Lua 5.1, Lua et ARéVi

D. Herviou

11 mai 2006

Table des matières

1 Introduction

Ce document effectue un survol rapide et doit permettre une prise en main du langage de programmation Lua. Avant de débuter voici les quelques liens qu'il est souhaitable de consulter pour approndir ce que contient ce document :

- http ://www.lua.org : site officiel lua
- http ://luaforge.net/ : un source forge pour lua
- http ://lua-users.org/ : le site des utilisateurs lua.

Nous allons tout d'abord introduire le langage Lua, puis nous verrons son utilisation en tant que langage d'extension embarqué. En dernière partie nous construirons un exemple permettant dans une application concrête d'intégrer du code lua.

Lua est un langage de prototype, typé à l'execution, fortement lié à l'utilisation des tables, proposant des définitions de functions anonymes. La mise en place de mécanismes clés (que nous évoquerons en détails) permette d'étendre son comportement et notamment de construire des mécanismes objets. Il est constitué d'un mécanisme de ramasse miettes.

2 Premiers Pas en Lua

Tout au long du document j'utiliserais des exemples commentés à des fins d'illustrations du langage.

2.1 Mots du langage

Les mots réservés sont les suivants :

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until
while
```

Les commentaires se font de deux façons :

- par ligne :

```
-- un commentaire de ligne
```

- par bloc :

```
--[[
        un commentaire
        de bloc
 --]]
```

2.2 Vue d'ensemble de Lua

Dans cette partie nous nous appuyons sur l'interpreteur 'stand alone' Lua-5.1 dont le prompt est le suivant :

```
Lua 5.1  Copyright (C) 1994-2006 Lua.org, PUC-Rio
>
```

Il existe 8 types en Lua :

1. nil
2. boolean
3. number
4. string
5. table
6. function
7. userdata
8. threads

Lorsque Lua est lancé, plusieurs librairies sont ouvertes. Il s'agit en fait de tables dont les clés sont des noms de fonctions et les valeurs associées des fonctions. Ces tables sont stockés dans la table générale de Lua `_G`.

```
Lua 5.1  Copyright (C) 1994-2006 Lua.org, PUC-Rio
>print(_G)
table: 0x91ca450
```

Il est possible de visualiser le contenu de cette table (les librairies). Pour cela nous utilisons le statement
`for [expression a] do [expression b] end`.
Ici nous allons utiliser un iterateur sur une table dont les clés ne sont pas des entiers positifs supérieur à 1. Cet iterateur est définit lors de l'utilisation du mot clé `pairs`, il renvoit 2 éléments : la clé et la valeur de clé itérée.

```
> for key,value in pairs(_G) do print(key,value) end
string  table: 0x91cb910
xpcall  function: 0x91cb318
package table: 0x91cbd98
tostring        function: 0x91cb270
print   function: 0x91cb388
os      table: 0x91cd470
unpack  function: 0x91cb2e0
require function: 0x91cc298
getfenv function: 0x91cafb0
setmetatable    function: 0x91cb1f8
next    function: 0x91cb0d0
assert  function: 0x91caf00
tonumber        function: 0x91cb238
io      table: 0x91ccd30
rawequal        function: 0x91cb3c0
collectgarbage  function: 0x91caf38
getmetatable    function: 0x91cb148
module  function: 0x91cbff0
rawset  function: 0x91cb430
math    table: 0x91ce038
debug   table: 0x91cea38
pcall   function: 0x91cb108
table   table: 0x91cb468
newproxy        function: 0x91cba18
```

```
type    function: 0x91cb2a8
coroutine       table: 0x91cba80
_G      table: 0x91ca450
select  function: 0x91ca478
gcinfo  function: 0x91caf78
pairs   function: 0x91caa00
rawget  function: 0x91cb3f8
loadstring      function: 0x91cb098
ipairs  function: 0x91ca9a0
_VERSION        Lua 5.1
dofile  function: 0x91caff0
setfenv function: 0x91cb1c0
load    function: 0x91cb060
error   function: 0x91cb028
loadfile        function: 0x91cb188
```

Nous remarquons au passage que `_G` est elle même répertorié dans sa table. De ce fait elle est referencée et non soumis au garbage collector de lua.

Parmis les éléments remarquables, nous pouvons citer :

- `unpack` : passage d'une table, à un ensemble de valeurs
- `require` : chargement d'un module
- `setmetatable` : affectation de la metatable d'une table
- `io` : entrée/sortie
- `getmetatable` : récupération de la metatable d'une table
- `module` : création d'un module
- `table` : opération sur les tables
- `type` : type d'un élément
- `pairs,ipairs` : itérateurs classique sur des clé,valeur ou nombre, valeur.

De la même façon que nous avons consulter le contenu de `_G` nous pouvons le faire, par exemple, pour `table` .

```
> for key,value in pairs(_G.table) do print(key,value) end
setn    function: 0x91cc9d8
insert  function: 0x91cb770
getn    function: 0x91cb700
foreachi        function: 0x91cb6c8
maxn    function: 0x91cb738
foreach function: 0x91cb690
concat  function: 0x91cb658
sort    function: 0x91cca10
remove  function: 0x91cb7a8
```

Il se trouve qu'il existe une fonction foreach dans `table`, elle aurait pu donc être utilisé pour faire cette consultation de la manière suivante :

```
>table.foreach(_G.table,print)
setn    function: 0x91cc9d8
insert  function: 0x91cb770
getn    function: 0x91cb700
foreachi        function: 0x91cb6c8
maxn    function: 0x91cb738
```

```
foreach function: 0x91cb690
concat  function: 0x91cb658
sort    function: 0x91cca10
remove  function: 0x91cb7a8
```

Remarque : le `_G` n'est pas nécessaire dans l'écriture de `table.foreach(_G.table,print)`, il est implicite dans la recherche des functions.

3 Premiers 'bout de codes'

3.1 Affectations

Les affectations peuvent être multiples dans un même statement. De plus le nombre d'arguments en automatiquement ajusté si il y en a trop (ignoré) ou pas assez (mis à `nil`).

```
A  > a,b = 3,4
B  > print(a,b)
   3       4
C  > a,b = b,a
   >  print(a,b)
   4       3
   > a,_,b = 4,5,6
   > print(a,b)
   4       6
   > a,b,c,d = 1,2,3
   > print(a,b,c,d)
   1       2       3       nil
   > a,b,c,d = 1,nil,4
   >  print(a,b,c,d)
   1       nil     4       nil
```

Remarqué le joli swap en C.

3.2 Création de tables et tableaux

Lorsque l'on crée une table en Lua, il y a automatiquement création d'un espace de tableau et d'un espace de table. La différence entre un table et une table provient des clés, qui dans le premier cas sont n'importe quelles valeurs et dans le deuxième cas sont uniquement des entiers : le tableau est donc un sous ensemble de table.

Pour créer une table :

```
> maTable={}
> =type(maTable)
table
> print(maTable)
table: 0x91d9728
```

La taille de cette table si il s'agit d'un tableau :

```
> =#maTable
0
```

affectation d'un élément dans la table :

```
> maTable["mot"]="bonjour"
> =#maTable
0
> maTable[1]="hello"
> =#maTable
1
> print(maTable["mot"],maTable.mot)
bonjour bonjour
> for i,j in ipairs(maTable) do print(i,j) end
1       hello
> for i,j in pairs(maTable) do print(i,j) end
1       hello
mot     bonjour
```

L'écriture `maTable.mot` est équivalente à `maTable["mot"]`. Nous voyons bien ici la différence qu'il existe entre les itérateurs `pairs` et `ipairs`, de plus nous pouvons effectivement constater qu'il y a deux partie dans une table en Lua : la partie tableau et la partie table proprement dite.

Ajout suppression d'éléments : le bon chemin.

```
A > maTable.mot=nil
B > for i,j in pairs(maTable) do print(i,j) end
  1       hello
C > maTable[2]=" every"; maTable[3]=" body"
D > for i,j in pairs(maTable) do print(i,j) end
  2        every
  1       hello
  3        body
E > for i,j in ipairs(maTable) do print(i,j) end
  1       hello
  2        every
  3        body
F > maTable[2]=nil
G > for i,j in ipairs(maTable) do print(i,j) end
  1       hello
H > =#maTable
  1
I > for i,j in pairs(maTable) do print(i,j) end
  1       hello
  3        body
  > table.insert(maTable,2," every")
J > table.remove(maTable,2)
K >  for i,j in ipairs(maTable) do print(i,j) end
  1     hello
  2      body
```

```
L > =#maTable
  2
```

En A nous affectons nil a une clé de *table*, celle-ci va être effacé. Un parcous de toutes les clés (B) le confirme. En C, nous affectons "every" et "body" aux éléments du tableau 2 et 3. Nous parcourons `maTable`, la sortie n'est pas ordonnée suivant un ordre croissant des clés : pour cela il faut utiliser l'itérateur `ipairs` (E). Nous faisons une tentative d'affectation à `nil`(F) de la clé 2 du tableau, puis nous le parcourons à nouveau. Maintenant `maTable` semble ne contenir qu'un seul élément (G), d'ailleurs la taille de ce tableau est considéré être à 1 (H), cependant un parcours sur toutes les clés contredit ce comportement (I) : Nous venons de créer un trou dans le tableau. La bonne solution pour effacer des éléments du tableau est de passer par la fonction `remove` (J) qui permet de déplacer tous les éléments suivants pour éviter les trous dans la partie tableau.

Il est cependant possible de construire des tables et des tableaux plus rapidement comme suit :

```
> t={ {["bonjour"]="hello"}, {tout="every"}, {le=""}, {["monde"]="body"} }
> for k,v in ipairs(t) do
>> local i,j = next(v)
>> print(i)
>> end
bonjour
tout
le
monde
```

3.3 Fonctions

Les fonctions peuvent être nommées ou anonymes en Lua.

3.3.1 Les fonctions nommées :

```
A > f1=function(a,b) return b,a end
B > function fprint(a,b) print(a,b) end
C > fprint(f1(1,2))
  2       1
D > fprint(f1(1))
  1       nil
E > function f1(...) return ... end
F > fprint(f1(1,2))
  1       2
G > fprint(f1(1,2,3))
  1       2
H > fprint = function (...) print(#{...} .. " arguments-->",...) end
I > fprint(f1(1,"toto",3))
  3 arguments-->        1       toto    3
```

La fonction `f1` (A) retourne 2 éléments et est déclaré de la première façon possible, la fonction `fprint` prend 2 arguments (B) et est déclarée suivant la

deuxième forme possible. L'appel à C, provoque l'affichage des deux éléments passer en paramêtres de `f1`. En D, le deuxième élément retourné par `f1` est ajusté à `nil` car inexistant, par conséquent il est affiché en tant que `nil` par `fprint`. Si maintenant, nous voulons un nombre variable d'éléments à retourner et un nombre variable d'arguments à prendre, il suffit d'utiliser la notation `...` qui signifie l'ensemble des arguments passer en paramêtres. Une deuxième définition de `f1` et de `fprint` peut donc être faite respectivement en E et H. Pour connaître le nombre d'arguments, nous pouvons créer un tableau contenant ces arguments et demander sa taille `#...`. J'ai également fait apparaître l'opérateur de concatenation `..` sur les chaînes.

3.3.2 Les fonctions avec arguments nommés :

Elles ne sont pas directement implémentés dans le langage, cependant il existe un moyen simple pour avoir le même effet comme l'illustre l'exemple suivant :

```
> function f(t) print(t.name or "undefined",t.color or "white",t.size or 15) end
> f{name="toto",color="green",size=3}
toto    green   3
> f{name="titi"}
titi    white   15
> f{name=false}
undefined       white   15
>  f{name=true}
true    white   15
```

Le mot clé `or` permet une alternative si l'élément de gauche est `nil` ou `false`

3.4 la portée des variables

Il s'agit ici d'illustrer le problème de portée des variables en Lua et le mot clé `local`.

```
  >=d
  nil
A > f2 = function(a,b)
B >>   d = 3
C >>   return a+b+d
D >> end
E > f2(1,1)
  5
F > d=4
G > =f2(1,1)
  5
H > =d
  3
```

Nous voyons clairement que la déclaration de `d=3` a une influence sur le contexte appelant H. Cette variable `d` est maintenant visible depuis le contexte global.

Pour éviter cela il faut utiliser le mot clé `local` pour limiter la portée de la variable au bloc dans lequel elle est définie (ce qui est implicite en C par exemple). Nous obtenons donc le code suivant une fois corrigé.

```
> =d
nil
>  f2 = function(a,b)
>>  local d = 3
>>  return a+b+d
>> end
> f2(1,1)
5
> =d
nil
> d=4
> =f2(1,1)
5
> =d
4
```

3.5 Les fonctions anonymes, les fermetures et les portées de variables :

A titre d'exemple voici un compteur qui utilise une fermeture et une fonction anonyme.

```
> function compteur()
>> local i = 0
>> return function()
>>  i = i + 1
>> return i
>> end
>> end
> cmpt = compteur()
> =cmpt
function: 0x91dc448
> =cmpt()
1
> =cmpt()
2
```

4 Les mécanismes des métatables en action

Nous explorons ici, les mécanismes qui font une grande partie la force de Lua.

4.1 Les métatables et métaméthodes

4.1.1 Introduction

Une métatable (appelons-là `mt`) est une table de référence pour une table donnée que nous appelerons `t`. Cette métatable contient notamment les éléments nécessaires au comportement de `t`. Les comportements définissables sont les suivants :

- la concaténation
- l'addition, la soustraction, la multiplication, la division, la mise à la puissance.
- l'accès, la création d'un élément.

Ces comportements sont stockés dans des champs particuliers de `mt`, par exemple l'addition dans `__add`, l'accès dans `__index`... Ces champs particulier sont appelés métaméthodes.

Le mécanisme de base est le suivant : lorsqu'une opération est requise sur `t` - disons une addition - alors le moteur Lua vérifie la présence d'une métatable `mt` pour `t`. Si `mt` existe alors le moteur vérifie que le champs `__index` est présent et contient le comportement pour l'addition (le champs `__add`).

L'exemple suivant illustre le propos :

```
A  > mt={}
B  >  mt.__add = function(ta,tb)
B.1>>  assert(getmetatable(ta)==getmetatable(tb))
   >>  local r={}
   >>  for i,j in ipairs(ta) do
   >>    table.insert(r,j)
   >>    end
   >>  for _,j in ipairs(tb) do
   >>    table.insert(r,j)
   >>  end
   >>   return r
   >> end
C  > t1={"salut","la","compagnie"}
D  > t2={"hello","guys"}
E  > setmetatable(t1,mt)
F  > setmetatable(t2,mt)
G  > t3=t1+t2
H  > table.foreachi(t3,print)
   1    salut
   2    la
   3    compagnie
   4    hello
   5    guys
I  > t3=t1-t2
   stdin:1: attempt to perform arithmetic on global 't1' (a table value)
   stack traceback:
           stdin:1: in main chunk
           [C]: ?
J  > t={"coucou"}
K  > t3=t1+t
```

```
   stdin:2: assertion failed!
   stack traceback:
        [C]: in function 'assert'
        stdin:2: in function <stdin:1>
        stdin:1: in main chunk
        [C]: ?
```

En A nous créons la métatable que nous utiliserons pour `t1` (C) et `t2` (D). En B nous définissons le comportement vis à vis de l'addition (il s'agit juste de faire l'union de 2 tableaux). En E et F nous affectons à `t1` et `t2` la métatable `mt` qui définira leur comportement vis des opérations évoquées précédement. En I l'opération de soustraction est demandé mais le comportement associé n'est pas défini, il en résulte une erreur. En J, nous définissons une nouvelle table et nous tentons de l'additionner avec `t1`, comme elle n'ont pas le même comportement vis à vis des opérations (ce que nous vérifions par l'assertion faite en B.1) il résulte une erreur.

4.1.2 Exemple :Tracker les accès aux tables

Nous illustrons ici une des utilisations des mécanismes des métatables en utilisant un exemple de suivi d'accès aux tables.

```
A  > function track(t)
B  >> local _t = t
C  >> local proxy={}
D  >> mt={}
E  >> mt.__index = function(tab,key)
      print("access "..key) ; return _t[key]
      end
F  >> mt.__newindex = function(tab,key,val)
      print("modify "..key.." with "..val)
      _t[key]=val
      end
G  >> setmetatable(proxy,mt)
H  >> return proxy
   >> end
   >
   > t={}
   > =t.coucou
   nil
   > t.coucou="hey ho"
I  > t=track(t)
   > =t.coucou
   access coucou
   hey ho
   > t.coucou="cou"
   modify coucou with cou
   >
```

En A nous créons une fonction de tracking. En B nous gardons une copie locale de la table à tracker. En C nous créons un proxy qui sera la table renvoyé à

l'utilisateur, c'est sur ce proxy que nous allons définir les comportements d'accès. En D nous créons une métatable pour définir le comportement du proxy. Les métaméthodes d'accès utilisent pour leurs parts la véritable table passer en arguments (E et F). En G, nous affectons le comportement du proxy, en H nous le retournons à l'utilisateur. Chaque fois qu'un accès au proxy en fait, la métaméthode correspondante est déclenché (`__index` ou `__newindex`), métaméthode qui accède à véritable table "trackée" !

4.2 L'approche objet à l'aide des métatables

Le mécanisme des métatables est à la base de l'implémentation du mécanisme objet en Lua. Nous allons illustré la construction d'un objet à l'aide d'un exemple en détaillant le fonctionnement de la machine Lua dans ce cas d'utilisation particulier.

4.2.1 Les prémices

```
A > Account={}
B > function Account.withdraw(self,v)
  >> self.balance = self.balance - v
  >> end
C > a={balance=0}
D > Account.withdraw(a,100)
E > print(a.balance)
  -100
F > a={balance=0,withdraw=Account.withdraw}
G > a:withdraw(100)
H > print(a.balance)
  -100
```

En A on créé une table pour contenir les fonctions d'un `Account`. En B, on crée une fonction `withdraw`. Le premier argument a été volontairement appelé self, même si ici il n'est pas nécessaire de le nommé ainsi. En C, on créé une table contenant un champs `balance`. On peut lui appliquer la fonction précédemment définie (D) et voir le résultat de l'application (E). En F on définie une nouvelle table qui contient un champ `balance` et un champ`withdraw` initialisé à la fonction `withdraw` de la table `Account`. La ligne G contient 2 éléments interessant dans ce que nous pouvons appelé les prémices de l'objet en Lua. En effet la notation `a :withdraw(100)` est un sucre syntaxique pour `a.withdraw(a,100)`, de plus `a.withdraw` est `Account.withdraw` donc la ligne G est équivalente à `Account.withdraw(a,100)`. Le résultat est visibe en H. Nous pourrions donc réécrire la ligne B de la façon suivante :
`function Account :withdraw(v)`, le paramètre self est tout de même accessible (comme `this` en C++).

4.2.2 Mon premier objet

Le principe de classe n'existe pas en Lua du fait de l'unicité des éléments qui l'on créer. Cependant il n'est pas difficile d'émuler le principe de classe en Lua. Nous avons vu précédemment le mécanisme de métatable pour définir le comportement d'une table. Le champs `__index` de la métatable est consulté chaque fois

qu'un champs indexé dans la table n'est pas présent. D'ore et déjà pour spécifier que a est un prototype de b il suffit d'écrire `setmetatable(a,__index=b)`.

Maintenant voyant précisement le principe de création d'un objet.

```
A  > function Account:new(o)
B  >> o = o or {balance=0}
C  >> self.__index=self
D  >> setmetatable(o,self)
E  >> return o
   >> end
F  > function Account:deposit(d)
   >> self.balance = self.balance + d
   >> end
G  > a=Account:new{balance=100}
   > =a.balance
   100
H  > a:deposit(100)
   > print(a.balance)
   200
```

En A nous créons un constructeur pour un `Account`. En B nous créons une table si aucune n'est fournie. En C, nous définissons le champ `__index` de la table `Account` à `Account`. Nous spécifions le comportement de notre objet à `Account` (D) et le retournons (E). Ceci nous permet ensuite de créer un `Account` (G) et d'y faire un dépot (H). La notation `a :deposit(100)` est décomposé de la façon suivante : `a :deposit(100)` est équivalent à `a.deposit(a,100)`. Nous faisons donc appel à la fonction `deposit` de la table `a`. Cette fonction n'existe pas, le moteur Lua cherche donc cette entrée dans le champs `__index` de la métatable de `a`. Nous avons donc : `getmetatable(a).__index.deposit(a,100)`. Comme `getmetatable(a).__index = Account` (ligne C), nous obtenons finalement : `Account.deposit(a,100)`.

4.2.3 L'héritage : là ou ça devient intéressant

Nous venons de définir une classe `Account`, nous allons maintenant définir une nouvelle classe `SpecialAccount`.

```
A  > SpecialAccount=Account:new()
B  > =SpecialAccount.balance
   0
C  > function SpecialAccount:withdraw(v)
   >> if self.balance - v < 0 then print("you have not enough money!")
   >> else self.balance = self.balance - v
   >> end
   >> end
   >
D  > b=SpecialAccount:new()
   > =b.balance
   0
E   > b:deposit(100)
   > =b.balance
```

```
   100
F  > b:withdraw(110)
   you have not enough money!
   > =b.balance
   100
```

En A, nous créons une instance de `Account`, cet objet va servir de base à la construction d'une nouvelle classe `SpecialAccount`. Comme `SpecialAccount` est une instance de `Account`, il possède les méthodes et attributs (B) de `Account`. En C, nous définissons une méthode `withdraw` qui controle le retrait d'argent. En D, nous créons une instance de `SpecialAccount`.L'interêt réside dans le fait que lors de l'appel à `new` l'argument caché `self` est maintenant `SpecialAccount`. Ce qui se passe donc lors de cet appel est l'affectation du champs `__index` de `SpecialAccount` à la table `SpecialAccount`. De plus l'objet créé par cet appel va voir sa métatable être `SpecialAccount`. N'oublions pas que `SpecialAccount` est une instance de `Account` et que par conséquent elle possède elle même pour métatable `Account`. L'étape de look-up (E et F) est expliquer ci-dessous.

```
 Account={
  function withdraw(self,v) ... end,
  function deposit(self,v) ... end,
  function new(self,o) ... end,
  __index=Account
}

getmetatable(SpecialAccount) == Account

SpecialAccount={
  function withdraw(self,v) ... end,
  __index=SpecialAccount
}


b=SpecialAccount:new()

-> Etape de look-up ligne F: 1 look-up
  b:withdraw(110)
  b.withdraw(b,110)              --pas de function withdraw dans la table b
  -- recherche dans le champs .__index de la métatable
  getmetatable(b).__index.withdraw(b,110)
  SpecialAccount.withdraw(b,100) --withdraw existe dans SpecialAccount


-> Etape de look-up ligne E: 2 look-up
  b:deposit(100)
  b.deposit(b,100)
  getmetatable(b).__index.deposit(b,100)
  SpecialAccount.deposit(b,100)  --deposit n'existe pas dans SpecialAccount
  -- recherche dans le champs .__index de la métatable
  getmetatable(SpecialAccount).__index.deposit(b,100)
```

```
Account.deposit(b,100)          -- deposit existe dans Account
```

Voilà, nous venons de créer une nouvelle classe à base d'un prototype.

5 Lua : la machine à pile

Lua est un langage à pile. De ce fait il existe un ensemble de fonctionnalité pour intéragir avec cette pile. Ce mécanisme est à la base de la possibilité d'étendre les fonctionnalités de Lua. L'interaction avec la pile ce fait à travers une API bas niveau écrite C. Nous allons décrire succintement les principes.

5.1 Appel de fonction et état de la pile

Voici du code Lua et l'état de la pile associé :

code	pile	index absolu	index relatif
`a=f(2,4,6)`	`2 4 6`	1 :2 2 :4 3 :6	-3 :2 -2 :4 -1 :6

5.2 Interaction avec la pile

5.2.1 ajout d'éléments sur la pile

Pour ajouter un élément sur la pile, il faut utiliser `lua_push*`, où `*` peut-être communément `{ boolean, nil, string, userdata, number, lightuserdata}`.

5.2.2 consultation des éléments de la pile

Pour vérifier les éléments sur la pile il faut utiliser `lua_is*` ou `*` est dans `{number, boolean, string, userdata, lightuserdata, nil }` (plus généralement tous les types définis dans Lua.

5.2.3 manipulation de la pile

Il est possible de récupérer les élements contenus dans la pile, mais aussi de modifier la pile comme par exemple dupliquer, enlever, ajouter des éléments sur la pile. Les fonctions qui permettent ces manipulations sont les suivantes :

```
//obtenir l'index absolue du haut de la pile
int   lua_gettop (lua_State *L);

//fixé le haut de la pile à l'index
void  lua_settop (lua_State *L, int index);

//ajouter en haut de la pile la valeur à l'index
void  lua_pushvalue (lua_State *L, int index);

//enlever l'élément à index et déplacer tous les éléments supérieurs
void  lua_remove (lua_State *L, int index);
```

```
//insere l'élément du haut de la pile à l'index et déplace tous
//les éléments supérieur
void  lua_insert (lua_State *L, int index);

//déplacé l'élément en haut de la pile à l'index (remplacement)
void  lua_replace (lua_State *L, int index);
```

6 Lua et AReVi

6.1 Etendre Lua

6.1.1 Appel de fonction C en Lua

Les fonctions de manipulation de la pile vont permettre d'étendre les fonctionnalités de Lua. Les fonctions d'extensions respectent toutes la même signature :
`int mafonction(luaState *L);`
L représente l'état de la machine Lua. C'est à travers ce pointeur que nous pourrons effectuer toutes les opérations évoquées précédemment. L'entier de retour permet de connaître le nombre d'élément ajouté sur la pile.

Voici par exemple une fonction qui permet l'affichage des éléments passer en paramêtres.

```
int printArg(lua_State *L){
  int nbArg = lua_gettop(L);

  lua_pushnumber(L,nbArg);

  while(--nbArg){
    int t=lua_type(L,nbArg);
    switch (t) {

    case LUA_TSTRING:  /* strings */
      printf("'%s'", lua_tostring(L, nbArg));
      break;

    case LUA_TBOOLEAN:  /* booleans */
      printf(lua_toboolean(L, nbArg) ? "true" : "false");
      break;

    case LUA_TNUMBER:  /* numbers */
      printf("%g", lua_tonumber(L, nbArg));
      break;

    default:  /* other values */
      printf("%s", lua_typename(L, t));
      break;

    }
  }
```

```
  return 1; //le nombre d'arguments
}
```

Il nous reste maintenant à enregistrer cette fonction dans la machine Lua pour y avoir accès. Pour cela, nous avons inclus dans le fichier `lua.c` (le mode interactif de Lua) le code ci-dessus et le code ci-dessous qui effectue l'enregistrement.

```
int main(...){
 ...
 lua_State *L = lua_open();  /* create state */

 ...
 lua_pushcfunction(L,printArg);
 lua_setglobal(L,"printMesArguments");
 ...
```

Maintenant nous compilons le standalone `lua` et relançons le mode interactif.

```
> =printArg
nil
> =printMesArguments
function: 0x81e9ad0
> a= printMesArguments("bonjour",1,{},true)
  bonjour  1  table  true
> =a
4
```

Voilà notre premier bindings en C.

6.1.2 Librairies C

Supposons que nous voulions maintenant créer un ensemble de fonctionnalités que nous voudrions retrouver dans une librairie appelé `addOn`.

Le code suivant permet de créer une librairie `addOn` contenant une fonction `printMesArguments`.

```
 1. const struct luaL_reg mesAddOn[]={
 2.   {"printMesArguments",printArg},
 3.   {NULL,NULL} //sentinelle
 4. };
 5.
 6. int luaopen_addOnlib(lua_State* L){
 7.   luaL_openlib(L,
 8.                "addOn",
 9.                mesAddOn,
10.                0);
11.   return 1; //ne sert pas ici
12. }
13.
```

```
14. int main(...){
15.
16.   ...
17.   luaopen_addOnlib(L);
18.   ...
19.
20. }
```

Ligne 1, nous déclarons un tableau de structure `luaL_reg`. Cette structure est définie comme suit :

```
typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;
```

En ligne 6, nous créons une fonction d'ouverture de notre librairie qui respecte le prototype des fonctions de l'api C Lua (pour un éventuel appel depuis Lua...). Ligne 7, nous appelons la fonction qui va permettre de créer un table qui s'appelle `"addOn"` et qui contient toutes les fonctions contenus dans le tableau `mesAddOn`

Après compilation et lancement du mode interactif nous avons :

```
 A > =printMesArguments
   nil
 B > =addOn
   table: 0x82f1b58
 C > =addOn.printMesArguments
   function: 0x82f1ba0
 D > a=addOn.printMesArguments("bonjour",true,3,{})
     bonjour  true  3  table
   > =a
   4
 E > pma=addOn.printMesArguments
   > a=pma("bonjour","c","facile",true)
     bonjour  c  facile  true
   > =a
   4
 F > function addOn.simplePrint(...)
   >> local a = addOn.printMesArguments(...)
   >> print("il y a "..a.." arguments")
   >> end
   > addOn.simplePrint("bonjour",true,3)
     bonjour  true  3
   il y a 3 arguments
 G > for i,j in pairs(addOn) do print(i,j) end
   printMesArguments    function: 0x82f1ba0
   simplePrint  function: 0x82ff330
   >
```

Maintenant la fonction `printMesArguments` n'est plus dans la table globale (A) mais dans la table `addOn`(C).Pour éviter d'avoir à faire appel à chaque fois à

`addOn.printMesArguments` on peut utiliser la ligne E. Maintenant, on peut même s'amuser à rajouter des fonctions dans cette table afin de compléter la librairie (F). Il est même possible de demander les fonctions disponibles dans cette librairie (G).

6.2 ARéVi et les bindings

Maintenant nous allons prendre un exemple concret que nous allons construire de A à Z en utilisant ARéVi. La majeure partie des fonctionnalités de l'API Lua ont été Wrappé à l'aide du plugin `LuaWrapper`.

Le wrappeur à juste pour vocation d'initialiser un ensemble de pointeurs de fonctions aux bonnes fonctions. Ces 'bonnes' fonctions contiennent l'appel aux fonctions C de l'API Lua.

Par convention les fonctions de l'API Lua sont disponibles à travers le plugins et possède la convention de nommage suivante :
`Lua : :*(void* L)` ou `*` est une fonction de l'API Lua. Le pointeur L fait référence à l'état Lua que nous dénomions `lua_State` précédemment.

Ces fonctions sont définies et implémentés respectivement dans les fichiers `AReVi/include/AReVi/Contrib/arLua.h` et `AReVi/src/AReVi/Contrib/arLua.cpp`. De plus pour la déclarations des classes la manipulation des objets, des fonctionnalités sont proposées afin de faciliter l'utilisation des objets AReVi en Lua. Ainsi pour exporter une classe AReVi vers Lua nous procéderons de la façon suivante (l'exemple porte sur le binding de la classe `Base3D`)

```
0.   int arevi_lua_base3D_new(void* L){
 .     ArRef<Base3D> b = new_Base3D();
 .     return LuaBind::shareArRef(L,b);
 .   }
 .
1.  int arevi_lua_base3D_roll(void *L){
 .    CHECK_RETRIEVE_ARREF(L,b,Base3D,1)
 .    CHECK_LUA_ARG(L,number,2)
 .
 .    b->roll(L,Lua::lua_tonumber(L,2));
 .    return 0;
 .  }
 .
2.  int arevi_lua_base3D_extractOrientation(void* L){
 .    CHECK_RETRIEVE_ARREF(L,b,Base3D,1)
 .
 .    double r,p,y;
 .
 .    b->extractOrientation(r,p,y);
 .
 .    Lua::lua_pushnumber(L,r);
 .    Lua::lua_pushnumber(L,p);
 .    Lua::lua_pushnumber(L,y);
 .
3.    return 3;
 .  }
```

```
.
4.  const arluaL_reg arevi_lua_base3D_method[]={
.     {"roll",arevi_lua_base3D_roll},
.   ...
.     {"extractOrientation",arevi_lua_base3D_extractOrientation},
.     {NULL,NULL}
.   };
.
5.  const arluaL_reg arevi_lua_base3D_function[]={
.     {"new",arevi_lua_base3D_new},
.     {NULL,NULL}
.   };
.
6.    int LuaExport::luaopen_arevi_lua_bindings(void* L){
      ...
      LuaBind::exportClass(L,"Base3D",
                           arevi_lua_base3D_method,
                           arevi_lua_base3D_function);
      ...
      }
```

En 1, nous declarons un bind pour la méthode `roll`, en 2 celui pour `extractOrientation`, nous remarquerons que la valeur de retour (en 3) est de 3 qui correspond au nombre d'éléments rajoutés sur la pile dans cette fonction. En 4, nous créons un tableau de structure appelé ici `arluaL_reg` (en référence à `luaL_reg`) dans laquelle nous stockons toutes les méthodes de notre classe `Base3D`. En 5, nous faisons de même pour le constructeur de la classe. En 6, nous exportons la classe `Base3D` avec ses méthodes et son constructeurs.

La fonction `LuaBind : :exportClass` se charge de plusieurs étapes. Premièrement elle génére la métatable nécessaire à la future notation objet (voir sous section 4.1). Elle remplie cette métatable avec les méthodes que l'on fournie, puis elle créé une table `Base3D` dans laquelle elle va mettre le constructeur. De plus elle met à jour la métaméthode `__gc` afin de spécifier un desctructeur pour l'objet lors d'un passage du ramasse miettes Lua.

Remarquons en 0 la fonction qui servira de constructeur de la classe. Dans celle si nous utilisons la fonction `LuaBind : :shareArRef(L,b)` qui permet de partager une `ArRef` avec Lua et de l'utiliser de manière transparente en Lua (utilisation à la mode objet).

En 1, nous utilisons `CHECK_RETRIEVE_ARREF(L,b,Base3D,1)` qui permet de récupérer une `ArRef` précédemment exporter vers Lua.

Une contrainte subsiste dans l'utilisation du `LuaBind : :exportClass`, en effet cette fonction se charge également d'effectuer le look-up vers les classes parentes. Il est donc indispensable de respecter un ordre dans l'exportation des classes. Par exemple l'exportation de la classe `Object3D` devra se faire après la classe `Base3D` afin qu'`Object3D` puisse atteindre les méthodes de `Base3D` De plus le nom donné aux classes doit être exactement celui obtenu par un appel à `obj->getClass()->getName()`, pour des raisons d'implémentations en interne.

6.3 Un exemple complet

7 Le côté caché du partage de données

Là va falloir attendre un peu.