Découverte d'un langage de script : *Lua*

David Anderson

Semestre A2004

Introduction

Au cours des TP de l'UV LO22, nous avons acquis une certaine maitrise du fonctionnement du shell Bash, et avons écrit ou corrigé plusieurs scripts interprétés par ce shell.

L'objectif de cette découverte étant, au dela de la familiarisation avec un environnement de travail de type Unix, de nous initier à un langage de script, il parait opportun d'en étudier un autre, afin d'élargir ses possibilités et outrepasser les limites du shell Bash.

J'ai choisi comme objet d'étude le langage *Lua* .

Fiche d'identité

Lua est un langage interprété conçu et implémenté par une équipe du département TecGraf de l'université de Rio de Janeiro. C'est un logiciel libre utilisé dans des domaines très variés :

- Robotique : Le cerveau du robot gagnant de la Robocup 2000 danoise était écrit en *Lua* .
- Matériel réseau : un switch rackable produit par Performance Technologies à une interface utilisateur écrite en *Lua* .
- Ludique : plusieurs dizaines de jeux (dont certains très connus) utilisent *Lua* comme langage d'extension du moteur.
- "glu" inter-logiciels : La NASA utilise *Lua* pour raccorder entre elles des applications mesurant et estimant les risques liés aux taux de gazs toxiques durant les lancements de la navette spatiale.
- Et bien d'autres : langage d'extension d'éditeurs de texte, analyseur de spectre d'IRM, modélisation moléculaire...

Lua a été conçu avec trois objectifs en tête : premièrement, avoir un langage extrêmement léger, au niveau syntaxique ainsi qu'au niveau des librairies fournies (tout le contraire de l'approche "tout fourni, piles incluses" de Python). Cet engagement permet à Lua de s'intégrer dans des applications embarquées ou la taille et le "poids" du code est un facteur décisif. Pour référence, sur plateforme x86, l'intégralité de l'interpréteur et des librairies fournies pèsent moins de 160ko de code compilé.

En plus de sa légèreté, l'accent a été mis sur la portabilité : l'intégralité du code source est de l'ANSI/ISO C pur[1], et compile donc sur toutes les plateformes ayant un compilateur C standard.

Enfin, le troisième pilier du langage est la philosophie "faites le vous mêmes". L'idée est que l'équipe *Lua* ne vous fournit pas un langage ultra-complet, avec pléthore de fonctions et structures dont vous n'avez que faire. Au lieu de cela, elle vous fournit les *moyens* de faire énormément de choses, et vous laisse implémenter exactement ce qu'il vous faut, en essayant de vous restreindre le moins possible. C'est ainsi qu'il n'y a pas d'orientation objet au sens strict du terme en Lua, mais les éléments structurels du langage permettent de mettre en place une infrastructure objet simple en moins de 20 minutes !

[1] Il y a une fonction dans une librairie qui fait exception actuellement (elle ne fonctionne que sur syst`emes POSIX), mais est conditionellement exclue `a la compilation le cas ´ech´eant. Son sort est actuellement en discussion sur la mailing-list *Lua* .

Lua vs. Bash

Maintenant que les présentations sont faites, abordons les principales différences entre ces deux langages.

But du langage

Le langage de script Bash a pour but premier de regrouper et automatiser des traitements répétitifs et fastidieux en ligne de commande. D'autres applications ont été trouvées, mais l'automatisation de tâches reste l'utilisation principale.

Exemple : les scripts de démarrage qui s'exécutent au boot d'un système GNU/Linux. Ces scripts automatisent l'exécution de centaines de commandes qui devraient autrement être entrées à la main par l'utilisateur.

Lua a été conçu pour être un langage d'extension avant tout. Il révèle toute sa puissance lorsque couplé à un autre langage comme le C, puisqu'il peut à ce moment la fournir des fonctions Lua à l'application mère et importer des fonctions de l'application mère pour piloter son fonctionnement et étendre ses possibilités. Il peut aussi être utilisé seul (il existe un interpréteur lua standalone), mais perd la puissance qu'aurait pu lui apporter le programme parent.

Exemple : le célèbre jeu *Baldur's Gate* est lourdement scripté avec *Lua* : le moteur expose à *Lua* des fonctions permettant de consulter et manipuler l'état du monde virtuel, et toute la logique de haut niveau est écrite en *Lua* . Cela permet de débugger le jeu très facilement (on peut écraser des fonctions en direct dans la console du jeu pour altérer le comportement du jeu), et rend l'extension de celui-ci bien moins ardue que dans un jeu tout-C/C++.

Types connus

Le langage de script Bash connait deux types fondamentaux, les chaines de caractères et les nombres entiers, et un type composé, les tableaux indexés numériquement. Ces variables sont par défaut globales, et supportent deux modificateurs : `local` pour limiter le champ d'une variable au script ou à la fonction courante et `readonly` pour créer une constante. Il n'y a pas de déclaration de type ; le langage est donc typé dynamiquement, en fonction de la valeur de la variable.

Lua connait, pour simplifier[2], cinq types fondamentaux :

nil Ce meta-type représente une valeur non-initialisée. Lorsqu'on tente de consulter une variable non-initialisée en *Lua* , sa valeur est `nil`. Lorsqu'on veut "effacer" une variable, on lui affecte la valeur `nil`. Il n'est utilisable que dans ces deux situations.

Les chaines de caractères Elles sont supportées de la même façon qu'en Bash.

Les booléens Une simple valeur : `true` ou `false`.

Les nombres Ils sont tous à virgule flottante double précision (équivalent du type `double` en C). Afin d'accomoder des processeurs n'ayant pas d'unité de calcul en virtule flottante (ou pour accélérer l'exécution, tout simplement), il est très facile de recompiler *Lua* pour utiliser des nombres entiers à la façon de Bash.

Les fonctions *Lua* se rapproche du Lisp à ce niveau, puisque les fonctions sont des variables de premier ordre. Cela permet par exemple à une fonction de renvoyer une fonction qu'elle aura construite dynamiquement, ou encore d'écraser n'importe quelle fonction par une autre que l'on définit nous mêmes, et ce pendant l'exécution du programme.

Lua connait également deux types composés :

Les tables Ce sont des tableaux associatifs, ou les valeurs sont indexées par des clés. Les clés et les valeurs peuvent être indifféremment l'un des 8 types fondamentaux/composés (`nil` n'est pas utilisable). Cela permet par exemple de créer une table ou d'autres tables sont indexées par des fonctions ! En parallèle à Unix ou "tout est fichier", dans Lua tout est table : même l'environnement global est en fait une table[3], et les noms des variables sont les clés indexant cette table.

Les clotures (de l'anglais *closure*) C'est un type composé d'une fonction et des variables nécéssaires à son fonctionnement. C'est un type qui peut être en pratique assimilé à une fonction, puisqu'il se comporte de la même façon. La cloture contient simplement plus d'informations qu'une fonction normale. Il est donc bon de reconnaitre les situations ou le code force *Lua* à créer une cloture, à des fins d'optimisation.

[2] Il y a deux types non-d´ecrits ici : `userdata` et `thread`, dont l'utilisation rel`eve de situations assez particuli`eres
[3] accessible via la table `_G` ; la table globale contient une r´ef´erence vers elle mˆeme, index´ee sous le nom `G`

Ces 6 types connaissent un seul modificateur : `local`, qui à le même effet qu'en Bash. Comme Bash, *Lua* est typé dynamiquement. Il dispose en outre de fonctions pour tester le type d'une variable.

Appel de fonctions

En Bash, on peut définir une fonction en donnant son nom et son implémentation. Il est possible de passer plusieurs paramètres à une fonction, et la fonction peut renvoyer une valeur à la fonction apellante.

En *Lua* , comme nous venons de le voir, les fonctions sont autrement plus flexibles. Elles sont définies uniquement par leur implémentation, et peuvent ensuite être référencées par plusieurs noms et à plusieurs endroits (dans une table, dans l'environnement global et en tant que variable locale dans une autre fonction par exemple). Il est possible de passer plusieurs paramètres à une fonction, mais une fonction peut également renvoyer plusieurs valeurs.

Par exemple, il est assez fréquent qu'une fonction renvoie en premier résultat le résultat du traitement effectué par la fonction, et en deuxième résultat un éventuel message d'erreur. Si le deuxième résultat n'est pas `nil`, on peut donc immédiatement dire qu'il y a eu erreur de traitement.

Gestion de mémoire

En Bash, les types existants ne nécessitent pas une gestion de mémoire bien complexe. Les variables globales existent jusqu'a ce qu'elles soient explicitement détruites, les variables locales sont détruites automatiquement lorsque le pointeur d'exécution sort du champ d'application de la variable.

En *Lua* , L'existence de fonctions comme type fondamental et des tables nécéssite une gestion de mémoire beaucoup plus poussée. Heureusement, tout cela est caché de l'utilisateur final, puisque Lua implémente un *g*arbage collector qui nettoie automatiquement les variables inutilisables (parce qu'on a perdu toute références vers elles). Il n'y a donc qu'a affecter la valeur nil à une variable pour la détruire : son ancien contenu n'étant plus référencé, il est détruit au prochain passage du *garbage collector*.

Fonctions disponibles

En Bash, il y a une grande quantité de fonctions intégrées (`builtin`) qui permettent de faire entre autres de la manipulation de chaines de caractères et des tests pour des branchements conditionnels. De plus, l'existence de la "fonction" backquote, qui permet d'exécuter n'importe quelle commande et de prendre sa sortie comme donnée, rend les scripts bash extensibles plus ou moins à l'infini du point de vue des fonctions. Par contre, il est impossible de changer le comportement des fonctions `builtin` et des types définis, autant qu'il est impossible d'en rajouter.

En *Lua* , la situation est quasiment diamétralement opposée lorsque *Lua* est utilisé seul. De par sa légèreté, il n'est fourni qu'avec 4 bibliothèques de fonctions, comportant chacune une dizaine de fonctions. Cela ne donne pas une grande marge, mais permet de faire de l'arithmétique (fonctions de mise à la puissance, d'exponentiation, etc.), de la manipulation de chaine (recherche de sous-chaine, extraction de sous-chaine, tronquage... Il y a même un moteur d'expression glob, pour traiter des motifs du type `foo*bar?`) et l'intéraction utilisateur/système (affichage, lecture d'entrée standard, I/O de fichiers, exécution de commandes). L'exécution de commandes est disponible, mais la donnée renvoyée est le code de sortie du programme. Il n'y a pas de façon portable d'obtenir la sortie du programme comme donnée, ce qui limite pas mal l'intérêt de l'invocation de commandes externes.

Par contre, *Lua* fournit un mécanisme extrêmement puissant pour altérer le comportement du type qui forme la fondation du langage : la métatable. Une métatable est une table normale, contenant une demi-douzaine de clés au nom défini. Cette table (apellons la `mt`) est ensuite rattachée en tant que métatable à une autre table `t`. Une fois le rattachement fait, le comportement de `t` sera dicté par les valeurs associées aux clés de la métatable.

Le sujet des métatables est beaucoup trop vaste pour l'aborder dans sa totalité dans cette comparaison, mais les métatables forment, avec les fonctions en tant que type fondamental, l'essentiel de la puissance et de la flexibilité unique de Lua.

Exécution parallèle

Bash n'a aucun support d'exécution parallèle de morceaux d'un seul script.

Lua implémente du multithreading coopératif, ou *coroutines* dans le jargon *Lua* . Une seule coroutine s'exécute à la fois, et elles peuvent se passer la main entre elles (implicitement, en abandonnant le contrôle du CPU, ou explicitement, en passant le contrôle à une coroutine précise). Le multithreading préemptif (que des logiciels comme Mozilla ou Open Office utilisent massivement) n'est pas implémenté car il requiert un support spécifique au niveau du noyau ou du matériel, et que cela nuirait donc à la portabilité de *Lua* . De plus, le multithreading préemptif nécessite des mécanismes de synchronisation qui alourdiraient considérablement le langage.

Base d'utilisateurs

Le Bash jouit d'une relative célébrité, ce qui rend les documentations, les didacticiels et les bouts de code accomplissant diverses tâches monnaie courante sur internet. En revanche, il n'y a aucune tentative de fédération du travail déjà accompli (à l'image du Comprehensive Perl Archive Network dont dispose Perl), ce qui conduit souvent à une réinvention involontaire de la roue par beaucoup d'utilisateurs du shell Bash.

Lua a, en comparaison avec Bash ou Perl, un "fan club" plus restreint (il atteint quand même plusieurs milliers d'utilisateurs !). Heureusement, l'équipe du TecGraf développe une documentation aussi aboutie que leur langage : pour la version 5 de *Lua* , il y a un manuel de référence, une introduction plus didactique, 8 notes techniques (explication sur un point précis ou une application précise) et plusieurs programmes d'exemple. L'auteur principal du langage a par ailleurs obtenu de ses éditeurs de mettre en ligne gratuitement une copie électronique de son livre *Programming in Lua*, un tour d'horizon complet de tous les aspects du langage.

A l'instar du Bash, il n'y a aucun projet officiel de fédération du travail accompli, mais une centralisation se met néanmoins en place, de façon non-officielle, sur le wiki du projet. Les utilisateurs y déposent les bouts de code intéressants qu'ils ont écrit, les modules de code pouvant servir, et y recensent des discussions sur des applications possibles. Ce wiki est en quelque sorte le petit frère du CPAN pour *Lua* .

Application d'exemple

Le *Lua* prenant toute sa puissance lorsqu'utilisé comme langage d'extension à une autre application, le script d'exemple que je vais donner ne va pas pouvoir accomplir grand chose d'utile, puisqu'il n'utilisera que les possibilités d'intéraction avec le monde mises à disposition par les 4 bibliothèques standard. Il servira donc principalement de vitrine pour les constructions complexes du langage, telles que les métatables.

Je me propose pour cette application de démonstration de reprendre l'un des programmes donnés en shell Bash et de le compléter. Ce programme est le script de saisie de carnet d'adresses. Dans sa version Lua, je l'ai étendu pour proposer une interface permettant de consulter le contenu du carnet, de rajouter ou supprimer des entrées, sauvegarder le carnet dans un fichier et charger le contenu d'un fichier et le fusionner avec le carnet en mémoire.

Il est à noter que pour tenter de réduire la taille du programme et préserver une lisibilité optimale, je n'ai pas inclus toutes les vérifications qu'il faudrait pour assurer le fonctionnement logique du programme.

Structuration des données

Le carnet d'adresses prendra la forme d'une table Lua, indexée numériquement. On associera à chaque clé numérique une autre table, contenant les informations du carnet proprement dites, indexée par le nom de l'information. On trouvera par exemple la paire `nom_famille = 'Durand'`.

Afin de rendre plus claire cette conceptualisation, voici un exemple de structure que cette table carnet d'adresses pourrait avoir, après quelques ajouts. Ce code est le code *Lua* pour un constructeur de table.

```
carnet =
{
  {
    nom = 'DURAND',
    prenom = 'Georges',
    tel = '0384424242',
  },{
    nom = 'DUPONT',
    prenom = 'Frederic',
    tel = '0388452945',
  },{
    nom = 'MICHEL',
    prenom = 'Marc',
    tel = '0384546232',
  },
}
```

Afin de rendre le programme plus flexible, nous allons rendre possible l'ajout de données supplémentaires. Pour uniformiser les entrées de carnet d'adresses, nous allons définir une table qui définit le modèle d'une entrée de carnet d'adresses. Cette table contiendra des fonctions, indexées par le nom de la propriété. La fonction est une fonction de validation : on l'appellera quand on aura besoin de vérifier que les données fournies par l'utilisateur sont dans un format adéquat. La fonction renverra `true` si la donnée fournie est valide, `false` sinon.

Ajout de données

Pour gérer les entrées dans le carnet d'adressess, nous allons utiliser l'une des astuces de lua : les métatables. En détail, une métatable permet de modifier le comportement d'une table en réponse à des événements précis. L'un de ces événements est la tentative d'accéder à un index qui n'existe pas. Par exemple, si `tab` est une table vide, alors `tab[foo]` apellera la méta-méthode `__index`.

Par défaut, une table n'a pas de métatable, et la métaméthode par défaut renvoit simplement `nil` lorsqu'on tente d'accéder a une valeur non-initialisée. Mais en rattachant une métatable appropriée, on peut altérer ce comportement. Voici par exemple le code qui définit une table qui génère une erreur (à la manière du set -u de bash) lorsqu'on tente d'accéder à une variable non-initialisée dans la table.

```
-- Définition de la métatable
ma_metatable =
{
  __index = function (table, cle)
      error(''Tentative d'accès à la variable non-initialisée '' .. cle)
  end
}

-- On prend une table vide, pour l'exemple
ma_table = { }

-- On affecte ma_metatable comme métatable de ma_table
setmetatable(ma_table, ma_metatable)

-- L'instruction suivante provoquera une erreur
print(ma_table.foo)
```

Nous allons utiliser la métatable différemment : nous allons utiliser la métaméthode __newindex, qui est apellée lorsque le script tente de créer une nouvelle entrée dans une table. Nous allons y mettre une fonction similaire à celle ci-dessus, qui provoquera une erreur. En effet, dans notre table carnet d'addresses, nous utiliserons une fonction d'insertion/extraction de la bibliothèque standard *Lua* , qui contourne les métatables pour écrire dans les tables. Ainsi, on pourra détecter les tentatives d'écriture qui n'utilisent pas

Sauvegarde et chargement de carnet

Pour gérer la sauvegarde d'un carnet d'adresses, nous allons utiliser une méthode assez connue : la sérialisation. Le principe est de convertir les structures de données en mémoire pour les stocker sur disque, pour pouvoir les recharger par la suite.

Python utilise intensivement la sérialisation, puisqu'on peut sérialiser absolument tout, jusqu'aux fonctions. Cela permet essentiellement de stocker tout l'environnement d'exécution sur disque pour le relancer plus tard.

En *Lua* , comme d'habitude, il n'y a pas de mécanisme aussi complexe implémenté dans le coeur du système. Par contre, la sérialisation est relativement facilitée par un facteur : il est possible de sauvegarder le contenu d'une table en écrivant le code lua d'un constructeur de table dans un fichier. Pour recharger les données, on incluera alors simplement le fichier *Lua* en question, avec les métaméthodes appropriées pour vérifier les données qui se font charger.

Cette sérialisation permet de stocker en clair les chaines de caractères et les nombres. On peut également enregistrer des fonctions, mais celles-ci seront enregistrées dans leur forme compilée (en bytecode *Lua*). Pour notre exemple d'application, ce sera suffisant.

Implémentation

J'ai tenté de commenter le code au maximum, afin de le rendre compréhensible sans connaissances approfondies du langage. L'implémentation met en oeuvre trois éléments qui donnent de la puissance à Lua : les fonctions en tant que variables de premier ordre, les métatables et métaméthodes, et la sérialisation de données.

L'implémentation est divisée en trois parties : la première contient les définitions des structures de données (les tables) utilisées, la deuxième contient l'implémentation de l'interface utilisateur, et la dernière est le programme principal, qui tourne en boucle sur les commandes entrées par l'utilisateur.

L'application n'est pas complète, et il y a des points qui mériteraient d'être améliorés. Ils sont commentés dans le code source. Je n'ai pas apporté les modifications que j'évoque parce qu'elles auraient inutilement complexifié le code, et que le but premier de cette application est d'illustrer le fonctionnement d'un programme en lua.

Conclusion

À la fin de cette comparaison, il apparait assez clairement que Lua est un langage bien plus flexible que le shell Bash. Néanmoins, les cas d'utilisation de chacun de ces langages sont si différents qu'il serait injuste de dire que l'un est supérieur à l'autre. Les deux excellent dans leurs domaines respectifs.

À quand un "shell Lua", qui apporterait la puissante flexibilité du *Lua* dans l'environnement de travail par excellence des scripts shell ? ;-)

Informations supplémentaires

Le site officiel du projet regorge d'informations complémentaires sur le langage *Lua* . si j'ai piqué votre curiosité avec ce rapport, je vous invite à vous rendre sur http ://www.lua.org pour en savoir plus sur ce langage.