Compilateur Lua - Projet IFT3150

Philippe Caron

29 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 [1]. 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 [2].

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 parenthèsés adéquatement. Tous les opérateurs du langage Lua sont supportés par **Luna**, et certains autres ont étés 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 :

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

Exemple 1 – Exemple du sens de l'associativité

Pour de plus amples détails vous pouver consulter le jeu d'instruction du au paragraphe **??**.

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 repésentation 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 suffisemment 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'addresse 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, certaine 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 addresses 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 addresse de retour. La reconnaissance par le GC n'est pas la seule utilité de cet alignment. 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 `false`
TRUE	`0x00000009`	Représente la constante `true`
NIL	`0x00000011`	Représente la constante `nil`
VOID	`0x00000021`	Indique que la pile est vide et ne contient plus de valeur
FRAH	`0x00000081`	(For Ahead) Signale au GC qu'une boucle `for` se trouve à cet endoroit sur la pile
NEW	`0x7FFFFFE1`	Signale au GC que le le tableau est en cours d'initialisation
UNKN	`0x7FFFFFF1`	(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 inconvéniant majeur lors du parcours des valeurs de la table, si on veut accéder au $100^{\text{ième}}$ élément, il faut faire 101 accès mémoire ! Considérant que Lua repose beaucoup sur l'utilisation des tables, et que la majoritée 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 aie 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 chaînées.

*(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 accomoder 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'usager 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 programe. 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 pointes toutes vers la même racine(ENV), ce sont les feuilles qui sont sauvegarder 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"`).

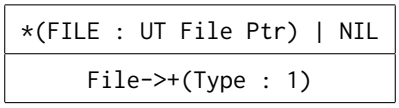


FIGURE 6 – Exemple de l'objet «file»

Piles - `101`

Les piles sont très importantes en Lua, une bonne partie du langage repose sur leur existance 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 sauvegardée 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).

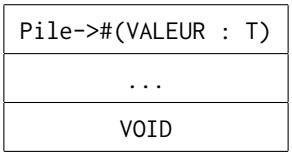


FIGURE 7 – Format d'une pile

Nombres à virgule flottante (doubles) - `110`

Toutes les manipulation numérique effectuées par l'utilisateur se font sur les doubles. Ceci évide 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 double 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 particulier (parfois les opérateur 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.

FIGURE 8 – Représentation d'un double dans la mémoire

Fonctions - `111`

Les fonctions sont représentées par deux cases, l'une contient l'addresse 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.

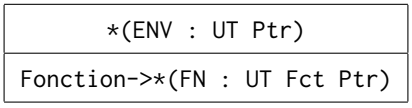
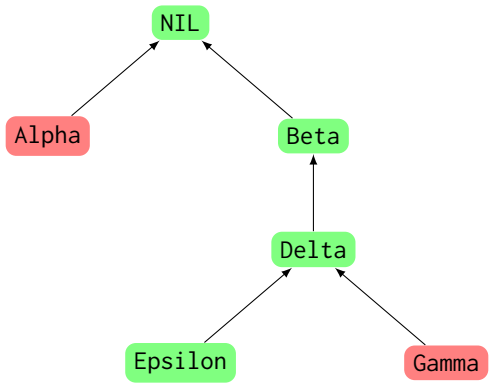


FIGURE 9 – Représentation d'un double dans la mémoire

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 environement 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 assignée 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, toute 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.

```
test = function (a)
   local x ; x = function (b)
      return (a + b)
   end
   a = (a + 2)
   return x
end

local x = test (3)
print (x (5))
```

Exemple 2 – Fermeture

1.1.4 Assignation

Le processus d'assignation de Lua est assez complexe malgré son apparente simplicité. Tout d'abord, les déclaration 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.

```
  local x, y, z = 1, 2, 3
  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 simultanéement.

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 corresondre chaque partie valeur de la pile à sa partie destination. Cette opération est beaucoup plus couteuse qu'une affection directe, mais elle est fidèle aux spécificités du langage [3].

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 empiles les destinations à l'envers, ils suffit alors de "poper" 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 posisition)

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 ou cette pratique est totalement interdite. Le problème vient du fait que les 6 premier paramètres sont passés par les registres, mais les paramètres suivant 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 :

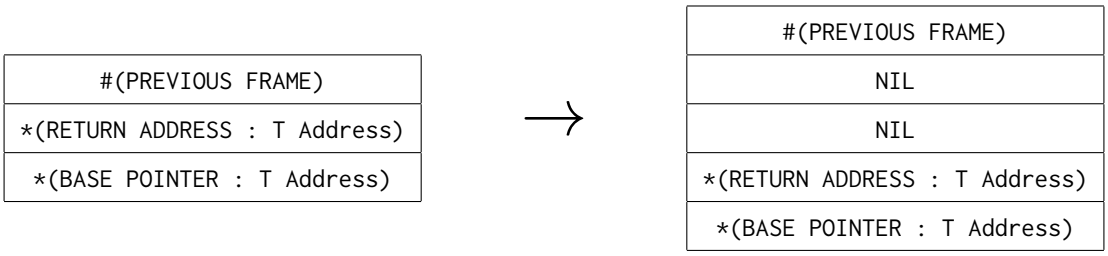


FIGURE 12 – Effet de `_nil_fill` sur la pile

Paramètres variables

Si le paramètre variable est inclu, la fonction va ensuite appeler la routine `_var_args`. Cette routine a pour bu 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'usager 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 défini ces arguments dans la table `arg`.

io

Cette librairie est presque complèment implémentée, elle décrit la table `io`. Cette table définie 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 référence à `print`, `tostring`, `tonumber`, etc...

os

La table `os`, définie dans cette libraire contient des fonctions relative 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 formattage, la recherche, ou le prélèvement.

table

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

*****Sans compter la libraire `io`, toutes les librairies sont à un stage très jeune de leur écriture.**

1.2 Collection de déchets

Le compilateur inclu 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 allocation ultérieures. À chaque fois que le programme effectue une opération qui remplie 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 sont 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) où 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 à 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 sont tour la taille de la mémoire afin d'alléger le stresse 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 opération interne sur les structures (qui sont nombreuse, 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 copiées dans la mémoire (à moins que celles-ci soient pourvue d'un tag lors de la copie). Pour faire plus simple, le programme fait tout simplement explorer récursivement toute 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 suceptibles de causer un *Stack Overflow*. Changer le GC pour un GC Stop & Copy ordinaire serait envisageable, et probablement relativement facile étant donné que toute 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 distingable (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 persite 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 formattage. 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 séparter 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 accompli plusieurs tâches diverses, allant du formattage 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 parenthèsage

Un des rôles du préprocesseur va être d'avertir l'usager 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'usager, 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 commentaire laissés par l'usager. Cette supression permettra au compilateur de lire le fichier Lua sans avoir à se préocupper 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 :

```
local x = ("tout" .. monde)
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 :

```
if
   y==x and
            b==3 and
   c==4
then
if y==2 then
 print("y=2")
   elseif
 x==3 then
   print("x=3")
   else
  print(c)
   end
end
```

Exemple 6 – Code mal formaté

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

```
if
   (((y == x) and
   (b == 3)) and
   (c == 4))
then
   if (y == 2) then
      print ("y=2")
   else if
         (x == 3) then
         print ("x=3")
      else
         print (c)
      end
   end
end
```

Bien évidemment, une telle fonction requière une certaine quantitée 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 Parenthèsage

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

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

Exemple 7 – Parenthèsage 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 :

```
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 suivante 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 :

```
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 contiennes pas de variable). Si l'évaluation est réussie, il remplace la parenthèse par le résultat de sont évaluation.

2.1.6 Standardisation

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

```
tmp = { allo = 9 }

function tmp.test(a)
   return a.x:sub(i, i)
end

local function loc(n)
   return n + 1
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 reviens exactement au même qu'écrire «`table ["index"]`». La seconde forme sera priorisé 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 :

```lua
tmp = { allo = 9 }

tmp ["test"] = function (a)
   return a : sub (i , i)
end
local loc ; loc = function (n)
   return (n + 1)
end
```

Lors de l'expansion de la déclaration d'une fonction locale, un point-virgule s'ajoute au code [4]. 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 :

```lua
table.x.y.z:sub(1, 1) ==
  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 destinée 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 reprise 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 un fonction robuste capable de différencier tous les termes réservés du langage de ceux utilisable 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 programe 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 guillements anglais, prenant soin de bien protéger les caractère spéciaux, comme le *newline* et le *tab* ainsi que les guillements présents avec la barre oblique inversée.

2.2.2 Lecture de ligne

À chaque ligne la fonction `nexttoken` est appelée jusqu'à ce que lecteur rencontre le caractère *newline*, atteigne un délimitateur 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 ajouter à une table d'expressions. C'est cette table qui sera utilisée pour le parenthèsage.

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"]`.

Parenthèsage

Le système de parenthèsage est plutôt simple, il est réalisé par les fonction `comb1` et `comb2`, depuis la fonction `scan`. Essentiellement, `scan` les appelles successivement avec une liste d'opérateur à associer (dans l'ordre de priorité du langage). La fonction de combinaison parcours 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.

```
expr = { "a", "+", "b", "*", "c", "(", "3", "+", "(", "-", "2", ")", ")"}
scan(expr):
> scan { "3", "+", "(", "-", "2", ")" }
   > scan { "-", "2" }
      -- niveau unaire
      > comb1 { {"-" , "2"}, "not", "-", "~", "#"}
         > trysolve ("-", "2", nil)
         << "-2"
      << "-2"
   << "(-2)"
      -- niveau additif
      > comb2 { {"+", "(-2)"}, "3", "+", "-")
         > trysolve ("+", "(-2)", "3")
         << "1"
      << "1"
<< "(1)"
   -- niveau multiplicatif
   > comb2 { {"a", "+", "b", "*", "c (1)"}, "*", "/", "\\", "%" }
   -- niveau additif
   > comb2 { {"a", "+", "(b * c (1))"}, "+", "-")
<< "(a + (b * c (1)))"
```

Exemple 11 – Trace partielle d'exécution de la fonction scan

`scan` s'exécute récursivement lorsque des parenthèses sont déja 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 parenthèse de trop sont utilisée 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 tomber sur un délimitateur d'environnement. Les fonctions décrites précédemment sont toutes à l'affût des délimitateurs, si l'une d'entre elles détecte un mot réservé qui débute un environnement, le préprocesseur s'appelera 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 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 cours possible.

Fichier avant traitement :

```
function add(v1, v2)
--[[
      Ceci est un exemple de fichier qui
      sera "preprocessed", similaire au code
      du compilateur-ir
   ]]
   local function mem(v)
      if true and v then
   v = "(" .. "%rbx" .. ")"
      end
   end
   return "\tmovq\t" .. v1 .. ", %rax\n" ..
      "\tmovq\t" .. v2 .. ", %rdx\n" .. -- v2 dans %rdx
      "\addq\t" .. "%rax" .. ", " .. "%rdx" .. "\n"
end
```

Exemple 12 – Récapitulatif des fonctionnalitées

Fichier après traitement :

```
add = function (v1 , v2)
   local mem ; mem = function (v)
      if v then
         v = "(%rbx)"
      end
   end
   return ("\tmovq\t" .. (v1 .. (", %rax\n" ..
      ("\tmovq\t" .. (v2 .. (", %rdx\n" ..
      "\addq\t%rax, %rdx\n"))))))
end
```

3 Compilateur

Jeu d'instruction IR

`m << n`	**bsal/bshl**	Décale l'entier `m` de `n` bits vers la gauche (décalage arithmétique ou logique).
`m >> n`	**bsar**	Décale l'entier `m` 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).
`m >>> n`	**bshr**	Décale `m` de `n` bits vers la droite. Les bits décalés depuis le MSB sont 0 (décalage logique, ne conserve pas la négativité).
`m | n`	**bor**	Effectue un 'OR' bit à bit entre `m` et `n`.
`m & n`	**band**	Effectue un 'AND' bit à bit entre `m` et `n`.
`m ~ n` `m ^^ n`	**bxor**	Effectue un 'XOR' bit à bit entre `m` et `n`.
`p === q`	**beq**	Compare les pointeurs, retourne **`true`** si ils pointent vers le même objet.
`p != q`	**bneq**	Compare les pointeurs, retourne **`false`** si ils pointent vers le même objet.
`a ^ b`	**pow**	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.
`-a`	**neg**	Inverse le signe du nombre `a`.
`~n`	**binv**	Inverse les bits du nombre entier `n`.
`not` `x`	**not**	`x` prend la valeur **`true`** si `x` vaut **`nil`** ou **`false`**, et la valeur **`false`** dans tous les autres cas.
`#x`	**len**	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 **`nil`**.
`a * b`	**mul**	Effectue la multiplication de deux doubles.
`a / b`	**div**	Effectue la division de deux doubles.
`a % b`	**mod**	Calcule le modulo de `a` dans la base `b`
`a \ b` `a // b`	**idiv**	Effectue la division entière de `a` par `b`.
`a + b`	**add**	Effectue l'addition de deux doubles.
`a - b`	**sub**	Effectue la soustraction de deux doubles.
`s .. t`	**con**	Effectue la concaténation de la chaîne de caractère `s` avec la chaîne de caractère `t`
`x == y`	**eq**	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 **`true`** si les valeurs sont les mêmes, **`false`** sinon.
`x == y`	**neq**	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 **`true`** si les valeurs sont différentes, **`false`** sinon.

`x <= y`	**lte**	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 `true` la première valeur est plus petite ou égale à la seconde, `false` sinon. Dans le cas des string, on utilise l'ordre lexicographique.
`x >= y`	**gte**	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 `true` la première valeur est plus grande ou égale à la seconde, `false` sinon. Dans le cas des string, on utilise l'ordre lexicographique.
`x < y`	**lt**	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 `true` la première valeur est plus petite que la seconde, `false` sinon. Dans le cas des string, on utilise l'ordre lexicographique.
`x > y`	**gt**	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 `true` la première valeur est plus grande que la seconde, `false` sinon. Dans le cas des string, on utilise l'ordre lexicographique.
`x and y`	**and**	Si `x` n'est pas `false` ou `nil`, poursuit l'exécution à `y`, sinon retourne la valeur de `x`, si l'exécution est poursuivie à `y`, retourne la valeur de `y`
`x or y`	**or**	Si `x` n'est pas `false` ou `nil`, retourne `x`, sinon poursuit l'exécution à `y`, et retourne `y`