

第二阶段实验报告：基于监听器模式的应用开发

姓名：闫悦斌 学号：3022244121

班级：计算机科学与技术 1 班 小组：第 9 小组

2025 年 12 月 8 日

摘要

本阶段实验在第一阶段完成的语法分析器基础上，利用 ANTLR 提供的监听器 (Listener) 机制，开发了四个实际应用：CSV 数据加载器、JSON-XML 转换器、Cymbol 调用图生成器以及 Cymbol 语义验证器。实验重点在于理解 ParseTreeWalker 的遍历机制，以及如何通过重写 exit/enter 方法在遍历过程中收集信息、转换数据和执行语义检查。

目录

1 实验概述	3
2 任务一：CSV 数据加载器	3
2.1 任务目标	3
2.2 实现逻辑详解	3
2.3 核心代码实现	3
2.4 运行结果	4
3 任务二：JSON 转 XML 翻译器	4
3.1 任务目标	4
3.2 实现逻辑详解	4
3.3 核心代码实现	5
3.4 运行结果	5
4 任务三：Cymbol 调用图生成	6
4.1 任务目标	6
4.2 实现逻辑详解	6
4.3 核心代码实现	7
4.4 运行结果	7
5 任务四：符号语义验证	8
5.1 任务目标	8
5.2 实现逻辑详解	8
5.3 核心代码实现	9
5.4 运行结果	9
6 实验总结	9

1 实验概述

第二阶段的核心任务是从“语法定义”转向“语义处理”。不再修改语法规则（除了添加 Label 以生成精确的事件回调），而是编写 Java 代码继承 `BaseListener` 类。通过维护符号表、栈或列表等辅助数据结构，在语法树的遍历过程中捕获关键节点信息，实现对上下文敏感逻辑的处理。

2 任务一：CSV 数据加载器

2.1 任务目标

将 CSV 文本文件解析并加载为 Java 中的 `List<Map<String, String>>` 结构，模拟数据库或配置文件的读取过程，将非结构化的文本转换为结构化的内存对象。

2.2 实现逻辑详解

由于 CSV 是扁平的行结构，采用“行缓冲”策略进行解析：

- **数据存储**：定义一个全局列表 `rows` 用于存储最终结果，以及一个临时列表 `currentRow FieldValues` 用于缓存当前正在解析的行数据。
- **行处理流程**：
 - 在 `enterRow` 时，初始化清空临时列表，准备接收新数据。
 - 在 `exitText` 或 `exitString` 时，提取字段文本（对于字符串需去除引号）并加入临时列表。
 - 在 `exitRow` 时，这是最关键的步骤。程序将临时列表中的数据与预先解析好的 `header`（表头）列表进行索引对齐。通过遍历，构建出 列名 -> 值的 `Map` 对象，并将其加入全局结果集中。
- **特殊处理**：需要区分当前行是标题行还是数据行，这通过检查当前上下文的父节点规则索引（`ctx.getParent()`）来实现。

2.3 核心代码实现

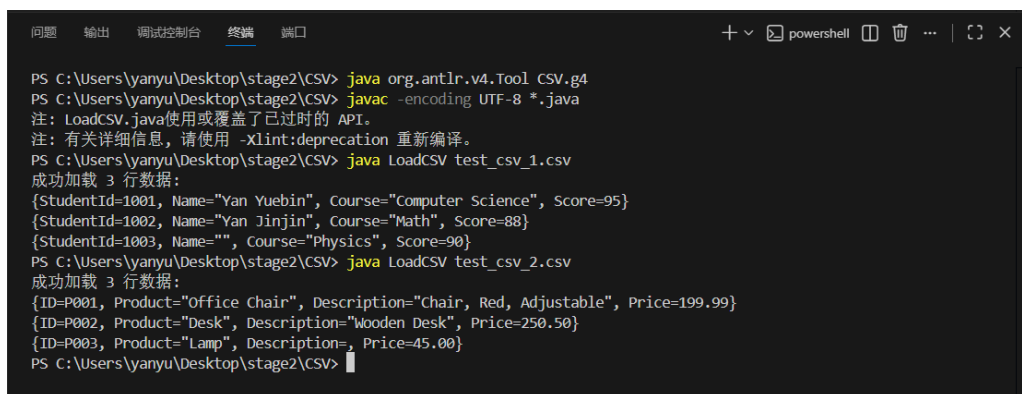
```
1 public void exitRow(CSVParser.RowContext ctx) {  
2     //如果是表头行则跳过处理  
3     if (ctx.getParent().getRuleIndex() == CSVParser.RULE_hdr) return;  
4  
5     Map<String, String> m = new LinkedHashMap<>();
```

```
6      int i = 0;
7      // 将当前行数据与表头对齐
8      for (String v : currentRowFieldValues) {
9          if (i < header.size()) {
10             m.put(header.get(i), v);
11         }
12         i++;
13     }
14     rows.add(m);
15 }
```

Listing 1: LoadCSV 数据对齐逻辑

2.4 运行结果

测试了包含特殊字符、空字段和价格小数点的 CSV 文件，程序正确输出了映射后的数据。



```
PS C:\Users\yanyu\Desktop\stage2\CSV> java org.antlr.v4.Tool CSV.g4
PS C:\Users\yanyu\Desktop\stage2\CSV> javac -encoding UTF-8 *.java
注: LoadCSV.java使用或覆盖了已过时的 API。
注: 有关详细信息, 请使用 -Xlint:deprecation 重新编译。
PS C:\Users\yanyu\Desktop\stage2\CSV> java LoadCSV test_csv_1.csv
成功加载 3 行数据:
{StudentId=1001, Name="Yan Yuebin", Course="Computer Science", Score=95}
{StudentId=1002, Name="Yan Jinjin", Course="Math", Score=88}
{StudentId=1003, Name="", Course="Physics", Score=90}
PS C:\Users\yanyu\Desktop\stage2\CSV> java LoadCSV test_csv_2.csv
成功加载 3 行数据:
{ID=P001, Product="Office Chair", Description="Chair, Red, Adjustable", Price=199.99}
{ID=P002, Product="Desk", Description="Wooden Desk", Price=250.50}
{ID=P003, Product="Lamp", Description="", Price=45.00}
PS C:\Users\yanyu\Desktop\stage2\CSV>
```

图 1: CSV 数据加载运行结果

3 任务二：JSON 转 XML 翻译器

3.1 任务目标

将嵌套的 JSON 数据结构转换为等价的 XML 格式字符串。这是一个典型的“源到源”翻译（Source-to-Source Translation）任务，要求保持数据层级结构不变。

3.2 实现逻辑详解

JSON 是递归结构，不能简单使用全局变量存储。本任务利用了 ANTLR 的 Parse TreeProperty 类来实现节点注解（Annotation）：

- **节点注解机制**: 由于 Listener 的方法没有返回值, 使用 `ParseTreeProperty<String>` 作为一个哈希映射, 将每个节点对应的翻译结果 (XML 片段) “挂载” 在该节点对象上。
- **自底向上翻译**:
 - **叶子节点** (如 `String`, `Atom`): 直接获取文本, 处理转义后存储为 XML 内容。
 - **中间节点** (如 `Pair`): 从子节点获取值, 结合 Key 生成 `<Tag>Value</Tag>` 格式的字符串。
 - **聚合节点** (如 `Object`, `Array`): 在 `exit` 方法中遍历所有子节点, 将它们的 XML 片段拼接起来, 作为当前节点的翻译结果。
- 这种方法避免了复杂的递归调用管理, 利用树遍历本身的顺序自然完成了数据的层层包裹。

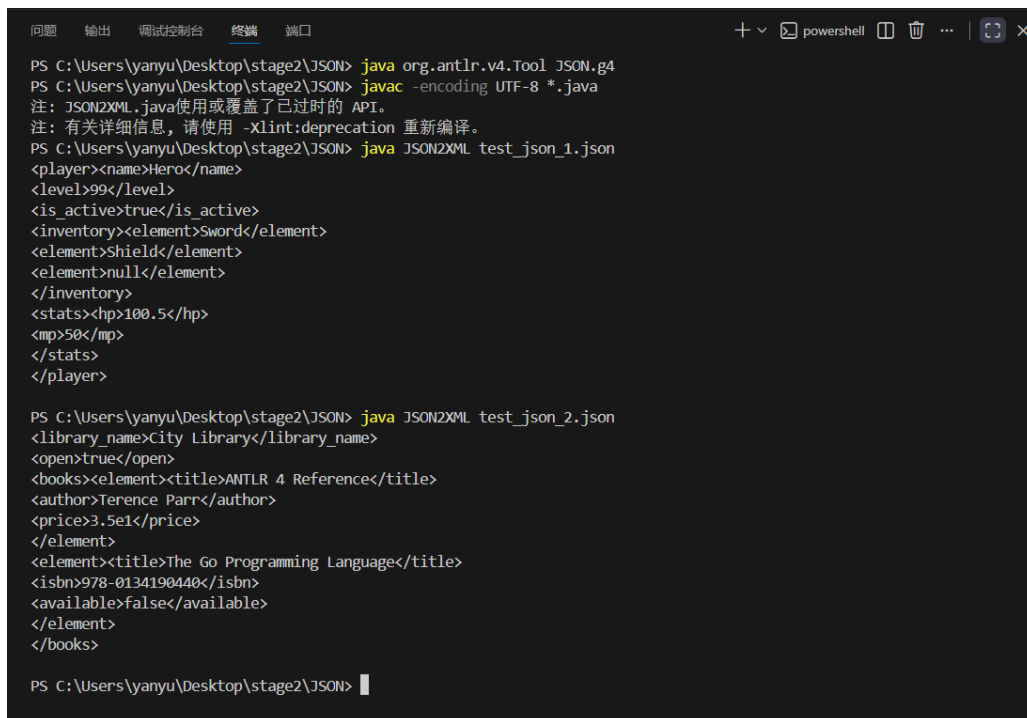
3.3 核心代码实现

```
1 public void exitPair(JSONParser.PairContext ctx) {
2     String tag = stripQuotes(ctx.STRING().getText());
3     // 获取子节点已经翻译好的 XML 内容
4     String content = getXML(ctx.value());
5     // 拼接成 XML 标签并挂载到当前节点
6     String result = String.format("<%s>%s</%s>\n", tag, content, tag);
7     setXML(ctx, result);
8 }
```

Listing 2: JSON2XML 节点注解逻辑

3.4 运行结果

测试了包含对象嵌套、数组以及科学计数法的 JSON 文件, 生成的 XML 结构准确无误。



```
PS C:\Users\yanyu\Desktop\stage2\JSON> java org.antlr.v4.Tool JSON.g4
PS C:\Users\yanyu\Desktop\stage2\JSON> javac -encoding UTF-8 *.java
注: JSON2XML.java使用或覆盖了已过时的 API。
注: 有关详细信息, 请使用 -Xlint:deprecation 重新编译。
PS C:\Users\yanyu\Desktop\stage2\JSON> java JSON2XML test_json_1.json
<player><name>Hero</name>
<level>99</level>
<is_active>true</is_active>
<inventory><element>Sword</element>
<element>Shield</element>
<element>null</element>
</inventory>
<stats><hp>100.5</hp>
<mp>50</mp>
</stats>
</player>

PS C:\Users\yanyu\Desktop\stage2\JSON> java JSON2XML test_json_2.json
<library_name>City Library</library_name>
<open>true</open>
<books><element><title>ANTLR 4 Reference</title>
<author>Terence Parr</author>
<price>3.5e1</price>
</element>
<element><title>The Go Programming Language</title>
<isbn>978-0134190440</isbn>
<available>false</available>
</element>
</books>

PS C:\Users\yanyu\Desktop\stage2\JSON>
```

图 2: JSON 转 XML 转换结果

4 任务三: Cymbol 调用图生成

4.1 任务目标

通过静态代码分析, 提取 Cymbol 语言中的函数调用关系, 并生成 Graphviz DOT 格式的图描述文件, 用于可视化程序结构。

4.2 实现逻辑详解

这是一个上下文敏感的分析任务, 核心在于知道“当前是谁在调用谁”:

- **上下文状态维护:** 定义一个成员变量 `currentFunctionName`。当遍历器进入函数定义节点 (`enterFunctionDecl`) 时, 将该变量更新为当前函数名; 当离开时, 理论上应置空或恢复, 但由于 Cymbol 不支持嵌套函数定义, 简单更新即可。
- **调用关系捕获:** 在 `exitCall` 事件中, 检查当前是否处于某个函数上下文中 (即 `currentFunctionName` 不为空)。如果是, 则说明发生了一次调用, 记录一条从“当前函数”指向“被调函数”的边。
- **数据去重:** 使用 `Set<String>` 存储边信息, 自动过滤掉重复的调用关系 (例如递归调用或循环中的多次调用), 确保生成的调用图简洁清晰。

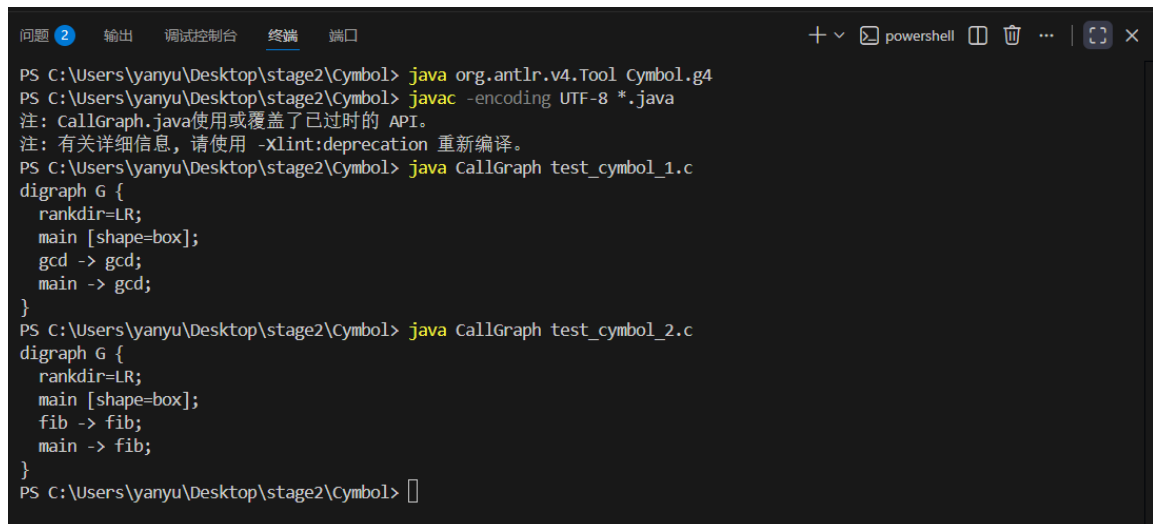
4.3 核心代码实现

```
1 public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {  
2     // 进入函数定义时, 更新上下文状态  
3     currentFunctionName = ctx.ID().getText();  
4 }  
5  
6 public void exitCall(CymbolParser.CallContext ctx) {  
7     String funcName = ctx.ID().getText();  
8     // 只有在函数内部的调用才会被记录  
9     if (currentFunctionName != null) {  
10         edges.add(" " + currentFunctionName + " -> " + funcName + ";");  
11     }  
12 }
```

Listing 3: CallGraph 上下文捕获逻辑

4.4 运行结果

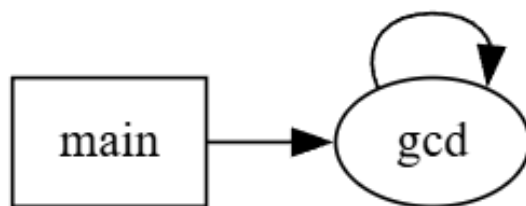
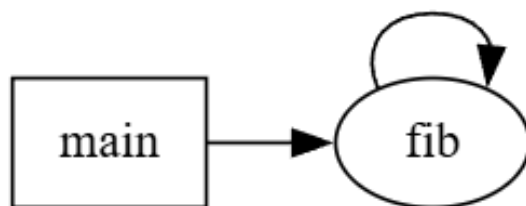
首先运行程序对 `test_cymbol_1.c` (GCD) 和 `test_cymbol_2.c` (Fibonacci) 进行分析, 控制台成功输出了对应的 DOT 格式描述代码。



```
问题 2 输出 调试控制台 终端 端口  
PS C:\Users\yanyu\Desktop\stage2\Cymbol> java org.antlr.v4.Tool Cymbol.g4  
PS C:\Users\yanyu\Desktop\stage2\Cymbol> javac -encoding UTF-8 *.java  
注: CallGraph.java使用或覆盖了已过时的 API。  
注: 有关详细信息, 请使用 -Xlint:deprecation 重新编译。  
PS C:\Users\yanyu\Desktop\stage2\Cymbol> java CallGraph test_cymbol_1.c  
digraph G {  
    rankdir=LR;  
    main [shape=box];  
    gcd -> gcd;  
    main -> gcd;  
}  
PS C:\Users\yanyu\Desktop\stage2\Cymbol> java CallGraph test_cymbol_2.c  
digraph G {  
    rankdir=LR;  
    main [shape=box];  
    fib -> fib;  
    main -> fib;  
}  
PS C:\Users\yanyu\Desktop\stage2\Cymbol> 
```

图 3: 程序运行输出: 生成 DOT 格式代码

随后将生成的 DOT 代码通过 Graphviz 可视化, 得到了如下的函数调用关系图。

图 4: 可视化结果 1: GCD 递归调用 ($\text{main} \rightarrow \text{gcd}$)图 5: 可视化结果 2: Fibonacci 递归调用 ($\text{main} \rightarrow \text{fib}$)

5 任务四：符号语义验证

5.1 任务目标

实现一个编译器前端的关键组件——语义分析器。主要检查作用域规则，具体包括“变量重复定义”检查和“变量未定义引用”检查。

5.2 实现逻辑详解

为了正确模拟编程语言的词法作用域 (Lexical Scoping)，实现了一个简单的符号表：

- **作用域栈设计：**使用 `Stack<Set<String>>` 数据结构。栈顶代表当前作用域，栈底代表全局作用域。Set 集合存储该作用域内已定义的变量名。
- **作用域生命周期：**
 - `enterFunctionDecl`：入栈一个新的 Set，表示进入局部作用域。同时将函数参数添加进该 Set。
 - `exitFunctionDecl`：弹栈 (pop)，表示销毁当前局部作用域。
- **定义检查 (check definition)：**当遇到变量声明时，只检查**栈顶**（当前作用域）是否已包含该变量名。如果包含则报错“重复定义”，否则加入 Set。这允许了局部变量遮蔽全局变量 (Shadowing)。

- **引用解析 (resolve):** 当遇到变量使用时, 从**栈顶向下**遍历整个栈。如果在任何一个作用域中找到了该变量, 则验证通过; 如果遍历完所有作用域仍未找到, 则报错“未定义变量”。

5.3 核心代码实现

```
1 void resolveVar(String name, Token token) {
2     // 从栈顶（当前作用域）向栈底（全局作用域）查找
3     for (int i = scopes.size() - 1; i >= 0; i--) {
4         if (scopes.get(i).contains(name)) {
5             return; // 找到了，验证通过
6         }
7     }
8     // 遍历完所有作用域仍未找到
9     System.err.println("错误 (Line " + token.getLine() + "): 未定义变量 " +
10        name);
11 }
```

辑。特别是通过实现符号验证器，对**作用域链**、**符号表管理**以及**上下文敏感分析**等编译原理核心概念有了代码层面的深刻认知。这些技术不仅用于编译器开发，也广泛应用于静态代码分析工具和 IDE 插件的编写中。