编译设计文档

编译器介绍

整个编译器分为四个部分:词法分析,文法分析,中间代码生成和目标代码生成。每个部分之间内部实现相互独立，只要规定了接口即可。这样保证了哪怕更换了目标程序或者源程序，整个编译器也不需要大概，只需要更改前端或者后端即可。

编译器总体设计

作为一个 cpp 苦手，选择了相对熟悉的 java，反正实现应该大差不差吧。

对于编译器的每一个部分都有一个文件夹进行打包，保证代码结构的清晰。

编译器入口为 Compile.java 文件，该文件只负责更改输入输出，调用各个 package 即可。

词法分析对应的 package 为 Lexer，该 package 负责读入代码，转换成一个一个的 word 以及对应的 TokenType。

语法分析对应的 package 为 Parser 和 Node， 负责读入 Lexer 的分析结果，根据 Node 里面的类搭建一个语法树。

词法分析设计

词法分析部分比较简单，读入整个文件，然后遍历每个单词获取word。

首先设计一个 Source 对象，该对象负责读入文件，然后对 Lexer 提供字母。该对象提供往前查看多个字符的接口。因此一个变量记录当前读入的位置，然后向前或者向后移动即可。 Java 的 ArrayList 真香。

Lexer 类则负责获取一个一个字符，放入 switch 块分类讨论。 初次之外，先初始化一个 word2Token 的 HashMap，并放入所有的关键字，这样在获取词法分析中，碰到了一个 identToken，不需要辨别是否是关键字，直接查找是否在 word2Token 中有这个 key，没有就是 identToken，有那就是 key 对应的 value。

另外定义一个枚举类 Token，用于存放词法分析阶段用到的 Token。

语法分析设计

语法分析设计比较简单,主题思路就是递归下降,每次从词法分析的结果中读入一个单词,然后根据文法递归即可。

首先是根据已有的文法定义不同的树结点，所有结点统一继承 Node 类。

然后是 Parser 类的编写，此类需要获得 word，以及构建一个语法树。

在构建方面，我参考了 Rust 的 Rowan 包写法，创建了两个栈：parent，children，调用 parse 函数时，往 parent 压栈，记录当前 children 的个数。离开 parse 函数的时候弹栈，此时多的 children 栈里面的元素则都为该 parent 的孩子。

words 则类似 source，提供往后看多个 word 的方法。

parse 类主要就是递归下降。但是在编写过程中，发现左递归,候选式无法决定等问题，下面是解决方式：

- 候选式无法决定问题：主要是 lVal 和 Ident 无法分辨，一开始选择改文法，但是考虑到要输出分终结符的问题，选择了往后面 peek，寻找特征 token。
- 左递归问题：这个必须要改文法了，但是为了非终结符输出不改，需要对 parent 栈进行操作，入 addExp 为例，当读到第二个 mulExp 的时候，就必须往 parent 栈中压入 addExp，其对应的 children point 和初始 addExp 相同，然后立马将其弹出栈，以实现对上一个 mulExp 的回收。然后再 parse 下一个mulExp。