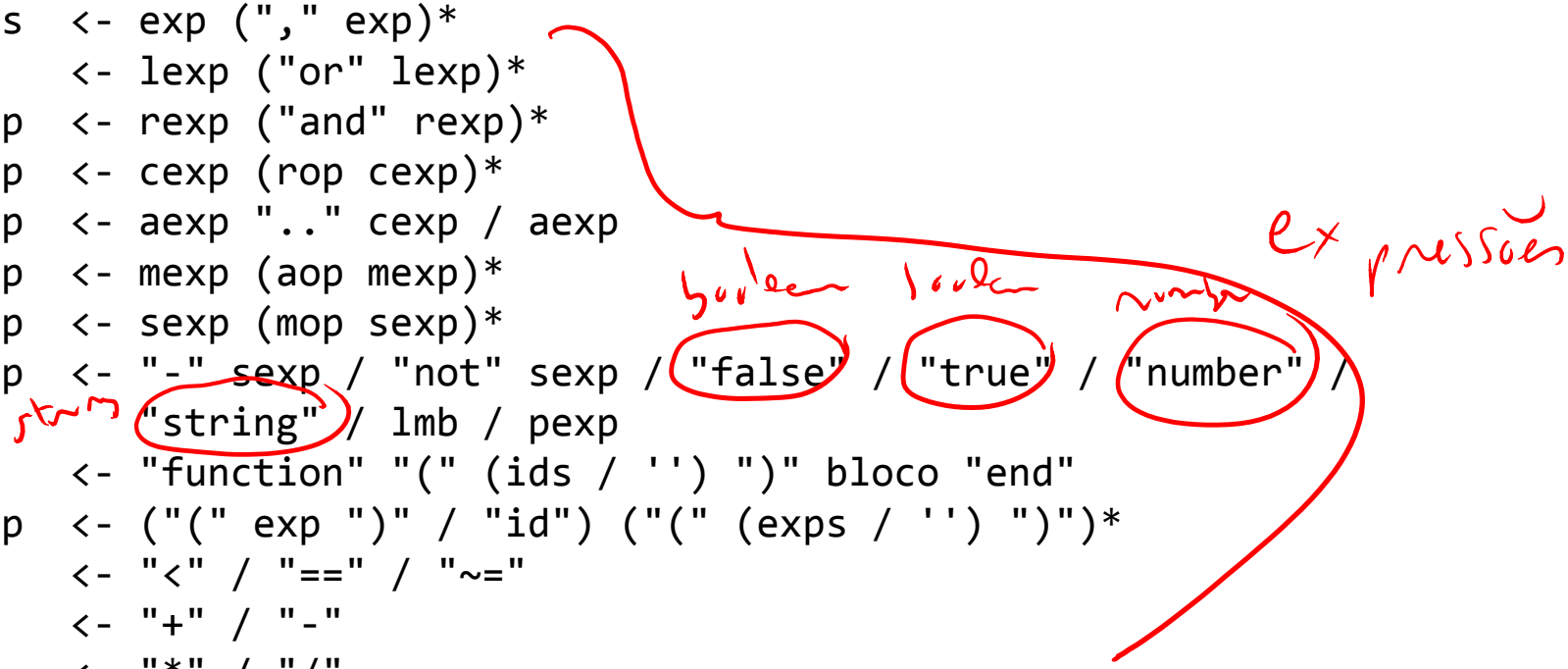
Compiladores II

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" / lmb / pexp
lmb   <- "function" "(" (ids / '') ")" bloco "end"
pexp  <- ("(" exp ")" / "id") ("(" (exps / '') ")")*
rop   <- "<" / "==" / "~="
aop   <- "+" / "-"
mop   <- "*" / "/"
```



Sequências

- Para o sistema de tipos de SmallLua ficar interessante, vamos acrescentar um tipo estruturado: *sequências*

- Um tipo `{ t }` é uma sequência de itens de tipo t

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

- Construímos sequências com os operadores `seq(...)`, que recebe os elementos da sequência como argumentos, e `..`, que concatena duas sequências

- Decompomos sequências com os operadores `byte(s, i)`, que retorna o i-ésimo elemento da sequência, e `sub(s, i, j)`, que retorna a subsequência entre os elementos i e j (inclusive)

Regras de dedução - sequências

$$\frac{\Gamma \vdash e_k : t}{\Gamma \vdash seq(e_1, \ldots, e_n) : \{t\}}$$

$$\frac{\Gamma \vdash e_1 : \{t\} \quad \Gamma \vdash e_2 : \{t\}}{\Gamma \vdash e_1 \cdot\cdot e_2 : \{t\}}$$

$$\frac{\Gamma \vdash e_1 : \{t\} \quad \Gamma \vdash e_2 : num \quad \Gamma \vdash e_3 : num}{\Gamma \vdash sub(e_1, e_2, e_3) : \{t\}}$$

$$\frac{\Gamma \vdash e_1 : \{t\} \quad \Gamma \vdash e_2 : number}{\Gamma \vdash byte(e_1, e_2) : t}$$

Operadores vs funções

- Classificamos `seq`, `byte` e `sub` como primitivas, ao invés de funções, por quê?
- Qual seria o tipo de `seq()` (uma sequência vazia)?
- Nosso sistema de tipos é *monomórfico*: cada termo do sistema pode ter apenas um tipo, e isso vale para as funções
- Logo, se `byte` e `sub` fossem funções só poderiam receber sequências de um tipo pré-determinado, e `seq()` sempre retornaria uma sequência do mesmo tipo
- Podemos resolver esses problemas com um sistema *polimórfico*

Parâmetros de tipos

- A ideia do *polimorfismo paramétrico* é poder ter *parâmetros* nos tipos, que são “buracos” onde podemos encaixar qualquer tipo

- Representamos esses parâmetros com *parâmetros de tipos*

- Um tipo polimórfico tem um ou mais parâmetros como parte dele (podemos usar o mesmo parâmetro mais de uma vez!)