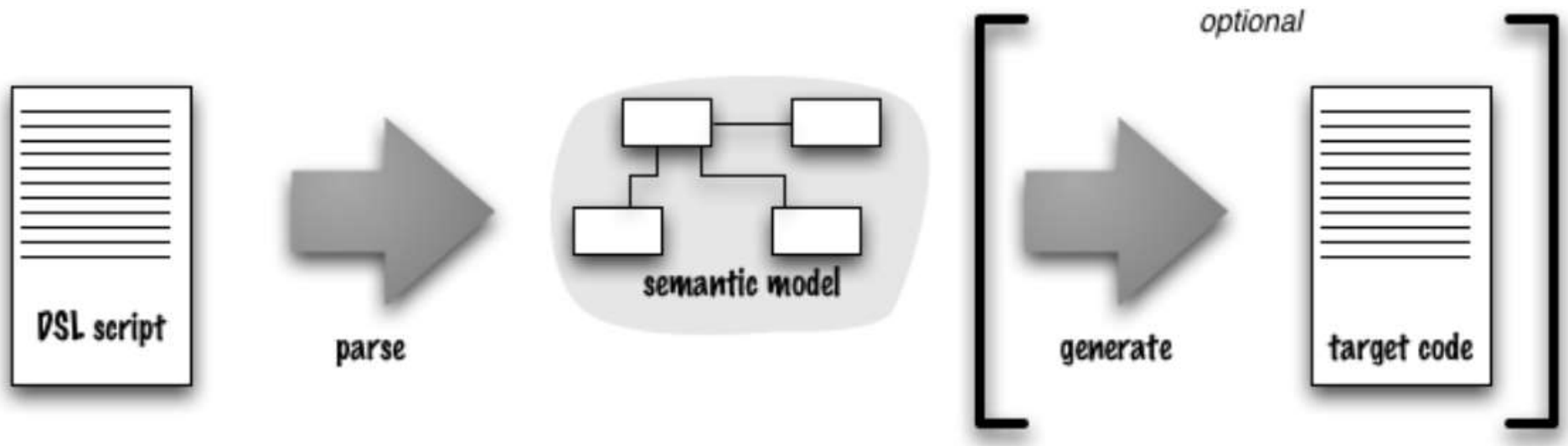
Linguagens de Domínio Específico

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

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;
  }
}
```

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); }
  }
}
```

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

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

Erros

- Uma falha em um parser de combinadores tem dois significados:
  - A alternativa que estamos tentando não está correta, mas outra estará
  - A entrada tem um erro de sintaxe
- A união dessas duas condições em um único estado do parser causa problemas para gerar boas mensagens de erro para o usuário!
- Um erro de sintaxe pode fazer o parser todo falhar sem indicar onde, ou fazer ele consumir apenas parte da entrada

Erros - exemplo

- Vamos ver o que acontece com a gramática a seguir:

```
bloco  -> stat*
stat   -> "while" exp "do" bloco "end" | NAME "=" exp
exp    -> aexp ">" aexp | aexp
aexp   -> termo (aop termo)*
termo  -> fator (mop fator)*
fator  -> NUM | NAME | "(" exp ")"
aop    -> "+" | "-"
mop    -> "*" | "/"
```

- Vamos analisar o programa abaixo, que tem um erro de sintaxe:

```
n = 5
f = 1
while n > 0 do
    f = f * n
    n n – 1       -- falta um =
end
```