

Compilador e desmontador simples para um subconjunto da linguagem e máquina virtual Lua.

Ricardo Ryoiti Sugawara Júnior

Dezembro de 2011

Sumário

- **Objetivo:** Explorar algumas das áreas da construção de compiladores:
 - Análise léxica
 - Análise sintática
 - Semântica e geração de código
 - Otimizações triviais
 - Formas intermediárias
 - Reconhecedores com autômatos de pilha estruturados.
- **Escopo:**
 - Construção de um compilador para uma linguagem simples, gerando *bytecode* para uma máquina virtual baseada em pilhas.
 - Desmontador deste *bytecode*.

Linguagem e máquina virtual Lua

- **Lua**: linguagem leve, integrável e extensível. Desenvolvida em 1993 na PUC-RJ por R. Ierusalimschy [1].
- Compilada para *bytecode* e executada em máquina virtual baseada em pilhas, inspirada na *p-code* de Pascal.

- **Exemplo:**

```
print("Alo, mundo!")  
for i = 1,5 do print (i) end
```

Linguagem e máquina virtual Lua

- **Operações:** 35 na versão 5.1. Exemplos:
MOVE / LOADK / LOADBOOL / LOADNIL
GETUPVAL / GETGLOBAL / SELF
ADD / SUB / MUL / DIV / MOD / POW / NOT
EQ / LT / LE / TEST / LEN / CONCAT
CALL / JMP / RETURN / FORLOOP
- **Instruções:** 3 formatos binários.

| | 31 | 24 | 23 | 16 | 15 | 8 | 7 | 0 |
|-------|--------|----|----|-----|----|-----|---|----------|
| | | | | | | | | |
| iABC | B:9 | | | C:9 | | A:8 | | Opcode:6 |
| iABx | Bx:18 | | | | | A:8 | | Opcode:6 |
| iAsBx | sBx:18 | | | | | A:8 | | Opcode:6 |
| | | | | | | | | |

Lua 5 Instruction Formats

Linguagem e máquina virtual Lua

- Formato do *bytecode* [3]:
 - Organizados em dois tipos de blocos binários (*binaries chunks*):
 - Cabeçalhos
 - Funções
- Blocos de *cabeçalhos*:
 - Informações sobre versão, tamanhos de variáveis, *endianess*, etc.
- Blocos de *funções*:
 - Listas de variáveis, constantes e instruções.
 - Informações de depuração.
- Blocos de cabeçalhos + funções são aninhados.

Linguagem e máquina virtual Lua

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

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) • 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) |

Linguagem e máquina virtual Lua

- Listas também possuem formatos específicos.
- **Exemplo:** lista de *strings*.

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.

Parte 1

Desmontador do *bytecode* da linguagem Lua.

Desmontador do *bytecode* da linguagem Lua.

- O trabalho foi desenvolvido na linguagem **Erlang** [2].
- **Erlang**: linguagem funcional.
 - Imutabilidade: variáveis não são variáveis.
 - Declarativa: Inexistência de efeitos colaterais, expressa a lógica da computação sem descrever o fluxo computacional.
- Ponto de entrada para o desmontador:

```
parse(Filename) when is_list(Filename) ->  
    {ok, Bin} = file:read_file(Filename),  
    parse(Bin, Filename).
```

Desmontador do *bytecode* da linguagem Lua.

- Função *parse* (com 2 parâmetros) faz o papel de um autômato estruturado em pilhas.
- Estados são passos da leitura e sub-rotinas. Cada sub-rotina consome um tipo de informação.
- **Exemplo:** Código para ler uma instrução:

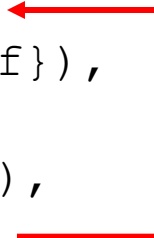
```
get_list({Type, Conf}, Sizelist, N, R, Acc) ->
  Sint = Conf#conf.sint,
  {Elem, Rest} = case Type of
    instruction ->
      <<Elem2:4/binary, Rest2/binary>> = R,
      {Elem2, Rest2};
      ... SUPRIMIDO ...
  end,
  get_list({Type, Conf}, Sizelist, N-1, Rest, [Elem|Acc]);

get_list({_, _}, Sizelist, 0, R, Acc) ->
  {Sizelist, lists:reverse(Acc), R}.
```

Desmontador do *bytecode* da linguagem Lua.

- Empilhamento é intrínseco à execução (chamadas recursivas).
- **Ilustração simplificada do empilhamento:** Blocos de funções com aninhamento.

```
parse(Filename) when is_list(Filename)
parse(Bin, Arg) when is_binary(Bin), is_list(Arg)
parse(Bin, {header, _})
» parse(R, {functionblock, Conf})
  » parse(Bin, {functionblock, Conf})
    get_list(Lists, {instruction, Conf}),
    get_list(Next, {constant, Conf}),
    get_list(Bin3, {prototypes, Conf}),
  » parse(R, {functionblock, Conf}),
    get_list(Bin4, {sourceline, Conf}),
    get_list(Bin5, {local, Conf}),
    get_list(Bin6, {upvalue, Conf}),
```



Desmontador do *bytecode* da linguagem Lua.

- Programa de exemplo:

```
for i = 1,10 do
    for j = 1,10 do
        print(i*j)
    end
end
```

- Globais: print, 1, 10
- Variáveis: i, j
- Blocos: Único.

Desmontador do *bytecode* da linguagem Lua.

- Exemplo de saída do *parser*: `./luadesmonta teste5.luac`

Parsing file teste5.luac.

```
* header:
  magic string: \27Lua
  version: 51
  format: 0
  endianness: 1
  integer: 4
  size_t: 4
  instruction: 4
  lua_Number: 8
  Integral: 0

* function block:
  source file: @teste5.lua
  Line defined: 0
  Last line: 0
  Upvalues: 0
  Parameters: 0
  Is VarArg: 2
  Max Stack: 10
```

(continua).

Desmontador do *bytecode* da linguagem Lua.

- Exemplo de saída do *parser*.

* code: 14 instructions.

| opcode | A | B | C | Bx | sBx | |
|-----------|----------|----------|----------|----------|-----------|-------------------------|
| LOADK | 0 | 0 | 0 | 0 | **** | (1) |
| LOADK | 1 | 0 | 1 | 1 | **** | (10) |
| LOADK | 2 | 0 | 0 | 0 | **** | (1) |
| FORPREP | 0 | 256 | 7 | **** | 8 | (até FORLOOP 2) |
| LOADK | 4 | 0 | 0 | 0 | **** | (1) |
| LOADK | 5 | 0 | 1 | 1 | **** | (10) |
| LOADK | 6 | 0 | 0 | 0 | **** | (1) |
| FORPREP | 4 | 256 | 2 | **** | 3 | (até FORLOOP 1) |
| GETGLOBAL | 8 | 0 | 2 | 2 | **** | (print) |
| MUL | 9 | 3 | 7 | 1543 | **** | |
| CALL | 8 | 2 | 1 | 1025 | **** | |
| FORLOOP | 4 | 255 | 507 | **** | -4 | (para GETGLOBAL) |
| FORLOOP | 0 | 255 | 502 | **** | -9 | (para LOADK 4 0) |
| RETURN | 0 | 1 | 0 | 512 | **** | |

Desmontador do *bytecode* da linguagem Lua.

- Exemplo de saída do *parser*

```
* constant: 3 constants.
```

```
  type value
```

```
    {3,1.0}
```

```
    {3,10.0}
```

```
    {4,{str,6,"print"}}
```

```
* functions: 0 functions.
```

```
* source line informations: 14
```

```
  1
```

```
  1
```

```
  (suprimido)
```

```
  5
```

Desmontador do *bytecode* da linguagem Lua.

- Exemplo de saída do *parser*.

```
* local informations: 8
  {{str,12,"(for index)"},3,13}
  {{str,12,"(for limit)"},3,13}
  {{str,11,"(for step)"},3,13}
  {{str,2,"i"},4,12}
  {{str,12,"(for index)"},7,12}
  {{str,12,"(for limit)"},7,12}
  {{str,11,"(for step)"},7,12}
  {{str,2,"j"},8,11}
```

(fim) .

Parte 2

Compilador para *bytecode* da máquina virtual Lua.

Análise léxica

- **Ferramenta:** `leex` [4]. Gerador de analisadores léxicos para a linguagem Erlang, com expressões regulares com regras compostas e códigos.
- **Exemplo:** Reconhecedor léxico para calculadora de inteiros.

Definitions.

`Dig = [0-9]`

Rules.

`({Dig}{Dig}*) : {token, {integer, TokenLine, list_to_integer(TokenChars)}}.`

`\+ : {token, {'+', TokenLine}}.`

`\- : {token, {'-', TokenLine}}.`

`* : {token, {'*', TokenLine}}.`

`\/ : {token, {'/', TokenLine}}.`

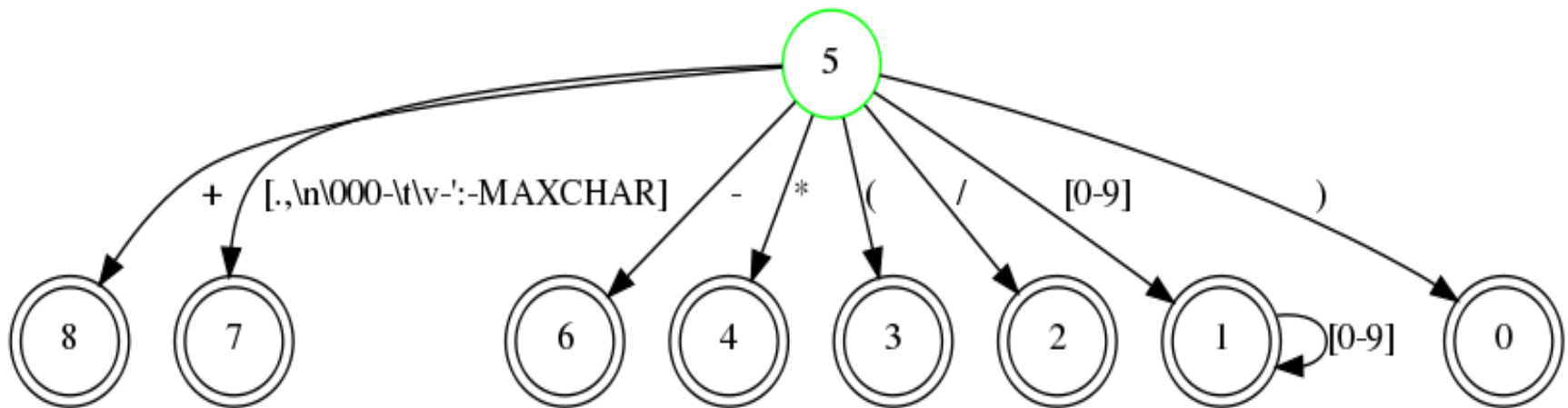
`\(: {token, {'(', TokenLine}}.`

`\) : {token, {')', TokenLine}}.`

`(.|\n) : skip_token.`

Análise léxica

- **Resultado:** A.F.D. para reconhecer os átomos.
- **Saída:** Formato intermediário (listas e tuplas da linguagem Erlang).
 - **Formato:** {ok, [lista de átomos], última linha}
 - **Átomo:** {tipo ou átomo, valor, linha}



Análise léxica

- **Exemplo:** Exemplo de execução:

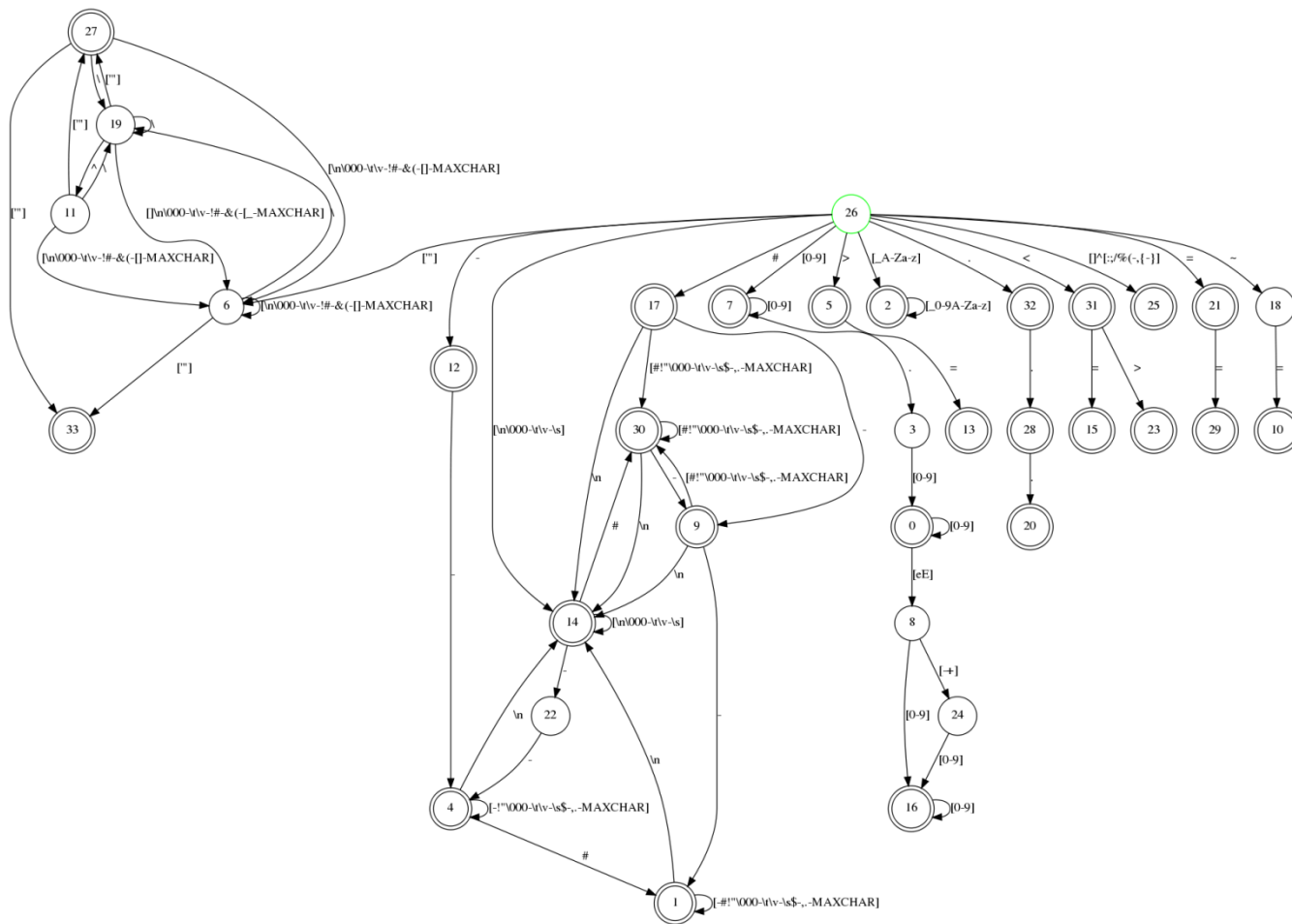
```
calc_example:string("1+2*3").
```

```
{ok, [{integer, 1, 1},  
      {'+', 1},  
      {integer, 1, 2},  
      {'*', 1},  
      {integer, 1, 3}],  
1}
```

Análise léxica

- Contemplam a análise léxica:
 - Definição de identificadores
 - Nomes: $[A-Za-z_][A-Za-z0-9_]^*$
 - Espaços em branco $([\backslash 000-\backslash s] | --.* | \#.*)$
 - Dígitos $[0-9]$
 - Regras de agrupamento
 - Operadores $==, <=, <>, \text{etc.}$
 - Identificadores: $\{ID\}^* = \text{nome ou átomo.}$
 - Identificação de palavras reservadas
 - Notações numéricas
 $\{D\}+\backslash.\{D\}+((E|e)(\backslash+|\backslash-)?\{D\}+)? :$
 $\{\text{token}, \{\text{float}, \text{TokenLine}, \text{list_to_float}(\text{TokenChars})\}\}.$
 - Tratamento de caracteres especiais em *strings*.

Análise léxica



Análise léxica

- Exemplo:
 - Análise léxica do seguinte código fonte:

```
for i = 1,10 do
    for j = 1,10 do
        print(i*j)
    end
end
```

Análise léxica

- **Exemplo:** Exemplo de saída:

```
{ok, [ {for,1}, {name,i,1},  
      {'=',1}, {integer,1,1},  
      {' ',1}, {integer,1,10},  
      {do,1}, {for,2},  
      {name,j,2}, {'=',2},  
      {integer,2,1}, {' ',2},  
      {integer,2,10}, {do,2},  
      {name,print,3}, {'(',3},  
      {name,i,3}, {'*',3},  
      {name,j,3}, {' ) ',3},  
      {'end',4}, {'end',5} ] ,  
6 }
```


Análise sintática

- **Ferramenta:** `yacc` [5]. Gerador de analisadores sintáticos LR(1) para a linguagem Erlang a partir de uma gramática BNF. Simililar ao *yacc/bison*.
- **Gramática:** Exemplo de reconhecedor léxico para calculadora de inteiros.

Nonterminals `exp.`

Terminals `integer '(' ')' '+' '-' '*' '/'.`

Rootsymbol `exp.`

Left 100 `'+' '-'.`

Left 150 `'*' '/'.`

`exp -> integer : '$1'.`

`exp -> exp '+' exp: {'$2', '$1', '$3'}.`

`exp -> exp '-' exp: {'$2', '$1', '$3'}.`

`exp -> exp '*' exp: {'$2', '$1', '$3'}.`

`exp -> exp '/' exp: {'$2', '$1', '$3'}.`

Análise sintática

- **Exemplo:** Exemplo de saída (formato intermediário). Observe a estrutura resultante.

```
{ok, R, _} = calc_example:string("1+2*3") .  
calc_example_parser:parse(R) .
```

```
{ok, {  
    {'+', 1},  
    {integer, 1, 1},    {{ '*', 1},  
    {integer, 1, 2},    {integer, 1, 3}}  
}
```

Análise sintática

- **Exemplo:** Saída do exemplo de *loops* aninhados.

(suprimido)

```
{ok, L} = luaparse:parse(R).
```

```
{ok, [{for2,  
      {{name,i,1},{integer,1,1},{integer,1,10}},  
      [{for2,  
        {{name,j,2},{integer,2,1},{integer,2,10}},  
        [{call,  
          {var,{name,print,3}},  
          {args,  
            [{binop,  
              {'*',3},  
              {var,{name,i,3}},  
              {var,{name,j,3}}}]}]}]}]}
```

Otimizações triviais

- Foi implementada uma otimização trivial de *folding* de expressões, avaliando-as em tempo de compilação.
 - Combinações: inteiro-inteiro, inteiro-float, e float-float.
- Exemplo:

```
{binop,{'%',1},{integer, 1, 5},{integer, 1, 2}}
```

```
{true,{integer,1,1}}
```

- Técnica: Chamada de função com guarda.

Otimizações triviais

- **Código:** Otimização trivial de *folding* em expressões com operadores binários.

```
optimize(Exp) ->
  case Exp of
    {binop, {'+', L}, {integer, _, A}, {integer, _, B}} ->
      {true, {integer, L, A+B}};
    {binop, {'-', L}, {integer, _, A}, {integer, _, B}} ->
      {true, {integer, L, A-B}};
    {binop, {'*', L}, {integer, _, A}, {integer, _, B}} ->
      {true, {integer, L, A*B}};
      ( ... suprimido ...)
    _ ->
      {false, {Exp}}
  end.
```

Geração de código

- **Entrada:** Estrutura intermediária resultante da análise sintática.
- **Saída:** Estrutura intermediária com tabelas de símbolos e variáveis, listagem de código.

```
generate(List) ->
```

```
  P = #gcp{},
```

```
  C = fun(Elem, Accin) ->
```

```
    {Code, A, B} = Elem,
```

```
    gc_stat(Accin, Code, A, B)
```

```
end,
```

```
io:format("Gerando codigo...\n"),
```

```
R = lists:foldl(C, P, List),
```

```
add_code(R, return, 0, 1, 0).
```

Geração de código

- **Exemplo:** Geração de código trivial com identificação de globais.
- **Código fonte**
a=7
local b=6
local c=7
- **Saída do parser.**
{ok, [
 {assign, {name, a, 1}, {integer, 1, 6}},
 {localassign, {name, b, 2}, {integer, 2, 6}},
 {localassign, {name, c, 3}, {integer, 3, 7}}
]}

Geração de código

- **Exemplo:** Geração de código trivial com identificação de globais.
- Código gerado, são ilustradas as tabelas de símbolos locais e constantes.

```
c      = [ {loadk,      0, 1},  
           {setglobal, 0, 0},  
           {loadk,      0, 2},  
           {loadk,      1, 1}           ],
```

```
l      = [ {0, {var, name, b}},  
           {1, {var, name, c}}           ],
```

```
k      = [ {0, {const, name, a}},  
           {1, {const, integer, 7}},  
           {2, {const, integer, 6}}      ],
```


Geração de código

- **Exemplo:** Expressão aritmética e print do resultado.

```
local a = 2
local b = 4
a = a + 4 * b - a / 2 ^ b % 3
print("Resultado: ",a)
```

- Tabela de constantes e variáveis locais.

```
k = [ #sym{n = 'Resultado: ',t = string,r = 4,s = global},
      #sym{n = print,t = string,r = 3,s = global},
      #sym{n = 3,t = number,r = 2,s = global},
      #sym{n = 4,t = number,r = 1,s = global},
      #sym{n = 2,t = number,r = 0,s = global}]

l = [ #sym{n = b,t = string,r = 1,s = local},
      #sym{n = a,t = string,r = 0,s = local}]
```

Geração de código

- Listagem de código

```
c=[{loadk,0,0,0},    R(0) <- k(0) = 2
   {loadk,1,1,0},    R(1) <- k(1) = 4
   {pow,8,256,1},    R(8) <- k(0) ^ R(1) = 2 ^ 4 = 16
   {dive,5,0,8},     R(5) <- R(0) / R(8) = 2 / 16 = 0.125
   {mod,3,5,258},    R(3) <- R(5) % k(2) = 0.125 % 3 =
                       = 0.125

   {mul,5,257,1},    R(5) <- k(1) * R(1) = 4 * 4 = 16
   {add,2,0,5},      R(2) <- R(0) + R(5) = 2 + 16 = 18
   {sub,0,2,3},      R(0) <- R(2) - R(3) = 18 - 0.125 =
                       = 17.875

   {loadk,4,4,0},    R(4) <- k(4) = "Resultado: "
   {move,5,0,0},     R(5) <- R(0) = 17,875
   {getglobal,3,3,0}, R(3) <- k(3) = @"print"
   {call,3,3,1},     CALL R(3) PARAMS = R(3+1) .. R(3+2)
                       @"print"("Resultado :", 17.875)

   {return,0,1,0}]
```

Montador do *bytecode*

- **Entrada:** Estrutura intermediária resultante da geração do código.
- **Saída:** Sequência binária compatível com VM e desmontador.

monta(P) ->

```
Conf = #conf{version = 16#51, format = 0, endianness  
        = 1, sint = 4, ssize_t = 4, sinstr = 4,  
        slua_number = 8,integral = 0 },
```

```
B1 = dump_header(Conf),  
B2 = dump_functionhdr(Conf, P),  
B3 = dump_code(Conf, P),  
B4 = dump_constants(Conf, P),  
B5 = dump_funcproto(Conf, P),
```

```
<<B1/binary, B2/binary, B3/binary, B4/binary,  
    B5/binary, 0:32, 0:32, 0:32>>.
```

Montador do *bytecode*

- **Detalhes:** Geração da lista binária com instruções. Fold na lista de instruções.

```
dump_code (Conf, P) ->  
    Size_t = Conf#conf.ssize_t,  
  
    C = fun (Elem, Accin) ->  
        Opc = dump_opcode (Elem),  
        <<Accin/binary, Opc/binary>>  
  
    end,  
  
    Code = lists:foldl (C, <<>>, P#gcp.c),  
    Szcode = P#gcp.sc,  
  
    <<Szcode:Size_t/little-unsigned-unit:8,  
        Code/binary>>.
```

Montador do *bytecode*

- **Detalhes:** Geração da lista binária com instruções. Formato binário do opcode.

```
dump_opcode({Code, A, B, C}) ->
    Opcode = oputil:opnum(atom_to_list(Code)),

    case oputil:optype(Opcode) of
        sABC ->
            <<Op:1/little-unsigned-unit:32>> =
            <<B:9, C:9, A:8, Opcode:6>>,
            <<Op:32>>;
        sABx ->
            <<Op:1/little-unsigned-unit:32>> =
            <<B:18, A:8, Opcode:6>>,
            <<Op:32>>
    end.
```

Montador do *bytecode*

- **Execução:** Gravação em arquivo e execução pela VM lua.
- Comandos no runtime erlang

```
rr(luagc3).  
File = "testearith.lua".  
{ok, Bin} = file:read_file(File).  
{ok, R, _} = luascan:string(binary_to_list(Bin)).  
{ok, L} = luaparse:parse(R).  
PP = luagc3:generate(L).  
P=PP#gcp{srcname="@++File}.  
luamonta:monta(File++"c", P).
```

- Execução do arquivo no interpretador lua

```
lua testearith.luac
```

```
Resultado:          17.875
```

Referências

- [1] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, *Lua - an extensible extension language*, Software: Practice & Experience 26 #6 (1996) 635–652.
- [2] Armstrong, J. (2007). *A history of Erlang*. Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III. pp. 6–1.
- [3] K. H. Man, *A No-Frills Introduction to Lua 5.1 VM Instructions*, <http://luaforge.net/docman/83/98/>. Acesso em 10/11/2011.
- [4] R. Virding, *leex*, <https://github.com/rvirding/leex>. Acesso em 22/10/2011.
- [5] C. W. Welin. *yecc: LALR-1 Parser Generator*. <http://www.erlang.org/doc/man/yecc.html>. Acesso em 22/10/2011.