编号：

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实习 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 九 | 十 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |  |  |

**武汉大学计算机学院**

**本科生实验报告**

**《解释器构造》实验**

**Mini语言词法语法分析器构造实验**

编 号：

实习题目：

专业（班）：

团队成员一： XXX（20XXXXXXXXX）

团队成员二： XXX（20XXXXXXXXX）

团队成员三： XXX（20XXXXXXXXX）

指导教师： 杜　卓　敏

２０18 年 11 月 日

**郑 重 声 明**

本团队呈交的实验报告，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。

团队成员签名： 日期：

**词法分析目录**

**第一部分 Mini语言形式化描述 ……………………… 1**

**第二部分 单词编码表 ………………………**

**第三部分 状态转换图 ………………………**

**第四部分 词法分析算法 ………………………**

**第五部分 主要数据结构 ………………………**

**第六部分 测试部分 ………………………**

**第一部分 Mini语言形式化描述**

Mini语言……。。。。。。。。。。。。。。。。

（宋体小4，正文行间距固定为23磅，字符间距为标准）

**第二部分 单词编码表**

ZZZ。

**第三部分 状态转换图**

ZZZ。

**第四部分 词法分析算法**

通过文本文件读入一个String，通过toCharArray变成char序列，创建一个currentChar变量指向char序列的第一个，创建lineNo变量保存行数，便于报错（每次遇到\n就令其+1）。反复读入char，根据自动机生成，能识别则生成对应的token，添加到token序列中，忽略\r,\n,\f,\t,空格，碰到不能识别的单词就报非法字符错误，不进行语法分析。自动机的关键代码如下

**public static void** makeTokenList() {  
 *readChar*();  
 **while** (**true**) {  
 **if** (*currentChar* == **';'**) {  
 *tokens*.add(**new** Token(Token.***SEMI***, **";"**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **','**) {  
 *tokens*.add(**new** Token(Token.***COMMA***, **","**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **'+'**) {  
 *readChar*();  
 **if** (*currentChar* == **'+'**) {  
 *tokens*.add(**new** Token(Token.***PLUSPLUS***, **"++"**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **'='**) {  
 *tokens*.add(**new** Token(Token.***PLUSEQUAL***, **"+="**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else** {  
 *tokens*.add(**new** Token(Token.***PLUS***, **"+"**, *lineNo*));  
 *//readChar();* **continue**;  
 }  
 } **else if** (*currentChar* == **'-'**) {  
 *readChar*();  
 **if** (*currentChar* == **'-'**) {  
 *tokens*.add(**new** Token(Token.***MINUSMINUS***, **"--"**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **'='**) {  
 *tokens*.add(**new** Token(Token.***MINUSEQUAL***, **"-="**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else** {  
 *tokens*.add(**new** Token(Token.***MINUS***, **"-"**, *lineNo*));  
 **continue**;  
 }  
 } **else if** (*currentChar* == **'\*'**) {  
 *readChar*();  
 **if** (*currentChar* == **'='**) {  
 *tokens*.add(**new** Token(Token.***MULTIEQUAL***, **"\*="**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else** {  
 *tokens*.add(**new** Token(Token.***MUL***, **"\*"**, *lineNo*));  
 **continue**;  
 }  
 } **else if** (*currentChar* == **'('**) {  
 *tokens*.add(**new** Token(Token.***LPARENT***, **"("**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **')'**) {  
 *tokens*.add(**new** Token(Token.***RPARENT***, **")"**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **'['**) {  
 *tokens*.add(**new** Token(Token.***LBRACKET***, **"["**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **']'**) {  
 *tokens*.add(**new** Token(Token.***RBRACKET***, **"]"**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **'{'**) {  
 *tokens*.add(**new** Token(Token.***LBRACE***, **"{"**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **'}'**) {  
 *tokens*.add(**new** Token(Token.***RBRACE***, **"}"**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **'/'**) {  
 *readChar*();  
 **if** (*currentChar* == **'\*'**) {  
 *readChar*();  
 **while** (**true**) {  
 **if** (*currentChar* == **'\*'**) {  
 *readChar*();  
 **if** (*currentChar* == **'/'**) {  
 *readChar*();  
 **break**;  
 }  
 } **else** {  
 *readChar*();  
 }  
 }  
 **continue**;  
 } **else if** (*currentChar* == **'/'**) {  
 **while** (*currentChar* != **'\n'**) {  
 *readChar*();  
 }  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **'='**) {  
 *tokens*.add(**new** Token(Token.***DIVEQUAL***, **"/="**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else** {  
 *tokens*.add(**new** Token(Token.***DIV***, **"/"**, *lineNo*));  
 **continue**;  
 }  
 } **else if** (*currentChar* == **'='**) {  
 *readChar*();  
 **if** (*currentChar* == **'='**) {  
 *tokens*.add(**new** Token(Token.***EQ***, **"=="**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else** {  
 *tokens*.add(**new** Token(Token.***ASSIGN***, **"="**, *lineNo*));  
 **continue**;  
 }  
 } **else if** (*currentChar* == **'>'**) {  
 *readChar*();  
 **if** (*currentChar* == **'='**) {  
 *tokens*.add(**new** Token(Token.***GET***, **">="**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else** {  
 *tokens*.add(**new** Token(Token.***GT***, **">"**, *lineNo*));  
 **continue**;  
 }  
 } **else if** (*currentChar* == **'<'**) {  
 *readChar*();  
 **if** (*currentChar* == **'='**) {  
 *tokens*.add(**new** Token(Token.***LET***, **"<="**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else if** (*currentChar* == **'>'**) {  
 *tokens*.add(**new** Token(Token.***NEQ***, **"<>"**, *lineNo*));  
 *readChar*();  
 **continue**;  
 } **else** {  
 *tokens*.add(**new** Token(Token.***LT***, **"<"**, *lineNo*));  
 **continue**;  
 }  
 } **else if** (*currentChar* >= **'0'** && *currentChar* <= **'9'**) {  
 **boolean** isReal = **false**;*//是否小数* **while** ((*currentChar* >= **'0'** && *currentChar* <= **'9'**) || *currentChar* == **'.'**) {  
 **if** (*currentChar* == **'.'**) {  
 **if** (isReal) **break**;  
 **else** isReal = **true**;  
 }  
 *sb*.append(*currentChar*);  
 *readChar*();  
 }  
 **if** (isReal) {  
 *tokens*.add(**new** Token(Token.***LITERAL\_DOUBLE***, *sb*.toString(), *lineNo*));  
 } **else** {  
 *tokens*.add(**new** Token(Token.***LITERAL\_INT***, *sb*.toString(), *lineNo*));  
 }  
 *sb*.delete(0, *sb*.length());  
 **continue**;  
 } **else if** (*currentChar* == **'\"'**) {  
 *readChar*();  
 StringBuilder sbr = **new** StringBuilder();  
 **try** {  
 **while** (*currentChar* != **'\"'**) {  
 sbr.append(*currentChar* + **""**);  
 *readChar*();  
 }  
 } **catch** (Exception e) {  
 System.***out***.println(**"引号不匹配"**);  
 } **finally** {  
 *tokens*.add(**new** Token(Token.***LITERAL\_STRING***, sbr.toString(), *lineNo*));  
 sbr.delete(0, sbr.length());  
 *readChar*();  
 }  
 } **else if** (*currentChar* >= **'A'** && *currentChar* <= **'Z'** || *currentChar* >= **'a'** && *currentChar* <= **'z'** || *currentChar* == **'\_'**) {  
 **if** (*currentChar* == **'i'** && *inputChar*[*count*] == **'n'** && *inputChar*[*count* + 1] == **'t'** && *inputChar*[*count* + 2] == **' '**) {  
 *readChar*();*readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***INT***, **"INT"**, *lineNo*));  
 **continue**;  
 } **else if** (*currentChar* == **'d'** && *inputChar*[*count*] == **'o'** && *inputChar*[*count* + 1] == **'u'** && *inputChar*[*count* + 2] == **'b'** && *inputChar*[*count* + 3] == **'l'** && *inputChar*[*count* + 4] == **'e'** && *inputChar*[*count* + 5] == **' '**) {*//可能越界？  
 readChar*();*readChar*();*readChar*();*readChar*();*readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***DOUBLE***, **"DOUBLE"**, *lineNo*));  
 **continue**;  
 } **else if** (*currentChar* == **'i'** && *inputChar*[*count*] == **'f'** && *inputChar*[*count* + 1] == **'('**) {  
 *readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***IF***, **"IF"**, *lineNo*));  
 **continue**;  
 } **else if** (*currentChar* == **'e'** && *inputChar*[*count*] == **'l'** && *inputChar*[*count* + 1] == **'s'** && *inputChar*[*count* + 2] == **'e'** && *inputChar*[*count* + 3] == **'{'**) {  
 *readChar*();*readChar*();*readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***ELSE***, **"ELSE"**, *lineNo*));  
 **continue**;  
 } **else if** (*currentChar* == **'v'** && *inputChar*[*count*] == **'o'** && *inputChar*[*count* + 1] == **'i'** && *inputChar*[*count* + 2] == **'d'** && *inputChar*[*count* + 3] == **' '**) {  
 *readChar*();*readChar*();*readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***VOID***, **"VOID"**, *lineNo*));  
 **continue**;  
 } **else if** (*currentChar* == **'r'** && *inputChar*[*count*] == **'e'** && *inputChar*[*count* + 1] == **'a'** && *inputChar*[*count* + 2] == **'d'** && *inputChar*[*count* + 3] == **' '**) {*//可能越界？  
 readChar*();*readChar*();*readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***READ***, **"READ"**, *lineNo*));  
 **continue**;  
 } **else if** (*currentChar* == **'w'** && *inputChar*[*count*] == **'h'** && *inputChar*[*count* + 1] == **'i'** && *inputChar*[*count* + 2] == **'l'** && *inputChar*[*count* + 3] == **'e'** && *inputChar*[*count* + 4] == **'('**) {*//可能越界？  
 readChar*();*readChar*();*readChar*();*readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***WHILE***, **"WHILE"**, *lineNo*));  
 **continue**;  
 } **else if** (*currentChar* == **'w'** && *inputChar*[*count*] == **'r'** && *inputChar*[*count* + 1] == **'i'** && *inputChar*[*count* + 2] == **'t'** && *inputChar*[*count* + 3] == **'e'** && *inputChar*[*count* + 4] == **' '**) {*//可能越界？  
 readChar*();*readChar*();*readChar*();*readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***WRITE***, **"WRITE"**, *lineNo*));  
 **continue**;  
 } **else if** (*currentChar* == **'s'** && *inputChar*[*count*] == **'t'** && *inputChar*[*count* + 1] == **'r'** && *inputChar*[*count* + 2] == **'i'** && *inputChar*[*count* + 3] == **'n'** && *inputChar*[*count* + 4] == **'g'** && *inputChar*[*count* + 5] == **' '**) {  
 *readChar*();*readChar*();*readChar*();*readChar*();*readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***STRING***, **"STRING"**, *lineNo*));  
 **continue**;  
 } **else if** (*currentChar* == **'r'** && *inputChar*[*count*] == **'e'** && *inputChar*[*count* + 1] == **'t'** && *inputChar*[*count* + 2] == **'u'** && *inputChar*[*count* + 3] == **'r'** && *inputChar*[*count* + 4] == **'n'** && *inputChar*[*count* + 5] == **' '**) {  
 *readChar*();*readChar*();*readChar*();*readChar*();*readChar*();*readChar*();  
 *tokens*.add(**new** Token(Token.***RETURN***, **"return"**, *lineNo*));  
 **continue**;  
 } **else** {  
 **while** (**true**) {  
 **if** (*currentChar* >= **'A'** && *currentChar* <= **'Z'** || *currentChar* >= **'a'** && *currentChar* <= **'z'** || *currentChar* == **'\_'** || *currentChar* >= **'0'** && *currentChar* <= **'9'**) {  
 *sb*.append(*currentChar*);  
 *readChar*();  
 } **else** {  
 *tokens*.add(**new** Token(Token.***ID***, *sb*.toString(), *lineNo*));  
 *//readChar();  
 sb*.delete(0, *sb*.length());  
 **break**;  
 }  
 }  
 }  
 } **else if** (*currentChar* == **'#'**) **break**;  
 **else if** (*currentChar* == **' '** || *currentChar* == **'\r'** || *currentChar* == **'\n'** || *currentChar* == **'\f'** || *currentChar* == **'\t'**)  
 *readChar*();  
 **else** {  
 *LexError* += **"非法字符:"** + *currentChar* + **"\tlineNo:"** + *lineNo*;  
 *LexError* += **"\n"**;  
 *readChar*();  
 }  
 }  
}

**第五部分 主要数据结构**

Token的结构如下，lineNo代表行数，tokenNo代表token的类型编号，value代表token的类型的实际对应的字符串。结构如下图  
**public class** Token {  
 **public int tokenNo**;  
 **public** String **value**=**""**;  
 **public int lineNo**;  
 **public** Token(**int** tokenNo,**int** lineNo){  
 **this**.**tokenNo** = tokenNo;  
 **this**.**lineNo** = lineNo;  
 }  
 **public** Token(**int** tokenNo,String value,**int** lineNo){  
 **this**.**tokenNo** = tokenNo;  
 **this**.**value** = value;  
 **this**.**lineNo** = lineNo;  
 }  
 **public static final int *START*** = 0;  
 **public static final int *IF*** = 1;  
 **public static final int *ELSE*** = 2;  
 **public static final int *WHILE*** = 3;  
 **public static final int *READ*** = 4;  
 **public static final int *WRITE*** = 5;  
 **public static final int *INT*** = 6;  
 **public static final int *DOUBLE*** = 7;  
 **public static final int *STRING*** = 8;  
 **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;  
 **public static final int *LT*** = 14;  
 **public static final int *EQ*** = 15;  
 **public static final int *NEQ*** = 16;  
 **public static final int *LPARENT*** = 17;  
 **public static final int *RPARENT*** = 18;  
 **public static final int *SEMI*** = 19;  
 **public static final int *LBRACE*** = 20;  
 **public static final int *RBRACE*** = 21;  
 **public static final int *LCOM*** = 22;  
 **public static final int *RCOM*** = 23;  
 **public static final int *SCOM*** = 24;  
 **public static final int *LBRACKET*** = 25;  
 **public static final int *RBRACKET*** = 26;  
 **public static final int *LET*** = 27;  
 **public static final int *GT*** = 28;  
 **public static final int *GET*** = 29;  
 **public static final int *ID*** = 30;  
 **public static final int *LITERAL\_INT*** = 31;  
 **public static final int *LITERAL\_DOUBLE*** = 32;  
 **public static final int *LITERAL\_STRING*** = 33;  
 **public static final int *LOGIC\_EXP*** = 34;  
 **public static final int *ADDTIVE\_EXP*** = 35;  
 **public static final int *TERM\_EXP*** = 36;  
 **public static final int *STRING\_EXP*** = 37;  
 **public static final int *RETURN*** = 38;  
 **public static final int *VOID*** = 39;  
 **public static final int *PLUSPLUS*** = 40;  
 **public static final int *PLUSEQUAL*** = 41;  
 **public static final int *MINUSMINUS*** = 42;  
 **public static final int *MINUSEQUAL*** = 43;  
 **public static final int *MULTIEQUAL*** = 44;  
 **public static final int *DIVEQUAL*** = 45;  
 **public static final int *COMMA*** = 46;  
}

**第六部分 测试部分**

ZZZ。

**语法分析目录**

**第一部分 Mini语言语法规则 ……………………… 1**

**第二部分 文法定义 ………………………**

**第三部分 语法分析算法 ………………………**

**第四部分 主要数据结构 ………………………**

**第五部分 出错处理出口 ………………………**

**第六部分 测试部分 ………………………**

**第一部分 Mini语言语法规则**

ZZZ。

**第二部分 文法定义**

ZZZ。

**第三部分 语法分析算法**

根据在词法分析阶段得到的token序列，先设计TreeNode函数，代表各种语法树的节点。如IF语句，WHILE语句等（详见下面的代码），语法树有一个type属性代表节点的类型，有四个子树，LEFT,MIDDLE,RIGHT,NEXT，代表语法节点中的各个子成分，如IF节点的LEFT子树代表IF的条件，MIDDLE子树代表如果为真的语句，RIGHT是只有ELSE存在的时候，存放IF为假的时候的语句。NEXT存放一个BLOCK中的父节点的下一条语法节点。Value存储表达式的字符串形式的值或变量名，DataType存放具体的变量类型或者操作符类型或者复合表达式或字面值，根据不同的语法节点有不懂的效果。以下是关键代码。  
**public static** LinkedList<TreeNode> graAnalysis(ArrayList<Token> tokens){  
 **if**(*treeNodeList*.size()!=0)  
 **for**(**int** i=*treeNodeList*.size()-1;i>=0;i--)  
 *treeNodeList*.remove(i);  
 *tokenList*=tokens;  
 **for**(**int** i=0;i<*a*.**length**;i++)  
 {  
 *a*[i]=0;  
 }  
 *//tokenList.add(0,new Token(Token.START));  
 iterator* = *tokenList*.listIterator();  
 TreeNode node = **new** TreeNode(TreeNode.***PROGRAM***);  
 TreeNode tmp=**null**;  
 *treeNodeList*.add(node);  
 **while**(*iterator*.hasNext()){  
 tmp=*parseStmt*();  
 node.**mNext**=tmp;  
 node=tmp;  
 }  
 *Show*();  
 *root* = *treeNodeList*.getFirst();  
 **return** *treeNodeList*;  
 }  
 **private static** TreeNode parseStmt() **throws** ParserException {  
 **switch** (*getNextTokenType*()) {  
 **case** Token.***IF***: **return** *parseIfStmt*();  
 **case** Token.***WHILE***: **return** *parseWhileStmt*();  
 **case** Token.***READ***: **return** *parseReadStmt*();  
 **case** Token.***WRITE***: **return** *parseWriteStmt*();  
 **case** Token.***INT***:  
 **case** Token.***DOUBLE***:  
 **case** Token.***STRING***:  
 **case** Token.***VOID***: **return** *parseDeclareStmt*();*//变量声明+函数声明* **case** Token.***LBRACE***: **return** *parseStmtBlock*();  
 **case** Token.***ID***: **return** *parseAssignStmt*();  
 **case** Token.***RETURN***: **return** *parseReturnStmt*();  
 **default**:  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : expected token"**);  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 }  
 }*//stmt-block* **private static** TreeNode parseReturnStmt() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[0]==0)  
 {  
 node = **new** TreeNode(TreeNode.***RETURN\_STMT***);  
 *a*[0]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***RETURN\_STMT***,*a*[0]);  
 *a*[0]++;  
 }  
 *consumeNextToken*(Token.***RETURN***);*//消耗一个return* node.**mLeft**=*parseExp*();*//exp  
 consumeNextToken*(Token.***SEMI***);  
 **return** node;  
 }  
 **private static** TreeNode parseIfStmt() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[1]==0)  
 {  
 node = **new** TreeNode(TreeNode.***IF\_STMT***);  
 *a*[1]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***IF\_STMT***,*a*[1]);  
 *a*[1]++;  
 }  
 *consumeNextToken*(Token.***IF***);*//消耗一个if  
 consumeNextToken*(Token.***LPARENT***);*//消耗一个左括号* node.**mLeft**=*parseExp*();*//exp  
 consumeNextToken*(Token.***RPARENT***);  
 node.**mMiddle**=*parseStmt*();*//stmt-block* **if** (*getNextTokenType*() == Token.***ELSE***) {  
 *consumeNextToken*(Token.***ELSE***);  
 node.**mRight**=*parseStmt*();*//stmt-block* }  
 **return** node;  
 }  
 **private static** TreeNode parseWhileStmt() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[2]==0)  
 {  
 node = **new** TreeNode(TreeNode.***WHILE\_STMT***);  
 *a*[2]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***WHILE\_STMT***,*a*[2]);  
 *a*[2]++;  
 }  
  
 *consumeNextToken*(Token.***WHILE***);*//消耗一个while  
 consumeNextToken*(Token.***LPARENT***);*//消耗一个左括号* node.**mLeft**=*parseExp*();*//exp  
 consumeNextToken*(Token.***RPARENT***);  
 node.**mMiddle**=*parseStmt*();*//stmt-block* **return** node;  
 }  
 **private static** TreeNode parseReadStmt() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[3]==0)  
 {  
 node = **new** TreeNode(TreeNode.***READ\_STMT***);  
 *a*[3]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***READ\_STMT***,*a*[3]);  
 *a*[3]++;  
 }  
  
 *consumeNextToken*(Token.***READ***);  
 node.**mLeft**=*variableName*();  
 *consumeNextToken*(Token.***SEMI***);  
 **return** node;  
 }  
 **private static** TreeNode parseWriteStmt() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[4]==0)  
 {  
 node = **new** TreeNode(TreeNode.***WRITE\_STMT***);  
 *a*[4]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***WRITE\_STMT***,*a*[4]);  
 *a*[4]++;  
 }  
  
 *consumeNextToken*(Token.***WRITE***);  
 node.**mLeft**=*parseExp*();  
 *consumeNextToken*(Token.***SEMI***);  
 **return** node;  
 }  
 **private static** TreeNode parseDeclareStmt() **throws** ParserException{  
 **if**(*checkNextTokenType*(Token.***INT***, Token.***DOUBLE***, Token.***STRING***, Token.***VOID***)){  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 *iterator*.next();*//foo* Token tmp = *iterator*.next();*//函数or变量  
 iterator*.previous();*iterator*.previous();*//iterator.previous();//少移动一个* **if**(tmp.**tokenNo**==Token.***LPARENT***){*//函数声明* **return** *parseDeclareFunStmt*();  
 }**else** {*//变量声明* **return** *parseDeclareVarStmt*();  
 }  
 }**else**{  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be variable type"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be variable type");* }  
  
 }  
 **private static** TreeNode parseDeclareFunStmt() **throws** ParserException{*//函数声明* TreeNode node;  
 **if**(*a*[5]==0)  
 {  
 node = **new** TreeNode(TreeNode.***DECLARE\_FUN\_STMT***);  
 *a*[5]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***DECLARE\_FUN\_STMT***,*a*[5]);  
 *a*[5]++;  
 }  
  
 TreeNode varNode;  
 **if**(*a*[6]==0)  
 {  
 varNode = **new** TreeNode(TreeNode.***FUN***);  
 *a*[6]++;  
 }  
 **else** {  
 varNode = **new** TreeNode(TreeNode.***FUN***,*a*[6]);  
 *a*[6]++;  
 }*//存储返回值类型和函数名  
 //if(checkNextTokenType(Token.INT, Token.DOUBLE, Token.STRING, Token.VOID)){* **if**(*currentToken*.**tokenNo**==Token.***INT***||*currentToken*.**tokenNo**==Token.***DOUBLE***||*currentToken*.**tokenNo**==Token.***STRING***||*currentToken*.**tokenNo**==Token.***VOID***){  
 **int** type = *currentToken*.**tokenNo**;  
 **if**(type==Token.***INT***){  
 varNode.**mDataType**=Token.***INT***;  
 }**else if**(type==Token.***DOUBLE***){  
 varNode.**mDataType**=Token.***DOUBLE***;  
 }**else if**(type==Token.***STRING***){  
 varNode.**mDataType**= Token.***STRING***;  
 }**else**{  
 varNode.**mDataType**= Token.***VOID***;  
 }  
 }**else**{  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be variable type"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 }  
 **if**(*checkNextTokenType*(Token.***ID***)){  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 varNode.**value**=*currentToken*.**value**;  
 }**else** {  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be ID"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 }  
 node.**mLeft**=varNode;  
 *consumeNextToken*(Token.***LPARENT***);  
 node.**mMiddle**=*parseParams*();*//有参数 会新建节点 否则无  
 consumeNextToken*(Token.***RPARENT***);  
 node.**mRight**=*parseStmtBlock*();  
  
 **return** node;  
 }  
 **private static** TreeNode parseParams() **throws** ParserException{  
 **if**(*checkNextTokenType*(Token.***RPARENT***))  
 {  
 TreeNode node;  
 **if**(*a*[7]==0)  
 {  
 node = **new** TreeNode(TreeNode.***PARAMS***);  
 *a*[7]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***PARAMS***,*a*[7]);  
 *a*[7]++;  
 }  
 **return** node;  
 }  
 **else if**(*checkNextTokenType*(Token.***INT***, Token.***DOUBLE***, Token.***STRING***)){  
  
 TreeNode node;  
 **if**(*a*[8]==0)  
 {  
 node = **new** TreeNode(TreeNode.***PARAM***);  
 *a*[8]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***PARAM***,*a*[8]);  
 *a*[8]++;  
 }  
  
 TreeNode header = node;  
 TreeNode temp= **null**;  
  
 **while**(*getNextTokenType*()!=Token.***RPARENT***){  
 temp=*parseParam*();  
 node.**mNext**=temp;  
 node=temp;  
 **if**(*getNextTokenType*()==Token.***COMMA***)  
 *consumeNextToken*(Token.***COMMA***);  
 }  
 **return** header;  
  
 }**else**{  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : wrong params"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : wrong params");* }  
 }  
 **private static** TreeNode parseParam() **throws** ParserException{  
 TreeNode node;  
 **if**(*a*[8]==0)  
 {  
 node = **new** TreeNode(TreeNode.***PARAM***);  
 *a*[8]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***PARAM***,*a*[8]);  
 *a*[8]++;  
 }  
  
 **if**(*checkNextTokenType*(Token.***INT***, Token.***DOUBLE***, Token.***STRING***)){  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 **int** type = *currentToken*.**tokenNo**;  
 **if**(type==Token.***INT***){  
 node.**mDataType**=Token.***INT***;  
 }**else if**(type==Token.***DOUBLE***){  
 node.**mDataType**=Token.***DOUBLE***;  
 }**else if**(type==Token.***STRING***){  
 node.**mDataType**= Token.***STRING***;  
 }**else**{  
 node.**mDataType**= Token.***VOID***;  
 }  
 }**else**{  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be variable type"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be variable type");* }  
  
 **if**(*checkNextTokenType*(Token.***ID***)){  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 node.**value**=*currentToken*.**value**;  
 }**else** {  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be ID"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be ID");* }  
  
 **return** node;  
 }  
 **private static** TreeNode parseDeclareVarStmt() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[9]==0)  
 {  
 node = **new** TreeNode(TreeNode.***DECLARE\_VAR\_STMT***);  
 *a*[9]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***DECLARE\_VAR\_STMT***,*a*[9]);  
 *a*[9]++;  
 }  
 TreeNode varNode;  
 **if**(*a*[10]==0)  
 {  
 varNode = **new** TreeNode(TreeNode.***VAR***);  
 *a*[10]++;  
 }  
 **else** {  
 varNode = **new** TreeNode(TreeNode.***VAR***,*a*[10]);  
 *a*[10]++;  
 }  
  
 **if**(*currentToken*.**tokenNo**==Token.***INT***||*currentToken*.**tokenNo**==Token.***DOUBLE***||*currentToken*.**tokenNo**==Token.***STRING***){  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 **int** type = *currentToken*.**tokenNo**;  
 *currentToken* = *iterator*.previous();  
 **if**(type==Token.***INT***){  
 varNode.**mDataType**=Token.***INT***;  
 }**else if**(type==Token.***DOUBLE***){  
 varNode.**mDataType**=Token.***DOUBLE***;  
 }**else**{  
 varNode.**mDataType**= Token.***STRING***;  
 }  
 }**else**{  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be variable type"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be variable type");* }  
 **if**(*checkNextTokenType*(Token.***ID***)){  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 varNode.**value**=*currentToken*.**value**;  
 }**else** {  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be ID"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be ID");* }  
 **if** (*getNextTokenType*() == Token.***ASSIGN***) {*//单个元素可以声明并赋值  
 consumeNextToken*(Token.***ASSIGN***);  
 node.**mMiddle**=*parseExp*();  
 } **else if** (*getNextTokenType*() == Token.***LBRACKET***) {*//数组元素要先声明后赋值  
 consumeNextToken*(Token.***LBRACKET***);  
 varNode.**mLeft**=*parseExp*();  
 *consumeNextToken*(Token.***RBRACKET***);  
 }**else**{}  
 *consumeNextToken*(Token.***SEMI***);  
 node.**mLeft**=varNode;  
 **return** node;  
 }  
 **private static** TreeNode parseStmtBlock() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[11]==0)  
 {  
 node = **new** TreeNode(TreeNode.***BLOCK***);  
 *a*[11]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***BLOCK***,*a*[11]);  
 *a*[11]++;  
 }  
  
 TreeNode header = node;  
 TreeNode temp= **null**;  
 *consumeNextToken*(Token.***LBRACE***);  
 **while**(*getNextTokenType*()!=Token.***RBRACE***){  
 temp=*parseStmt*();  
 node.**mNext**=temp;  
 node=temp;  
 }  
 *consumeNextToken*(Token.***RBRACE***);  
 **return** header;  
 }  
 **private static** TreeNode parseAssignStmt() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[12]==0)  
 {  
 node = **new** TreeNode(TreeNode.***ASSIGN\_STMT***);  
 *a*[12]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***ASSIGN\_STMT***,*a*[12]);  
 *a*[12]++;  
 }  
  
 node.**mLeft**=*variableName*();  
 **int** type = *consumeNextToken*(Token.***ASSIGN***,Token.***PLUSEQUAL***,Token.***MINUSEQUAL***,Token.***MULTIEQUAL***,Token.***DIVEQUAL***);*// a=1; a+=1; a++;  
// TreeNode signNode = new TreeNode(type);  
// signNode.mLeft = new TreeNode()* node.**mMiddle**=*parseExp*();  
 node.**mDataType**=type;  
 *consumeNextToken*(Token.***SEMI***);  
  
 **return** node;  
 }  
 **private static** TreeNode parseExp() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[13]==0)  
 {  
 node = **new** TreeNode(TreeNode.***EXP***);  
 *a*[13]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***EXP***,*a*[13]);  
 *a*[13]++;  
 }  
 node.**mDataType**=Token.***LOGIC\_EXP***;  
 TreeNode leftNode = *addtiveExp*();*//exp or exp <> exp2* **if**(*checkNextTokenType*(Token.***EQ***, Token.***NEQ***, Token.***GT***, Token.***GET***, Token.***LT***, Token.***LET***)){  
 node.**mLeft**=leftNode;  
 node.**mMiddle**=*logicalOp*();  
 node.**mRight**=*addtiveExp*();  
 **return** node;  
 }**else**{  
 **return** leftNode;  
 }  
  
 }  
 **private static** TreeNode addtiveExp() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[13]==0)  
 {  
 node = **new** TreeNode(TreeNode.***EXP***);  
 *a*[13]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***EXP***,*a*[13]);  
 *a*[13]++;  
 }  
 node.**mDataType**=Token.***ADDTIVE\_EXP***;  
 TreeNode leftNode = *term*();  
  
 **if** (*checkNextTokenType*(Token.***PLUS***,Token.***MINUS***)) {  
 node.**mLeft**=leftNode;  
 node.**mMiddle**=*addtiveOp*();  
 node.**mRight**=*addtiveExp*();  
 **return** node;  
 } **else** {  
 **return** leftNode;  
 }  
 }  
 **private static** TreeNode term() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[13]==0)  
 {  
 node = **new** TreeNode(TreeNode.***EXP***);  
 *a*[13]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***EXP***,*a*[13]);  
 *a*[13]++;  
 }  
 node.**mDataType**=Token.***TERM\_EXP***;  
 TreeNode leftNode = *factor*();  
  
 **if** (*checkNextTokenType*(Token.***MUL***, Token.***DIV***)) {  
 node.**mLeft**=leftNode;  
 node.**mMiddle**=*multiplyOp*();  
 node.**mRight**=*term*();  
 **return** node;  
 } **else** {  
 **return** leftNode;  
 }  
 }  
 **private static** TreeNode factor() **throws** ParserException {  
 **if** (*iterator*.hasNext()) {  
 TreeNode expNode;  
 **if**(*a*[14]==0)  
 {  
 expNode = **new** TreeNode(TreeNode.***FACTOR***);  
 *a*[14]++;  
 }  
 **else** {  
 expNode = **new** TreeNode(TreeNode.***FACTOR***,*a*[14]);  
 *a*[14]++;  
 }  
  
 **switch** (*getNextTokenType*()) {  
 **case** Token.***LPARENT***:*//(exp)  
 consumeNextToken*(Token.***LPARENT***);  
 expNode = *parseExp*();  
 *consumeNextToken*(Token.***RPARENT***);  
 **break**;  
 **case** Token.***LITERAL\_INT***:  
 **case** Token.***LITERAL\_DOUBLE***:  
 expNode.**mLeft**=*literal*();  
 **break**;  
 **case** Token.***MINUS***:*//+a* expNode.**mDataType**=Token.***MINUS***;  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 expNode.**mLeft**=*term*();  
 **break**;  
 **case** Token.***PLUS***:*//-a* expNode.**mDataType**=Token.***PLUS***;  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 expNode.**mLeft**=*term*();  
 **break**;  
 **default**:*// [a]+b  
 //返回的不是expNode* **return** *variableName*();  
 }  
 **return** expNode;  
 }  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be factor"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be factor");* }  
 **private static** TreeNode literal() **throws** ParserException {*//实际值节点* **if** (*iterator*.hasNext()) {  
 *currentToken* = *iterator*.next();  
 **int** type = *currentToken*.**tokenNo**;  
 TreeNode node;  
 **if**(*a*[15]==0)  
 {  
 node = **new** TreeNode(TreeNode.***LITERAL***);  
 *a*[15]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***LITERAL***,*a*[15]);  
 *a*[15]++;  
 }  
 node.**mDataType**=type;  
 node.**value**=*currentToken*.**value**;  
 **if** (type == Token.***LITERAL\_INT*** || type == Token.***LITERAL\_DOUBLE***|| type == Token.***LITERAL\_STRING***) {  
 **return** node;  
 } **else** {  
 *// continue execute until throw* }  
 }  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be literal value"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be literal value");* }  
 **private static** TreeNode logicalOp() **throws** ParserException {*//== <> >= <= > < 逻辑运算符* **if** (*iterator*.hasNext()) {  
 *currentToken* = *iterator*.next();  
 **int** type = *currentToken*.**tokenNo**;  
 **if** (type == Token.***EQ*** || type == Token.***GET*** || type == Token.***GT*** || type == Token.***LET*** || type == Token.***LT*** || type == Token.***NEQ***) {  
 TreeNode node;  
 **if**(*a*[16]==0)  
 {  
 node = **new** TreeNode(TreeNode.***OP***);  
 *a*[16]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***OP***,*a*[16]);  
 *a*[16]++;  
 }  
  
 node.**mDataType**=type;  
 **return** node;  
 }  
 }  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be logical operator"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be logical operator");* }  
 **private static** TreeNode addtiveOp() **throws** ParserException {*//+ -* **if** (*iterator*.hasNext()) {  
 *currentToken* = *iterator*.next();  
 **int** type = *currentToken*.**tokenNo**;  
 **if** (type == Token.***PLUS*** || type == Token.***MINUS***) {  
 TreeNode node;  
 **if**(*a*[16]==0)  
 {  
 node = **new** TreeNode(TreeNode.***OP***);  
 *a*[16]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***OP***,*a*[16]);  
 *a*[16]++;  
 }  
 node.**mDataType**=type;  
 **return** node;  
 }  
 }  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be addtive operator"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be addtive operator");* }  
 **private static** TreeNode multiplyOp() **throws** ParserException {  
 **if** (*iterator*.hasNext()) {  
 *currentToken* = *iterator*.next();  
 **int** type = *currentToken*.**tokenNo**;  
 **if** (type == Token.***MUL*** || type == Token.***DIV***) {  
 TreeNode node;  
 **if**(*a*[16]==0)  
 {  
 node = **new** TreeNode(TreeNode.***OP***);  
 *a*[16]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***OP***,*a*[16]);  
 *a*[16]++;  
 }  
 node.**mDataType**=type;  
 **return** node;  
 }  
 }  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be multiple operator"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *//throw new ParserException("line " + currentToken.lineNo + " : next token should be multiple operator");* }  
 **private static** TreeNode variableName() **throws** ParserException {  
 TreeNode node;  
 **if**(*a*[10]==0)  
 {  
 node = **new** TreeNode(TreeNode.***VAR***);  
 *a*[10]++;  
 }  
 **else** {  
 node = **new** TreeNode(TreeNode.***VAR***,*a*[10]);  
 *a*[10]++;  
 }  
 **if** (*checkNextTokenType*(Token.***ID***)) {  
 **if**(*iterator*.hasNext()) *currentToken* = *iterator*.next();  
 node.**value**=*currentToken*.**value**;  
 } **else** {  
 **while**(*iterator*.hasNext()){  
 *currentToken* = *iterator*.next();  
 **if**(*currentToken*.**tokenNo**==Token.***RBRACE***||*currentToken*.**tokenNo**==Token.***RPARENT***||*currentToken*.**tokenNo**==Token.***SEMI***) {  
 **break**;  
 }  
 }  
 *GraError*+=(**"line "** + *currentToken*.**lineNo** + **" : next token should be ID"**);  
 *GraError*+=**"\n"**;  
 **return new** TreeNode(TreeNode.***WRONG***);  
 *// throw new ParserException("line " + currentToken.lineNo + " : next token should be ID");* }  
 **if** (*getNextTokenType*() == Token.***LBRACKET***) {*//a[0]  
 consumeNextToken*(Token.***LBRACKET***);  
 node.**mLeft**=*parseExp*();  
 *consumeNextToken*(Token.***RBRACKET***);  
 }  
 **return** node;  
 }

**第四部分 主要数据结构**

TreeNode节点的代码如下：

**public class** TreeNode {  
 **public** TreeNode(**int** val){  
 **type** = val;  
 **name**=getString();  
 }  
 **public** TreeNode(**int** val,**int** i){  
 **type** = val;  
 **name**=getString()+i;  
 }  
 **public** String getString(){  
 **if**(**type**==0) **return "NULL"**;  
 **if**(**type**==1) **return "IF\_STMT"**;  
 **if**(**type**==2) **return "WHILE\_STMT"**;  
 **if**(**type**==3) **return "READ\_STMT"**;  
 **if**(**type**==4) **return "WRITE\_STMT"**;  
 **if**(**type**==5) **return "DECLARE\_VAR\_STMT"**;  
 **if**(**type**==6) **return "ASSIGN\_STMT"**;  
 **if**(**type**==7) **return "EXP"**;  
 **if**(**type**==8) **return "VAR"**;  
 **if**(**type**==9) **return "OP"**;  
 **if**(**type**==10) **return "FACTOR"**;  
 **if**(**type**==11) **return "LITERAL"**;  
 **if**(**type**==12) **return "DECLARE\_FUN\_STMT"**;  
 **if**(**type**==13) **return "FUN"**;  
 **if**(**type**==14) **return "PARAM"**;  
 **if**(**type**==15) **return "RETURN\_STMT"**;  
 **if**(**type**==16) **return "PARSE\_STMT"**;  
 **if**(**type**==17) **return "PROGRAM"**;  
 **if**(**type**==18) **return "BLOCK"**;  
 **return null**;  
 }  
 **public static final int *WRONG*** = -1;  
 **public static final int *NULL*** = 0;  
 **public static final int *IF\_STMT*** = 1;  
 **public static final int *WHILE\_STMT*** = 2;  
 **public static final int *READ\_STMT*** = 3;  
 **public static final int *WRITE\_STMT*** = 4;  
 **public static final int *DECLARE\_VAR\_STMT*** = 5;  
 **public static final int *DECLARE\_FUN\_STMT*** = 12;  
 **public static final int *ASSIGN\_STMT*** = 6;  
 **public static final int *EXP*** = 7;  
 **public static final int *VAR*** = 8;  
 **public static final int *FUN*** = 13;  
 **public static final int *OP*** = 9;  
 **public static final int *FACTOR*** = 10;  
 **public static final int *LITERAL*** = 11;  
 **public static final int *PARAM*** = 14;  
 **public static final int *RETURN\_STMT*** = 15;  
 **public static final int *PARSE\_STMT*** = 16;  
 **public static final int *PROGRAM*** = 17;  
 **public static final int *BLOCK*** = 18;  
 **public static final int *PARAMS***=19;  
 **public int type**;  
 **public** TreeNode **mLeft**;  
 **public** TreeNode **mMiddle**;  
 **public** TreeNode **mRight**;  
 **public int mDataType**;  
 **public** String **value**;  
 **public** TreeNode **mNext**;  
 **public** String **name**;  
 }

**第五部分 出错处理出口**

**throws** ParserException

或者是保存到ERROR 在main里面如果error不为空（有错）就显示，否则显示语法树

**第六部分 测试部分**

ZZZ。