Compiladores II

Fabio Mascarenhas - 2014.2

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

Erros

- Uma falha em um parser de combinadores ou PEGs tem dois significados:
  - A alternativa que estamos tentando não está correta, mas outra estará
    ↳ entrada correta!
  - 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   <- "*" / "/"
```

- 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 - exemplo

- Vamos usar o mesmo exemplo de antes, e escrever uma função que faz o pós-processamento do menor sufixo e da entrada inicial para gerar uma mensagem de erro com linha e coluna

Falha mais distante – terminais esperados

- Podemos melhorar ainda mais as mensagens de erro mantendo, além do menor sufixo, um conjunto de *terminais esperados*
- A ideia é incluir um terminal 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

Combinadores “LL(1)”

- Uma outra maneira de detectar erros em parsers de combinadores e PEGs é introduzir um valor de “erro” explícito, além da falha
- Esse valor interrompe a análise, diretamente no local onde o erro aconteceu
- Mas quando um parser deve sinalizar um erro?
- Uma ideia é assumir que uma restrição análoga à restrição LL(1) de gramáticas livres de contexto: no `bind`, se o primeiro parser consumiu algo da entrada, uma falha no parser subsequente é transformada em um erro
- LL(1) é bastante restrito, então também acrescentamos um jeito de aumentar o “lookahead”: um combinador `try` que transforma um erro que tenha acontecido em uma falha

Outras extensões

- A ideia de erros em parsers determinísticos e PEGs pode ser estendida a vários tipos de erros, correspondendo a diferentes tipos de *exceções* em uma linguagem de programação

- Podemos permitir a parametrização dos não-terminais de PEGs, dando um poder de abstração similar ao dos combinadores

- A proibição de não haver recursão à esquerda também pode ser removida, permitindo escrever gramáticas de expressões sem o uso de chainr/chainl, e sem não-terminais diferentes para as várias classes de prioridade

SmallLua

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