Compiladores II

Fabio Mascarenhas - 2014.2

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

Parsers

- Um parser processa uma sequência de entrada, consumindo uma parte (ou toda a sequência) para produzir um resultado
- Um analisador léxico é um tipo de parser: a entrada é uma sequência de caracteres, e o analisador produz um token
- Um analisador sintático é outro tipo: a entrada é uma sequência de tokens, e o analisador produz uma árvore sintática (abstrata ou não)
- Em um analisador sintático recursivo, cada parte do analisador também é um parser, cada uma consumindo uma parte da entrada e produzindo um resultado parcial; o analisador completo é feito através da composição dessas partes

Combinadores de parsing

- Combinadores de parsing são uma técnica para expressar parsers recursivos em uma linguagem onde funções são valores de primeira classe

- A ideia é construir parsers mais complexos a partir da composição de parsers mais simples, mas usando *combinadores* ao invés da composição sintática de um analisador recursivo tradicional

- Um parser é uma função que recebe a entrada e retorna um resultado e um sufixo dessa entrada

- Um combinador é uma função que recebe uma ou mais funções que descrevem parsers e as combina em um novo parser

Lista de resultados

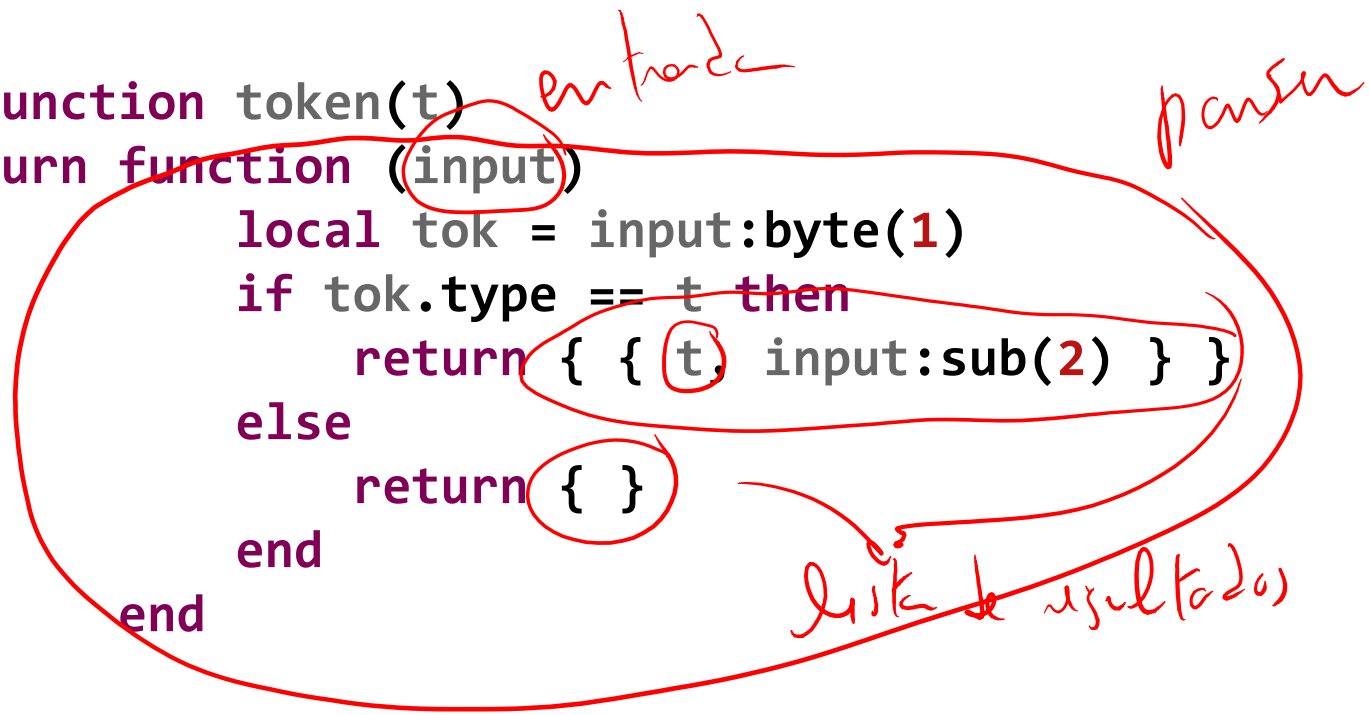
- Nem sempre um parser é bem sucedido
- Uma determinada entrada também pode ter mais de uma análise possível
- Para representar essas duas possibilidades definimos que um parser retorna uma *lista de resultados* ao invés de um resultado só
- Cada elemento dessa lista é um par com o resultado em si e um sufixo da entrada
- Se a lista for vazia, o parser falhou

Um combinador simples

- No domínio da análise sintática, os parser mais simples são aqueles que reconhecem um único token

- O combinador `token` retorna um desses parsers, dado o tipo do token desejado:

```
local function token(t)
    return function (input)
            local tok = input:byte(1)
            if tok.type == t then
                return { { t, input:sub(2) } }
            else
                return { }
            end
        end
end
```



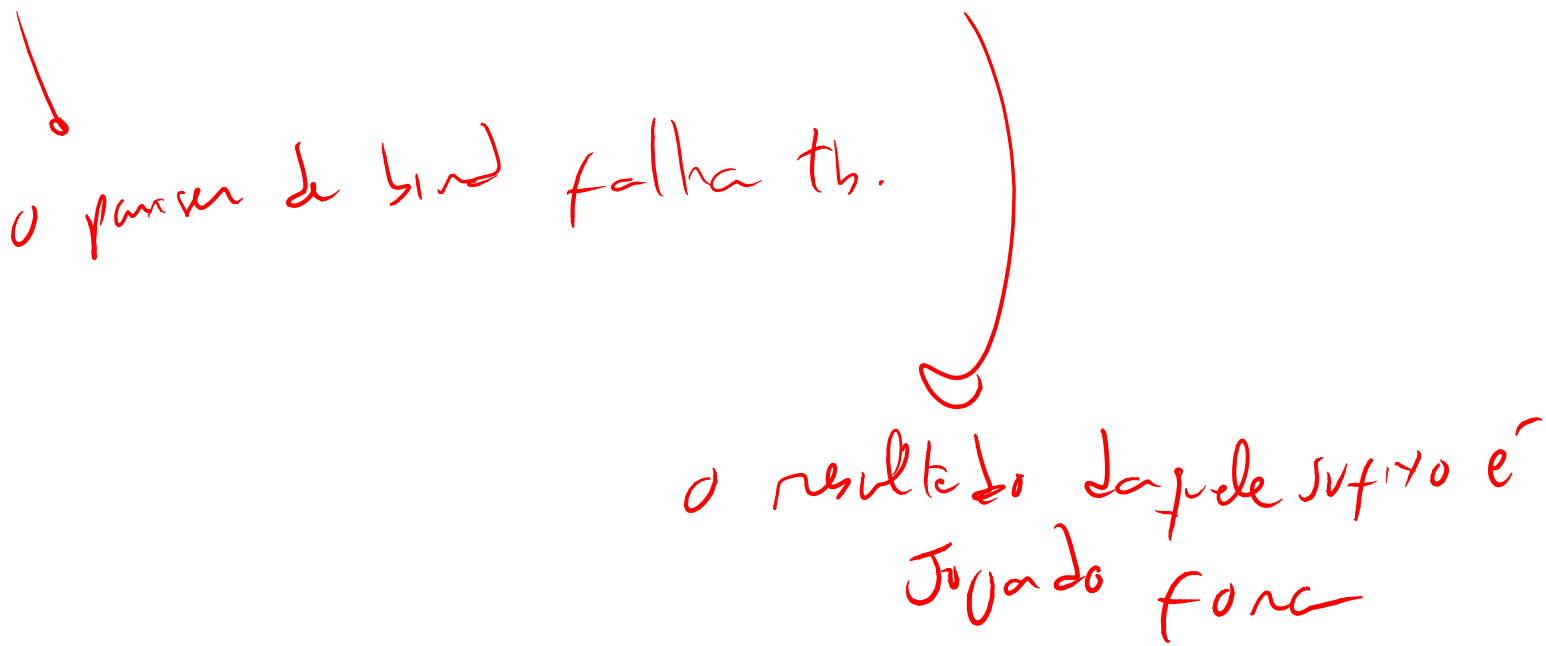
Bind

- O combinador `bind` junta um parser com uma função que recebe o resultado de um parser e retorna um novo parser:

```
local function bind(p, f)
    return function (input)
        local lres = p(input)
        local out = {}
        for _, par in ipairs(lres) do
            local lres = f(par[1])(par[2])
            for _, par in ipairs(res) do
                out[#out+1] = par
            end
        end
        return out
    end
end
```

Quiz

- O que acontece se o parser passado para `bind` falhar (retornar uma lista vazia de resultados)? E se o parser retornado por `f` falhar para algum sufixo da entrada?



Sequência

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

```
local function unit(x)
    return function (input)
               return { { x, input } }
           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({ res1, res2 })
                                    end)
                  end)
end
```

Dojo

- Construa um combinador `char` equivalente a `token`, mas para reconhecer um caractere em uma sequência de caracteres. Esse combinador recebe uma string contendo apenas um caractere → em parse.lua

- Construa uma versão aprimorada de seq que recebe uma quantidade arbitrária de parsers, retornando um parser que aplica todos eles em sequência, juntando os resultados em tuplas

- O que acontece se um parser em uma determinada sequência falha?

↳ fuga de falha

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)
            local out = {}
            local lres1 = p1(input)
            for _, par in ipairs(lres1) do
                out[#out+1] = par
            end
            local lres2 = p2(input)
            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 many(p)
  return choice(bind(p, function (res)
                          return bind(many(p), function (lres)
                                                 local out = { res }
                                                 for _, item in ipairs(lres) do
                                                   out[#out+1] = item
                                                 end
                                                 return out
                                               end)
                        end), unit({}))
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)
        local lres1 = p1(input)
        if #lres1 > 0 then
            return { lres1[1] }
        else
            local lres2 = p2(input)
            return { lres2[1] }
        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({})`

- 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