编译原理小组实验

实验报告

田丰源　2017013630
从业臻　2017013599
黄　翔　2017013570

Ⅰ. 实验概述

开发平台：Windows

开发语言：Python

词法语法工具：Lex/Yacc（Python 外部库 ply）

实验内容：实现了从前端语言 C/C++ 到后端语言 Python 的简易编译器

- ✓ 支持 #include / #define 预处理指令
- ✓ 支持全局变量、局部变量、简单作用域特性
- ✓ 支持分支结构（if-else 语句）、循环结构（for 循环 / while 循环）
- ✓ 支持数组（禁用指针形式）和结构体
- ✓ 支持浮点数、科学计数法等
- ✧ 成功实现回文转换、KMP 字符串匹配、四则运算计算这三个标准测例的翻译
- ✧ 给出专门的测例证明作用域、结构体等特性的正确实现

Ⅱ. 实现原理

【运行流程】

程序的运行流程如下图：

【词法分析】

实现　词法分析部分利用了 Python 的 ply.lex 模块。在词法分析器 lex.py 中，通过定义 tokens 与 reserved 列表，即可定义令牌与保留字列表；通过定义正则表达式，即可匹配各类符号。我们参考 **C99 标准文档**，写入了所有的保留字、合乎 C 语法

的标识符以及包括十进制整型数字、十六进制整型数字、浮点数、字符、字符串在内的各类常量值。

功能 词法分析器将完成词法分析工作，包括令牌的识别、空白字符与程序注释的过滤，并在匹配出错时报错。输出 token 流，供语法分析利用。

【语法分析】

实现 语法分析部分利用了 Python 的 ply.yacc 模块。在语法分析器 yacc 中，通过定义推导符号与产生式，即可自底向上地完成语法分析过程。我们参考 **C99 标准文档**，写入了所有的标准文法。同时，我们设计了**抽象语法树类（AST.py）**，在文法产生式匹配成功时调用语法树的构造函数，将产生式右侧作为左侧的子结点。如此，当语法分析完毕时，也即建立了一棵完整的抽象语法树。抽象语法树类的定义如下，语法树中所有的非终结符将对应 ASTInternalNode 类型对象，语法树中所有的终结符将对应 ASTExternalNode 类型对象：

功能 完成语法分析工作，生成完整的抽象语法树，供下一步使用。

【语义分析与中间代码生成】

实现 使用了语法制导的语义处理技术，为每个节点设置了 **code** 和 **flag** 两个属性，**code** 是以这个节点为根的语法分析树对应的后端语言语句，**flag** 是为了处理结构体类型而设置的属性，标志了一个节点是否处于一个结构体的范围之内（如果是，值为结构体的名称）。这两个属性都是综合属性，也就是说，采用“自下而上”的方式传递信息。其中根节点的 **code** 属性直接就是翻译结果，故我们并不中间存储这些属性，而是直接以递归的方式一遍求出翻译结果。Translator 类的 *code_compose* 函数即是 **code** 属性语义规则的计算函数。

由于 Python 和 C/C++关于变量作用域差异较大，我们采取了变量名称预处理的方式实现了 C/C++的部分作用域特性。在处理语法树之前，先遍历语法树，将一些变量的名称进行替换，并插入 global xxx 的语句，具体实现方式详见下文功能说明。

总体的流程：先对语法树进行包括变量重命名的预处理，然后对语法树进行遍历，求得根节点的 **code** 属性（即初步翻译得到的 Python 代码），再进行包括代码风格优化在内的后处理，最后在头尾加上必要的代码，得到可运行的 Python 代码。

功能 由语法树通过预处理、语义处理、后处理，得到可运行的 Python 代码。

Ⅲ. 功能说明

【预处理指令】

编译器支持 C/C++中包括#include 与#define 在内的预处理指令。对于#include 指令，程序将自动检索指定文件，处理后插入相应位置；对于#define 指令，程序将利用正则表达式自动进行宏替换。

【全局变量/局部变量】

编译器能够区分 C/C++中变量的作用域区别，支持全局变量与局部变量。首先对于所有声明，将声明的变量加上“_0”的后缀，计入一个变量表。之后对于每个函数，碰到变量就判断变量表中是否出现，再根据这个变量是第一次声明还是之前声明过，进行相应的变量表维护/变量重命名操作（重命名就是根据嵌套作用域的数量改变后缀数字）。在进入函数/进入循环/进入选择的时候，都深拷贝变量表备份，离开时再还原变量表，以达到简单的作用域机制。后缀的机制保证了重命名后的变量不会与其他变量冲突。

【分支结构/循环结构】

编译器支持 C/C++分支结构 if-else 语句与循环结构 for 语句、while 语句。分支和循环机构都可以视为 while/for 等关键词、左右大括号和 statement 的组合，翻译成 Python 语句是对这些子模块的重新取舍组合。

【函数】

编译器支持 C/C++函数的定义，并能自动识别出 main 函数，将之作为程序入口点。由于 C/C++中函数的定义在一个文件中是同级的，故可以先对整棵语法树分拣出所有的函数定义和所有的变量及函数声明，然后分别处理这两个森林里的树。只声明不定义函数的语句会先被过滤掉。之后，只需要将每个 C/C++函数定义翻译成 Python 函数，随意顺序放于所有全局变量之后即可。

【数组】

编译器支持 C/C++数组（一维）的定义。由于 Python 无论如何也无法完全模拟 C/C++中的指针，我们人为规定数组的定义和使用不涉及指针形式。我们采用 Python 列表模拟数组。具体来说，数组的下标运算与列表基本相同；声明则使用列表

名称 = [None] * 长度的方式；初始化则额外在声明语句后追加赋值语句。我们支持形如 char a[10]="abs"; 和 int a[10]={1, 2, 3}; 的初始化方式。

【结构体】

编译器支持 C/C++结构体的定义。结构体比较棘手，因为和数组等在语法上有很大不同。使用 Python 的类来模拟结构体。在声明结构体变量时，struct X y; 翻译为 Python 应为 y = X() 。声明结构体数组时，列表名称 = [None] * 长度的方式是错误的，因为每个结构体的对象就是同一个了；因此使用列表名称 = [结构体名称() for i in range(长度)] 的形式。结构体成员变量的赋值与使用上 C 与 Python 语法基本相同。此外还需注意预处理变量名称时跳过结构体成员变量。详见测例与代码注释。

【缩进】

Python 的缩进与其他语言不同，处理时需要格外留心。我们使用了一种有趣的方法，将每个节点的 **code** 属性以嵌套列表的形式存储，每一层嵌套就代表一层缩进。比如，一个函数的语法树的根节点 **code** 属性可以是：["def foo:", ["pass"]]。这种方法可以很好地兼容自下而上的语义处理，不用产生额外开销。

【C 标准库支持】

通过在翻译得到的文件开头加入一些与 C 标准库函数同名的事先编写的 Python 函数代码，可以模拟一些 C 标准库函数，如 *printf, gets, system, strlen* 等。

Ⅳ. 难点与创新点

总体来说，我们有如下难点与创新点：

- ✧ 通过预处理重命名变量的方式，简单实现了部分的 C/C++作用域机制；
- ✧ 通过嵌套列表表示的方式，解决了 C/C++翻译成 Python 棘手的缩进问题；
- ✧ 实现了 C/C++的结构体特性以及部分数组特性；
- ✧ 配备有丰富的预处理、后处理，简单支持#define、#include，并保证输出代码具有良好的风格。

相关原理已在上文功能说明部分展开说明，不再赘述。

Ⅴ. 分工

田丰源负责语义处理和测例编写，从业臻负责语义处理和测例编写，黄翔负责词法分析和语法分析。