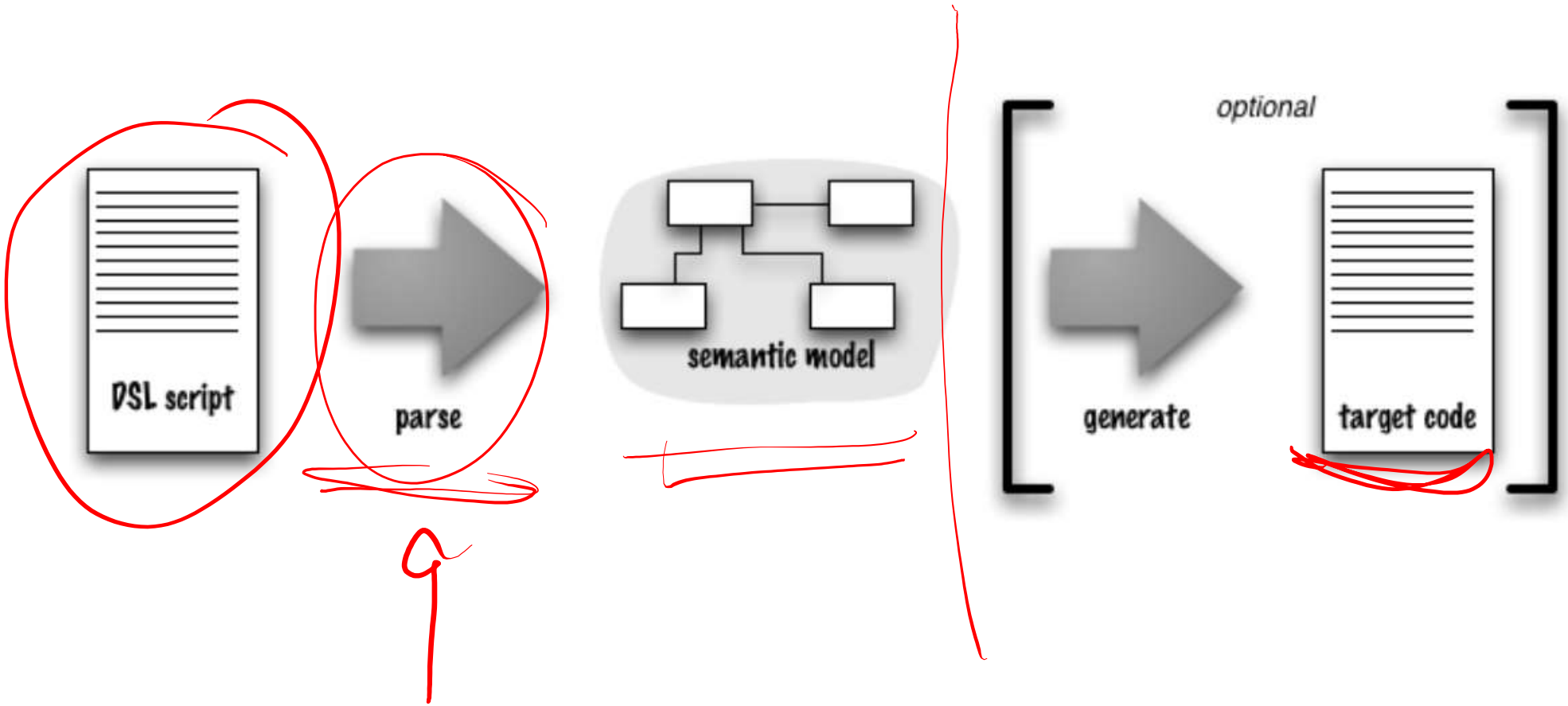
Linguagens de Domínio Específico

Fabio Mascarenhas – 2016.1

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

Processamento de uma DSL



PEGs

- As gramáticas de expressões de parsing, ou PEGs (parsing expression grammars) são uma linguagem para especificar parsers determinísticos

- Ao contrário das gramáticas livres de contexto, PEGs têm um mapeamento natural para os combinadores que estamos usando

- Uma *expressão de parsing* pode ser a expressão vazia `''`, um *terminal* `'a'`, um *não-terminal* `A`, uma *sequência* `pq`, onde `p` e `q` são expressões de parsing, uma *escolha ordenada* `p/q`, uma *repetição* `p*`, ou um *predicado* `!p`

- Uma PEG é simplesmente um mapeamento entre nomes de não-terminais e suas expressões de parsing

Exceções para falhas

- Para construir árvores vamos ter algumas sequências bem mais complicadas do que o combinador `seq` binário

- Expressar essas sequências fica mais fácil se as falhas forem exceções

```
public class Fail extends RuntimeException {
  public final int error;
  public final Set<String> expected;
  ...
}
```

```
public Fail fail(int pos, String exp) {
  if(pos < error) {
    return new Fail(pos, exp);
  }
  if(pos == error) {
    expected.add(exp);
  }
  return new Fail(error, expected);
}
```

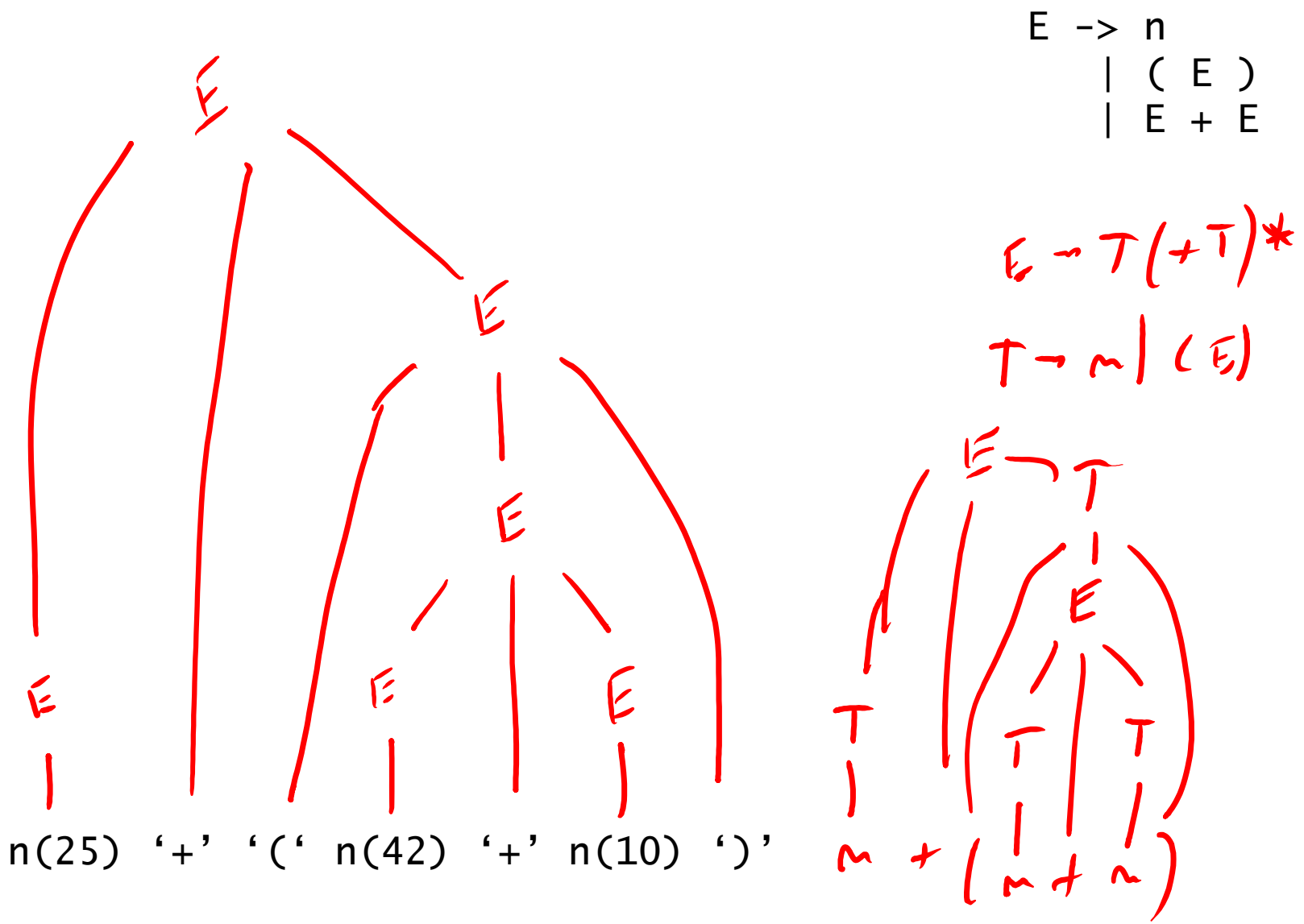
Árvore Sintática e AST

- A estrutura gramatical de uma entrada forma naturalmente uma árvore: tokens são folhas, outros termos sintáticos são nós internos

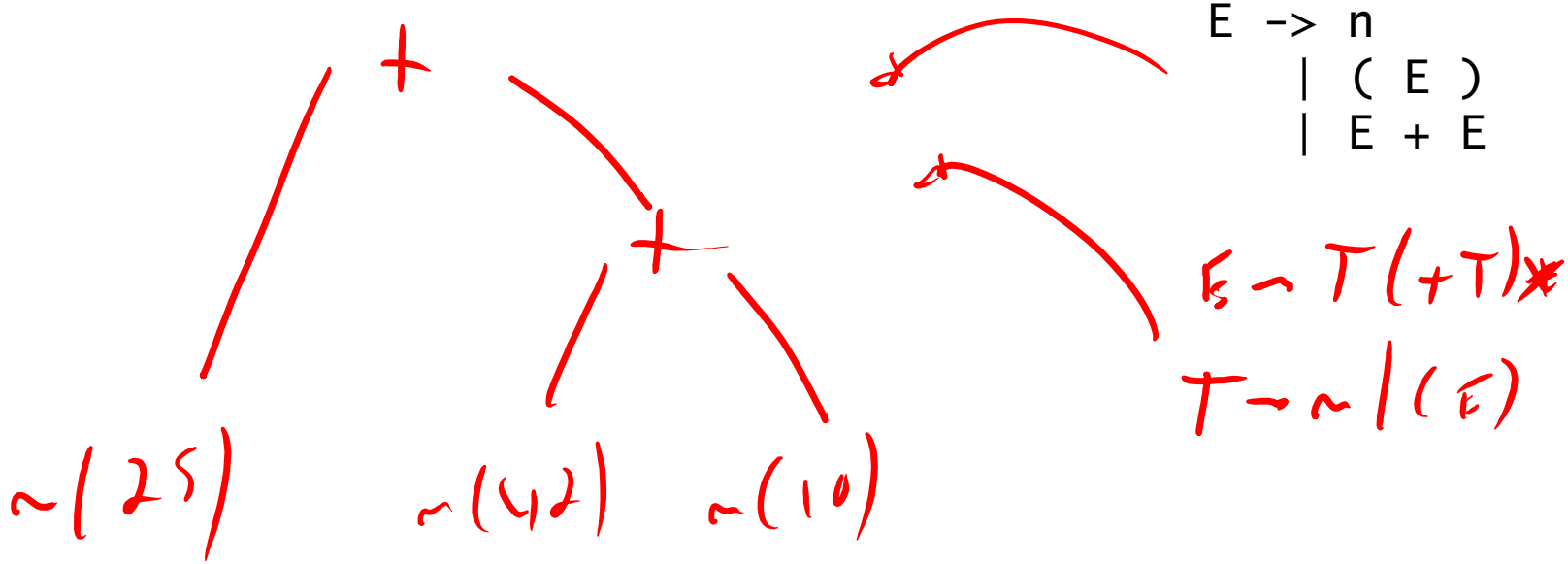
- Uma árvore sintática tem muita informação redundante, e vimos também que a gramática mais natural para uma linguagem nem sempre é a melhor para escrever um analisador para ela

- Resolvemos os dois problemas com uma *árvore sintática abstrata*: toda informação redundante é descartada, e procuramos a representação ideal da estrutura gramatical daquela linguagem, independentemente da gramática concreta que usamos

Exemplo – árvore sintática



Exemplo - AST




```
n(25) ‘+’ ‘(‘ n(42) ‘+’ n(10) ‘)’
```

Características de uma AST

- *Densa*: não há nós redundantes
- *Conveniente*: é fácil de percorrer e processar
- *Significativa*: Enfatiza operadores, operandos, e a relação entre eles, e não artefatos da gramática
- Podemos precisar escrever múltiplas passadas no processo de análise de um programa e a construção de seu modelo semântico
- Mudanças na gramática devido à conveniência da análise sintática não devem afetar a AST

ASTs homogêneas

- Uma forma de implementar uma AST em uma linguagem OO é como uma simples árvore:

```
public class AST {
    Token token;        // node is derived from which token?
    List<AST> children; // operands
    public AST(Token token) { this.token = token; }
    public void addChild(AST t) {
        if ( children==null ) children = new ArrayList<AST>();
        children.add(t);
    }
}
```

- O principal problema dessa representação é que perdemos todo o suporte do compilador para checar se estamos usando a AST de modo correto

- Por outro lado, essa implementação facilita escrever ferramentas para criar e manipular ASTs de modo mais automático

ASTs heterogêneas

- Outra forma de implementar uma AST é usar uma classe diferente para cada tipo de nó que temos

- Classes abstratas ou interfaces implementam estruturas sintáticas que possuem diversos tipos concretos de nó

```
public abstract class AST {
    Token token;
    // missing normalized list of children; subclasses define fields
}
public abstract class ExprNode extends AST { ... }
public class AddNode extends AST {
    ExprNode left, right;          // irregular, named fields
    «fields-specific-to-AddNode»
}
```

- Se estamos escrevendo o código para criar e processar as árvores manualmente essa forma é a ideal

Construindo árvores (1)

- Transformar um analisador sintático que apenas reconhece em um que gera uma árvore sintática é simples; para um analisador recursivo, bastam pequenas mudanças no código de `match` e de cada regra

```
void «rule»() {
    RuleNode r = new RuleNode("«rule»");
    if ( root==null ) root = r;    // we're the start rule
    else currentNode.addChild(r); // add this rule to current node
    ParseTree _save = currentNode;
    currentNode = r;               // "descend" into this rule
    «normal-rule-code»
    currentNode = _save;           // restore node to previous value
}
```

```
class MyParser extends Parser {
    ParseTree root;              // root of the parse tree
    ParseTree currentNode;       // the current node we're adding children to
    public void match(int x) {       // override default behavior
        currentNode.addChild(LT(1)); // add current lookahead token node
        super.match(x);              // match as usual
    }
    «rule-methods»
}
```

Construindo árvores (2)

- Mesmo que o processamento seja melhor feito em uma AST, construir árvores sintáticas completas pode ser útil no processo de depuração de uma gramática

- Em uma gramática de combinadores, precisamos definir novos combinadores que, ao invés de retornar apenas um sufixo da entrada, retornam uma combinação de uma lista de nós e um sufixo

```
public interface Parser<T> {
  Result<T> parse(State<T> in);
}

public class Result<T> {
  public final List<ParseTree> nodes;
  public final State<T> out;
  ...
}
```