

# Paradigmas de Linguagens de Programação (PLP)

**Professor: Carlo Marcelo Revoredo da Silva**

**Contato: [revoredo@gmail.com](mailto:revoredo@gmail.com)**



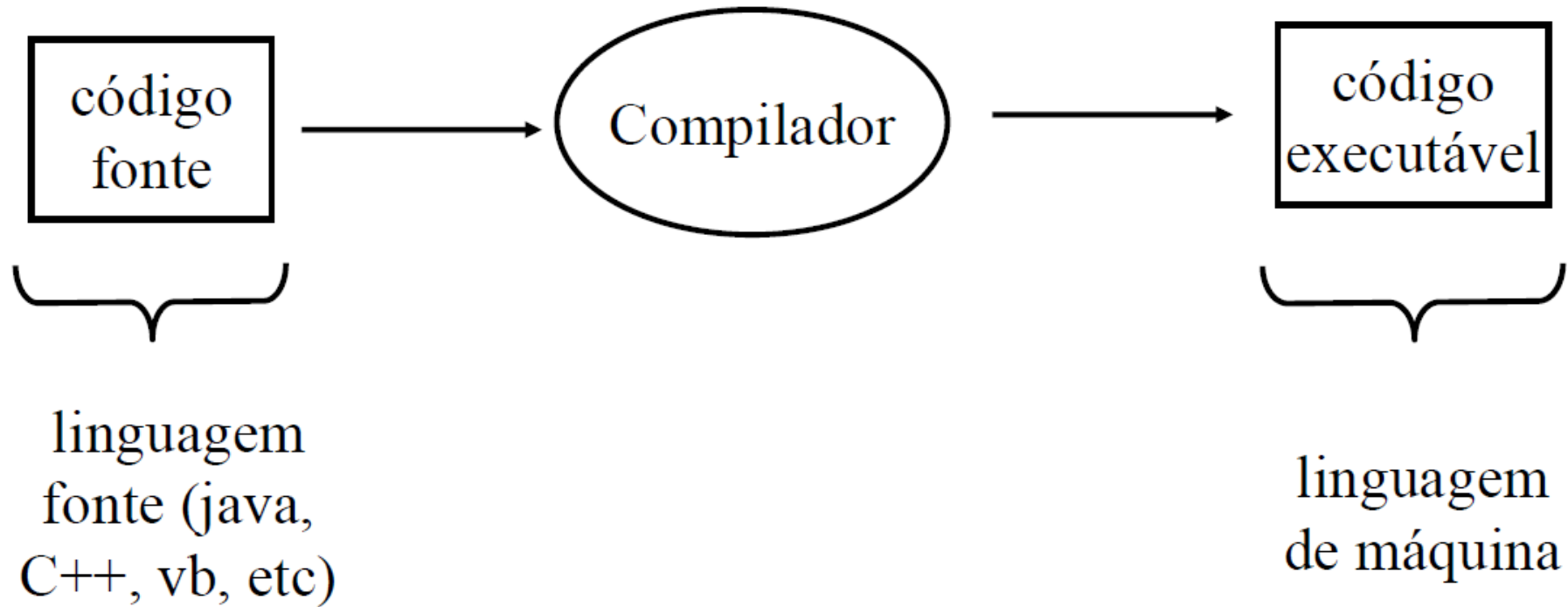
# Sumário

- **O processo de compilação: análises léxica, sintática e semântica**
- **Compilação x Interpretação**
- **ANTLR**
- **DSL**

# O processo de compilação

# Compiladores

## O processo de compilação



# Compiladores

- A compilação de um programa envolve tipicamente os seguintes eventos:
  - Análise léxica
  - Análise sintática
  - Análise semântica
  - Tradução

# Compiladores

- Análise Léxica
  - Ler o texto do código-fonte, caractere a caractere
  - Identificar elementos léxicos da linguagem: identificadores, palavras reservadas, constantes, operadores (TOKENS)
  - Ignorar comentários, brancos, tabs, quebra de linhas
  - Processar importações de bibliotecas

# Compiladores

- Análise Sintática

- Identificar na sequência de elementos léxicos as construções da linguagem
- ```
if (x > y) {  
    System.out.println("Maior!");  
} else {  
    System.out.println("Menor ou igual!");  
}
```
- Identifica a expressão relacional  $x > y$
- Identifica uma estrutura de seleção composta
- Quais operações estão associadas aos desvios da estrutura



# Compiladores

- Análise Sintática

- A parte sintática da linguagem gera muitos nós de forma homogênea, resultando em uma árvore de sintaxe;
- Essa árvore, genuinamente construída pelo programador, pode trazer uma série de problemas:
  - Muita informação redundante (ambiguidades)
  - Muitos separadores, terminadores, etc de forma desnecessária
  - É preciso um mecanismo que possa “enxugar”, “compactar”, toda a regra sintática de tal forma que facilite a vida do compilador
  - O mecanismo é a árvore de sintaxe abstrata (AST)
  - O analisador sintático que usa uma AST é conhecido por Parser



# Compiladores

- Análise Léxica x Análise Sintática
  - Tem papeis parecidos, já que as duas tratam sequências de símbolos
  - O escopo das duas são separadas por questão de simplicidade e separação de responsabilidades do compilador

# Compiladores

- Análise Semântica
  - Verificar se as construções identificadas pela análise sintática estão de acordo com as “regras semânticas” da linguagem.
  - Por exemplo, por ser fortemente tipada, a linguagem java exige que todas as variáveis devem ser declaradas antes de seu uso.
  - É função da análise semântica verificar se as variáveis foram devidamente declaradas.

# Compiladores

- Análise Semântica
  - Muitos erros no programa não podem ser detectados sintaticamente, pois precisam de um contexto
    - Exemplo: nomes de variáveis x escopo da variável
  - Diferenciar variáveis x métodos
  - Possibilitar recursos da linguagem (Herança)
    - Atributo herdado e atributo sobrescrito
    - Métodos herdados e métodos sobrescritos
    - Métodos sobrecarregados

# Compiladores

- O que é uma gramática?
  - Ferramenta para descrever uma linguagem e seus símbolos, ou seja, a definição dos analisadores léxicos e sintáticos

# Compilação x Interpretação

# Compiladores x Interpretadores

- O Interpretador traduz um código de alto nível para uma linguagem intermediária, que não é de alto nível mas também não é de máquina, e aguarda alguma solicitação de chamada das instruções traduzidas.
- A medida que as chamadas ocorrem, o interpretador vai traduzindo a linguagem intermediária para linguagem de máquina a medida que o fragmento é solicitado.
- Já o compilador traduz tudo de uma vez para código de máquina. Por isso que, em geral, o tempo de compilação é mais longo, porém, o tempo de execução de um código compilado é menor do que do código interpretado.

# Compiladores x Interpretadores

- Ou seja, na interpretação, por exemplo um arquivo HTML, é possível fazer modificações sem precisar compilar toda a aplicação. Geralmente são múltiplos arquivos fragmentados com responsabilidades distintas.
- Já um código compilado resulta em um único executável, qualquer mudança em uma parte do código precisa compilar todo o código fonte.
- Cada um tem suas vantagens e desvantagens



# ANTLR

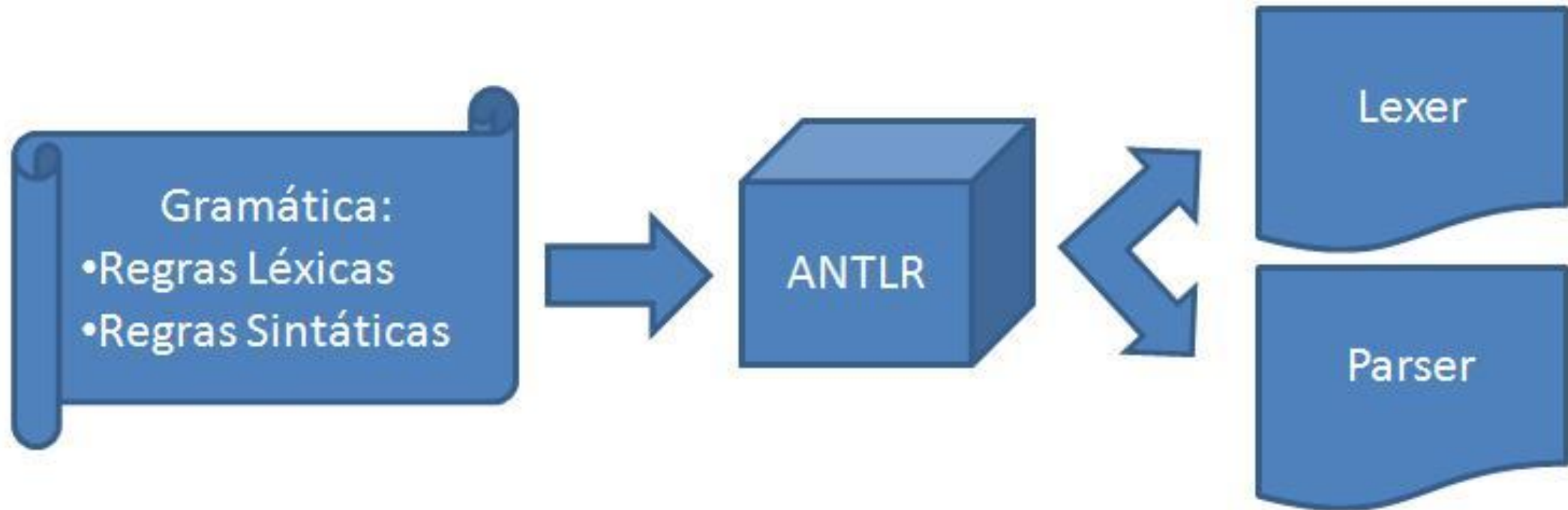
# ANTLR

- **AN**other **T**ool for **L**anguage **R**ecognition
- Processo de construção de uma linguagem de programação
- Possibilita o desenvolvimento de compiladores, interpretadores, e afins
- Com ele você constrói sua própria gramática, ou seja, define seu analisador léxico e sintático
- Desenvolvida por Terrence Parr



# Funcionamento

- Através da definição de uma gramática, o ANTLR fica responsável por gerar o analisador léxico (Lexer) e o analisador Sintático (Parser)



# Por que usar o ANTLR?

- Tem uma IDE própria (ANTLRWORKS)
- Plugin no Eclipse
- Ferramenta bem consolidada e leve
- Da suporte a diversas linguagens como C/C++, C#, Java, Python, etc;

# Por que usar o ANTLR?

ANTLR  
Editor



The screenshot displays the ANTLR Editor interface. The top window shows the grammar file `/Users/bovet/ Demo/objc.g` with the following rules:

```
compound_statement
: RCURLY declaration_list? statement_list? LCURLY
;

statement_list
: statement+
;

selection_statement
: 'if' LPAREN expression RPAREN statement ('else' statement)?
;
'switch' LPAREN expression RPAREN statement
;

iteration_statement
: 'while' LPAREN expression RPAREN statement
;
'do' statement
;
'for' LPAREN expression SEMI expression SEMI expression RPAREN statement
;

jump_statement
: 'goto' identifier SEMI statement
;
'continue' SEMI statement
;
'break' SEMI statement
;
'return' expression SEMI statement
;
```

The left sidebar shows a tree of rules, with `Statement` expanded. The bottom window shows a syntax diagram for the `iteration_statement` rule, illustrating the flow of tokens and non-terminals. The diagram shows three paths for the `iteration_statement` rule: `'while' LPAREN expression RPAREN statement`, `'do' statement`, and `'for' LPAREN expression SEMI expression SEMI expression RPAREN statement`. The status bar at the bottom indicates 129 rules and 452:23 lines.

# Por que usar o ANTLR?

ANTLR  
Interpretador



The screenshot displays the ANTLR IDE interface. At the top, the file path is `/Users/bovet/ Grammars/java.g`. The editor shows a grammar rule for `expression` with the comment `// the mother of all expressions` and the definition `expression : assignmentExpression ;`. A sidebar on the left lists the grammar's non-terminals: `handler`, `expression`, `expressionList`, `assignmentExpression`, and `conditionalExpression`. Below the editor, a tab labeled `field` is active, showing a Java code snippet: `public void main() { int a = 2+3; }`. The main area displays a syntax tree for the `field` rule. The root node is `<grammar JavaParser>`, which branches into `field`. This `field` node further branches into `modifiers`, `typeSpec`, `main`, `{`, `parameterDeclarationList`, `}`, and `declaratorBrackets`. The `modifiers` node branches into `modifier`, which then branches into `public`. The `typeSpec` node branches into `builtinTypeSpec`, which then branches into `builtinType`, which finally branches into `void`. The `declaratorBrackets` node branches into `{`, `modifiers`, `typeSpec`, `builtinTypeSpec`, `builtinType`, `a`, and `declaratorBrackets`. The `builtinType` node branches into `int`. The `a` node branches into `=`. The `declaratorBrackets` node branches into `}`. At the bottom, there is a `Zoom` slider and a tab bar with `Syntax Diagram`, `Interpreter` (selected), `Debugger`, and `Console`. The status bar at the bottom left shows `132 rules` and `528:1`.

# Por que usar o ANTLR?

ANTLR  
*Debugger*



The screenshot displays the ANTLR Debugger interface with the following components:

- File Editor:** Shows the file `/Users/bovet/Development/Research/depot/antlr/examples-v3/java/java/java.g`. The grammar rules for `variableDeclaratorId`, `variableInitializer`, `arrayInitializer`, and `modifier` are visible. The `modifier` rule is currently selected, showing its definition: `annotation, 'public', 'protected', 'private', 'static', 'abstract', 'final', 'native', 'synchronized', 'transient', 'volatile', 'strictfp'.`
- Parse Tree:** A hierarchical tree structure showing the parsed input. The root is `compilationUnit`, which contains `typeDeclaration`, which in turn contains `classOrInterfaceDeclaration`. This node branches into `modifier` (with value `public`) and `classDeclaration`. `classDeclaration` further branches into `normalClassDeclaration`, which contains `class` (with value `Sample`) and `classBody`. The `classBody` node contains `classBodyDeclaration`, which branches into `modifier` (with value `public`) and another `classBodyDeclaration`.
- Input:** The source code being parsed: 

```
public class Sample {  
    public void main() {  
        System.out.println("Hello, world");  
    }  
}
```
- Stack:** A list of the current stack of rules, numbered 0 to 7:
  - 0: `compilationUnit`
  - 1: `typeDeclaration`
  - 2: `classOrInterfaceDeclaration`
  - 3: `classDeclaration`
  - 4: `normalClassDeclaration`
  - 5: `classBody`
  - 6: `classBodyDeclaration`
  - 7: `modifier`
- Bottom Bar:** Includes tabs for `Syntax Diagram`, `Interpreter`, `Debugger` (active), and `Console`. It also shows `148 rules (2 warnings)` and `254:9 Warnings reported in console`.



**Vamos construir nossa  
primeira DSL**

# DSL

- **Uma DSL**

- **Domain Specific Language** (Linguagem específica de domínio)
- Usa a abstração para aumentar o entendimento de domínio
- Ou seja, estritamente relacionada com um domínio de aplicação
- Muito poderosa para o domínio em questão, muito provavelmente inútil quando fora do seu respectivo domínio
- Exemplos: SQL, HTML

- **DSL é diferente de uma GPL**

- **General Purpose Language** (Linguagens para propósitos gerais)
- Java, C#, etc

# Let's Code!



# Grato pela atenção

**Disciplina:** Paradigmas de Linguagens de Programação (PLP)

**Professor:** Carlo Marcelo Revoredo da Silva

**Contato:** [revoredo@gmail.com](mailto:revoredo@gmail.com)

