使用ANTLR4实现的C++词法分析器和语法解析器

本项目使用ANTLR4实现了C++的词法分析器和语法解析器。词法分析器可以将C++代码标记化，并识别各种语言元素，如关键字、运算符、字面量、标识符和注释。语法解析器可以解析C++代码并识别各种语言元素，如语句、表达式和声明。

团队分工

@苏伟铭：搭建框架
@汪佳宇：基础语法逻辑
@陈立心@王旭冉：完善语法逻辑，进行相关测试
文档由大家统一完成。

测试用例

1. **KMP字符串匹配算法**（`test_KMP.cpp`）: 实现了经典的KMP字符串匹配算法，包含部分匹配表(next数组)的计算、字符串匹配过程、多重匹配支持和错误处理。
2. **多种排序算法**（`test_bubbleSort.cpp` 等）: 实现了基础的排序算法，支持动态数组输入、整数排序和结果可视化输出。
3. **回文检测算法**（`test_palindrome.cpp`）: 实现了回文检测的算法，支持动态字符串输入、忽略大小写和非字母数字字符，并且在遇到不匹配时提前终止检测，结果可视化输出。
4. **四则运算算法**（`test_calc.cpp`）: 实现了自定义栈结构的四则运算算法，支持负数处理、中缀表达式处理和运算符优先级，支持动态字符串输入。

输出示例

对于冒泡排序测试用例(test_bubbleSort.cpp):

输入（C++源代码）：

```
#include <iostream>

int main() {
    int num;
    int a[100];  // 固定大小数组

    // 输入数组长度
    std::cout << "Input the array length: ";
    std::cin >> num;

    // 输入整数
    std::cout << "Input integers: ";
    for (int i = 0; i < num; i++) {
        std::cin >> a[i];
    }
    // ... 代码其余部分
}
```

词法分析输出:

```
Token: INCLUDE              Text: #include
Token: LT                   Text: <
Token: ID                   Text: iostream
Token: GT                   Text: >
Token: INT                  Text: int
Token: ID                   Text: main
Token: LPAREN               Text: (
Token: RPAREN               Text: )
Token: LBRACE               Text: {
...
```

语法分析输出（AST节选）：

```
<AST>
<program>
  <includeStatement>
    <INCLUDE>#include</INCLUDE>
    <LT><</LT>
    <includeID>
      <ID>iosteam</ID>
      ...
```

特性支持

当前版本支持以下C++语言特性：

- 基本数据类型 (int, char, bool等)
- 控制流语句 (if-else, for, while)
- 数组操作
- 基本输入输出
- 函数定义和调用
- 运算符和表达式
- 变量声明和赋值

错误处理

解析器能够处理多种错误情况：

- 语法错误检测
- 输入文件错误处理
- 解析过程错误报告
- AST生成异常处理

难点与创新点

一、解析器规则的设计创新

- 模块化的语法规则设计
  - 文法规则通过模块化的方式拆分成不同部分，如类定义（classDefinition）、函数定义（functionDefinition）、声明（declaration）和语句（statement），这种设计提升了文法的可读性和可维护性，方便后续扩展。
- 支持类成员的多样化声明
  - 类体规则（classBody）支持访问控制符（accessSpecifier）和构造函数（constructor）的解析。
    允许类成员既可以是普通变量（memberDeclaration），也可以是函数和构造函数，从而灵活适配C++类的特性。
- 构造函数初始化列表的处理
  - 在constructor规则中，增加了对初始化列表（COLON functionCall+）的支持，这是C++类构造函数的一个重要特性。这种设计能够解析类似以下代码：

```
MyClass(int x) : a(x), b(0) {}
```

- 紧凑的变量声明和初始化支持
  - 将变量声明分为decl_（无初始化）和decl_assign（带初始化）两个独立规则。
    支持多变量声明和初始化的解析，例如：

```
int a, b = 2, c = 3;
```

- 灵活的类型解析
  - typeSpecifier支持基本数据类型（如int、double等）和用户自定义类型（如ID）。
    引入了作用域解析（ID SCOPE ID），允许解析诸如std::string这样的类型。
- 表达式优先级的分层处理
  - 表达式解析从logicalOrExpression一直到multiplicativeExpression，严格遵守C++标准的优先级。
  - 每层规则通过嵌套形式明确了操作符的优先级和结合性，例如+优先于<。
- 支持复杂的复合语句和作用域
  - compoundStatement规则允许在大括号内嵌套声明和语句，这种递归规则能够解析复杂嵌套代码块：

```
{
    int x = 0;
    if (x < 10) { x++; }
}
```

- 简单而实用的控制语句支持
  - selectionStatement支持if-else语句的解析。
  - iterationStatement规则解析了while循环和for循环，包括初始化表达式、循环条件和增量表达式。

二、AST构建

在实现AST构建的过程中，主要的难点在于如何通过CPPParserVisitor类递归遍历语法树并构建相应的AST节点。每个解析规则对应一个visit方法，在这些方法中，需要根据节点类型生成合适的Node对象，并将它们组织成树状结构。尤其是在处理复杂的语法结构时，如compoundStatement和functionDefinition等，需要确保正确地处理子节点的递归关系，并维护父子节点之间的连接。这种递归式的AST构建方式不仅保证了语法树的准确表达，还需要有效处理空节点和非法节点的情况，确保解析过程的健壮性和完整性。

三、创新点总结

- 面向需求的裁剪与优化
  - 当前实现的文法聚焦于C++的核心特性（如类定义、函数、控制结构等），避免了复杂特性（如模板、异常处理等）带来的解析复杂度。
- 面向实践的灵活扩展
  - 通过将类型、声明、表达式等设计为独立规则，提供了灵活的扩展能力，便于未来添加更多C++特性（如指针、switch语句等）。
- 提升代码复用性与可维护性
  - 模块化的文法定义不仅降低了开发难度，还便于单独调试和改进某些功能（如表达式优先级解析）。

开发过程

1. **语法开发：**
   - 实现基本表达式：算术和逻辑运算符、函数调用、变量引用、字面值
   - 添加语句支持：If-else条件、循环（for, while、Return语句、表达式语句
   - 实现函数定义：参数列表、返回类型、函数体、前向声明
   - 添加数组支持：数组声明、数组访问、多维数组、数组初始化
   - 集成错误处理：语法错误恢复、错误消息、警告生成、错误位置跟踪
2. **测试策略：**
   - 单个语法规则的单元测试
   - 完整程序的集成测试
   - 错误处理的边缘情况测试

参考资料

- ANTLR4文档
- C++文档