# expression-analyzer 设计文档

作者 Email：shanxuecheng@163.com，QQ：502817600



# 简介

在本表达式解析器中，把表达式解析的实现分为两个部分：词法分析器和语法分析器。

词法分析器使用有限自动机实现。词法分析器的任务是识别出表达式中的符号（Token），即数字、变量、界符等等。词法分析器的结果就是这些 Token 按其原始顺序组成的序列。例如，表达式“a = 100;”，词法分析后的结果便是：变量 a、赋值操作符、常量 100、分号，它们都是由特定类型的 Token 对象表示的。此外，词法分析器需要能够定位表达式中的词法错误，并给出提示。

语法分析器接受词法分析器生成的 Token 序列，然后使用 LL(1)分析法和属性翻译文法解析 Token 序列，计算解析结果。语法分析器也需要定位 Token 序列中存在的语法错误，并给出提示。

下文将分别介绍词法分析器和语法分析器的实现。

# 词法分析

词法分析的目标就是使用有限自动机识别出表达式字符序列中的各类 Token。

词法分析器需要识别的 Token 包括：常量、变量、界符、关键字、函数。其中常量、变量具有数据类型，目前支持的数据类型包括：数字、日期、布尔、字符、字符串。

下面将描述有限自动机的设计和词法分析器主要的类和函数。

## 有限自动机的设计

有限状态自动机拥有有限数量的状态，每个状态可以迁移到零个或多个状态，输入字符决定迁移到哪个状态。有限状态自动机可以表示为一个有向图。词法分析器的有限自动机的设计如下图所示。

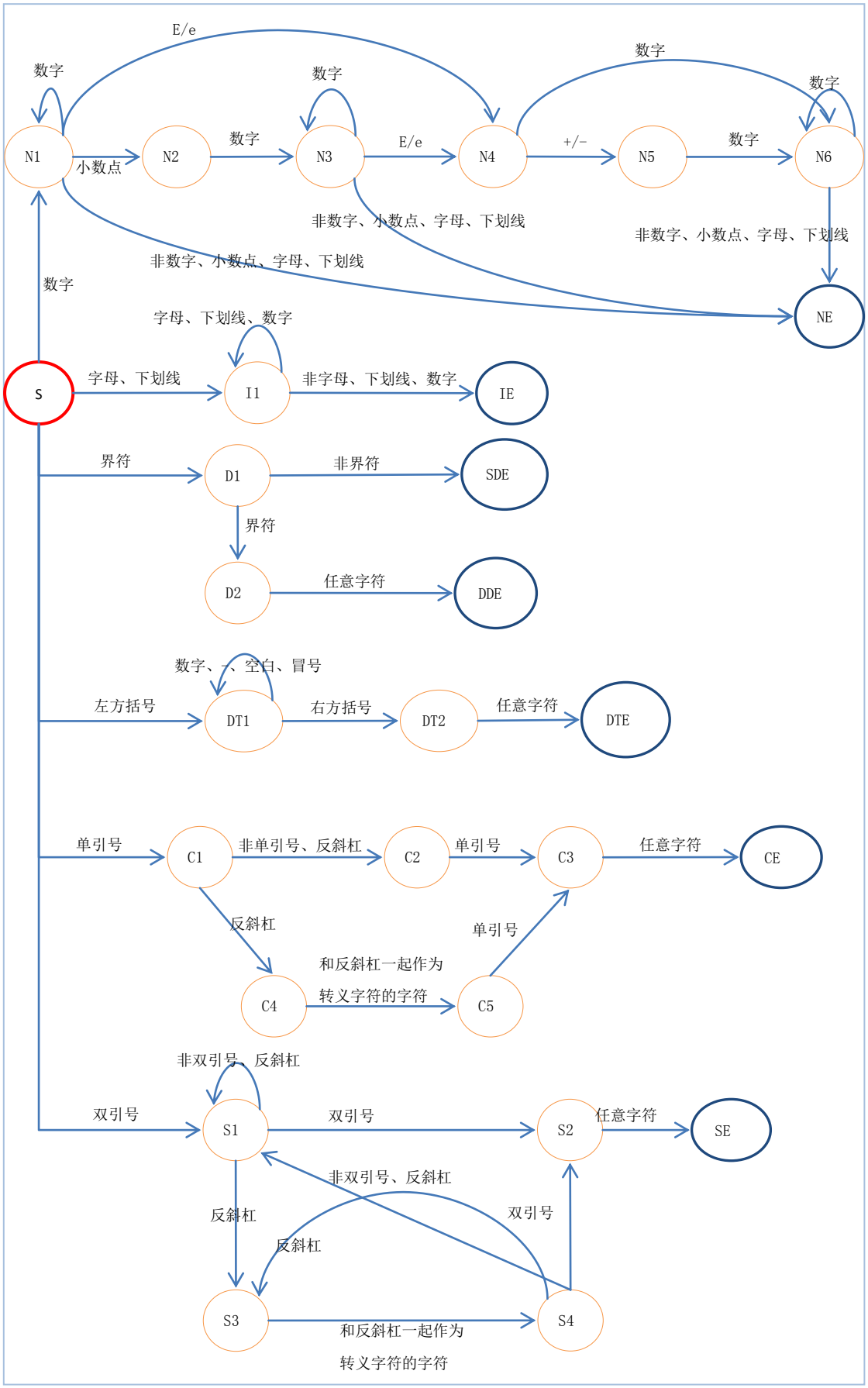



图 1 词法分析器的有限自动机

在上图中，圆圈表示有限自动机的状态，有向边表示输入字符。状态分为开始状态、中间状态和结束状态。开始状态只有一个，即图中红色标识的状态 S，结束状态有多个，即图中蓝色标识的 NE、IE 等，其余状态为中间状态。

每一个结束状态表示一种类型 Token 的识别结束，图中结束状态对应的 Token 类型从上至下依次为：数字（NE）、变量名（IE）、单字符界符（SDE）、双字符界符（DDE）、日期（DTE）、字符（CE）、字符串（SE）。其中变量名在识别过程中将会被区别为普通变量名、函数名还有布尔常量（true 或 false）。单字符界符是指+等，双字符界符是指>=等。

# 有限自动机的机内表示

在词法分析器中使用枚举定义有限自动机的状态代码。

枚举 neu.sxc.expression.lexical.dfa.DFAMidStateCode 定义了中间状态的代码(包括开始状态)，如 START 对应图 1 中的 S，NUMBER_1 对应图 1 中的 N_1。

枚举 neu.sxc.expression.lexical.dfa. DFAEndStateCode 定义了结束状态的代码，如 NUMBER_END 对应图 1 中的 NE，SINGLE_DELIMITER_END 对应图 1 中的 SDE。

中间状态使用类 neu.sxc.expression.lexical.dfa. DFAMidState 表示，如以下类图所示。

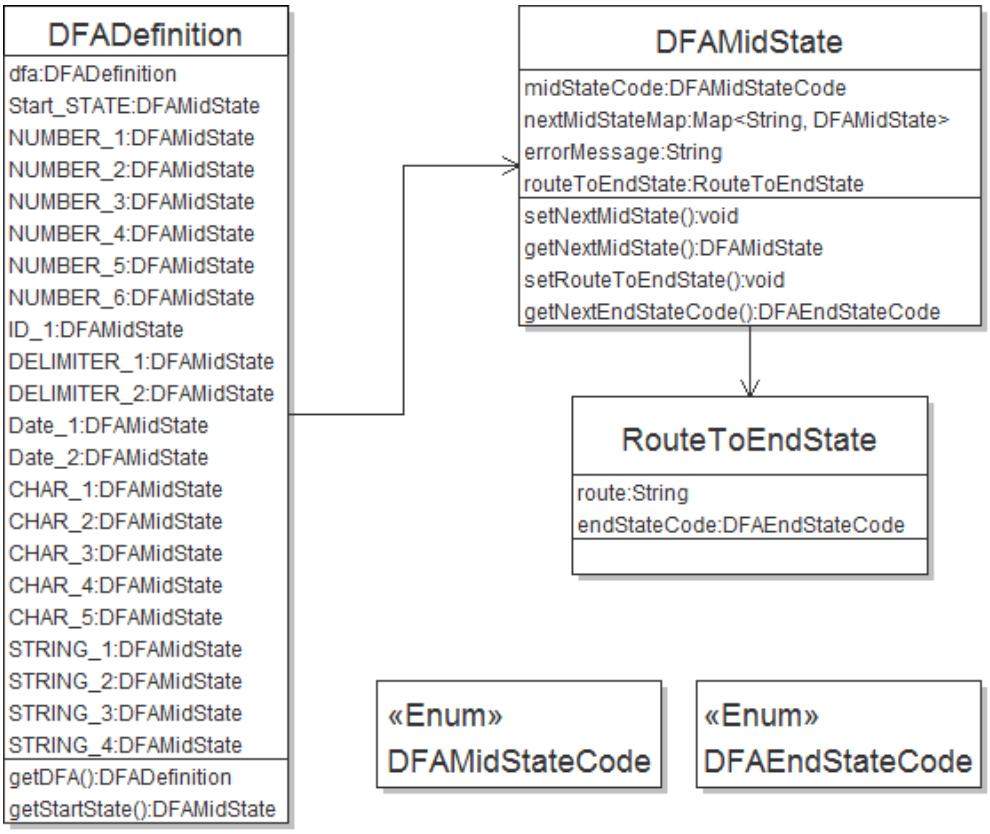



图 2 有限自动机类图

在词法分析器中，使用正则表达式表示有限自动机的有向边（即输入字符），具体定义见 neu.sxc.expression.lexical. LexicalConstants。

DFAMidState 将有向边和相应的迁移状态存放在 nextMidStateMap 中。DFAMidState 的属性 errorMessage 表示当输入字符在 nextMidStateMap 中无法找到匹配的有向边时将返回的错误信息。

DFAMidState 还持有 RouteToEndState 对象的引用，RouteToEndState 表示中间状态到结束状态的路由，它只有两个属性，route（中间状态到结束状态的有向边）、endStateCode（结束状态代码）。

DFADefinition 是一个单例的类，其中初始化了有限自动机的所有状态。

# 使用有限自动机进行词法分析

使用有限自动机进行词法分析，即从有限自动机的开始状态输入字符，每到达一个结束状态就说明一个 Token 的解析结束，然后继续从开始状态输入字符，循环此过程，直到输入表达式中的全部字符。其间如果出现词法错误（输入字符过程中不能够到达某一结束状态），就停止解析，并给出错误信息。

注意，如果在某一中间状态输入一个字符后到达结束状态，该字符不属于当前识别出的 Token 中的字符，它将作为识别下一 Token 的开始字符。

neu.sxc.expression.lexical.LexicalAnalyzer 中函数 doAnalysis()实现了上述解析过程。

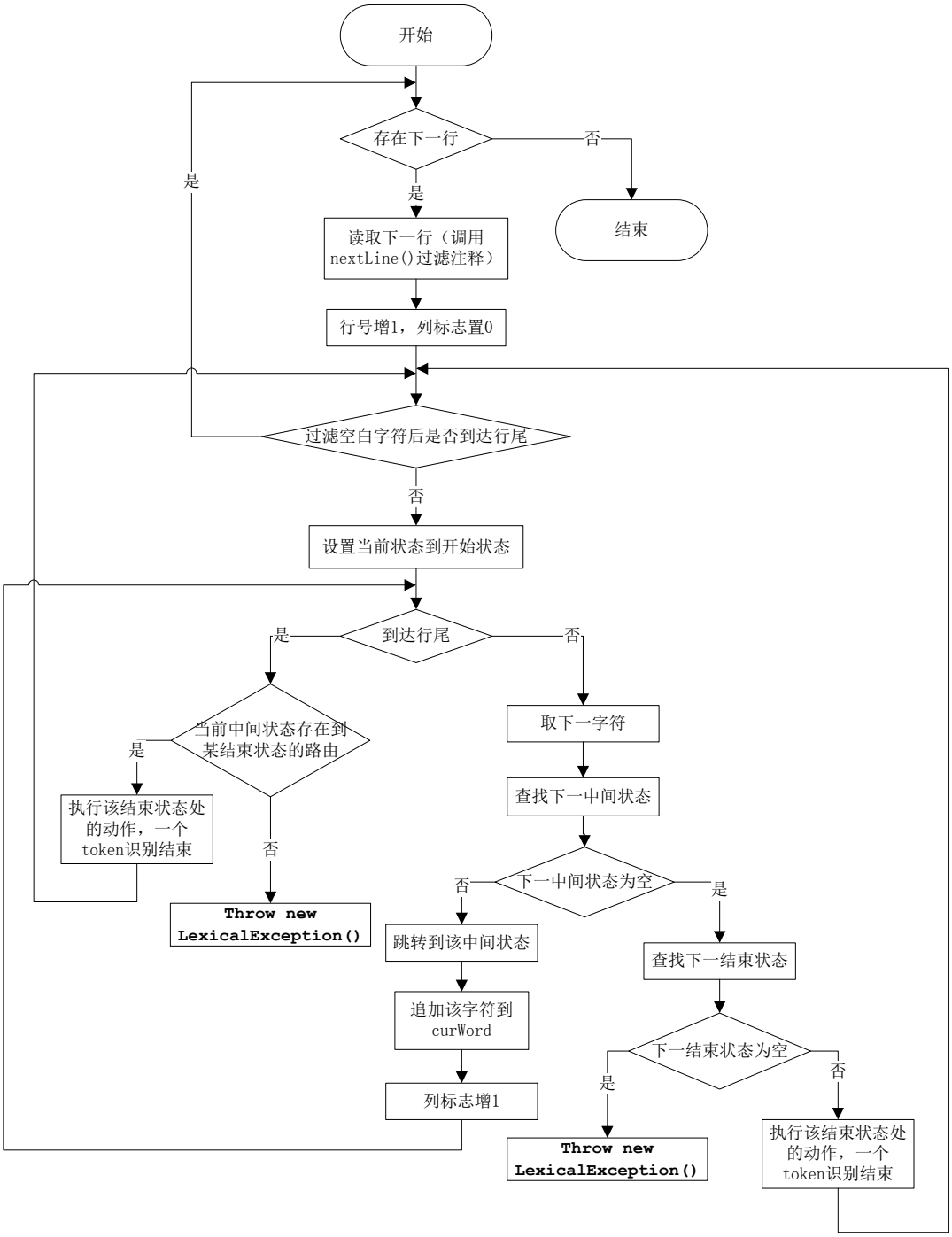



图 3 LexicalAnalyzer.doAnalysis()

# 语法分析

## LL(1)文法和 LL(1)分析法

语法分析器使用 LL(1)文法和 LL(1)分析法对词法分析器生成的 Token 序列进行解析。

LL(1)文法是指文法中，具有相同左部的各产生式，其选择集合不相交。LL(1)分析法是指从左到右扫描(第一个 L) 、最左推导(第二个 L)和只查看一个当前符号(括号中的 1)之意。

LL(1)分析法有三个基本要点：利用一个分析栈（语法栈），记录分析过程；利用一个分析表（LL(1)分析表），登记如何选择产生式；此分析法要求文法必须是 LL(1)文法。

下面以图 4 中的简易文法 G(Z)为例说明 LL(1)分析表的构造过程。

$$Z \longrightarrow a\ A\ b\ (1) \mid A\ c\ A\ (2)$$

$$A \longrightarrow d\ A\ (3) \mid \varepsilon\ (4)$$

图 4 示例文法

现在计算文法 G(Z)的选择集，判断其是否为 LL(1)文法（Z 为开始符号，A 为非终结符，a、b、c、d 为终结符）。各产生式的选择集：select(1)={a}，select(2)={c, d}，select(3)={d}，select(4)={b, d, #}，‘#’代表结束符。由于{d}∩{b}=Φ 又 {a}∩{b, d, #}= Φ，所以文法 G(Z)是 LL(1)文法。文法 G(Z)的 LL(1)分析表如表 1 所示，可理解为：非终结符 Z 遇到字符 a 可使用产生式（1），遇到字符 c 可使用产生式（2）等等，可以用函数“L(栈顶符，当前符）= 产生式序号”来获取相应产生式。

表 1 文法 G(Z)的 LL(1)分析表

| | a | b | c | d | # |
|---|---|---|---|---|---|
| Z | (1) | | (2) | (2) | |
| A | | (4) | (4) | (3) | (4) |

LL(1)分析法的执行将在下文详细描述。

# Token 数据结构

如下图所示，所有符号类均实现接口 Token。

抽象类 TerminalToken 是所有终结符的父类，其中定义的方法 equalsInGrammar，用于在语法分析中判断输入字符是否与终结符匹配。

终结符包括 FunctionToken（函数符号）、KeyToken（关键字）、DelimiterToken（界符）、ConstToken（常量）、VariableToken（变量）。

接口 Valuable 中方法 getDataType()用于返回数据类型，另外还有一系列取值方法，如 getNumberValue、getStringValue 等。抽象类 ValueToken 继承 TerminalToken 并实现了 Valuable 接口。

RuntimeValue 实现 Valuable 接口，RuntimeValue 表示运算过程中的中间结果。

ExecutionToken 表示要在分析过程中执行的动作符号。

ContextOperationToken 表示对表达式中上下文的操作。

NonTerminalToken 表示非终结符。

枚举 ContextOperation、DataType、TokenType 分别定义了上下文操作类型、数据类型、Token 类别。

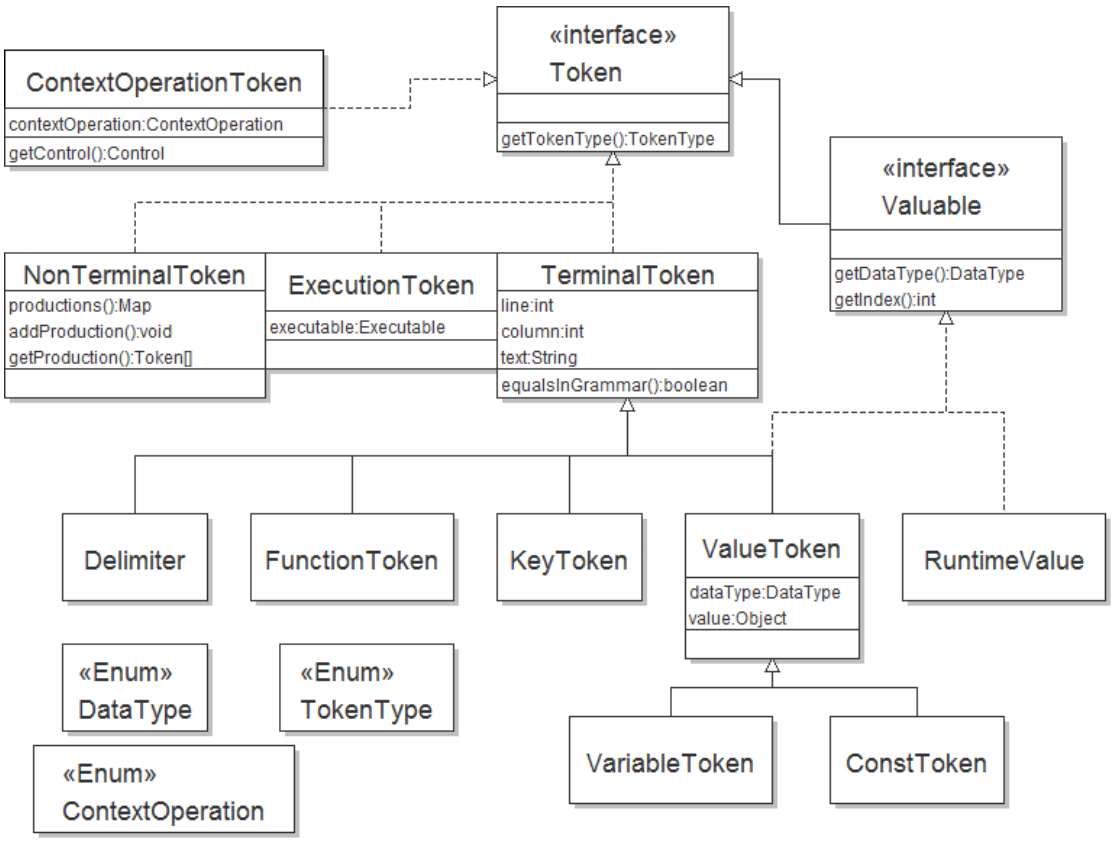



图 5 Token 类图

# 文法设计

neu.sxc.expression.syntax.Grammar 中定义了表达式文法，包括所有符号，所有产生式和产生式的 select 集。下面列出所有产生式（其中 ε 表示空产生式，**黑色**符号为非终结符，蓝色符号为终结符，绿色符号为语义动作符号，橘黄色符号为上下文操作符号）。

start → sentence

start → ifStatement

ifStatement → ifKey leftBracket bolExpression ifConditionCo rightBracket newContextCo block endContextCo elseSection endIFCo endIfKey

block → sentence block

block → ifStatement block

block → ε

elseSection → elseKey elseConditionCo newContextCo block endContextCo

elseSection → ε

sentence → variableToBeAssigned assignMark bolExpression assignExe semicolon

sentence → bolExpression semicolon

bolExpression → bolTerm _bolExpression

_bolExpression → orMark bolTerm orExe _bolExpression

_bolExpression → ε

bolTerm → bolFactor _bolTerm

_bolTerm → andMark bolFactor andExe _bolTerm

_bolTerm → ε

bolFactor → compare

bolFactor → notMark bolFactor notExe

compare → expression _compare

_compare → equalMark expression equalExe

_compare → notEMark expression notEqualExe

_compare → greatMark expression greatExe

_compare → greatEMark expression greatEExe

_compare → lessMark expression lessExe

_compare → lessEMark expression lessEExe

_compare → ε

expression → term _expression

_expression → addMark term addExe _expression

_expression → minusMark term minusExe _expression

_expression → ε

term → factor _term

_term → multiplyMark factor multiplyExe _term

_term → divideMark factor divideExe _term

_term → modMark factor modExe _term

_term → ε

factor → variable

factor → constant

factor → minusMark factor negativeExe

factor → leftBracket bolExpression rightBracket

factor → function leftBracket parameters rightBracket functionExe

parameters → bolExpression _parameters

parameters → ε

_parameters → comma bolExpression _parameters

_parameters → ε

neu.sxc.expression.tokens.NonterminalToken 的方法 addProduction 将选择集和产生式以键值对的形式存入 Map，方法 getProduction 判断输入的 token 是否存在于某一选择集中并返回相应产生式，详见 neu.sxc.expression.syntax.Grammar。

# LL(1)分析法的执行

neu.sxc.expression.syntax. SyntaxAnalyzer 中方法 analysis 首先创建默认上下文（Context）对象并压入 contextStack（对上下文的各种操作将在后面的部分详细描述），然后调用方法 analysisSentence 解析各语句。analysisSentence 实现了 LL(1)分析法，下图为 analysisSentence 的流程图。

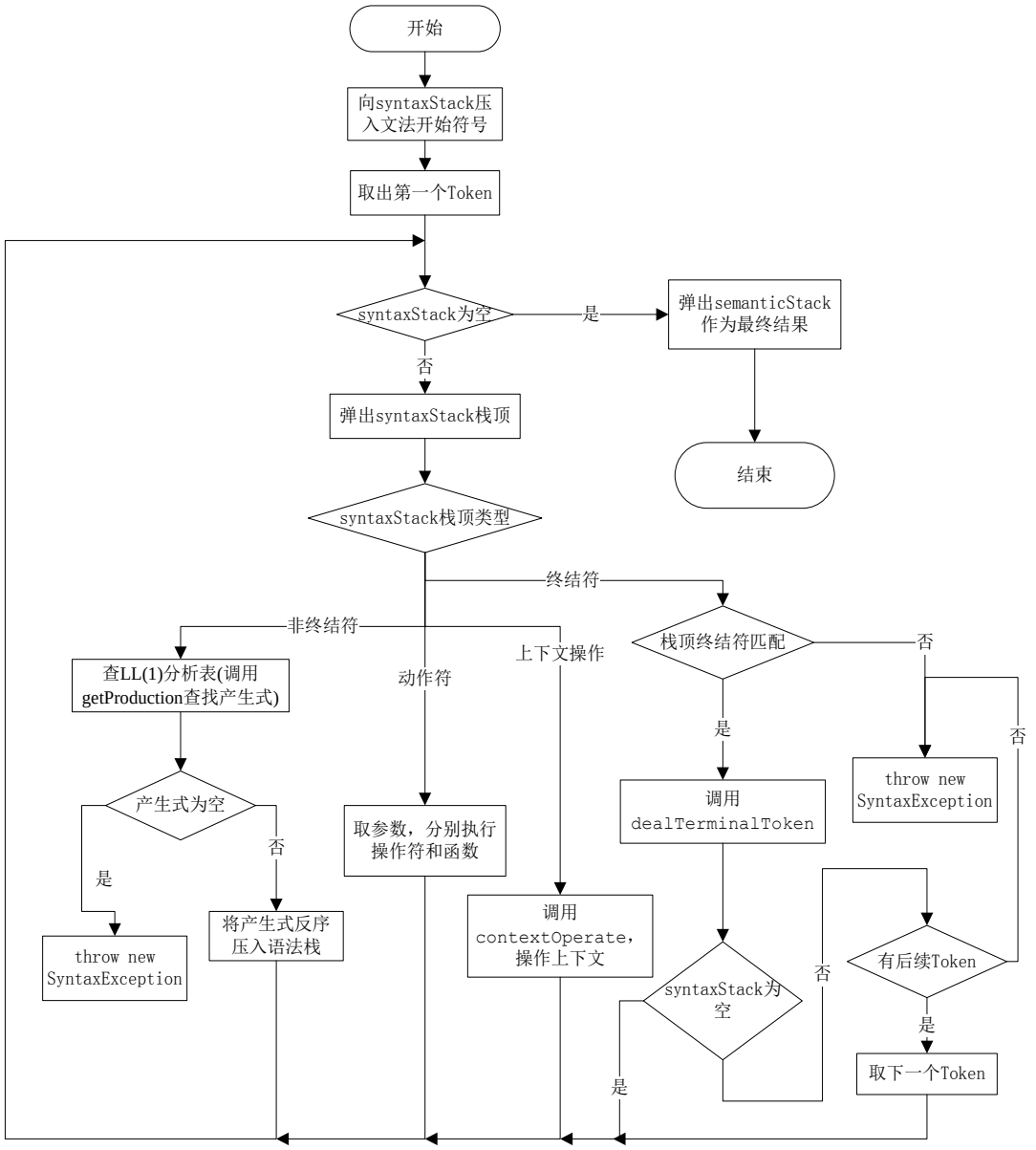



图 6 analysisSentence 的流程图

# 操作符和函数

操作符类 Operator 和函数类 Function 均实现接口 Executable。

抽象类 BinaryOperator 和 UnaryOperator 继承 Operator，分别表示二元操作符和一元操作符。其他具体的操作符类如 AddOperator（加法）、NotOperator（取非）等都是 BinaryOperator 或 UnaryOperator 的子类。

抽象类 Function 是所有函数的父类，用户自定义的函数都需要继承 Function。

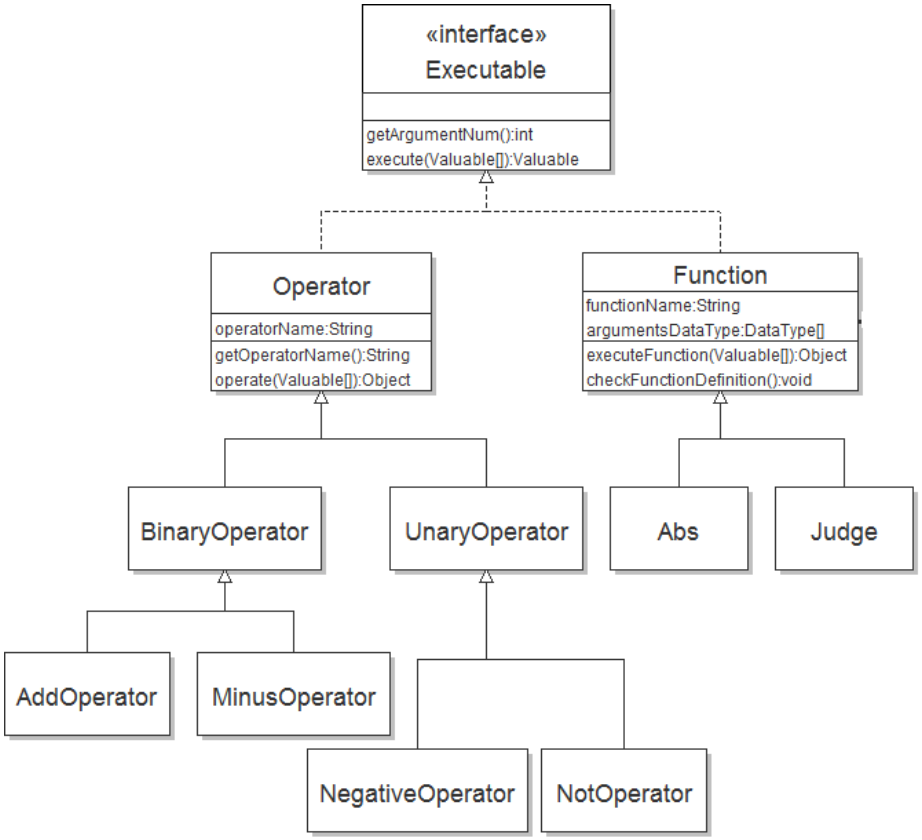



图 7 操作符和函数类图

neu.sxc.expression.syntax.operator. OperatorFactory 是操作符的工厂类，Grammar 类中初始化操作符动作时会从 OperatorFactory 取出操作符对象：

```
private ExecutionToken addExe = TokenBuilder. getBuilder(). executable (OperatorFactory. getOperator("ADD")) .
buildExecution();
```

在 LL(1)分析法的执行过程中，当遇到函数 Token 时，会向 functionTokenStack 和 argumentStartIndexStack 压入函数符号和该函数参数的起始位置。根据参数起始位置可以从语义栈中取得函数参数列表。解析过程中遇到函数执行命令时，分别从这两个栈中取出函数符号和参数起始位置，这样可以解决函数嵌套调用的问题，如（max(1, 2, abs(-3))）。

# 自定义函数

自定义函数需要继承抽象类 Function，有四个方法需要实现：getName()、getArgumentNum()、getArgumentsDataType()、executeFunction()。

getName 返回函数名，函数名不能为空。

getArgumentNum 返回参数个数，当返回值小于 0 时，表示参数个数不限。

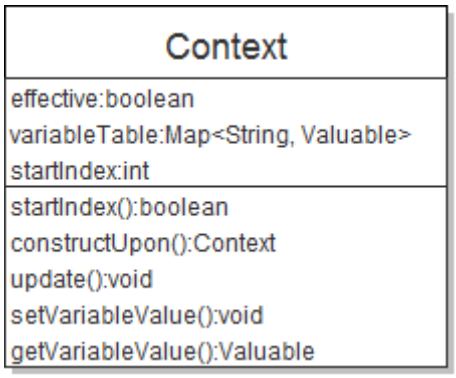
getArgumentsDataType 返回函数参数类型数组。当参数个数不限时，所有参数类型必须相同，本方法须提供一个参数类型。

executeFunction 接收参数数组，实现函数执行逻辑，并返回结果值。

系统函数不需要注册，可以直接使用，类 SystemFunctions 中保存了系统函数实例。使用自定义函数，必须先调用 Expression 的 addFunction 方法，将函数注册到表达式，才能使用。

# 上下文

上下文类 Context 的定义如下：



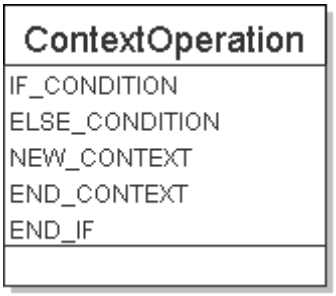

属性 effective 表示 Context 是否有效，如果 Context 所在 if-else 分支的条件为 true，则 effective 设置为 true，否则 effective 设置为 false。

属性 variableTable 存储该上下文的变量和变量值，startIndex 指示 Context 在语义栈 semanticStack 中的开始位置。

方法 constructUpon 用于在当前 Context 基础上创建新的 Context，新创建 Context 的 VariableTable 为当前 Context 中 VariableTable 的拷贝。

方法 update 用于将参数 Context 的 variableTable 更新到当前 Context 中，只更新两个 variableTable 的交集。

枚举类型 ContextOperation 定义了对上下文的各种操作，定义如下：




neu.sxc.expression.syntax.Grammar 中创建了对应 ContextOperation 枚举类型的 ContextOperationToken。neu.sxc.expression.syntax.SyntaxAnalyzer 函数 contextOperate 实现了各个 ContextOperation 类型的执行方式：

IF_CONDITION：在计算出 if 关键字后的条件后执行，向 conditionStack 压入该条件；

ELSE_CONDITION：在遇到 else 关键字后执行，从 conditionStack 中弹出该 else 对应的 if 后的条件，再取反重新压入 conditionStack。

NEW_CONTEXT：新建上下文，首先从 conditionStack 中弹出当前分支的条件，再从上下文栈 contextStack 中弹出栈顶上下文对象，调用其 constructUpon 方法，传入当前分支的条件和开始位置，创建新的上下文并压入 contextStack。

END_CONTEXT：当前上下文结束，从上下文栈 contextStack 中弹出栈顶上下文对象，如果其 effective 标志为 true，将其更新到新的栈顶上下文中，否则不更新，并且从语义栈 semanticStack 中删除该上下文对象开始位置后的所有 Token。

END_IF：表示 if-else 语句结束，从 conditionStack 弹出条件。