A Implementação de Lua 5.0

Roberto Ierusalimschy, PUC-Rio

O Diferencial de Lua

- Portabilidade
  - Unix, Windows, MS-DOS, Mac OS, BeeOS, OS/2, EPOC, Playstation II, etc.
- Pequeno Tamanho
  - distribuição completa cabe confortavelmente em um diskete
- Simplicidade: poucos (mas poderosos) mecanismos
  - e.g. tabelas
- Eficiência
  - apesar dos pontos anteriores

O Diferencial versus Versão 5.0

- Portabilidade
  - melhor suporte para tipos não convencionais
- Pequeno Tamanho
  - $\sim 10\%$ maior
- Simplicidade: poucos (mas poderosos) mecanismos
  - visibilidade léxica, co-rotinas
- Eficiência
  - $\sim 20\%$ mais rápida

Como Manter Eficiência?

- Pequeno tamanho restringe o uso de otimizações complexas
- Simplicidade restringe o uso de construções específicas
- Por outro lado, pequeno tamanho contribui para eficiência
  - cash, paginação, etc.
- Simplicidade permite nos concentrarmos em poucas otimizações
  - e.g. tabelas

Algumas Melhorias de Lua 4.0 para 5.0

- Visibilidade léxica
- Nova máquina virtual
- Otimização no uso de tabelas como arrays

VISIBILIDADE LÉXICA

- Funções aninhadas podem acessar qualquer variável externa, seguindo regras usuais de escopo

```
function f (x)
  local f1 = function (y) return x+y end
  return f1(10)
end
```

- Problema: variável pode ser acessada após término da função que a define

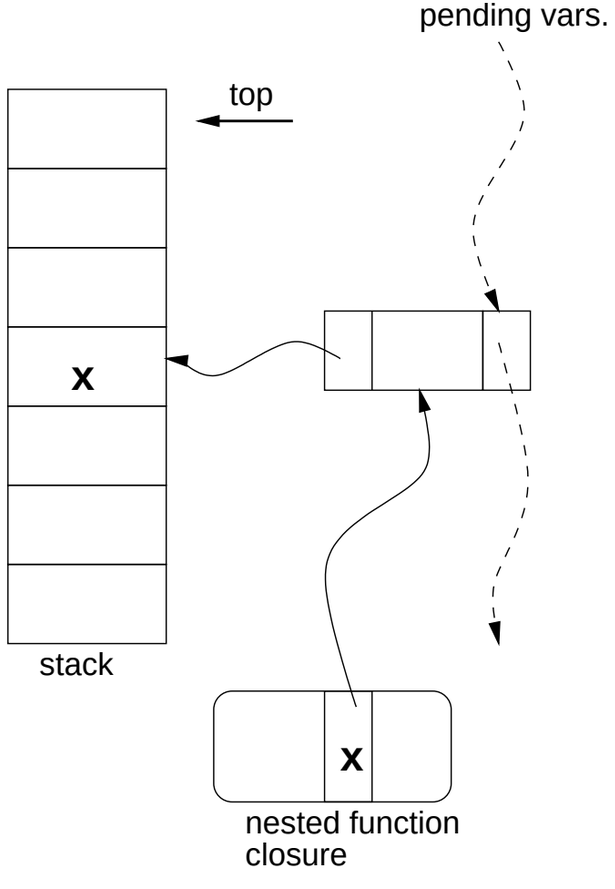
```
function f (x)
  return function (y) return x+y end
end
f1 = f(10); print(f1(20))
```

VISIBILIDADE LÉXICA (CONT.)

- Visibilidade Léxica dá enorme poder a uma linguagem . . .
  - variáveis locais a um módulo
  - base de programação funcional
  - opção para programação OO
- . . . mas implementação é complicada . . .
  - no exemplo anterior, x não pode morar em pilha/registrador
- . . . principalmente para um compilador de um passo
  - ao ver uma variável, compilador não sabe se ela será usada por alguma função aninhada

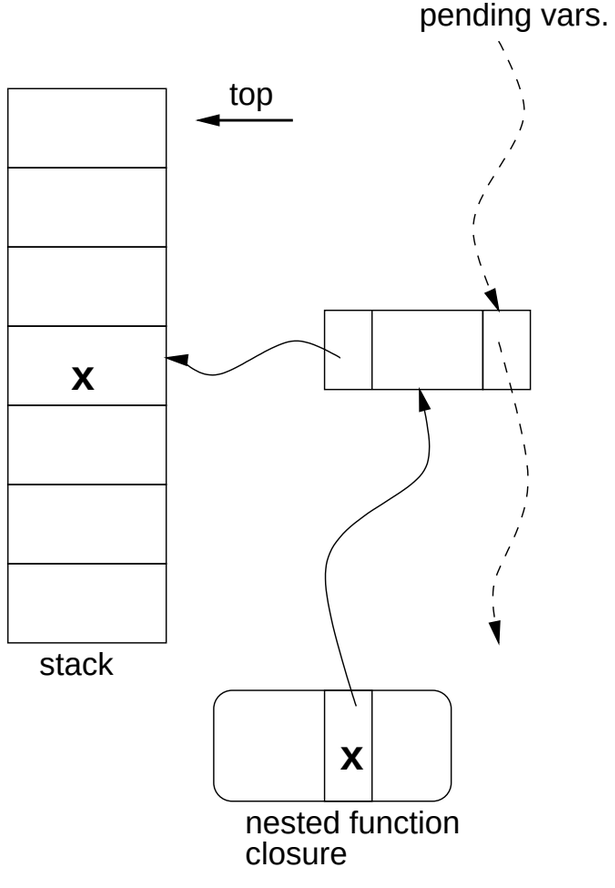
Implementação de Visibilidade Léxica

- Via indireção: função aninhada se refere a variável externa via um nó, que pode apontar para a pilha ou para o heap

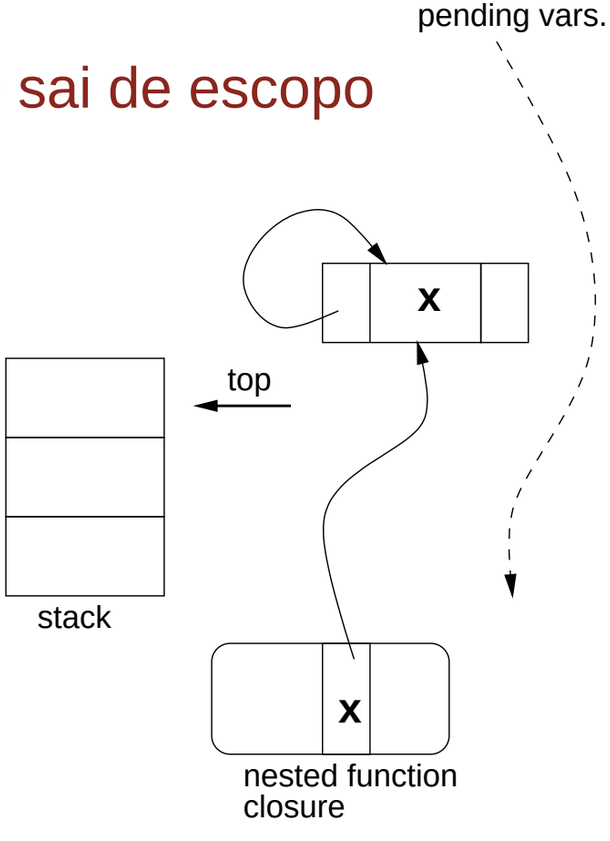


Implementação de Visibilidade Léxica

- Quando variável sai de escopo, seu valor é movido da pilha para o heap, e o ponteiro no nó é corrigido



variável x sai de escopo



MÁQUINAS DE PILHA

- Maioria das máquinas virtuais usam modelo de máquina de pilha
  - tradição iniciada com *p-code*, de Pascal
  - seguida por Java, Perl, etc.
- Exemplo Lua 4.0:

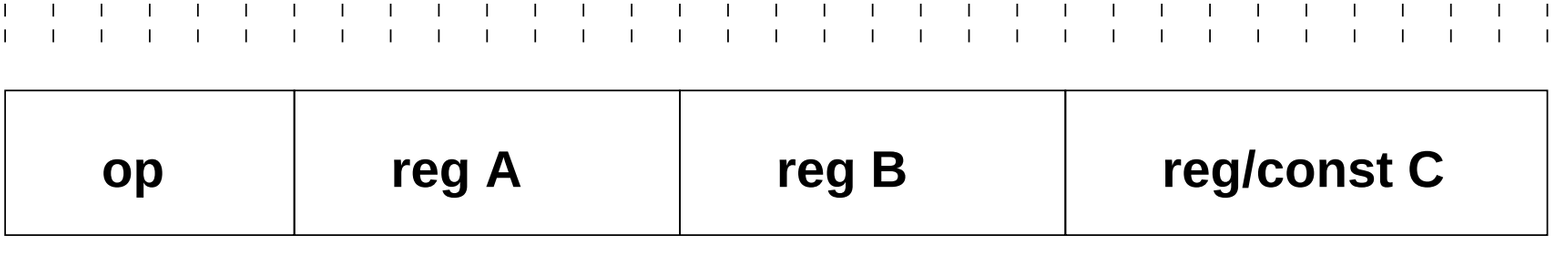
```
 while a<lim do a=a+1 end

3  GETLOCAL   0        ; a
4  GETLOCAL   1        ; lim
5  JMPGE      4        ; to 10
6  GETLOCAL   0        ; a
7  ADDI       1
8  SETLOCAL   0        ; a
9  JMP       -7        ; to 3
```

Outro Modelo para Máquina Virtual

- Máquinas de pilha têm instruções muito básicas
- Máquinas interpretadas têm alto custo extra por instrução
- Máquinas de registradores podem ter instruções mais poderosas
  - `ADD 0 0 [1] ; a=a+1`
- Custo de decodificar instruções mais complexas é compensado pelo menor número de instruções
- “registradores” são pré-alocados na pilha, no início de cada função.
  - número ilimitado de registradores facilita geração de código

Formatos das Instruções da Máquina Virtual

- O formato 1 é usado para todos os tipos de operadores
  - aritmética, comparações, indexação, etc.
- Permite até 256 “registradores”
- Terceiro operando ainda permite 768 constantes
  - Qualquer valor Lua, guardado em uma tabela de constantes (por função)



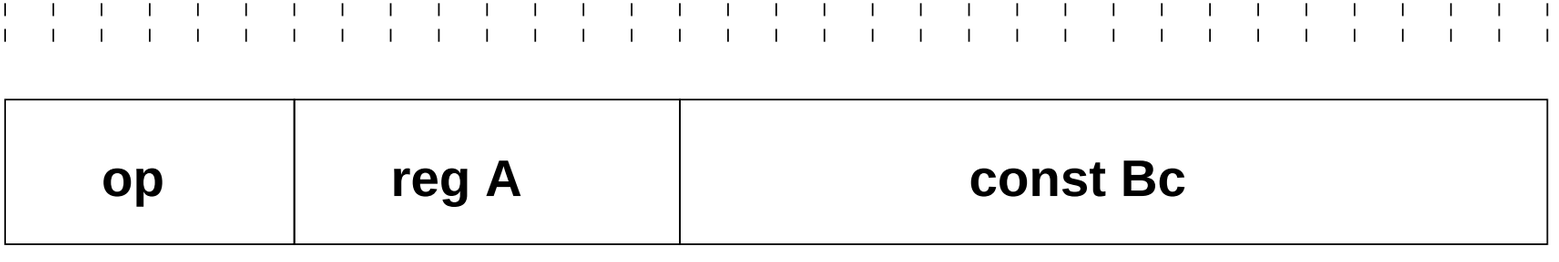
EXEMPLOS DE INSTRUÇÕES

```
MUL       0  0  [2]     ; a = 2*a

GETTABLE  2  1  ["x"]   ; t.x

SETTABLE  0  1  ["y"]   ; a = t.x

CMP       0 '<' [10]    ; a<10 ?
```

- assumindo que a está no registrador 0 e t no 1

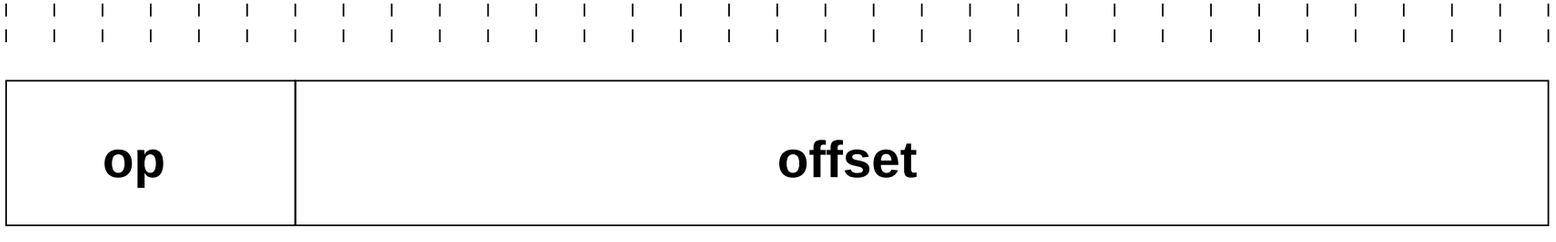
FORMATOS DAS INSTRUÇÕES DA MÁQUINA VIRTUAL

- O formato 2 é usado para acesso a globais e constantes que não cabem no formato 1
  - Isto é, em funções com mais de 768 constantes . . .



Formatos das Instruções da Máquina Virtual

- O formato 3 é usado para *jumps*
  - jumps são sempre relativos



EXEMPLO MAIS COMPLETO

```
while a<lim do a=a+1 end
```

```
-- Lua 4.0

3  GETLOCAL 0   ; a
4  GETLOCAL 1   ; lim
5  JMPGE    4   ; to 10
6  GETLOCAL 0   ; a
7  ADDI     1
8  SETLOCAL 0   ; a
9  JMP     -7   ; to 3
```

```
-- Lua 5.0

3  JMP   0  1        ; to 5
4  ADD   0  0  [1]   ; a = a+1
5  CMP   0  2  1     ; a < b?
6  JMP   0 -3        ; to 4
```

TABELAS EM LUA

- Único mecanismo de estruturação de dados da linguagem
  - simplicidade
- Arrays associativos com chaves genéricas
- Permite implementação fácil e eficiente de diversas estruturas de dados usuais
  - arrays, records, conjuntos, listas, etc.

TABELAS (CONT.)

- Arrays são simplesmente tabelas cujas chaves são inteiros

```
a = {}
for i=1,N do a[i] = 0 end
```

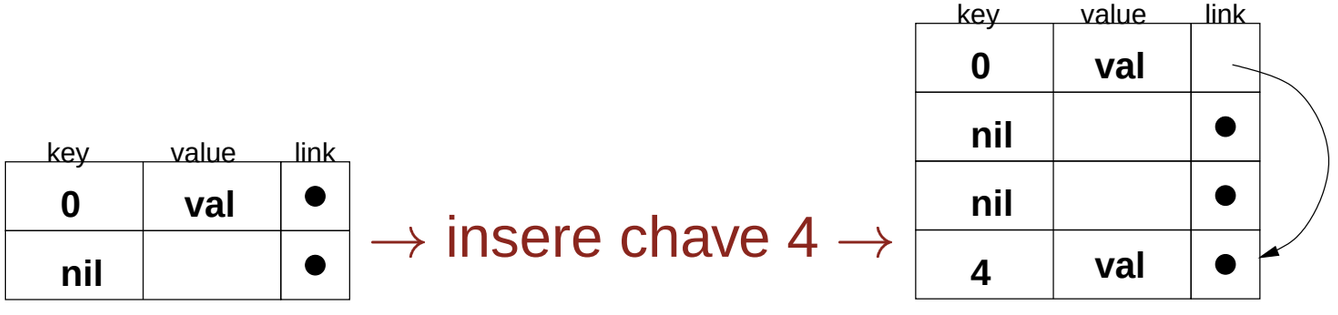
- Records são tabelas com os nomes dos campos como chaves
  - açucar sintático: `a.x` ≡ `a["x"]`

- Conjuntos representam seus elementos como chaves de uma tabela

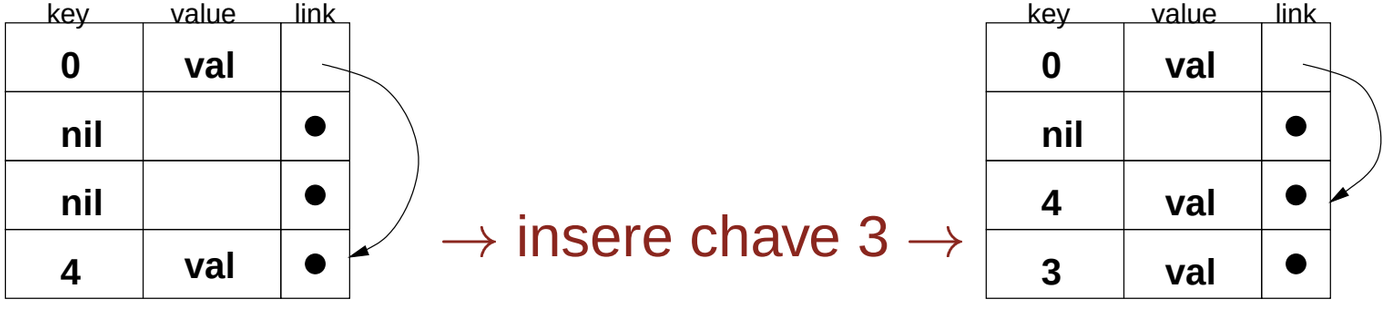
```
a[x] = 1                -- inserção
if a[x] then ...        -- pertinência
```

IMPLEMENTAÇÃO ORIGINAL DE TABELAS

- Algoritmo de hash, com listas internas para tratamento de colisão
- Executa *rehash* quando tabela fica cheia:



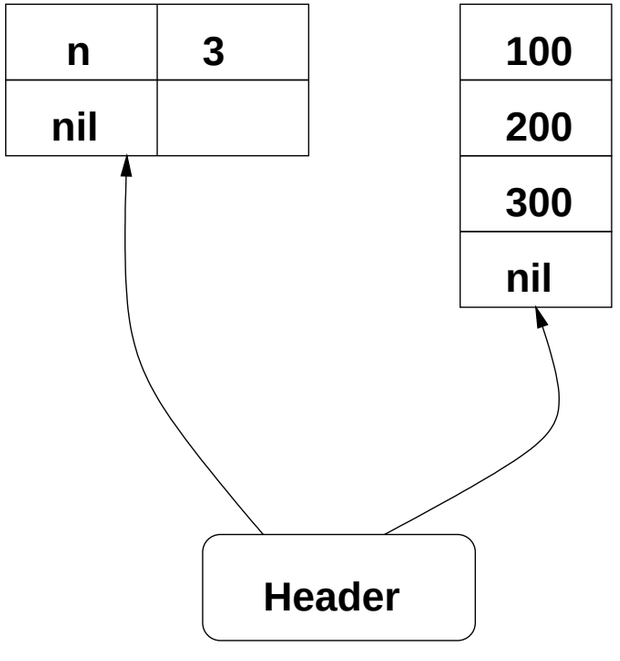
- Evita colisões secundárias, movendo elementos durante inserção:



Nova Estrutura de Tabelas

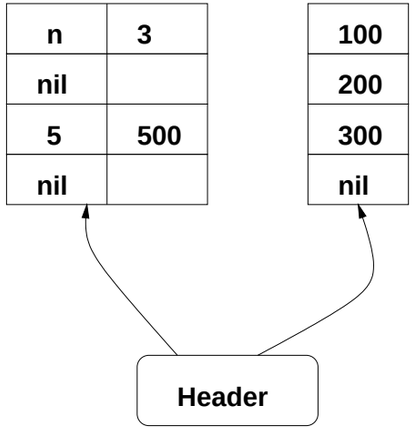
- Formada por duas partes, uma parte “hash” e uma parte “array”
- Exemplo:

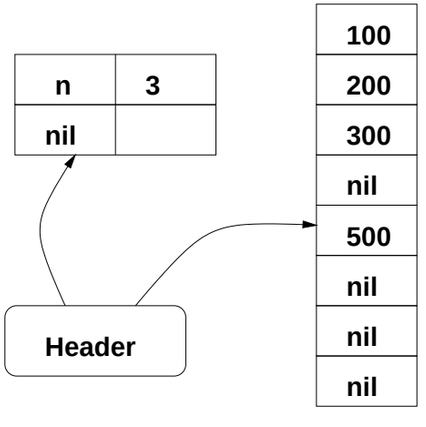
```
{n = 3; 100, 200, 300}
```



ESTRUTURA INTERNA DE TABELAS

- Problema: como distribuir os elementos entre as partes?
  - ou: qual o melhor tamanho para a parte array?
- Arrays esparsos podem desperdiçar muito espaço
  - uma tabela com um único elemento no índice 1000 não deve ter 1000 elementos
  - como a tabela anterior deve crescer ao adicionarmos o índice 5?





Cálculo de Tamanho de Tabelas

- Algoritmo executado somente durante o rehash
- Calcula tamanho N da parte array seguindo as regras abaixo:
  - N é uma potência de 2
  - a tabela contém pelo menos $N/2$ índices inteiros no intervalo $[1, N]$
  - a tabela contém pelo menos um índice inteiro no intervalo $[N/2 + 1, N]$
- Algoritmo é $O(n)$, onde n é o número de elementos na tabela

CÁLCULO DE TAMANHO DE TABELAS (CONT.)

- Idéia básica: criar um array onde a_i é o número de chaves inteiras no intervalo $(2^{i-1}, 2^i]$
  - array precisa ter apenas 32 elementos

- Tarefa fácil, dado um algoritmo eficiente para calcular $\lfloor \log_2 x \rfloor$
  - que é o índice do maior bit 1 de x

Cálculo de Tamanho de Tabelas (cont.)

- Depois, basta percorrer o array procurando o melhor tamanho:

```
total = 0
for i=0,32 do
  if a[i] > 0 then
    total += a[i]
    if total >= 2^(i-1) then
      bestsize = i
    end
  end
end
```

Resultados Parciais

- Visibilidade léxica: sem custo para quem não usa
- Nova máquina virtual: até 30% mais eficiente
- Nova estrutura de tabelas: economia de mais de 50% de memória na representação de arrays
  - sem prejuizo para representação de arrays esparsos