《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

错误处理与符号表建立

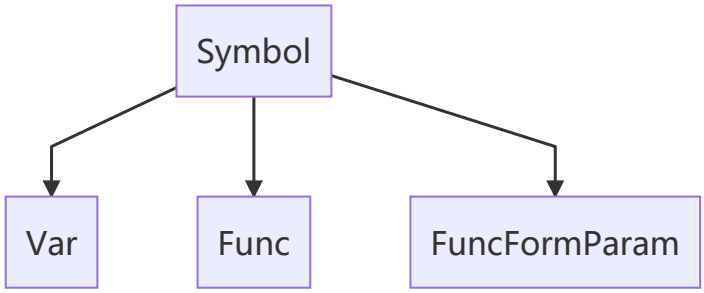
编码前设计

- **符号表**：采用树形符号表，遵循一块一表，每次进入一个块时就建立一张符号表，表中存储新建的变量符号、函数符号、函数形参符号（当前为函数块），同时函数符号中存储该函数符号对应的函数块符号表以便查询函数中的变量以及形参

  其中符号一共分为三类，均实现`Symbol`接口：

  - `Var`：所有`int`和`const int`变量

- `Func`：所有函数定义
- `FuncFormParam`：函数形参



- **错误处理表**：建立一个`Error`类用于标识特定错误，将错误类型作为`Error`中的一个枚举类，再封装`ErrorTable`类，提供对于`Error`根据错误位置排序、打印、保存的功能
- **遍历语法树**：
  - **符号表建立**：首先维护一个当前符号表`curTable`，初始时即为全局符号表，之后对已经建立好的语法树进行自顶向下遍历，期间算法（不考虑错误处理）

    > - 若遍历到`Block`或者`Func`，那么新建一个符号表，将`curTable`设置为新建的符号表，注意如果是`Func`需要将`Func`对应符号加入上一级符号表同时在其中记录新建的符号表引用`curTable`
    > - 遍历到`Decl`项或者`FuncFormParm`，则将其加入当前符号表
    > - 当从`Block`或者`Func`的遍历返回时将当前符号表设置为其父节点

  - **错误处理**：按错误类型分别讨论：
    - `a`：直接在词法分析中对解析到的`FormatString`判断即可
    - `i`、`j`、`k`：均属于同类错误，在语法分析递归下降的过程中判别，同时注意递归下降过程中不要偷懒，识别每个语法单元时要根据其完整的`First`来，否则可能出错
    - `b`、`c`：对于这两类错误可以通过在遍历语法树时根据`curTable`在该节点中查询是否存在名字或者在该节点以及其祖先节点中查询是否存在该名字，注意对于重复定义的函数其内部也要解析，对于函数形参其作用域为函数体本身，对于局部变量名可以和函数名相同
    - `d`、`e`：在解析函数实参时通过`curTable`找到该函数的定义，然后根据其中保存的函数的符号表中存储的函数的形参定义判断
    - `f`、`g`：除了维护一个`curTable`再维护一个当前的`curFunc`（`Symbol`），解析到`return`语句时就可以根据`curFunc`中存储的返回值类型对于`f`判断，`g`则是在解析`funcDef`的最后取出其`Block`中的最后一条语句判断是否为`return`
    - `h`：直接查找当前节点以及其祖先节点判断变量是否为`Const`类型
    - `l`：解析到`print`语句时直接遍历查找`%d`个数判断即可
    - `m`：维护一个全局变量`cycLevel`，进入循环块加1，出循环块减1，当遇到`break`或者`continue`语句时判断`cycLevel`是否大于0即可

编码后修改

并未将左值与其定义链接起来同时也并未做常量传播，因此对于错误`e`的判断比预想要难，对于函数调用的实参需要单独重新遍历每一个实参即`Exp`的第一个`PrimaryExp`（不需要检查`Exp`内中参与表达式计算的每一个算子`LVal`、`Number`的类型间是否匹配，所以相当于只用看`Exp`的第一个`PrimaryExp`的维数即可并不需要遍历完），然后查询并计算其维数再在符号表看实参与形参维数是否可以匹配，最复杂的还要在遍历的过程中判断除了维数问题还有没有未定义的问题，此时需要遍历所有的`PrimaryExp`（按我的编码结构是先判断未定义问题再检测类型是否匹配，并且无法在检查类型是否匹配时判断是否已经检测出过未定义的问题）若有就返回，以免对于一行重复报错（`c`、`e`）。预计后续中间代码生成时会继续修改符号表定义