Compiladores II

Fabio Mascarenhas - 2014.2

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

Combinadores de recursão

- Recursão em uma gramática é usada para *listas* e para *operações binárias*

- Podemos definir combinadores genéricos para esses dois tipos, e assim definir regras gramaticais sem precisar usar recursão explícita

```
local function listof(p, sep)
    return seq(p, poss(sep ^ function (_) return p end)) ^
                        function (tup)
                            local a, as = tup()
                            return unit(as:add(a, 1))
                        end
end
```

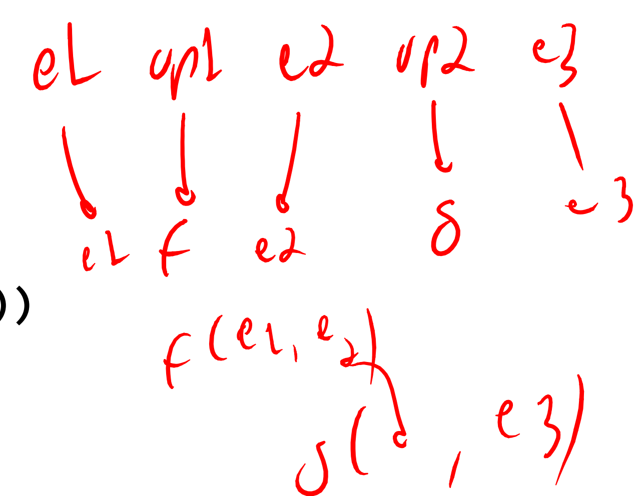
- O combinador `listof` reconhece uma lista de elementos com algum separador, jogando fora os resultados produzidos pelos separadores

ids = listof(token("id"), token(","))
exps = listof(exp, token(","))

Combinadores de recursão (2)

- Os combinadores chainl e chainr reconhecem expressões binárias associando à esquerda ou à direita

```
local function chainl(p, op)
    local function rest(e1)
        return seq(op, p) ^ function (tup)
                                local f, e2 = tup()
                                return rest(f(e1, e2))
                            end / unit(e1)
    end
    return p ^ rest
end
```



```
local function chainr(p, op)
    return p ^ function (e1)
                   return seq(op, chainr(p, op)) ^ function (tup)
                                                       local f, e2 = tup()
                                                       return unit(f(e1, e2))
                                                   end / unit(e1)
               end
end
```

Dojo

- Construa um analisador sintático para a gramática a seguir, usando os tokens gerados pelo analisador léxico:

```
exp  -> exp aop term | term
term -> term mop fac | fac
fac  -> number | id | '(' exp ')'
aop  -> '+' | '-'
mop  -> '*' | '/'
```

- Modifique o parser para fazer análise léxica em paralelo com a análise sintática (analisador sintático “scannerless”)

test-parser.lua

test_parser_sl.lua

recursão mútua!

parser.promise

Parsers determinísticos

- Se todos os nossos parsers primitivos produzem no máximo um resultado, a única maneira de um parser produzir mais de um resultado é usando o combinador `choice`

- Abrindo mão dele temos parsers que sempre produzem no máximo um resultado

- Assim podemos simplificar o tipo do nosso parser: ao invés de produzir uma lista de resultados, ele produz apenas um par `(resultado, resto)` ou `nil`, que sinaliza uma *falha*

parser.lua

PEGs

- As gramáticas de expressões de parsing, ou PEGs (parsing expression grammars) são uma linguagem para especificar parsers determinísticos

- Ao contrário das gramáticas livres de contexto, PEGs têm um mapeamento natural para os combinadores que estamos usando

- Uma *expressão de parsing* pode ser a expressão vazia `''`, um *terminal* `'a'`, um *não-terminal* `A`, uma *sequência* `pq`, onde `p` e `q` são expressões de parsing, uma *escolha ordenada* `p/q`, uma *repetição* `p*`, ou um *predicado* `!p`

- A únicas expressões que não correspondem a combinadores que já usamos são `!p`, que é fácil definir como um combinador, e os não-terminais, que vamos a seguir

Não-terminais e gramáticas

- Uma PEG é um mapeamento de não-terminais para expressões de parsing

- Quando aplicamos o parser de uma PEG a uma entrada, fica implícito que qualquer não-terminal encontrado é resolvido no contexto dessa PEG: o efeito de aplicar um não-terminal é o efeito de aplicar a expressão de parsing correspondente

- Não-terminais dão o poder de *recursão* às PEGs, mas com duas restrições:

  - Todo não-terminal referenciado tem que ser definido

  - Não pode haver *recursão à esquerda* direta ou indireta