décrit plus haut. Si on a une liste chaînée de 100 valeurs, ce qui est très plausible, le GC devra descendre récursivement jusqu'à la 100ième valeur puis retourner tout en haut. Non seulement le trajet est deux fois plus long mais des listes chaînées de grande taille sont susceptibles de causer un *Stack Overflow*. Changer le GC pour un GC Stop & Copy ordinaire serait envisageable, et probablement relativement facile étant donné que toutes les structures de copies sont en place.

Un autre désavantage (qui ne peut être corrigée par une architecture de GC plus efficace) provient de la manière dont sont traitées les assignations en Lua. Puisque les destinations sont d'abord empilées puis les valeurs sont transférées, on se retrouve avec une série de destinations sur la pile qui doivent être modifiées. Or les objets sur lesquels celles-ci pointent ne sont pas distinguables (peut-être un index de l'environnement de fermeture, peut-être un index de la table globale, peut-être une case sur la pile). Il faut donc attendre que toute la mémoire vivante soit copiée avant de pouvoir regarder le tag de «cœur brisé» et faire l'ajustement nécessaire.

### 1.2.3 Bugs

Le GC fonctionne bien avec toutes les fonctions natives du compilateur, mais il persiste un bug non résolu qui se produit lorsque le GC est appelé avant qu'une fonction C relative aux fichiers soit appelée (popen, fopen).

## 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 4 – 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 5 – Commentaires multiples

devient en fait 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 6 – 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 7 – 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 contraction 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 8** – 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>4</sup>. Ce point-virgule est en fait essentiel car en Lua les assignations sont faites après l'évaluation de toute 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 raison 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 9** – 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.

---

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

## 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 10 – 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 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.

#### 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" ].

## Parenthésage

Le système de parenthésage est plutôt simple, il est réalisé par les fonction `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 au 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 11 – 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 troncation de la séquence de retour, ainsi «`(var)`» n'a pas la même signification que «`var`».

### 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. Ce faisant, il transforme également les `elseif` en `else if` pour que le code soit plus facile à traiter.

## 2.3 Récapitulation

Le préprocesseur est utile pour de nombreuse raison, 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 courts possible.

Fichier avant traitement :

```
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 12 – Récapitulatif des fonctionnalités

Fichier après traitement :

```
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 Compilateur

Le compilateur est la partie de Luna chargée de transformer le code Lua en représentation intermédiaire, une représentation qui sera plus facile à interpréter pour le compilateur-IR. C'est essentiellement ce programme qui donne la structure que prendra le code final. Afin de décrire le langage Lua, le compilateur IR possède son propre jeu d'instruction.

### 3.1 Jeu d'instruction IR

<code>a + b</code>	<b>add</b>	Effectue l'addition de deux doubles.
<code>x and y</code>	<b>and</b>	Si <code>x</code> n'est pas <b>false</b> ou <b>nil</b> , poursuit l'exécution à <code>y</code> , sinon retourne la valeur de <code>x</code> , si l'exécution est poursuivie à <code>y</code> , retourne la valeur de <code>y</code>
<code>{, y}</code>	<b>append</b>	Ajoute un élément à la table en cours de définition au prochain index vide après sa conclusion.
<code>m &amp; n</code>	<b>band</b>	Effectue un 'AND' bit à bit entre <code>m</code> et <code>n</code> .
<code>p === q</code>	<b>beq</b>	Compare les pointeurs, retourne <b>true</b> si ils pointent vers le même objet.
<code>~n</code>	<b>binv</b>	Inverse les bits du nombre entier <code>n</code> .
<code>p != q</code>	<b>bneq</b>	Compare les pointeurs, retourne <b>false</b> si ils pointent vers le même objet.
<code>m   n</code>	<b>bor</b>	Effectue un 'OR' bit à bit entre <code>m</code> et <code>n</code> .
<code>a</code>	<b>brk</b>	Saute à l'extérieur de la boucle courrante.
<code>m &lt;&lt; n</code>	<b>bsal/bshl</b>	Décale l'entier <code>m</code> de <code>n</code> bits vers la gauche (décalage arithmétique ou logique).
<code>m &gt;&gt; n</code>	<b>bsar</b>	Décale l'entier <code>m</code> de <code>n</code> 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>m &gt;&gt;&gt; n</code>	<b>bshr</b>	Décale <code>m</code> de <code>n</code> bits vers la droite. Les bits décalés depuis le MSB sont 0 (décalage logique, ne conserve pas la négativité).
<code>m ~ n</code> <code>m ^^ n</code>	<b>bxor</b>	Effectue un 'XOR' bit à bit entre <code>m</code> et <code>n</code> .
	<b>call</b>	Procède à l'appel de la fonction préparée dans <code>params</code> ou <code>fparams</code> .
<code>x = y</code>	<b>chg</b>	Effectue le transfert de la valeur de <code>y</code> à l'adresse de <code>x</code> , pour les variables globales ou déjà définie.
<code>s .. t</code>	<b>con</b>	Effectue la concaténation de la chaîne de caractère <code>s</code> avec la chaîne de caractère <code>t</code> .
	<b>clo</b>	Réfère à un élément de l'environnement de fermeture.
<code>a / b</code>	<b>div</b>	Effectue la division de deux doubles.
<b>do</b>	<b>do</b>	Commence l'environnement <b>do</b> .
<code>}</code>	<b>done</b>	Conclu la définition d'une table après <b>init</b> .
<b>else</b>	<b>else</b>	Marque le début de la seconde branche du <b>if</b> .
	<b>encl</b>	«Enferme» une valeur dans l'environnement de fermeture.
<b>end</b>	<b>end</b>	Termine l'environnement <b>do</b> .

<code>x == y</code>	<b>eq</b>	Effectue la comparaison de x avec y, si x et y sont des string ou des doubles, cette comparaison compare l'objet pointé et non les pointeurs. Retourne <b>true</b> si les valeurs sont les mêmes, <b>false</b> sinon.
	<b>exit</b>	Termine le programme.
<b>function</b>	<b>fct</b>	Marque le début d'une fonction.
<b>end</b>	<b>fend</b>	Marque la fin d'une fonction.
<b>do</b>	<b>fido</b>	Marque l'entrée dans la boucle <b>for ... in</b> .
<b>end</b>	<b>fiend</b>	Marque la fin de la boucle <b>for ... in</b> .
<code>x:y</code>	<b>findex</b>	Indexe un objet, en conservant celui-ci sur la pile pour qu'il soit utilisé par <b>fparams</b> .
	<b>fname</b>	Spécifie le nom de la fonction.
<b>for</b>	<b>for</b>	Marque le début de la boucle <b>for</b> .
<b>for ... in</b>	<b>forin</b>	Marque le début de la boucle <b>for ... in</b> .
<code>f:y(x)</code>	<b>fparams</b>	Suite à <b>findex</b> , appelle une fonction à la notation contractée.
<b>do</b>	<b>frdo</b>	Marque l'entrée dans la boucle <b>for</b> .
	<b>free</b>	Libère les variables locales du context courant.
<b>end</b>	<b>frend</b>	Marque la fin de la boucle <b>for</b> .
<code>x</code>	<b>gbl</b>	Accède ou définit une variable globale.
<code>x &gt; y</code>	<b>gt</b>	Effectue la comparaison de x avec y, si x et y sont des string ou des doubles, cette comparaison compare l'objet pointé et non les pointeurs. Retourne <b>true</b> la première valeur est plus grande que la seconde, <b>false</b> sinon. Dans le cas des string, on utilise l'ordre lexicographique.
<code>x &gt;= y</code>	<b>gte</b>	Effectue la comparaison de x avec y, si x et y sont des string ou des doubles, cette comparaison compare l'objet pointé et non les pointeurs. Retourne <b>true</b> la première valeur est plus grande ou égale à la seconde, <b>false</b> sinon. Dans le cas des string, on utilise l'ordre lexicographique.
<code>a \ b</code> <code>a // b</code>	<b>idiv</b>	Effectue la division entière de a par b.
<b>end</b>	<b>iend</b>	Termine l'environnement <b>if</b> .
<b>if</b>	<b>if</b>	Marque le début de l'environnement <b>if</b> .
<code>x[y]</code>	<b>index</b>	Index une table avec la valeur spécifiée.
<code>{</code>	<b>init</b>	Début la définition d'une table.
<code>#x</code>	<b>len</b>	Si x est une table ou un string, retourne sa longueur. La longueur d'une table est considérée comme le nombre d'éléments non-nuls entre l'index 1 et le premier élément valant <b>nil</b> .
<code>x &lt; y</code>	<b>lt</b>	Effectue la comparaison de x avec y, si x et y sont des string ou des doubles, cette comparaison compare l'objet pointé et non les pointeurs. Retourne <b>true</b> la première valeur est plus petite que la seconde, <b>false</b> sinon. Dans le cas des string, on utilise l'ordre lexicographique.

<code>x &lt;= y</code>	<b>lte</b>	Effectue la comparaison de x avec y, si x et y sont des string ou des doubles, cette comparaison compare l'objet pointé et non les pointeurs. Retourne <b>true</b> la première valeur est plus petite ou égale à la seconde, <b>false</b> sinon. Dans le cas des string, on utilise l'ordre lexicographique.
<code>a % b</code>	<b>mod</b>	Calcule le modulo de a dans la base b
<code>a * b</code>	<b>mul</b>	Effectue la multiplication de deux doubles.
	<b>narg</b>	Spécifie le nombre d'arguments d'une fonction ordinaire.
	<b>nargs</b>	Spécifie le nombre d'arguments d'une fonction variadique.
<code>-a</code>	<b>neg</b>	Inverse le signe du nombre a.
<code>x == y</code>	<b>neq</b>	Effectue la comparaison de x avec y, si x et y sont des string ou des doubles, cette comparaison compare l'objet pointé et non les pointeurs. Retourne <b>true</b> si les valeurs sont différentes, <b>false</b> sinon.
<code>{, [x] = y}</code>	<b>new</b>	Ajoute un élément à la table en cours de à l'index spécifié.
<b>not</b> x	<b>not</b>	x prend la valeur <b>true</b> si x vaut <b>nil</b> ou <b>false</b> , et la valeur <b>false</b> dans tous les autres cas.
	<b>num</b>	Pousse un double sur la pile.
<code>x or y</code>	<b>or</b>	Si x n'est pas <b>false</b> ou <b>nil</b> , retourne x, sinon poursuit l'exécution à y, et retourne y
<code>f(x)</code>	<b>params</b>	Début l'évaluation des paramètres d'une fonction à venir.
	<b>place</b>	Suite à <b>chg</b> , consolide la pile de destinations avant de recevoir les valeurs.
<code>a ^ b</code>	<b>pow</b>	Effectue de calcul de a à la puissance b, si a est négatif, alors b doit impérativement être un entier sans quoi le résultat est indéfini.
<code>{, [x] = y}</code>	<b>put</b>	Suite à <b>new</b> ou <b>append</b> , place le dernier élément à la place voulue dans la table.
<code>a</code>	<b>ref</b>	Réfère à une variable locale sur la pile.
<code>a</code>	<b>ret</b>	Prépare à retourner des valeur d'une fonction.
	<b>rend</b>	Marque la fin de l'environnement <b>repeat</b> .
<b>repeat</b>	<b>repeat</b>	Marque le début de l'environnement <b>repeat</b> .
	<b>req</b>	Charge un autre programme lua.
	<b>rfct</b>	Retourne l'adresse de la fonction spécifiée.
<b>local</b> x = y	<b>set</b>	Début l'empilage de valeurs.
	<b>stack</b>	Suite à <b>set</b> , consolide la pile. Suite à <b>place</b> effectue le transfert des valeurs vers les destinations.
	<b>struct</b>	Inique que la valeur finale est en position pour retourner plusieurs valeurs.
<code>"..."</code>	<b>str</b>	Pousse une chaîne de caractères.
<code>a - b</code>	<b>sub</b>	Effectue la soustraction de deux doubles.
<code>,</code>	<b>tac</b>	Marque la séparation de deux éléments.
<b>return</b> f(x)	<b>tcall</b>	Appel terminal de la fonction spécifiée par <b>params</b> .
<b>then</b>	<b>then</b>	Marque le début de la première branche du <b>if</b> .
<code>x, y, (z)</code>	<b>trunc</b>	Suprime les valeurs de retour supplémentaires

<b>until</b>	<b>until</b>	Test la condition de sortie de la boucle <b>repeat</b> .
	<b>var</b>	Réfère à une variable à l'extérieur du programme.
<b>do</b>	<b>wdo</b>	Teste la condition de sortie de la boucle while <b>while</b> .
<b>end</b>	<b>wend</b>	Termine la boucle while <b>while</b> .
<b>while</b>	<b>while</b>	Marque le début de la boucle while <b>while</b> .



## 3.2 Mécanisme

Ici, le préprocesseur montre son importance. Maintenant que toutes les expressions sont séparées et parenthésées correctement le compilateur n'a qu'à lire les expressions une à la fois en suivant le parenthésage donné.

### 3.2.1 Lecture des expressions

Le compilateur lit les expressions grâce à la fonction `nextexpr`. Contrairement à `nexttoken`, cette fonction lit les expressions dans leur intégralité, c'est-à-dire que si une parenthèse ouvrante suit un symbole, la fonction va continuer de lire jusqu'à atteindre un équilibre de parenthèse.

### 3.2.2 Compilation

La compilation est divisée principalement entre deux fonctions : `translate` et `evaluate`, ainsi que toutes les fonctions d'environnement qui spécifient le format attendu dans lequel ces fonctions devraient être appelées.

#### Evaluate

`evaluate` gère essentiellement 3 modes différents de modifications de variables : le mode local, le mode accès/global, et le mode retour. Chacun de ces environnements se comporte de manière similaire et requiert donc l'étiquette à leur fin. Afin de «deviner» le nom de chaque variable, la fonction place d'abord toutes les valeurs dans une table et se souvient de la position du «=». Elle fait ensuite correspondre un nom à chaque variable en calculant le décalage. Si il n'y a pas assez de valeurs, elle remplit les valeurs manquantes avec des NIL.

#### Translate

La fonction `translate` elle reçoit ses paramètres de `evaluate`. Ce sont des segments de code garantis d'être des expressions. En obéissant au parenthésage, `translate` descend récursivement dans l'évaluation de l'intérieur de chaque parenthèse, et ajoute les opérateurs ainsi que les opérandes à une table. En sortant, la fonction possède donc une table facile à manipuler de la forme :

```
1 { { op, op1, op2 }, { op, op1, op2 }, ... } -- op: opérateur, op1-2:
    opérandes
```

ceci rend l'écriture du code IR beaucoup plus simple, et permet la gestion des cas spéciaux, comme `and` et `or` qui requièrent un traitement particulier.

## 4 Compilateur-IR

Le compilateur-IR est sans doute le fichier le plus complexe du compilateur **Luna**. C'est lui qui est chargé de reproduire la représentation IR en représentation assembleur, prête à être compilée par gcc. Le plus grand défi du compilateur-IR est de produire du code similaire au code C produit par gcc afin de pouvoir interfacer les deux langages. Pour ce faire, le compilateur se doit de respecter les standard de l'architecture.

### 4.1 Standards x86-64

#### 4.1.1 Convention d'appel

En x86-64, les fonction sont appelées avec un stack aligné sur un multiple de 16, et les 6 premiers paramètres sont passés par les registres (dans l'ordre) %rdi, %rsi, %rdx, %rcx, %r8, %r9. Les paramètres suivant sont poussés sur la pile de telle manière à ce que le plus bas soit le prochain, et le dernier le plus haut.

#### 4.1.2 Utilisation des registres

Registre	Sauvegardé	Fonction
%rax	Non	Valeur de retour, calcul
%rbx	Oui	Usage varié
%rcx	Non	Paramètre 4, calcul
%rdx	Non	Paramètre 3, calcul
%rbp	Oui	Base pointer
%rsp	Oui	Stack pointer
%rsi	Non	Paramètre 2, calcul
%rdi	Non	Paramètre 1, calcul
%r8	Non	Paramètre 5, calcul
%r9	Non	Paramètre 6, calcul
%r10	Non	Calcul
%r11	Non	Calcul
%r12	Oui	Usage varié
%r13	Oui	Usage varié
%r14	Oui	Usage varié
%r15	Oui	Usage varié

## 4.2 Luna et le standard

Les fonctions produites par Luna ne sont pas supposées être appelées par du code «étrangé», donc les standard de sauvegarde ne sont pas si importants au sens où tant que le compilateur gère correctement les registres pour lui, aucun programme extérieur ne risque d'être compromis par l'utilisation de registres supposément sauvegardés. Ceci dit, les fonctions produites par Luna ont toutes un préambule spécifique au langage, mais se comportent de manière très similaire à du code C par la suite.

```
.align 8
_FN1:
    movq    $1, %rax
    call    _nil_fill
    call    _prep_gc
test:                                # Appelee a partir d'ici, une fonction normale.
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -8(%rbp)
    movq    -8(%rbp), %rdx
```

C'est pourquoi la directive IR `fname` est pratique, le but est qu'en rajoutant un flag `-f` à la compilation, le compilateur protège les registres sauvegardés (**\*\*\*pas implémenté**). Le compilateur profite également du fait que certains registres sont sauvegardés pour y stocker des variables permanentes.

### 4.2.1 Utilisation des registres par le compilateur Luna

Registre	Sauvegardé	Fonction
%rax	Non	Valeur de retour, calcul
%rbx	Non	Pointeur de la pile de retour, calcul
%rcx	Non	Paramètre 4, calcul
%rdx	Non	Paramètre 3, calcul
%rbp	Oui	Base pointer
%rsp	Oui	Stack pointer
%rsi	Non	Paramètre 2, calcul
%rdi	Non	Paramètre 1, calcul
%r8	Non	Paramètre 5, calcul
%r9	Non	Paramètre 6, calcul
%r10	Non	Calcul
%r11	Non	Calcul
%r12	Oui	Pointeur d'allocation
%r13	Oui	Pointeur de la table globale
%r14	Oui	Pointeur de l'environnement de fermeture
%r15	Non	Calcul sur les opérandes mémoire, nombre d'arguments

Luna fait donc deux contravention au standard, mais comme mentionné plus haut, ces contravention serait rapidement réglée par un push-pop s'il le fallait.

### 4.2.2 Alignement de la pile

Le compilateur respecte également la convention d'alignement de la pile, chaque appel de fonction extérieure ressemble à ceci :

```
1  subq  $15, %rsp
2  andq  $-16, %rsp
3  .align 8
4  .fill 3, 1, 0x90
5  callq _io_open
6  call  _clear_regs
7  leaq  -16(%rbp), %rsp
8
```

### 4.2.3 Initialisation

Avant de commencer le programme, lua protège tous les registres à sauvegarder, ainsi que les registres %rsi et %rdi qui contiennent respectivement argv et argc (paramètres du main). Le compilateur nettoie également les registres car si ceux-ci ne sont pas typés, cela pose un risque de faire planter le GC

```
1  pushq %rbx
2  pushq %r12
3  pushq %r13
4  pushq %r14
5  pushq %r15
6  pushq %rdi
7  pushq %rsi
8  pushq %rbp
9  movq  %rsp, %rbp
10 xor  %rbx, %rbx
11 movq  %rbp, _stack_base(%rip)
12 movq  $134217728, %rax
13 movq  %rax, _mem_size(%rip)
14 call  _minit
15 movq  %rax, %r12
16 addq  $100663296, %rax
17 movq  %rax, _mem_max(%rip)
18 movq  $0, (%r12)
19 movq  $17, 8(%r12)
20 movq  $17, 16(%r12)
21 lea 3(, %r12, 8), %r13
22 addq  $24, %r12
23 movq  $17, %r14
24 call  _clear_regs
25 .align 8
26 .fill 3, 1, 0x90
27 callq _load_lua  #Chargement des librairies
28
```

## 4.3 Gestion des registres et de la pile virtuelle

### 4.3.1 Fonctionnement de la pile virtuelle et attribution des registres

Pour améliorer l'efficacité du compilateur, celui-ci utilise une pile virtuelle pour charger des valeurs dans les registres au lieu de les empiler. Afin de minimiser les risques d'erreur, les 8 registres de calculs se voient attribuer une place fixe (mod 8) dans la pile d'exécution. Ceci garantit que la restauration peut-être faite sans conflit (si le registre 3 doit être restauré, alors forcément qu'il n'est pas occupé car la pile ne s'y rend pas).

### 4.3.2 Masquage des pointeurs

Au paragraphe ?? on explique que pour des raisons de précision, les doubles sont stockés dans la mémoire. En choisissant un tel format, la précision est peut-être augmentée un peu ou, le calcul est extrêmement inefficace. C'est pourquoi durant les segments de calcul les doubles sont masqués par les registres %xmm4-%xmm15, tant qu'aucune opération sur la pile n'est effectuée et que les registres %xmm ne sont pas requis, ils peuvent y demeurer jusqu'à la fin du programme.

### 4.3.3 Buffer de la pile virtuelle

Finalement, les valeurs poussées ne sont pas directement ajoutées à la pile virtuelle, mais sont plutôt dirigées vers un espace tampon. Ceci permet les séquences du type :

```
1  ## Sans buffer :
2  movq  $1, %rdx
3  movq  $2, %rcx
4  addq  %rcx, %rdx
5  ## Alors qu'on voudrait :
6  movq  $1, %rdx
7  addq  $2, %rdx
8
```

Exemple 13 – Comparaison du code produit avec et sans buffer

Quand une valeur successive est poussée dans le buffer, alors seulement la valeur du buffer est enregistrée dans la pile virtuelle et requiert un registre. Le code produit par cette technique est très compact et a un aspect naturel :

```
1  movq  -8(%rbp), %rdx
2  movq  -16(%rbp), %rax
3  sarq  $3, %rax
4  movq  %rdx, %rbx
5  sarq  $3, %rbx
6  movsd (%rbx), %xmm4
7  addsd (%rax), %xmm4
8  movq  -24(%rbp), %rax
9  sarq  $3, %rax
10 addsd (%rax), %xmm4
11
```

Exemple 14 – Exemple d'utilisation du buffer et des masques

Grâce au buffer et aux masques, l'addition de 3 doubles se fait en un minimum d'opération. Les références successives à %rdx seront masquées par %xmm4, et le processus coûteux de désétiquetage est évité.

#### 4.3.4 Valeurs de retour

Les valeurs de retour sont toujours dans %rax comme le veut le standard, ce pendant en Lua, il est possible de retourner plusieurs valeurs. Quand une fonction retourne plusieurs valeurs, les valeurs supplémentaires sont empliées, et %rbx pointe sur le stack (non-étiqueté). Ici l'étiquetage n'est pas très important car un stack de retour est volatile. Cette méthode est efficace car les fonctions à valeurs multiples qui sont utilisées dans les calculs vont se comporter comme des fonctions à une seule valeur de retour, mais lorsque le compilateur le demande (généralement après la directive IR struct) il peut transférer les valeurs du stack dans des variables. Dans le cas des fonctions variadiques, l'adresse du stack est stockée dans un registre. Lorsqu'on veut avoir accès aux arguments variables, le programme simule un retour de fonction en incrémentant le pointeur du stack et en plaçant la première valeur dans %rax.

#### 4.3.5 Protection de la pile d'exécution réelle

La plupart des opérateurs sont traduits directement dans le code produit, ce qui ne nécessite aucune protection, mais quand le programme fait appel à des fonction extérieure, il doit protéger la pile d'exécution, c'est pourquoi le compilateur-IR garde trace de la taille réelle du stack, ainsi que la position actuelle de %rsp. Lors des appels de fonction, il place %rsp juste en haut de la dernière valeur locale.

## 5 Compilateur Luna

Après avoir appelé le préprocesseur, le compilateur, puis le compilateur-IR, **Luna** appelle gcc et compile le code assembleur produit. Une série de *switches* peuvent être utilisée lors de la compilation.

### 5.1 Modificateurs

Les arguments suivants peuvent être entrés dans **Luna**, voici un tableau de leur effet :

Argument	Effet
-lib	Compile un fichier .o pour être inclu dans une librairie
-tmp	Compile un fichier .o temporaire pour être inclu dans un programme
-i	Donne de l'information sur le compilateur
-S	Arrête le processus de compilation au niveau assembleur
-f	***Non implémenté : protège les registres pour que les fonctions puissent être appelées par C
-s	Laisse les sous-fichier de compilation
-npc	Désactive le précompilateur
-sas	Active la possibilité de définir la taille d'une table avec n = taille
-h	Affiche l'aide

### 5.2 Commentaires

Le compilateur **Luna** n'est pas complètement fini, mais il couvre déjà une partie importante du langage Lua 5.3. Les prochaines étapes sont de finaliser les fonctionnalités non implémentées et le doter d'un meilleur GC.