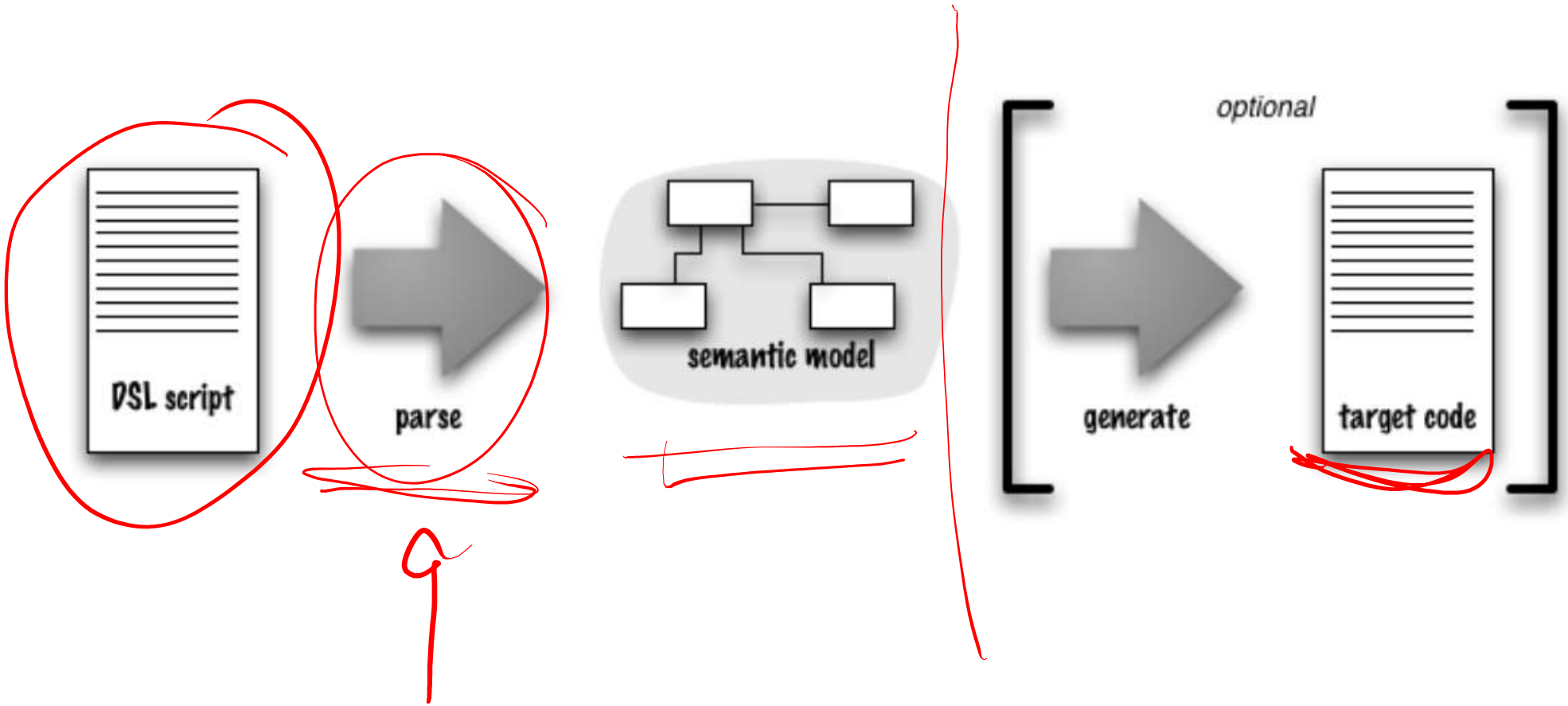
Linguagens de Domínio Específico

Fabio Mascarenhas – 2016.1

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

Processamento de uma DSL



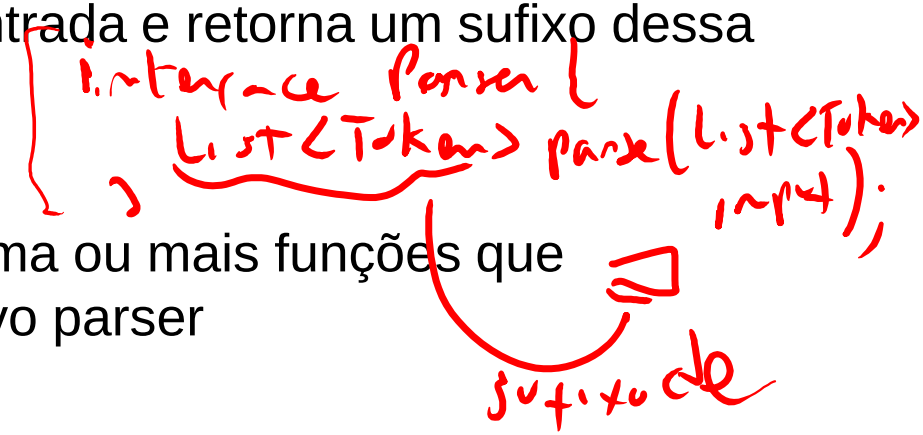
Combinadores de parsing

- Combinadores de parsing são uma técnica para expressar parsers recursivos em uma linguagem com funções anônimas ou objetos

- 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 uma entrada e retorna um sufixo dessa entrada caso reconheça uma parte dela

- 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 List<List<Token>>
- Cada elemento dessa lista é um sufixo da entrada original
- 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:

```
public class TokenParser implements Parser {
  public int tipo;
  public TokenParser(int _tipo) {
    tipo = _tipo;
  }
  public List<List<Token>> parse(List<Token> input) {
    Token tok = input.get(0);
    ArrayList<List<Token>> result = new ArrayList<List<Token>>();
    if(tok.tipo == tipo) {
      result.add(input.subList(1, list.size());
    }
    return result;
  }
}
```

Seq

- O combinador `seq` faz a sequência de dois parsers:

```
public class Seq implements Parser {
  public Parser p1;
  public Parser p2;
  public Seq(Parser _p1, Parser _p2) {
    p1 = _p1; p2 = _p2;
  }
  public List<List<Token>> parse(List<Token> input) {
    List<List<Token>> res1 = p1.parse(input);
    ArrayList<List<Token>> result = new ArrayList<List<Token>>();
    for(List<Token> suf: res1) {
      result.addAll(p2.parse(suf));
    }
    return result;
  }
}
```

Escolha

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

```
public class Choice implements Parser {
  public Parser p1;
  public Parser p2;
  public Choice(Parser _p1, Parser _p2) {
    p1 = _p1; p2 = _p2;
  }
  public List<List<Token>> parse(List<Token> input) {
    List<List<Token>> res1 = p1.parse(input);
    List<List<Token>> res2 = p2.parse(input);
    ArrayList<List<Token>> result = new ArrayList<List<Token>>();
    result.addAll(res1);
    result.addAll(res2);
    return result;
  }
}
```

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, sequência e recursão:

```
public class Many implements Parser {
  public Parser p;
  public Many(Parser _p) {
    p = new Choice(new Seq(_p, this), new Empty());
  }
  public List<List<Token>> parse(List<Token> input) {
    return p.parse(input);
  }
}
```

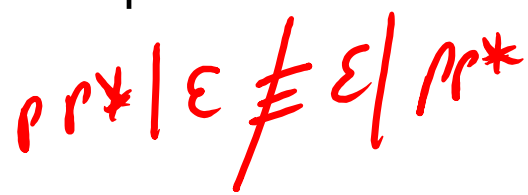
- `empty` é um parser que sempre retorna a própria entrada

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*:

```
public class OrdChoice implements Parser {
  public Parser p1;
  public Parser p2;
  public OrdChoice(Parser _p1, Parser _p2) {
    p1 = _p1; p2 = _p2;
  }
  public List<List<Token>> parse(List<Token> input) {
    List<List<Token>> res = p1.parse(input);
    if(!res.isEmpty()) { return res; } else {return p2.parse(input); }
  }
}
```

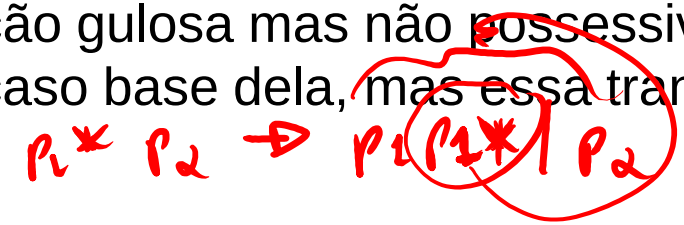


Repetição gulosa e possessiva

- Substituindo `choice` por `ordchoice` 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 o parser que seguiria a repetição ser caso base dela, mas essa transformação é global



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

Escolha LL(1)

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

```
public class PredChoice implements Parser {
  public Predicate<Token> pred;
  public Parser p1;
  public Parser p2;
  public PredChoice(Predicate<Token> _pred, Parser _p1, Parser _p2) {
    pred = _pred; p1 = _p1; p2 = _p2;
  }
  @Override
  public List<List<Token>> parse(List<Token> input) {
    Token fst = input.get(0);
    if(pred.test(fst)) {
      return p1.parse(input);
    } else {
      return p2.parse(input);
    }
  }
}
```