使用ANTLR4实现的C++转python翻译器

本项目使用ANTLR4实现了C++转python翻译器。基于上次词法分析器和语法分析器的背景，通过将生成的AST树进行遍历后进行转换，生成与C++语法相配的Python语法。翻译器通过词汇分析、解析和代码生成，将C++程序转换为语义上等价的Python代码，同时维护Python的习惯用法和最佳实践。该系统成功地处理了基本的C++构造，包括控制结构、函数、类和基本数据类型。

一、 团队分工

@苏伟铭：搭建框架 @汪佳宇：基础翻译逻辑 @陈立心：完善翻译逻辑，进行相关测试

@王旭冉：完善翻译逻辑，进行相关测试 文档由大家统一完成。

二、实验内容

1、开发环境: Visual Studio Code， Windows 11

2、实验工具: ANTLR4

3、文件结构:

项目结构如下：

- src/tests-所有的测试用例，及生成的Python代码
- src/translation-从AST树翻译成Python的逻辑实现
- src/CPPLexer.g4 - 定义词法分析规则的ANTLR4语法文件
- src/CPPParser.g4 - 定义语法解析规则的ANTLR4语法文件
- src/main.py - 使用生成的翻译器的Python脚本
- src/cppParserBase.py - 解析器基类实现
- src/CPPParserVisitor.py - 访问者模式实现，用于生成AST
- src/CPPToPythonVisitor.py-访问者模式实现，用于遍历AST树
- src/cpp_to_python_transpiler.py-访问者模式实现，用于生成Python代码

4、难点与创新点

- 维护变量的作用域
    - 实现方法：通过定义外部作用域传递参数，并在函数内使用局部变量维护变量的作用域。
        - 举例：

```
def convert_function(self, function_node: Node, current_vars:
dict[str, str],
                     custom_classes: list[str],
```

```
current_functions: list[str]) -> list[str]:
        """Convert a function definition to Python code"""
        if function_node.node_type != "functionDefinition":
            raise TypeError("Expected functionDefinition node!")
```

- C++中的++与--可以在Python中正确翻译
    - 实现方法：
        1. 假设传入的是一个表示自增或自减运算符的 expression_node，其子节点的 node_type 可能是 'INCREMENT' 或 'DECREMENT'。
        2. 处理：根据子节点的类型，生成对应的 Python 表达式代码：
            - 如果是 'INCREMENT'（++），则在 py_statement 上附加 +1)。
            - 如果是 'DECREMENT'（--），则在 py_statement 上附加 -1)。
        3. 输出：返回更新后的 py_statement，它包含了相应的 Python 表达式部分。
            - 举例：

```
# ++/-- unaryExpression
if expression_node.children[0].node_type == 'INCREMENT':
    py_statement += '+1)'
    return py_statement
elif expression_node.children[0].node_type == 'DECREMENT':
    py_statement += '-1)'
    return py_statement
else:
    raise SyntaxError("invalid unaryExpression node!")
```

- C++中的vector在Python中没有直接等价的部分
    - 实现方法：使用Python的类型模块的通用类型转换
        - 举例：

```
CPP_TO_PYTHON_SCOPES = {
  'std' : {
      'vector' : 'list',
      'string' : 'str'
      }
  }
```

5.特性支持

- 基本C++语言语法
- 函数和类定义
- 基本数据类型和操作

三、测试用例

- **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。
- **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。
- **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。
- **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。

注：所有已生成的Python代码均可直接运行。

四、运行方法

详情请见readme.txt

五、输出展示

举例：test_quickSort

quickSort.cpp

```
#include <iostream>

class QuickSort {
public:
    void sort(int* arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            ...
        }
    }

private:
    int partition(int* arr, int low, int high) {
        int pivot = arr[high];
        ...
        return i + 1;
    }
};

int main() {
    int arr[8];
    int size = 8;
    ...
    std::cout << std::endl;
```

```
    return 0;
}
```

quickSort.py

```
prom typing import List

class QuickSort:
    def sort(self, arr: int, low: int, high: int) -> None:
        if low < high:
            ...
    def partition(self, arr: int, low: int, high: int) -> int:
        pivot = None
        pivot = arr[high]
        ...
        return i + 1
    def __init__(self):
        pass
def main():
    arr = [0] * 8
    size = None
    ...
    return 0

if __name__ == '__main__':
    main()
```

六、参考资料

- ANTLR4文档
- Python文档
- C++文档

附:关于AST和语法解析器等问题详见第一次提交报告