



24/03/2025

Rapport Projet VM pour bytecode LUA 5.1

Compilation avancée



MARSSO Denn – MOUGAMADOUBOUGARY
Mohamed
SORBONNE UNIVERSITE – STL

***Pour tester la code il taper dans le terminal « make all » et ensuite
« ./headerParser <nomDuFichier.out> »***

Introduction

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'interprète Lua est écrit en C et est hautement portable, en plus d'être extrêmement compact (moins de 200ko). Il peut donc être intégré virtuellement à 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.

Le but de ce projet est de programmer un code qui permet l'évaluation d'un bytecode Lua.

Pour cela il faut passer par 4 étapes : le parsing (analyse) du bytecode, l'affichage, le dumping du bytecode (Afin de voir la liste d'instructions, nécessaire pour la dernière étape) et l'implémentation de la VM qui interprète le bytecode parsé.

Nous avons décidé d'implémenter tout cela en C et nous sommes arrivés jusqu'à l'interprétation de fonctions (et du print()).

Nos 2 machines sont en Little Endian et le byte code Lua aussi, nous n'avons donc eu aucun problème avec cela.

Nous allons donc expliquer dans ce rapport comment nous avons implémenté ces différentes étapes et pourquoi nous avons fait les choix qui nous ont mené à cette implémentation.

Parsing du bytecode

Pour le parsing nous nous sommes inspirés de l'implémentation du parseur en python (voir ressources en annexe), nous avons ensuite lu la documentation Lua pour comprendre comment analyser le byte code.

Le byte code est composé d'un header et de plusieurs blocs de fonctions (qu'on appelle *chunk*).

Pour l'analyser il faut donc lire un à un chaque élément qui compose le byte code en commençant par le header.

Header block of a Lua 5 binary chunk	
Default values shown are for a 32-bit little-endian platform with IEEE 754 doubles as the number format. The header size is always 12 bytes.	
4 bytes	Header signature: ESC, "Lua" or 0x1B4C7561 • Binary chunk is recognized by checking for this signature
1 byte	Version number, 0x51 (81 decimal) for Lua 5.1 • High hex digit is major version number • Low hex digit is minor version number
1 byte	Format version, 0=official version
1 byte	Endianness flag (default 1) • 0=big endian, 1=little endian
1 byte	Size of int (in bytes) (default 4)
1 byte	Size of size_t (in bytes) (default 4)
1 byte	Size of Instruction (in bytes) (default 4)
1 byte	Size of lua_Number (in bytes) (default 8)
1 byte	Integral flag (default 0) • 0=floating-point, 1=integral number type
On an x86 platform, the default header bytes will be (in hex): 1B4C7561 51000104 04040800	

Figure 1 : Parsing de l'en-tête

L'en-tête (header) est un bloc statique, il ne change jamais, il est donc simple de le parser. Il faut simplement lire chaque élément de l'en-tête dans le fichier. On a donc utilisé la fonction `fread()` de c en lui donnant comme paramètre la taille de l'élément. Il faut juste dans un premier temps récupérer la signature et la vérifier, pour cela on va lire 4 octets dans le fichier et comparer le résultat reçu avec la signature Lua qu'on a stocké dans notre code en variable globale. On ne continue à parser que si la signature est bonne.

Une fois cela fait il suffit de lire byte par byte chaque élément du header ce qui donne le code de la *Figure 2*.

```
// Lire et vérifier la signature Lua
char signature[4];
fread(signature, 1, 4, file);
if (memcmp(signature, LUA_SIGNATURE, 4) != 0) {
    printf("Fichier non valide : ce n'est pas un bytecode Lua.\n");
    fclose(file);
    return 1;
}

// Lire l'en-tête du fichier
uint8_t version = read_byte(file);
uint8_t format = read_byte(file);
uint8_t endianness = read_byte(file);
uint8_t int_size = read_byte(file);
uint8_t size = read_byte(file);
uint8_t instr_size = read_byte(file);
uint8_t number_size = read_byte(file);
uint8_t integral_flag = read_byte(file);
```

Figure 2 : Code pour analyser l'en-tête

Après le header le byte code contient le premier *chunk* (bloc de fonctions).

The header block is followed immediately by the top-level function or chunk:

Function block of a Lua 5 binary chunk	
Holds all the relevant data for a function. There is one top-level function.	
String	source name
Integer	line defined
Integer	last line defined
1 byte	number of upvalues
1 byte	number of parameters
1 byte	is_vararg flag (see explanation further below) <ul style="list-style-type: none">• 1=VARARG_HASARG• 2=VARARG_ISVARARG• 4=VARARG_NEEDSARG
1 byte	maximum stack size (number of registers used)
List	list of instructions (code)
List	list of constants
List	list of function prototypes
List	source line positions (optional debug data)
List	list of locals (optional debug data)
List	list of upvalues (optional debug data)

Figure 3 : Parsing des chunks

Le *chunk* contrairement au header est dynamique car les tailles des éléments à l'intérieur peuvent varier. De plus il peut y avoir plusieurs *chunks*, il y en a autant que de nombres de fonctions dans le code Lua.

Le premier chunk correspond toujours au main (s'il n'y a pas de fonctions dans le code il n'y aura que le main). La seule différence entre le *chunk* du main et le *chunk* des autres fonctions est que les paramètres *line defined*, *last line defined*, *number of upvalues* et *number of parameters* sont toujours nuls pour le main.

Chaque *chunk* est malgré tout composé d'une partie statique (de *line defined* à *maximum stack size*).

```
// Structure d'un chunk (fonction Lua compilée)
typedef struct Chunk Chunk;

struct Chunk {
    char *name;
    uint32_t first_line;
    uint32_t last_line;
    uint8_t numUpvals;
    uint8_t numParams;
    bool isVarg;
    uint8_t maxStack;

    Instruction instructions[MAX_INSTRUCTIONS];
    uint32_t instruction_count;

    Constant constants[MAX_CONSTANTS];
    uint32_t constant_count;

    Local locals[MAX_LOCALS];
    uint32_t local_count;

    Upvalue upvalues[MAX_UPVALUES];
    uint32_t upvalue_count;

    uint32_t source_lines[MAX_LINES];
    uint32_t line_count;

    Chunk *protos[MAX_PROTOS];
    uint32_t proto_count;
};
```

Figure 4 : Structure de chunk

Le parsing d'un chunk est donc en 3 parties, parsing du *source name*, parsing de la partie statique et parsing de la partie dynamique avec toutes les listes.

Parsing du *source name*

Le *source name* est une chaîne de caractère correspondant au nom du fichier, sa taille varie donc en fonction du nom du fichier. Dans la *Figure 5*, on peut voir comment dans Lua, les chaînes de caractères sont encodées.

A **String** is defined in this way:

All strings are defined in the following format:	
Size_t	String data size
Bytes	String data, includes a NUL (ASCII 0) at the end
The string data size takes into consideration a NUL character at the end, so an empty string ("") has 1 as the size_t value. A size_t of 0 means zero string data bytes; the string does not exist. This is often used by the source name field of a function.	

Figure 5 : Structure de chunk

MARSSO Denn

MOUGAMADOUBOUGARY Mohamed

On a donc décidé d'écrire une fonction *read_string*, qui permet de lire une chaîne de caractère en se basant sur cette documentation.

```
// Fonction pour lire une chaîne de caractères
char* read_string(FILE *file) {
    size_t string_length;
    if (fread(&string_length, sizeof(size_t), 1, file) != 1) {
        fprintf(stderr, "Error reading string length\n");
        return "";
    }

    char* string;
    string = malloc(string_length + 1);
    if (!string) {
        fprintf(stderr, "Memory allocation string\n");
        return "";
    }

    if (fread(string, sizeof(char), string_length, file) != string_length) {
        fprintf(stderr, "Error reading string\n");
        free(string); // Libérer la mémoire en cas d'erreur
        return "";
    }
    string[string_length] = '\0'; // Ajouter le caractère nul de fin

    return string;
}
```

Figure 6 : Fonction *read_string*

Cette fonction va donc récupérer la taille de la chaîne de caractère dans un *size_t*, allouer la taille correspondante puis lire la chaîne de caractère dans l'espace alloué. Elle ajoute aussi le caractère nul à la fin de la chaîne pour garantir sa compatibilité. Pour parser le *source name* il suffit donc d'appeler *read_string* et de l'associer au *champ name* de notre *chunk*.

Parsing de la partie statique du *chunk*

Pour la partie statique du chunk, on a suivi la même méthode que pour le header à la différence que dans le chunk il y'a des entiers. On a donc créé une fonction *read_uint32* qui fait un *fread* dans un *int32* puis on a appelé les fonctions de lecture correspondantes au type de chaque élément pour tous les éléments statiques du chunk.

```
// Fonction pour charger un chunk Lua
✓ Chunk* load_chunk(FILE *file) {
    Chunk *chunk = malloc(sizeof(Chunk));
    chunk->name = read_string(file);
    chunk->first_line = read_uint32(file);
    chunk->last_line = read_uint32(file);
    chunk->numUpvals = read_byte(file);
    chunk->numParams = read_byte(file);
    chunk->isVarg = read_byte(file) != 0;
    chunk->maxStack = read_byte(file);
}
```

Figure 7 : Fonction *load_chunk*

Parsing de la partie dynamique du chunk

Vient ensuite la partie dynamique du chunk qui est la plus compliquée du parsing. Dans cette partie chaque élément à sa propre définition, nous les avons donc toutes récupérer dans la doc Lua puis avons parsé chaque élément un par un. Les instructions sont composées d'une taille de liste puis d'une liste d'instructionset les instructions qui sont encodés dans un format précis, voir *Figure 8*.

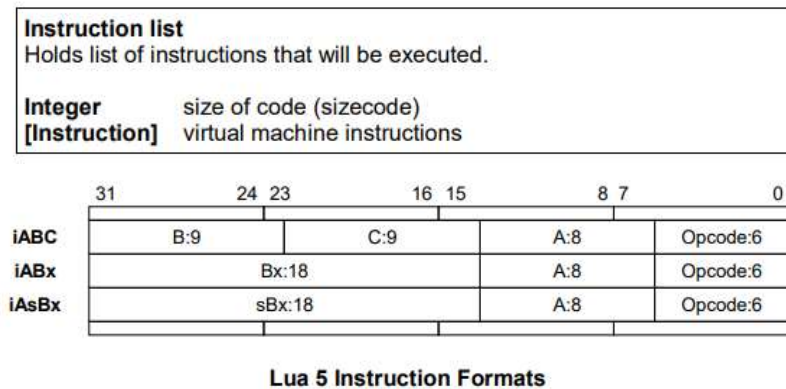


Figure 8 : Instructions

Nous avons donc créé une structure Instruction correspondante et en avons ajouté une liste dans le chunk avec la taille de cette liste.

```
// Types d'instructions Lua
typedef enum {
    ABC,
    ABx,
    AsBx
} InstructionType;

// Structure d'une instruction
typedef struct {
    InstructionType type;
    char name[16];
    uint8_t opcode;
    uint8_t A;
    uint16_t B;
    uint16_t C;
    uint32_t Bx;
} Instruction;
```

```
Instruction instructions[MAX_INSTRUCTIONS];
uint32_t instruction_count;
```


Figure 9 : Structure des instructions

Pour la parser ensuite il ne reste plus qu'à récupérer la taille n de la liste puis faire une boucle qui va récupérer n fois 32b puis les décoder :

```
chunk->instruction_count = read_uint32(file);
for (uint32_t i = 0; i < chunk->instruction_count; i++) {
    uint32_t instr_data = read_uint32(file);
    chunk->instructions[i] = decode_instruction(instr_data);
}
```

Pour décoder chaque instruction il faut ensuite se référer au format donné dans la documentation et associer les bons bits à chaque variable de notre structure Instruction :

```
// Fonction de décodage d'une instruction
Instruction decode_instruction(uint32_t data) {
    Instruction instr;
    instr.opcode = data & 0x3F;
    instr.A = (data >> 6) & 0xFF;
    instr.C = (data >> 14) & 0x1FF;
    instr.B = (data >> 23) & 0x1FF;
    instr.Bx = (data >> 14);
    return instr;
}
```

A cela, il faut rajouter une liste d'op code :

```
// Table des noms des opcodes
const char *opcode_names[] = {
    "MOVE", "LOADK", "LOADBOOL", "LOADNIL", "GETUPVAL", "GETGLOBAL", "GETTABLE",
    "SETGLOBAL", "SETUPVAL", "SETTABLE", "NEWTABLE", "SELF", "ADD", "SUB", "MUL",
    "DIV", "MOD", "POW", "UNM", "NOT", "LEN", "CONCAT", "JMP", "EQ", "LT", "LE",
    "TEST", "TESTSET", "CALL", "TAILCALL", "RETURN", "FORLOOP", "FORPREP", "TFORLOOP",
    "SETLIST", "CLOSE", "CLOSURE", "VARARG"
};
```

Qui nous servira à afficher le nom de l'op code correspondant au code (car dans Lua c'est un entier).

Constant list	
Holds list of constants referenced in the function (it's a constant pool.)	
Integer	size of constant list (sizek)
[
1 byte	type of constant (value in parentheses): <ul style="list-style-type: none">• 0=LUA_TNIL, 1=LUA_TBOOLEAN,• 3=LUA_TNUMBER, 4=LUA_TSTRING
Const	the constant itself: this field does not exist if the constant type is 0; it is 0 or 1 for type 1; it is a Number for type 3, or a String for type 4.
]	

Figure 10 : Constantes

Chaque constante est composée d'un octet définissant son type et de la constante correspondante. Les constantes sont donc composées d'un entier définissant la taille de la liste et d'une liste de constantes. Nous avons donc créé la structure correspondante et l'avons ajouté dans le chunk.

```
// Structure d'une constante
typedef struct {
    uint8_t type;
    union {
        double number;
        char *string;
        bool boolean;
    } value;
} Constant;

Constant constants[MAX_CONSTANTS];
uint32_t constant_count;
```

Figure 11 : Structure des constantes

Afin de pouvoir représenter les différentes constantes possibles nous avons utilisé une union qui contient tous les types de constante possible. Le parsing se déroule de la même manière que pour les instructions à la différence qu'au lieu d'avoir un décodage nous avons une gestion des cas qui va pour chaque type de constante lire une variable différente dans le byte code et l'associer à la constante du chunk.

```
chunk->constant_count = read_uint32(file);
for (uint32_t i = 0; i < chunk->constant_count; i++) {
    chunk->constants[i].type = read_byte(file);
    if (chunk->constants[i].type == 3) {
        fread(&chunk->constants[i].value.number, sizeof(double), 1, file);
    } else if (chunk->constants[i].type == 4) {
        chunk->constants[i].value.string = read_string(file);
    } else if (chunk->constants[i].type == 1) {
        chunk->constants[i].value.boolean = read_byte(file) != 0;
    }
}
```

Figure 13 : Récupération des constantes

Function prototype list	
Holds function prototypes defined within the function.	
Integer	size of function prototypes (sizep)
[Functions]	function prototype data, or function blocks

Figure 14 : Functions prototypes

Les prototypes de fonctions sont représentés par un entier qui correspond à la taille de la liste (nombre de fonctions) et par une liste de fonctions. Ces fonctions sont chacune représentée par un chunk, c'est pour cela que nous avons défini la structure chunk afin de supporter la récursivité.

```
Chunk *protos[MAX_PROTOS];
uint32_t proto_count;
```

Pour les prototypes de fonctions c'est toujours la même chose que pour constantes et instructions, sauf que pour chaque prototype on va rappeler la méthode loadChunk (qui charge tout le fonction block) et mettre le résultat dans le prototype de fonction du chunk principal correspondant.

```
chunk->proto_count = read_uint32(file);
for (uint32_t i = 0; i < chunk->proto_count; i++) {
    chunk->protos[i] = load_chunk(file);
}
```

Source line position list	
Holds the source line number for each corresponding instruction in a function. This information is used by error handlers or debuggers. In a stripped binary, the size of this list is zero. The execution of a function does not depend on this list.	
Integer	size of source line position list (sizelineinfo)
[Integer]	list index corresponds to instruction position; the integer value is the line number of the Lua source where the instruction was generated

Upvalue list	
Holds list of upvalue names.	
Integer	size of upvalue list (sizeupvalues)
[String]	name of upvalue

Figure 15 : UpValues et Source Lines

MARSSO Denn

MOUGAMADOUBOUGARY Mohamed

Pour les upvalues et les source lines la structure est quasiment la même, ils sont composés d'un entier définissant la taille de la liste puis d'une liste d'entier (pour source lines) ou de chaîne de caractères (pour up values).

Le parsing est donc le même que pour les autres éléments dynamiques, sauf qu'on lit directement un entier ou une chaîne de caractère et qu'on met le résultat dans la variable du chunk correspondant.

```
chunk->line_count = read_uint32(file);
for (uint32_t i = 0; i < chunk->line_count; i++) {
    chunk->source_lines[i] = read_uint32(file);
}
```

```
chunk->upvalue_count = read_uint32(file);
for (uint32_t i = 0; i < chunk->upvalue_count; i++) {
    chunk->upvalues[i].name = read_string(file);
}
```

Figure 16 : Récupération des upValues et source lines

Local list

Holds list of local variable names and the program counter range in which the local variable is active.

Integer	size of local list (sizelocvars)
[
String	name of local variable (varname)
Integer	start of local variable scope (startpc)
Integer	end of local variable scope (endpc)
]	

Figure 17 : Variables locales

Chaque variable locale est composée d'un nom de variable puis de deux entiers représentant le début du stack pointer (dans la pile) et la fin du stack pointer. Les variables locales sont donc représentées par un entier définissant la taille de la liste et une liste de variable local.

```
// Structure d'une variable locale
typedef struct {
    char *name;
    uint32_t start_pc;
    uint32_t end_pc;
} Local;
```

```
Local locals[MAX_LOCALS];
uint32_t local_count;
```

On va donc de la même manière que pour les autres éléments dynamiques lire autant de fois

MARSSO Denn

MOUGAMADOUBOUGARY Mohamed

qu'il n'y a d'éléments dans la liste une chaîne de caractère et 2 entiers puis les assigner à la variable locale correspondante.

```
chunk->local_count = read_uint32(file);
for (uint32_t i = 0; i < chunk->local_count; i++) {
    chunk->locals[i].name = read_string(file);
    chunk->locals[i].start_pc = read_uint32(file);
    chunk->locals[i].end_pc = read_uint32(file);
}
```

Figure 18 : Récupération des variables locales

Affichage

Une fois toutes les parties dynamiques parsées il est important d'afficher le résultat obtenu. Nous avons donc parser les parties dynamiques une par une et à chaque fois nous l'affichions pour voir si le résultat était bon. Pour vérifier le résultat nous comparions le résultat obtenu avec le résultat du parser python donné en ressources.

Les figures ci-dessous montrent les affichages des différentes parties.

```
printf("HEADER \n");
printf("\n");
printf("Lua Version: %.1f\n", version / 16.0);
printf("Format: %d\n", format);
printf("Endianness: %s\n", endianness ? "Little" : "Big");
printf("Integer Size: %d\n", int_size);
printf("Size_t Size: %d\n", size);
printf("Instruction Size: %d\n", instr_size);
printf("Lua Number Size: %d\n", number_size);
printf("Integral Flag: %d\n", integral_flag);
printf("\n");
```

Figure 19 : Affichage de l'en-tête

```
printf("STATIC FUNCTION BLOCK\n");
printf("\n");
printf("Source Name: %s\n", chunk->name ? chunk->name : "(none)");
printf("Line Defined: %u\n", chunk->first_line);
printf("Last Line Defined: %u\n", chunk->last_line);
printf("Number of Upvalues: %u\n", chunk->numUpvals);
printf("Number of Parameters: %u\n", chunk->numParams);
printf("Is Vararg Flag: %u\n", chunk->isVarg);
printf("Maximum Stack Size: %u\n", chunk->maxStack);
printf("Number of Instructions: %u\n", chunk->instruction_count);
printf("Number of Constants: %u\n", chunk->constant_count);
printf("Number of Locals: %u\n", chunk->local_count);
printf("Number of Upvalues: %u\n", chunk->upvalue_count);
printf("Number of Source Lines: %u\n", chunk->line_count);
printf("\n");
```

Figure 20 : Affichage du bloc de la fonction statique

```
// Afficher Les instructions
printf("\n=== PARSING de %s ===\n", chunk->name ? chunk->name : "<main>");
for (uint32_t i = 0; i < chunk->instruction_count; i++) {
    printf("[%3d] OP: %2d A: %3d B: %3d C: %3d\n", i, chunk->instructions[i].opcode,
           chunk->instructions[i].A, chunk->instructions[i].B, chunk->instructions[i].C);
}

// Afficher Les constantes
printf("\n=== Constantes ===\n");
for (uint32_t i = 0; i < chunk->constant_count; i++) {
    printf("Const[%3d] Type: %d\n", i, chunk->constants[i].type);
    switch (chunk->constants[i].type) {
        case 0: printf("Constant does not exist\n"); break;
        case 1: printf("Boolean: %s\n", chunk->constants[i].value.boolean ? "true" : "false"); break;
        case 3: printf("Number: %f\n", chunk->constants[i].value.number); break;
        case 4: printf("String: %s\n", chunk->constants[i].value.string); break;
        default: printf("Unknown type\n"); break;
    }
}

// Afficher Les variables Locales
printf("\n=== Locals ===\n");
for (uint32_t i = 0; i < chunk->local_count; i++) {
    printf("Local[%3d] Name: %s, Start: %u, End: %u\n", i, chunk->locals[i].name, chunk->locals[i].start_pc, chunk->locals[i].end_pc);
}

// Afficher Les upvalues
printf("\n=== Upvalues ===\n");
for (uint32_t i = 0; i < chunk->upvalue_count; i++) {
    printf("Upvalue[%3d] Name: %s\n", i, chunk->upvalues[i].name);
}

// Afficher Les lignes sources
printf("\n=== Source Lines ===\n");
for (uint32_t i = 0; i < chunk->line_count; i++) {
    printf("Line[%3d]: %u\n", i, chunk->source_lines[i]);
}
```

Figure 21: Affichage des parties dynamiques

Et ensuite, pour chaque fonction nous avons appliqué le même code d’affichage car une fonction correspond à une fonction block.

Dumping

L’affichage permet de voir en brute les résultats du parseur, mais pour les instructions il n’est pas suffisant. Afin de pouvoir bien comprendre les instructions générées par le parseur (pour pouvoir les évaluer avec la VM), il faut les dumper sous un format textuel lisible et compréhensible.

```
// Fonction pour dumper une instruction
void dump_instruction(Instruction instr) {
    printf("Instruction: %10s A: %d B: %d C: %d\n",
           opcode_names[instr.opcode], instr.A, instr.B, instr.C);
}
```

Nous avons donc fait une fonction dump instruction qui pour une instruction affiche l’instruction correspondant au opcode grâce à la liste d’opcode names avec la valeur de chaque élément de l’instruction (A, B, C et Bx). Cela permet donc d’avoir un tableau avec chaque instruction ce qui facilite beaucoup l’interprétation via la VM.

VM

Pour la VM on va récupérer le nombre d'instructions, puis pour chaque instruction récupérer le code OP puis traiter chaque code au cas par cas. Pour cela nous avons un grand switch case sur les codes OP.

Il y a 37 codes OP sur Lua, nous ne les avons pas tous implémentés.

Nous avons d'abord implémenté les plus importants (nécessaires pour toutes les opérations) LOADK/0, MOVE/1, GETGLOBAL/5 (pour les fonctions natives) et RETURN/30.

Nous avons ensuite rajouté les opérations arithmétiques basiques ADD/12, SUB/13, MUL/14 et DIV/15.

Puis finalement les opcode nécessaires aux appels de fonctions et donc au print CALL/28 et CLOSURE/36.

Pour tous les autres cas nous avons un case default qui print que l'op code n'est pas pris en charge. Tout d'abord avant de faire l'évaluation il faut construire la structure de la VM, pour évaluer le code Lua la VM a besoin des registres, des instructions, des constantes et des fonctions.

```
// Structure VM
typedef struct {
    VMValue registers[256];
    Constant *constants;
    Instruction *instructions;
    uint32_t instruction_count;
    Chunk **protos;
    uint32_t proto_count;
} VM;
```

Figure 22 : Structure de la VM

Les registres peuvent avoir différentes valeurs, dans notre cas ça peut être un float (number), une fonction native (print par exemple) ou une fonction (closure).

```
typedef enum {
    VAL_NUMBER,
    VAL_NATIVE,
    VAL_CLOSURE
} ValueType;
```

```
typedef struct {
    ValueType type;
    union {
        double number;
        NativeFunction native;
        Chunk *closure;
    } as;
} VMValue;
```

Figure 23 : Les autres structures pour la VM

On a donc pour chaque valeur un type (parmi ceux au-dessus) puis une union pour représenter la valeur du registre (float, fonction native ou fonction).


```
// Définition des fonctions natives (GETGLOBAL/print)
typedef void (*NativeFunction)(double *args, int n_args);
```

Figure 24 : Fonction Native

Les fonctions natives sont simplement des fonctions qui ont des arguments. On remplit ces différents éléments directement avec les valeurs récupérer par le parser pour initier notre VM :

```
VM vm = {
    .constants = chunk->constants,
    .instructions = chunk->instructions,
    .instruction_count = chunk->instruction_count,
    .protos = chunk->protos,
    .proto_count = chunk->proto_count
};
```

Gestion du PRINT

Pour le print il faut donc créer une liste de fonctions natives, et dans cette liste ajouter le print. Comme mot clé on met "print" et on associe à ce mot la fonction qui fera le print.

```
void native_print(double *args, int n_args) {
    for (int i = 0; i < n_args; i++) {
        printf("%f ", args[i]);
    }
    printf("\n");
}

GlobalFunction globals[] = {
    {"print", native_print}
};

#define NUM_GLOBALS (sizeof(globals) / sizeof(globals[0]))
```

Figure 24 : Print Native

On a donc après ça notre liste de fonctions natives et notre VM est prête pour l'évaluation. Une fois la structure faite on peut commencer par implémenter les op codes basiques et arithmétiques.

Pour les différentes implémentations nous nous sommes basés sur la Définition des opcodes dans le code source de Lua 5.1 (voir ressources).

Voir la fonction **vm_execute** dans le fichier de code.

Pour le move (0) on va récupérer le numéro du registre que l'on copie via instr.A et le numéro du registre dans lequel on veut mettre la valeur via instr.B puis on va copier un registre dans l'autre.

Pour le load (1) on va assigner le type de valeur au registre (toujours number pour un load) puis lui donner la valeur de la constante correspondante. Pour trouver le registre dans lequel on load, on prend instr.A et pour trouver la constante que l'on cherche on prend instr.Bx (suite à la doc Lua). Il faut bien faire attention à bien mettre les bons types pour respecter l'union (.as.number et .value.number).

Pour le GETGLOBAL(5) on va récupérer la constante via instr.Bx, si c'est une chaine de caractère on va regarder dans notre liste de constantes globales (notre liste de fonctions natives) si le nom de la constante correspond à l'une de nos fonctions. Si c'est le cas on met

MARSSO Denn

MOUGAMADOUBOUGARY Mohamed

notre fonction native dans le registre (en mettant le type de valeur à NATIVE), sinon on ne fait rien.

Pour ADD, SUB, MUL et DIV on récupère le registre 1 et le registre 2 (correspondant aux valeurs sur lequel on fait l'opération) sur instr.b et instr.c on effectue l'opération correspondant sur leurs valeurs puis on stocke le résultat dans le registre instr.A avec une valeur de type Number. Pour le return on return. Avec la gestion de ces op codes on peut déjà interpréter un test lua basique qui fait des opérations arithmétiques.

On peut ensuite ajouter les codes OP pour les fonctions.

Gestion du CALL

Voir la fonction *call_prototype* dans le fichier de code.

Pour le call c'est un peu plus compliqué, on récupère la fonction avec le registre[instr.A] et le nombre d'arguments avec le registre [instr.B].

Puis en faisant un décalage de 1 à chaque fois (si A est 5, donc 6,7 etc.), on va récupérer les valeurs des arguments sur les différents registres.

Une fois les valeurs récupérées on aura 2 cas : fonction native et fonction normal. Si c'est une fonction native on appelle la fonction correspondante avec les arguments récupérer.

Si c'est une fonction normale on va appeler notre méthode *call_prototype* qui va récupérer le résultat et le mettre dans le registre instr.A sous forme de number si instr.C est plus grand que 1.

Notre *call_prototype* va instancier sa propre VM et interpréter le *chunk* correspondant à la fonction sur laquelle *call_prototype* a été appelé avec sa VM. Le code d'interprétation est le même, juste cette fois-ci les registres sont chargés avec les arguments et non les constantes (environnement local).

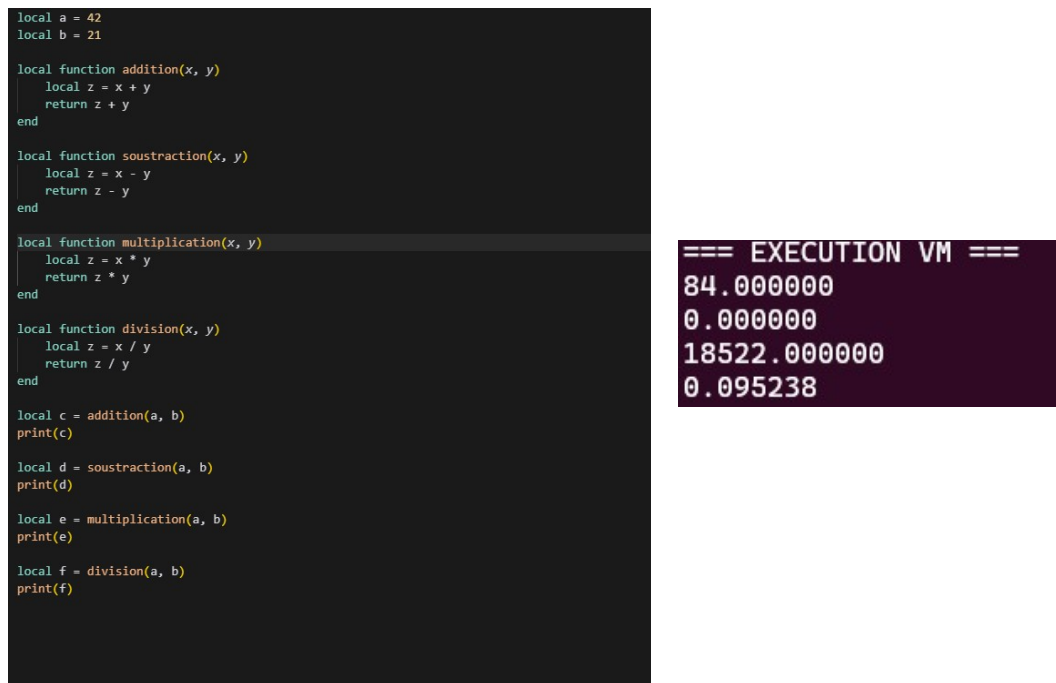
Gestion du CLOSURE

```
case 36:
  vm->registers[instr.A].type = VAL_CLOSURE;
  vm->registers[instr.A].as.closure = vm->protos[instr.Bx];
  break;
```

On charge simplement le chunk de la fonction dans le registre de instr.A

Conclusion

Pour conclure notre code permet bien à partir d'un byte code lua d'évaluer le résultat de ce code, voir figure ci-dessous le code et le résultat.



```
local a = 42
local b = 21

local function addition(x, y)
    local z = x + y
    return z + y
end

local function soustraction(x, y)
    local z = x - y
    return z - y
end

local function multiplication(x, y)
    local z = x * y
    return z * y
end

local function division(x, y)
    local z = x / y
    return z / y
end

local c = addition(a, b)
print(c)

local d = soustraction(a, b)
print(d)

local e = multiplication(a, b)
print(e)

local f = division(a, b)
print(f)
```

```
=== EXECUTION VM ===
84.000000
0.000000
18522.000000
0.095238
```

Figure x : Code Lua à gauche et résultat du programme à droite

Ce qui montre que notre code supporte les opérations arithmétiques, les appels de fonctions et les print. Plusieurs améliorations sont possibles, des améliorations sur le code (comme par exemple la répétition de la vm dans call_prototype au lieu d'une fonction qui serait réutiliser) et des améliorations sur la vm (l'implémentation des opcode manquants).

Ressources

— Spécifications du bytecode Lua 5.1 :

<https://www.mccours.net/cours/pdf/hascllic3/hassscllic818.pdf>

— Blog post implémentant un parser pour bytecode Lua en Python :

<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>