设计文档

使用方式

将testfile放置于根目录下，通过设置flag控制不同阶段output的输出内容。

整体架构

```
public class Compiler {
    public static void main(String[] args) throws IOException {
        File file = new File("testfile.txt");
        if (!file.exists()) {
            System.out.println("Filename error");
            System.exit(0);
        }
        String content = TurnToFile.readFile(file);

        //词法
        LexicalAnalyzer lexicalAnalyzer = new LexicalAnalyzer(content);
        ArrayList<LexicalAnalyzerForm> lexicalAnalyzerForms =
lexicalAnalyzer.LexicalAnalyze();
        TurnToFile.LexicalToFile(false, lexicalAnalyzerForms, "output.txt");
        //语法
        GrammerAnalyzer grammerAnalyzer = new GrammerAnalyzer(lexicalAnalyzerForms);
        TurnToFile.GrammerToFile(false,grammerAnalyzer.grammerAnalyze(),"output.txt" );
        //错误处理

    }
}
```

第一遍扫描testfile时将生成词法分析文件，并作为语法分析文件的输入进行相关操作，在此基础上需要添加错误处理和符号表建立的相关过程，需要后期进行完善。

设计思路

IO工具

```
package IOTool;
public class TurnToFile {
    public static String readFile(File file) {}

    public static void LexicalToFile(boolean flag, ArrayList<LexicalAnalyzerForm> res,
String outFileName){}

    public static void GrammerToFile(boolean flag, ArrayList<String> res, String
outFileName){}

    public static void ErrorToFile(boolean flag, ArrayList<String> res, String
outFilename){}
```

将所有文件输入输出相关的函数打包为IOTools方便日后进行修改更新，避免与主要代码混淆造成不必要的错误。

词法分析

LexicalForm

```
package Lexical;
public class LexicalAnalyzerForm {
    private String value;//词
    private String CategoryCode;//词类

    public String getCategoryCode() { return CategoryCode; }

    public LexicalAnalyzerForm(String value, int kind){}//首次分类 分为关键字 符号 数字串 格
式字符串

    public void LexicalIden(String value){}//二次分类 进行关键字的category赋值

    public void LexicalSymbol(String value){}//二次分类 进行符号的category赋值
}
```

将词法分析中每一个识别到的单词视作一个单独的对象，并对其赋予词法属性，方便语法分析阶段的调用，后续可直接通过这个类直接添加想要的词法属性，方便后续进行错误处理代码生成的相关操作。

LexicalJudge

```
package Lexical;

public class LexicalJudge {
    public boolean judgeNum(char chara) {}

    public boolean judgeWordThen(char chara) {}

    public boolean judgeWordFirst(char chara) {}
```

```
    public boolean judgeSingleSym(char chara) {}

    public boolean judgeDoubleSym(char chara) {}

    public boolean judgeFormatChar(char chara) {}
}
```

根据文法定义，在判断需要对每个词的构造方式进行判断，从而能够判断其具体的category属性值，特别地对于关键字而言，文法定义中有较为严格的要求，故需要对首字符和接下来的字符都进行判断来斟酌其合法性和正确性，合法性可方便后续错误处理任务的进行。

LexicalAnalyzer

```
package Lexical;
public class LexicalAnalyzer {
    public String content;

    public LexicalAnalyzer(String content) {}//content赋值

    public String getContent() {}

    public ArrayList<LexicalAnalyzerForm> LexicalAnalyze() {
      if (lexicalJudge.judgeNum(chara)) {}
      else if (lexicalJudge.judgeWordFirst(chara)) {}
      else if (lexicalJudge.judgeSingleSym(chara)) {}
      else if (chara == '\"') {}
      else if (chara == '\n') {}
      else if (chara == '/') {}
    }//词法分析程序
}
```

进行具体的词法分析，规则为读取第一个字或者加上下一个字来判断其主要类别，对其进行分类，并对每一个可能的词法中建立一个buffer，如果符合预期则将buffer内容存入content。为了方便debug和后续处理，在每一个if的else情况下（即非正常情况）都会打印前一个和后一个字符的内容，并输出error位置。

语法分析

词法分析阶段已经获得了一个标准的词法分析输出文件，所以在语法分析阶段只需要使用递归下降的思想，并利用词法分析输出文件判断单词的FIRST集合即可。出于简洁考虑，下述代码只表征了语法分析阶段不同类之间的调用关系。

由于个人感觉ctrl7多个class看的更清楚，也更方便互相调用防止死锁，所以最终没有采用分文件的构造方式，而是将整个语法分析都放在了一个大类中

GrammerAnalyer

```
package Grammer;
public class GrammerAnalyzer {
    private ArrayList<LexicalAnalyzerForm> GrammerAnalyzerOutput;
    private int index = 0;

    public GrammerAnalyzer(ArrayList<LexicalAnalyzerForm> LexicalAnalyzerOutput) {
        this.GrammerAnalyzerOutput = LexicalAnalyzerOutput;
    }

    public ArrayList<String> grammerAnalyze() {
        index = 0;
        return new ArrayList<>(CompUnit());
    }

    //编译单元
    //CompUnit → {Decl} {FuncDef} MainFuncDef
    public ArrayList<String> CompUnit() {
        ArrayList<String> res = new ArrayList<>();
        // 全局变量的声明
        Decl();
        // 函数声明
        FuncDef();
        //主函数
        MainFuncDef();
        res.add("<CompUnit>");
    }

    //声明 Decl不输出
    //Decl → ConstDecl | VarDecl
    public ArrayList<String> Decl() {
        ConstDecl();
        VarDecl();
    }

    //常量声明
    //ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
    public ArrayList<String> ConstDecl(){
        ConstDef();
        res.add("<ConstDecl>");
    }

    //常数定义
    //ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal
    public ArrayList<String> ConstDef(){
        ConstExp();
        ConstInitVal();
        res.add("<ConstDef>");
```

```
    }

    //常量初值
    //ConstInitVal → ConstExp | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
    public ArrayList<String> ConstInitVal(){
        ConstExp();
        res.add("<ConstInitVal>");
        ConstInitVal();
        ConstInitVal();//通过读取comma得到是否还有后续定义
        res.add("<ConstInitVal>");
    }


    //变量声明
    //VarDecl → BType VarDef { ',' VarDef } ';'
    public ArrayList<String> VarDecl() {
        ArrayList<String> res = new ArrayList<>();
        VarDef();
        VarDef();//通过读取comma得到是否还有后续定义
        res.add("<VarDecl>");

    }

    //变量定义
    //VarDef → Ident { '[' ConstExp ']' } |  Ident { '[' ConstExp ']' } '=' InitVal
    public ArrayList<String> VarDef() {
        ConstExp();
        InitVal();
        res.add("<VarDef>");
    }

    //变量初值
    //InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'
    public ArrayList<String> InitVal() {
        InitVal();
        InitVal();//通过读取comma得到是否还有后续定义
        res.add("<InitVal>");
        Exp();
        res.add("<InitVal>");
    }

    //函数定义
    //FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
    public ArrayList<String> FuncDef() {
        FuncType();
        FuncFParams();
        Block();
        res.add("<FuncDef>");
    }
```

```
//主函数定义
//MainFuncDef → 'int' 'main' '(' ')' Block
public ArrayList<String> MainFuncDef() {
    Block();
    res.add("<MainFuncDef>");
}

//函数类型
//FuncType → 'void' | 'int'
public ArrayList<String> FuncType() {
    res.add("<FuncType>");
}

//函数形参表
//FuncFParams → FuncFParam { ',' FuncFParam }
public ArrayList<String> FuncFParams() {
    FuncFParam();
    FuncFParam();//通过读取comma得到是否还有后续定义
    res.add("<FuncFParams>");
}

//函数形参
//FuncFParam → BType Ident ['[' ']' { '[' ConstExp ']' }]
public ArrayList<String> FuncFParam() {
   ConstExp();
   res.add("<FuncFParam>");
}

//语法块
//Block → '{' { BlockItem } '}'
public ArrayList<String> Block(){
    BlockItem();
    res.add("<Block>");
}

//语句块项
//BlockItem → Decl | Stmt
public ArrayList<String> BlockItem() {
    switch (GrammerAnalyzerOutput.get(index).getCategoryCode()) {
        case "CONSTTK":
            ConstDecl();
            break;
        case "INTTK":
            VarDecl();
            break;
        case "IDENFR":
        case "LBRACE":
        case "IFTK":
```

```
            case "WHILETK":
            case "BREAKTK":
            case "CONTINUETK":
            case "RETURNTK":
            case "PRINTFTK":
            case "LPARENT":
            case "INTCON":
            case "NOT":
            case "PLUS":
            case "MINU":
            case "SEMICN":
                Stmt();
                break;
        }
    }

    //语句
    /*
    Stmt → LVal '=' Exp ';' // 每种类型的语句都要覆盖
     | [Exp] ';' //有无Exp两种情况
     | Block
     | 'if' '( Cond ')' Stmt [ 'else' Stmt ] // 1.有else 2.无else
     | 'while' '(' Cond ')' Stmt
     | 'break' ';'
     | 'continue' ';'
     | 'return' [Exp] ';' // 1.有Exp 2.无Exp
     | LVal = 'getint''('')'';'
     | 'printf''('FormatString{,Exp}')'';'
     */
    public ArrayList<String> Stmt() {
        switch (GrammerAnalyzerOutput.get(index).getCategoryCode()) {
            //| Block
            case "LBRACE":
                Block();
                res.add("<Stmt>");
                break;
            //| 'if' '( Cond ')' Stmt [ 'else' Stmt ] // 1.有else 2.无else
            case "IFTK":
                Cond();
                Stmt();
                Stmt();
                res.add("<Stmt>");
                break;
            //| 'while' '(' Cond ')' Stmt
            case "WHILETK":
                Cond();
                Stmt();
                res.add("<Stmt>");
                break;
```

```
            //| 'break' ';'
            case "BREAKTK":
            //| 'continue' ';'
            case "CONTINUETK":
                res.add("<Stmt>");
                break;
            //| 'return' [Exp] ';' // 1.有Exp 2.无Exp
            case "RETURNTK":
                Exp();
                res.add("<Stmt>");
                break;
            //| 'printf''('FormatString{,Exp}')'';'
            case "PRINTFTK":
                Exp();
                res.add("<Stmt>");
                break;
            // |;
            case "SEMICN":
                res.add("<Stmt>");
                break;
            //| LVal '=' Exp ';'
            //| LVal = 'getint''('')'';'
            case "IDENFR"://这一阶段会不可避免的出现回溯，故需要特殊处理
                Exp();
                res.add("<Stmt>");
                Lval();
                //| LVal = 'getint''('')'';'
                //| LVal '=' Exp ';'
                Exp();
                res.add("<Stmt>");
                Exp();
                res.add("<Stmt>");
                break;
            //[Exp] ';'
            case "LPARENT":
            case "INTCON":
            case "NOT":
            case "PLUS":
            case "MINU":
                Exp();
                res.add("<Stmt>");
                break;
        }
    }

    //表达式
    //Exp → AddExp
    public ArrayList<String> Exp() {
        AddExp();
```

```
        res.add("<Exp>");
    }

    //条件表达式
    //Cond → LOrExp
    public ArrayList<String> Cond() {
        LOrExp();
        res.add("<Cond>");
    }

    //左值表达式
    //LVal → Ident {'[' Exp ']'}
    public ArrayList<String> LVal() {
        Exp();
        res.add("<LVal>");
    }

    //基本表达式
    //PrimaryExp → '(' Exp ')' | LVal | Number
    public ArrayList<String> PrimaryExp() {
        switch (GrammerAnalyzerOutput.get(index).getCategoryCode()) {
            case "LPARENT":
                Exp();
                res.add("<PrimaryExp>");
                break;
            case "IDENFR":
                LVal();
                res.add("<PrimaryExp>");
                break;
            case "INTCON":
                res.add("<Number>");
                res.add("<PrimaryExp>");
                break;
        }
    }

    //数值
    //Number → IntConst
    public ArrayList<String> Number() {
        res.add("<Number>");
    }

    //一元表达式
    //UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
    //注意先识别调用函数的Ident '(' [FuncRParams] ')', 再识别基本表达式PrimaryExp
    public ArrayList<String> UnaryExp() {
        res.add("<UnaryOp>");
        UnaryExp();
        res.add("<UnaryExp>");
```

```
        FuncRParams();
        res.add("<UnaryExp>");
        PrimaryExp();
        res.add("<UnaryExp>");
    }

    //单目运算符
    //UnaryOp → '+' | '−' | '!'
    public ArrayList<String> UnaryOp() {
        res.add("<UnaryOp>");
    }

    //函数实参表
    //FuncRParams → Exp { ',' Exp }
    public ArrayList<String> FuncRParams() {
        Exp();
        Exp();//通过读取comma得到是否还有后续定义
        res.add("<FuncRParams>");
    }

    //乘除模表达式
    //MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
    //MulExp → UnaryExp { ('*' | '/' | '%') UnaryExp }
    public ArrayList<String> MulExp(){
        res.add("<MulExp>");
        UnaryExp();
        res.add("<MulExp>");
    }

    //加减表达式
    //AddExp → MulExp | AddExp ('+' | '−') MulExp
    //AddExp → MulExp { ('+' | '−') MulExp }
    public ArrayList<String> AddExp(){
        res.add("<AddExp>");
        MulExp();
        res.add("<AddExp>");
    }

    //关系表达式
    //RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
    //RelExp → AddExp { ('<' | '>' | '<=' | '>=') AddExp }
    public ArrayList<String> RelExp() {
        AddExp();
        res.add("<RelExp>");
        AddExp();//通过读取comma得到是否还有后续定义
        res.add("<RelExp>");
    }

    //相等性表达式
```

```
    //EqExp → RelExp | EqExp ('==' | '!=') RelExp
    //EqExp → RelExp { ('==' | '!=') RelExp }
    public ArrayList<String> EqExp() {
        RelExp();
        res.add("<EqExp>");
        RelExp();
        res.add("<EqExp>");
    }

    //逻辑与表达式
    //LAndExp → EqExp | LAndExp '&&' EqExp
    //LAndExp → EqExp { '&&' EqExp }
    public ArrayList<String> LAndExp() {
        EqExp();
        res.add("<LAndExp>");
    }

    //逻辑或表达式
    //LOrExp → LAndExp | LOrExp '||' LAndExp
    //LOrExp → LAndExp { '||' LAndExp }
    public ArrayList<String> LOrExp() {
        LAndExp();
        res.add("<LOrExp>");
    }

    //常量表达式
    //ConstExp → AddExp
    public ArrayList<String> ConstExp(){
        AddExp();
        res.add("<ConstExp>");
    }
}
```