
中山大学计算机院本科生实验报告

(2023 学年春季学期)

课程名称: 编译原理

批改人:

实验	基于表达式的计算器 EXPR-Eval	专业 (方向)	计算机科学与技术计科一班
学号	21307099	姓名	李英骏
Email	liyj323@mail2.sysu.edu.cn	完成日期	2024 年 5 月 19 日

目录

1 程序结构设计	3
2 实验过程和核心代码	4
2.1 语法的二义性	4
2.2 设计并实现词法分析程序	5
2.2.1 DFA 的设计	5
2.2.2 运算符的分类	6
2.2.3 Token 的设计与实现	7
2.2.4 DFA 的代码实现	8
2.2.5 实验文档中关注的问题	9
2.2.6 Scanner(词法分析器) 的实现	10
2.3 构造算符优先关系表	11
2.4 设计并实现语法分析和语义处理程序	16
2.4.1 stack 的说明	18
2.4.2 Parser(语法分析器) 的实现	18
3 实验结果	20

3.1 新增测例说明	20
3.2 测试结果	21
3.3 程序运行截图	22
4 心得体会	23

1 程序结构设计

按照实验文档要求:

- 完成的所有源程序代码，全部存放在 `src\parser` 文件夹中。

图 1: requirement

如下图,(由于代码文件是最后才移进 parser 去的,bin 里面的目录结构不对).

```
liyj@LiYJ: ~          命令提示符          Windows PowerShell
卷 系统 的 文件夹 PATH 列表
路径列表为 C:\Windows\system32
C:\USERS\LiYJ\Desktop\COMPILER\LAB\LAB2\LABINSTRUMENT02-EXPREVAL\21307099_李英骏
+--idea
|   +--bin
|   |   +--arithmetic
|   |   +--DFA
|   |   +--exceptions
|   |   +--junit
|   |   +--parser
|   |   +--scanner
|   |   +--test
|   |   +--token
|   |       +--Function
|   |       +--Operator
|   |       +--Symbol
|   |       +--Value
|
|   +--doc
|       +--arithmetic
|       +--DFA
|       +--legal
|       +--parser
|       +--resource-files
|       +--resources
|       +--scanner
|       +--script-files
|       +--tokens
|           +--Function
|               +--Operator
|               +--Symbol
|               +--Value
|
|   +--out
|       +--production
|           +--实验软类型 (基于表达式的计算器ExprEval)
|               +--arithmetic
|               +--DFA
|               +--parser
|               +--scanner
|               +--test
|               +--token
|                   +--Function
|                       +--Operator
|                       +--Symbol
|                       +--Value
|
+--ref
    +--exceptions
    +--gui
    +--resources
    +--test
+
+--src
    +--parser
        +--arithmetic
        +--DFA
        +--scanner
        +--test
        +--token
            +--Function
            +--Operator
            +--Symbol
            +--Value
+
+--testcases
(base) PS C:\Users\liyj\Desktop\Compiler\LAB\LAB2\LabInstrument02-ExprEval\21307099_李英骏> |
```

图 2: File Tree

其中

- `src/parser/arithmetic` 是执行计算的代码
- `src/parser/DFA` 是 DFA 的代码
- `src/parser` 是语法分析器的代码
- `src/parser/scanner` 包是词法分析器的代码
- `src/parser/test` 是写代码时使用的 Junit 测试
- `src/parser/token` 是 token 的代码
- `testcases` 中新增的 `mytest.xml` 和 `./mytest.bat`

2 实验过程和核心代码

为了截图能尽可能截到更多代码, 下面的代码截图时基本去掉了注释, 最后的代码是有 javadoc 注释的.

2.1 语法的二义性

原始的 BNF 显然具有二义性, 因为它没有定义任何算符的优先级和结合性, 以 $decimal^{decimal^{decimal}}$ 为例:

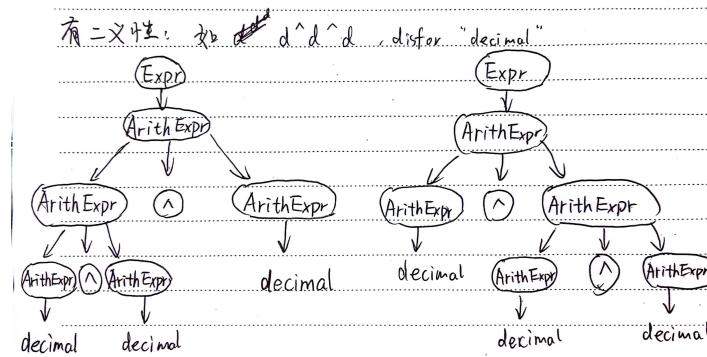


图 3: BNF

解析 只要设定了合适的优先级和结合性, 就可以保证语法树的唯一性. 按题目要求, 我们根据下图的优先级和结合性即可解析二义性.

另外, 表中未给出的部分:decimal 和 boolean 为最高优先级, 终结符 \$ 的最低, 逗号的优先级仅次于终结符.

级别	描述	算符	结合性质
1	括号	()	
2	预定义函数	sin cos max min	
3	取负运算 (一元运算符)	-	右结合
4	求幂运算	^	右结合
5	乘除运算	* /	
6	加减运算	+ -	
7	关系运算	= <> < <= > >=	
8	非运算	!	右结合
9	与运算	&	
10	或运算		
11	选择运算 (三元运算符)	? :	右结合

图 4: Priority

2.2 设计并实现词法分析程序

2.2.1 DFA 的设计

从文档中提取支持的表达式，语言的词法规则，并绘制识别其中所有合法单词的有限自动机，如图（下面有用手画的更美观图8）：

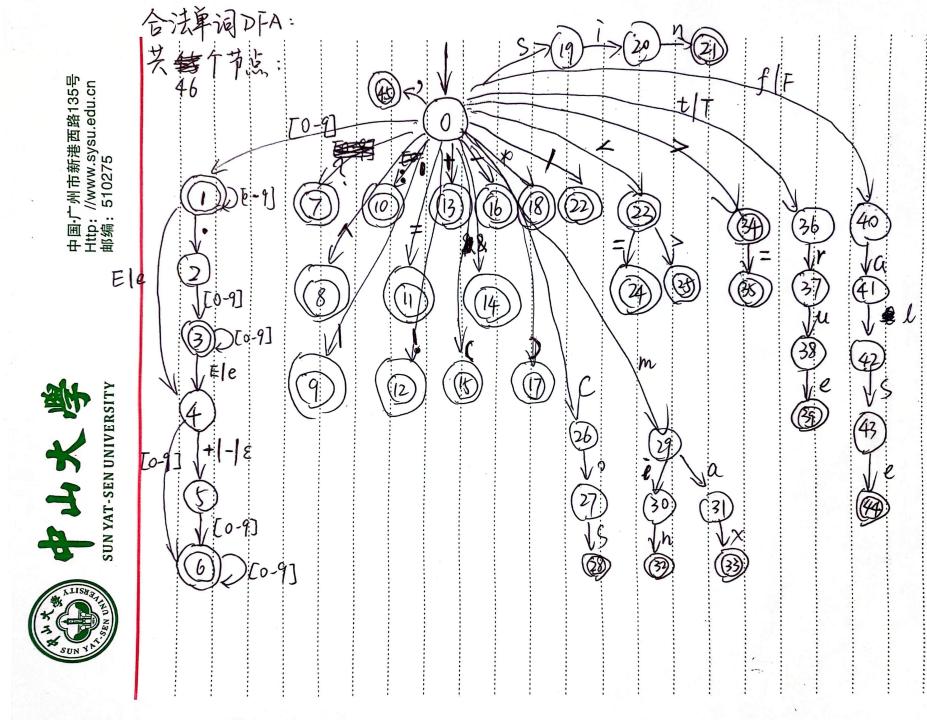


图 5: DFA

其中多出来的逗号运算符,”用于 min 和 max 中, 其优先级显然应该是最低的(仅次于 \$, 因为它用于分割两个表达式运算结果)

2.2.2 运算符的分类

我们把运算符分为以下 9 类:

- Values

1. Numerical Constants (Decimal Values): decimal, 满足下图:

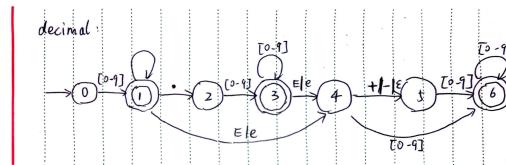


图 6: Decimal

2. Boolean Values: true, false, True, False

- Operators

1. Unary Operators: - (negative sign), !
2. Binary Operators: +, - (minus sign), *, /, ^, <, >, <=, >=, <>, =,&, |
3. Ternary Operator: ? :

- Functions

1. Functions: sin, cos, max, min

- Others

1. Parentheses: (,)
2. Comma: , (used in min and max)
3. End of Expression (EoE): \$

2.2.3 Token 的设计与实现

我们采用接口-实现的方式来设计 Token, 即: Token 是一个 Interface, 在每一个大类 (Operator Symbol Value) 中, 我们实现一个 Token 的 implement 作为大类的主类, 然后实现主类的子类, 如下图所示:

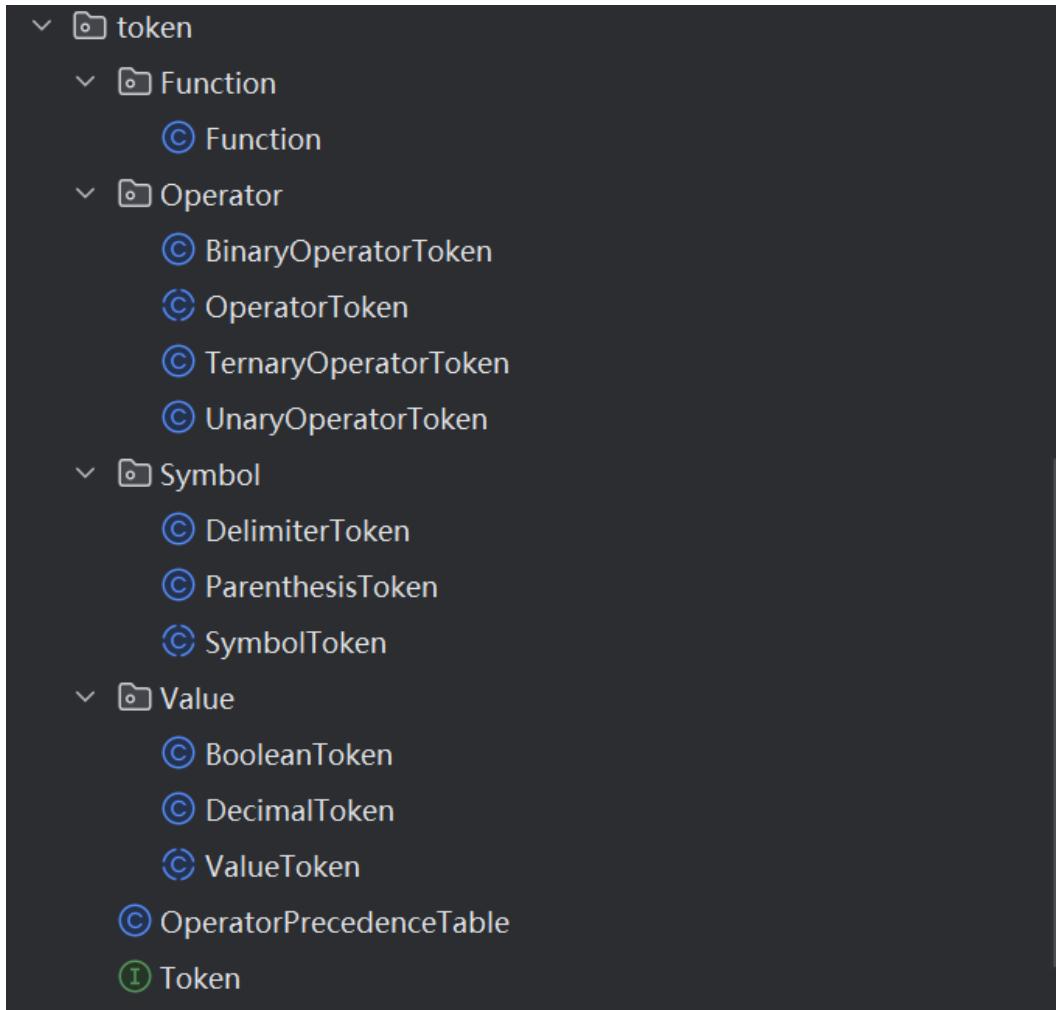


图 7

2.2.4 DFA 的代码实现

代码中实际使用的 dfa, 即把上面的 [0-9] 拆开 (由 scanner.DFA →toGraphviz() 方法生成代码, 在WebGraphviz中生成图像. 在 DFA.java 中运行 main 方法即可):

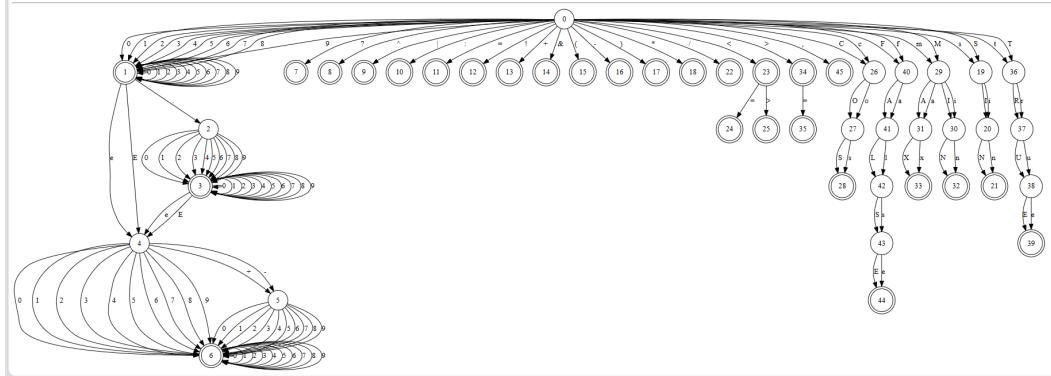


图 8: DFA2

DFA 节点如下图:

```
18  public class DFANode { 5个用法
19      private final int state; 2个用法
20      private final boolean isAccept; 2个用法
21      private final Map<Character, List<Integer>> edges; 3个用法
22      private final String type; 3个用法
23      public DFANode(int state, boolean isAccept, String type) { 1个用法
24          this.state = state;
25          this.isAccept = isAccept;
26          this.edges = new HashMap<>();
27          this.type = type;
28      }
29
30      public void addEdge(char symbol, int toState) { 1个用法
31          edges.computeIfAbsent(symbol, k -> new ArrayList<>()).add(toState);
32          // 检查边的添加
33          //System.out.println("Adding edge from state " + state + " to state " + toState + " with
34      }
```

图 9: DFA Node

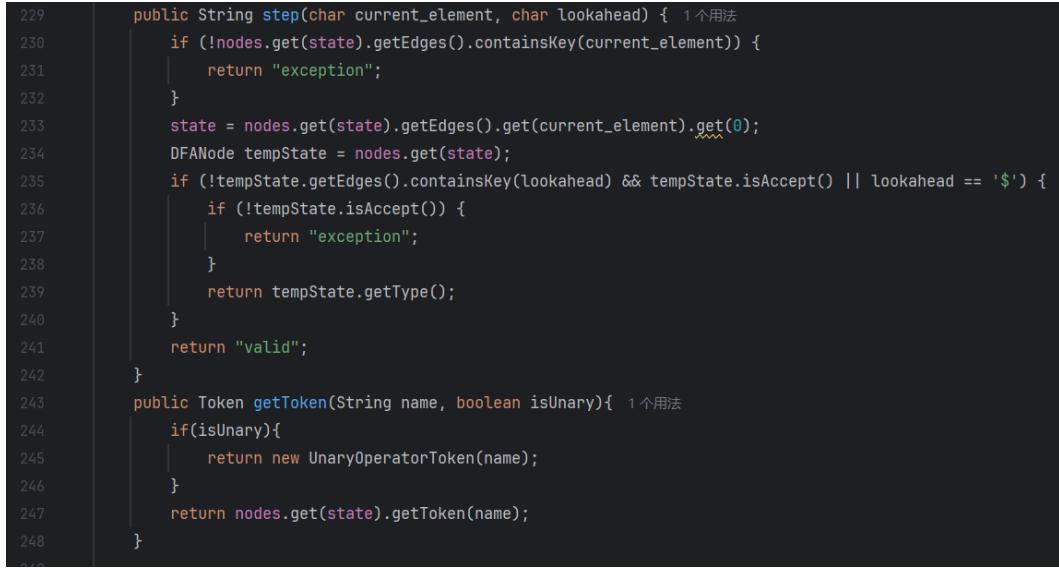
每个节点根据类型返回不同的 Token, 如下图所示:



```
52     public Token getToken(String name){ 1个用法
53         return switch (type) {
54             case "decimal" -> new DecimalToken(name);
55             case "boolean" -> new BooleanToken(name);
56             case "ternary" -> new TernaryOperatorToken(name);
57             case "arithmetic", "relation" -> new BinaryOperatorToken(name);
58             case "unary" -> new UnaryOperatorToken(name);
59             case "comma" -> new DelimiterToken(name);
60             case "parenthesis" -> new ParenthesisToken(name);
61             case "function" -> new Function(name);
62             default -> null;
63         };
64     }
```

图 10: NODE Token

然后在 DFA 中进行转移和 Token 的生成, 如下图所示:



```
229     public String step(char current_element, char lookahead) { 1个用法
230         if (!nodes.get(state).getEdges().containsKey(current_element)) {
231             return "exception";
232         }
233         state = nodes.get(state).getEdges().get(current_element).get(0);
234         DFAState tempState = nodes.get(state);
235         if (!tempState.getEdges().containsKey(lookahead) && tempState.isAccept() || lookahead == '$') {
236             if (!tempState.isAccept()) {
237                 return "exception";
238             }
239             return tempState.getType();
240         }
241         return "valid";
242     }
243     public Token getToken(String name, boolean isUnary){ 1个用法
244         if(isUnary){
245             return new UnaryOperatorToken(name);
246         }
247         return nodes.get(state).getToken(name);
248     }
```

图 11: DFA Token

2.2.5 实验文档中关注的问题

1. 如何处理对预定义函数名和布尔常量的识别 由上面的 DFA 即可识别.
2. 如何处理科学记数法表示的数值常量 如上图6, 在 DFA 中进行状态转移, 从而提取科学计数法的系数和指数.
3. 如何处理字符串的边界 我们模仿 LAB1, 在设计一个 lookahead(非 static) 变量, 然后在 DFA 中转移它, 即可通过在 DFA 中的状态知道字符串是否到达边界.

2.2.6 Scanner(词法分析器) 的实现

Scanner 负责将输入的字符流通过 DFA 转为 Token 队列并检测词法错误(也可以检测出空输入的语法错误). 如下图:

```

    public class scanner {
        public static void main(String[] args) {
            // 测试表达式
            String expression = "-31 + 5 * ( 2 - 8e1 ) < sin(-2.1)? 2 : 1";
            scanner myScanner = new scanner(expression);
            ArrayList<Token> tokens = myScanner.scan();
            // 输出扫描结果
            for (Token token : tokens) {
                System.out.println(token);
            }
        }
    }
  
```

图 12: Scanner Test

下图是 Scanner 的主要功能方法 scan, 详见注释:

```

    public class Scanner {
        /*
        * 扫描表达式
        */
        public ArrayList<Token> scan() throws ExpressionException {
            // 1. 读取表达式
            int expressionLength = formula_stream.length(); // 输入表达式的长度
            int index = 0; // 当前字符索引

            StringBuilder currentToken = new StringBuilder(); // 当前扫描到的token
            dfa.reset(); // 重置dfa状态
            boolean isLetter = false; // 当前token是否以字母开头
            boolean isDecimal = false; // 当前token是否以数字开头

            while (index < expressionLength) {
                char now = formula_stream.charAt(index); // 当前字符
                char lookahead = index + 1 < expressionLength ? formula_stream.charAt(index + 1) : '$'; // 下一个字符

                // 真过空格
                if (now == ' ') {
                    index++;
                    continue;
                }

                // 如果识别出开始状态, 检测当前字符是否是字母或数字
                if (dfa.getState() == 0) {
                    if (Character.isLetter(now))
                        isLetter = true;
                    else if (Character.isDigit(now) || now == '.')
                        isDecimal = true;
                }

                currentToken.append(now); // 将当前字符添加到当前token中
                String tokenType = dfa.step(now, lookahead); // 通过dfa的step方法获取token类型
            }
        }
    }
  
```



```

    public class Scanner {
        public ArrayList<Token> scan() throws ExpressionException {
            // 1. 读取表达式
            String tokenType = dfa.step(now, lookahead); // 通过dfa的step方法获取token类型

            // 如果dfa读到合法元素token, 继续扫描下一个字符
            if (tokenType.equals("valid")) {
                index++;
                continue;
            }

            // 如果dfa读到非法元素token, 抛出相应的异常
            else if (tokenType.equals("exception")) {
                if (isLetter)
                    throw new IllegalIdentifierException(); // 非法标识符异常
                else if (isDecimal)
                    throw new IllegalDecimalException(); // 非法小数异常
                else
                    throw new IllegalSymbolException(); // 非法符号异常
            } else {
                // 如果dfa读到合法元素token, 将其添加到tokens的队列中
                tokens.add(dfa.getToken(currentToken.toString(), isLetter(currentToken.toString())));
                dfa.reset(); // 重置dfa状态
                currentToken = new StringBuilder(); // 重置当前token
                isLetter = false; // 重置字母标志
                isDecimal = false; // 重置小数标志
                index++;
            }

            // 如果表达式为空, 抛出异常
            if (tokens.isEmpty())
                throw new EmptyExpressionException();
        }
    }
  
```

图 13: scan

2.3 构造算符优先关系表

我们根据表4来构造算符右舷关系表，显然在此阶段可以部分识别出语法异常（除了空表达式）：

```
|-- SyntacticException (表达式中的语法错误)
|   |
|   |-- MissingOperatorException (缺少运算符)
|   |-- MissingOperandException (缺少运算量)
|   |-- MissingLeftParenthesisException (遗漏了左括号)
|   |-- MissingRightParenthesisException (遗漏了右括号)
|   |-- FunctionCallException (预定义函数调用的语法形式错误)
|   |-- TrinaryOperationException (三元运算的语法形式错误)
|   |-- EmptyExpressionException (输入表达式为空)
```

图 14: Enter Caption

我们把前 6 个异常记为 E1-E6，并在表中标记。用下面的程序生成优先关系表（为了方便用优先级查表，我对上表中优先级有略微修改，但前后相对顺序是不变的）：

```
15     static {
16         // 定义运算符的优先级
17         precedence.put("boolean", 0);
18         precedence.put("decimal", 1);
19         precedence.put("(", 2);
20         precedence.put(")", 3);
21         precedence.put("function", 4);
22         precedence.put("negative", 5);
23         precedence.put("^", 6);
24         precedence.put("*", 7);
25         precedence.put("/", 8);
26         precedence.put("+", 9);
27         precedence.put("-", 10);
28         precedence.put("relation", 11);
29         precedence.put("!", 12);
30         precedence.put("&", 13);
31         precedence.put("|", 14);
32         precedence.put "?", 15);
33         precedence.put ":", 16);
34         precedence.put " ", 17);
35         precedence.put "$", 18);
```

图 15: Procedence

```

34     // 定义运算符的结合律（左结合为 true，右结合为 false）
35     associativity.put("function", false);
36     associativity.put("negative", false);
37     associativity.put("!", false);
38     associativity.put "?", false);
39     associativity.put "^", false);
40     associativity.put ",", false);
41
42     associativity.put "*", true);
43     associativity.put "/", true);
44
45     associativity.put "+", true);
46     associativity.put "-", true);
47
48     associativity.put("relation", true);
49     associativity.put "&", true);
50     associativity.put "|", true);
51     associativity.put ":", true);
52     associativity.put "$", true);
53 }

```

图 16

异常检测, 详见注释:

```

public class OperatorPrecedenceTable {
    private static String errorDetected(String op1, String op2) { 1 个方法
        // E1: Missingoperator
        // 连续两个或者以上的操作符
        if (op1.equals("decimal") || op1.equals("boolean")) {
            if (op2.equals("decimal") || op2.equals("boolean") || op2.equals("!")) {
                return "E1";
            }
        }
        if (op1.equals("")) && (op2.equals("Boolean") || op2.equals("decimal") || op2.equals("(") || op2.equals("function") || op2.equals("!")) {
            return "E1";
        }
        // E2: Missingoperand
        if (op1.equals(")")) {
            if (op2.equals(")")){
                return "E2";
            }
        }
        // E3: LeftParenthesis
        // 此阶段只有在$后的才能判断缺少左括号
        if (op1.equals("$") && op2.equals(")")) {
            return "E3";
        }
        // E4: RightParenthesis
        // 此阶段只有在(后的才能判断缺少右括号
        if (op1.equals("(") && op2.equals("$")) {
            return "E4";
        }
        // E5: FunctionCall
        // function后面只能出现左括号, 因此任何非左括号的跟随着都是E5
        if (op1.equals("function") && !op2.equals("(")) {
            return "E5";
        }
        // E6: TernaryOperation
        // 三目运算符: 的异常有
        // ( : -- $)
        // $ : -- $)
        if ((op1.equals(":") && op2.equals(":")) || (op1.equals("?") && op2.equals("$")) || (op1.equals("$") && op2.equals(":")) {
            return "E6";
        }
        return null;
    }
}

```

图 17: ErrorDetect

填充 shift/reduce/accept:

```

7     public class OperatorPrecedenceTable {
70      private static String getRelation(String op1, String op2) { 1个用法
71          String E = errorDetected(op1, op2);
72          if (E != null) {
73              return E;
74          }
75          if (op1.equals("$") && op2.equals("$")) {
76              return "acc";
77          }
78          // 左括号应shift
79          if (op1.equals("(") || op2.equals("(")) {
80              return "shift";
81          }
82          // 右括号说明子式结束 应reduce(一般情况下)
83          if (op1.equals(")") || (op2.equals(")") && !op1.equals(","))) {
84              return "reduce";
85          }
86          // ?后面不报错就应shift
87          if (op1.equals("?")){
88              return "shift";
89          }
90          // ,后面不报错就应shift
91          if (op1.equals(",")){
92              return "shift";
93          }
94          int prec1 = precedence.get(op1);
95          int prec2 = precedence.get(op2);
96          if (prec1 == prec2) {
97              Boolean assoc = associativity.get(op1);
98              return Boolean.TRUE.equals(assoc) ? "reduce" : "shift";
99          }
100         return prec1 < prec2 ? "reduce" : "shift"; // 优先级高的遇到低的shift(一般情况下)
101     }

```

7099_Liyj > src > creator > OperatorPrecedenceTable > getRelation

图 18: Precedence Table

构造表格如下 (在\src\token\OperatorPrecedenceTable.java 运行 main 即可):

	boolean	decimal	()	function	negative	^	*	/	+	-	relation	!	&		?	:	,	\$
boolean	E1	E1	E1	reduce	E1	reduce	reduce	reduce	reduce	reduce	reduce	reduce	E1	reduce	reduce	reduce	reduce	reduce	reduce
decimal	E1	E1	reduce	E1	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	E1	reduce	reduce	reduce	reduce	reduce	reduce
(shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	E6	shift	E4	reduce
)	E1	E1	E1	reduce	E1	reduce	reduce	reduce	reduce	reduce	reduce	reduce	E1	reduce	reduce	reduce	reduce	reduce	reduce
function	E5	E5	shift	E5	E5	E5	E5	E5	E5	E5	E5	E5	E5	E5	E5	E5	E5	E5	E5
negative	shift	shift	shift	reduce	shift	shift	shift	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce
^	shift	shift	shift	shift	reduce	shift	shift	shift	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce
*	shift	shift	shift	shift	shift	shift	shift	shift	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce
/	shift	shift	shift	shift	shift	shift	shift	shift	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce	reduce
+	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift
-	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift
relation	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift
!	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift
&	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift
	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift
?	shift	shift	shift	E2	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	E6	shift	shift
:	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift
,	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift
\$	shift	shift	shift	E3	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	shift	E6	shift	acc

图 19: Priority Table

getCodedPrecedenceTable 返回一个这样的二维数组:

\$	shift	shift	shift	E3	shift	accept																	
boolean	-1	-1	-1	1	-1	1	1	1	1	1	-1	1	1	1	1	1	1	1	1	1	1	1	1
decimal	-1	-1	-1	1	-1	1	1	1	1	1	-1	1	1	1	1	1	1	1	1	1	1	1	1
(0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-4
)	-1	-1	-1	1	-1	1	1	1	1	1	-1	1	1	1	1	1	1	1	1	1	1	1	1
function	-5	-5	0	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5
negative	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
*	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
/	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+	0	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
relation	0	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
!	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
&	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
?	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-6	0
:	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
,	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
\$	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-6	0	2

图 20: CodedPrecedenceTable

大致说明

1. 高优先级的运算符后面跟低优先级的运算符时 shift, 反之 reduce
2. 对角线上右结合运算符 shift, 左结合 reduce
3. 表的: ? function 行 (列) 关注了以下要求:

3.4.11 FunctionCallException 异常

在调用预定义函数时, 若函数调用的语法形式错误, 则应抛出此异常。例如, `sin()` 和 `cos()` 函数的参数表中多于一个参数时应抛出此异常; 所有函数调用缺少左括号时, 亦抛出此异常(注意, 此时并不当作缺少左括号来处理)。

还应注意, 当所有预定义函数的参数个数不足(未达到该函数最低要求的参数数目)时, 你的程序应抛出 `MissingOperandException`, 而不是抛出此异常。

运行以下测试用例应抛出 `FunctionCallException` 异常:

- `sin(2, 1)`
- `max5, 6, 8`

3.4.12 TrinaryOperationException 异常

当三元运算符的“?”和“:”出现不配对时, 应抛出此异常, 例如表达式 `6 ? 7 : 7 : 9`; 当三元运算符与括号的匹配出问题时, 亦应抛出此异常, 例如表达式 `5 ? (8 : 8)`。

图 21: requirement

如何处理一些较为敏感的关系：

- 一元取负运算符和二元减法运算符
 1. 对于一元取负运算符, 其后方可以连接除逗号和终结符以外所有的运算符, 但前一个 token 只能是 operators,functions, 左括号, 逗号.
 2. 对于二元减法运算符, 其前方的一个 token 只能是 value, 右括号.

如下图所示:

```
private boolean isUnary(String current){ 0个用法
    if(current.equals("+")){
        return true;
    }
    if(current.equals("-")){
        // 如果是首个token则必然是负号,
        // 如果不是则看前一个token
        int token_length = formula_stream.length();
        if(token_length>0){
            Token lasttoken = tokens.getToken(token_length-1);
            if(lasttoken.getType().equals("decimal")|||
               lasttoken.getType().equals(".")|||
               lasttoken.getType().equals("boolean")){
                return false;
            }
            return true;
        }
        return true;
    }
}
```

图 22

- 三元运算符与其他运算符之间的关系 由文档中的要求:

在输入表达式中可使用关系运算和逻辑运算写出布尔表达式, 这些布尔表达式只能作为三元运算“?”中的第一个子表达式, 不可直接作为计算结果。可使用圆括号确定算

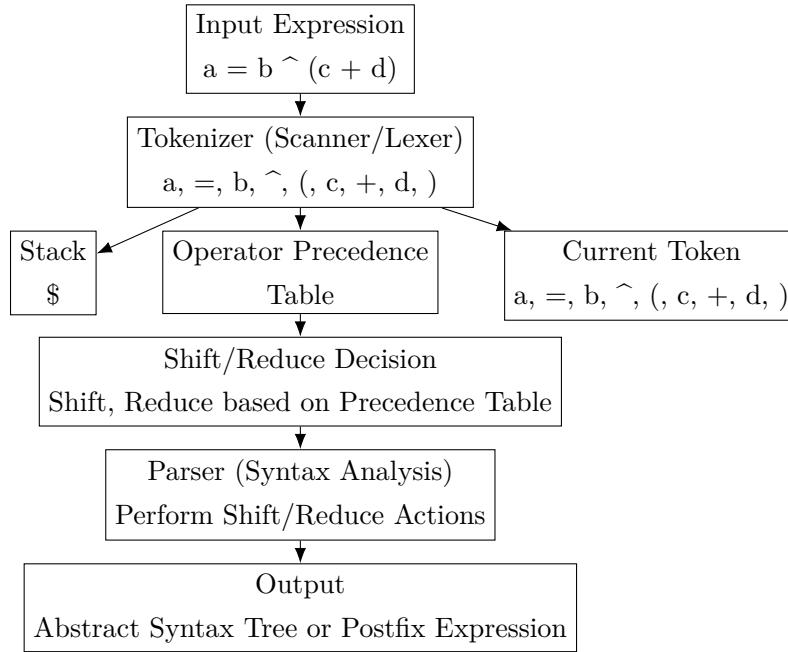
图 23: Triple

需要确保 boolean value 只出现在三元运算符的第一个子表达式中

- 预定义函数与其他运算符之间的关系 需确保预定义函数中的参数量正确. 如 sin cos 需要 1 个参数,min 和 max 需要至少 2 个参数

2.4 设计并实现语法分析和语义处理程序

OPP 维护输入队列, 运算符栈和上述优先表, 通过在表中查找 [栈顶][lookahead] 处的值来发出下一步行为. 详见下方2.4.2



算法如下:

Algorithm 1 Operator Precedence Parsing

```

1: stack.push("$")
2: while true do
3:   top ← stack.top()
4:   lookahead ← input[0]
5:   if table[top][lookahead] == shift then
6:     shift()
7:     continue
8:   else if table[top][lookahead] == reduce then
9:     reduce()
10:    continue
11:   else if table[top][lookahead] == accept then
12:     accept()
13:     return
14:   else if table[top][lookahead] == exception then
15:     throw exception()
16:   end if
17: end while

```

shift 和 reduce 的算法如下:

Algorithm 2 Shift 操作

```
1: function SHIFT
2:   stack.push(input[0])
3:   input.erase(0)
4: end function
```



```
109  /**
110  * 进行移入操作
111  *
112  * @param lookahead 前瞻符号
113  * @throws IllegalSymbolException 当符号非法时抛出异常
114  */
115  private void shift(Token lookahead) throws IllegalSymbolException { 1个用法
116      stack.add(AddInStack(lookahead));
117      input_buffer.removeFirst();
118  }
```

图 24: Parser->shift()

Algorithm 3 Reduce 操作

我们采用一个 Reducer Class 来进行规约操作, 见2.4.2节

```
1: function REDUCE
2:   while table[stack.top()][input[0]] == reduce do
3:     result ← calculator(stack)
4:     stack.pop()
5:     stack.push(result)
6:   end while
7: end function
```



```
93  /**
94  * 进行规约操作
95  *
96  * @throws ExpressionException 当表达式出错时抛出异常
97  */
98  private void reduce() throws ExpressionException { 1个用法
99      Token stackTop = getStackTop();
100     Token lookahead = input_buffer.getFirst();
101     int action = operator_precedence_table[stackTop.getPriority()][lookahead.getPriority()];
102     while (action == 1) {
103         stack = new Reducer(stack).calculate(stackTop.getType());
104         stackTop = getStackTop();
105         action = operator_precedence_table[stackTop.getPriority()][lookahead.getPriority()];
106     }
107 }
```

图 25: Parser->reduce()



```
53  /**
54  * 规约操作，将结果替换对应位置的令牌
55  *
56  * @param location 规约位置
57  * @param times 规约次数
58  * @param result 规约结果
59  */
60  private void reduce(int location, int times, Token result) { 6个用法
61      for (int i = 0; i < times; i++) {
62          stack.remove(location);
63          stack.add(location, result);
64      }
65 }
```

图 26: Reducer->reduce()

2.4.1 stack 的说明

我们对 stack 做如下的设计:

1. 只处理终结符 忽略非终结符可以简化对栈的操作。在规约操作中，只需要考虑终结符，可以更容易地进行计算和规约。
2. 统一操作符行为：通过明确每个操作符的行为，可以更方便地进行各种操作符的处理，提高算法的效率。

2.4.2 Parser(语法分析器) 的实现

Parser 负责计算 Token 队列 (过程中转为语法树)，并检测语法错误。Parser 维护一个 stack 和一个输入 buffer，并根据优先表进行 shift 和 reduce 操作 (opp):

reduce 方法 `reduce` 规约操作的角色，将表达式中的部分符号按照优先级进行计算

操作流程

1. 获取堆栈顶部的终结符和输入缓冲区中的第一个 lookahead token。
2. 在操作符优先级表中查找这两个 token 对应的动作。
3. 如果这个动作指示进行规约 (`action` 等于 1)，那么我们就使用 `Reducer` 类对堆栈中的元素进行计算，并更新堆栈。这个过程会持续，直到不再需要规约

shift 方法 `shift` 方法用于将输入缓冲区中的 token 移入到堆栈中

操作流程

1. 根据 lookahead token 的类型，调用 `addInStack` 方法，将新的 token 对象加入栈。
2. 然后 token 从输入缓冲区中移除

opp 方法 `opp` 方法是整个解析过程的核心，负责根据操作符的优先级决定是进行移入、规约或者返回最终计算结果。

操作流程

1. 在开始时，堆栈中会加入一个表示终止的分隔符 "\$"。
2. 解析器会不断处理，直到整个表达式被解析完毕。具体步骤包括：
 - (a) 获取当前堆栈顶部的终结符以及输入缓冲区中的 lookahead token。
 - (b) 查找两者在优先级表中对应的动作 `action`。

(c) 根据 `action` 的值, 执行相应的操作:

- 若为 移入 (0): 执行 `shift`, 加入新的 token。
- 若为 规约 (1): 执行 `reduce`, 更新堆栈。
- 若为 完成解析 (2): 调用 `getAnswer`, 返回计算结果。
- 若为 异常情况 (负数): 根据错误类型抛出相应异常。

表达式的计算和 Reducer

我们采用下表的规则进行 reduce, 注意上面的 Token 中只有 decimal 和 boolean 可能为 non-terminal, 所以大体上只有这两种 Token 可能会被 reduce, 其他 Token 只会被 shift.

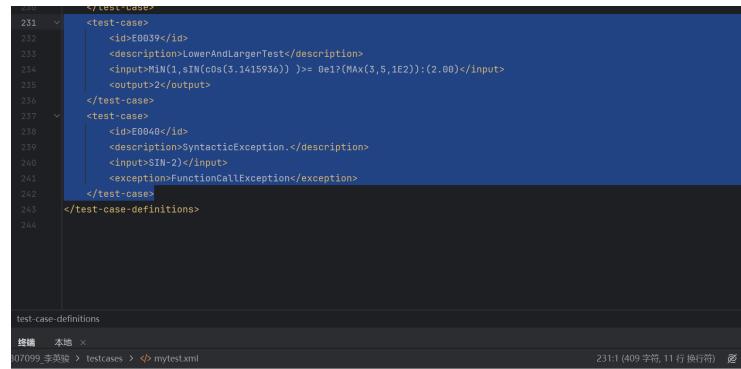
输出类型	操作	返回值	错误处理
Number/Boolean	无需操作	该元素非终结符的恒等	无
Unary Operators	弹出栈顶上的一个非终结符元素	该元素取负或逻辑非后的副本	如果栈顶上的元素缺失, 报告缺失操作数错误; 如果取负的元素不是数字或逻辑非的元素不是布尔值, 报告类型不匹配错误
Binary Operators	弹出栈顶上下的一个非终结符元素	对弹出元素进行运算的结果的副本	如果栈顶上下的元素缺失, 报告缺失操作数错误; 如果算术运算符和比较运算符用于非数字元素/逻辑运算符用于非布尔元素, 报告类型不匹配错误
Ternary Operations	找到栈顶下方的第一个问号, 提取问号前的元素作为 A; 提取问号后的元素作为 B; 提取冒号后的元素作为 C	如果 A 为真, 返回 B 的非终结符副本; 否则, 返回 C 的非终结符副本	如果没有问号, 报告三元运算符错误; 如果问号和冒号之间的元素数量不是 1, 报告缺失操作符错误; 如果 A、B 或 C 缺失或不是非终结符, 报告缺失操作数错误; 如果 A 不是布尔值, 报告类型不匹配错误
Parentheses	找到右括号下方的第一个左括号, 提取括号之间的所有元素作为参数。如果左括号下方的元素为空或不是函数, 执行常量操作; 否则, 执行函数操作。	常量操作无操作, 函数操作对参数执行	如果常量操作的参数数量大于 1, 报告缺失操作符错误; 如果参数为空或不是非终结符, 报告缺失操作数错误; 函数操作根据参数进行判断, 一元函数通过方法判断, 多参数函数要求参数交替为数字和逗号
Others	无需操作	无	报告缺失操作符错误

3 实验结果

3.1 新增测例说明

- **mytest.xml** 测试了所有的运算符, 包括一元运算符, 二元运算符, 三元运算符, 函数运算符, 括号运算符, 逗号运算符, 终结符, 非终结符, 以及各种可能的异常情况.
- **mytest.bat** 用于运行 mytest.xml
- **./src/paser/test** 是写代码时使用的 Junit 测试, 里面加了几个调试过程中没有 pass 的测试用例, 方便调试.

例如以下两个 mytest.xml 中的测例检测了文档中要求的**不区分大小写**和**函数缺少左括号应当抛出 FunctionCallException**:



The screenshot shows a code editor displaying XML test cases. The XML code includes several test cases, one of which is highlighted with a blue selection bar. The highlighted test case is as follows:

```
<test-case>
    <id>E0039</id>
    <description>LowerAndLargerTest</description>
    <input>MIN(1, SIN(COS(3.1415936)))>= 0e1?(&#4x(3,5,1E2)):(2.00)</input>
    <output>2</output>
</test-case>
<test-case>
    <id>E0040</id>
    <description>SyntacticException.</description>
    <input>SIN-2</input>
    <exception>FunctionCallException</exception>
</test-case>
```

The code editor interface includes a status bar at the bottom showing file information: 107099_李莎璇 > testcases > <> mytest.xml, 231:1 (409 字符, 11 行 执行符).

图 27: test cases

3.2 测试结果

我的代码通过了所有能想到的测试用例:

```
Passed !
-----
> Statistics Report (8 test cases):
>
    Passed case(s): 8 (100.0%)
    Warning case(s): 0 (0.0%)
    Failed case(s): 0 (0.0%)
-----
请按任意键继续. . .
21307099_李英骏 > testcases > mytest.xml
```

(a) Simple

```
-----
Statistics Report (16 test cases):

    Passed case(s): 16 (100.0%)
    Warning case(s): 0 (0.0%)
    Failed case(s): 0 (0.0%)
-----
```

(b) Standard

```
-----
Statistics Report (40 test cases):

    Passed case(s): 40 (100.0%)
    Warning case(s): 0 (0.0%)
    Failed case(s): 0 (0.0%)
-----
```

(c) MyTest

图 28: Testing Results

3.3 程序运行截图

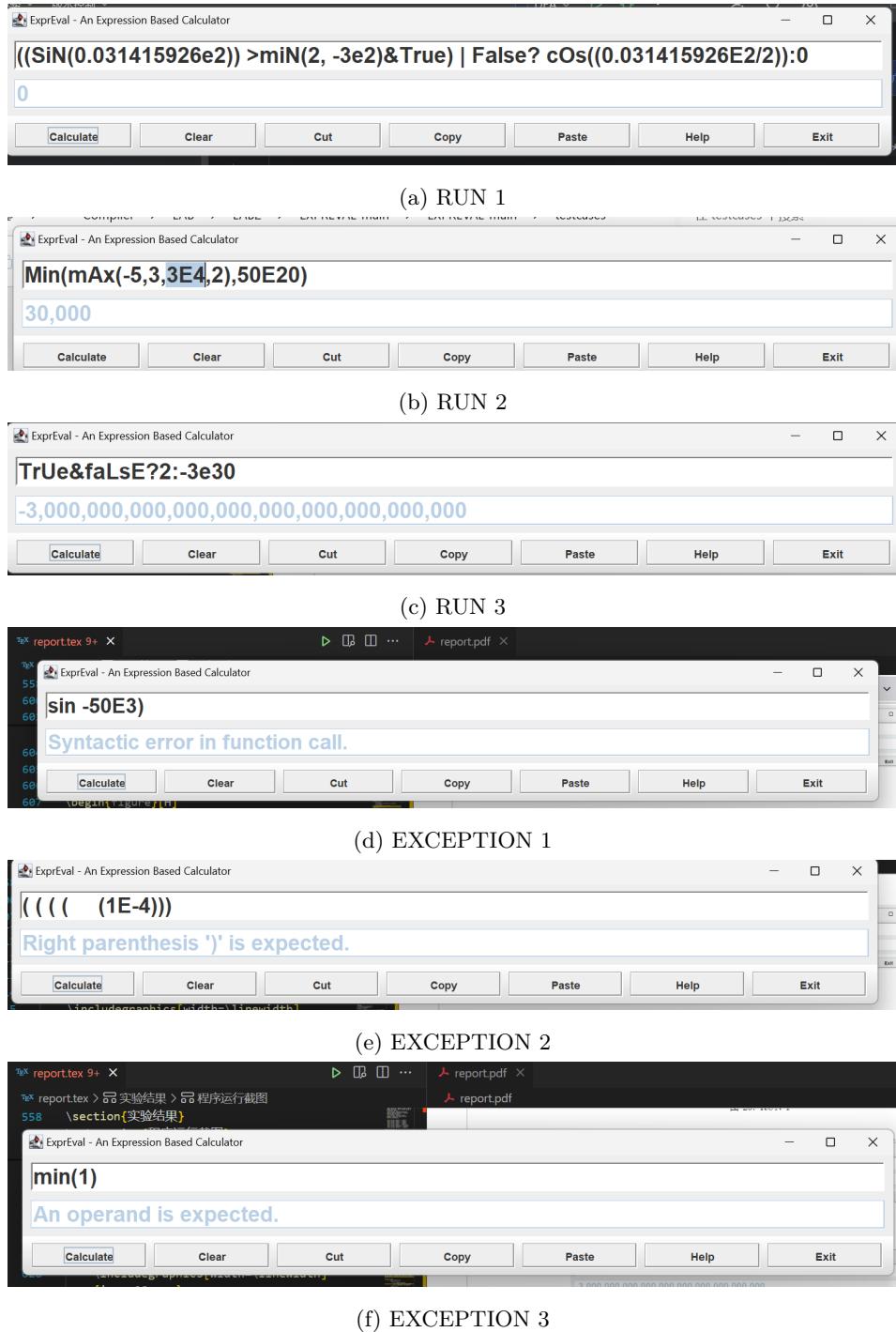


图 29: Runs and Exceptions

4 心得体会

- 对于编译原理的理解 通过这次实验, 我对编译原理的一些基本概念有了更深的理解, 如 DFA, OPP, 语法分析等, 更深刻地认识到了如何处理各种特例中的细节.
- 对于 Java 的理解 通过这次实验, 我对 Java 的一些特性有了更深的理解, 如接口, 继承, 多态等. 由于之前 Java 写得比较少, 这次实验阅读了大量 Java 的文档, 对 Java 的理解有了很大的提高.
- 对于代码的理解 通过这次实验, 我对代码的设计和实现有了更深的理解, 如如何设计一个类, 如何设计一个接口等. 提高了我的软件工程水平.