《编译技术》课程设计文档

姓名：黄雨石

学号：20376156

词法分析

编码前设计

首先定义`Token`类、`Type`枚举类、`Lexer`类，`Token`用于记录每个单词的类型、值、所在行数，`Type`则是由类别码构成的枚举类、`Lexer`提供词法分析以及存储结果的功能

在`Lexer`类中按字符读取从文件中读入的源码，由`if-else`分支语句进行类别判断，然后进入对应分支读取完整单词，得到一个`Token`实例，不断重复上述操作直到读到源码末尾

- 对于保留字采用`HashMap`进行查寻，保留字为键，类别码为值，当得到一个保留字单词就查询`HashMap`获得类别码
- 对于注释空白符跳过不识别，即直接读到注释之后的第一个字符
- 对于行号记录，读到`\n`字符行号就加一

本次不需要考虑错误处理的问题

编码后修改

与编码前基本一致，但是考虑到评测系统为`Linux`系统于是对于要跳过的空白符需要注意判断`\r`的情况

语法分析

编码前设计

封装`Parser`类，将`Lexer`类解析得到的所有`Token`以数组形式一次性传入`Parser`进行递归下降解析同时维护一个指向`Token`数组中元素的指针，然后使用自己封装的`peek()`、`retract(int step)`函数在递归下降过程中取出当前指针指向的元素或者将指针回溯。对于每一个非终结符建立相应的类作为递归下降树的非叶子结点，每一个叶子节点对应的为一个代表终结符的`Token`

其中我对于文法的具体分析处理主要如下：

- **表达式**：对于部分与`*Exp`相关的推导规则均存在左递归现象，需要改写文法如下：

```
改写前
-------
改写后
```

```
    MulExp -> UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
    ------------------------------------------------------
    MulExp -> UnaryExp { ('*' | '/' | '%') UnaryExp }

    AddExp -> MulExp | AddExp ('+' | '-') MulExp
    ---------------------------------------------
    AddExp -> MulExp { ('+' | '-') MulExp }

    RelExp -> AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
    ----------------------------------------------------------
    RelExp -> AddExp { ('<' | '>' | '<=' | '>=') AddExp }

    EqExp -> RelExp | EqExp ('==' | '!=') RelExp
    ---------------------------------------------
    EqExp -> RelExp { ('==' | '!=') RelExp }

    LAndExp -> EqExp | LAndExp '&&' EqExp
    -------------------------------------
    LAndExp -> EqExp { '&&' EqExp }

    LOrExp -> LAndExp | LOrExp '||' LAndExp
    ---------------------------------------
    LOrExp -> LAndExp { '||' LAndExp }
```

同时注意到以上列出的 `*Exp` 中均为 `*Exp -> T {operator T}`（其中 `T` 可以认为是低一级的 `*Exp`）的形式，于是可以建立一个基类 `MultiExp` 去统一这种形式，再使 `*Exp` 去继承它，`MultiExp` 接口如下

```java
public class MultiExp<T> {
    private final String name;
    private final T first;
    private final ArrayList<Token> operators = new ArrayList<>();
    private final ArrayList<T> Ts = new ArrayList<>();
    // ...
}
```

于是可以通过上述方式将需要改写文法的 `*Exp` 放置于 `Multi` 软件包内，而其它诸如 `UnaryExp`、`PrimaryExp`、`Number`、`LVal`、`FuncRParams` 这种更低一级的表达式放置于 `Unary` 软件包中

- **语句**：由于推导规则中左侧为 `Stmt` 的规则非常复杂于是我们对于该规则右侧的每一个分支建立 `*Stmt` 类，即改写文法规则新增加 `*Stmt` 非终结符，具体如下：

```
    <AssignStmt> -> LVal '=' Exp
    <ExpStmt> -> Exp
    <LoopStmt> -> 'break' | 'continue'
    <ReturnStmt> -> 'return' [Exp]
    <InputStmt> -> <LVal> '=' 'getint' '(' ')'
    <OutputStmt> ->  'printf''('FormatString{','Exp}')'
-------------------------------------------------------------------------
-------
    <SimpleStmt> -> [ <AssignStmt> | <ExpStmt> | <LoopStmt> |
<ReturnStmt> |
                     <InputStmt> | <OutputStmt> ] ';'
    <IfStmt> -> 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
    <WhileStmt> -> 'while' '(' Cond ')' Stmt
```

```
    <BlockStmt> -> Block
-------------------------------------------------------------------------
--------
    <Stmt> -> <SimpleStmt> | <IfStmt> | <WhileStmt> | <BlockStmt>
```

可以按照带分号与不带分号分为`SimpleStmt`以及`IfStmt`、`WhileStmt`、`BlockStmt`，由于`SimpleStmt`推导规则右侧涉及到较多的新建非终结符，因此可以建立`Simple`软件包统一存储。同时为了方便在`SimpleStmt`中保存，可以创建一个`Simple`接口，由`AssignStmt`、`ExpStmt`...共同实现

- **函数**：注意到其实`MainFuncDef`与`FuncDef`非常相似，于是可以将`MainFuncDef`视作`FuncDef`的特例去继承`FuncDef`类
- **变量声明**：由于不需要输出`Decl`可以直接将`ConstDecl`、`VarDecl`都视为`Decl`用一个布尔变量标识即可

对于上述分析已经我们已经可以消除所有的左递归，但是实际过程中考虑到有部分非终结符的`FIRST`是相同的，于是采取超前扫描的方法，同时定义了`pre`、`nxt`分别表示当前遍历的的数组中的`Token`的前一个和后一个以减少指针回溯，除了在处理`SimpleStmt`的过程中指针可能需要回溯一整个`LVal`的大小，其它时候至多回溯一步即可

对于`SimpleStmt`中的`AssignStmt`、`InputStmt`、`ExpStmt`可能它们的`FIRST`均为`LVal`这时侯需要先记录指针的值然后超前解析一个`LVal`判断其后的`Token`类型即可判断需要进入的分支，然后回溯到之前记录的位置进入对应分支的解析函数再次开始解析

编码后修改

- 编码前并未考虑如何输出的问题，编码时采取对于每个节点重写它的`toString`方法，最后直接调用顶层`CompUnit`的`toString`方法即可
- 由于文法中很多地方都会使用`'[' [*Exp] ']'`这种方式来表示数组变量的维度以及下标，因此，我单独建立了一个`Index`类存储这样的结构，同时完成该类的`toString`方法，具体接口如下：

```java
public class Index {
    private Token lBTk;
    private Token rBTk;
    private Exp exp;
    // ...
}
```

- 编码前并未对于之后的错误处理预留接口，但是在递归下降的编码过程中可以通过判断当前`Token`非常自然的找到相关的错误并预留错误处理的空间