

Compiladores II

Fabio Mascarenhas - 2014.2

<http://www.dcc.ufrj.br/~fabiom/comp2>

Objetivo

- Durante esse curso iremos revisar as técnicas de implementação de linguagens de programação
- Veremos novas técnicas de análise sintática e de verificação de tipos, e também veremos geração de código intermediário para otimização de código e algumas otimizações comuns
- Por último, veremos a construção de máquinas virtuais como uma alternativa à construção de um compilador na implementação de uma linguagem de programação

Avaliação

- Presença será cobrada!
- Avaliação com dois componentes (mesmo peso):
 - Apresentação de 30 minutos sobre um artigo científico, a escolha do artigo é livre, mas o tema deve ser relacionado aos vistos em sala
 - Trabalhos práticos de implementação
- A avaliação é individual!

Estrutura Básica de um Compilador

- Cinco grandes *fases*

- Análise léxica

- Análise sintática

- Análise semântica

- Otimização

- Geração de Código

} AGOSTO

~ TIPOS (SETEMBRO)

→ SSA (OUTUBRO)

→ VMs e JITs (NOVEMBRO)

- As duas primeiras cuidam da *sintaxe* do programa, as duas intermediárias do seu *significado*, e a última da *tradução* para a linguagem destino

- As três primeiras fases formam o *front-end* do compilador, e as duas outras seu *back-end*

Lua

- Usaremos Lua como linguagem de implementação dos vários algoritmos vistos em aula (junto com C na parte do curso sobre máquinas virtuais)
- Lua é uma linguagem de script nos moldes de Python, Ruby e JavaScript, mas mais simples que as duas primeiras, e mais elegante que a última
- A página do curso tem um executável do interpretador Lua para Windows, e todas as distribuições Linux disponibilizam Lua nos seus gerenciadores de pacotes
- Existem IDEs para a linguagem, mas qualquer editor de texto também serve

Chunks e comandos

- A base de Lua é imperativa: um programa Lua é uma sequência de comandos chamada de *chunk*
- Qualquer sequência de comandos é um chunk: o corpo de um programa, o corpo de um laço, o corpo da definição de uma função...
- Os comandos em um chunk podem ser separados simplesmente por espaços, mas normalmente usamos quebras de linha ou ponto e vírgula:

```
> a = 1 | b = 2 | print(a, b)
1      2
```

```
> a = 1; b = 2; print(a, b)
1      2
```

REPL e dofile

- O interpretador Lua embute um *modo interativo*, ou REPL (Read-Eval-Print Loop), útil para experimentar
- Um inconveniente do REPL é que ele esquece tudo que foi definido quando saímos dele, por isso escrevemos as definições em um arquivo separado, e usamos o REPL apenas para interagir com elas, e definir valores temporários
- Usamos `dofile` para carregar as definições em um arquivo:

```
> dofile("defs.lua")
> print(fact(5))
120
```
- Tanto `dofile` quanto `print` são *funções pré-definidas*, e a sintaxe para chamar uma função é parecida com a de outras linguagens

Variáveis

- Como qualquer linguagem imperativa, Lua tem variáveis; em Lua, uma variável é *global* por padrão
- Não é preciso declarar variáveis globais, basta atribuir; seu escopo é qualquer chunk executado a partir do ponto em que ela foi definida, a não ser oculta por alguma variável local

```
> print(x)
nil
> x = 2
> print(x)
2
```

- Uma variável global que ainda não foi definida tem o valor *nil*

Detalhes léxicos

- Nomes de variáveis são como em outras linguagens, mas `_` (um único underscore) é um nome de variável válido, e nomes começando com `_` seguidos por letras maiúsculas são considerados “reservados”

`_VERSION_`

- As seguintes palavras chave são reservadas e não podem ser usadas como nomes de variáveis

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

- Comentários começam com `--` (dois hífen) e vão até o fim da linha

Valores

- Valores em Lua pertencem a um de oito tipos básicos:
 - nil (o valor nil), boolean (os valores true e false), number (números de ponto flutuante), string (vetores imutáveis de bytes), table (vetores associativos ou tabelas hash), function (funções), userdata (handles opacos para dados de bibliotecas externas) e thread (corotinas)
- Variáveis podem apontar para valores de qualquer tipo
- nil é a ausência de um valor: variáveis não inicializadas, campos inexistentes em uma tabela, parâmetros de uma função que não foram passados...

Booleanos

- Operações relacionais produzem booleanos, mas qualquer valor pode ser usado em uma expressão booleana: false e nil são falsos, e qualquer outro valor é verdadeiro

0 é verdadeiro "" é verdadeiro

- As operações lógicas and e or produzem um dos seus argumentos, e dão idiomas úteis:

```
function greeting(s)
  s = s or "Hello"
  print(s .. ", World!")
end
```

Parameter "optional"

```
greeting()
greeting("Olá")
```

```
function max(a, b)
  return (a > b) and a or b
end
```

"operator ternário"

Números

- Números são sempre de ponto flutuante, como em JavaScript
 - Divisão por 0 não é um erro!
- Além das quatro operações aritméticas usuais, Lua também tem \wedge (exponenciação) e $\%$ (módulo)
- Igualdade é `==`, como nas linguagens estilo C, mas “diferente de” é `~=`

Strings

- Uma string é uma sequência imutável de bytes, inclusive zeros, então podem representar qualquer dado binário
- Quando usadas com texto, normalmente usa-se a codificação UTF-8
- A operação de concatenação é `..`, e a operação `#` dá o tamanho de uma string, em bytes
- Strings não podem ser indexadas como vetores, mas funções dentro do namespace `string` podem retornar partes de uma string
- Strings simples podem ser delimitadas com aspas simples ou duplas; strings delimitadas com `[[e]]` podem se estender por várias linhas

Tabelas e funções

- Tabelas associam chaves a valores
- Qualquer valor Lua pode ser um chave, exceto `nil`, mas normalmente usamos números e strings como chaves
- Existe suporte da linguagem para usar tabelas como vetores, estruturas, tipos abstratos de dados, objetos, módulos...
- Funções são valores como quaisquer outros, e código Lua pode guardar funções em variáveis, passar funções como argumento para outra função, retornar funções, guardar funções em tabelas...

if-then-else, elseif

- Comandos condicionais são como em outras linguagens:

```
if a < 0 then
  print("a is negative")
  a = -a
else
  print("a is positive")
end
```

- Se quisermos testar várias condições em cascata usamos elseif:

```
if op == "+" then
  r = a + b
elseif op == "-" then
  r = a - b
elseif op == "*" then
  r = a * b
elseif op == "/" then
  r = a / b
else
  error("invalid operation")
end
```

while e repeat

- Lua tem um laço `while` como as linguagens estilo C:

```
i = 1; sum = 0
while i <= 5 do
    sum = sum + (2 * i - 1)
    i = i + 1
end
print(sum)
```

- E também um laço `repeat` estilo Pascal, que executa seu corpo até a condição ser verdadeira, garantindo que ele executa ao menos uma vez:

```
i = 1; sum = 0
repeat
    sum = sum + (2 * i - 1)
    i = i + 1
until i > 5
print(sum)
```

for numérico

- Lua tem dois laços for; o primeiro serve para percorrer sequências numéricas, similar ao for de Pascal, com um valor inicial, um final e um *passo*:

```
sum = 0
for i = 1, 10, 2 do
    sum = sum + (2 * i - 1)
end
print(sum)
```

Handwritten annotations in red:
- "inicial" above the 1 in the for loop.
- "final" above the 10 in the for loop.
- "passo" above the 2 in the for loop.

- A variável de controle é local ao corpo do laço, e não pode ser atribuída dentro do laço
- Expressões podem aparecer no valor inicial, final e no passo, mas só são avaliadas uma vez, antes do laço começar

Variáveis locais

- O comando local declara uma nova variável que é visível até o final do chunk corrente:

```
local sum = 0          -- local ao programa
for i = 1, 5 do
  local n = 2 * i - 1  -- local ao corpo do for
  sum = sum + n
end
print(sum, n)
```

→ nil

- Uma variável local oculta uma variável já existente com o mesmo nome, seja local ou global
- Vamos usar variáveis locais sempre que possível

do-end

- O bloco do introduz um novo escopo sem afetar o controle de fluxo:

```
sum = 0
do
  local i = 1
  while i <= 5 do
    sum = sum + (2 * i - 1)
    i = i + 1
  end
end
print(sum)
```

Atribuição múltipla

- Uma atribuição em Lua pode ter várias variáveis no lado esquerdo:

```
> a, b = 10, 2 * sum
> print(a, b)
10      50
```

- Todas as expressões do lado direito são avaliadas antes da atribuição começar, então podemos usar uma atribuição múltipla para trocar o valor de duas variáveis:

```
> a, b = b, a
> print(a, b)
50      10
```

- Um comando `local` também pode introduzir várias variáveis

Definindo funções locais

- Podemos definir novas funções locais facilmente:

```
local function max(a, b)
  return (a > b) and a or b
end
```

- Funções são valores, e não vivem em um espaço de nome separado, logo podem ser ocultadas por outras variáveis locais:

```
> print(print)
function: 0000000068B94B40
> do
>> local print = 10
>> print(print)
>> end
stdin:3: attempt to call local 'print' (a
number value)
stack traceback:
  stdin:3: in main chunk
  [C]: in ?
```

Funções “globais” e anônimas

- As funções globais que definimos até o momento não são realmente globais; o que o comando `function` faz é atribuir a função àquela variável:

```
local max
function max(a, b)
  return (a > b) and a or b
end
```

- O trecho acima é equivalente à definição do slide anterior
- Também podemos definir funções anônimas, omitindo o nome e usando `function` como uma expressão:

```
local max = function (a, b)
  return (a > b) and a or b
end
```

Múltiplos resultados

- Uma função pode retornar múltiplos valores com return
- Os valores extras podem ser usados em uma atribuição múltipla, contanto que a chamada seja a última expressão sendo atribuída:

```
> s, e = string.find("hello Lua users", "Lua")  
> print(s, e)  
7      9
```

- E também podem ser passados para outra função, contanto que a chamada seja o último argumento passado:

```
> print(string.find("hello Lua users", "Lua"))  
7      9
```

- Se a chamada não for a última expressão apenas o primeiro valor retornado é usado

Funções variádicas

- O ultimo parâmetro de uma função pode ser o token ...
- Dentro da função, ... produz todos os valores passados para a função a parte da posição do ...

```
> function id(...) return ... end
> print(id(1, 2, 3))
1      2      3
```

```
function printf(fmt, ...)
  io.write(string.format(fmt, ...))
end
```

```
> printf("%s(%d, %d)\n", "maxmin", 2, 3)
maxmin(2, 3)
```

Quiz

- Qual a saída do programa abaixo?

```
local function range(a, b, c)
  if a > b then
    return
  else
    return a, range(a + c, b, c)
  end
end
print(range(1, 9, 2))
```

todos os valores