

武汉大学计算机学院

本科生实验报告

CMM 语言总体架构的设计和搭建

专 业 名 称 ： 软件工程

课 程 名 称 ： 解释器构造与实践

指 导 教 师 一： 李蓉蓉

指 导 教 师 二：

学 生 学 号 ： 2016302580188

学 生 姓 名 ： 朱江源

二〇一八年十月

郑 重 声 明

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名： 朱江源

日期： 12.26

摘要

CMM 语言词法分析实验的实验目的是设计 CMM 语言解释器的总体架构, 设计并编制调试一个分析单词的词法分析器, 加深对词法分析原理的理解。

实验内容主要包括:

- a) 对词法现象的形式化描述, 附上所定义的种别码表。
- b) 程序结构说明, 包括程序的总体结构, 以及主要的数据结构、算法说明。
- c) 设计充分的测试数据, 给出调试的数据及结果。

实验结论为总结本次实验内容, 对所完成的结果进行分析评价。(包括在设计、实现中遇到的问题、解决的方法, 实验中用到的独到的方法和见解等等。评述设计与实现的优缺点, 存在的问题等)

关键词: 词法分析 解释器总体框架 CMM

目录

1 实验目的和意义	5
1.1 实验目的	5
1.2 实验要求	5
2 实验设计	6
2.1 总体框架构建	6
2.2 词法分析器的构建	8
2.3 语法分析器的构建	17
2.4 语义分析器的构建	28
2.5 Javacc 的使用	50
3 结论	64
4. 参考文献	66

1 实验目的和意义

1.1 实验目的

- 设计 CMM 语言解释器的总体架构；
- 设计并编制调试一个分析单词的词法分析器，加深对词法分析原理的理解。

1.2 实验要求

词法分析器的功能要求小结如下：

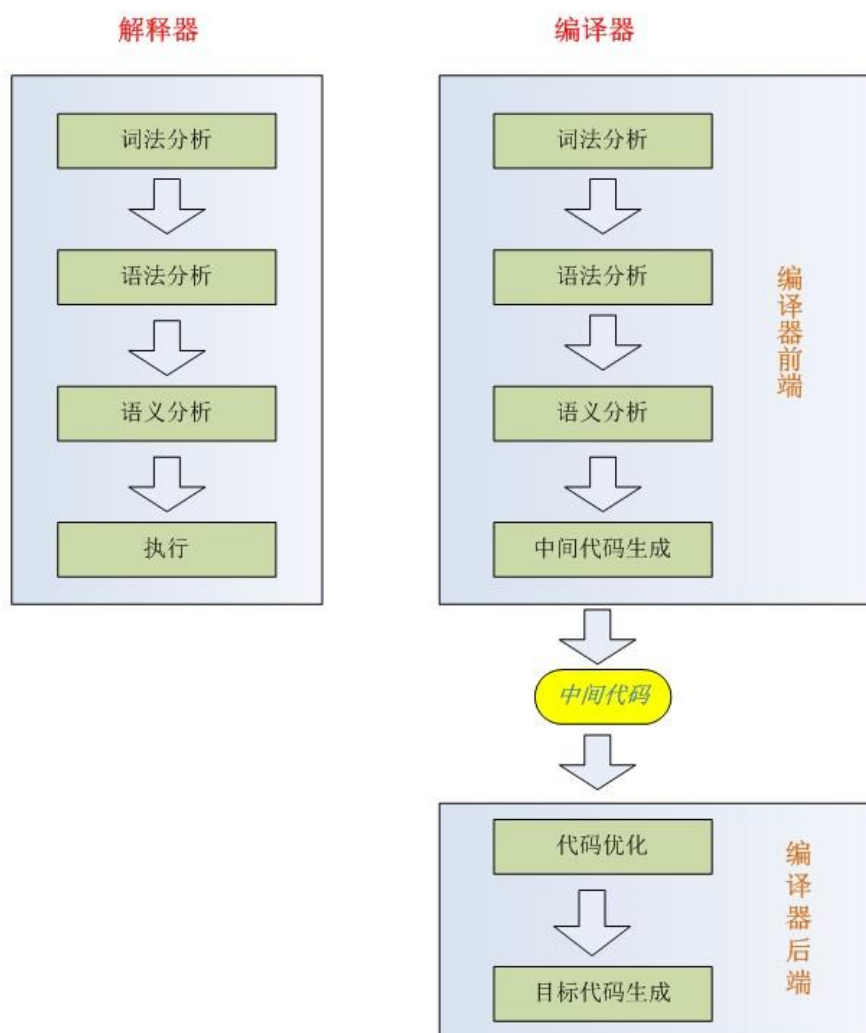
- a) 忽略空格、tab 键、回车换行等分隔符；
- b) 识别不同类型的记号；
- c) 识别并忽略注释；
- d) 记录下每个记号的行号或位置； e) 将识别的记号输出；
- f) 如果输入串存在词法分析错误，则报错。

输入形式为文件输入。

输出的结果需要呈现出词法分析器对输入内容分析的结果

2 实验设计

2.1 总体框架构建



(图片摘自武汉大学计算机学院李蓉蓉老师提供的 WORD 文档)

解释器和编译器也是如此，读入源语言后，解释器和编译器都要进行词法分析、语法分析和语义分析，之后，二者开始有所分别。解释器在语义分析后选择了直接执行语句；编译器在语义分析后选择将语义存储成某一种中间语言，之后通过不同的后端翻译成不同的机器语言（可执行程序）。

这门课我们的最终目的就是做出一个完整的能够对 CMM 语言源文件进行语法分析 语法分析 语义分析并且给出执行结果的这么一个工具。

实验总的来开分为三个部分词法分析 语法分析 语义分析。

第一阶段是词法分析

词法分析的目的在于对源码文件进行读取，并且解析出其中的单词的信息，

比如是标识符，保留字，操作符还是其他等等信息，得到的 tokenList 供给语法分析，不符合既定的文法规则

词法分析部分的输入是源码文件

词法分析部分的输出是一个 tokenList 里面含有源码分析得到的所有 token

第二阶段是语法分析

语法分析的目的在于针对之前词法分析之后的 tokenList 对照着既定文法进行语法检查，检查源码是否符合文法规则，并且将之转换为一颗语法树供给语义分析使用。我选用了 LL 递归下降子程序法来进行分析

语法分析部分的输入是语法分析得到的 tokenList

语法分析部分的输出是一颗链表存储连接的语法树 LinkedList<TreeNode>

第三阶段是语义分析

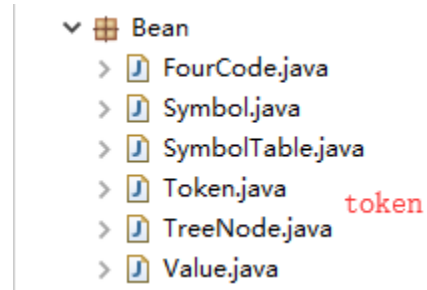
语义分析的目的在于对语法分析得到的语法树进行语义检查，查看是否出现语义错误并且给出程序的执行结果。在此过程中我打算采用生成中间代码之后根据中间代码来运行给出结果。

语义分析部分的输入是词法分析得到的语法树

语义分析部分的输出是执行结果

2.2 词法分析器的构建

2.2.1 数据体构建



构建数据实体 token 用于之后存储一个个读取到的单词

Token 有这些属性

```
private int type;  
  
private String value;  
//行号  
private int lineNum;  
//列号  
private int columnNum;
```

Type 用于存储这个 token 的类型 具体映射在 token.java 文件里面，可以随时实现增删查改。


```

//保留字
// if
public static final int IF = 1;
//else
public static final int ELSE = 2;
//do
public static final int DO = 3;
// while
public static final int WHILE = 4;
//read
public static final int READ = 5;
//write
public static final int WRITE = 6;
// int
public static final int INT = 7;
//double
public static final int DOUBLE = 8;

//单目操作符
//+
public static final int PLUS = 9;
//-
public static final int MINUS = 10;
//*
public static final int MUL = 11;
// /
public static final int DIV = 12;
// =
public static final int ASSIGN = 13;

//双目操作符
// >
public static final int GT = 14;

```

2.2.2 用于判断字符的 JudgeFunction

所有需要 judege 的 token

```

//保留字
public static String[] RESERVEDWORD= {"do",
    "if","else","read","write","while","int","double"
};

//算术符号"+","-","*","/","=","==",">","<","<>"
public static char[] OPERATEWORD= {
    '+','-','*','/','=','>','<','<>'
};

//分界符
public static char[] DIVIDEWORD= {
    '(',')','{','}','[',']',';',':'
};

public static char[] WHITESPACE= {
    ' ','\n','\t','\r'
};

public static char[] CONNECTOR= {
    '&','|'
};

```

具体用于 judge 一个 token 是不是某类符号的函数如下

```

2 //查找保留字
3 public static boolean IsReservedWord(String keyword) {
4     boolean flag=false;
5     for(int i=0;i<RESERVEDWORD.length;i++) {
6         if(keyword.equals(RESERVEDWORD[i])) {
7             flag=true;
8             return flag;
9         }
10    }
11    return flag;
12 }
13
14 //判断是否是字母
15 public static boolean IsLetter(char letter) {
16     boolean flag=false;
17     if(letter>='a' && letter<='z' || letter>='A' && letter<='Z' || letter=='_') {
18         flag=true;
19         return flag;
20     }
21     return flag;
22 }

```

```
// 判断是否是数字
public static boolean IsDigit(char digit) {
    boolean flag=false;
    if(digit>='0' && digit<='9') {
        flag=true;
        return flag;
    }
    return flag;
}

// 判断一个数字是不是Int
public static boolean IsInteger(String digit) {
    try {
        int num=Integer.valueOf(digit);//把字符串强制转换为数字
        return true;//如果是数字，返回True
    } catch (Exception e) {
        return false;//如果抛出异常，返回False
    }
}

// 判断一个数字是不是double
public static boolean IsDouble(String digit) {
    try {
        double num=Double.valueOf(digit);//把字符串强制转换为数字
        return true;//如果是数字，返回True
    } catch (Exception e) {
        return false;//如果抛出异常，返回False
    }
}
```

```
//判断是否是界符
public static boolean IsWhiteSpace(char ch) {
    boolean flag=false;
    for(int i=0;i< WHITESPACE.length;i++) {
        if(ch== WHITESPACE[i]) {
            flag=true;
            return flag;
        }
    }
    return flag;
}
```

```
//判断是否是操作符
public static boolean IsOperator(char divide) {
    boolean flag=false;
    for(int i=0;i< OPERATEWORD.length;i++) {
        if(divide== OPERATEWORD[i]) {
            flag=true;
            return flag;
        }
    }
    return flag;
}
```

```
//判断是否是界符
public static boolean IsDivide(char c) {
    boolean flag=false;
    for(int i=0;i< DIVIDEWORD.length;i++) {
        if(c== DIVIDEWORD[i]) {
            flag=true;
            return flag;
        }
    }
    return flag;
}
```

2.2.3 Lexer 主体逻辑

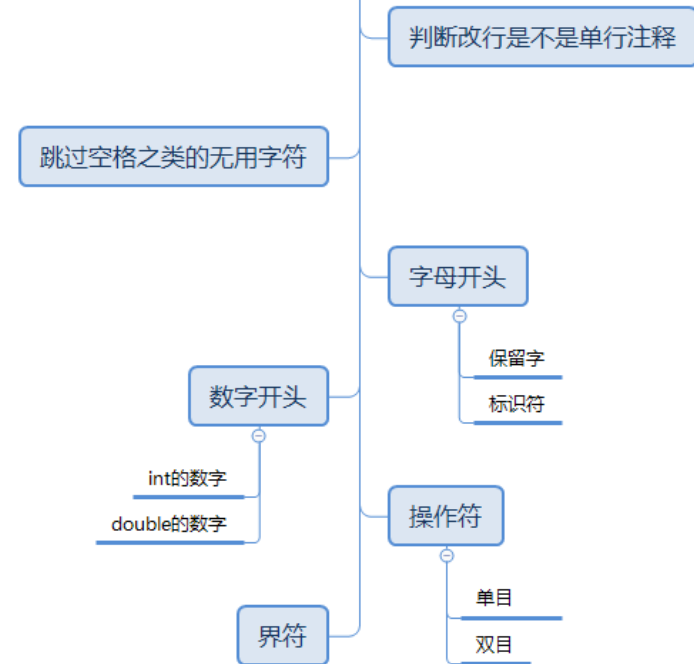
首先读取文件 然后按行进行 Lexer 解析

```
public static LinkedList<Token> lexerCMMFromFile(String filePath) throws IOException, LexerException {
    Lexer tempLexer=new Lexer();
    tokenList = new LinkedList<Token>();
    Errors = new LinkedList<Exception>();
    curLine=1;
    curColumn=0;

    // 打开文件创建输入流
    File f=new File(filePath);
    FileReader fileReader=new FileReader(f);
    BufferedReader bufferedReader=new BufferedReader(fileReader);
    // 从输入流逐行读取
    String templLine;
    while((templLine=bufferedReader.readLine())!=null) {
        // 对该行CMM代码进行解析
        tempLexer.lexer(templLine);
        // 行数++,列数变为1重新分析下一行
        curColumn=0;
        curLine++;
    }
    if(isMultilineCom) {
        throw new LexerException(curLine,curColumn, "多行注释未闭合");
    }
    for(Exception curException: Errors) {
        System.out.println(curException.toString());
    }
    return tokenList;
}
```

Lexer 主体逻辑

判断之前的多行注释有没有结束 没有结束就跳过改行



主体流程大概就这样 具体代码再项目当中。

```

while(curColumn < tempLine.length()) {
    if(isMultilineCom) {
        if(tempLine.charAt(tempLine.length()-1) == '/' && tempLine.charAt(tempLine.length()-2) == '*') {
            isMultilineCom = false;
            curMultilineComLine--;
            break;
        }
        else {
            break;
        }
    }

    else {
        // 若为注释//,则去除注释后面的东西
        if (tempLine.charAt(curColumn) == '/' && tempLine.charAt(curColumn+1) == '/') {
            break;
        }

        if (tempLine.charAt(curColumn) == '/' && tempLine.charAt(curColumn+1) == '*') {
            isMultilineCom = true;
            curMultilineComLine = curLine;
            break;
        }
    }
}
  
```

读取之后解析一个个 token 把他们存到同一个 tokenList 中用于之后的分析。

```

private static LinkedList<Token> tokenList;
  
```

2.2.4 错误处理

词法分析之中会出现这些错误，我们需要对错误情况进行报告

1. 多行注释未结束

我设计了一个全局变量 `isMultilineCom` 用于检测多行注释是否结束，如果全部行的数据都读取之后，该值还是 1 的话，就说明多行注释没有结束，就报错。

```
if(isMultilineCom) {  
    throw new LexerException(curLine, curColumn, "多行注释未闭合");  
}  
for(Exception curException: Errors) {  
    System.out.println(curException.toString());  
}  
return tokenList;
```

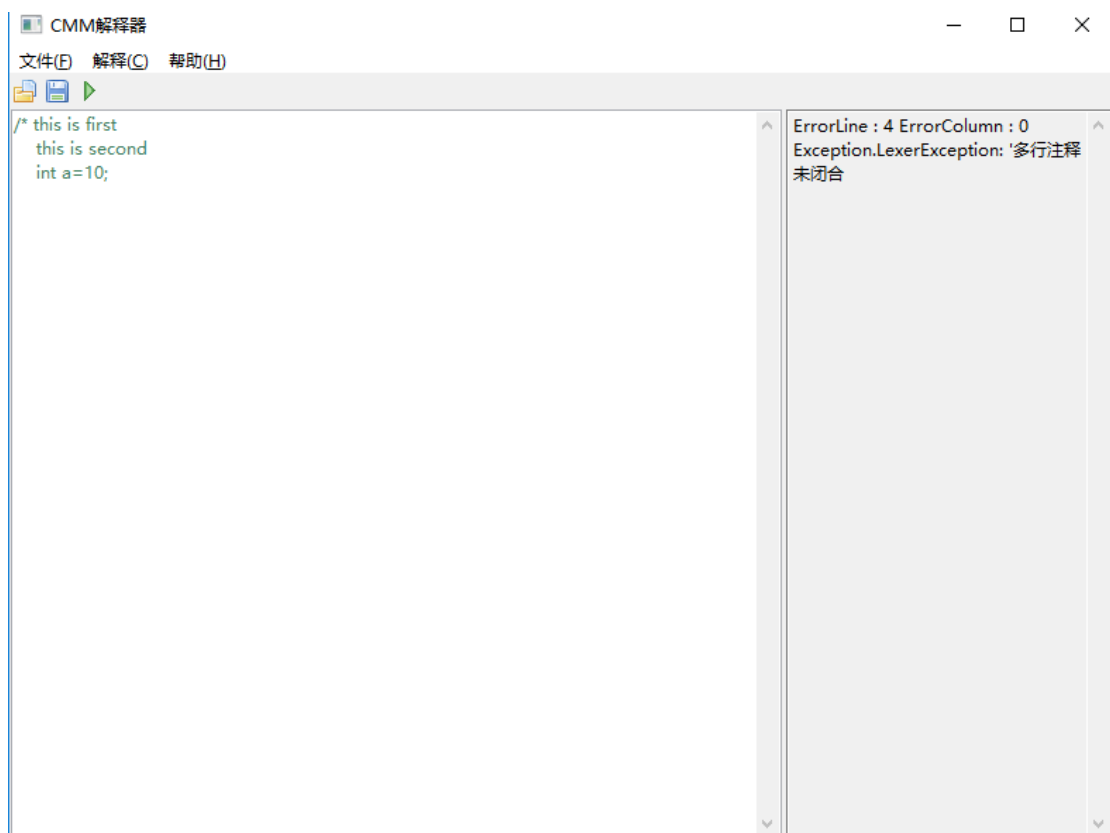
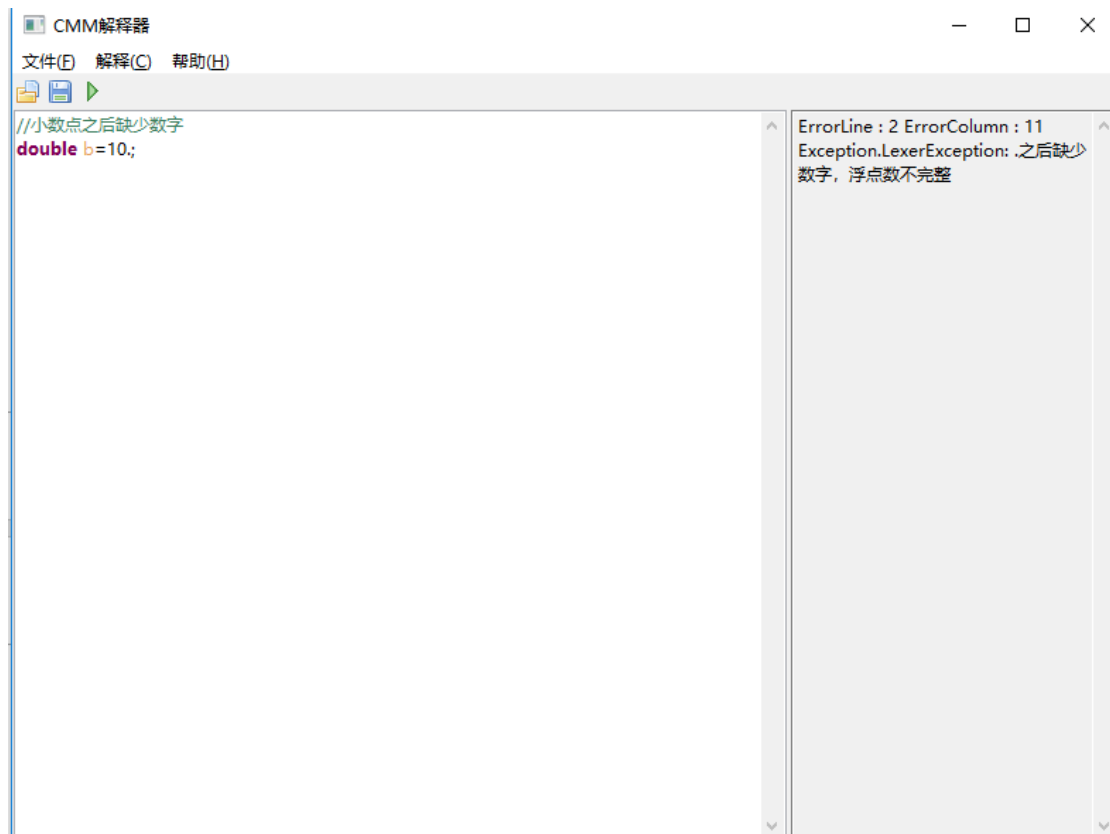
2. 浮点数缺少小数位

小数之后没有检测到数字的话，就表示浮点数少了小数位，就报错。

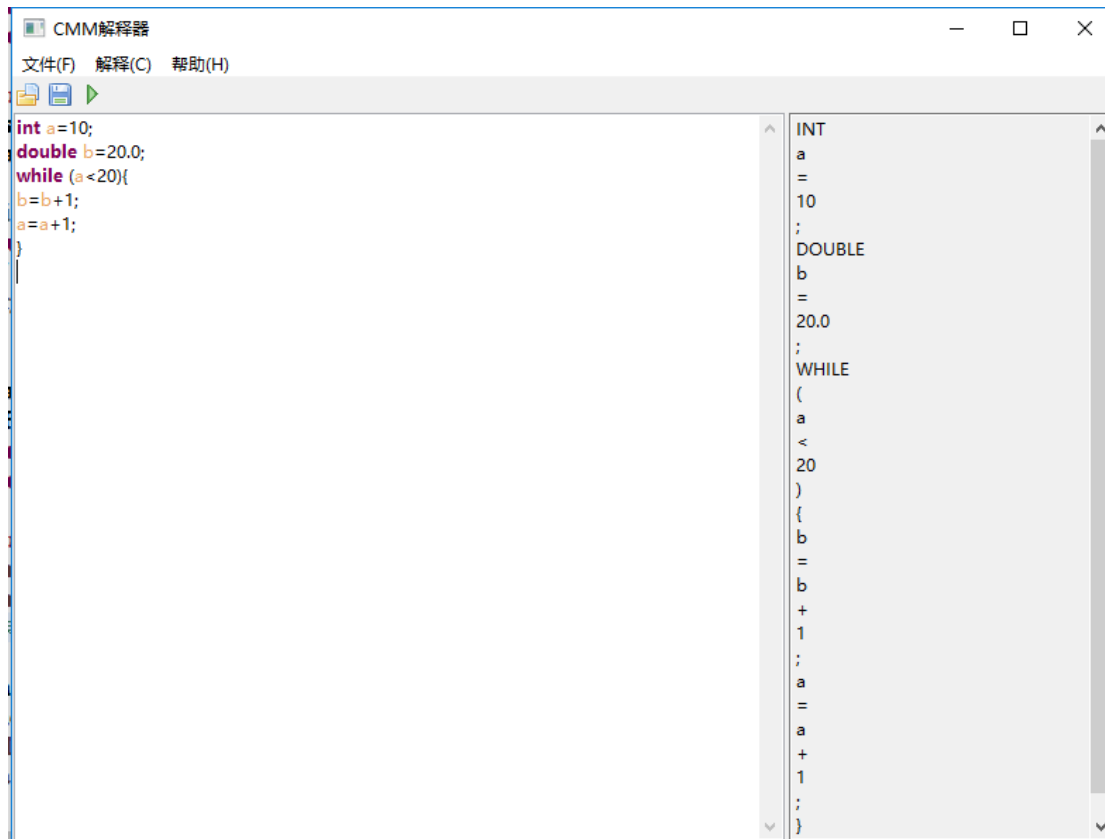
```
        }  
        //若在数字后有小数点，继续判断  
        if(tempLine.charAt(curColumn+1)=='.')  
        {  
            curColumn++;  
            stringBuilder.append(tempLine.charAt(curColumn));  
            //小数点后无数字，检测出错  
            if(!JudgeFunction.IsDigit(tempLine.charAt(curColumn+1))){  
                throw new LexerException(curLine, curColumn, ".之后缺少数字，浮点数不完整");  
                stringBuilder.delete(0, stringBuilder.length());  
                continue;  
            }  
            //小数点后有数字，检测为小数  
        }  
        else  
        {
```

2.2.5 测试

错误情况



正常情况



2.2.6 集成为工具到 Service 之中供给后面使用

这样把每个类的主要功能都集中到 Service 之中就会方便很多。利于代码的组织与交互。

```
public class Service {  
    public static LinkedList<Token> LexerFromFile(String filePath) throws Exception, LexerException{  
        return Lexer.LexerCMMFromFile(filePath);  
    }  
}
```


2.3 语法分析器的构建

2.3.1 文法构建

```
program -> Stmts
Stmts -> Stmt; Stmts | Stmt | ε
Stmt -> IfStmt | WhileStmt | DoWhileStmt | ReadStmt | WriteStmt |
DeclareStmt | StmtBlock | AssignStmt
IfStmt -> if ( Exp ) Stmt else Stmt
WhileStmt -> while (Exp) Stmt
DoWhileStmt -> do Stmt While(Exp);
ReadStmt -> read (Var);
WriteStmt -> write(Exp);
DeclareStmt -> Type Var;
StmtBlock -> { Stmt }
AssignStmt -> Var = Exp;
variable -> identifier [ [ exp ] ]
exp -> additive-exp logical-op additive-exp | additive-exp
additive-exp -> term add-op additive-exp | term
term -> factor mul-op term | factor
factor -> ( exp ) | number | variable | Add-op exp
logical-op -> > | < | >= | <= | <> | ==
add-op -> + | -
mul-op -> * | /
```

在我这个项目里面 我才用的递归下降分析法(LL)所以就针对以上文法构建递归下降子程序。

2.3.2 数据体构建

语法树节点构建

```

//该节点的类型
private int type;
//左孩子
private TreeNode mLeft;
//中孩子
private TreeNode mMiddle;
//右孩子
private TreeNode mRight;

private int mDataType;

private String value;

private TreeNode mNext;

```

其中

```

/**
 * {@link TreeNode#getType()}为{@link TreeNode#VAR}时存储变量类型
 * {@link TreeNode#getType()}为{@link TreeNode#OP}时存储操作符类型
 * {@link TreeNode#getType()}为{@link TreeNode#EXP}时表示复合表达式
 * {@link TreeNode#getType()}为{@link TreeNode#FACTOR}表示因子,mDataType处存储表达式的前置符号
 * {@link TreeNode#getType()}为{@link TreeNode#LITREAL}表示字面值,存储类型
 */
private int mDataType;

/**
 * {@link TreeNode#getType()}为{@link TreeNode#FACTOR}时存储表达式的字符串形式的值
 * {@link TreeNode#getType()}为{@link TreeNode#VAR}时存储变量名
 */

```

2.3.3 递归下降子程序构建

首先构建出看下位符号内容，类别等等的函数工具便于后面进行判断分支分析

```

private static int getNextTokenLineNum() {
    if (iterator.hasNext()) {
        int lineNo = iterator.next().getLineNum();
        iterator.previous();
        return lineNo;
    }
    return -1;
}

private static int getNextTokenColumnNum() {
    if (iterator.hasNext()) {
        int lineNo = iterator.next().getColumnNum();
        iterator.previous();
        return lineNo;
    }
    return -1;
}

private static String getNextTokenValue(int type) {
    String value=null;
    if (type==getNextTokenType()) {
        value = iterator.next().getValue();
        iterator.previous();
    }
    return value;
}
}

```

然后构建出用于匹配看到的和期望的是不是同一个符号的函数。

```

*
* @param type
* @throws ParseException 匹配失败则抛出相关异常
*/
private static void matchNextToken(int type) throws ParseException {
    if (iterator.hasNext()) {
        currentToken = iterator.next();
        if (currentToken.getType() == type) {
            return;
        }
    }
    throw new ParseException(getNextTokenLineNum(), getNextTokenColumnNum(),
        " 下个符号应该是 -> " + new Token(type, 0));
}

/**
 * 检查下一个token的类型是否和type中的每一个元素相同,调用此函数currentToken位置不会移动
 *
 * @param type
 * @return 相同为true,不同为false
 */
private static boolean checkNextTokenType(int... type) {
    if (iterator.hasNext()) {
        int nextType = iterator.next().getType();
        iterator.previous();
        for (int each : type) {
            if (nextType == each) {
                return true;
            }
        }
    }
    return false;
}
}

```

接下来就根据文法构建出分析框架

```
private static TreeNode parseStmt() throws ParserException {
    switch (getNextTokenType()) {
        case Token.IF:
            return parseIfStmt();
        case Token.WHILE:
            return parseWhileStmt();
        case Token.DO:
            return parseDoWhileStmt();
        case Token.READ:
            return parseReadStmt();
        case Token.WRITE:
            return parseWriteStmt();
        case Token.INT:
        case Token.DOUBLE:
            return parseDeclareStmt();
        case Token.LBRACE:
            return parseStmtBlock();
        case Token.ID:
            return parseAssignStmt();
        default:
            throw new ParserException(getNextTokenLineNum(), getNextTokenColumnNum(), ": expected token");
    }
}
```

然后针对每种不同的语句构建不同的语法树

IfStmt



匹配构建语法树代码如下

```
/**
 * if语句
 * if ( Exp ) Stmt else Stmt
 * @throws ParserException
 */
private static TreeNode parseIfStmt() throws ParserException {
    TreeNode node = new TreeNode(TreeNode.IF_STMT);
    matchNextToken(Token.IF);
    matchNextToken(Token.LPARENT);
    node.setLeft(parseExp());
    matchNextToken(Token.RPARENT);
    node.setMiddle(parseStmt());
    if (getNextTokenType() == Token.ELSE) {
        matchNextToken(Token.ELSE);
        node.setRight(parseStmt());
    }
    return node;
}
```

WhileStmt



匹配构建代码树代码如下

```

/**
 * while 语句
 * while (Exp) Stmt
 * @throws ParseException
 */
private static TreeNode parseWhileStmt() throws ParseException {
    TreeNode node = new TreeNode(TreeNode.WHILE_STMT);
    matchNextToken(Token.WHILE);
    matchNextToken(Token.LPARENT);
    node.setLeft(parseExp());
    matchNextToken(Token.RPARENT);
    node.setMiddle(parseStmt());
    return node;
}
  
```

DoWhileStmt



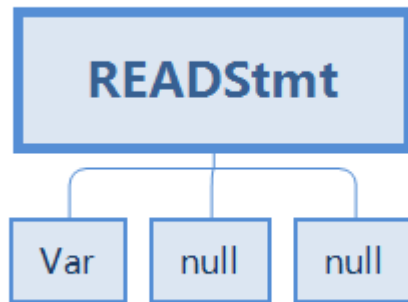
匹配构建代码树代码如下

```

/**
 * Dowhile语句
 * do Stmt While(Exp);
 * @throws ParseException
 */
private static TreeNode parseDoWhileStmt() throws ParseException {
    TreeNode node = new TreeNode(TreeNode.DOWHILE_STMT);
    matchNextToken(Token.DO);
    node.setLeft(parseStmt());
    matchNextToken(Token.WHILE);
    matchNextToken(Token.LPARENT);
    node.setMiddle(parseExp());
    matchNextToken(Token.RPARENT);
    matchNextToken(Token.SEMI);
    return node;
}

```

ReadStmt



匹配构建代码树代码如下

```

/**
 * read语句
 * read (Var);
 * @throws ParseException
 */
private static TreeNode parseReadStmt() throws ParseException {
    TreeNode node = new TreeNode(TreeNode.READ_STMT);
    matchNextToken(Token.READ);
    matchNextToken(Token.LPARENT);
    node.setLeft(variableName());
    matchNextToken(Token.RPARENT);
    matchNextToken(Token.SEMI);
    return node;
}

```

WriteStrmt



匹配构建代码树代码如下

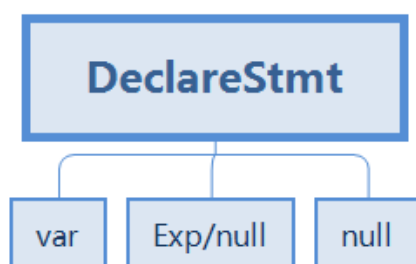
```

/**
 * write语句
 * write(Exp);
 * @throws ParseException
 */
private static TreeNode parseWriteStmt() throws ParseException {
    TreeNode node = new TreeNode(TreeNode.WRITE_STMT);
    matchNextToken(Token.WRITE);
    matchNextToken(Token.LPARENT);
    node.setLeft(parseExp());
    matchNextToken(Token.RPARENT);
    matchNextToken(Token.SEMI);
    return node;
}
  
```

DeclareStmt

声明但是未赋值的话就是中间节点就是 Null

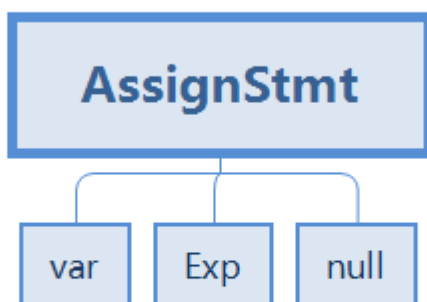
声明并且赋值的话就是中间节点就是 Exp



匹配构建代码树代码如下

```
* declare语句
* Type Var
* @throws ParseException
*/
private static TreeNode parseDeclareStmt() throws ParseException {
    TreeNode node = new TreeNode(TreeNode.DECLARE_STMT);
    TreeNode varNode = new TreeNode(TreeNode.VAR);
    if (checkNextTokenType(Token.INT, Token.DOUBLE)) {
        currentToken = iterator.next();
        int type = currentToken.getType();
        if (type == Token.INT) {
            varNode.setDataType(Token.INT);
        } else if (type == Token.DOUBLE) {
            varNode.setDataType(Token.DOUBLE);
        }
    } else {
        throw new ParseException(getNextTokenLineNum(), getNextTokenColumnNum(), " 下个符号应该是 variable type");
    }
    if (checkNextTokenType(Token.ID)) {
        currentToken = iterator.next();
        varNode.setValue(currentToken.getValue());
    } else {
        throw new ParseException(getNextTokenLineNum(), getNextTokenColumnNum(), " 下个符号应该是 ID");
    }
    if (getNextTokenType() == Token.ASSIGN) {
        matchNextToken(Token.ASSIGN);
        node.setMiddle(parseExp());
    } else if (getNextTokenType() == Token.LBRACKET) {
        matchNextToken(Token.LBRACKET);
        varNode.setLeft(parseExp());
        matchNextToken(Token.RBRACKET);
    }
    matchNextToken(Token.SEM);
    node.setLeft(varNode);
    return node;
}
```

AssignStmt



匹配构建代码树代码如下


```

/**
 * assign语句
 * Var = Exp;
 * @throws ParseException
 */
private static TreeNode parseAssignStmt() throws ParseException {
    TreeNode node = new TreeNode(TreeNode.ASSIGN_STMT);
    node.setLeft(variableName());
    matchNextToken(Token.ASSIGN);
    node.setMiddle(parseExp());
    matchNextToken(Token.SEMI);
    return node;
}

```

2.3.4 Paser 程序入口以及集成到 Service 之中

```

public static LinkedList<TreeNode> PaserFromFile(String filePath) throws ParseException, IOException, LexerExc
// try {
//     Errors = new LinkedList<Exception>();
//     LinkedList<Token> tokens=Lexer.lexCMMFromFile(filePath);
//     Paser parser=new Paser();
//     LinkedList<TreeNode> treeNodeList=parser.syntacticAnalyse(tokens);
// } catch (Exception e) {
//     System.out.println(e);
// }
// return treeNodeList;
}

private LinkedList<TreeNode> syntacticAnalyse(LinkedList<Token> tokenList) throws ParseException {
    treeNodeList = new LinkedList<TreeNode>();
    iterator = tokenList.listIterator();
    while (iterator.hasNext()) {
        treeNodeList.add(parseStmt());
    }
    return treeNodeList;
}

```

在此我提供了两个函数，PaserFromFile 使用了文件路径作为参数来直接从文件一直执行出语法分析的结果。

而 syntacticAnalyse 则是用 **LinkedList<Token> tokenList**

作为参数，不断的从 **tokenList** 之中取 **token** 来递归下降分析。

最后把他们封装然后在 **Service** 中提供服务

```

public static LinkedList<TreeNode> PaserFromFile(String filePath) throws ParseException, IOException, LexerExc
return Paser.PaserFromFile(filePath);
}

```

2.3.5 错误处理

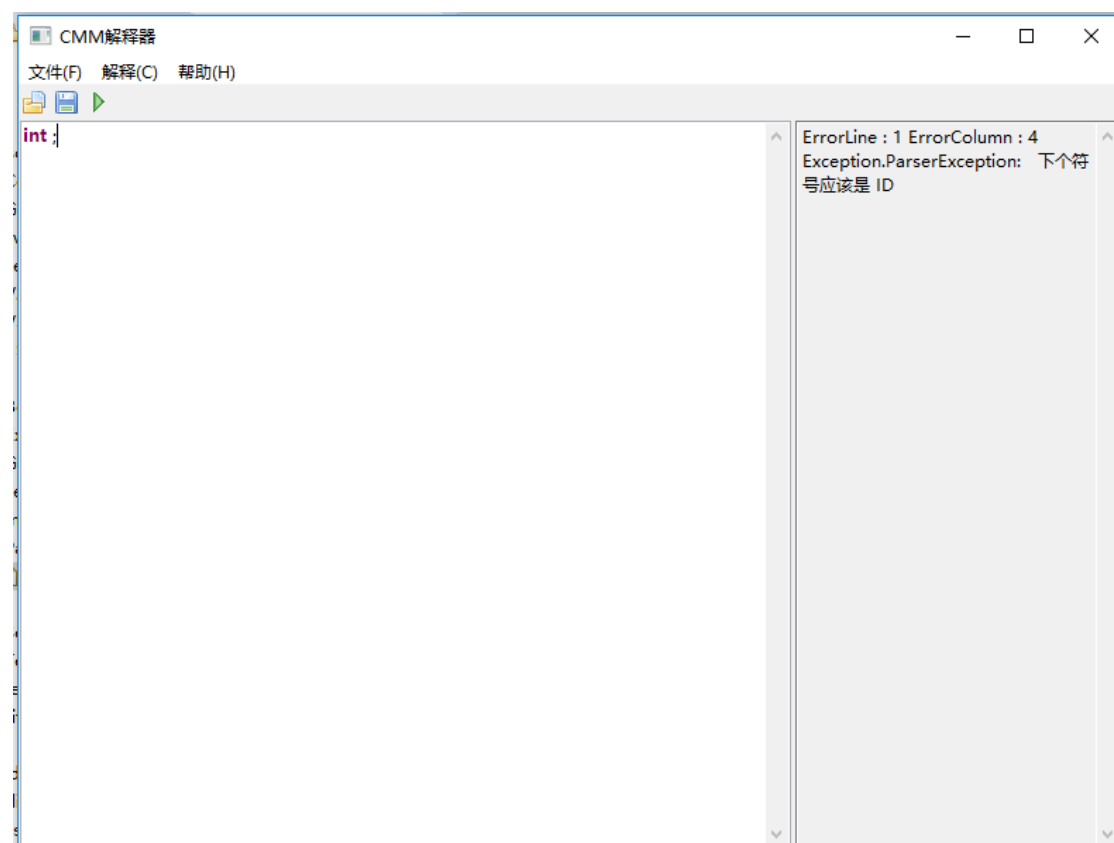
语法分析之中只会检查出一种错误，就是当前看到的符号和预期的符号不符合的话，就会进行报错。

```
    } else {  
        throw new ParseException(getNextTokenLineNum(),getNextTokenColumnNum(), " 下个符号应该是 ID");  
    }  
    throw new ParseException( getNextTokenLineNum(),getNextTokenColumnNum(), " 下个符号应该是 DOUBLE value");  
}
```

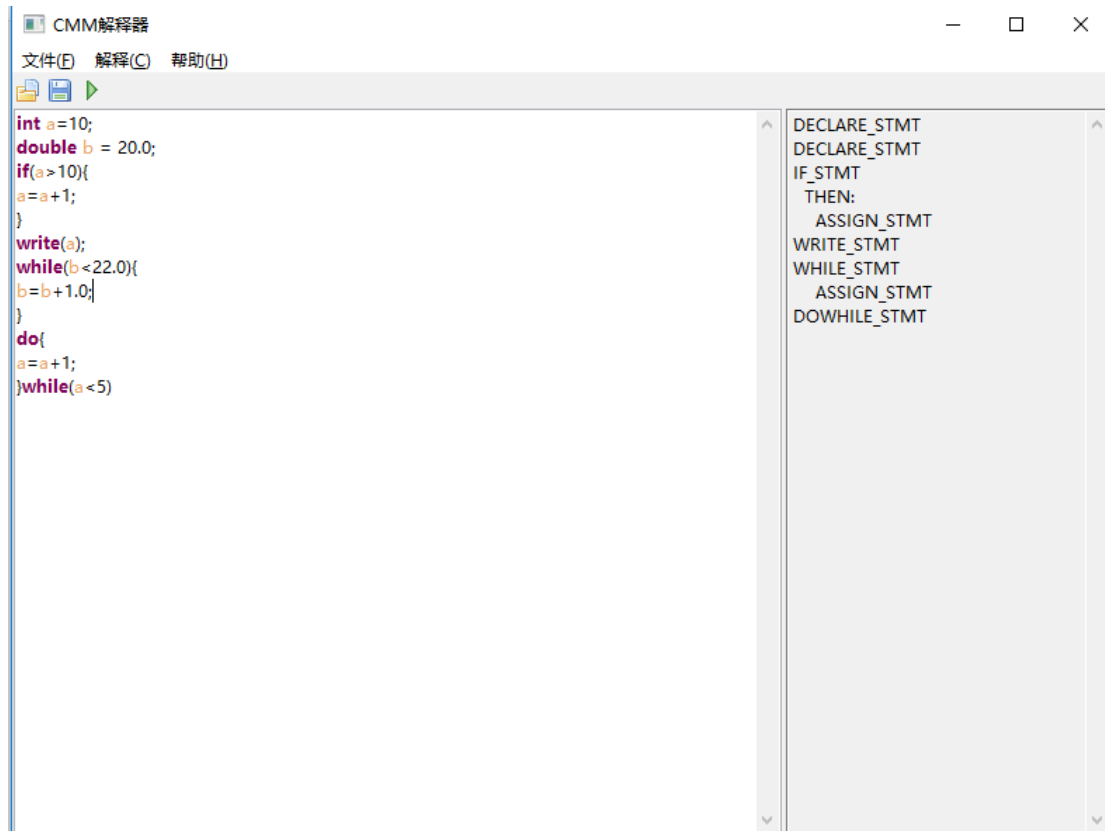
有很多情况会报错，但是都是这种情况，就不一一列举了。具体可参见源码。

2.3.6 测试

异常情况



正常情况



2.4 语义分析器的构建

2.4.1 数据实体构建

语义分析之中要使用符号表对变量进行存储，所以首要的任务就是构建符号表之中的符号。符号表的属性以及解释见下图。

```
//符号表中的符号，符号表采用链表实现
public class Symbol {
    public static final int TEMP = -1;
    public static final int SINGLE_INT = 0;
    public static final int SINGLE_Double = 1;
    public static final int ARRAY_INT = 2;
    public static final int ARRAY_Double = 3;
    //仅供value使用
    public static final int TRUE = 4;
    public static final int FALSE = 5;
    //符号名字
    private String name;
    //符号类型
    private int type;
    //符号的值
    private Value value;
    //符号的作用域
    private int level;
    //下一个符号
    private Symbol next;
```

其中这个 Value 是用于进行存储符号的值的。Value 类的设计如下。可以存储 int, double 单值。也能够存储 int, double 数组的值，并且能够实现加减乘除以及逻辑比较的运算。

```
public class Value {
    //存储值对象的类型,常量存储在Symbol中
    private int mType;
    //用于类型为int的数据
    private int mInt;
    //用于存储类型为double的数据
    private double mDouble;
    //用于类型为int[]的数组数据
    private int[] mArrayInt;
    //用于类型为double[]的数组数据
    private double[] mArrayDouble;
```

其中数组初始化之后的初始值都是 0

//初始化数组

```
public void initArray(int dim) {  
    if (mType == Symbol.ARRAY_INT) {  
        mArrayInt = new int[dim];  
    } else {  
        mArrayDouble = new double[dim];  
    }  
}
```

加法运算举例

//加法

```
public Value PLUS(Value value) throws ConvertException {  
    if (this.mType == Symbol.SINGLE_Double) {  
        Value rv = new Value(Symbol.SINGLE_Double);  
        if (value.mType == Symbol.SINGLE_INT) {  
            rv.setDouble(this.mDouble + value.mInt);  
            return rv;  
        } else if (value.mType == Symbol.SINGLE_Double) {  
            rv.setDouble(this.mDouble + value.mDouble);  
            return rv;  
        }  
    } else if (this.mType == Symbol.SINGLE_INT) {  
        if (value.mType == Symbol.SINGLE_INT) {  
            Value rv = new Value(Symbol.SINGLE_INT);  
            rv.setInt(this.mInt + value.mInt);  
            return rv;  
        } else if (value.mType == Symbol.SINGLE_Double) {  
            Value rv = new Value(Symbol.SINGLE_Double);  
            rv.setDouble(this.mInt + value.mDouble);  
            return rv;  
        }  
    }  
}
```

完成了上面的准备工作之后就是要开始构建存储 Symbol 的 SymbolTable 了。
SymbolTable 的属性如下。

SymbolList 用于存储主体的 Symbol tempNames 则用于存储临时使用的
Symbol 。

```

public class SymbolTable {
    private static final String TEMP_PREFIX = "**temp";

    private static SymbolTable symbolTable = new SymbolTable();
    private static LinkedList<Symbol> tempNames;

    private ArrayList<Symbol> symbolList;

```

SymbolTable 需要实现这些功能 setSymbol 的一些属性以及 getSymbol 的一些属性

实现如下:

```

public void setSymbolValue(String name, Value value) throws ConvertException {
    getSymbol(name).setValue(value);
}

public void setSymbolValue(String name, int value, int index) throws ConvertException {
    if (getSymbol(name).getValue().getArrayInt().length > index) {
        getSymbol(name).getValue().getArrayInt()[index] = value;
    } else {
        throw new ConvertException("数组 < " + name + "> 下标 " + index + " 越界");
    }
}

public int getSymbolType(String name) throws ConvertException {
    return getSymbol(name).getType();
}

//取单值用这个函数
public Value getSymbolValue(String name) throws ConvertException {
    return getSymbolValue(name, -1);
}

//取值用这个函数
public Value getSymbolValue(String name, int index) throws ConvertException {
    Symbol s = getSymbol(name);
    if (index == -1) { //单值
        return s.getValue();
    } else {
        if (s.getValue().getArrayInt().length < index + 1) {
            throw new ConvertException("数组 < " + name + "> 下标 " + index + " 越界");
        }
        if (s.getType() == Symbol.ARRAY_INT) {
            Value rv = new Value(Symbol.SINGLE_INT);
            rv.setInt(s.getValue().getArrayInt()[index]);
            return rv;
        } else {
            Value rv = new Value(Symbol.SINGLE_Double);
            rv.setDouble(s.getValue().getArrayDouble()[index]);
            return rv;
        }
    }
}

```

然后还需要实现作用域的区别, 因此就需要实现在退出一个层次的作用域时, 删除该层次的 Symbol, 并且在声明一个变量的时候, 检查之前有没有声明过, 具体实现如下。

```

public void register(Symbol symbol) throws ConvertException {
    for (int i=0; i<symbolList.size(); i++) {
        if (symbolList.get(i).getName().equals(symbol.getName())) {
            if (symbolList.get(i).getLevel() < symbol.getLevel()) {
                symbol.setNext(symbolList.get(i));
                symbolList.set(i, symbol);
                return;
            } else {
                throw new ConvertException("变量 <" + symbol.getName() + "> 重复声明");
            }
        }
    }
    symbolList.add(symbol);
}

public void deregister(int level) {
    for (int i=0; i<symbolList.size(); i++) {
        if (symbolList.get(i).getLevel() == level) {
            symbolList.set(i, symbolList.get(i).getNext());
        }
    }
    for (int i=symbolList.size()-1; i>=0; i--) {
        if (symbolList.get(i) == null) {
            symbolList.remove(i);
        }
    }
}
}

```

接下来就是中间代码 FourCode 的实现了，FourCode 用于记录中间代码。属性如下：

```

public class FourCode {
    public static final String JMP = "jmp";
    public static final String READ = "read";
    public static final String WRITE = "write";
    public static final String IN = "in";
    public static final String OUT = "out";
    public static final String INT = "int";
    public static final String DOUBLE = "double";
    public static final String ASSIGN = "assign";
    public static final String PLUS = "+";
    public static final String MINUS = "-";
    public static final String MUL = "*";
    public static final String DIV = "/";
    public static final String GT = ">";
    public static final String LT = "<";
    public static final String GET = ">=";
    public static final String LET = "<=";
    public static final String EQ = "==";
    public static final String NEQ = "!=";

    private String op;
    private String arg1;
    private String arg2;
    private String result;
}

```

图示就是这样



Op 用于记录该操作是干什么

Arg1 是第一个参数

Arg2 是第二个参数

Result 用于指明操作目的

2.4.2 转换成中间代码

转换类属性有这些

```
public class Convert2FourCode {  
    // 记录当前的作用域层次  
    private int mLevel;  
    // 记录当前的四元式索引  
    public int mLine;  
    // 存储生成的四元式  
    public static LinkedList<FourCode> codes;  
    // 符号表用于记录符号  
    private static SymbolTable symbolTable;
```

针对不同的 TreeNode 来实现相应的转换代码。主体逻辑如下：


```

1 public void Convert(TreeNode node) throws ConvertException {
2     while (true) {
3         switch (node.getType()) {
4             case TreeNode.IF_STMT:
5                 ConvertIfStmt(node);
6                 break;
7             case TreeNode.WHILE_STMT:
8                 ConvertWhileStmt(node);
9                 break;
10            case TreeNode.DOWHILE_STMT:
11                ConvertDoWhileStmt(node);
12                break;
13            case TreeNode.READ_STMT:
14                ConvertReadStmt(node);
15            case TreeNode.WRITE_STMT:
16                ConvertWriteStmt(node);
17            case TreeNode.DECLARE_STMT:
18                ConvertDeclareStmt(node);
19                break;
20            case TreeNode.ASSIGN_STMT:
21                ConvertAssignStmt(node);
22                break;
23        }
24        symbolTable.clearTempNames();
25        if (node.getNext() != null) {
26            node = node.getNext();
27        } else {
28            break;
29        }
30    }
31 }

```

IfStmt Node 的转换逻辑

```

// 将IFStmt转换为四元式
// 0 Exp=>(op,arg1,arg2,*temp1)
// 1 (jmp,*temp1,null,n+1)
// 2 (in,null,null,null)
// ...
// n (out, null, null, null)
// n+1 (...,...,...)
private void ConvertIfStmt(TreeNode node) throws ConvertException {
    if (node.getType() == TreeNode.IF_STMT) {
        // 条件跳转 jmp 条件 null 目标 条件为假时跳转
        FourCode falsejmp = new FourCode(FourCode.JMP, ConvertExp(node.getLeft()), null, null);
        codes.add(falsejmp);
        mLine++;
        codes.add(new FourCode(FourCode.IN, null, null, null));
        mLine++;
        mLevel++;
        Convert(node.getMiddle());
        SymbolTable.getSymbolTable().deregister(mLevel);
        mLevel--;
        codes.add(new FourCode(FourCode.OUT, null, null, null));
        mLine++;
        if (node.getRight() != null) {
            FourCode outjump = new FourCode(FourCode.JMP, null, null, null);
            codes.add(outjump);
            mLine++;
            falsejmp.setResult(String.valueOf(mLine + 1));
            codes.add(new FourCode(FourCode.IN, null, null, null));
            mLine++;
            mLevel++;
            Convert(node.getRight());
            codes.add(new FourCode(FourCode.OUT, null, null, null));
            mLine++;
            SymbolTable.getSymbolTable().deregister(mLevel);
            mLevel--;
            outjump.setResult(String.valueOf(mLine + 1));
        } else {
    }
}

```

WhileStmt Node 的转换逻辑

```

// 将WHILEStmt转换为四元式 While(Exp){...}
// WHILEStmt得到的四元式
// 0 Exp=>(op,arg1,arg2,*temp1)
// 1 (jmp,*temp1,null,n+1)
// 2(in,null,null,null)
// ...
// n-1(jmp,null,null,1)
// n(out,null,null,null)
// n+1(...,...,...)
private void ConvertWhileStmt(TreeNode node) throws ConvertException {
    if (node.getType() == TreeNode.WHILE_STMT) {
        int jmline = mLine;
        FourCode falsejmp = new FourCode(FourCode.JMP, ConvertExp(node.getLeft()), null, null);
        codes.add(falsejmp);
        mLine++;
        codes.add(new FourCode(FourCode.IN, null, null, null));
        mLine++;
        mLevel++;
        Convert(node.getMiddle());
        SymbolTable.getSymbolTable().deregister(mLevel);
        mLevel--;
        codes.add(new FourCode(FourCode.OUT, null, null, null));
        mLine++;
        codes.add(new FourCode(FourCode.JMP, null, null, jmline + ""));
        mLine++;
        falsejmp.setResult(String.valueOf(mLine));
    }
}

```

DoWhile Node 的转换逻辑

```

// 将DOWHILEStmt转换为四元式
// DOWHILEStmt得到的四元式
// o(in,null,null,null)
// ...
// ...
// n-3 Exp=>(op,arg1,arg2,*temp1)
// n-2(jmp,*temp1,null,n+1)
// n-1(out,null,null,null)
// n (jmp,null,null,0)
// n+1(...,...,...)
private void ConvertDoWhileStmt(TreeNode node) throws ConvertException {
    if (node.getType() == TreeNode.DOWHILE_STMT) {
        int jmpline = mLine + 1;
        codes.add(new FourCode(FourCode.IN, null, null, null));
        mLine++;
        mLevel++;
        Convert(node.getLeft());
        FourCode falsejmp = new FourCode(FourCode.JMP, ConvertExp(node.getMiddle()), null, null);
        codes.add(falsejmp);
        mLine++;
        SymbolTable.getSymbolTable().deregister(mLevel);
        mLevel--;
        codes.add(new FourCode(FourCode.OUT, null, null, null));
        mLine++;
        codes.add(new FourCode(FourCode.JMP, null, null, jmpline + ""));
        mLine++;
        falsejmp.setResult(String.valueOf(mLine));
    }
}

```

ReadStmt Node 的转换逻辑

```

// 将READStmt转换为四元式
// Single READStmt得到的四元式=>(read,null,null,Varname)
// Array READStmt得到的四元式=>(read,null,index,Varname)
private void ConvertReadStmt(TreeNode node) throws ConvertException {
    if (node.getType() == TreeNode.READ_STMT) {
        String varname = null;
        int type = symbolTable.getSymbolType(node.getLeft().getValue());
        switch (type) {
            case Symbol.SINGLE_INT:
            case Symbol.SINGLE_Double:
                codes.add(new FourCode(FourCode.READ, null, null, node.getLeft().getValue()));
                mLine++;
                break;
            case Symbol.ARRAY_INT:
            case Symbol.ARRAY_Double:
                codes.add(new FourCode(FourCode.READ, null, ConvertExp(node.getLeft().getLeft()),
                    node.getLeft().getValue()));
                mLine++;
                break;
            case Symbol.TEMP:
            default:
                throw new ConvertException("输入语句有误");
        }
    }
}

```

WriteStmt Node 的转换逻辑

```

// 将WRITESTmt转换为四元式
// WRITESTmt得到的四元式=>(write,null,null,Exp)
private void ConvertWriteStmt(TreeNode node) throws ConvertException {
    if (node.getType() == TreeNode.WRITE_STMT) {
        codes.add(new FourCode(FourCode.WRITE, null, null, ConvertExp(node.getLeft())));
        mLine++;
    }
}

```

DeclareStmt Node 的转换逻辑

```
// 将DeclareStmt转换为四元式
// Single声明没有赋值得到的四元式=>(int/double,null,null,Varname)
// Single声明之后赋值得到的四元式=>(int/double,value,null,Varname)
// Array声明得到的四元式=>(int/double,null,len,Varname)
private void ConvertDeclareStmt(TreeNode node) throws ConvertException {
    if (node.getType() == TreeNode.DECLARE_STMT) {
        SymbolTable table = SymbolTable.getSymbolTable();
        TreeNode var = node.getLeft();
        if (var.getLeft() == null) { // 单值
            String value = null;
            if (node.getMiddle() != null) { // 声明并且赋值
                value = ConvertExp(node.getMiddle());
            }
            if (var.getDataType() == Token.INT) {
                codes.add(new FourCode(FourCode.INT, value, null, var.getValue()));
                mLine++;
                Symbol symbol = new Symbol(var.getValue(), Symbol.SINGLE_INT, mLevel);
                table.register(symbol);
            } else if (var.getDataType() == Token.DOUBLE) {
                codes.add(new FourCode(FourCode.DOUBLE, value, null, var.getValue()));
                mLine++;
                Symbol symbol = new Symbol(var.getValue(), Symbol.SINGLE_Double, mLevel);
                table.register(symbol);
            }
        } else { // 数组
            String len = ConvertExp(var.getLeft());
            if (var.getDataType() == Token.INT) {
                codes.add(new FourCode(FourCode.INT, null, len, var.getValue()));
                mLine++;
                Symbol symbol = new Symbol(var.getValue(), Symbol.ARRAY_INT, mLevel);
                table.register(symbol);
            } else {
                codes.add(new FourCode(FourCode.DOUBLE, null, len, var.getValue()));
                mLine++;
                Symbol symbol = new Symbol(var.getValue(), Symbol.ARRAY_Double, mLevel);
                table.register(symbol);
            }
        }
    }
}
```

AssignStmt Node 的转换逻辑

```
// 将AssignStmt转换为四元式
// SingleASSIGN得到的四元式=>(assign,value,null,Varname)
// ArrayASSIGN得到的四元式=>(assign,value,null,Varname[index])
private void ConvertAssignStmt(TreeNode node) throws ConvertException {
    if (node.getType() == TreeNode.ASSIGN_STMT) {
        String value = ConvertExp(node.getMiddle());
        TreeNode var = node.getLeft();
        if (var.getLeft() == null) { // 单值
            codes.add(new FourCode(FourCode.ASSIGN, value, null, var.getValue()));
            mLine++;
        } else {
            String index = ConvertExp(var.getLeft());
            codes.add(new FourCode(FourCode.ASSIGN, value, null, var.getValue() + "[" + index + "]"));
            mLine++;
        }
    }
}
```

Exp Node 的转换逻辑

```

// 表达式转四码
private String ConvertExp(TreeNode node) throws ConvertException {
    if (node.getType() == TreeNode.EXP) {
        return Convert_Exp(node);
    } else if (node.getType() == TreeNode.FACTOR) {
        return ConvertFactor(node);
    } else if (node.getType() == TreeNode.VAR) {
        return ConvertVar(node);
    } else if (node.getType() == TreeNode.LITREAL) {
        return ConvertLitreal(node);
    }
    throw new ConvertException("表达式非法");
}

// Exp分三种
// 1.LOGIC_EXP
// 2.ADDITIVE_EXP
// 3.TERM_EXP
private String Convert_Exp(TreeNode node) throws ConvertException {
    switch (node.getDataType()) {
        case Token.LOGIC_EXP:
            return ConvertLogicExp(node);
        case Token.ADDITIVE_EXP:
            return ConvertAdditiveExp(node);
        case Token.TERM_EXP:
            return ConvertTermExp(node);
        default:
            throw new ConvertException("复合表达式非法");
    }
}
}

```

其余的就不再列举，具体结构可以参看源代码。

之后就可以讲这些函数封装，然后放到 Service 之中给外界提供服务。

入口函数如图

```

public static LinkedList<FourCode> ConvertFormFile(String filePath) throws ConvertException, ParserException, I
    Convert2FourCode convert2FourCode=new Convert2FourCode();
    LinkedList<TreeNode> treeNodeList=Paser.PaserFromFile(filePath);
    LinkedList<FourCode> codes=convert2FourCode.ConvertFormTree(treeNodeList);
    return codes;
}

private LinkedList<FourCode> ConvertFormTree(LinkedList<TreeNode> nodeList) throws ConvertException {
    mLine = -1; // 代码编号从0开始
    mLevel = 0;
    codes = new LinkedList<FourCode>();
    symbolTable = SymbolTable.getSymbolTable();
    symbolTable.newTable();
    Convert2FourCode generator = new Convert2FourCode();
    for (TreeNode node : nodeList) {
        generator.Convert(node);
    }
    symbolTable.deleteTable();
    return codes;
}

```

封装之后放到 Service 之中提供服务

```

public static LinkedList<FourCode> ConvertFromFile(String filePath) throws ConvertException, ParserException, I
    return Convert2FourCode.ConvertFormFile(filePath);
}

```

2.4.3 执行中间代码

针对之前解析得到的四元式序列执行的到结果。RunFourCode 类有这些属性

```
public class RunFourCode {  
    //记录作用域  
    private static int mLevel;  
    //下一条执行的指令的索引  
    private static int pc;  
    //符号表  
    private static SymbolTable symbolTable;
```

主体逻辑如图：

```
// 解释执行四元式  
private void interpretFourCode(FourCode code) throws ConvertException, LexerException {  
    String instype = code.getOp();  
    if (instype.equals(FourCode.JMP)) { //跳转指令  
        RunJMP(code);  
        return;  
    }  
    if (instype.equals(FourCode.READ)) {  
        RunREAD(code);  
    }  
    if (instype.equals(FourCode.WRITE)) {  
        RunWRITE(code);  
    }  
    if (instype.equals(FourCode.IN)) {  
        RunIN(code);  
    }  
    if (instype.equals(FourCode.OUT)) {  
        RunOUT(code);  
    }  
    if (instype.equals(FourCode.INT)) {  
        RunDECLARE(code, instype);  
    }  
    if (instype.equals(FourCode.DOUBLE)) {  
        RunDECLARE(code, instype);  
    }  
    if (instype.equals(FourCode.ASSIGN)) {  
        RunASSIGN(code);  
    }  
    if ((instype.equals(FourCode.PLUS)) || (instype.equals(FourCode.MINUS)) ||  
        (instype.equals(FourCode.MUL)) || (instype.equals(FourCode.DIV)) ||  
        (instype.equals(FourCode.GT)) || (instype.equals(FourCode.LT)) ||  
        (instype.equals(FourCode.GET)) || instype.equals(FourCode.LET)) ||  
        (instype.equals(FourCode.EQ)) || instype.equals(FourCode.NEQ)) {  
        RunOP(code, instype);  
    }  
}
```

JMP 就是改变 pc 的指数

```
//(JMP,null,null,destination)  
//(JMP,arg1,null,destination)  
//(JMP,arg1,arg2,destination)  
private void RunJMP(FourCode code) throws ConvertException {  
    if (code.getArg1() == null || symbolTable.getSymbolValue(code.getArg1()).getType() == Symbol.FALSE) { //需要跨  
        pc = getValue(code.getResult()).getInt();  
    }  
}
```

IN 代表进入作用域 mLevel++即可

```

// (in, null, null, null)
private void RunIN(FourCode code) {
    mLevel++;
}

```

OUT 退出作用域时，需要把符号表中该作用域的符号注销

```

// (out, null, null, null)
private void RunOUT(FourCode code) {
    symbolTable.deregister(mLevel);
    mLevel--;
}

```

READ 指令

```

// (read, null, null, Varname)
// (read, null, null, Varname[index])
private void RunREAD(FourCode code) throws ConvertException {
    // 输入指令
    Scanner sc = new Scanner(System.in);
    String input = sc.next();
    int type = symbolTable.getSymbolType(getId(code.getResult()));
    switch (type) {
        case Symbol.SINGLE_INT:
        case Symbol.ARRAY_INT:
        {
            Value value = parseValue(input);
            if (value.getType() == Symbol.SINGLE_INT) {
                setValue(code.getResult(), value);
            } else {
                throw new ConvertException("类型不匹配");
            }
            break;
        }
        case Symbol.SINGLE_Double:
        case Symbol.ARRAY_Double:
        {
            Value value = parseValue(input);
            setValue(code.getResult(), value);
            break;
        }
        case Symbol.TEMP: // impossible
        default:
            break;
    }
}

```

WRITE 指令


```

// (write, null, null, Exp)
private void RunWRITE(FourCode code) throws ConvertException {
    int index = -1;
    if (isArrayElement(code.getResult())) {
        index = getIndex(code.getResult());
    }
    System.out.println(symbolTable.getSymbolValue(code.getResult(), index));
}

```

声明指令

```

// Single声明没有赋值得到的四元式=>(int/double,null,null,Varname)
// Single声明之后赋值得到的四元式=>(int/double,value,null,Varname)
// Array声明得到的四元式=>(int/double,null,len,Varname)
private void RunDECLARE(FourCode code, String type) throws ConvertException {
    // int Declare
    if (type.equals(FourCode.INT)) {
        if (code.getArg2() != null) {
            Symbol symbol = new Symbol(code.getResult(), Symbol.ARRAY_INT, mLevel);
            symbol.getValue().initArray(getInt(code.getArg2()));
            symbolTable.register(symbol);
        } else {
            int intvalue = 0;
            if (code.getArg1() != null) {
                intvalue = getInt(code.getArg1());
            }
            Symbol symbol = new Symbol(code.getResult(), Symbol.SINGLE_INT, mLevel, intvalue);
            symbolTable.register(symbol);
        }
    }
    // double Declare
    if (type.equals(FourCode.DOUBLE)) {
        if (code.getArg2() != null) {
            Symbol symbol = new Symbol(code.getResult(), Symbol.ARRAY_Double, mLevel);
            symbol.getValue().initArray(getInt(code.getArg2()));
            symbolTable.register(symbol);
        } else {
            double doublevalue = 0;
            if (code.getArg1() != null) {
                doublevalue = getDouble(code.getArg1());
            }
            Symbol symbol = new Symbol(code.getResult(), Symbol.SINGLE_Double, mLevel, doublevalue);
            symbolTable.register(symbol);
        }
    }
}

```

ASSIGN 指令

```

// SingleASSIGN得到的四元式=>(assign,value,null,Varname)
// ArrayASSIGN得到的四元式=>(assign,value,null,Varname[index])
private void RunASSIGN(FourCode code) throws ConvertException {
    Value value = getValue(code.getArg1());
    setValue(code.getResult(), value);
}

```

对一些操作指令的解释执行

```

private void RunOP(FourCode code,String op) throws ConvertException, LexerException {
    switch (op) {
        //(+,arg1,arg2,result)
        case FourCode.PLUS:
            setValue(code.getResult(), getValue(code.getArg1()).PLUS(getValue(code.getArg2())));
            break;
        //(-,arg1,arg2,result)
        case FourCode.MINUS:
            if (code.getArg2() != null) {
                setValue(code.getResult(), getValue(code.getArg1()).MINUS(getValue(code.getArg2())));
            } else {
                setValue(code.getResult(), Value.NOT(getValue(code.getArg1())));
            }
            break;
        //(*,arg1,arg2,result)
        case FourCode.MUL:
            setValue(code.getResult(), getValue(code.getArg1()).MUL(getValue(code.getArg2())));
            break;
        //(/,arg1,arg2,result)
        case FourCode.GT:
            setValue(code.getResult(), getValue(code.getArg1()).GT(getValue(code.getArg2())));
            break;
        //(<=,arg1,arg2,result)
        case FourCode.LT:
            setValue(code.getResult(), getValue(code.getArg1()).LT(getValue(code.getArg2())));
            break;
        //(>=,arg1,arg2,result)
        case FourCode.GET:
            setValue(code.getResult(), getValue(code.getArg1()).GET(getValue(code.getArg2())));
            break;
        //(<=,arg1,arg2,result)
        case FourCode.LET:
            setValue(code.getResult(), getValue(code.getArg1()).LET(getValue(code.getArg2())));
            break;
        //(==,arg1,arg2,result)
        case FourCode.EQ:
            setValue(code.getResult(), getValue(code.getArg1()).EQ(getValue(code.getArg2())));
    }
}

```

其次因为之前的 FourCode 四个属性 op arg1 arg2 result 都是 String 属性的，所以需要一些小工具来解析本身不是 String 的参数。

```

//传入形如 xx[xx],获取其中的索引值
private int getIndex(String id) throws ConvertException {
    String indexstr = id.substring(id.indexOf("[") + 1, id.length() - 1) + "";
    return getInt(indexstr);
}

//传入一个字面值或者标识符,获取对应int值
private int getInt(String value) throws ConvertException {
    if (value.matches("^(-?\\d+)$")) {
        return Integer.parseInt(value);
    }
    Value valueint = symbolTable.getSymbolValue(value);
    if (valueint.getType() == Symbol.SINGLE_INT) {
        return valueint.getInt();
    } else {
        throw new ConvertException("不是整数");
    }
}

```

// 传入一个字面值或者标识符,获取对应double值

```
private double getDouble(String value) throws ConvertException {
    if (value.matches("(\\.?(\\d+)(\\.\\d+)?$")) {
        return Double.parseDouble(value);
    }
    Value valueint = symbolTable.getSymbolValue(value);
    return valueint.toDouble().getDouble();
}
```

// 传入形如xx[xx]或者xx 获取前面的id

```
private String getId(String id) {
    if (isArrayElement(id)) {
        return id.substring(0, id.indexOf("[") + 1); // prevent from memory leak
    }
    return id;
}
```

其余工具就不再列举

主体逻辑写好之后封装放到 Service 之中供给后续使用

```
public static void GetResult(String filePath) throws LexerException, IOException{
    RunFourCode.RunFromFile(filePath);
}
```

2.4.4 语义错误

* 1. 重复声明变量

```
public void register(Symbol symbol) throws ConvertException {
    for (int i=0; i<symbolList.size(); i++) {
        if (symbolList.get(i).getName().equals(symbol.getName())) {
            if (symbolList.get(i).getLevel() < symbol.getLevel()) {
                symbol.setNext(symbolList.get(i));
                symbolList.set(i, symbol);
                return;
            } else {
                throw new ConvertException("变量 < " + symbol.getName() + "> 重复声明");
            }
        }
    }
    symbolList.add(symbol);
}
```

* 2. 使用未声明的变量

```
if (name.startsWith(TEMP_PREFIX)) {
    Symbol s = new Symbol(name, Symbol.TEMP, -1);
    tempNames.add(s);
    return s;
}
throw new ConvertException("变量 < " + name + "> 不存在");
}
```

...

* 3. 变量和赋值类型不符合

单值

```
Value value = parseValue(input);
if (value.getType() == Symbol.SINGLE_INT) {
    setValue(code.getResult(), value);
} else {
    throw new ConvertException("类型不匹配");
}
break;
```

数组

```
private void setValue(String id, Value value) throws ConvertException {
    int index = -1;
    if (isArrayElement(id)) {
        index = getIndex(id);
    }
    int type = symbolTable.getSymbolType(getId(id));
    switch (type) {
        case Symbol.SINGLE_INT:
        case Symbol.SINGLE_Double:
        {
            if (type == Symbol.SINGLE_Double) {
                symbolTable.setSymbolValue(getId(id), value.toDouble());
            } else {
                if (value.getType() == Symbol.SINGLE_Double) {
                    throw new ConvertException("表达式 <" + id + "> 与变量类型不匹配");
                } else {
                    symbolTable.setSymbolValue(getId(id), value);
                }
            }
            break;
        }
        case Symbol.ARRAY_INT:
        case Symbol.ARRAY_Double:
        {
            if (symbolTable.getSymbolValue(getId(id), index).getType() == Symbol.SINGLE_Double) {
                symbolTable.setSymbolValue(getId(id), value.toDouble().getDouble(), index);
            } else {
                if (value.getType() == Symbol.SINGLE_Double) {
                    throw new ConvertException("表达式 <" + id + "> 与变量类型不匹配");
                } else {
                    symbolTable.setSymbolValue(getId(id), value.getInt(), index);
                }
            }
        }
    }
}
```

* 4. 数组越界

```
public void setSymbolValue(String name, int value, int index) throws ConvertException {
    if (getSymbol(name).getValue().getArrayInt().length > index) {
        getSymbol(name).getValue().getArrayInt()[index] = value;
    } else {
        throw new ConvertException("数组 <" + name + "> 下标 " + index + " 越界");
    }
}
```

```

9  */
10 public Value getSymbolValue(String name, int index) throws ConvertException {
11     Symbol s = getSymbol(name);
12     if (index == -1) { // 单值
13         return s.getValue();
14     } else {
15         if (s.getValue().getArrayInt().length < index + 1) {
16             throw new ConvertException("数组 < " + name + "> 下标 " + index + " 越界");
17         }
18         if (s.getType() == Symbol.ARRAY_INT) {
19             Value rv = new Value(Symbol.SINGLE_INT);
20             rv.setInt(s.getValue().getArrayInt()[index]);
21             return rv;
22         } else {
23             Value rv = new Value(Symbol.SINGLE_Double);
24             rv.setDouble(s.getValue().getArrayDouble()[index]);
25             return rv;
26         }
27     }
28 }
29

```

* 5. 除 0

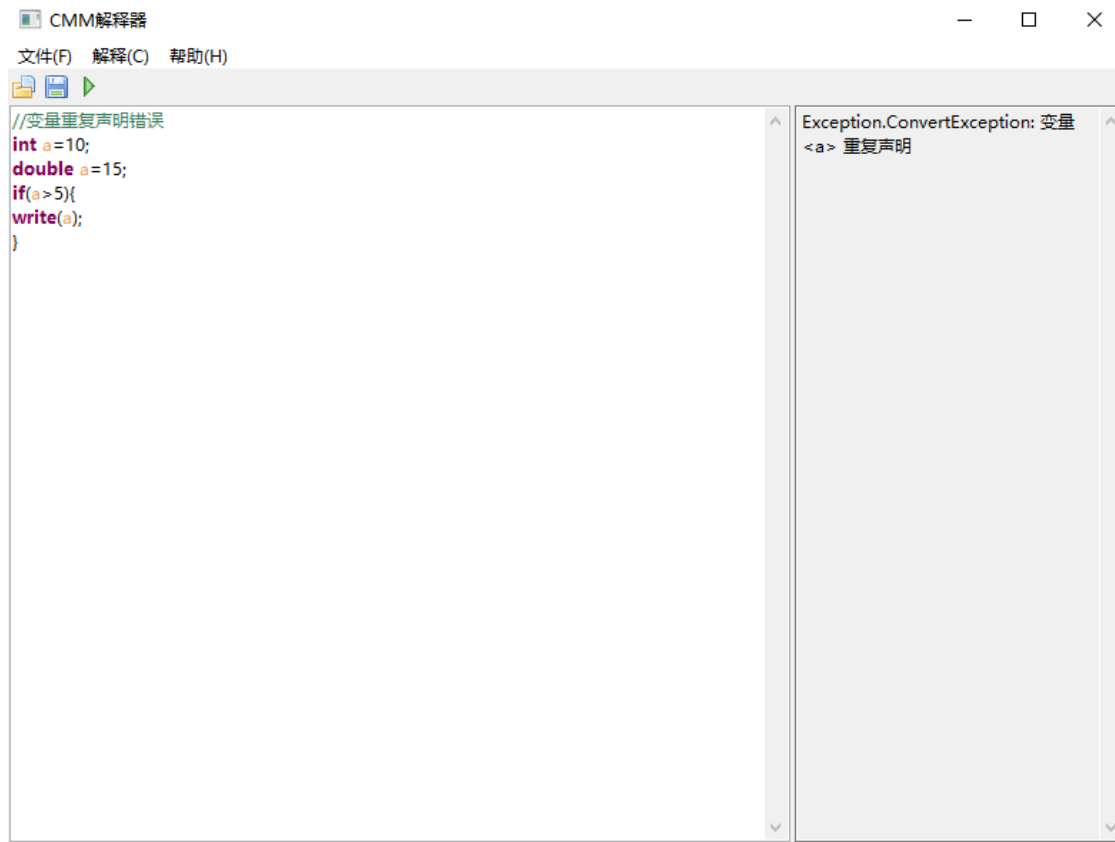
```

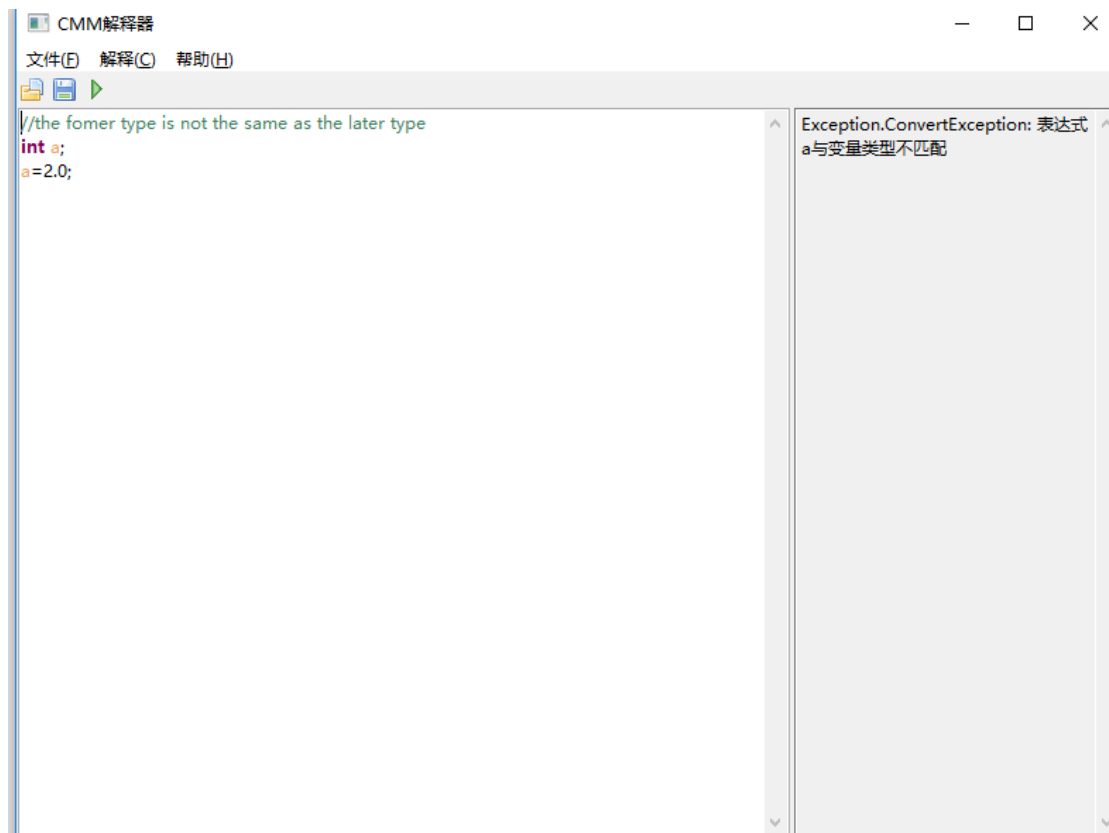
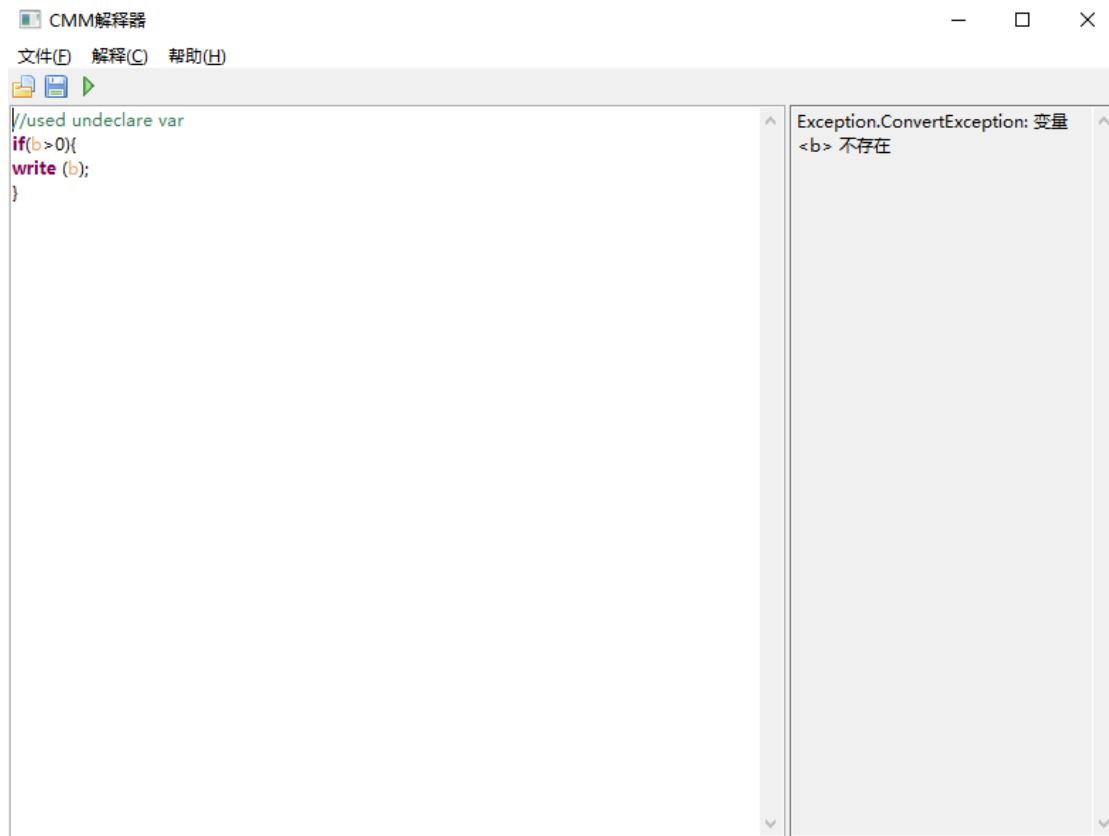
public Value DIV(Value value) throws ConvertException {
    if (this.mType == Symbol.SINGLE_Double) {
        Value rv = new Value(Symbol.SINGLE_Double);
        if (value.mType == Symbol.SINGLE_INT) {
            if (value.getInt() == 0) {
                throw new ConvertException("不能除0");
            }
            rv.setDouble(this.mDouble / value.mInt);
            return rv;
        } else if (value.mType == Symbol.SINGLE_Double) {
            if (value.getDouble() == 0) {
                throw new ConvertException("不能除0");
            }
            rv.setDouble(this.mDouble / value.mDouble);
            return rv;
        }
    } else if (this.mType == Symbol.SINGLE_INT) {
        if (value.mType == Symbol.SINGLE_INT) {
            if (value.getInt() == 0) {
                throw new ConvertException("不能除0");
            }
            Value rv = new Value(Symbol.SINGLE_INT);
            rv.setInt(this.mInt / value.mInt);
            return rv;
        } else if (value.mType == Symbol.SINGLE_Double) {
            if (value.getDouble() == 0) {
                throw new ConvertException("不能除0");
            }
            Value rv = new Value(Symbol.SINGLE_Double);
            rv.setDouble(this.mInt / value.mDouble);
            return rv;
        }
    }
}

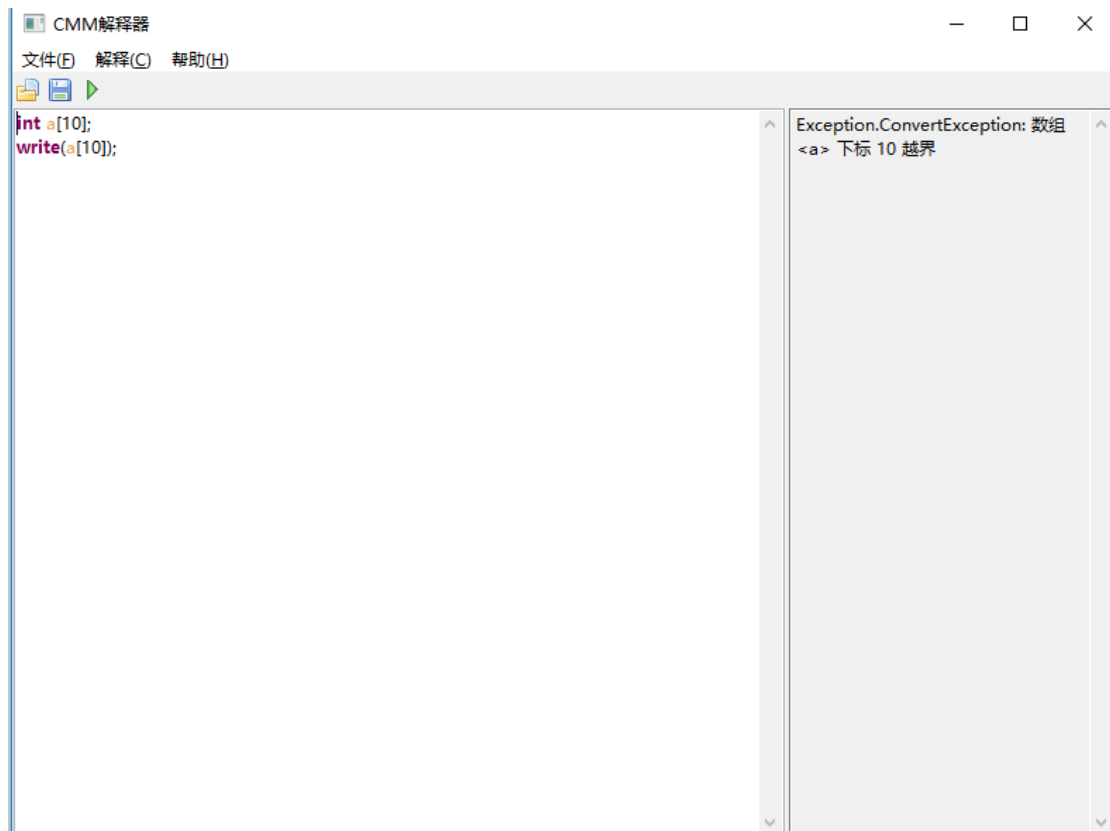
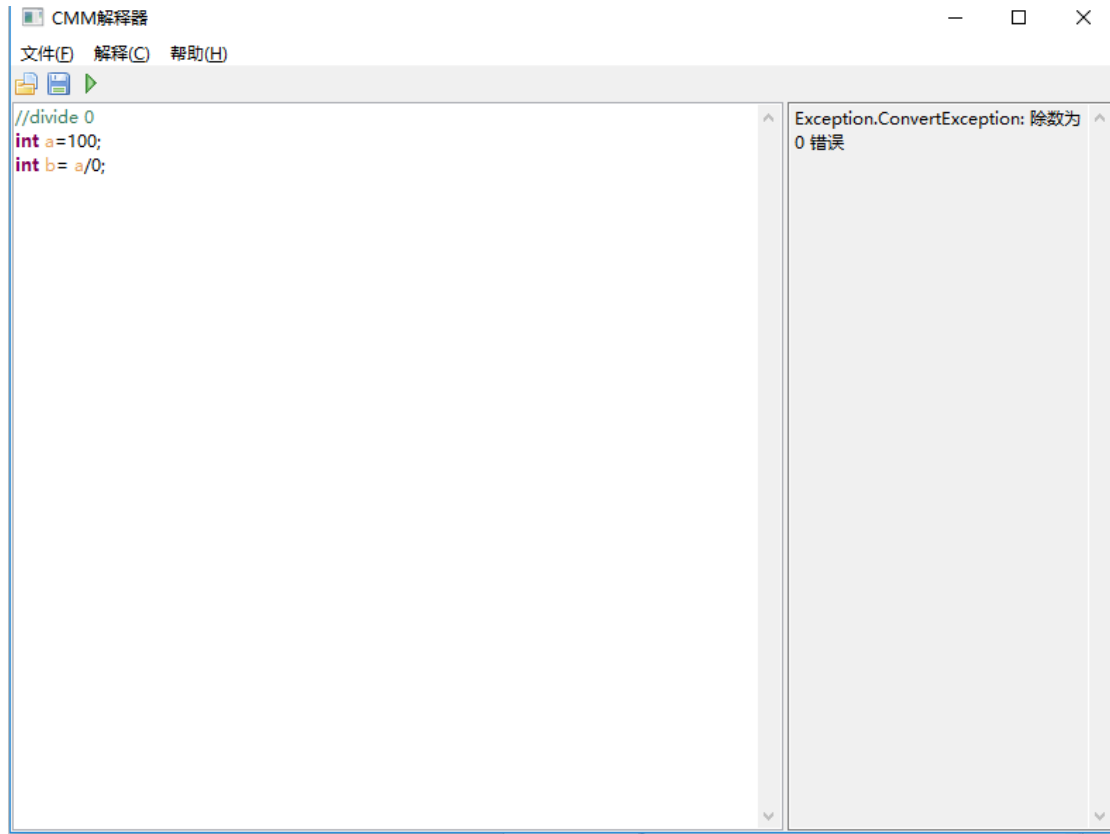
```

2.4.5 测试

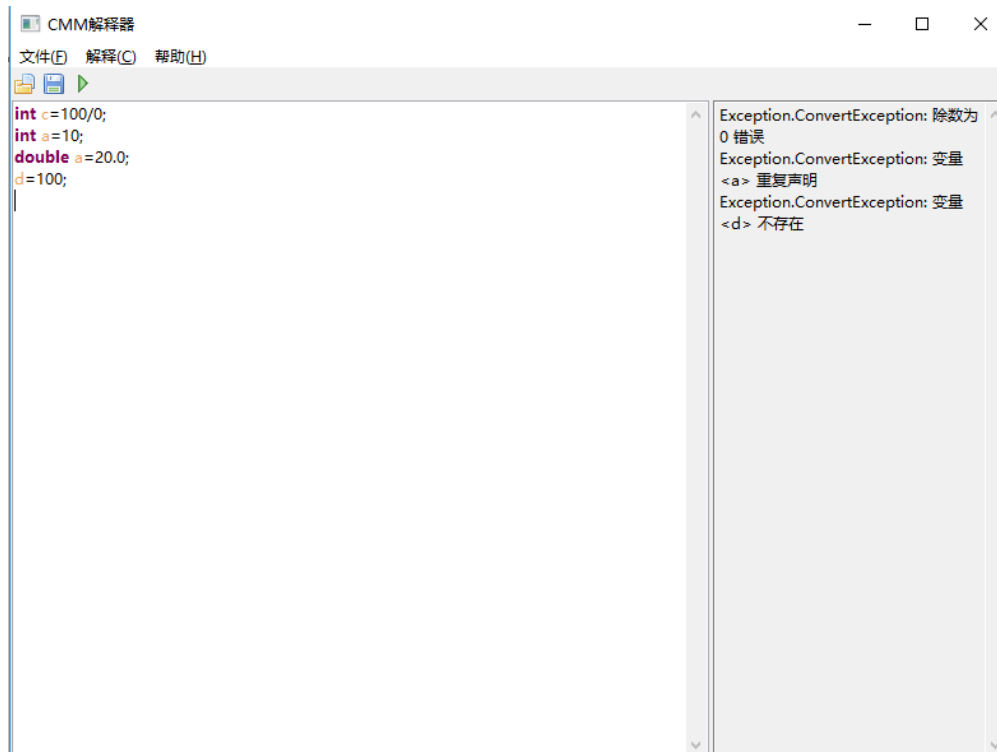
异常情况



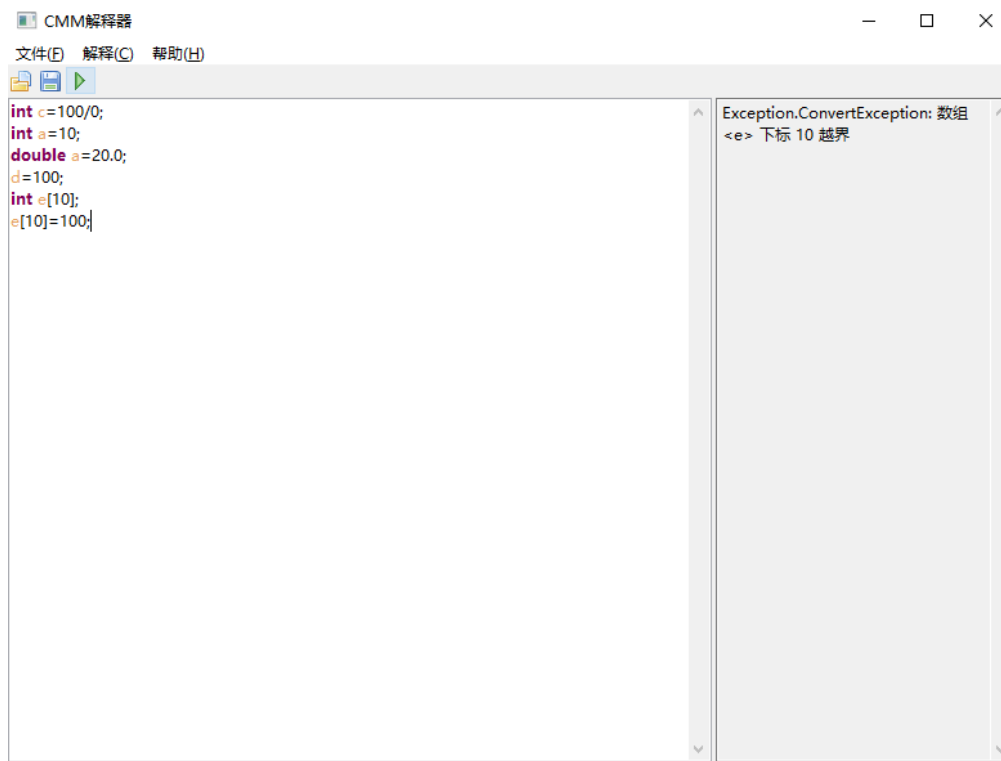




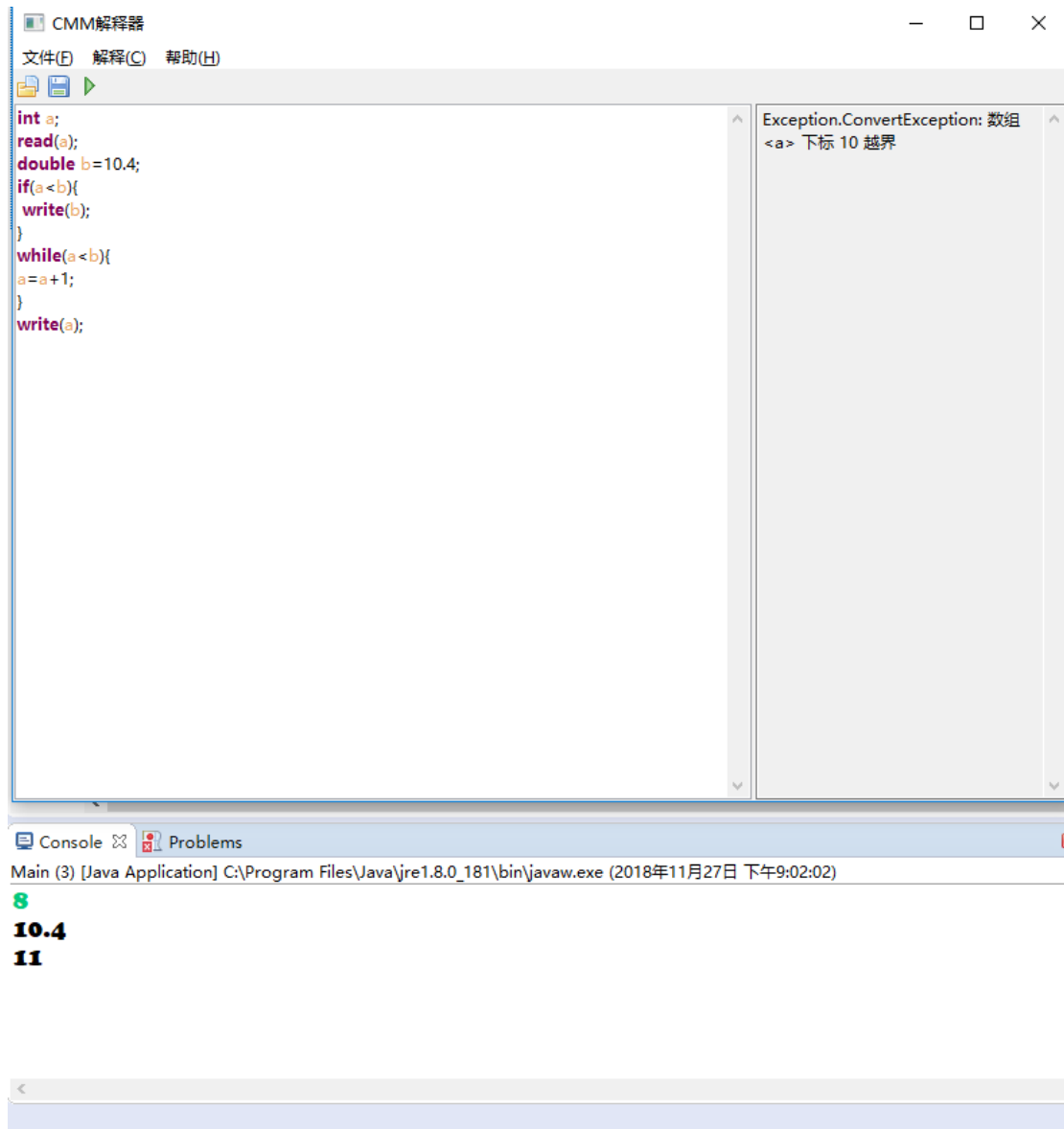
同时报告多个错误 部分错误是采用链表存储的，另外一部分运行时的错误必须使用 throw 抛出异常情况



对于数组越界和类型不匹配的错误，这两种错误是进行 throw 的，解释器会优先提示这些错误。改正这些错误之后，就会显示其他错误。



正常情况



2.5 Javacc 的使用

2.5.1 文法说明

Javcc 的文法大致和之前实验之中的 CMM 相同, 额外的部分就是我添加了 For 循环

```
void Statement(): { }  
{  
    IfStmt()  
    | WhileStmt()  
    | DoWhileStmt()  
    | ForStmt()  
    | DeclareStmt()  
    | AssignStmt() ";"  
    | BlockStmt()  
    | ReadStmt()  
    | WriteStmt()  
}
```

For 循环的文法就是

```
For(declareStmt or AssignStmt; Expression; AssignStmt)  
Statement
```

其他部分和前面一样, 在此就不再赘述。

2.5.2 词法部分

在 Javacc 的词法部分, 我们需要告诉 Javacc 我们在文法之中会出现那些 token, 这样他就会自动帮我们生成一个对应的词法分析器, 可以说是及其方便了。根据文法, 我列出来的 token 有这些种类

首先是文法之中会忽略跳过的字符：这里有如下几种
空格符、换行符、注释等等

```
⊖ SKIP :/*跳过的字符*/  
{  
  "  
  | "\t"  
  | "\n"  
  | "\r"  
⊖ | < "//" (~ [ "\n", "\r" ] ) *  
  (  
    "\n"  
    | "\r"  
    | "\r\n"  
  ) >  
⊖ | < "/" (~ [ "*" ] ) * "<br>"  
  (  
    - [ "/" ] (~ [ "*" ] ) * "<br>"  
  ) *  
  "/" >  
}
```

其次就是对数字的定义，数字有整数 int 和实数 double 两类

```
⊖ TOKEN :/* LITERALS */  
{  
  < CONSTANT :<INTEGER_LITERAL> | <DOUBLE_LITERAL> >  
  //数字  
  | <#DIGIT: ["0"-"9"] >  
  //整数  
  | <INTEGER_LITERAL: ["1"-"9"] (<DIGIT>)* >  
  //实数  
⊖ | <DOUBLE_LITERAL: (<DIGIT>)+  
  | (<DIGIT>)+ "."  
  | (<DIGIT>)+ "." (<DIGIT>)+ |  
  | "." (<DIGIT>)+ >  
}
```

然后就是一些文法之中会用到的保留字

④ **TOKEN** :/*定义关键字*/

```
{  
  <IF : "if">  
  | <ELSE : "else">  
  | <WHILE : "while">  
  | <DO : "do">  
  | <FOR : "for">  
  | <READ : "read">  
  | <WRITE : "write">  
  | <INT : "int">  
  | <DOUBLE : "double">  
  | <VOID : "void">  
  | <RETURN : "return">  
}
```

⑤ **TOKEN** :/*操作符*/

操作符

⑥ **TOKEN** :/*操作符*/

```
{  
  <PLUS : "+">  
  | <MINUS : "-">  
  | <MUL : "*">  
  | <DIV : "/">  
  | <ASSIGN : "=">  
  | <LT : "<">  
  | <GT : ">">  
  | <LET : "<=">  
  | <GET : ">=">  
  | <EQ : "==">  
  | <NEQ : "<>">  
}
```

一些符号

```

TOKEN : /* RELATIONSHIP OPERATOR */
{
    <LPS:"(">
    | <RPS:")">
    | <COMMA:",">
    | <SEMI: ";">
    | <LBRACE:"{">
    | <RBRACE:"}">
    | <LBRACKET:"[">
    | <RBRACKET:"]">
    | <SINQS:""">
    | <DOUQS:"\">
    | <ADDR:"&">
}

```

标识符

```

TOKEN : /*定义标识符*/
{
    <IDENTIFIERS: ["a"-"z", "A"-"Z", "_"] (["a"-"z", "A"-"Z", "_", "0"-"9"])*>
}

```

在这里我还遇到了一些问题，之前把保留字放在了 IDENTIFIERS 之后就出现了识别错误的 BUG，在网上搜寻了很久之后才发现是需要把保留字放在 IDENTIFIERS 之前才能够准确是别的。

文法部分就结束了。

2.5.3 语法语义部分

语法部分我们只需要把文法转换了 Javacc 对应的表示放到里面就好了。

这里我们需要的就是 jjt 文件了，之前一直用的 jj 文件一直没找到怎么使用之后生成的语法树，查阅之后发现新建 jjt 文件就好了。

要记住这样一句话：“JavaCC 把文法识别完全地做到了函数过程中”，这个思想将贯穿在下面的代码之中。

首先第一个函数是 start() 函数，这是我们整个程序的“切入点”

```

/*语法规则*/
SimpleNode start(): { }
{
    procedure()
    <EOF>
    {
        return jjtThis;
    }
}

void procedure(): { }
{
    (Statement()*
}

```

Statement 分支

```

void Statement(): { }
{
    IfStmt()
    | WhileStmt()
    | DoWhileStmt()
    | ForStmt()
    | DeclareStmt()
    | AssignStmt() ";"
    | BlockStmt()
    | ReadStmt()
    | WriteStmt()
}

```

```

IfStmt
If(expression)
Statement
    (else Statement)

```

```

//循环
void IfStmt(): {}
{
    < IF>< LPS>expression()< RPS>Statement()
    (
        < ELSE > Statement()
    )?
}

WhileStmt
While(expression)
Statement

void WhileStmt(): {}
{
    <WHILE><LPS>expression()<RPS>Statement()
}

DoWhileStmt
Do
Statement
While(expression)

void DoWhileStmt(): {}
{
    <DO>Statement()<WHILE><LPS>expression()<RPS>
}

ForStmt
For(declareStmt or AssignStmt;Expression;AssignStmt)
Statement

void ForStmt(): {}
{
    <FOR><LPS>ForInitStmt()";"expression()";"AssignStmt()<RPS>
    Statement()
}

void WriteStmt(): {}
{
    <WRITE><LPS>expression()<RPS>";"
}

```

上面的这几个之中都不需要加入语义分析的代码

下面的这几个 Stmt 就需要加入语义分析代码了
首先是符号和符号表的构造

Symbol

```
/**
 * @author 作者：朱江源
 * Data: 创建日期：2018年12月25日
 * Time: 创建时间：下午4:14:45
 * Declaration: All Rights Reserved
 */

public class Symbol {
    private String name;
    private String value;
    private int level;
    private int type;

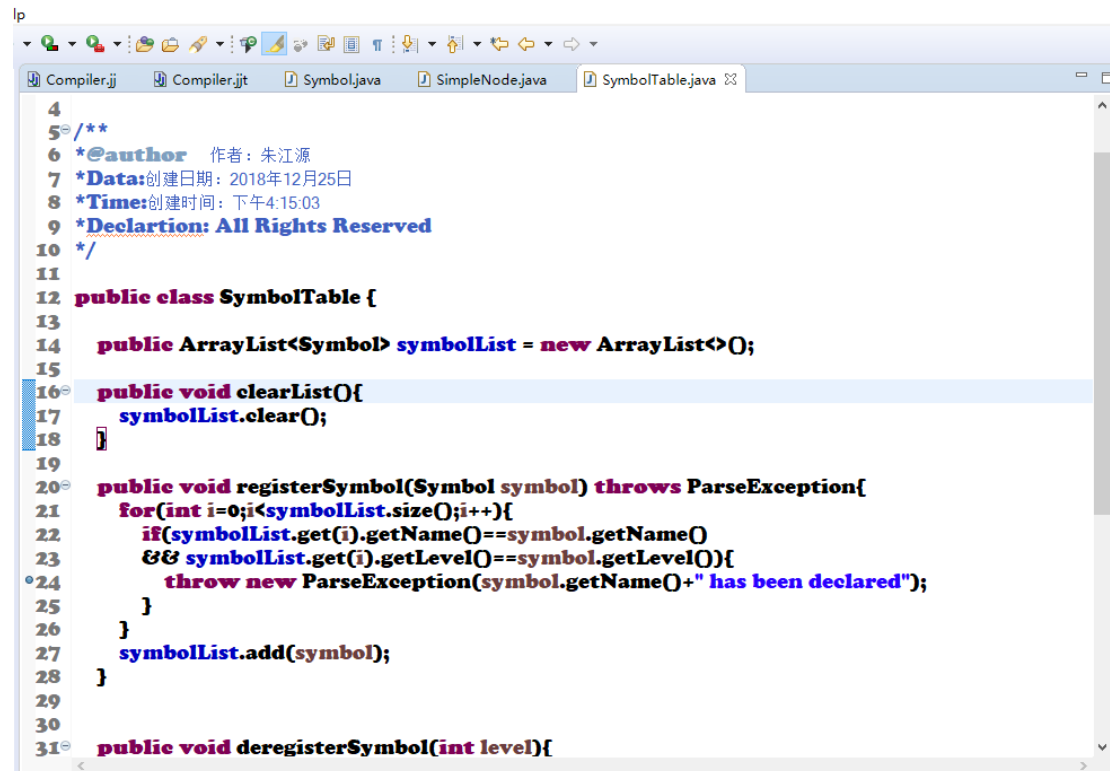
    public Symbol(int level, String name) {
        this.name = name;
        this.level = level;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getValue() {
        return value;
    }
}
```

SymbolTable



```
4
5 /**
6  * @author 作者: 朱江源
7  * @Data: 创建日期: 2018年12月25日
8  * @Time: 创建时间: 下午4:15:03
9  * @Declaration: All Rights Reserved
10 */
11
12 public class SymbolTable {
13
14     public ArrayList<Symbol> symbolList = new ArrayList<>();
15
16     public void clearList(){
17         symbolList.clear();
18     }
19
20     public void registerSymbol(Symbol symbol) throws ParseException{
21         for(int i=0;i<symbolList.size();i++){
22             if(symbolList.get(i).getName()==symbol.getName()
23                 && symbolList.get(i).getLevel()==symbol.getLevel()){
24                 throw new ParseException(symbol.getName()+" has been declared");
25             }
26         }
27         symbolList.add(symbol);
28     }
29
30
31     public void deregisterSymbol(int level){
```

符号和符号表用来记录程序之中的一些变量的信息，这个之前介绍过，这里也就不再赘述。

之后就是在 javacc 之中加入相应的语义处理代码就好了

DeclareStmt 需要给注册符号，保存符号的信息，检查是否重复声明等工作

```

}

void DeclareStmt():{
    Token tk;
    String name;
    Symbol tmp;
    int type;
    String value;}
{
    (type=type()tk=< IDENTIFIERS>
    {
        name = tk.image;
        tmp = new Symbol(level,name);
        tmp.setType(type);
        table.registerSymbol(tmp);
    }(<ASSIGN> ( value=expression() {table.symbolList.get(table.symbolList.size()-1).setValue(v
    tk=< IDENTIFIERS >
    {
        <IDENTIFIERS:([a-z,A-Z,_,])([a-z,A-Z,_,0-9])*>
        tmp = new Symbol(level,name);
        tmp.setType(type);
        table.registerSymbol(tmp);
    }(< ASSIGN> ( value=expression() {table.symbolList.get(table.symbolList.size()-1).setValue
    ";"}
}

```

AssignStmt 需要给从符号表中取得符号并且赋值，需要报未声明的错

```

void AssignStmt(): {
    Token tk;
    Symbol tmp;
    String value; }
{
    tk=< IDENTIFIERS> {tmp=table.getSymbol(tk.image);}
    < ASSIGN> value=expression() {tmp.setValue(value);}";"
}

```

报错功能都在符号表之中实现

```

public void registerSymbol(Symbol symbol) throws ParseException{
    for(int i=0;i<symbolList.size();i++){
        if(symbolList.get(i).getName()==symbol.getName()
        && symbolList.get(i).getLevel()==symbol.getLevel()){
            throw new ParseException(symbol.getName()+" has been declared");
        }
    }
    symbolList.add(symbol);
}

public void deregisterSymbol(int level){
    while(!symbolList.isEmpty()&&symbolList.get(symbolList.size()-1).getLevel()==level)
        symbolList.remove(symbolList.size()-1);
}

public Symbol getSymbol(String name) throws ParseException{
    for(int i=symbolList.size()-1;i>=0;i--){
        Symbol tmp = symbolList.get(i);
        if(tmp.getName().equals(name))
            return tmp;
    }
    throw new ParseException(name+" has not been declared");
}

```

BlockStmt

在这需要 level++来实现作用域的管理

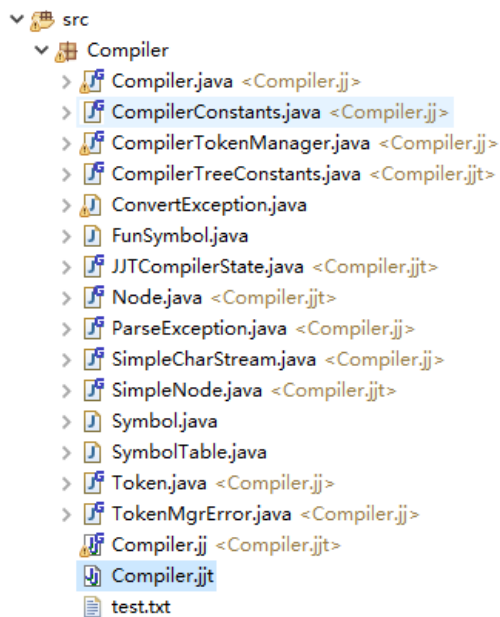
```

void BlockStmt(): {}
{
    <LBRACE>[level++;](Statement()*<RBRACE>
    {
        table.deregisterSymbol(level);
        level--;
    }
}

```

2.5.4 自动生成的代码

在写好 jjt 和需要的 java 文件之后，在 eclipse 之中右键点击 jjt 文件会有一个 compile with javacc 点击之后就会生成词法分析，语法分析的代码了



```

203     tmp = new Symbl
204     tmp.setType(ty
205     table.registerSy
206 }(<ASSIGN>(valu
207 tk=< IDENTIFIER
208 {
209     name = tk.ima
210     tmp = new Sy
211     tmp.setType(
212     table.register
213 }(< ASSIGN> ( va
214 ";")
215
216 void AssignStmtC
217 Token tk;
218 Symbol tmp;
219 String value; }
220 {
221     tk=< IDENTIFIE

```

2.5.5 测试

测试用例

```

1 int a;
2 a=10;
3 double b;
4 while(a<13)
5 {
6 a=a+1;
7 }
8 for(int i=0;i<3;i=i+1)
9 {
10 a=a+1;
11 }
12 read(b);
13 if(a<20)
14 {
15 write(a);
16 }
17 else
18 {
19 write(b);
20 }

```

测试输出结果

start
procedure
Statement
DeclareStmt
type
Statement
AssignStmt
expression
polynomial
term
factor
Statement
DeclareStmt
type
Statement
WhileStmt
expression
polynomial
term
factor
polynomial
term
factor
Statement
BlockStmt
Statement
AssignStmt
expression
polynomial
term
factor
polynomial
term
factor
Statement
ForStmt

Updates Available

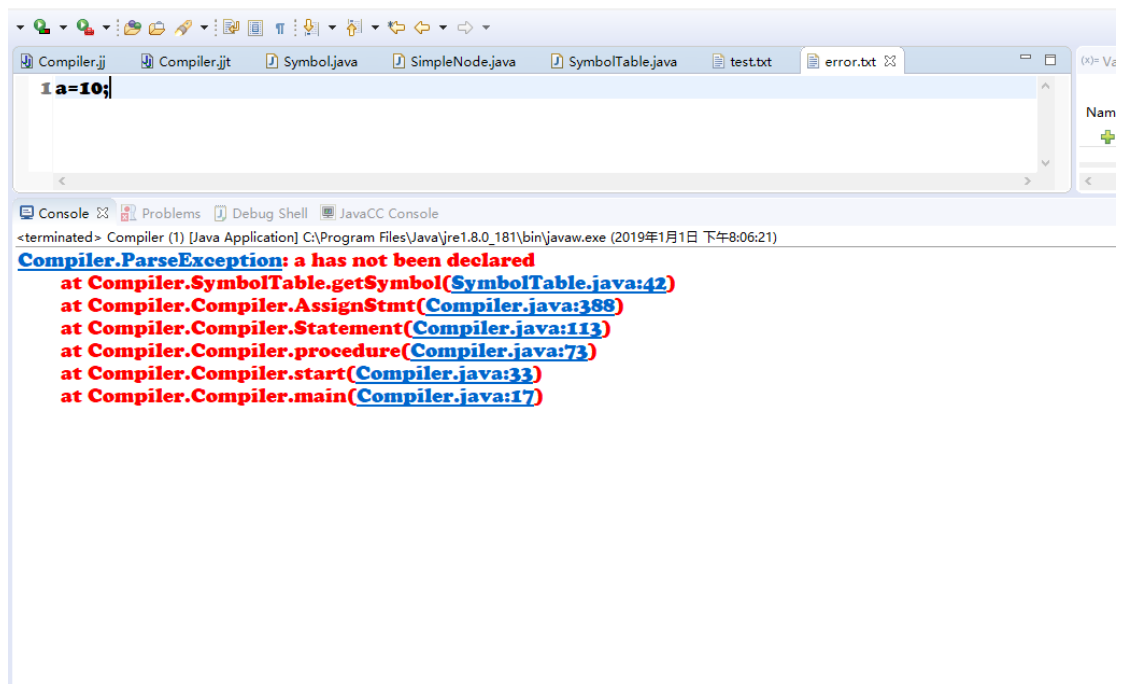
Updates are available for your software.
Click to review and install updates.

Statement
AssignStmt
expression
polynomial
term
factor
polynomial
term
factor
Statement
ReadStmt
Statement
IfStmt
expression
polynomial
term
factor
polynomial
term
factor
Statement
BlockStmt
Statement
WriteStmt
expression
polynomial
term
factor
Statement
BlockStmt
Statement
WriteStmt
expression
polynomial
term
factor

Updates Available

Updates are available for your software.
Click to review and install updates.

错误处理测试用例 使用未声明的变量



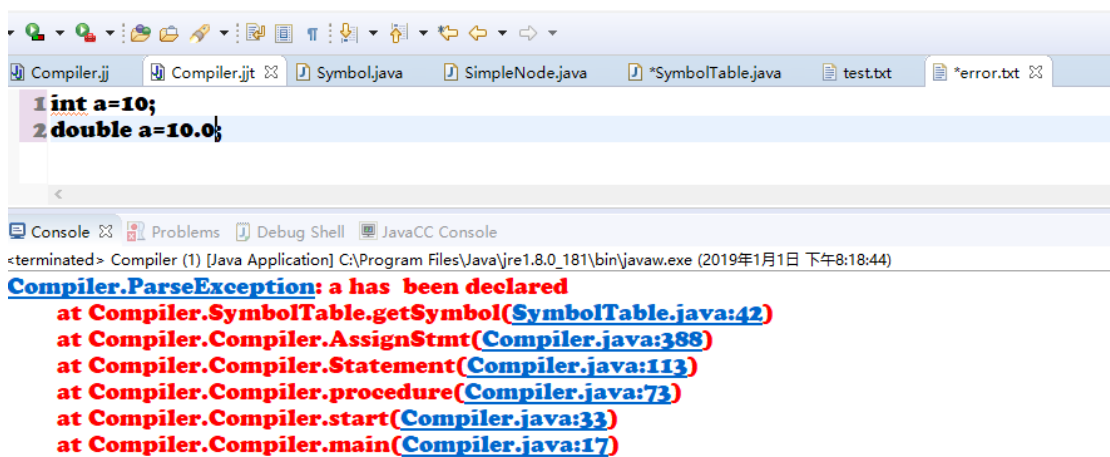
```
1 a=10;
```

Console <terminated> Compiler (1) [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (2019年1月1日 下午8:06:21)

Compiler.ParseException: a has not been declared

- at Compiler.SymbolTable.getSymbol(SymbolTable.java:42)
- at Compiler.Compiler.AssignStmt(Compiler.java:388)
- at Compiler.Compiler.Statement(Compiler.java:113)
- at Compiler.Compiler.procedure(Compiler.java:73)
- at Compiler.Compiler.start(Compiler.java:33)
- at Compiler.Compiler.main(Compiler.java:17)

重复申明



```
1 int a=10;  
2 double a=10.0;
```

Console <terminated> Compiler (1) [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (2019年1月1日 下午8:18:44)

Compiler.ParseException: a has been declared

- at Compiler.SymbolTable.getSymbol(SymbolTable.java:42)
- at Compiler.Compiler.AssignStmt(Compiler.java:388)
- at Compiler.Compiler.Statement(Compiler.java:113)
- at Compiler.Compiler.procedure(Compiler.java:73)
- at Compiler.Compiler.start(Compiler.java:33)
- at Compiler.Compiler.main(Compiler.java:17)

2.5.6 javacc 使用心得体会

当一开始下手的时候，实在是有些困难，网上也没有什么太多的详细的资料。后来从 javacc 官网，github, CSDN 上面看了几份 javacc 的附带讲解的代码之后才弄懂了 javacc 的工作原理才懂得了怎样使用 javacc 去快速的构建一个词法分析器和语法分析器。实验过程之中碰到了很多困难，例如 javacc 的一些语法问题，刚刚开始读的时候一头雾水，根本看不懂那些代码就是是在干嘛，慢慢的才知道能够在 javacc 之中插入 java 代码执行从而达到一些语义分析的效果。对于我目前的这个使用 JAVACC，为 CMM 语言构造一个编译器，所能识别的语句和类型都还太少，都只是最基本的简单的语句（如变量的声明，变量的赋值，read 语句，write 语句，if-else 语句，while 语句），对于其他方面的功能，以后空闲的时候在抽空研究研究，感觉这对于帮助我理解编译器底层实现原理有很大的帮助！总体来说，这个工具真的很好用，简洁易懂，开发一个分析器只需要十几分钟，这次实验受益匪浅。

3 结论

这学期的解释器实践构造是上学期编译原理的拓展学习实践，大体来看从词法分析，语法分析，语义分析三个方面进行了实际的解释器项目开发。第一次实验之中的总体框架构建我觉得还是特别重要的。在真正上手做项目之前去查阅了大量的资料，对于三个阶段的工作有了答题的了解并且确定了其中选用的分析方法，为后面的项目编写提供了一定的指导意义。

首先就词法分析来说，词法分析决定了该程序接受什么样的输入，这就是这个编译器的输入规则，很多根本的东西都是在这里确定的，比如说保留字，数组，变量符命名规则都是在这个地方定下来的。我们从源码之中读取到的字符串序列在经过过滤无用符号和注释之后将一个单词编程一个含有单词信息的 TokenList 传输到后面以供后续的分析使用。

然后就是语法分析阶段了，这个阶段就定义了所编写的语言到底有怎样的文法特征。你能根据自己想要实现的语言来构建相应的文法，然后就根据文法来进行分析程序的构建了。大体来说，我们在上学期的编译原理之中学习了两种分析方法：LL 和 LR。这两种方法都是可行的，在该项目之中，我觉得 LR 在有了预测分析表之后很好写程序，便尝试去构建预测分析表。但是发现构建这个文法的预测分析表并没有那么容易，我就转而采用了 LL 递归下降子程序来分析。LL 来说，难度也不算很大，只要理清楚文法脉络以及其中的各种 First、Follow 和 Select 集合之后就比较容易来写程序了。我认为语法分析的难点在于语法树的构建，语法树的节点很容易构建，难点在于你需要在语法书之中怎样保存原始语句在接下来的分析之中能够用到的信息，我大部分的时间都是花费在了这个地方。具体的语法树的结构可以参看本文的 2.3 部分。语法树构建之后，就能对语法树进行相应的语义分析了。

语义分析阶段的话，有两种实现方式，一种是直接针对树的结构直接进行分析给出结果，另外一种则是先根据语法树生成相应的中间代码，然后根据中间代码进行执行然后给出结果。因为前者不够直观，我就选用了先生成中间代码，然后针对中间代码执行给出结果的方案。恰好这学期学习了系统级，通过系统级的学习，我对底层的代码执行原理也有了一定的了解。对于各种各样的循环结果，不过是利用 JUMP 指令进行跳转执行而已。所以对于 If、While、DoWhile 结构，只要设计出不同的 JUMP 进行相应的跳转就能够实现对应的功能了，具体可以参见 2.4 之中的设计。这个阶段还有一个重要的地方就是符号保留以及对作用域的实现，我们在进入一个 block 之后需要能够进行对于外部符号的访问，并且能够声明一个同名符号覆盖外面的符号。对于符号的保留，我采用了 Symbol 进行变量的存储，而 Symbol 又保存在 SymbolTable 之中。Symbol 都具有一个 Level 表明他所在的作用域，这样子我们就能实现作用域的功能了。

然后的话就是对于中间代码的执行了。中间代码的实现就很容易了，In 代表了进入一个作用域，Out 代表了出一个作用域，在退出一个作用域的时候，我们应该对该作用域的 Symbol 进行清楚。然后 Assign 和 Declare 就是将信息保留进 SymbolTable 之中就可以了。JUMP 指令就是改变下一条执行语句的 PC 就可以了。

总体来说，通过这次解释器构造实践课程的学习，我对于解释器的工作原理的了解和以前来说可以是完全不一样了，通过实际上手构建一个解释器，一步一步的摸索原理，并且编程实现功能，很有成就也学到了很多。

最后特别感谢李蓉蓉老师在本学期为我们提供的帮助，李老师在平时群里发的一些文章对于项目的实现以及原理的理解帮助都很大，而且平时也很乐于解答同学们的疑问，在验收的时候也提出了一些很有启发性的提问，对于我在项目之中的开发有着很大的启发作用，给了我很大的帮助，在此特别感谢李老师。

4.参考文献

词法分析器实现

<https://www.cnblogs.com/zyrblog/p/6885922.html>

词法分析器 (Lexer)

<https://www.cnblogs.com/attilax/p/5423472.html>

第七章——语义分析和中间代码产生

<https://blog.csdn.net/analogyy/article/details/80564094>

【编译原理】词法分析 (一)

https://www.baidu.com/link?url=ChgHjcpCNB75L3ryv8ZEQq_lrFit5V_eC9jk-v4eUxDtLwWXQr0z3XmciPbbUXREhb0S1r-ZY107-LXmHwjQnbtoYI9IPwVQ8hXC0bLDLXi&wd=&eqid=81c9e65e0001c528000000035c04e938

938

语法分析 (四)

https://www.baidu.com/link?url=lTnoA5kCCqCXEk4sPupBKXvuMUchd4SbrclKhA7yFZR2nj1K_m29sYxIOLMQdUWsUoa011E5jS7zBbqsh8hHAjYkpiY-nuGnKqjuNB04bZm&wd=&eqid=f721a86b00018eb1000000035c04e959

简单编译器之语法分析

<https://www.cnblogs.com/sorheart/p/3250944.html>

语义分析和中间代码生成

<https://blog.csdn.net/yongchaocsdn/article/details/79056504>

编译原理——中间代码生成

https://blog.csdn.net/qq_24421591/article/details/50283093

语言解释器构造实践 (一) ——前言

<https://blog.csdn.net/TomMMRunNEr/article/details/78164796>

编译原理 (清华大学出版社)

Javacc 学习笔记

<https://blog.csdn.net/u011775183/article/details/39788231>

教师评语评分

评语： _____

评分： _____

评阅人：

年 月 日

（备注： 对该实验报告给予优点和不足的评价，并给出百分之评分。）