# 手工编写递归下降预测分析程序

## 设计 Oberon-0 语言的翻译模式

语法规则改写如下：主要操作是消除左递归，提取左公因子。其中的语义信息主要涉及终结符和非终结符的类型，用于判断是否出现类型不匹配的语义错误，其他语法错误和语义错误将在程序编写中实现查找。

```
module -> "MODULE" identifier ";" declaration beginStatementSequence "END"
identifier "."
beginStatementSequence -> "BEGIN" statement_sequence | ;
declaration -> constBlock typeBlock varBlocks procedureBlock
constBlock -> "CONST" identifierExpressionBlock | ;
iExpressionBlock -> identifier "=" expression ";" iExpressionBlock | ;
typeBlock ->  "TYPE" identifierTypeBlock | ;
identifierTypeBlock -> identifier "=" typeKind ";" typeList | ;
varBlocks -> "VAR" identifierListTypeBlock | ;
iListTypeBlock -> identifierList ":" typeKind ";" iListTypeBlock | ;
procedureBlock ->procedureHeading";" procedureBody ";" procedureBlock | ;
procedureBody -> declaration beginStatementSequence "END" identifier;
procedureHeading -> "PROCEDURE" identifier formalParametersBlock ;
formalParametersBlock -> "(" fpSection ")" | ;
fpSection -> varBlock identifierList ":" typeKind | varBlock identifierList ":"
typeKind ";" fpSection | ;
varBlock -> "VAR" | ;
identifierList -> identifier | identifier "," identifierList1;
typeKind -> identifier| arrayType | recordType | "INTEGER" | "BOOLEAN" ;
arrayType -> "ARRAY" expression "OF" typeKind {arrayType.type = ARRAY}
recordType -> "RECORD" fieldList "END" {recordType.type = RECORD};
fieldList -> fieldOne ";" fieldList | fieldOne ;
fieldOne -> identifierList ":" typeKind | ;
statementSequence -> statement | statement ";" statementSequence ;
statement -> assignment | procedureCall | ifStatement | whileStatement |
rwStatement;
rwStatement -> "READ" "LPAREN" identifier "RPAREN" | "WRITE" "LPAREN" identifier
"RPAREN" | "WRITELN" "LPAREN" identifier "RPAREN" | "WRITELN" "LPAREN" "RPAREN"
;
assignment -> identifier selector ":=" expression
                {   if (selector.type == null) {
                        if (identifier.type != expression.type) throw
TypeMismatched;
                    }elif (selector.type != expression.type) throw
TypeMismatched;
                };
procedureCall -> identifier actualParameters;
actualParameters -> "(" ")" | "(" expressionList ")" | ;
expressionList -> expression {expressionList += expreesion.type}
| expression "," expressionList1 {expressionList1 = expressionList +
expression.type};
ifStatement -> "IF" expression "THEN" statementSequence elsifStatement
elseStatement
                    {if(expression.type != BOOLEAN) throw TypeMismatched};
```

```
elsifStatement -> "ELSIF" expression "THEN" statementSequence elsifStatement
                    {if(expression.type != BOOLEAN) throw TypeMismatched};| ;
elseStatement -> "ELSE" statementSequence "END" | ;
whileStatement -> "WHILE" expression "DO" statementSequence "END"
                    {if(expression.type != BOOLEAN) throw TypeMismatched};
expression -> simpleExpression1 reOp simpleExpression2
            { if (simpleExpression1.type != INTEGER || simpleExpression2.type
!=INTEER)
                    throw TypeMismatched;
                else expression.type = BOOLEAN;}
            | simpleExpression {expression.type = simpleExpression.type};
reOp -> "=" | "#" | "<" | "<=" | ">" | ">=" ;
simpleExpression -> termHead term {simpleExpression.type = term.type}
            | termHead term lowOp simpleExpression1
            {if(term.type=lowOp.type=simpleExpression1.type)
                simpleExpression.type = term.type
                else throw TypeMismatched };
termHead -> "+" | "-" | ;
lowOp -> "+" {lowOp.type = INTEGER}| "-"{lowOp.type = INTEGER}
            | "OR" {lowOp.type = BOOLEAN};
term -> factor {term.type = factor.type}
        | factor highOp term1 { if(factor.type=highOp.type=term1.type)
                                    term.type = factor.type;
                                else throw TypeMismatched };
highOp -> "*" {highOp.type = INTEGER} | "DIV" {highOp.type = INTEGER}
            | "MOD" {highOp.type = INTEGER} | "&" {highOp.type = BOOLEAN};
factor -> identifier selectorBlock {factor.type = selectorBlock.type}
            | NUMBER {factor.type = INTEGER}
            | "(" expression ")" {factor.type = expression.type}
            | "~" factor1  {if (factor1.type != BOOLEAN) throw TypeMismatched;
                                else factor.type =BOOLEAN;}
selectorBlock -> selector selectorBlock {selectorBlock.type = selector.type }| ;
selector -> "DOT" IDENTIFIER {selector.type = 程序记录;}
            | "LMIDPAR" expression "RMIDPAR" {selector.type = expression.type;};
```

whilestate 中的expression需要有类型变量 也就是expression返回类型，那么simpleexpression也应该返回类型，term也应该返回类型，factor也是。

关于selector，相当于一个record或array中的成员，这就需要记录过程的数组？？

## 编写递归下降预测分析程序

创建一个流程图需要的语句：

`Module sampleModule = new Module("Sample");` 创建模块，在 `modulesBlock()` 使用，创建的模块作为参数传递到子结点 `beginStatementSequence()`，最终收集各个过程的返回，`sampleModule.add(proc)`，`sampleModule .show();` 展示

`Procedure proc = sampleModule.add(procedureHeading);` 创建过程，在 `procedureDeclaration()` 中使用，作为返回值传回 `modulesBlock()`

`PrimitiveStatement state = (new PrimitiveStatement(statement));` 创建普通语句，在 `assignment()` 中创建，作为返回值返回到 `statement()` ,传递到 `statementSequence()` 时后存储有多个 `statement()` 中创建的语句包括 `whileStatement()` `ifStatement()` 等，将数组返回到 `beginStatementSequence()` `whileStatement()`、`ifStatement()` 等上级结点中，使用

`proc.add(state)`、`wstmt.getLoopBody().add(state)`、`istmt.getTrueBody().add(state)`语句添加

`WhileStatement wstmt = new WhileStatement(expression);`创建，在`whileStatement()`中创建，创建的参数为`expression()`的返回值，并使用`statementSequence()`的返回值添加在`wstmt.getLoopBody()`中

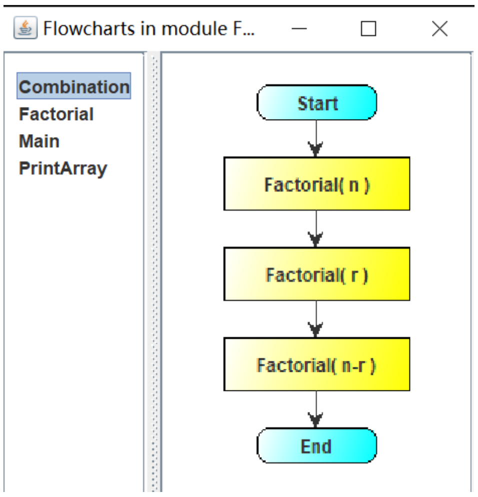
`ifStatement`与`whileStatement`类似，`statementSequence`添加在`ifStmt.getTrueBody()`中，`elsifBlock`、`elseBlock`添加在`ifStmt.getFalseBody()`中

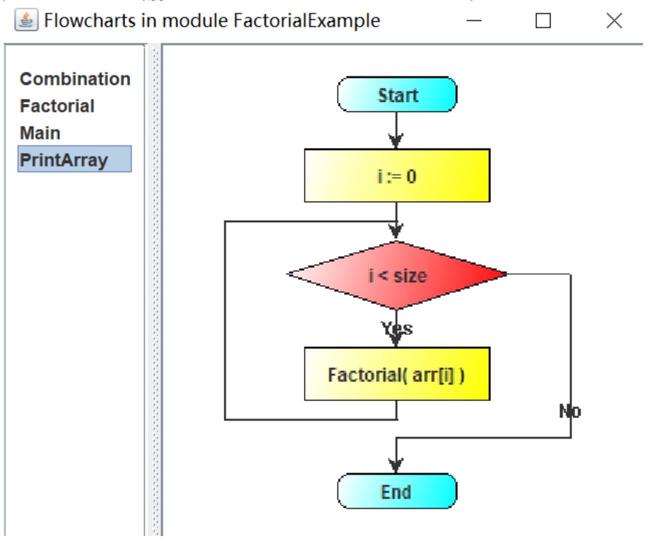
`procedureCall()`、`readBlock`、`writeBlock`、`writelnBlock`则都是创建为普通语句，返回到`statement()`。

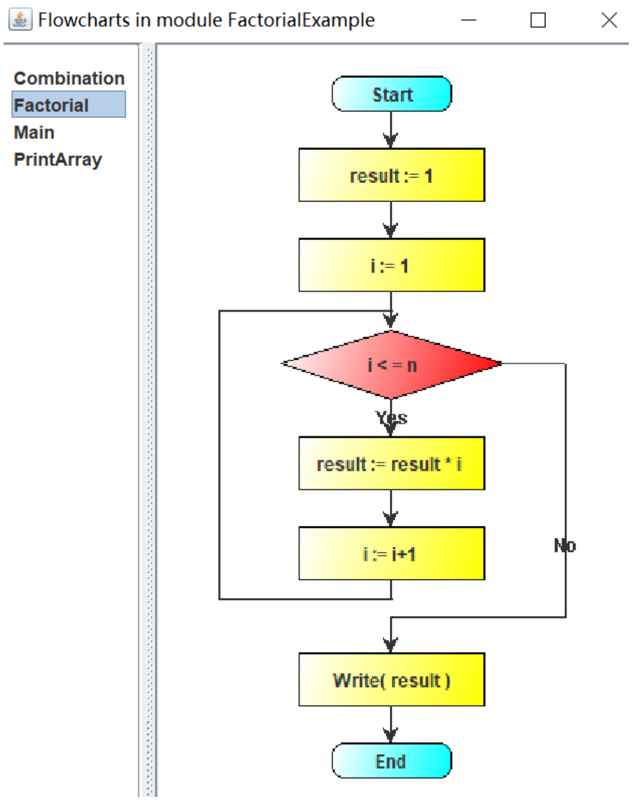
以上过程涵盖了创建一个流程图的参数传递情况，结合前面的语义信息编写语法分析器代码。

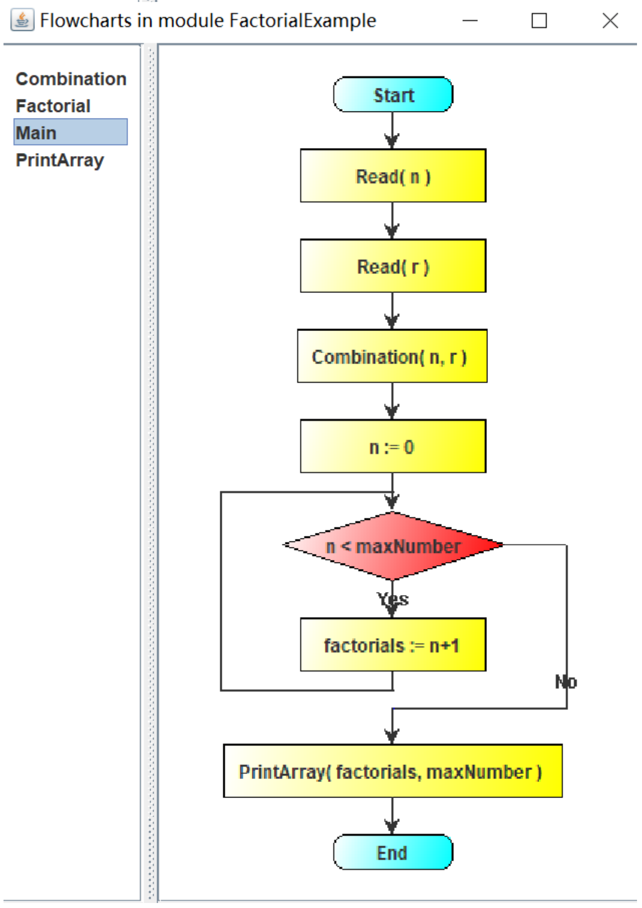
展示运行结果：

`run.bat`

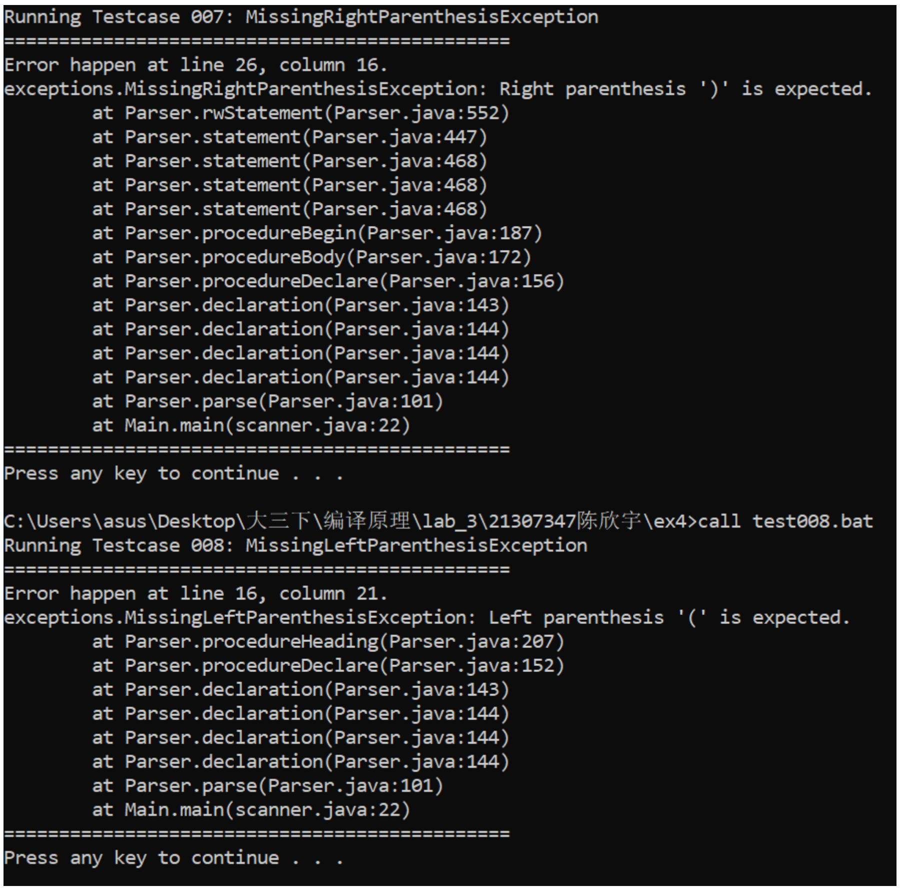




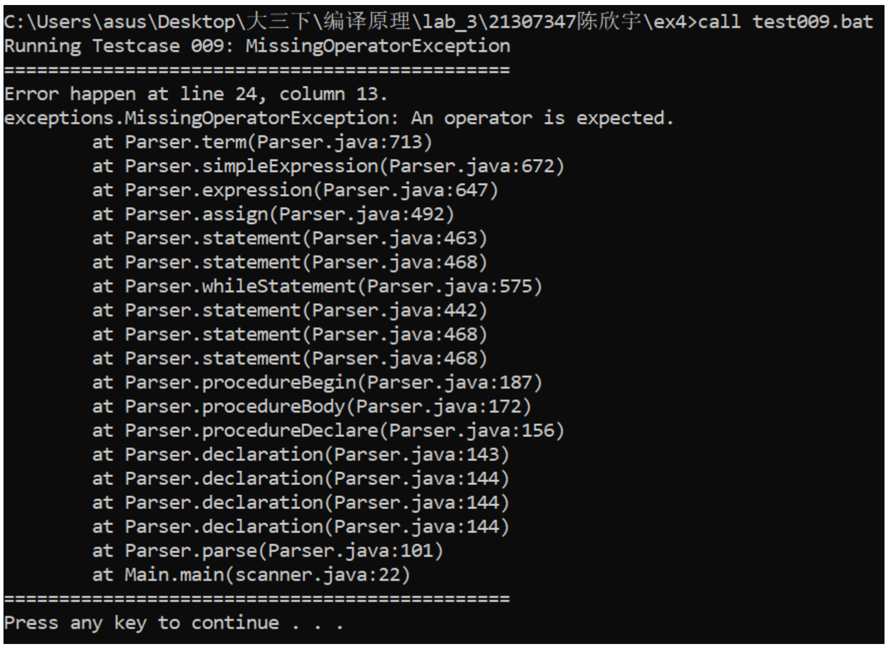




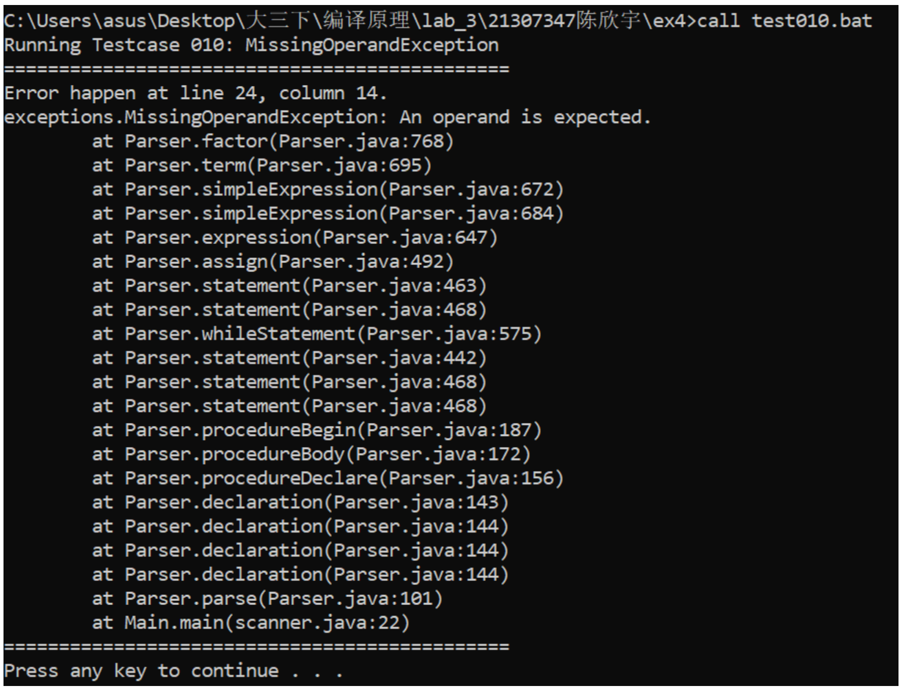
`test.bat`： 测试结果符合预期



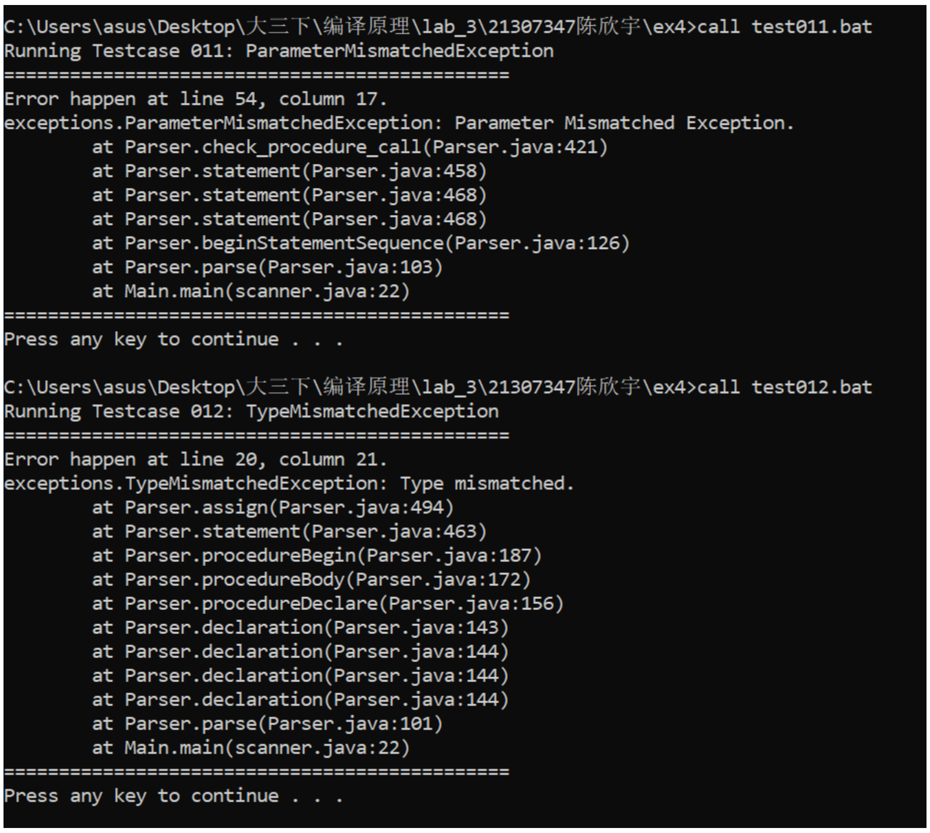
```
Running Testcase 007: MissingRightParenthesisException
==============================================
Error happen at line 26, column 16.
exceptions.MissingRightParenthesisException: Right parenthesis ')' is expected.
        at Parser.rwStatement(Parser.java:552)
        at Parser.statement(Parser.java:447)
        at Parser.statement(Parser.java:468)
        at Parser.statement(Parser.java:468)
        at Parser.statement(Parser.java:468)
        at Parser.procedureBegin(Parser.java:187)
        at Parser.procedureBody(Parser.java:172)
        at Parser.procedureDeclare(Parser.java:156)
        at Parser.declaration(Parser.java:143)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.parse(Parser.java:101)
        at Main.main(scanner.java:22)
==============================================
Press any key to continue . . .

C:\Users\asus\Desktop\大三下\编译原理\lab_3\21307347陈欣宇\ex4>call test008.bat
Running Testcase 008: MissingLeftParenthesisException
==============================================
Error happen at line 16, column 21.
exceptions.MissingLeftParenthesisException: Left parenthesis '(' is expected.
        at Parser.procedureHeading(Parser.java:207)
        at Parser.procedureDeclare(Parser.java:152)
        at Parser.declaration(Parser.java:143)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.parse(Parser.java:101)
        at Main.main(scanner.java:22)
==============================================
Press any key to continue . . .
```



```
C:\Users\asus\Desktop\大三下\编译原理\lab_3\21307347陈欣宇\ex4>call test009.bat
Running Testcase 009: MissingOperatorException
==============================================
Error happen at line 24, column 13.
exceptions.MissingOperatorException: An operator is expected.
        at Parser.term(Parser.java:713)
        at Parser.simpleExpression(Parser.java:672)
        at Parser.expression(Parser.java:647)
        at Parser.assign(Parser.java:492)
        at Parser.statement(Parser.java:463)
        at Parser.statement(Parser.java:468)
        at Parser.whileStatement(Parser.java:575)
        at Parser.statement(Parser.java:442)
        at Parser.statement(Parser.java:468)
        at Parser.statement(Parser.java:468)
        at Parser.procedureBegin(Parser.java:187)
        at Parser.procedureBody(Parser.java:172)
        at Parser.procedureDeclare(Parser.java:156)
        at Parser.declaration(Parser.java:143)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.parse(Parser.java:101)
        at Main.main(scanner.java:22)
==============================================
Press any key to continue . . .
```



```
C:\Users\asus\Desktop\大三下\编译原理\lab_3\21307347陈欣宇\ex4>call test010.bat
Running Testcase 010: MissingOperandException
==============================================
Error happen at line 24, column 14.
exceptions.MissingOperandException: An operand is expected.
        at Parser.factor(Parser.java:768)
        at Parser.term(Parser.java:695)
        at Parser.simpleExpression(Parser.java:672)
        at Parser.simpleExpression(Parser.java:684)
        at Parser.expression(Parser.java:647)
        at Parser.assign(Parser.java:492)
        at Parser.statement(Parser.java:463)
        at Parser.statement(Parser.java:468)
        at Parser.whileStatement(Parser.java:575)
        at Parser.statement(Parser.java:442)
        at Parser.statement(Parser.java:468)
        at Parser.statement(Parser.java:468)
        at Parser.procedureBegin(Parser.java:187)
        at Parser.procedureBody(Parser.java:172)
        at Parser.procedureDeclare(Parser.java:156)
        at Parser.declaration(Parser.java:143)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.parse(Parser.java:101)
        at Main.main(scanner.java:22)
==============================================
Press any key to continue . . .
```



```
C:\Users\asus\Desktop\大三下\编译原理\lab_3\21307347陈欣宇\ex4>call test011.bat
Running Testcase 011: ParameterMismatchedException
==============================================
Error happen at line 54, column 17.
exceptions.ParameterMismatchedException: Parameter Mismatched Exception.
        at Parser.check_procedure_call(Parser.java:421)
        at Parser.statement(Parser.java:458)
        at Parser.statement(Parser.java:468)
        at Parser.statement(Parser.java:468)
        at Parser.beginStatementSequence(Parser.java:126)
        at Parser.parse(Parser.java:103)
        at Main.main(scanner.java:22)
==============================================
Press any key to continue . . .

C:\Users\asus\Desktop\大三下\编译原理\lab_3\21307347陈欣宇\ex4>call test012.bat
Running Testcase 012: TypeMismatchedException
==============================================
Error happen at line 20, column 21.
exceptions.TypeMismatchedException: Type mismatched.
        at Parser.assign(Parser.java:494)
        at Parser.statement(Parser.java:463)
        at Parser.procedureBegin(Parser.java:187)
        at Parser.procedureBody(Parser.java:172)
        at Parser.procedureDeclare(Parser.java:156)
        at Parser.declaration(Parser.java:143)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.declaration(Parser.java:144)
        at Parser.parse(Parser.java:101)
        at Main.main(scanner.java:22)
==============================================
Press any key to continue . . .
```

## 语法分析讨论：自顶向下 vs. 自底向上

- 技术简单性

二者更有优缺点，自顶向下的程序只需要根据当前lookahead直接采取动作，但如果文法不满足LL(1)性质，需进行文法改造；自底向上则需要保留更多的信息，在实验三中更多是依赖JavaCUP工具生成，分析表的构造和状态转移的理解更为困难，在一定程度上加大调试难度。

- 技术通用性

自顶向下可以处理的是 LL（1）文法的语言但不适用于更广泛的上下文无关文法，而自底向上处理语言可以是LR(0), LR(1), SLR(1), LALR(1)，范围更广，可根据需求例如期望报错的时机选择适合的语言。

- 语义动作的表达

自顶向下的分析技术在递归下降分析中，语义动作可以直接嵌入在相应的递归函数中，语义动作执行和语法规则匹配是同步进行的，较为直观；而自底向上的分析技术中，语义动作通常与规约操作绑定，这使得语义动作和语法规则的匹配分离，可以保持分析表和语义动作的独立性，方便维护。

- 出错恢复

自底向上在这方面能够更快发现语法错误，出错恢复需将当前部分token弹出栈，继续程序的分析过程。而自顶向下中，除了将当前token弹出，还必须将有关的正在返回的token弹出，直到可以进行下一次正确的分析，在递归函数中的错误处理难度较大。

- 表格驱动

递归下降分析直接使用递归函数实现，在复杂文法下，递归函数会变得繁琐；自底向上的分析表提供了使得分析过程自动化，但LR 分析表可能很大，特别是针对 LR(1) 文法，增加了内存消耗。

- 速度

自底向上分析的速度更快，因为自顶向下看到一个token就开始了动作的执行，会有重复试错的过程，耗时较大；自底向上分析主要消耗在构建表上，能够看到了若干token，每个token只需处理一次，规约和移进操作高效。