www.lua.org

PET

Sumário

1. Introdução
2. Sintaxe
3. Tipos
   3.1. Tipagem
   3.2. Polimorfismo
   3.3. Funções
   3.4. Metamétodos
4. Confiabilidade
   4.1. Exceções
5. Paralelismo
6. Eficiência
7. Portabilidade

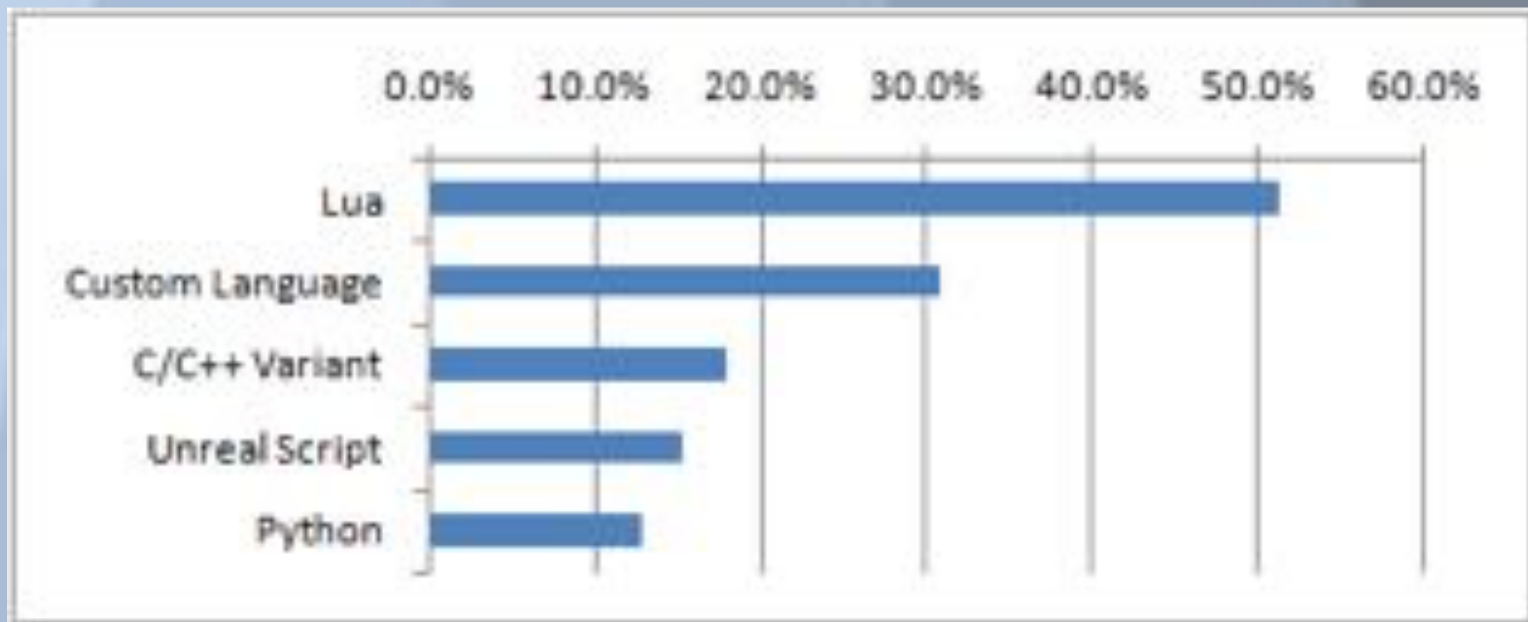
PET

Introdução

- Lua é uma linguagem de programação poderosa, rápida e leve, projetada para estender aplicações.
- Desenvolvida inteiramente no Brasil pela Tecgraf (PUC-Rio) em 1993 para ser usada na Petrobrás
  - Devido a sua eficiência, clareza e facilidade, foi amplamente adotada (desenvolvimento de jogos, controle de robôs, processamento de texto e propósito geral).
  - É atualmente a linguagem de script mais utilizada no desenvolvimento de jogos.
  - http://www.lua.org/uses.html
- Atualmente mantida pelo LabLua (PUC-Rio)

PET

Introdução



http://www.satori.org/2009/03/the-engine-survey-general-results/

Introdução

CRYENGINE® 3

mindstorms education EV3

ANGRY BIRDS

Introdução

- Múltiplos paradigmas
  - Conjunto de características genéricas que podem ser manipuladas para se encaixar em diversos paradigmas
- Pequeno número de estruturas primárias
  - Metamecanismos permitem a implementação de grande parte das estruturas convencionais
- Integrável
  - Extende programas em C and C++, Java, C#, Smalltalk, Fortran, Ada, Erlang e até mesmo em outras linguagens de script tal como Perl e Ruby.

PET

Introdução

- Portável
  - Executa em qualquer ambiente que possua um compilador C.
- Livre
  - Licença MIT - Pode ser usada para qualquer propósito sem qualquer custo.
  - Código aberto.

PET

Sintaxe

- Lua é uma linguagem de formato livre.
- Nomes (ou identificadores) podem ser qualquer cadeia de letras, dígitos e sublinhados, que não iniciam com um dígito e são usados para representar variáveis, campos de tabela e rótulos.

lua (válido)

lua_1 (válido)

_LUA (válido - variáveis usada por Lua)

1lua (inválido)

Sintaxe

- Palavras reservadas:

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

- Lua é case-sensitive. “and” é uma palavra reservada, porém “And” não é.

PET

Sintaxe

- Comentários em LUA
  - Exemplos:

Sintaxe

- Variáveis
  - Variáveis Locais
  - Variáveis Globais
  - Campos de tabelas

- Atribuição
  - <nomeDaVariável> = <valor>
  - <nomeDaTabela> = {<valor1>,<valor2>,...,<valorn>}

Sintaxe

- Tipos de variáveis
  - string
    - “Hello World”
    - ‘Papel’
  - number
    - 2.4
    - 2
  - boolean
    - true, false.

PET

Sintaxe

- Caracteres Especiais
- Exemplos:
  - print (“linha\noutra\n”);
  - print (“aspas simples\’”);

Sintaxe

- Operadores Aritméticos
  - Soma
    - a = 1 + 2;
  - Subtração
    - a = 1- 2;
  - Multiplicação
    - a = 2*3;
  - Divisão
    - a = 2/3;
    - a = 2%3;
  - Negação
    - a = -2;

PET

Sintaxe

- Operadores de Comparação
  - <, >, <=, >= , ==, ~=
- OBSERVAÇÃO
  - Podem ser utilizados para qualquer tipo de valor.
    - a = 2 < 3; OK!
    - a = ‘x’ < ‘y’; OK!
  - CUIDADO AO UTILIZAR PARA DIFERENTES TIPOS
    - a = “0” == 0; -- False
    - a = 2 < 15; -- True
    - a = “2” < “15”; -- False
    - a = 2 < ‘t’; NÃO OK!

Sintaxe - Operadores Lógicos

- and, or e not
- false e nil = false

- AND
  - Retorna o primeiro argumento se falso, ou o segundo se for verdadeiro.
  - Exemplo:
    - print (4 and 5);
    - print (nil and 13);

PET

Sintaxe

- OR
  - Retorna o primeiro argumento se verdadeiro, e o segundo se falso;
  - Exemplo:
    - print (4 or 5);
    - print (false or 5);
  - Equivalências
    - x = x or v;
      - if not x then x = v end;

PET

Sintaxe - Operadores Lógicos

- Precedência
  - and > or

- Retornar o maior número entre x e y.
  - max = (x > y) and x or y;

Sintaxe - Operadores Lógicos

- NOT
  - Sempre retorna true ou falso.
  - Exemplos:
    - print (not false);
    - print (not 0); -- retorna false

PET

Sintaxe

- Precedência
  - ^
  - not, -
  - *, /
  - +, -
  - ..
  - <, >, <=, >=, ~=, ==
  - and
  - or

- Exemplos
  - a+i < b/2+1
  - a < y and y <= z
  - -x^2

Sintaxe

- Concatenação
  - <tipo>..<tipo>
  - Exemplo:
    - print (“Hello “ .. ”World”)
    - print (0 .. 1 .. 2)
    - print (0 .. 1 .. ola) -- ERRO!

Sintaxe

- Construtores de Table
  - Expressões que criam e inicializam tables
  - emptyTable = {}
  - days = {Segunda,Terça,Quarta,Quinta,Sexta}
    - days[1] == Segunda
    - days[2] == Terça
    - days[3] == Quarta
    - ...
  - a = {x = 0, y = 0}
    - a == {}; a.x = 0; a.y = 0;

PET

Sintaxe

- Observações
  - a = {72, Nome = 'Rodrigo', Idade = 20, Altura = 1.72};
  - print (a[1]);
  - print (a.Nome);

  - b = {Nome = “Renan”, Idade = 2};
  - print (a[1]);
  - print (a.Idade);

PET

Sintaxe

- Comandos - Atribuição
  - <listaVariáveis> = <listaExpressões>
    - Valores separados por ‘,’
  - Exemplo:
    - var = 2;
    - a, b, c = 1, 2 ,3;
    - a, b = 1, 2, 3, 4; ou a, b, c, d = 1, 2;
    - a, b = 2, “Hello”;
    - a, b = b, a;
    - a, b = b, a;
    - a,b = f();

Sintaxe

- Variáveis Locais e Blocos
  - local variavel = 2;
  - Limitadas ao bloco onde foram declaradas

- Blocos
  - Corpo de uma estrutura de controle
  - Corpo de uma função
  - Ao arquivo onde o código dessa variável está
  - Inicia com “do” e termina com “end”

PET

Sintaxe

Sintaxe - Estruturas de Controle

- Condicionais
  - if, then, else, elseif
- Iterativo
  - while, repeat, for

- until -> termina a função repeat
- end -> termina as demais funções

PET

Sintaxe

- IF, THEN, ELSE, ELSEIF
  - Exemplos:
  - if (x> 0)

    then return “positivo”

    else return “negativo

    end
  - if op == ‘+’ then

    r = a + b;

    elseif op == ‘-’ then … end

PET

Sintaxe

- WHILE
  - while <expressãoCondicional> do

        <comandos>

    end

- Repeat
  - repeat

        <comando>

    until <condição>

Sintaxe

- FOR
  - Pode ser numérico ou genérico.
  - NUMÉRICO
    - for var = e1, e2, e3 do

      <comandos>

      end
      - e1 -> Início, e2 -> Final, e3 -> incremento
      - Se e3 não for especificada - e3 = 1;
      - A variável “var” não existirá depois do for
      - comando ‘break’, comando ‘return’

PET

Sintaxe

- FOR - OBSERVAÇÃO
  - Para usar uma variável encontrada dentro de um for precisamos salva numa outra variável, veja o exemplo:

```
for i = 1,5 do
    print (i)
end
max = i;
```

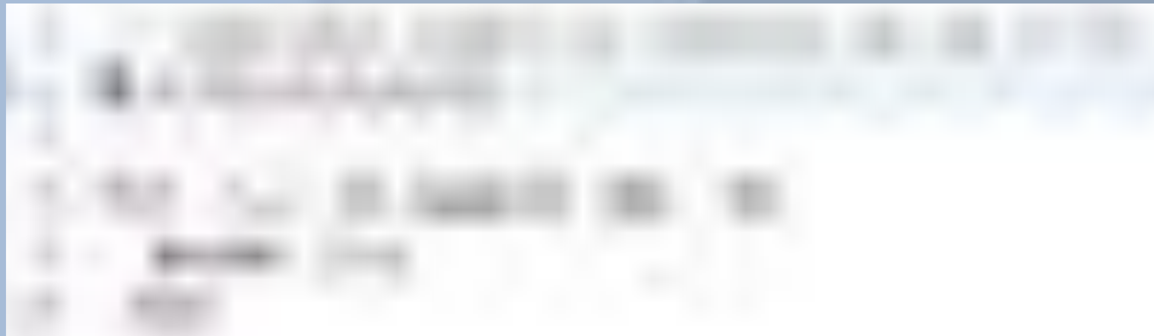
ERRADO!!

```
local found = nil;
a = {1,2,3,4,5}
for i = 1, 5 do
    if a[i] == 3 then
        found = i;
    end
end
```

CORRETO!!

Sintaxe

- FOR - Genérico
    - andar sobre todos os valores retornados por uma função iterativa
    - Exemplo:



Tipos

- Pequeno número de estruturas primárias
  - Dados atômicos, valores booleanos, números e strings
  - Estrutura genérica table que pode ser extendida para diversas outras estruturas mais específicas (vetores, matrizes, listas e até mesmo classes - *metatables*)
  - Não é **nativamente** orientada a objeto
    - Maioria das características pode ser implementada - *metamecanismos*.
  - Todas as variáveis são globais a não ser que especificado o contrário (local).

PET

Tipagem

- Dinamicamente e fortemente tipada
  - Valores possuem tipos, variáveis não.
  - Internamente variáveis possuem uma tag *type* que pode ser consultada
- Checagem de tipos em tempo de execução.
- Não possui especificação de tipos. Todos são deixados não especificados.
- Uso de *assertion* aliado à biblioteca de checagem para certificação do tipo.
- Entidades são cidadãos de primeira classe.

PET

Polimorfismo

- Lua só possui uma estrutura de dados: Tables.
  - Arrays associativas de primeira classe, criadas dinamicamente
  - Cada table define seu próprio comportamento
    - Necessidade de compartilhar métodos entre Tables da mesma "classe" (?)
  - Closures:
    - Possível, mas custoso, uma vez que cada objeto necessitaria sua própria closure para cada método.
  - Passagem do objeto chamador como parâmetro
    - obj.fun(self) *==* obj:fun()

PET

Polimorfismo

- Em qualquer caso, obj deverá conter um campo fun que referencie a função.
  - Necessário implementar herança!
  - Metatables:
    - Tabelas especiais que definem um comportamento comum a todas as que se configurarem como instâncias (classe).
    - Tabelas, por padrão, não são associadas a nenhuma metatable.
    - Possibilidade de implementar herança!

PET

Polimorfismo

- Ao fazer obj:fun(), o campo fun será buscado em obj, e então na metatable e sua hierarquia.
  - Possível definir uma função como pai de uma metatable (ou table), possibilitando então herança múltipla.
- Sobrecarga tradicional não está disponível!
- Variáveis sofrem coerção quando possível ou retorna erro:
  - “2,2” + 2 é uma operação válida e retornará 4,2
  - “2,2” + “2” também.
  - “apples” + “oranges” não é.

Funções

- Lua se adapta a múltiplos paradigmas:

```
(function (a,b) print(a+b) end)(10, 20)
```

  - Cria uma função anônima que imprime a soma de dois parâmetros e aplicá-a sobre 10 e 20.
  - Todas as funções em Lua são valores dinâmicos anônimos, criados em tempo de execução
  - Sintaxe convencional de funções:

```
function fact (n)
    if n <= 1 then return 1
    else return n * fact(n - 1)
    end
end
```

PET

Funções

- Simples "syntatic sugar" de:

```
fact = function (n)
        if (n < 2) then return 1
        else return n * fact (n-1)
        end
    end
```

- Funções são cidadãos de primeira classe!

- Comportamentos estranhos:

```
imprime = print             -- Imprime agora referencia a função seno
imprime("Hello World")
print = math.sin            -- Print agora referencia a função seno
imprime(print(1))
```

```
Hello World
0.8414709848079
```

PET

Funções

- Funções podem ser declaradas como locais (só serão acessíveis dentro do escopo)
- Suporte a número variável de argumentos:

```
function select(n, ...)
    return arg[n]
end
```

- Suporte a closures.
  - Função aninhada dentro de outra função compartilhando o mesmo escopo (escopo léxico).

```
function newCounter ()
    local i = 0
    return function ()
        i = i + 1
        return i
    end
end
```

PET

Metamétodos

- Funções especiais que definem o comportamento da tabela:

Tabelas			
__index	__newindex	__mode	__call
__metatable	__tostring	__len	__gc
Matemáticos			
__unm	__add	__sub	__mul
__div	__mod	__pow	__concat
Equivalência			
__eq	__lt	__le	

Confiabilidade

- Não é uma ênfase da linguagem:
  - Usualmente a aplicação faz uma chamada ao script em Lua para executar um chunk, que lança um erro a ser tratado na aplicação.
  - Aproxima-se da abordagem do C++.
  - Possível o lançamento de exceções (erros) nas funções, mas não torna obrigatório o tratamento por quem chama.

PET

Exceções

- Bloco Try/Catch pode ser aproximado utilizando *error()* e *pcall()*
  - Error() lança uma mensagem de erro (que pode ser qualquer entidade) de volta a quem chama (throw).
  - pcall() faz uma chamada segura à função. Retornando **true** caso não haja erro e os demais valores de retorno da função, ou **false** e o erro retornado. (catch)

```
retorno, erro = pcall(function () error({code=121}) end)
print(retorno, erro.code)
```

```
false    121
```

PET

Paralelismo

- Lua oferece a ferramenta co-rotinas para trabalhar com diferentes fluxos de execução.
- As co-rotinas apenas suspendem a sua execução quando é feita a chamada coroutine.yield()
- Uma co-rotina em Lua representa um fluxo de execução independente.
- O funcionamento é baseado nas três seguintes chamadas:

```
coroutine.create() -- Cria uma nova co-rotina
coroutine.yield() -- Faz a co-rotina ceder a sua execução
coroutine.resume() -- Reinicia a co-rotina do ponto onde havia parado
```

PET

Paralelismo

Exemplo:

```
count = 1
function foo (a)
    print("foo", a, count)
    count = count + 1
    return coroutine.yield(2*a)
end

co = coroutine.create(function
(a,b)
    print("co-body 1", a, b) --
    local r = foo(a+1) --
    print("co-body 2", r)
    local r, s = coroutine.yield
(a+b, a-b)
    print("co-body 3", r, s)
    return b, "end"
end)
```

```
print("main 1", coroutine.resume
(co, 1, 10)) print("main 2",
coroutine.resume(co, "r"))
print("main 3", coroutine.resume
(co, "x", "y"))
print("main 4", coroutine.resume
(co, "x", "y"))
```

```
co-body1    1    10
foo    2    1
main1    true    4
co-body2    r
main2    true    11    -9
co-body3    x    y
main3    true    10    end
main4    false    cannot resume dead
coroutine
```

Paralelismo

- Existem outras possibilidades, como o uso de Lua Process, que utiliza uma API C para Lua.
- É criado um tipo abstrato (Lua State) que controla a execução
- Um Lua State guarda informações do interpretador, variáveis globais e funções.
- Não há compartilhamento de memória.
- Qualquer comunicação interprocessos é feita por troca de mensagens.
- O uso desta biblioteca permite o uso de multiprocessamento, diferente das corotinas

PET

Paralelismo

Exemplo:

```
require "luaproc"
luaproc.createworker()
luaproc.newproc( [==[
    luaproc.newchannel( "achannel" )
    luaproc.newproc( [=[
         luaproc.send( "achannel", "hello
world" )
    ]=] )    luaproc.newproc( [=[
        msg = luaproc.receive( "achannel" )
        print( msg )
    ]=] )
]==] )
luaproc.exit()
```

PET

Eficiência

- Conhecida por ser uma linguagem bem rápida. Sendo bem avaliada em diversos benchmarks.
- Quanto ao uso de memória, o Lua utiliza o gerenciamento automático de memória, abstraindo do usuário os conceitos de alocação e desalocação.
- O coletor de lixo é executado de tempos em tempos e elimina os objetos mortos, utilizando o algoritmo de marca-e-limpa (*mark-and-sweep*) incremental.

PET

Eficiência

- O algoritmo do coletor de lixo:

1. Para todos os objetos no sistema, limpa o bit de marcação;
2. Passa por todos os ponteiros, começando das variáveis globais, pilha e registrados, passando em todos os objetos, marcando os objetos.
3. No terceiro passo, passa por toda a pilha linearmente de novo, eliminando todos os objetos não marcados.

PET

Eficiência

- Considerações do garbage collector:

```
local buff = ""
for line in io.lines() do
   buff = buff .. line .. "\n"
end
```

  - Leva quase um minuto para ler um arquivo de 350kb!
  - Outras linguagens com true garbage collection também sofre deste problema (Java -> StringBuffer)

Eficiência

- Outro ponto positivo é que incluir Lua numa aplicação não aumenta quase nada o seu tamanho. O pacote de Lua 5.2.3, contendo o código fonte e a documentação, ocupa 246K comprimido e 960K descompactado.

Eficiência

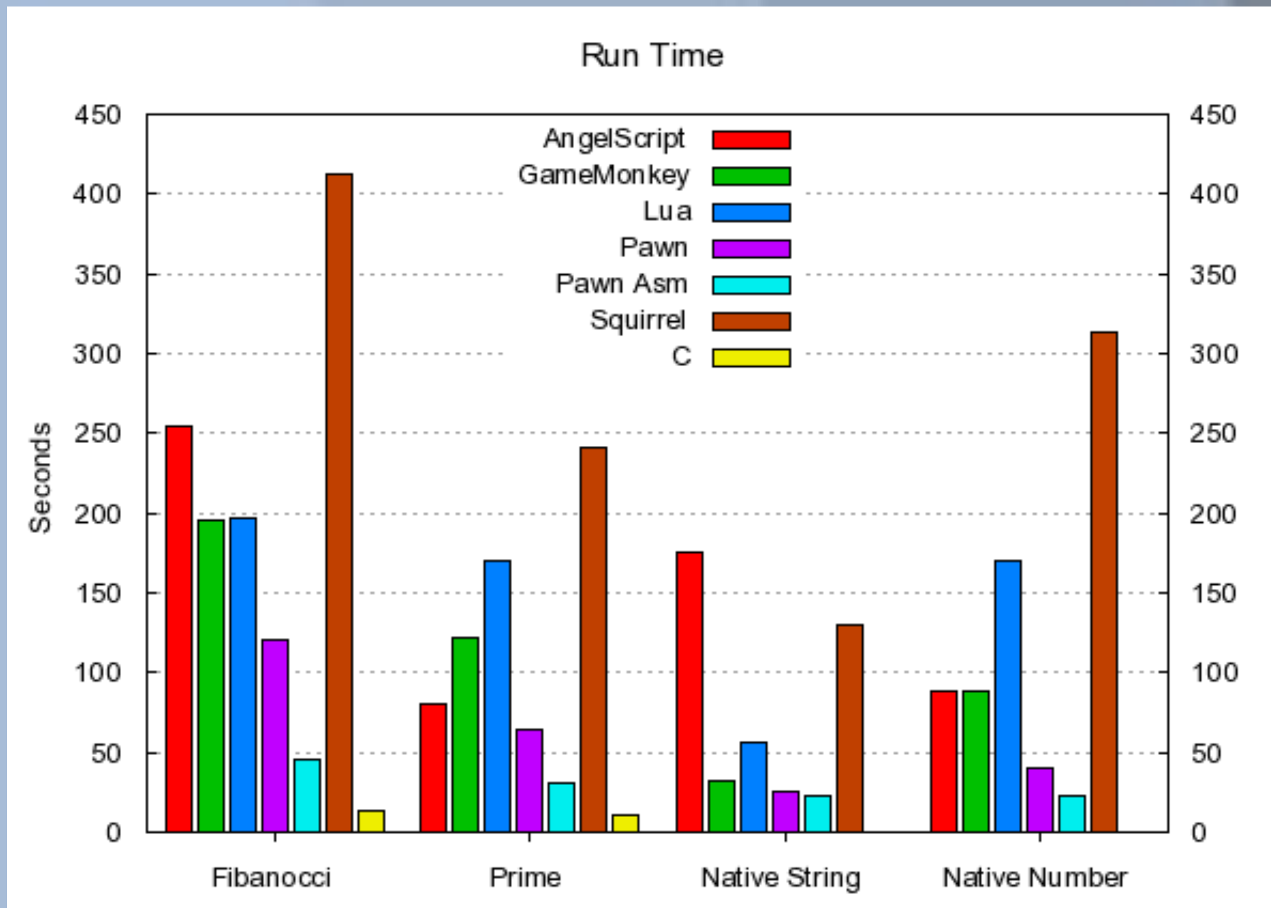
- Segue um benchmark onde são comparadas seis linguagens de programação muito utilizadas em jogos, ponto onde Lua é forte.

Language	Version
AngelScript	2.16.0
GameMonkey	1.25
Lua	5.1.4
Pawn	3.3.4058
Squirrel	2.2.2
TinyScheme	1.39

PET

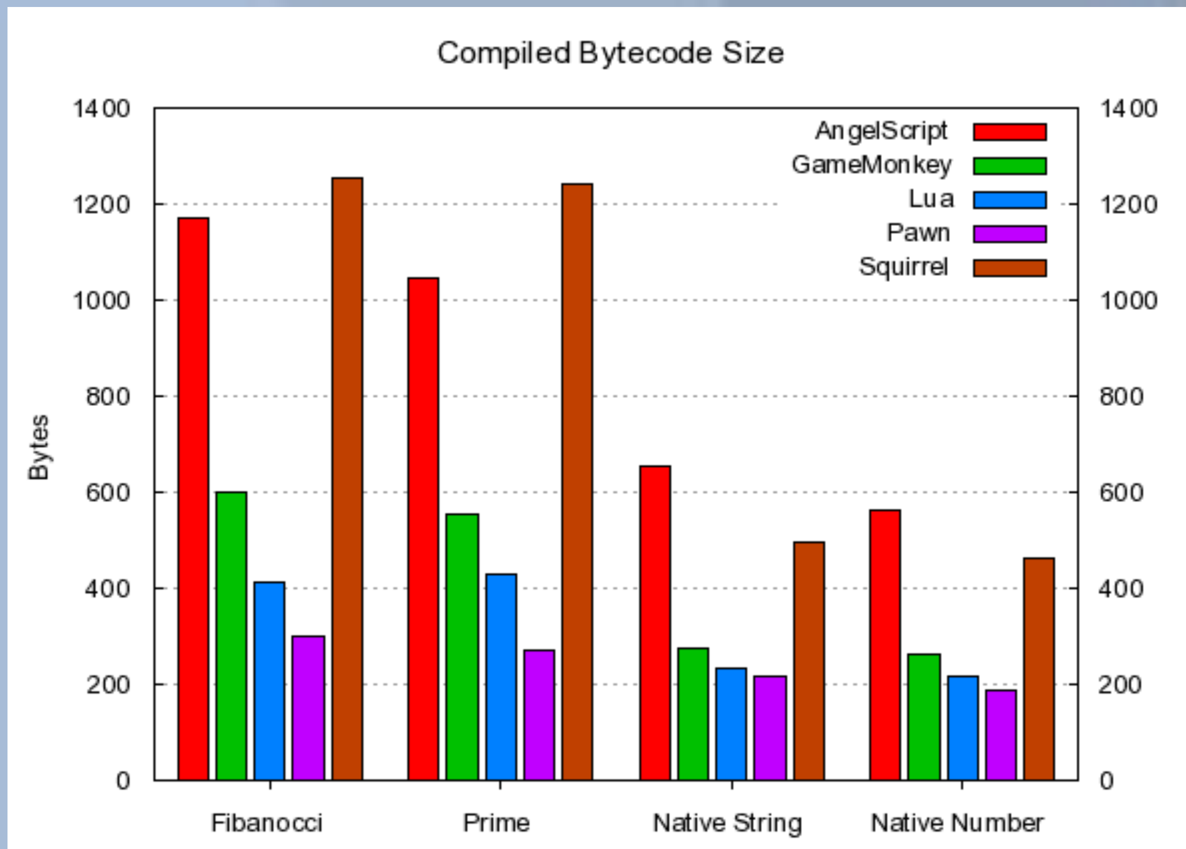
Eficiência

- Velocidade



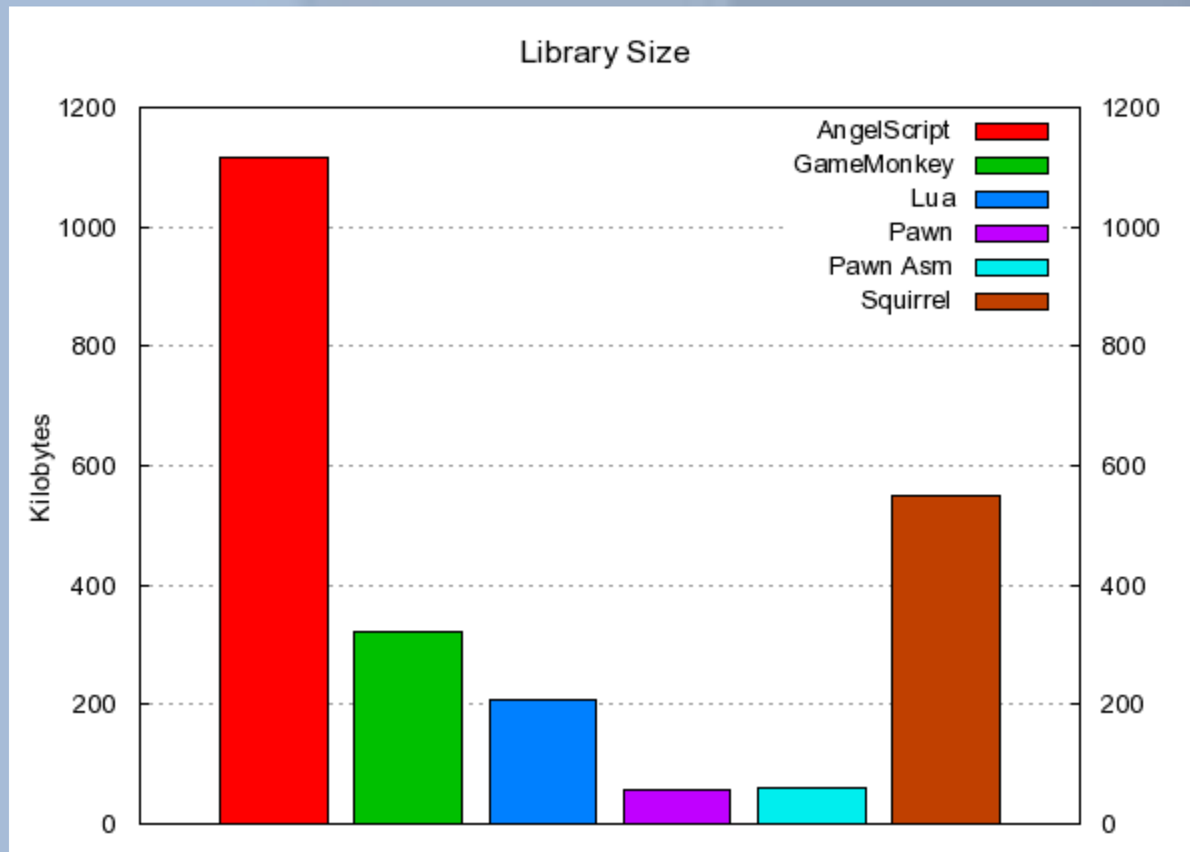
Eficiência

- Tamanho do Byte-Code



Eficiência

- Tamanho da Biblioteca



PET

Language	Time Seconds		Relative Time
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]
[illegible]	[illegible]		[illegible]

PET

Eficiência

- Pode-se utilizar também um compilador para Lua, luac.
- Ele permite uma interface bem mais eficiente com C do que o interpretador.
- Segundo a comunidade Lua, o desempenho se compilado chega a ser de 2-3x melhor .

PET

Portabilidade

- Lua é uma "freestading application":
  - Desenvolvida inteiramente sobre ANSI-C sem qualquer dependência de SO.
  - Núcleo completamente portável
    - Devido a sua pureza, já foi possível portar facilmente até mesmo para plataformas raramente consideradas (sistemas embarcados *eLua*)
  - Uso **cauteloso** de C89 produziria um programa altamente portável, cabeçalhos de Lua já tomam esse cuidado.

PET

Referências

https://github.com/davifrossard/LP20142_LUA/

http://www.inf.puc-rio.br/~roberto/docs/ry09-03.pdf

http://www.lua.org/docs.html

http://www.lua-users.org

http://c2.com/cgi/wiki?MarkAndSweep

http://codeplea.com/game-scripting-languages

http://lua-users.org/lists/lua-l/2013-01/msg00554.html

http://www.inf.puc-rio.br/~roberto/docs/ry08-05.pdf

PET

PET

UFES

EngComp