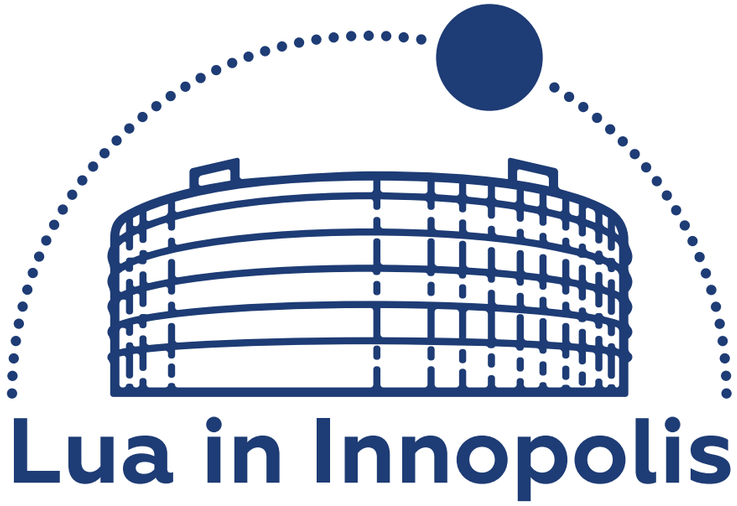


Разработка транслятора Brainfuck в Lua на Lua с использованием библиотеки LPeg

Бобров Алексей

Lua In Innopolis, 26 октября 2016 года

"Hello World!" на Brainfuck

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++
.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.
------.--------.>+.>.
```

Более простой пример "Hello World!"

```
++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++.++++++++++++++++
++++++++++++.+++++++..+++.------------------
---------------------------------------------
---------------.++++++++++++++++++++++++++++
++++++++++++++++++++++++.++++++++++++++++++
++++++.+++.------.--------.------------------
---------------------------------------------
----.-----------------------.
```

Программист работает с массивом байтов, есть указатель на текущую ячейку. Операторы:

- `<` / `>` — переход к предыдущей/следующей ячейке
- `+` / `-` — увеличить/уменьшить значение в текущей ячейке
- `,` / `.` — ввести/вывести значение
- `[тело цикла]` — аналог цикла while с условием "пока в текущей ящейке не 0"

[Почти] программа cat на Brainfuck

```
,[.,]
```

- `,` / `.` — ввести/вывести значение
- `[тело цикла]` — аналог цикла while с условием "пока в текущей ящейке не 0"

Что хотелось хочется увидеть в Lua-коде

Brainfuck:

```
,[.,]
```

Lua:

```lua
local data = setmetatable(
  { array = {} },
  {
    __index = function(self, i) return self.array[i] or 0; end,
    __newindex = function(self, i, value) self.array[i] = value % 256; end,
  }
)
local i = 0
--------------------------------------------------------------------------------
data[i] = string.byte(io.read(1))
while data[i] ~= 0 do
  io.write(string.char(data[i]))
  data[i] = string.byte(io.read(1))
end
```

LPeg

Библиотека для разбора текста. LPeg используют компиляторы нескольких языков программирования, например, Moonscript и Typed Lua.

Зачем LPeg

Зачем для разбора такого простого языка вообще использовать какую-то библиотеку? Зачем это всё?

- just for fun
- попрактиковаться с LPeg
- проще расширять синтаксис языка (если нужно (нужно))

Функции LPeg, которые нам пригодятся

- P — точное совпадение, любой символ, конец ввода, грамматика
- S — любой символ из набора
- V — подстановка правила, определённого в грамматике
- C — захват
- Ct — множественный захват в массив

Начнём конструировать парсер

```
local brainfuck = P{
  "program", -- название правила-корня грамматики
  program = --
}
```

```
local brainfuck = P{
  "program",
  program    = V"expression"^0 * P(-1),
  epxression = --
}
```

```
local brainfuck = P{
  "program",
  program    = V"expression"^0 * P(-1),
  expression = V"operator" + V"loop",
  loop       = --
  operator   = --
}
```

```
local brainfuck = P{
  "program",
  program    = V"expression"^0 * P(-1),
  expression = V"operator" + V"loop",
  loop       = P"[" * V"expression"^0 * P"]",
  operator   = --
}
```

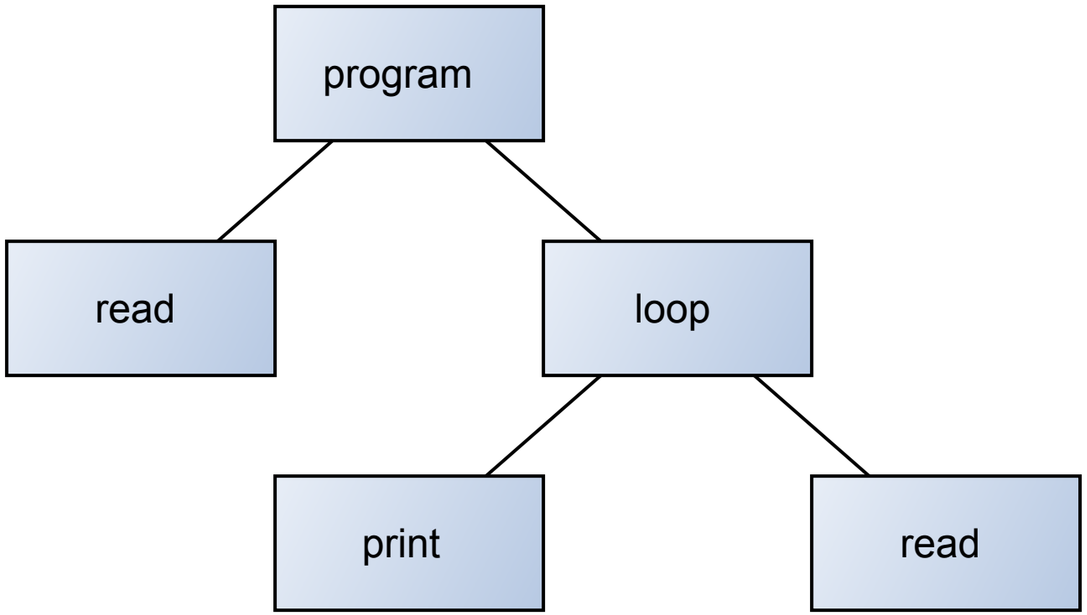
```
local brainfuck = P{
  "program",
  program    = V"expression"^0 * P(-1),
  expression = V"operator" + V"loop",
  loop       = P"[" * V"expression"^0 * P"]",
  operator   = S"<>-+,.",
}
```

```
local brainfuck = P{
  "program",
  program    = V"expression"^0 * P(-1),
  expression = V"operator" + V"loop",
  loop       = P"[" * V"expression"^0 * P"]",
  operator   = S"<>-+,.",
}
```

Полученный парсер успешно распознает любую Brainfuck-программу, но это всё, что он сейчас умеет. Следующий шаг — пометить те части грамматики, которые, необходимо "захватить" — и преобразовать их в AST.

Что примерно хотим получить

```
,[.,]
```



Доработанная грамматика

```
local brainfuck = P{
  "program",
  program    = Ct(V"expression"^0) * P(-1) / program,
  expression = V"operator" + V"loop",
  loop       = P"[" * Ct(V"expression"^0) * P"]" / loop,
  operator   = S"<>-+,." / operator,
}
```

Изменения:

```
local brainfuck = P{
  "program",
  program    = Ct(V"expression"^0) * P(-1) / program,
  expression = V"operator" + V"loop",
  loop       = P"[" * Ct(V"expression"^0) * P"]" / loop,
  operator   = S"<>-+,." / operator,
}
```

Реализуем функции program, loop, oper

```
local operator = function(operator)
  return { type = "operator", data = operator }
end

local loop = function(expr)
  return { type = "loop", data = expr }
end

local program = function(data)
  return { type = "program", data = data }
end
```

Пример AST

```
brainfuck:match(",[.,]")
```

Результат:

```
{
  type = "program",
  data = {
    { type = "operator", data = "," },
    {
      type = "loop", data = {
        { type = "operator", data = "." },
        { type = "operator", data = "," }
      }
    }
  }
}
```

Пришло время для трансляции AST в Lua-код

```lua
local ast2lua

local function indent(n) return string.rep("  ", n); end

local translators = {} -- ???

ast2lua = function(ast, info)
  return translators[ast.type](ast.data, info)
end

local function translate(code)
  local ast = brainfuck:match(code)
  return ast2lua(ast)
end
```

program

```
translators.program = function(expression_list)
  local code = [[
local data = setmetatable(
  { array = {} },
  {
    __index = function(self, i) return self.array[i] or 0; end,
    __newindex = function(self, i, value) self.array[i] = value % 256; end
  }
)
local i = 0
----------------------------
]]
  local info = { loop_depth = 0 }
  for _, expression in ipairs(expression_list) do
    code = code .. ast2lua(expression, info)
  end
  return code
end

---
ast2lua = function(ast, info)
  return translators[ast.type](ast.data, info)
end
```

loop

```
translators.loop = function(body, info)
  local code = indent(info.loop_depth) .. "while data[i] ~= 0 do\n"
  info.loop_depth = info.loop_depth + 1
  for _, expression in ipairs(body) do
    code = code .. ast2lua(expression, info)
  end
  info.loop_depth = info.loop_depth - 1
  code = code .. indent(info.loop_depth) .. "end\n"
  return code
end

---
ast2lua = function(ast, info)
  return translators[ast.type](ast.data, info)
end
```

operator

```
translators.operator = function(op, info)
  return indent(info.loop_depth) .. ({
    ["."] = "io.write(string.char(data[i]))\n",
    [","] = "data[i] = string.byte(io.read(1))\n",
    ["+"] = "data[i] = data[i] + 1\n",
    ["-"] = "data[i] = data[i] - 1\n",
    [">"] = "i = i + 1\n",
    ["<"] = "i = i - 1\n",
  })[op]
end
```

Пример транслированного кода

```
translate(",[.,]")
```

Результат:

```
local data = setmetatable(
  { array = {} },
  {
    __index = function(self, i) return self.array[i] or 0; end,
    __newindex = function(self, i, value) self.array[i] = value % 256; end
  }
)
local i = 0
----------------------------
data[i] = string.byte(io.read(1))
while data[i] ~= 0 do
  io.write(string.char(data[i]))
  data[i] = string.byte(io.read(1))
end
```

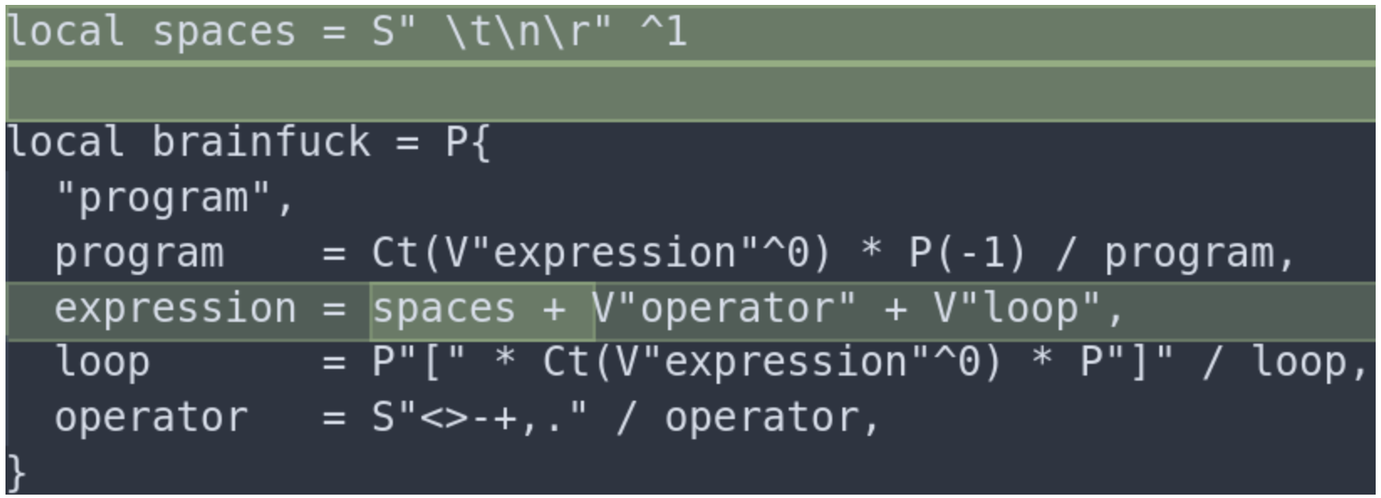
Чего-то не хватает

- Хочется переноса строк и отступов
- Сообщения об ошибках синтаксиса?

Добавляем перенос строк

```
local spaces = S" \t\n\r" ^1

local brainfuck = P{
  "program",
  program    = Ct(V"expression"^0) * P(-1) / program,
  expression = spaces + V"operator" + V"loop",
  loop       = P"[" * Ct(V"expression"^0) * P"]" / loop,
  operator   = S"<>-+,." / operator,
}
```



Добавляем обнаружение ошибок

Подключаем **lpeglabel** вместо **lpeg**, и используем функцию **T**, чтобы поставить метку:

```
local spaces = S" \t\n\r" ^1

local brainfuck = P{
  "program",
  program    = Ct(V"expression"^0) * (P(-1) + T"Error") / program,
  expression = spaces + V"operator" + V"loop",
  loop       = P"[" * Ct(V"expression"^0) * (P"]" + T"Error") / loop,
  operator   = S"<>-+,." / oper,
}
```

```
local brainfuck = P{
  "program",
  program    = Ct(V"expression"^0) * (P(-1) + T"Error") / program,
  expression = spaces + V"operator" + V"loop",
  loop       = P"[" * Ct(V"expression"^0) * (P"]" + T"Error") / loop,
  operator   = S"<>-+,." / operator,
}
```

Добавляем сообщения о синтаксических ошибках

```
local relabel = require "relabel" -- часть библиотеки lpeglabel

---
local function translate(code)
  local ast, label, position = brainfuck:match(code)
  if not ast then
    local line, col = relabel.calcline(code, position)
    return nil, ("[%s] at line %d (col %d)"):format(label, line, col)
  end
  return ast2lua(ast)
end
```

```
local code = [[
,.
-+---+1--
]]

local lua_code, msg = translate(code)
print(lua_code, msg)
```

Результат:

```
nil     [Error] at line 2 (col 7)
```

Hello world

Попробуем запустить Hello World из Википедии

```
local hello_world = [[
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++
.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.
------.--------.>+.>.
]]
local lua_code = translate(hello_world)
loadstring(lua_code)()
```

Результат:

```
Hello World!
```

Ссылки

- Официальный сайт LPeg: http://www.inf.puc-rio.br/~roberto/lpeg
- Статья про LPeg от Leafo: http://leafo.net/guides/parsing-expression-grammars.html
- Статья про Brainfuck: https://en.wikipedia.org/wiki/Brainfuck
- Toy Lisp interpreter in Lua: https://gist.github.com/polymeris/857a7ae31db0d240ef3f
- Moonscript: https://github.com/leafo/moonscript
- Typed Lua: https://github.com/andremm/typedlua
- Реализация Brainfuck из доклада (с дополнениями): https://github.com/Penguinum/brainfuck2lua

Чат Lua In Moscow в Telegram

https://t.me/luainmoscow