Projet de Compilation Avancée
Machine Virtuelle pour bytecode Lua 5.1
v0.1

Mael Cravero

10 février 2025

Présentation du sujet

Contexte : Lua est un langage de script relativement minimaliste, conçu pour être embarqué au sein d'autres applications et en étendre les capacités via l'écriture de plug-ins. L'interpréteur Lua est écrit en C et est hautement portable, en plus d'être extrêmement compact (moins de 200ko). Il peut donc être intégré à virtuellement n'importe quelle application compilée, ce qui fait de Lua un langage d'extension particulièrement répandu, dans des domaines allant de la configuration de serveurs web au jeu vidéo.

Ce projet se focalise sur le bytecode de Lua, version 5.1. Il est **essentiel** de travailler sur cette version de Lua, et pas une autre, car le bytecode Lua ne dispose pas de spécification et peut donc varier d'une version à l'autre du langage.
A titre indicatif, le bytecode de Lua 5.0.2 dispose de 35 opcodes contre 38 pour Lua 5.1, certaine ayant été supprimées, d'autres rajoutées, et d'autres encore modifiées.

Objectif : Implémenter une VM pour bytecode Lua version 5.1.

Le projet peut être fait dans le langage de votre choix entre OCaml, C et C++. Si vous souhaitez le faire dans un autre langage, c'est possible à condition que je le valide auparavant. D'autres langages système comme Rust, Zig ou Ada seraient particulièrement adaptés.

Rendu : Vous devrez rendre, avant le **24/03/2025 à 23h59 (UTC+1) une archive** au format `.tar.gz` contenant le code de votre implémentation, ce qui inclut les sources, le build-system (Makefiles, Dune, ...) et les tests, ainsi qu'**un rapport** (en français ou anglais selon votre préférence) de maximum 10 pages décrivant la structure générale du projet, vos choix d'implémentation, et détaillant quelles améliorations potentielles pourraient y être faites.
L'implémentation et le rapport comptent à parts égales dans la notation. Un code ne compilant pas ne sera pas évalué. Tout warning durant la compilation devra être solidement justifié dans ce rapport.
Le rendu du projet se fera sur Moodle.

1 Introduction

Lua est un langage de script dynamiquement typé et multi-paradigme. On peut y faire de la programmation fonctionnelle facilement, et il est possible d'utiliser des *meta-tables* afin de se rapprocher de la programmation objet.

Le schéma de compilation de Lua est simple. Partons du programme Lua suivant :

```
local a = 9
local b = 33
print(a + b)
```

On peut le compiler avec `luac program.lua`, ce qui génère un fichier `luac.out` contenant le bytecode. Ce bytecode peut ensuite être passé à l'interpréteur via `lua luac.out` pour nous afficher 42. On peut également directement l'interpréter avec `lua program.lua`, ce qui nous affiche le même résultat, en conservant le bytecode en mémoire plutôt qu'en le stockant dans un fichier.

Si vous essayez d'ouvrir `luac.out` avec un éditeur de texte, vous remarquerez qu'il s'agit d'un fichier binaire. Il n'est pas possible de générer de bytecode sous représentation textuelle, il faudra donc commencer ce projet par *désassembler* ce bytecode, c'est-à-dire parser cette représentation binaire.

Le bytecode correspondant à ce program est composé de métadonnées comprenant entre autres une liste des constantes utilisées, une liste de noms de variables, et 6 instructions :

```
[...]
Constants:
  [0] 9.000000
  [1] 33.000000
  [2] print
Locals:
  [0] a
  [1] b
[...]
loadk         0 0
loadk         1 1
getglobal     2 2
add           3 0 1
call          2 2 1
return        0 1
[...]
```

Le bytecode indique donc que le programme contient 3 constantes et 2 variables locales. On appellera "registre X", noté R(X), les pseudos-registres utilisés par les instructions. Il ne s'agit pas réellement de registres mais en réalité d'emplacements dans la stack frame de la fonction.

- `loadk 0 0` charge dans R(0) la 0-ième constante de la fonction (9)
- `loadk 1 1` charge dans R(1) la 1-ième constante de la fonction (33)
- `getglobal 2 2` met dans R(2) la globale correspondant à la 2-ème constante de la fonction(`print`)
- `add 3 0 1` met dans R(3) la somme de R(0) et R(1)
- `call 2 2 1` effectue un appel à R(2) avec 2 - 1 = 1 argument et 1 - 1 = 0 valeurs de retour.
- `return 0 1` effectue un `return` à la fonction appelante avec 1 - 1 = 0 valeur de retour. Vu qu'on est déjà dans la fonction top-level, le programme se termine.

On a donc bien un appel à la fonction `print` (récupérée depuis un environnement global) avec pour argument la somme de `a` et `b` (stockée dans R(3)).

2 Structure du bytecode

Le bytecode Lua 5.1 n'est pas linéaire, mais arborescent. Il se compose de deux sections principales :

Header. Un en-tête comprenant des informations générales sur le bytecode, en particulier sa version et des informations spécificiques à la plate-forme (taille d'un entier, taille d'un pointeur, endianness, etc). Cet en-tête commence avec les 4 caractères ESC "Lua" ou `0x1B4C7561` en héxadécimal.

Function block. Contient les informations propres à une fonction en bytecode, dont des métadonnées (nom de la fonction, lignes de définition...), les instructions de la fonction, et des listes indiquant les variables locales, constantes, *upvalues* (variables capturées), et sous-fonctions (elles-mêmes représentées par des *function blocks*).

Un fichier bytecode typique contiendra un header, suivi d'un seul function block correspondant au code de l'intégralité du fichier compilé. Si des fonctions sont définies dans notre fichier, elles seront accessibles sous la forme de function blocks enfants du function block principal.

Le fichier suivant :

```
local a = 42
c = "Global!"
local function f(b)
    print(a)
end
```

est donc compilé en un fichier bytecode composé ainsi :

- Un header.
- Un function block représentant le fichier avec :
  - Des instructions bytecode.
  - Deux variables locales (`a` et `f`)
  - Trois constantes (`42`, `c` et `"Global!"`)
  - Une sous-fonction correspondant à `f`, elle-même un function block avec :
    - Des instructions bytecode.
    - Une variable locale (`b`)
    - Une upvalue (`a`)
    - Une constante (`print`)

Il faut bien noter qu'en Lua, toute variable qui n'est pas explicitement locale est globale (comme `c` et `print` dans notre exemple). Les variables globales ne résident pas dans la pile de la VM mais dans une table dédiée, où elles sont indexées par leur nom. Il faudra donc que la VM implémente à la fois une pile (utilisée pour les variables locales et les stack frames des fonctions) et cet environnement global.

3 Déroulement conseillé

Pour bien débuter le projet, je vous recommande de commencer par gérer les fonctionnalités dans cet ordre :

Parsing du header. Commencer par parser la structure simple des headers et vous assurer que vous pouvez manipuler le format binaire. Pensez à bien suivre la spécification liée dans la section Ressources lorsque vous parsez les fichiers. Notez bien que, les fichiers étant binaires, vous devrez directement manipuler cette représentation binaire et donc lire directement les octets du fichiers (donc n'essayez pas d'utiliser un générateur de parseur style ocamlyacc ou bison).

Parsing des function blocks. Parser le format binaire des function blocks est comparable au fait de parser le header du fichier, avec la nuance que les function blocks n'ont pas une taille fixe.

Dump des instructions. Ajouter une option pour pouvoir dump le bytecode sous format textuel est essentiel pour vérifier que le résultat de votre parseur est correct, et est en particulier important pour les instructions. Encore une fois, référez vous à la spécification du bytecode.

Opérations arithmétiques simples. Maintenant que vous pouvez afficher le contenu d'un fichier, vous pouvez commencer l'implémentation de la VM à proprement parler. Commencez par gérer des fichiers très simples comme :

```
local a = 1
local b = 2
local c = a + b
```

Si vous dumpez la stack d'exécution de votre VM, vous devriez voir qu'un registre contient bien la valeur 3.

Gestion de la primitive `print`. On veut maintenant pouvoir directement afficher à l'utilisateur des informations sans passer par un dump de l'état mémoire de la VM. On implémentera donc une gestion basique des appels de fonctions et la primitive `print`, qui affiche sur la sortie standard nos valeurs Lua. Cette primitive étant une variable globale, vous pouvez l'implémenter via une valeur pré-insérée dans la table de globales et qui pointera vers une fonction interne à la VM.

Une fois tout-ceci fait, vous devriez être capables d'exécuter correctement ce programme :

```
local a = 1
local b = 2
print(a + b)
```

qui affiche 3 sur `stdout`. Il s'agit du **minimum** de ce que j'attends pour ce projet : à vous ensuite d'ajouter des features supplémentaires à votre VM, notamment le support d'instructions supplémentaires du bytecode.

Comme le disait Donald Knuth, *Premature optimization is the root of all evil.* N'essayez pas de sur-optimiser votre VM d'entrée, commencez par implémenter quelque chose de fonctionnel avant de chercher à l'améliorer.

Bon courage !

4 Ressources

— Spécifications du bytecode Lua 5.1 : `https://www.mcours.net/cours/pdf/hasclic3/hassclic818.pdf`
— Blog post implémentant un parser pour bytecode Lua en Python, dont vous pouvez vous inspirer : `https://openpunk.com/pages/lua-bytecode-parser/`
— Définition des opcodes dans le code source de Lua 5.1 : `https://www.lua.org/source/5.1/lopcodes.h.html`