# 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 /</pre>
         "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)*</pre>
rexp <- cexp (rop cexp)*</pre>
cexp <- aexp ".." cexp / aexp
                                                             ex presson
aexp <- mexp (aop mexp)*</pre>
mexp <- sexp (mop sexp)*</pre>
sexp <- "-" sexp / "not" sexp / ("false")</pre>
                                                      "number"
   "string" | lmb / pexp
lmb <- "function" "(" (ids / '') ")" bloco "end"</pre>
pexp <- ("(" exp ")" / "id") ("(" (exps / '') ")")*</pre>
    <- "<" / "==" / "~="
rop
aop <- "+" / "-"
    <- "*" / "/"
mop
```

## Generalização

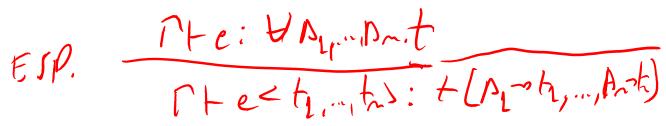
The:t

The local x= e: voi2, 
$$\Gamma[x - a]$$
 generalize  $(t, \Gamma)$ 

generalize  $(t, \Gamma) = \int t$ , params  $(t) - paons(\Gamma) = \emptyset$ 
 $\{A, B, \dots, t, panily - paons(\Gamma) \in \{P, B, \dots\}\}$ 

# Especialização





- O inverso da generalização é a especialização, que precisamos fazer toda vez que queremos usar um valor de um tipo polimórfico: no nosso caso, chamar uma função polimórfica, concatenar uma sequência com tipo polimórfico, ou passar uma função ou sequência com tipos polimórficos como argumento
- Na especialização, substituimos os parâmetros livres por tipos concretos
- O problema é: quais?
- Uma alternativa é incluir uma sintaxe para generalização:
- Podemos fazer melhor!

```
local e = seq()
local s = seq(1)
local n = byte<number>(s, 1)
local t = s .. e<number>
```

### Unificação

- Em uma chamada de função polimórfica, a ideia é usar os tipos dos argumentos para dizer quais devem ser os tipos que vamos usar na especialização
- Na concatenação, usamos o tipo do outro lado da concatenação
- Na passagem como argumento, usamos o tipo esperado
- Para fazer a atribuição de parâmetros a tipos usamos um algoritmo que mistura casamento de padrões com atribuição: a unificação

# Unificação - Variáveis

Va, b. (a, (c1-15) → shing)

(X, (X) → Y) n stry

- A unificação é um algoritmo fácil de definir quando trocamos nossos parâmetros de tipo por variáveis de tipo
- Uma variável inicialmente não está associada a nenhum tipo, mas pode vir a ficar durante o processo de unificação
- Uma vez associada, ela n\u00e3o pode mais mudar; efetivamente ela assume aquele tipo
- Duas (ou mais) variáveis não associadas podem se unir: quando uma for associada, a outra também fica associada ao mesmo tipo

(x, (x) - y) - stry (nucle) shy) stry

#### Unificação - pseudocódigo

```
function unify(t1, t2):
    t1 = prune(t1)
    t2 = prune(t2)
    case t1, t2:
        match var, :
            if occurs(t1, t2):
                error("ciclo")
            else:
                t1.type = t2
        match , var: unify(t2, t1)
        match base, base:
            if t1.tag ~= t2.tag:
                error("incompatible")
        match func(params1, ret1), func(params2, ret2):
            if #args1 ~= #args2:
                error("arity")
            zip(unify, params1, params2)
            unify(ret1, ret2)
        match seq(elem1), seq(elem2):
            unify(elem1, elem2)
        otherwise: error("incompatible")
```

#### Dojo

• Implemente a função unify e como uma função definida por casos (sem o teste occurs)