

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

Manual de utilização dos programas – Projeto da disciplina PCS5730, turma de 2011.

Dezembro de 2011

Ricardo Ryoiti Sugawara Júnior

## **1. Introdução**

A construção de um desmontador de *bytecode* permite exercitar alguns conceitos de reconhecedores baseados em autômatos de pilha, além de oferecer informações sobre o conjunto de instruções e modos de operação da máquina alvo, virtual ou não. A linguagem Lua [1] implementa uma máquina virtual baseada em pilhas. O sistema é inspirado no *p-code* originado na linguagem Pascal [2], oferecendo uma plataforma hipotética para a geração de código independente de máquina. O formato do *bytecode* final é relativamente simples, contendo três formatos de instruções de 32 bits, compreendendo um total de 38 *opcodes* na sua versão 5.1 [3]. Estruturas aninhadas formam blocos de funções, que abrigam as tabelas de símbolos, informações de depuração e a listagem das instruções do mesmo<sup>1</sup>.

As informações obtidas acerca da estrutura do *bytecode*, durante o desenvolvimento deste demontador, favorecem a construção de um compilador completo para esta máquina virtual, complementando o aprendizado de uma disciplina de compiladores, já que se cobre uma variedade dos aspectos usualmente abordados.

O projeto desenvolvido no presente trabalho consiste em um demontador completo para o *bytecode* Lua (capaz de reconhecer todas as informações presentes no arquivo binário) e um compilador simples para um subconjunto restrito da mesma linguagem. A seguir serão descritos os módulos que os constituem, seus procedimentos de compilação e utilização, bem como os resultados de algumas execuções de desmonte e reconhecimento de binários pré-compilados, e os passos de compilação de alguns programas desde os seus códigos fonte.

## **2. Requisitos, descrição sumária e compilação dos módulos do projeto**

Os programas foram escritos na linguagem Erlang [4], uma linguagem funcional (portanto, não algorítmica) multi-plataforma. O autor entende que este tipo de linguagem permite uma melhor aproximação dos conceitos formais da disciplina e, uma vez que também oferece ferramentas para a construção de analisadores léxicos [5] e sintáticos [6], facilita e é bem aderente à construção deste tipo de programa.

Assim, os requisitos para a execução dos programas são os seguintes:

2.1. Compiador e runtime Erlang/OTP, versão 14B04.

[http://www.erlang.org/download\\_release/12](http://www.erlang.org/download_release/12)

---

<sup>1</sup> Para maior detalhamento sugere-se como referência a apresentação do projeto da disciplina (arquivo [apresentacao-pcs5730-rsugawara.pdf](#)) e o artigo [3].

## 2.2. Plataforma da linguagem Lua, versão 5.1.4.

<http://code.google.com/p/luaforwindows/>

As URLs apresentadas oferecem instaladores automáticos para ambas as plataformas. Será necessário baixar os arquivos **LuaForWindows\_v5.1.4-45.exe** e **otp\_win32\_R14B04.exe** e instalá-los. Serão criados ícones no *desktop* do usuário para invocar os interpretadores e runtimes.

Descompacte o pacote **compilador-pcs5730-rsugawara.zip** para uma pasta (sugestão: **c:\comp**). Serão extraídos os seguintes arquivos e diretórios:

<code>script.txt</code>	Listagem de comandos para compilação e utilização.
<code>luascan.xrl</code>	Definição para o gerador de analisador léxico2.
<code>luaparse.yrl</code>	Definição para o gerador de analisador sintático2.
<code>oputil.erl</code>	Listagem de opcodes e seus tipos.
<code>luaop.erl</code>	Otimizador elementar de operações aritméticas
<code>luadesmonta.erl</code>	Desmontador completo de bytecode Lua.
<code>luagc3.erl</code>	Gerador de código.
<code>luamonta.erl</code>	Montador do binário do bytecode no formato da VM.
<code>recs.hrl</code>	Especificação de estruturas usadas no compilador.
<code>Testes-analisadores\</code>	Diretório com fontes escritos em subconjunto da linguagem Lua para testes de análise léxica e sintática (inclui loops, funções, e todas as outras estruturas e chamadas comuns).
<code>testes-compilador\</code>	Diretório com fontes escritos para testes totalmente compiláveis (códigos simples – funções aritméticas, chamadas de funções, variáveis locais e globais, etc).
<code>testes-desmontador\</code>	Diretório com bytecodes (e seus fontes) para testes com o desmontador. Contém códigos mais complexos provenientes da distribuição da linguagem lua, plenamente tratáveis pelo desmontador.

O processo de compilação dos programas do projeto precisa ser feito apenas uma vez. A listagem do código fonte abaixo (extraída do arquivo **script.txt**) apresenta os comandos para o *runtime* do Erlang gerar os arquivos binários no formato apropriado para a execução.

```
%%
%% PARTE I - COMPILAÇÃO DOS MÓDULOS
%%
%
% Substitua com o caminho onde o pacote foi extraído.
%
cd("c:/comp") .
%
% Gera analisador lexico - leex
%
leex:file("luascan",[dfa_graph]).
```

<sup>2</sup> Sugere-se como referência a prova da disciplina [7], questão 6 (Automatização da construção de compiladores).

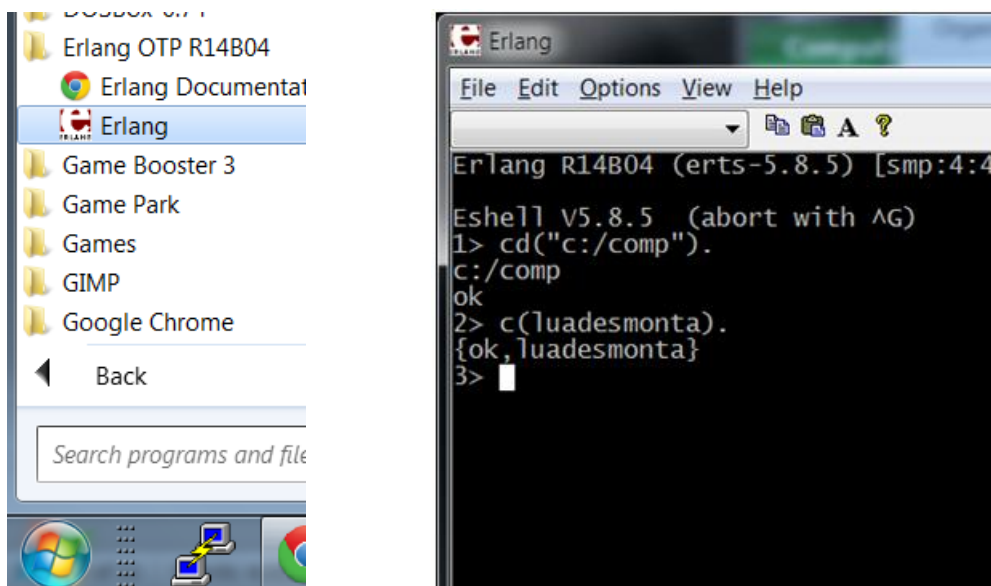
```

%
% Gera analisador sintático - yacc
%
yecc:file("luaparse").
%
% Compila pacotes:
%
c(luascan).
c(oputil).
c(luaop).
c(luadesmonta).
c(luaparse).
c(luagc3).
c(luamonta).

```

Para utilizá-lo, carregue o Erlang Shell (**werl.exe**) e execute os comandos indicados. As figuras abaixo ilustram o carregamento do shell e os comandos para a compilação apenas do módulo **luadesmonta.erl**.

Sugere-se abrir o arquivo **script.txt** no bloco de notas e utilizar as teclas **Ctrl+C** após selecionar os comandos desejados e **Ctrl+V** no shell do Erlang para executá-los.



A cada compilação com sucesso, (instrução **c** do *shell*) é gerada uma *tupla* com as informações: {ok, *nome do módulo*}. Um arquivo *.beam* estará disponível no diretório, para cada módulo, após este procedimento. Esses arquivos são binários interpretáveis pelo *runtime*.

Os comandos dos módulos podem ser chamados no *shell* com a sintaxe "**nome\_do\_módulo:função(parâmetros)**". As instruções apresentadas anteriormente para os módulos **leex** e **yecc** ilustram este esquema de funcionamento, onde ambos geram os códigos fontes dos reconhecedores a partir dos arquivos de definição (**luascan.xrl** e **luaparse.yrl**). As duas seções seguintes apresentam a utilização dos módulos dos dois programas desenvolvidos (desmontador e compilador).

### 3. Desmontador completo de *bytecode*.

O módulo **luadesmonta** é um desmontador completo para o Bytecode da linguagem Lua. É capaz de reconhecer todas as estruturas do binário, que são descritas com detalhes na referência [3].

Sua implementação é baseada num autômato de pilha estruturado. A função de entrada do desmontador é uma função descrita do seguinte modo:

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

A função `parse` recebe como parâmetro um nome de arquivo (`Filename`), corresponde ao bytecode compilado (com o compilador do projeto ou com o compilador oficial da linguagem Lua – `luac.exe`). Os estados do autômato são implementados como rotinas declarativas que consomem uma sequência de *bits* (o conteúdo do *bytecode* é convertido para um “bitstream” e inserido ao analisador). Cada função consome um tipo de estrutura (representação de strings, listas de constantes, instruções, informações de debug, etc), conforme apresentado na listagem abaixo, que é uma seção do código de leitura de listas de instruções:

```
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]);
```

Existe uma hierarquia de complexidade, conforme descrição das estruturas de dados na referência [3]. Por exemplo, uma representação de string é uma chamada de sub-máquina (rotina `get_string`) a partir de um empilhamento de uma outra rotina de nível superior (por exemplo, uma lista de constantes, que pode conter em seu anterior uma representação de string).

Desse modo, a pilha do autômato é a própria pilha de execução das sub-máquinas (rotinas) do desmontador. A listagem a seguir ilustra o empilhamento durante o processamento:

```
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}),
```

Observe que a função `parse` com a diretiva `functionblock` é chamada recursivamente. A diferença entre elas é que o parâmetro `R` da segunda chamada é a cadeia de entrada remanescente.

### 3.1. Modo de utilização e exemplo.

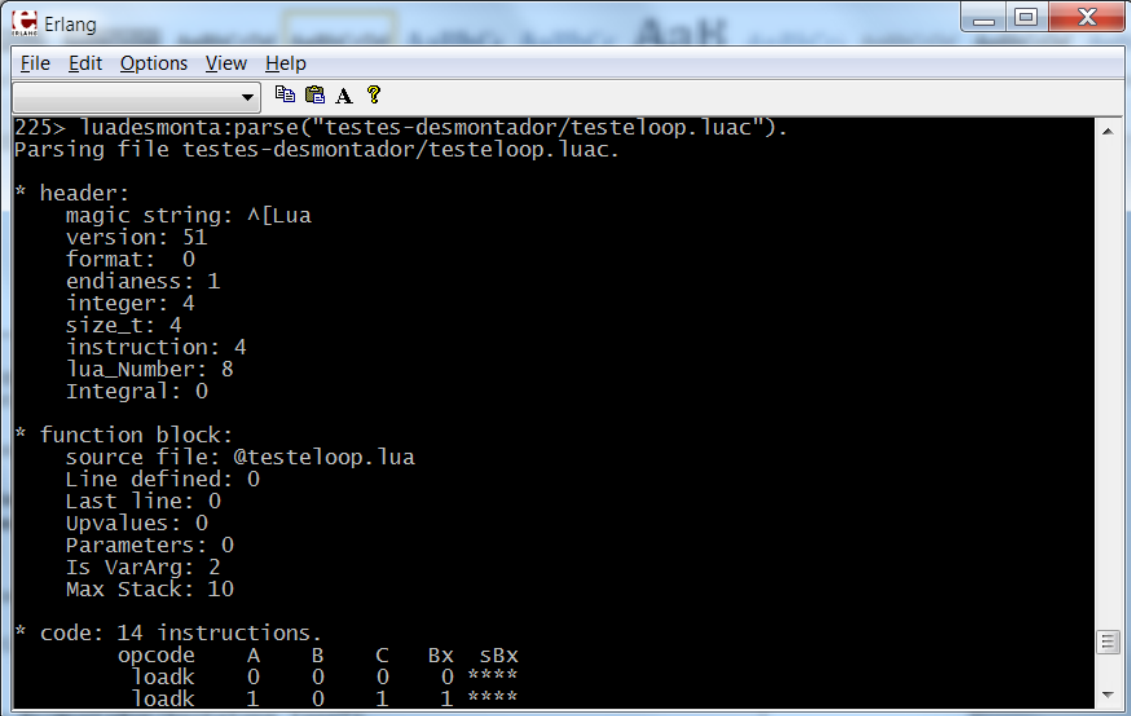
Para utilizar o desmontador basta chamar a função `parse` do módulo `luadesmonta` com um parâmetro apontando para um programa lua compilado (normalmente com extensão `.luac`). Esta compilação se dá com o programa `luac.exe`, conforme o trecho extraído do arquivo `script.txt`.

```

%%
%% PARTE II - TESTES COM O DESMONTADOR
%%
%%
% 1o. Compilação dos códigos fontes de exemplo
%     provenientes da distribuição da linguagem Lua.
%
% Executar no prompt de comando (DOS) para compilar
%
cd c:\comp\testes-desmontador
luac -o testeloop.luac testeloop.lua
    ( ... suprimido ... )
%
% 2o. Execução do desmontador com bytecode previamente compilado
%
% Executar no shell do erlang um a um para desmontar
%
cd("c:/comp") .
luadesmonta:parse("testes-desmontador/testeloop.luac") .
    ( ... suprimido ... )

```

A figura abaixo ilustra a chamada desses comandos no *shell* do Erlang. No diretório `testes-desmontador` existem diversos bytecodes pré-compilados e seus códigos fontes. Todos eles (exceto o `testeloop.lua` apresentado a seguir) são exemplos extraídos da própria distribuição da linguagem Lua.



```

Erlang
File Edit Options View Help
225> luadesmonta:parse("testes-desmontador/testeloop.luac").
Parsing file testes-desmontador/testeloop.luac.

* header:
  magic string: ^[Lua
  version: 51
  format: 0
  endianness: 1
  integer: 4
  size_t: 4
  instruction: 4
  lua_Number: 8
  Integral: 0

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

* code: 14 instructions.
      opcode   A    B    C   Bx   sBx
      loadk    0    0    0    0   ****
      loadk    1    0    1    1   ****

```

O resultado do desmontador é uma sequência de texto com as informações que seguem a estrutura do bytecode. Para ilustrar, o seguinte programa compõe o `testeloop.lua`:

```

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

```

Uma inspeção deste fonte revela dois loops (nas variáveis globais `i` e `j`, com os símbolos globais `1` e `10`, chamando a função global `print`). Há apenas um bloco de função (estrutura de bloco de função no bytecode não será aninhada).

A primeira estrutura é o cabeçalho do arquivo, que contém informações sobre o ambiente de compilação, nome do arquivo, etc.

```
Parsing file testes-desmontador/testeloop.luac.
```

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

O único bloco de função segue com seu cabeçalho e listagem de instruções<sup>3</sup>:

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

* code: 14 instructions.
  opcode   A    B    C    Bx   sBx
  loadk    0    0    0     0  ****
  loadk    1    0    1     1  ****
  loadk    2    0    0     0  ****
  forprep   0  256    7  ****    8
  loadk    4    0    0     0  ****
  loadk    5    0    1     1  ****
  loadk    6    0    0     0  ****
  forprep   4  256    2  ****    3
  getglobal 8    0    2     2  ****
  mul      9    3    7  1543  ****
  call     8    2    1  1025  ****
  forloop   4  255  507  ****   -4
  forloop   0  255  502  ****   -9
  return    0    1    0   512  ****
```

E em seguida é apresentada a listagem de constantes e funções daquele bloco. A estrutura é `{tipo, valor}`, onde `tipo` denota a representação do tipo de dado no *bytecode* (3 para números, 4 para *strings*). Valor é o valor numérico ou uma *tupla* representando *string*, por exemplo.

```
* constant: 3 constants.
  type value
  {3,1.0}
  {3,10.0}
  {4,{str,6,"print"}}

* functions: 0 functions.
```

---

<sup>3</sup> A interpretação da listagem dessas instruções está disponível no slide 14 da apresentação do projeto.

Finalmente, são apresentadas as informações de *debug*, que associa cada instrução da listagem à posição do código fonte associado à mesma, bem como apresenta a listagem dos símbolos locais. No exemplo, a primeira instrução (`loadk 0 0 0`) aparece na linha 1 do fonte. A estrutura dos símbolos locais é: `{{tipo tupla, tamanho, nome}, registro, linha}`, onde *tipo* é uma *string* (str para strings) para identificar o tipo do dado. `registro` indica o registrador da máquina virtual onde está guardado e `linha` denota em que linha do código fonte o símbolo é criado.

```
* source line informations: 14
1
1
1
( ... suprimido ... )
2
1
5

* 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}
```

A seguir são apresentados os componentes do compilador desenvolvido, seus modos de utilização e comentários.

## 4. Compilador

O compilador foi desenvolvido com quatro passos básicos: Análise léxica, com o auxílio da ferramenta `leex` [5], análise sintática com analisador gerado pelo `yacc` [6], geração de código e otimização, montador do bytecode executável. Cada um desses passos é integrado por meio de representações intermediárias que empregam as estruturas de dados bem definidas do *runtime* da linguagem Erlang (listas e tuplas). A seguir é apresentado um sumário da implementação de cada um desses módulos, seu modo de utilização e exemplos de execução.

### 4.1. Analisador léxico

As definições básicas do analisador léxico estão no arquivo `luascan.xrl` (vide ref. [7], questão 6, onde se descreve o arquivo de entrada do `lex`, compatível) começam com a marcação de nomes, espaços em branco e dígitos. Estas marcações são agrupadas em regras que identificam strings citadas (entre apóstrofes e aspas), números inteiros e representações de ponto flutuante. É também nesta etapa que são identificados (com função auxiliar) quais das strings são ou não palavras reservadas. A seguir são apresentadas as expressões regulares.

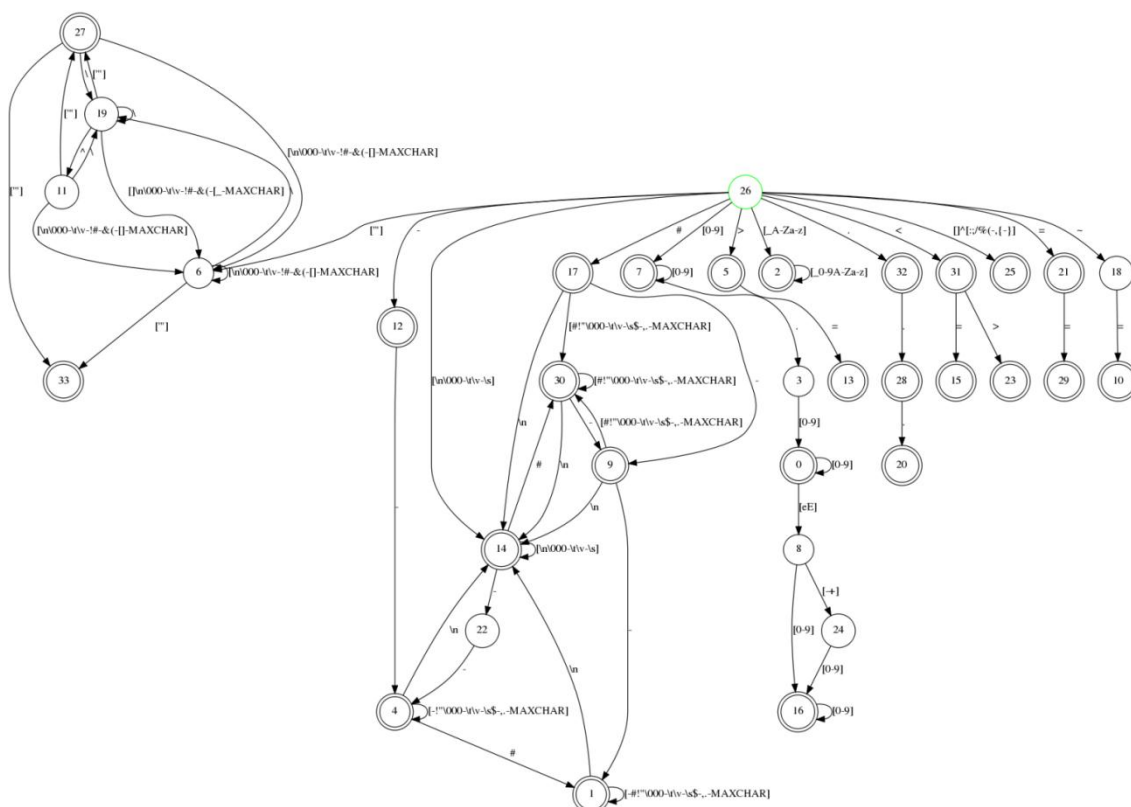
```
D = [A-Za-z_][A-Za-z0-9_]*
WS = ([\000-\s]|--.*|#.*)
D = [0-9]
```

Observe que identificadores iniciados com `--` e `#` são comentários na linguagem Lua e por isto vão para a regra `WS` (*whitespace*). As regras de agrupamento são definidas em seguida. Como exemplo, segue a definição da representação em ponto flutuante:

```
%representacao de ponto flutuante
{D}+\\. {D}+((E|e) (\\+|\\-)? {D})+? :
```

O `leex` gera um scanner em forma de autômato determinístico finito. A máquina pode ser ilustrada por meio de um grafo gerado no arquivo `luascan.dot` com a diretiva `dfa_graph` ao invocar o `leex` para codificar o reconhecedor. A figura<sup>4</sup> é apresentada a seguir.

```
leex:file("luascan", [dfa graph]).
```



A saída do analisador sintático é uma tupla no formato `{ok, Lista, Linhas}` onde `Lista` é uma sequência de tuplas que definem os átomos identificados, e `linhas` denota a quantidade de linhas reconhecidas do código fonte.

Para todos os exemplos a seguir, será usado o programa `testes-compilador/teste3.lua`, sua listagem é apresentada a seguir. O analisador sintático reconhece todas as convenções léxicas da linguagem Lua, listadas em [8] sec. 2.1.

```
local a = 2
local b = 4
a = a + 4 * b - a / 2 ^ b % 3
```

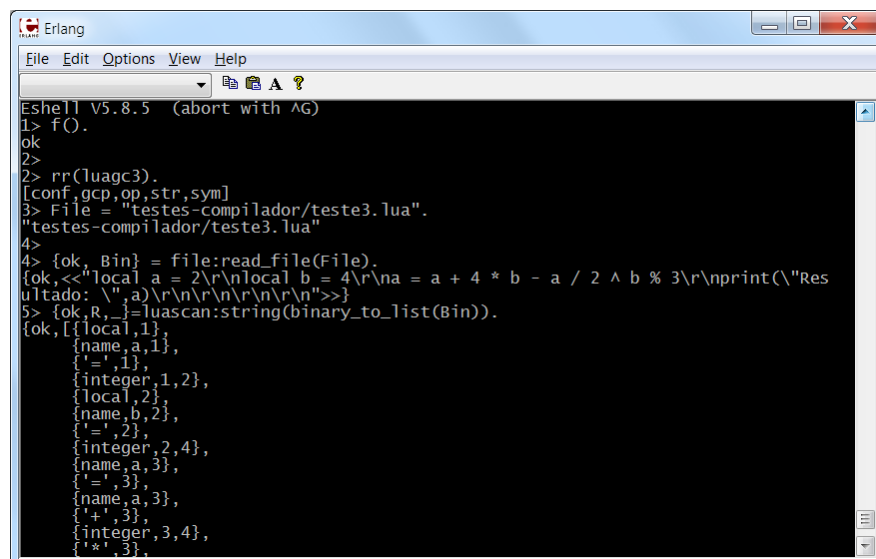
<sup>4</sup> A representação do autômato gerado na linguagem DOT, que pode ser tratada com a ferramenta GraphViz para gerar imagens. <http://www.graphviz.org>



```
print("Resultado: ",a)
```

Ao aplicar os comandos do `script.txt` listados abaixo ao shell, o analisador será executado.

```
%%
%% PARTE III - TESTES COM O COMPILADOR
%%
%
% Executar os comandos abaixo para limpar
% variáveis do shell do Erlang e carregar
% estruturas de dados.
%
f().
rr(luagc3).
%
% Defina arquivo de entrada (fonte) e gere bitstream
%
File = "testes-compilador/teste3.lua".
{ok, Bin} = file:read_file(File).
```



```
Eshell V5.8.5 (abort with ^G)
1> f().
ok
2>
2> rr(luagc3).
[conf,gcp,op,str,sym]
3> File = "testes-compilador/teste3.lua".
"testes-compilador/teste3.lua"
4>
4> {ok, Bin} = file:read_file(File).
{ok,<<"local a = 2\r\nlocal b = 4\r\na = a + 4 * b - a / 2 ^ b % 3\r\nprint(\"Res
ultado: \",a)\r\n\r\n\r\n\r\n\r\n">>}
5> {ok,R}_=luascan:string(binary_to_list(Bin)).
{ok,[{local,1},
{name,a,1},
{'=',1},
{integer,1,2},
{local,2},
{name,b,2},
{'=',2},
{integer,2,4},
{name,a,3},
{'-',3},
{name,a,3},
{'/',3},
{integer,3,2},
{'^',3},
{name,b,3},
{'%',3},
{integer,3,3},
{name,print,4},
{'(',4},
(...suprimido...),
8}
```

O resultado parcial desta análise é listado a seguir.

```
{ok,[{local,1},      {name,a,1},      {'=',1},      {integer,1,2},      {local,2},
      {name,b,2},      {'=',2},      {integer,2,4},      {name,a,3},      {'=',3},
      {name,a,3},      {'+',3},      {integer,3,4},      {'*',3},      {name,b,3},
      {'-',3},      {name,a,3},      {'/',3},      {integer,3,2},      {'^',3},
      {name,b,3},      {'%',3},      {integer,3,3},      {name,print,4},      {'(',4},
      (...suprimido...),
      8]}
```

A lista `R` resultante pode então ser aplicada ao próximo passo, de análise sintática.

## 4.2. Analisador sintático

A gramática da linguagem, definida em [8], sec. 8, juntamente com a precedência dos operadores, apresentada em [8], sec. 2.5.6, é aplicada a ao gerador de *parsers* LR `yacc` [6]. Foram realizadas algumas simplificações para reduzir a complexidade da implementação. Por exemplo, não são aceitas listas de variáveis (como no código `a,b,c = 1,2,3`, apenas `a =`

1; b = 2; c = 3, apenas blocos condicionais simples como `if condição then bloco end`).

No arquivo de definições do yecc, denominado `luaparse.yrl`, são configurados os agrupamentos sintáticos associados à gramática. Observe que, apesar das gramáticas de atributos possibilitarem a geração de código diretamente a partir da sintaxe, o autor optou por gerar apenas uma nova representação intermediária que bem descrevesse a árvore sintática do programa. Para ilustrar o fato, a seguir é apresentada o agrupamento básico denominado `stat` (bloco estático), com o simples retorno das respectivas estruturas nas “ações semânticas”.

```
%%
%% stat
%%
stat -> var '=' exp                : {assign,'$1','$3'}.
stat -> local var '=' exp          : {localassign,'$2','$4'}.
stat -> functioncall              : {'$1'}.
stat -> do block end               : {do,'$2'}.
stat -> while exp do block end     : {while,'$2','$4'}.
stat -> repeat block until exp     : {repeat,'$2','$4'}.
stat -> if exp then block end      : {ifblock,'$2','$4'}.
stat -> for name '=' exp ',' exp do block end : {for2,{'$2','$4','$6','$8'}.
stat -> function name funcbody    : {function,'$2','$3'}.
```

Portanto, a aplicação da representação léxica resultante ao analisador deve oferecer um retorno no formato `{ok, Arvore}`, conforme o exemplo abaixo, ainda referente ao arquivo `testes-compilador/teste3.lua`.

```
7> {ok, L} = luaparse:parse(R) .
{ok, [{localassign, {name,a,1}, {integer,1,2}},
      {localassign, {name,b,2}, {integer,2,4}},
      {assign,
        {name,a,3},
        {binop,
          {'-',3},
          {binop,
            {'+',3},
            {var, {name,a,3}},
            {binop, {'*',3}, {integer,3,4}, {var, {name,b,3}}}},
          {'%',3},
          {binop,
            {'/',3},
            {var, {name,a,3}},
            {binop, {'^',3}, {integer,3,2}, {var, {name,b,3}}}},
          {integer,3,3}}}},
      {call,
        {var, {name,print,4}},
        {args, [{string,4,'Resultado: '}, {var, {name,a,4}}]}}]}
```

A figura a seguir demonstra a utilização do shell erlang, após a análise léxica retornada em `R`, que resultou no retorno do analisador sintático em `L`.

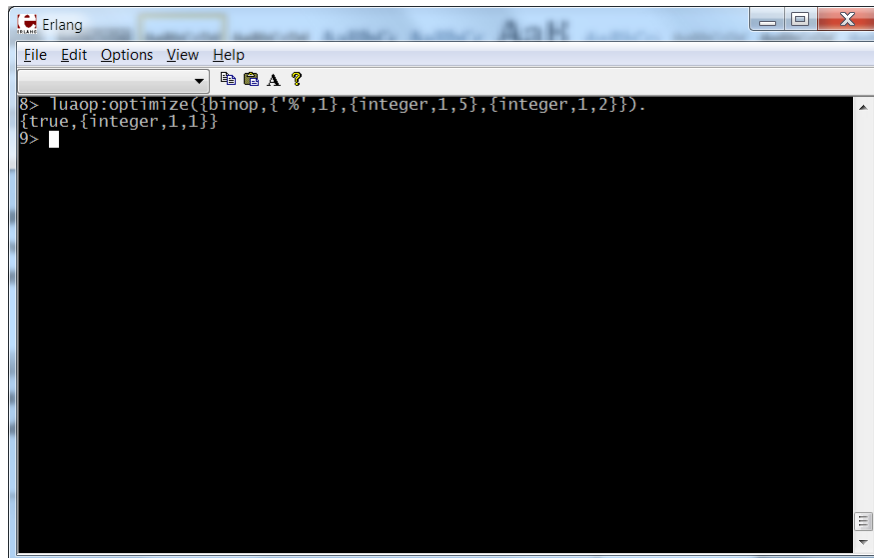
### 4.3. Otimizações e gerador de código

A geração de código é responsável por montar as instruções necessárias para a execução do programa descrito pela árvore sintática. Neste passo, foram implementadas otimizações triviais de operações aritméticas (*folding* de expressões com constantes), nos arquivos `luaop.erl`.

A função `luaop:optimize()` identifica se as expressões aritméticas são operações unárias de negativo ou binárias em constantes. Nestes casos, efetua a operação em tempo de execução e devolve ao gerador de código o resultado. Segue uma seção do código de otimização:

```
optimize(Exp) ->
case Exp of
%%
%% inteiro, inteiro
%%
{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}};
{binop, {'/', L}, {integer, _, A}, {integer, _, B}} ->
{true, {integer, L, A div B}};
(suprimido)
```

Esta função `luaop:optimize()` é chamada pelo gerador de código durante a avaliação de expressões aritméticas (rotina `luagc3:gc_exp`). Um exemplo de otimização sobre uma operação de módulo de divisão inteira é apresentada na figura a seguir.



A geração de código está apoiada na estrutura denominada `gcp`, responsável por abrigar as listas e seus respectivos contadores (`c` – lista de instruções, `k` – lista de constantes, `l` – lista de variáveis locais, `r` – último registrador usado, `ctx` – contexto da chamada do gerador, `srcname` – arquivo fonte de onde veio o bloco de função a ser gerado).

```
-record(gcp, {
    k = [],
    sk = 0,
    c = [],
    sc = 0,
    l = [],
    sl = 0,
    r = 0,
    dr = 0,
    ctx = any,
    srcname
} ).
```

O ponto de entrada do gerador é a função `luagc3:generate()`. Como argumento, é colocada a saída do analisador sintático.

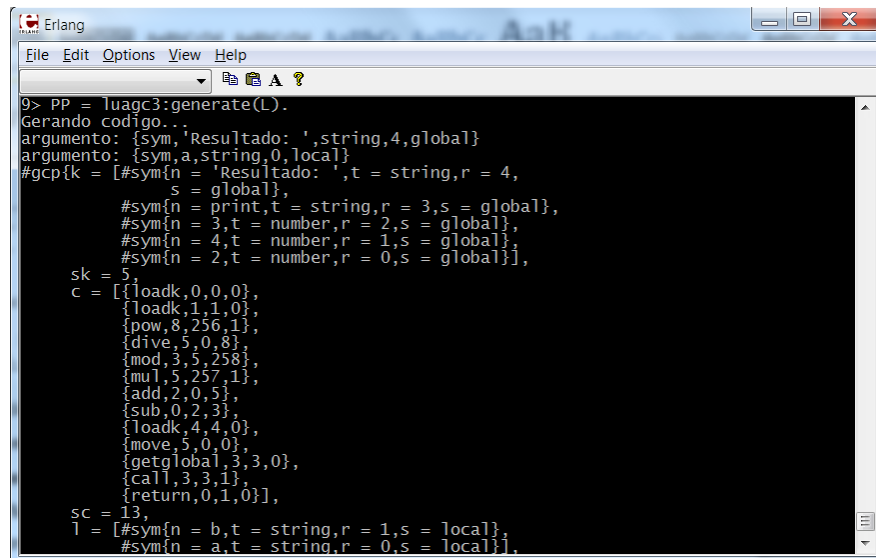
```
generate(List) ->
  P = #gcp{},
  C = fun(Elem, Accin) ->
    {Code, A, B} = Elem,
    gc_stat(Accin, Code, A, B)
  end,

  io:format("Gerando codigo...~n"),

  R = lists:foldl(C, P, List),
  add_code(R, return, 0, 1, 0).
```

A rotina cria uma instância da estrutura `gcp` e, para cada membro da árvore, aplica a função anônima `C`, responsável por invocar o gerador de código para aquele bloco estático (`gc_stat`). Isto é feito por meio de uma recursão do tipo *tail* e, ao final, é gerado um código de retorno obrigatório pela VM Lua (`return 0 1`). A saída é a própria estrutura `gcp`, porém com seus membros (listas e contadores) modificados.

Para gerar esta estrutura, aplique ao `generate` a saída `L` do analisador sintático e grave o resultado numa variável, por exemplo `PP`. Segue um exemplo para o arquivo `testes-compilador/teste3.lua`.



```
9> PP = luac3:generate(L).
Gerando código...
argumento: {sym,'Resultado:',string,4,global}
argumento: {sym,a,string,0,local}
#gcp{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}],
sk = 5,
c = [{loadk,0,0,0},
     {loadk,1,1,0},
     {pow,8,256,1},
     {div,5,0,8},
     {mod,3,5,258},
     {mul,5,257,1},
     {add,2,0,5},
     {sub,0,2,3},
     {loadk,4,4,0},
     {move,5,0,0},
     {getglobal,3,3,0},
     {call,3,3,1},
     {return,0,1,0}],
sc = 13,
l = [#sym{n = b,t = string,r = 1,s = local},
     #sym{n = a,t = string,r = 0,s = local}]}
```

A interpretação dos resultados foi realizada na apresentação [8], slides 33 e 34. Observe que estão populadas as listas de símbolos `k`, de opcodes `c`, de símbolos locais `l`, etc.

Esta estrutura `PP` resultante é uma representação abstrata das informações necessárias para a execução pela VM Lua, apenas não está gravada num formato binário apropriado. Esta tarefa cabe ao próximo e último passo do compilador, o montador do arquivo binário do *bytecode*.

A parte de geração de código é a mais complexa e trabalhosa, e por este fato sua implementação neste projeto ficou restrita aos casos mais elementares, disponíveis para compilação no diretório `testes-compilador`. É capaz, no entanto, de demonstrar as propriedades mais relevantes para a geração de códigos para uma máquina virtual como a da linguagem Lua.

#### 4.4. Montador do *bytecode* e execução do programa compilado com a VM Lua.

Este módulo `luamonta.erl` realiza o trabalho inverso do desmontador. Ela é responsável por montar um arquivo binário compatível com o lido pela máquina virtual Lua, que possa ser por ela executado. Como entrada, é aplicada a estrutura `gcp` resultante do gerador de código (gravada na variável `PP` no exemplo anterior).

Sua rotina de entrada é a função `luamonta:monta(File, P)`, que recebe como argumento uma estrutura do tipo `gcp` e o nome de um arquivo de destino `File`. O resultado é uma cadeia de bits que é então gravada num arquivo.

```
monta(File, P) when is_list(File) ->
    file:write_file(File, monta(P)).
```

A função `monta` possui uma variante de um único argumento, que efetua a montagem propriamente dita:

```
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),

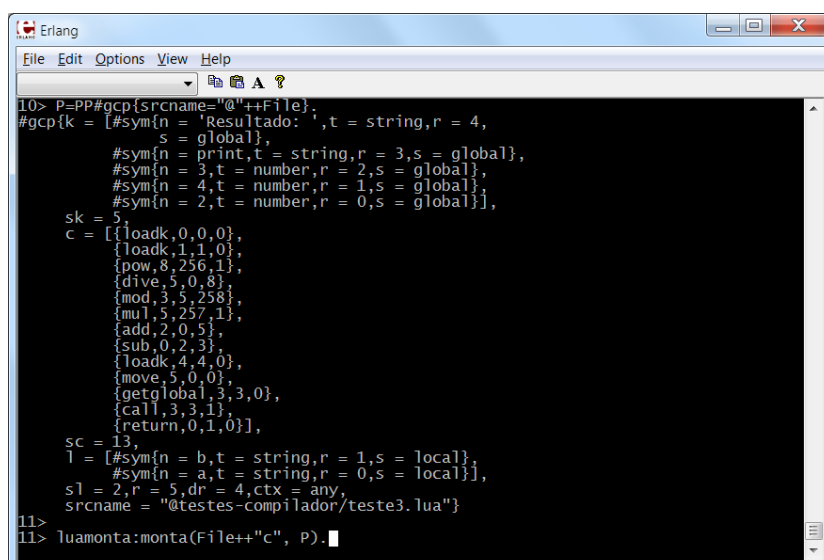
    (informações de print na tela suprimidas)

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

A descarga das sub-cadeias de bits é feita na ordem do especificado pelo formato do bytecode da máquina virtual [3]. Ao final, são colocadas listas vazias relativas às seções opcionais de debug (`0:32, 0:32, 0:32`).

Para executar o montador, deve-se primeiro definir o nome do arquivo na estrutura `gcp` (campo `srcname`) e depois inserir a estrutura na função de entrada. Isto se faz necessário porque a implementação não carrega esta informação desde o analisador léxico. Segue um trecho do `script.txt` relativo a esta parte:

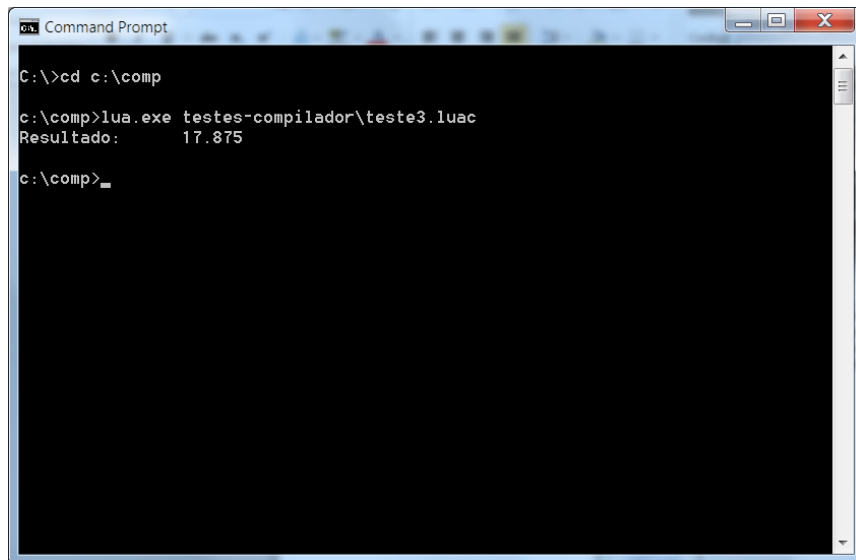
```
%
% MONTADOR DO BYTECODE
%
% Montar arquivo de bytecode a partir
% da estrutura PP. No início é definido o nome
% do código fonte para o cabeçalho (srcname).
% saída será gravada no arquivo extensão .luac.
%
P=PP#gcp{srcname="@++File}.
luamonta:monta(File++"c", P).
```



The screenshot shows the Erlang shell interface with the following code and output:

```
10> P=PP#gcp{srcname="@++File}.
#gcp{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}],
        sk = 5,
        c = [{loadk,0,0,0},
              {loadk,1,1,0},
              {pow,8,256,1},
              {dive,5,0,8},
              {mod,3,5,258},
              {mul,5,257,1},
              {add,2,0,5},
              {sub,0,2,3},
              {loadk,4,4,0},
              {move,5,0,0},
              {getglobal,3,3,0},
              {call,3,3,1},
              {return,0,1,0}],
        sc = 13,
        l = [#sym{n = b,t = string,r = 1,s = local},
              #sym{n = a,t = string,r = 0,s = local}],
        s1 = 2,r = 5,dr = 4,ctx = any,
        srcname = "@testes-compilador/teste3.lua"}
11>
11> luamonta:monta(File++"c", P).
```

Após a compilação será gravado um arquivo com extensão adicionada de “c”, testes-compilador/teste3.luac para o fonte testes-compilador/teste3.lua. Este arquivo pode ser executado diretamente pela máquina virtual lua, como na figura abaixo.



```
Command Prompt
C:\>cd c:\comp
c:\comp>lua.exe testes-compilador\teste3.luac
Resultado:      17.875
c:\comp>
```

## 5. Referências

- [1] Ierusalimschy, R. et al. *The Programming Language Lua*. <http://www.lua.org>. Acesso em 17/11/2011.
- [2] Nori, K.V.; Ammann, U.; Jensen; Nageli, H. (1975); *The Pascal P Compiler Implementation Notes*. Zurich: Eidgen. Tech. Hochschule.
- [3] Man, K. H., *A No-Frills Introduction to Lua 5.1 VM Instructions*, <http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf>. Acesso em 17/11/2011.
- [4] Armstrong, J. (2007). A history of Erlang. Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III. pp. 6–1.
- [5] R. Viriding, leex, <https://github.com/rviriding/leex>. Acesso em 22/10/2011.
- [6] C. W. Welin. yecc: LALR-1 Parser Generator. <http://www.erlang.org/doc/man/yecc.html>. Acesso em 22/10/2011.
- [7] Sugawara, R. *Entrega dos materiais da disciplina PCS5730*. <http://rsuga.sdf.org/pcs5730>. Acesso em 27/12/2010.
- [8] Ierusalimschy, R. et al. *Lua 5.1 Reference Manual*. <http://www.lua.org/manual/5.1/manual.html>. Acesso em 17/11/2011.