

# Compiladores II

---

Fabio Mascarenhas - 2014.2

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

# SmallLua - sintaxe

---

```
bloco <- stat* (ret / '')
stat  <- "while" exp "do" bloco "end" / "local" "id" "=" exp /
      "id" "=" exp / "function" "id" "(" (ids / '') ")" bloco "end" /
      "if" exp "then" bloco ("else" bloco / '') "end" /
      pexp
ret   <- "return" exp
ids   <- "id" ("," "id")*
exps  <- exp ("," exp)*
exp   <- lexp ("or" lexp)*
lexp  <- rexp ("and" rexp)*
rexp  <- cexp (rop cexp)*
cexp  <- aexp ".." cexp / aexp
aexp  <- mexp (aop mexp)*
mexp  <- sexp (mop sexp)*
sexp  <- "-" sexp / "not" sexp / "false" / "true" / "number" /
      string "string" / lmb / pexp
lmb   <- "function" "(" (ids / '') ")" bloco "end"
pexp  <- "(" (" exp ")" / "id") "(" (" (exps / '') ")" )"*
rop   <- "<" / "==" / "~="
aop   <- "+" / "-"
mop   <- "*" / "/"
```

*boolean boolean number expressions*

# Tipagem estática

---

- Lua é uma linguagem *dinamicamente tipada*: o interpretador Lua sabe o tipo de cada valor, e verifica cada operação antes de ser executada, para checar se os tipos dos operandos estão corretos
- Em linguagens estaticamente tipadas, o *compilador* sabe o tipo de cada *termo* (expressão e variável) do programa, e pode verificar se os tipos dos operandos de cada operação estão corretos sem precisar executar o programa
- Um *sistema de tipos* é um sistema lógico que dá suporte à verificação de tipos em uma linguagem estaticamente tipada

# Um sistema de tipos simples para SmallLua

---

- Nossa linguagem SmallLua tem três tipos *atômicos*: number, boolean e string
- Funções são valores de primeira classe em SmallLua, então também temos *tipos compostos*: tipos da forma  $(t_1, \dots, t_n) \rightarrow tr$  representam funções de  $n$  parâmetros de tipos  $t_1$  a  $t_n$ , e tvalor de retorno de tipo  $tr$ .  
*(number, string) → string*     *(string, (string) → number) → string*
- Funções que não retornam nada têm tipo de retorno  $()$ , que chamamos de **unit** (é como o tipo **void** de C, só que aqui é um tipo de primeira classe)
- A PEG abaixo dá a sintaxe dos tipos de SmallLua:

```
type <- "number" / "string" / "boolean" / tfunc / "(" " ")"  
tfunc <- "(" types ")" "->" type / "(" " " ")" "->" type  
types <- type ("," type)*
```

# Anotações de tipos

---

- A forma mais simples de implementar tipagem estática é exigindo *anotações de tipos* explícitas
- Em SmallLua, podemos anotar apenas parâmetros e tipos de retorno de funções, já que as variáveis locais podem pegar seus tipos da sua expressão de inicialização

```
-- o mesmo para stat
lmb  <- "function" "(" (prms / '') ")" ":" type bloco "end"
prms <- "id" ":" type ("," "id" ":" type)*
```

# Regras de dedução

---

- As regras de dedução de um sistema de tipos dão um esquema de como podemos *deduzir* o tipo de uma expressão dados os tipos de suas subexpressões
- Tradicionalmente usamos uma notação “barra” para as regras de dedução, em que as hipóteses da regra ficam acima de uma barra horizontal e a conclusão abaixo dessa barra
- Tanto as hipóteses quanto a conclusão são escritas da forma  $\vdash e: t$ , onde  $e$  é uma expressão,  $t$  um tipo e o símbolo  $\vdash$  é a “roleta”

$\vdash n: \text{number}$  *axioma*

- Lê-se “pode-se provar que  $e$  tem tipo  $t$ ”

$\vdash s: \text{string}$

$$\frac{\vdash e_1: \text{number} \quad \vdash e_2: \text{number}}{\vdash e_1 + e_2: \text{number}}$$

# Tipagem de variáveis

---

- Qual o tipo de uma variável?
- Não podemos determinar esse tipo sintaticamente, ele depende do *contexto*
- Vamos dar esse contexto usando um ambiente de tipos que associa nomes de variáveis a seu tipo:

$$\frac{(x \rightarrow t) \in T}{\mathbb{T} \vdash x : t}$$

- Declarações de variáveis *estendem* o ambiente:

$$\frac{\mathbb{T} \vdash e : t}{\mathbb{T} \vdash \text{local } x = e : (), \mathbb{T}[x \rightarrow t]}$$

# Sequência de comandos

$T \vdash \text{cond}; \text{boolean} \quad T \vdash b : t, T'$   
 $T \vdash \text{while cond do b end} : (), T$

- Para tipar uma sequência de comandos, examinamos o primeiro comando da sequência: se o tipo dele for  $()$ , usamos o ambiente estendido para tipar o resto da sequência
- Se o tipo dele não for  $()$ , ignoramos o resto da sequência, e esse tipo é o tipo da sequência
- Isso permite uma maneira simples de tipar o efeito do comando return

$T_1 \vdash c : (), T_2 \quad T_2 \vdash cs : t, T_3$   
 $T_1 \vdash c cs : t, T_3$

$T_1 \vdash c : t, T_2$   
 $T_1 \vdash c cs : t, T_2$

$t \vdash e : t \quad t = T[\$ret]$   
 $T \vdash \text{return } e : t, T$



# Funções

---

- Para tipar a declaração de uma função, estendemos o ambiente associando o nome de cada parâmetro a seu tipo, depois tipamos o corpo da função, e checamos se o tipo bate com o tipo de retorno declarado
- Tipar a chamada de uma função é mais simples, bastando verificar que o número de argumentos bate com o de parâmetros (aridade), e o tipo de cada argumento bate com o de cada parâmetro

$$\frac{T[p_1 : t_1, \dots, p_n : t_n, \{ret\} \rightarrow t_n] \vdash b : t_1 T' \quad t = t_n}{T \vdash \text{function } (p_1 : t_1 \dots p_n : t_n) : t_n \quad b \text{ and } : (t_1 \dots t_n) \rightarrow t_n}$$

$$T \vdash f : (t_1 \dots t_n) \rightarrow t_n \quad T \vdash e_1 : t_1 \dots T \vdash e_n : t_n$$

$$T \vdash f(e_1 \dots e_n) : t_n$$