Machine Virtuelle Lua 5.1

Écrit par:

BENAISSA Meriem 21418006

Encadrant :

Cravero Mael

Promotion 2024 - 2025

Contents

1 Introduction

Lua est un langage de script léger, conçu pour être intégré facilement dans d'autres applications. Il est particulièrement utilisé dans le domaine du jeu vidéo, de la configuration de logiciels et du scripting embarqué. Son interpréteur, écrit en C, est optimisé pour être compact et portable, ce qui le rend très populaire dans divers environnements.

Dans ce projet, nous nous concentrons sur le **bytecode** de Lua, plus précisément la version 5.1. Contrairement aux langages disposant d'une spécification formelle de leur bytecode, Lua 5.1 présente une structure interne non standardisée qui évolue d'une version à l'autre. Cette particularité impose une approche rigoureuse pour comprendre et manipuler son exécution.

L'objectif principal de ce projet est donc d'implémenter une **machine virtuelle** capable d'exécuter du bytecode Lua 5.1. Pour cela, nous devons effectuer plusieurs étapes clés :

- Analyser la structure d'un fichier bytecode Lua.
- Mettre en place un système de *parsing* et de *dump* du bytecode pour en afficher une version lisible.
- Implémenter une machine virtuelle capable d'interpréter et d'exécuter ces instructions.
- Gérer les primitives essentielles comme les opérations arithmétiques et les appels de fonctions globales.

Dans ce rapport, nous décrirons en détail les différentes étapes de l'implémentation, les choix techniques effectués ainsi que les défis rencontrés. Nous conclurons par une discussion sur les améliorations possibles et les perspectives d'évolution de ce projet.

2 Analyse du Bytecode Lua 5.1

Le bytecode de Lua 5.1 est une représentation binaire intermédiaire d'un programme écrit en Lua. Il est produit par le compilateur Lua et est conçu pour être exécuté par l'interpréteur Lua ou une machine virtuelle dédiée. Ce bytecode est optimisé pour être compact et rapide à exécuter tout en conservant la structure et la logique du programme source.

2.1 Structure Générale du Bytecode

Un fichier bytecode Lua 5.1 est composé de plusieurs sections essentielles :

- **L'en-tête** : Il contient des informations sur la version de Lua utilisée, la plateforme cible, ainsi que des paramètres relatifs à la représentation des nombres et des tailles de données.

- **Le bloc de fonction principal** : Il représente la fonction globale du script et contient les instructions, les constantes, les variables locales et les sous-fonctions.
- **Les constantes** : Liste des valeurs constantes utilisées dans le script, telles que les nombres, les chaînes de caractères et les identifiants de fonctions globales.
- **Les variables locales** : Déclarations des variables internes utilisées dans les fonctions, avec leur portée.
- **Les instructions bytecode** : Séquence d'opcodes qui définissent les opérations à exécuter par la machine virtuelle.

2.2 Structure des Données Utilisée

Pour représenter un programme Lua sous forme de bytecode, la structure suivante a été définie :

```
type lua_undump = {
  mutable bytecode : bytes;  (* Contient le bytecode du programme Lua *)
  mutable index : int;  (* Position actuelle dans le bytecode *)
  mutable vm_version : int;  (* Version de la machine virtuelle *)
  mutable bytecode_format : int;  (* Format du bytecode *)
  mutable big_endian : bool;  (* Indique l'endianness de la plateforme *)
  mutable int_size : int;  (* Taille des entiers *)
  mutable size_t : int;  (* Taille des pointeurs *)
  mutable instr_size : int;  (* Taille d'une instruction *)
  mutable l_number_size : int;  (* Taille d'un nombre Lua *)
  mutable integral_flag : int;  (* Indique si les nombres sont représentés en entier *)
  mutable root_chunk : chunk;  (* Représente le bloc fonction principal *)
}
```

tel que voila le type chunk (block)

```
    type chunk = {
  mutable constants : constant list; (* couple de (const_type,data) *)
  mutable instructions : instruction list;
  mutable protos : chunk list;
  mutable name_chunk : string;
  mutable frst_line : int;
  mutable last_line : int;
  mutable numUpvals : int;
  mutable numParams : int;
  mutable isVarg : bool;
```

```
  mutable maxStack : int;
  mutable upvalues : string list;
  mutable lineNums : int list;
  mutable locals : local list; (* triplet de (name_local, start, end_pc *)
}
```

Cette structure contient directement le bytecode du programme Lua et permet de stocker les informations essentielles du fichier bytecode pour son interprétation par la machine virtuelle.

2.3 Représentation des Instructions Bytecode

Les instructions dans le bytecode Lua 5.1 sont classées en trois types :

```
type instruction = {
  instr_type : instruction_type;  (* Type d'instruction : ABC, ABx, AsBx *)
  name_instr : string;                  (* Nom de l'instruction *)
  mutable opcode : int option;    (* Opcode associé, mutable car modifiable *)
  mutable a : int option;         (* Champ A (optionnel) *)
  mutable b : int option;         (* Champ B (optionnel) *)
  mutable c : int option;         (* Champ C (optionnel) *)
}
```

et ça est déduis par :

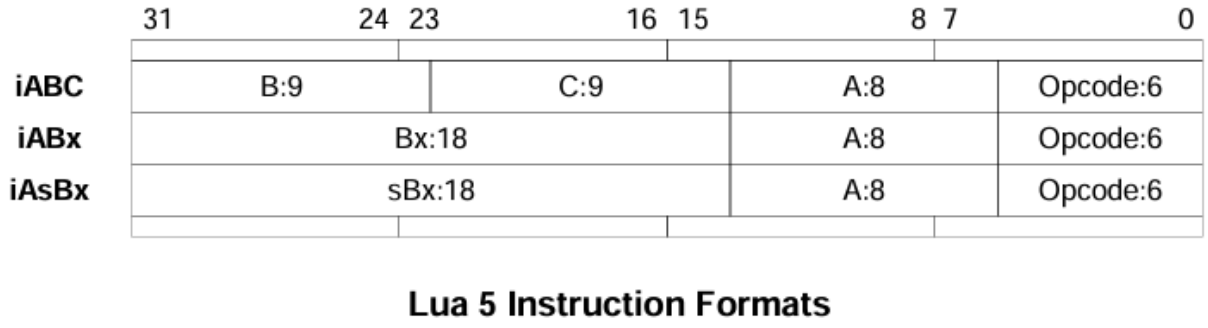

Figure 1: lua instruction format

Cette structure permet de modéliser les différentes instructions du bytecode et de faciliter leur décodage et leur interprétation par la machine virtuelle.

Un exemple d'instructions bytecode pour un programme simple :

```
LOADK   0   0    -- Charge la constante 0 dans le registre 0
LOADK   1   1    -- Charge la constante 1 dans le registre 1
ADD     2   0   1  -- Additionne les registres 0 et 1 et stocke le résultat dans 2
CALL    3   2   1  -- Appelle la fonction stockée dans le registre 3 avec 2 arguments
RETURN  0   1    -- Termine l'exécution
```

Chaque opcode joue un rôle spécifique dans l'exécution du programme et est traité par la machine virtuelle Lua. Pour plus de détails, vous pouvez consulter la définition des opcodes sur le site officiel de Lua : Définition des opcodes Lua 5.1.

3 Architecture et Conception de la Machine Virtuelle

La machine virtuelle (VM) que nous avons implémentée est conçue pour interpréter et exécuter le bytecode Lua 5.1. Son architecture repose sur plusieurs composants essentiels permettant la gestion des instructions, de la pile d'exécution et des variables globales.

3.1 Structure Générale de la VM

La VM est construite autour des éléments suivants :

- **Une pile d'exécution** pour stocker les valeurs intermédiaires et les variables locales.
- **Une table des variables globales** stockant les fonctions et valeurs accessibles globalement.
- **Un compteur de programme (PC)** indiquant l'instruction en cours d'exécution.
- **Un interpréteur d'instructions** capable d'exécuter les 38 opcodes du bytecode Lua 5.1.

3.2 Définition des Types de Données

Pour structurer la VM, nous avons défini plusieurs types :

```
type table = (value, value) Hashtbl.t

and func = {
  chunk : chunk;
  globals_function : (string, value) Hashtbl.t;
}

and value =
  | Const of constant
```

```
  | Table of table
  | Func of func

type vm_state = {
  mutable stack : value array;  (* Pile d'exécution *)
  mutable globals : (string, value) Hashtbl.t; (* Variables globales *)
  mutable pc : int; (* Compteur de programme *)
  mutable chunk : chunk; (* Chunk contenant le code et les constantes *)
  mutable upvalues : (int, value) Hashtbl.t; (* Upvalues *)
}
```

3.3 Initialisation et Exécution de la VM

La machine virtuelle est initialisée avec une pile vide et un environnement global contenant la fonction `print` :

```
let init_vm =
  let vm = {
        stack = Array.make 10 (Const { type_const = NIL; data = "" });
        globals = Hashtbl.create 10;
        pc = 0;
        chunk = create_chunk();
        upvalues = Hashtbl.create 10;
      } in
  Hashtbl.add vm.globals "print" (Const { type_const = STRING; data = "print" });
  vm
```

L'exécution suit une boucle où chaque instruction est traitée successivement :

```
and run_vm vm chunk =
  vm.chunk <- chunk;
  vm.pc <- 0;
  while vm.pc < List.length chunk.instructions do
    let instr = List.nth chunk.instructions vm.pc in
    execute_instruction vm instr;
  done
```

3.4 Gestion des Instructions

Chaque instruction est analysée et exécutée en fonction de son opcode. Voici un exemple de gestion d'instructions :

```
let rec execute_instruction vm instr =
  match instr.name_instr with
  | "MOVE" -> (
    match instr.a, instr.b with
    | Some a, Some b ->
        vm.stack.(a) <- vm.stack.(b);  (* R(A) := R(B) *)
        vm.pc <- vm.pc + 1
    | _ -> failwith "MOVE : paramètres invalides"
  )
```

L'exécution continue jusqu'à ce que l'instruction `RETURN` soit rencontrée ou qu'une erreur survienne.

3.5 Gestion des Upvalues

Les *upvalues* sont gérées via une table de hachage :

```
let get_upvalues vm chunk =
  let upvalues_table = Hashtbl.create 100 in
  List.iter (fun local ->
    let value = get_upvalue vm chunk local in
    Hashtbl.add upvalues_table local.name_local value;
  ) chunk.locals;
  upvalues_table
```

4 Implémentation et Déroulement du Projet

L'implémentation de la machine virtuelle pour le bytecode Lua 5.1 repose sur plusieurs étapes : le chargement des fichiers, la désérialisation (undump), l'exécution et la sérialisation (dump). Cette section détaille ces processus et leur intégration dans le projet.

4.1 Chargement des Fichiers Bytecode

Les fichiers `.luac` sont lus à partir du répertoire `test/` ou via un argument en ligne de commande. Le programme commence par vérifier l'existence des répertoires de sortie `resultat_dump` et `resultat_undump`, puis les crée si nécessaire. Ensuite, il identifie les fichiers à traiter :

- Si un fichier `.luac` est fourni en argument, seul ce fichier est traité.
- Si aucun argument n'est passé, le programme parcourt le répertoire `test/` et sélectionne tous les fichiers `.luac`.
- Si le répertoire `test/` n'existe pas, une erreur est affichée.

4.2 Désérialisation du Bytecode (Undump)

Le processus d'undump consiste à convertir le fichier binaire `.luac` en une structure exploitable par la machine virtuelle. Cela inclut l'extraction des métadonnées du bytecode ainsi que la récupération des instructions, constantes et fonctions internes.

L'en-tête du fichier est d'abord analysé pour extraire les informations essentielles :

```
let decode_bytecode bytecode =
  let lua_undump = create_lua_undump bytecode in
  lua_undump.vm_version <- _get_byte lua_undump;
  lua_undump.bytecode_format <- _get_byte lua_undump;
  lua_undump.big_endian <- (_get_byte lua_undump = 0);
  lua_undump.int_size <- _get_byte lua_undump;
  lua_undump.size_t <- _get_byte lua_undump;
  lua_undump.instr_size <- _get_byte lua_undump;
  lua_undump.l_number_size <- _get_byte lua_undump;
  lua_undump.integral_flag <- _get_byte lua_undump;
  (* Décodage du chunk principal *)
  lua_undump.root_chunk <- decode_chunk lua_undump;
  lua_undump
```

Une fois l'en-tête décodé, la structure principale du programme est reconstituée à partir du bytecode :

```
let rec decode_chunk lua_undump =
    let chunk = create_chunk () in
    let size = _get_size_t lua_undump in
    chunk.name_chunk <- _get_string lua_undump size;
    chunk.frst_line <- _get_uint lua_undump;
    chunk.last_line <- _get_uint lua_undump;
    chunk.numUpvals <- _get_byte lua_undump;
    chunk.numParams <- _get_byte lua_undump;
    chunk.isVarg <- (_get_byte lua_undump) <> 0;
    chunk.maxStack <- _get_byte lua_undump;

    (* Extraction des instructions *)
    let num_instr = _get_uint lua_undump in
    for _ = 1 to num_instr do
      let instr = decode_instr (_get_uint32 lua_undump) in
      append_instruction chunk instr
    done;
```

```
    (* Extraction des constantes, prototypes, var local etc*)
```

Cette fonction permet de reconstituer entièrement la structure du programme Lua compilé à partir du bytecode, rendant ainsi son exécution possible sur la machine virtuelle.

4.3 Sérialisation du Bytecode (Dump)

Après exécution, il est possible de générer un nouveau fichier bytecode à partir du programme chargé en mémoire. Cette opération est réalisée par la fonction `dump`, qui reconstruit un fichier `.luac`.

```
let dump dump =
  dump_header dump;
  dump_chunk dump dump.root_chunk;
  dump.bytecode
```

Le bytecode résultant est ensuite sauvegardé dans le répertoire `resultat_dump`, permettant une réutilisation ou une comparaison avec le fichier original pour vérifier la justesse de la structure produite par l'Undump.

4.4 Exécution du Bytecode sur la Machine Virtuelle

Une fois le bytecode chargé et désérialisé, il est exécuté par la machine virtuelle. La VM utilise une pile d'exécution et un environnement global pour stocker les variables globales et les fonctions prédéfinies. L'exécution suit un modèle itératif où chaque instruction est analysée et exécutée en fonction de son opcode.

```
and run_vm vm chunk =
  vm.chunk <- chunk;
  vm.pc <- 0;
  while vm.pc < List.length chunk.instructions do
    let instr = List.nth chunk.instructions vm.pc in
    execute_instruction vm instr;
  done
```

L'instruction `execute_instruction` gère les opcodes et met à jour la pile d'exécution en fonction de l'opération à effectuer.

```
let rec execute_instruction vm instr =
  match instr.name_instr with
  | "MOVE" -> (
```

```
    match instr.a, instr.b with
    | Some a, Some b ->
        vm.stack.(a) <- vm.stack.(b);  (* R(A) := R(B) *)
        vm.pc <- vm.pc + 1
    | _ -> failwith "MOVE : paramètres invalides"
  )
```

4.5 Gestion des Résultats et Affichage

Les résultats de l'exécution sont affichés sur la console sous un format structuré, et les informations détaillées sont stockées dans des fichiers texte dans `resultat_undump` et `resultat_dump` . Chaque fichier `.luac` traité génère :

- Un fichier contenant l'undump du bytecode (représentation textuelle du programme désassemblé).
- Un fichier contenant le bytecode reconstitué après transformation.
- Une exécution affichée directement dans la console.

5 Difficultés et Optimisations Possibles

L'implémentation de la machine virtuelle Lua 5.1 a présenté plusieurs défis techniques, principalement liés à la gestion du bytecode, l'interprétation des instructions et l'optimisation des performances. Cette section aborde les principales difficultés rencontrées ainsi que les pistes d'amélioration possibles.

5.1 Difficultés Rencontrées

- **Gestion des instructions** : La diversité des opcodes et la nécessité de respecter le format d'exécution exact ont exigé une attention particulière lors de l'implémentation des instructions de la VM.
- **Manipulation de la pile d'exécution** : Assurer la gestion correcte des registres et des appels de fonctions a nécessité plusieurs itérations pour éviter des erreurs de segmentation ou des incohérences d'exécution.
- **Optimisation des accès mémoire** : Certaines structures de données ont dû être adaptées pour réduire les allocations inutiles et améliorer l'efficacité de l'exécution.

5.2 Optimisations Possibles

Plusieurs améliorations peuvent être envisagées pour optimiser l'exécution de la machine virtuelle:

- **Optimisation de la gestion de la mémoire** : Utiliser une allocation plus efficace pour les registres et la pile d'exécution afin de réduire les accès coûteux en mémoire.
- **Compilation Juste-à-Temps (JIT)** : Intégrer un moteur JIT permettrait d'améliorer considérablement les performances en traduisant dynamiquement le bytecode en instructions natives du processeur.
- **Multithreading** : Bien que Lua soit conçu pour être exécuté dans un seul thread, l'exploitation du parallélisme pourrait améliorer l'efficacité de certaines opérations.
- **Optimisation des structures de données** : L'utilisation de tables de hachage plus performantes et la réduction des copies inutiles de données pourraient accélérer le traitement des instructions.
- **Ajout d'un mode de débogage avancé** : Une meilleure visualisation de la pile d'exécution et des registres faciliterait l'identification des erreurs et l'optimisation du code généré.

6 Conclusion

L'implémentation de cette machine virtuelle Lua 5.1 a permis d'explorer en profondeur le fonctionnement d'un interpréteur de bytecode et les défis liés à l'exécution d'un langage compilé en bytecode. À travers ce projet, nous avons pu :

- Comprendre la structure et le format du bytecode Lua 5.1.
- Développer un module d'undump pour désérialiser le bytecode et en extraire les instructions et constantes.
- Mettre en place une VM capable d'interpréter et d'exécuter ces instructions en respectant le comportement attendu.
- Générer un bytecode modifié via la sérialisation (dump) pour tester la reconstitution d'un programme Lua.

En conclusion, ce projet nous a offert une expérience enrichissante en matière d'analyse de bytecode, de conception d'interpréteur et d'optimisation des performances dans un environnement bas niveau.

References

[1] Les cours sur moodle : https://moodle-sciences-24.sorbonne-universite.fr/course/view.php?id=4405

[2] Visual studio IDE : https://visualstudio.microsoft.com/fr/downloads//

[3] LaTeX pour le rapport : https://fr.overleaf.com/project

[4] Spécifications du bytecode Lua 5.1 : https://www.mcours.net/cours/pdf/hasclic3/hassclic818.pdf

[5] Blog post implémentant un parser pour bytecode Lua en Python: https://github.com/CPunch/LuaPytecode

[6] Définition des opcodes dans le code source de Lua 5.1: https://www.lua.org/source/5.1/lopcodes.h.html

[7] Site explicant Qu'est-ce qu'un fichier LUA et quelques autre détails : https://docs.fileformat.com/fr/programming/lua/

[8] Site pour comprendre mieux les structures utiliser pour Lua : https://www.lua.org/manual/5.1/manual.html