Tópicos em LP

Fabio Mascarenhas – 2018.1

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

Exercício

- A gramática livre de contexto abaixo descreve a linguagem das PEGs:

```
grammar -> id '<-' exp grammar | id '<-' exp
exp -> term '/' exp | term
term -> pred term | pred
pred -> '!' pred | many
many -> simp '*' | simp
simp -> string | id | '[' id ']' | '(' exp ')'
```

- Construa um parser (juntando as partes léxica e sintática) para PEGs, que lê uma expressão como entrada e dá o parser correspondente, construído com os combinadores que já vimos

Erros

- Uma falha em um parser de combinadores ou PEGs tem dois significados:
  - A alternativa que estamos tentando não está correta, mas outra estará
  - A entrada tem um erro de sintaxe
- A união dessas duas condições em um único estado do parser causa problemas para gerar boas mensagens de erro para o usuário do parser!
- Um erro de sintaxe pode fazer o parser todo falhar sem indicar onde, ou fazer ele consumir apenas parte da entrada

Erros - exemplo

- Vamos ver o que acontece com a PEG a seguir:

```
bloco   <- stat*
stat    <- 'while' exp 'do' bloco 'end' / id '=' exp
exp     <- aexp '>' aexp / aexp
aexp    <- term (aop term)*
term    <- fac (mop fac)*
fac     <- number / id / '(' exp ')'
aop     <- '+' / '-'
mop     <- '*' / '/'
number  <- blanks [isdigit] [isdigit]*
id      <- blanks [isidstart] [isidrest]*
blanks  <- [isblank]*
```

- Vamos analisar o programa abaixo, que tem um erro de sintaxe:

```
n = 5
f = 1
while n > 0 do
    f = f * n
    n n - 1        -- falta um =
end
```

Falha mais distante

- Uma estratégia para indicar ao usuário onde um erro aconteceu é guardar a posição na entrada onde aconteceu a falha mais distante
  - Ou, de maneira equivalente, o tamanho do menor sufixo da entrada onde uma falha aconteceu
- Isso requer que o parser mantenha mais esse estado: ao invés de receber a entrada e dar o resultado, ele recebe a entrada e o menor sufixo, e retorna o resultado e o menor sufixo (que pode ser outro)
- Todos os combinadores que manipulam diretamente a entrada precisam ser mudados

Falha mais distante – terminais esperados

- Podemos melhorar ainda mais as mensagens de erro mantendo, além do menor sufixo, um conjunto de *termos esperados*
- A ideia é incluir um termo nesse conjunto toda vez que ele falhar com um sufixo de tamanho igual do do menor sufixo
- Na atualização do menor sufixo um novo conjunto é usado
- Ao final do parsing temos não apenas a posição onde o provável erro aconteceu, mas quais terminais (ou tokens) eram esperados naquela posição, ajudando o usuário a corrigir o erro
- Criamos um novo combinador para introduzir esses termos nomeados

SmallLua

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

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 quatro tipos *atômicos*: **number**, **boolean, string** e **nil**

- Funções são valores de primeira classe em SmallLua, então também temos *tipos compostos*: tipos da forma **(t1, ..., tn) -> tr** representam funções de *n* parâmetros de tipos **t1** a **tn**, e tvalor de retorno de tipo **tr**.

- Funções que não retornam nada têm tipo de retorno **nil**

- A PEG abaixo dá a sintaxe dos tipos de SmallLua:

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

```
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 subexpresões

- 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 $\vdash \boldsymbol{e}\boldsymbol{:}\ \boldsymbol{t}$, onde *e* é uma expressão, *t* um tipo e o símbolo $\vdash$ é a "roleta"

  - Lê-se "pode-se provar que *e* tem tipo *t*"

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

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

Tipagem de variáves

- 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:

$$\frac{(x \to t) \in T}{\mathrm{T} \vdash x : t}$$

- Declarações de variáveis *estendem* o ambiente:

$$\frac{\mathrm{T} \vdash e : t}{\mathrm{T} \vdash local\ x = e : T[x \to t]}$$

Escopo

Comandos e blocos

- Comandos não têm tipo, mas podem mudar o ambiente
- Em uma sequência de comandos, vamos “costurando” o ambiente por cada comando da sequência
- Para o comando `return`, o tipo de retorno fica em uma variável especial no ambiente
- Comandos que têm blocos (`if` e `while`) “jogam fora” o ambiente ao final de seus blocos, para manter as regras de escopo

$$\frac{T_1 \vdash s_1 : T_2 \quad T_2 \vdash s_2 : T_3}{T_1 \vdash s_1\ s_2 : T_3}$$

$$\frac{T \vdash e : t \quad T(\$) = t}{T \vdash \texttt{return}\ e : T}$$

Funções

- Para tipar a declaração de uma função, estendemos o ambiente associando o nome de cada parâmetro a seu tipo, assim como seu tipo de retorno

- 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