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。

团队成员

@王旭冉@苏伟铭@汪佳宇@陈立心

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
- 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，特点：

- 动态数组输入
- 整数排序
- 结果可视化输出
- 交换操作优化

3. 回文检测算法 (test_palindrome.cpp)

实现了回文检测的算法，特点：

- 动态字符串输入
- 忽略大小写和非字母数字字符
- 提前终止检测（遇到不匹配时立即返回结果）
- 结果可视化输出

4. 四则运算算法 (test_calc.cpp)

实现了四则运算的算法，特点：

- 自定义栈结构
- 支持四则运算
- 负数处理
- 中缀表达式处理
- 运算符优先级
- 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
- 有参数的构造函数

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

1. 语法设计

- **基本标记的定义与处理**：语法设计的核心挑战之一是如何准确地定义语言的基本标记（如关键字、运算符、标识符等）。不同于一般的语言，C++的语法结构复杂且多样，需要对关键字、运算符的优先级和结合性进行详细分析。这不仅要求能够正确处理简单的标记，还需要确保对复杂表达式的解析能够保持高效和准确。

- **表达式语法的优先级与结合性管理**：C++支持复杂的表达式，包括各种运算符的组合及其不同的优先级、结合性规则。实现一个高效的优先级爬升方法，使得解析器能够根据运算符的优先级正确地解析不同类型的表达式，避免运算符优先级错误，是设计中的一个重要难点。

- **综合类型系统的支持**：C++具有强大的类型系统，包括基本数据类型、指针、引用、类、模板等。如何在语法解析过程中管理并验证这些类型的关系，以及如何处理类型的推导和转换，是一个极具挑战性的部分。集成类型系统的过程中，如何确保类型信息的准确性与一致性，是创新点之一。

- **词法与语法的分离**：为了提高代码的可维护性与扩展性，必须将词法分析与语法分析明确分开。词法分析器负责源代码的标记化，而语法解析器则负责构建语法树。分离关注点的设计有助于简化后续的调试与功能拓展，同时提高了系统的模块化和灵活性。

2. AST（抽象语法树）构建

- **访问者模式的应用**：AST的构建是语法分析中的核心部分。使用访问者模式（Visitor Pattern）能够有效地遍历和处理语法树节点，确保每种语法构造都能得到准确的解析和处理。设计一个通用的`CPPParserVisitor`类来处理不同类型的节点，既能增强代码的可扩展性，也能在后续对语言的扩展过程中保持结构的一致性和清晰性。

- **语法树节点的设计与类型信息的存储**：构建AST时，如何设计合适的节点类型以表示不同的C++元素（如函数声明、变量声明、控制结构、数组访问等）是一个难点。每个节点需要能够承载特定的属性，如类型信息、符号表信息、源代码的位置信息等，这些信息对后续的分析和代码生成至关重要。

- **父子节点关系的管理与作用域支持**：构建层次化的父子节点关系，管理每个语法结构的上下文和作用域是一个创新点。在AST中，父子节点的引用不仅要保证准确性，还要支持多层次的作用域管理。这要求在AST节点中不仅能存储局部作用域的信息，还能支持跨作用域的符号解析与交叉引用。

- **符号表与作用域的集成**：每个代码块（如函数、类、控制语句等）都有其独立的作用域，如何高效管理这些作用域并确保符号的正确解析是关键。特别是对于变量遮蔽、函数重载等复杂情况，需要在AST的构建过程中精确追踪作用域的进入与退出，确保符号表的正确更新和引用解析。

- **注释与注解的处理**：在抽象语法树的构建过程中，如何妥善地处理源代码中的注释与注解是一个创新点。注释和注解虽然不会直接影响语法树的执行逻辑，但它们包含了程序员的意图和附加信息，因此需要在构建AST时被保留，并且能够在后续分析中提供辅助。

3. 代码块作用域支持

- **块级作用域的符号管理：**C++中的作用域包括函数作用域、类作用域和块作用域等。如何在不同作用域之间准确区分和管理符号，并保证符号的生命周期与作用域规则的一致性，是AST构建中的难点。尤其是要处理局部变量遮蔽（如相同名称的变量在不同块中的出现）和全局/局部变量的冲突，确保在语法解析和后续优化中不出现作用域错误。

- **声明与引用的解析：**C++的声明与引用解析非常复杂，尤其是在多重嵌套作用域中，如何有效地解析变量和函数的声明与引用，确保正确的符号绑定，是AST构建中必须精确处理的问题。设计时需要确保在一个作用域内的变量遮蔽不会影响外层作用域的正确引用，同时还要支持复杂的函数参数解析、默认参数值等功能。

通过这些难点的解决与创新，最终的语法分析器和AST构建过程能够为C++代码的后续分析、优化和代码生成奠定坚实的基础，确保整个编译过程的高效性和正确性。

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