Linguagens de Domínio Específico

Fabio Mascarenhas – 2017.1

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

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);
    }
  }
}
```

Recursão

- Uma regra gramatical não recursiva pode ser construída usando os combinadores que já temos e usando variáveis simples, mas com isso não podemos representar regras recursivas ou declarar regras em qualquer ordem

- Um jeito de obter recursão (inclusive recursão indireta) é representanto uma *gramática* como um mapeamento entre nomes e parsers: `Map<String,Parser>`

- Agora podemos ter um combinador `variable` que, dada uma gramática e um nome, consulta aquele nome na gramática em seu método parse

- Outro modo de ter recursão é com um `LazyParser` que recebe um `Supplier<Parser>`

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 sufixo da entrada
- Podemos até usar uma exceção para sinalizar uma falha
- Vamos reconstruir nossos combinadores para serem determinísticos