# Tópicos em LP

Fabio Mascarenhas – 2018.1

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

# Sequência

- O combinador unit produz um parser que não consome nada, apenas gera um resultado:

```
local function unit(x)
    return function (input, pos)
                return { { x, pos } }
            end
end
```

- Podemos juntar unit e bind para fazer um combinador que aplica dois parsers em sequência, juntando cada combinação de resultados em um par

```
local function seq(p1, p2)
  return bind(p1, function (res1)
    return bind(p2, function (res2)
      return unit(seq:new{ res1, res2 })
    end)
  end)
end
```

# Escolha

- O combinador `choice` junta dois parsers em um que tenta ambos os parsers, combinando suas listas de resultado:

```
local function choice(p1, p2)
    return function (input, pos)
                local out = {}
                local lres1 = p1(input, pos)
                for _, par in ipairs(lres1) do
                    out[#out+1] = par
                end
                local lres2 = p2(input, pos)
                for _, par in ipairs(lres2) do
                    out[#out+1] = par
                end
                return out
            end
end
```

# Ambiguidade

- O combinador de escolha definido no slide anterior introduz *ambiguidade* em nossos parsers, já que é ele quem irá produzir listas com mais de um resultado possível

- Por exemplo, podemos expressar *repetição* usando escolha e recursão:

```
local function star(p)
  return choice(bind(p, function (res)
                          return bind(star(p), function (lres)
                                                 return unit(seq:new{ res } .. lres)
                                               end)
                        end), unit(seq:new{}))
end
```

# Escolha ordenada

- A repetição de many dá todas as possibilidades como resultado: o primeiro resultado dá o máximo de repetições possíveis, mas os seguintes dão todos os outros, até zero repetições, cada um com um sufixo diferente da entrada

- Geralmente queremos mais determinismo em um parser! Uma possibilidade para isso é usar a *escolha ordenada*:

```
local function ochoice(p1, p2)
    return function (input, pos)
                local res1 = p1(input, pos)
                if #res1 > 0 then
                    return res1
                else
                    return p2(input, pos)
                end
            end
end
```

# Repetição gulosa e possessiva

- Substituindo `choice` por `ochoice` em `many` temos uma repetição *gulosa* e *possessiva*

- Se fazemos uma sequência de uma repetição possessiva e outro parser a repetição possessiva pode fazer o parser seguinte falhar mesmo que um número menor de repetições fizesse ele ter sucesso

- Podemos ter uma repetição gulosa mas não possessiva fazendo a sequência da repetição ser o caso base dela, ao invés de `unit(seq.new{})`

- Uma terceira possiblidade de repetição é a *preguiçosa*, onde pegamos a repetição gulosa e invertemos a ordem da escolha, e aí teremos o número *mínimo* de repetições

# Escolha LL(1)

- Outro tipo de escolha útil é a escolha guiada por determinado predicado aplicado ao primeiro item da entrada:

```
local function pchoice(pred, p1, p2)
  return function (input, pos)
            local item = input:byte(pos)
            if pred(item) then
                return p1(input)
            else
                return p2(input)
            end
         end
end
```

# Recursão

- Uma regra gramatical não recursiva pode ser construída usando os combinadores que já temos, mas uma regra recursiva precisa ser expressa como um parser que constrói o seu lado direito de forma preguiçosa

- Podemos agrupar um conjunto de regras mutuamente recursivas em uma *gramática*, e ter um parser que recebe uma gramática e o nome de um não-terminal e, quando executado, consulta esse não-terminal na gramática

# 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 map(pseq(p, pstar(bind(sep, function (_) return p end))),
                       function (pair)
                           local a, as = pair:byte(1), pair:byte(2)
                           return seq:new{a} .. as
                       end)
end
```

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

# 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 ochoice(bind(pseq(op, p), function (pair)
                                            local f, e2 = pair:byte(1), pair:byte(2)
                                            return rest(f(e1, e2))
                                          end, unit(e1)))
    end
    return bind(p, rest)
end

local function chainr(p, op)
    return bind(p, function (e1)
                     return ochoice(bind(pseq(op, chainr(p, op)),
                                         function (pair)
                                           local f, e2 = pair:byte(1), pair:byte(2)
                                           return unit(f(e1, e2))
                                         end), unit(e1))
                   end)
end
```

# Sobrecarga de operadores

- Podemos deixar os combinadores mais fáceis de serem usados usando sobrecarga de operadores para os combinadores binários mais comuns: + ou | para escolha (um pode ser ordenada e o outro não), ^ para bind (flatmap), / para map

- Para isso os parsers precisam deixar de serem funções, e passam a ser objetos (tabelas com uma metatabela), podemos usar o metamétodo `__call` para manter a forma atual de usar um parser

# Exercício

- 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”)