Lua 5.2 Bytecode and Virtual Machine

by Dirk Laurie

Preface

I wrote this because writing out something is the only way I can get to understand it for myself. It's aimed at readers who think they know the Lua 5.2 Reference Manual (LRM5.2) pretty well, have a little experience with the machine instructions of at least one machine (Knuth's `MIX` will do) and would like to do some tinkering with code for the Lua 5.2 Virtual Machine (VM5.2).

Thanks: I've borrowed a lot from Elijah Frederickson's site https://github.com/mlnlover11/LuaAssemblyTools (LAT), which not only has a set of assembly tools for Lua 5.1 and Lua 5.2, but also a fairly complete collection of useful earlier stuff by other people, in particular, Kein-Hong Man's *A No-Frills Introduction to Lua 5.1 VM Instructions* (NFI), which despite being written for Lua 5.1 is still extremely useful. And of course, sooner or later you will feel compelled to read the very condensed but absolutely definitive comments in the Lua 5.2 source code, particularly `lopcodes.h` and `lundump.c`.

Binary chunks

> A chunk can be stored in a file or in a string inside the host program. To execute a chunk, Lua first precompiles the chunk into instructions for a virtual machine, and then it executes the compiled code with an interpreter for the virtual machine.
>
> Chunks can also be precompiled into binary form; see program `luac` for details. Programs in source and compiled forms are interchangeable; Lua automatically detects the file type and acts accordingly. — §3.3.2

The first paragraph above is the only one in in LRM5.2 containing the phrase "virtual machine". Three standard library functions deal with binary chunks: `load`, `loadfile` and `string.dump`. That is as much as the user is allowed to see officially. In closed-source code that would be as much as the user would ever know.

But Lua is open-source. Not only is the source code well commented: we are allowed to reverse-engineer. Let us try `luac` first.

```
$ luac
luac: no input files given
usage: luac [options] [filenames]
Available options are:
  -l       list (use -l -l for full listing)
  -o name  output to file 'name' (default is "luac.out")
  -p       parse only
  -s       strip debug information
  -v       show version information
  --       stop handling options
  -        stop handling options and process stdin
```

Right, we will give it a Hello World program, slightly convoluted to make it interesting. After the second line, terminal input stops (I typed Control-D) and `luac` printout starts.

```
$ luac -l -l -v -s -
Lua 5.2.1  Copyright (C) 1994-2012 Lua.org, PUC-Rio
local hello = "Hello"
print (hello.." World!")

main <stdin:0,0> (7 instructions at 0x9984970)
0+ params, 4 slots, 1 upvalue, 1 local, 3 constants, 0 functions
    1   [1] LOADK       0 -1    ; "Hello"
    2   [2] GETTABUP    1 0 -2  ; _ENV "print"
    3   [2] MOVE        2 0
    4   [2] LOADK       3 -3    ; " World!"
    5   [2] CONCAT      2 2 3
    6   [2] CALL        1 2 1
    7   [2] RETURN      0 1
constants (3) for 0x9984970:
    1   "Hello"
    2   "print"
    3   " World!"
locals (1) for 0x9984970:
    0   hello   2   8
upvalues (1) for 0x9984970:
    0   _ENV    1   0
```

We'll discuss this output later: at this stage all I want you to see are four lists: lightly commented assembly code, constants, locals and upvalues.

Now we go to Lua itself. `load` the same chunk, dump it to get the bytecode, write it to a file. I patched my Lua to give a prompt of three blanks instead of > so that you can just cut-and-paste this code.

```
$ lua
Lua 5.2.1  Copyright (C) 1994-2012 Lua.org, PUC-Rio
   func = load 'local  hello = "Hello" print (hello.." World!")'
   chunk = string.dump(func)
   io.open("hello.luac","wb"):write(chunk)
```

`luac` accepts binary chunks too. We can use it to take a look at what `load` did.

```
$ luac -l -l -v hello.luac
Lua 5.2.1  Copyright (C) 1994-2012 Lua.org, PUC-Rio

main <(string):0,0> (7 instructions at 0x90d0b50)
0+ params, 4 slots, 1 upvalue, 1 local, 3 constants, 0 functions
    1   [1] LOADK       0 -1    ; "Hello"
    2   [1] GETTABUP    1 0 -2  ; _ENV "print"
    3   [1] MOVE        2 0
    4   [1] LOADK       3 -3    ; " World!"
    5   [1] CONCAT      2 2 3
    6   [1] CALL        1 2 1
    7   [1] RETURN      0 1
constants (3) for 0x90d0b50:
    1   "Hello"
    2   "print"
    3   " World!"
locals (1) for 0x90d0b50:
    0   hello   2   8
upvalues (1) for 0x90d0b50:
    0   _ENV    1   0
```

Spot the differences?

- `(string)` vs `stdin`
- `0x90d0b50` vs `0x9984970` in four places
- `[1]` vs `[2]` on all but one of the instructions

I.e. `load` and `luac` basically generate exactly the same code.

Bytecode dissected

What exactly is in the bytecode? Here is the hexdump of `hello.luac` (made by hd on my system).

```
00000000  1b 4c 75 61 52 00 01 04  04 04 08 00 19 93 0d 0a  |.LuaR...........|
00000010  1a 0a 00 00 00 00 00 00  00 00 00 01 04 07 00 00  |................|
00000020  00 01 00 00 00 46 40 40  00 80 00 00 00 c1 80 00  |.....F@@........|
00000030  00 96 c0 00 01 5d 40 00  01 1f 00 80 00 03 00 00  |.....]@.........|
00000040  00 04 06 00 00 00 48 65  6c 6c 6f 00 04 06 00 00  |......Hello.....|
00000050  00 70 72 69 6e 74 00 04  08 00 00 00 20 57 6f 72  |.print...... Wor|
00000060  6c 64 21 00 00 00 00 00  01 00 00 00 01 00 30 00  |ld!...........0.|
00000070  00 00 6c 6f 63 61 6c 20  20 68 65 6c 6c 6f 20 3d  |..local  hello =|
00000080  20 22 48 65 6c 6c 6f 22  20 70 72 69 6e 74 20 28  | "Hello" print (|
00000090  68 65 6c 6c 6f 2e 2e 22  20 57 6f 72 6c 64 21 22  |hello.." World!"|
000000a0  29 00 07 00 00 00 01 00  00 00 01 00 00 00 01 00  |)...............|
000000b0  00 00 01 00 00 00 01 00  00 00 01 00 00 00 01 00  |................|
000000c0  00 00 01 00 00 00 06 00  00 00 68 65 6c 6c 6f 00  |..........hello.|
000000d0  01 00 00 00 07 00 00 00  01 00 00 00 05 00 00 00  |................|
000000e0  5f 45 4e 56 00                                    |_ENV.|
```

Much of that is for debugging, so pass `hello.luac` through `luac -s` and try again.

```
00000000  1b 4c 75 61 52 00 01 04  04 04 08 00 19 93 0d 0a  |.LuaR...........|
00000010  1a 0a 00 00 00 00 00 00  00 00 00 01 04 07 00 00  |................|
00000020  00 01 00 00 00 46 40 40  00 80 00 00 00 c1 80 00  |.....F@@........|
00000030  00 96 c0 00 01 5d 40 00  01 1f 00 80 00 03 00 00  |.....]@.........|
00000040  00 04 06 00 00 00 48 65  6c 6c 6f 00 04 06 00 00  |......Hello.....|
00000050  00 70 72 69 6e 74 00 04  08 00 00 00 20 57 6f 72  |.print...... Wor|
00000060  6c 64 21 00 00 00 00 00  01 00 00 00 01 00 00 00  |ld!.............|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00        |..............|
```

The format is not officially documented, and needs to be reverse-engineered. The necessary material is in the Lua source code, of course, in several places, mainly `ldump.c` and `lundump.c`. I have also cross-checked with NFI and LAT, but any remaining errors are mine.

The code starts with an 18-byte file header, which is the same for all official Lua 5.2 bytecode compiled on a machine like yours, whether by `luac` or `load` or `loadfile`. Lua 5.1 only had a 12-byte header, similar to the first 12 bytes of this one.

Byte numbers are in origin-1 decimal (mostly showing the arithmetic) and origin-0 hex.

`1 x00: 1b 4c 75 61`
: `LUA_SIGNATURE` from `lua.h`.

`5 x04: 52  00`
: Binary-coded decimal 52 for the Lua version, `00` to say the bytecode is compatible with the "official" PUC-Rio implementation.

`5+2 x06: 01  04  04  04  08  00`

Six system parameters. On x386 machines they mean: little-endian, 4-byte integers, 4-byte VM instructions, 4-byte `size_t` numbers, 8-byte Lua numbers, floating-point. These parameters must all match up between the bytecode file and the Lua interpreter, otherwise the bytecode is invalid.

`7+6 x0c: 19 93 0d 0a 1a 0a`

Present in all bytecode produced by Lua 5.2 from PUC-Rio. Described in `lundump.h` as "data to catch conversion errors". Might be constructed from binary-coded decimal 1993 (the year it all started), Windows line terminator, MS-DOS text file terminator, Unix line terminator.

After these 18 bytes come the functions defined in the file. Each function starts with an 11-byte function header.

`13+6 x12:  00 00 00 00`

Line number in source code where chunk starts. 0 for the main chunk.

`19+4 x16:  00 00 00 00`

Line number in source code where chunk stops. 0 for the main chunk.

`23+4 x1a:  00  01  04`

Number of parameters, vararg flag, number of registers used by this function (not more than 255, obviously). Local variables are stored in registers; there may not be more than 200 of them (see `lparser.c`).

Then a list of instructions. We'll do their format in more detail later.

`27+3 x1d:  07 00 00 00`

There are 7 instructions in the list (little-endian byte order, as specified in the file header).

`30+4*1 x21:  01 00 00 00`

Bytecode for `LOADK 0 -1`.

`30+4*2 x25:  46 40 40 00`

Bytecode for `GETTABUP 1 0 -2`.

`30+4*3 x29:  80 00 00 00`

Bytecode for `MOVE 2 0`.

`30+4*4 x2d:  c1 80 00 00`

Bytecode for `LOADK 3 -3`.

`30+4*5 x31:  96 c0 00 01`

Bytecode for `CONCAT 2 2 3`.

`30+4*6 x35:  5d 40 00 01`

Bytecode for `CALL 1 2 1`.

`30+4*7 x39:  1f 00 80 00`

Bytecode for `RETURN 0 1`.

Next a list of constants.

`58+4 x3d:  03 00 00 00`

There are 3 constants.

`62+4 x41:  04`

First constant is a string,

`66+1 x42:  06 00 00 00`

with 6 bytes in it,

`67+4 x46:  48 65 6c 6c 6f 00`

containing "Hello" and a terminating zero byte.

`71+6 x4c:  04  06 00 00 00  70 72 69 6e 74 00`

Second constant is the 6-byte string "print".

`77+5+6 x57:  04  08 00 00 00  20 57 6f 72 6c 64 21 00`

Third constant is the 8-byte string " World!"

Followed by a list of function prototypes.

`88+5+8 x64:  00 00 00 00`

There are no function prototypes.

Finally followed by a list of upvalues.

```
101+4 x68:  01 00 00 00
```
There is 1 upvalue.
```
105+4 x6c:  01 00
```
It can be found in a stack 1 level up, at position 0 in that stack.

At this point, the two files differ. In the stripped version, the absence of debugging information shows as 16 zero bytes.

```
109+2 x6e:  00 00 00 00
```
There is no source code
```
111+4 x72:  00 00 00 00
```
The list of line numbers is empty
```
115+4 x76:  00 00 00 00
```
The list of local variable names is empty
```
119+4 x7a:  00 00 00 00
```
The list of upvalue names is empty

In the unstripped version, the information is given thus:

```
111 x6e:  30 00 00 00
```
The source code is 48 bytes long. Skipping past the source,
```
111+4+48 xa2:  07 00 00 00
```
There are seven line numbers. Skipping past those numbers,
```
163+4+28 xc2:  01 00 00 00
```
One local variable name is listed.
```
199 xc6:  06 00 00 00  68 65 6c 6c 6f 00
```
It's `hello`.
```
199+4+6 xd0:  01 00 00 00
```
It comes into scope at instruction 1.
```
209+4 xd4:  07 00 00 00
```
It goes out of scope at instruction 7.
```
213+4 xd8:  01 00 00 00
```
One upvalue name is listed.
```
217+4 xdc:  05 00 00 00  5f 45 4e 56 00
```
It's `_ENV`.

After this, more functions may be encoded in the same way.

That's as far as I go. The following areas have not been covered:

- How non-string constants are stored.
- How function prototypes look.

NFI has all that and more, but it's 5.1 only, so you will have to check with LAT and/or `lundump.c`.

Snooping around in the bytecode

This document is accompanied by two module files: `vm52.lua` and `bytecode.lua`. From the first, we use only `numberAt` at this stage. This function converts a four-byte substring to a 32-bit number, taking into account the endian-ness of the host system. We know from the above that a Lua 5.2 dumped function has the varargs flag at 28, the register count at 29, and the instruction count at position 30, followed by that many four-byte instructions.

So let us take a look.

```
$ lua -l vm52
Lua 5.2.1  Copyright (C) 1994-2012 Lua.org, PUC-Rio
```

```
   func = load 'local  hello = "Hello" print (hello.." World!")'
   chunk = string.dump(func)

   function hasvarargs(chunk) return chunk:sub(28,28):byte()>0 end
   function nstack(chunk) return chunk:sub(29,29):byte() end
   function ninstr(chunk) return vm52.numberAt(chunk,30) end
   function instr(chunk,i) return vm52.numberAt(chunk:sub(30+4*i,33+4*i)) end

   ns = nstack(chunk)
   print ("This function" ..
     (hasvarargs(chunk) and " has a variable number of arguments and" or "") ..
     " uses "..ns.." register"..(ns~=1 and "s" or ""))

   for i=1,ninstr(chunk) do
      print(string.format("%08X",instr(chunk,i)))
   end
00000001
00404046
00000080
000080C1
0100C096
0100405D
0080001F
```

You can see the bytes of the bytecode file, reversed because my machine is little-endian.

The module file `vm52.lua` has more than just `numberAt`: it has the four functions I defined in the above code, and more important, it has `assemble` and `disassemble`. The latter does much the same as `luac -l` does when given dumped bytecode. Following on the previous Lua session,

```
   for i=1,vm52.ninstr(code) do
      print(vm52.disassemble(vm52.instr(code,i)))
   end
LOADK 0 -1
GETTABUP 1 0 -2
MOVE 2 0
LOADK 3 -3
CONCAT 2 2 3
CALL 1 2 1
RETURN 0 1
```

Instruction anatomy

The above hex listing corresponds exactly with the bytecode file, except that the bytes are reversed. This is a consequence of the little endian-ness of my system: little-endian byte order is really hard to read when you work with 32-bit numbers, given that bytes are universally coded with big-endian bits.

So the instruction bytes need to be reversed before we decompose them into bits and display them. The `numberAt` function does that if necessary.

As we have seen, the instructions are:

```
00 00 00 01   LOADK 0 -1
00 40 40 46   GETTABUP 1 0 -2
00 00 00 80   MOVE 2 0
00 00 80 c1   LOADK 3 -3
01 00 c0 96   CONCAT 2 2 3
```

```
01 00 40 5d   CALL 1 2 1
00 80 00 1f   RETURN 0 1
```

We now expand the bytes into bits and group them. If `lopcodes.c` says the instruction has mode `iABC`, we group the bits as (9,9,8,6), defining the values `B`,`C`,`A`,`OP`; if it says `iABx` or `iAsBx`, as (18,8,6), defining the values `Bx`,`A`,`OP` or `sBx`,`A`,`OP`; if it says `iAx`, as (26,6), defining the values `Ax`,`OP`.

```
    B    Bx   C         A      OP
000000000000000000  00000000 000001   LOADK 0 -1       (A,Bx)
000000000 100000001 00000001 000110   GETTABUP 1 0 -2  (A,B,C)
000000000 000000000 00000010 000000   MOVE 2 0         (A,B)
000000000000000010  00000011 000001   LOADK 3 -3       (A,Bx)
000000010 000000011 00000010 010110   CONCAT 2 2 3     (A,B,C)
000000010 000000001 00000001 011101   CALL 1 2 1       (A,B,C)
000000001 000000000 00000000 011111   RETURN 0 1       (A,B)
```

In the above table, the bits are numbered left to right as 31, 30, …, 0. The notation "0+6" means the 6 bits with low-order bit (rightmost in the above table) at position 0. If the instruction is stored as a Lua number, these bits are given by `bit32:extract(0,6)`.

1. `OP` is the opcode, at 0+6 (see below).
2. `A` is the first operand in all instructions, at 6:8, treated as an unsigned number. Its maximum value is 255.
3. `B` is the second operand in instructions with mode `iABC`, at 23+9, treated as a signed number. The actual instruction does not always use `B`.

   The first bit is the sign bit. If this is 1, the other eight bits encode one less than the absolute value of `B`. Thus there is no -0, the minimum value of `B` is -256 and its maximum value is 255.
4. `C` is the third operand in instructions with mode `iABC`, at 14+9, treated as a signed number like `B`. The actual instruction does not always use `C`.
5. `Bx` is the second operand in instructions with mode `iABx`, at 14+18, treated as a 19-bit signed number like `B` but not storing the sign bit since `Bx` is always negative.
6. `sBx` means "Bx treated as an 18-bit signed number."
7. `Ax` is the only operand in instructions with mode `iAx`, at 6+26, treated like `Bx` (27-bit signed number, always negative, sign bit not stored).

The opcodes and a brief description of what the instructions do are given as an `enum` in `lopcodes.h`. We have grouped them in groups of related functionality. The description (taken verbatim from `lopcodes.h`) uses the following notation, in which `B`, `Bx` etc refer not to numbers encoded in bits, but to the numbers visible in the disassembly shown by `luac`.

`R(A)`, `R(B)`, `R(C)`
: The register numbered `A`, `B` or `C`.

`Kst(Bx)`
: The constant numbered `|Bx|.`

`KPROTO[Bx]`
: The function prototype numbered `|Bx|`.

`RK(B)`, `RK(C)`
: The register numbered `B` or `C`; if `B` or `C` is negative, the constant numbered `|B|` or `|C|`.

`FPF`
: Fields per flush. Read NFI's explanation.

`UpValue[B]`
: The upvalue numbered `B`.

`pc`
: Program counter: which instruction will be executed next.

`closure`
: Make a closure (function plus upvalues).

Loading constants

opcode	name and args	description
1	LOADK(A,Bx)	R(A) := Kst(Bx)
2	LOADKX(A)	R(A) := Kst(extra arg)
39	EXTRAARG(Ax)	extra (larger) argument for previous opcode

Unary functions

opcode	name and args	description
0	MOVE(A,B)	R(A) := R(B)
19	UNM(A,B)	R(A) := -R(B)
20	NOT(A,B)	R(A) := not R(B)
21	LEN(A,B)	R(A) := length of R(B)

Binary functions

opcode	name and args	description
13	ADD(A,B,C)	R(A) := RK(B) + RK(C)
14	SUB(A,B,C)	R(A) := RK(B) - RK(C)
15	MUL(A,B,C)	R(A) := RK(B) * RK(C)
16	DIV(A,B,C)	R(A) := RK(B) / RK(C)
17	MOD(A,B,C)	R(A) := RK(B) % RK(C)
18	POW(A,B,C)	R(A) := RK(B) ^ RK(C)

Table access

opcode	name and args	description
7	GETTABLE(A,B,C)	R(A) := R(B)[RK(C)]
10	SETTABLE(A,B,C)	R(A)[RK(B)] := RK(C)
11	NEWTABLE(A,B,C)	R(A) := {} (size = B,C)
12	SELF(A,B,C)	R(A+1) := R(B); R(A) := R(B)[RK(C)]
36	SETLIST(A,B,C)	R(A)[(C-1)*FPF+i] := R(A+i), 1 <= i <= B)

Dealing with tuples

opcode	name and args	description

opcode	name and args	description
4	LOADNIL(A,B)	R(A), R(A+1), ..., R(A+B) := nil
22	CONCAT(A,B,C)	R(A) := R(B).. ... ..R(C)
29	CALL(A,B,C)	R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))
31	RETURN(A,B)	return R(A), ... ,R(A+B-2) (see note)
38	VARARG(A,B)	R(A), R(A+1), ..., R(A+B-2) = vararg
30	TAILCALL(A,B,C)	return R(A)(R(A+1), ... ,R(A+B-1))
34	TFORCALL(A,C)	R(A+3), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));

Interaction with upvalues

opcode	name and args	description
5	GETUPVAL(A,B)	R(A) := UpValue[B]
9	SETUPVAL(A,B)	UpValue[B] := R(A)
6	GETTABUP(A,B,C)	R(A) := UpValue[B][RK(C)]
8	SETTABUP(A,B,C)	UpValue[A][RK(B)] := RK(C)

Logical functions

opcode	name and args	description
3	LOADBOOL(A,B,C)	R(A) := (Bool)B; if (C) pc++
24	EQ(A,B,C)	if ((RK(B) == RK(C)) ~= A) then pc++
25	LT(A,B,C)	if ((RK(B) < RK(C)) ~= A) then pc++
26	LE(A,B,C)	if ((RK(B) <= RK(C)) ~= A) then pc++
27	TEST(A,C)	if not (R(A) <=> C) then pc++
28	TESTSET(A,B,C)	if (R(B) <=> C) then R(A) := R(B) else pc++

Branches, loops and closures

opcode	name and args	description
23	JMP(A,sBx)	pc+=sBx; if (A) close all upvalues >= R(A) + 1
32	FORLOOP(A,sBx)	R(A)+=R(A+2); if R(A) <?= R(A+1) then
		{ pc+=sBx; R(A+3)=R(A) }
33	FORPREP(A,sBx)	R(A)-=R(A+2); pc+=sBx
35	TFORLOOP(A,sBx)	if R(A+1) ~= nil then
		{ R(A)=R(A+1); pc += sBx }

opcode	name and args	description
37	CLOSURE(A,Bx)	R(A) := closure(KPROTO[Bx])

Notes

Taken nearly verbatim from `lopcodes.h`.

- In CALL, if (`B == 0`) then `B = top`. If (`C == 0`), then `top` is set to `last_result+1`, so next open instruction (CALL, RETURN, SETLIST) may use `top`.
- In VARARG, if (`B == 0`) then use actual number of varargs and set top (like in CALL with `C == 0`).
- In RETURN, if (`B == 0`) then return up to `top`.
- In SETLIST, if (`B == 0`) then `B` =top`; if (`C == 0`) then next 'instruction' is EXTRAARG(real C).
- In LOADKX, the next 'instruction' is always EXTRAARG.
- For comparisons, A specifies what condition the test should accept (true or false).
- All 'skips' (`pc++`) assume that next instruction is a jump.

Very brief remarks on VM5.2

A little learning is a dangerous thing.
Drink deep, or taste not the Pierian spring.

Those well-known words of Alexander Pope serve as a warning here.

You should really read NFI first and treat this section as a mere summary of the points needed to understand the rest of this document. Actually, it contains only the stuff I had to look up for myself after reading the description of the operations above, so it may be a good idea to re-read that before continuing.

I found it helpful to disassemble short snippets of Lua code and figure out why the VM instructions achieve the desired result. For example:

```
$ lua -l vm52
   src = "local x, y = ...; return x+y, ..."
   chunk = string.dump(load(src))
   for i=1,vm52.ninstr(chunk) do
      print(vm52.disassemble(vm52.instr(chunk,i)))
   end
VARARG 0 3
ADD 2 0 1
VARARG 3 0
RETURN 2 0
RETURN 0 1
```

`VARARG 0 3`
: Load the first (3-1) arguments from the vararg list into consecutive registers starting at register 0.

`ADD 2 0 1`
: Add registers 0 and 1, storing the result in register 2.

`VARARG 3 0`

Load all of the vararg list into consecutive registers starting at register 3.

`RETURN 2 0`

Return everything from register 2 to the stack top.

`RETURN 0 1`

Lua can't be sure in general that there is no condition branch skipping the preceding return statement and always inserts another return statement for good measure.

The C API of Lua does everything on a virtual stack. Seen in a certain way, the virtual machine is merely an interface to the API, providing a layer of abstraction between Lua source code and the API. However, the VM has access to the private fields of a `lua_State`, whereas the C API works entirely via the documented routines.

The Virtual Machine itself is not stack-based. It is allowed exclusive use of a portion of the stack called its *frame*, but within that frame, the VM works with constant register numbers: there is no pushing and popping. An important difference in usage is that negative indices don't mean the same as in the C API. They refer to constants stored elsewhere, not in the stack frame.

The VM can access items on other active stack frames that have been provided as the upvalues.

Local variables are aliases for registers. The `luac` listing grabs their names from the debugging information, but instruction disassembly by itself cannot do that. There is no overhead involved in creating them.

That's it for now

When I have time and inclination to add more, I'll do so. Please send corrections and comments. If I am still active and about, you will find me on `lua-l`.