# Toureter Interpreter

——**Syntax Analysis Design Document**

**Written by Xiaoran Zhang**

# Produced by Toureter Group

**All Rights Reserved, Copy Reserved**

目录

[Toureter Interpreter 0](#_Toc345285516)

[Produced by Toureter Group 0](#_Toc345285517)

[1 整体概述 3](#_Toc345285518)

[2 实现概念介绍 3](#_Toc345285519)

[2.1 语法分析对象介绍 3](#_Toc345285520)

[2.1.1 CMM语言简介 3](#_Toc345285521)

[2.1.2 语法分析对象简介 4](#_Toc345285522)

[2.3 总体流程图 5](#_Toc345285523)

[2.4 拓展CMM语法规则总结 5](#_Toc345285524)

[2.4.1 程序 6](#_Toc345285525)

[3 实现方法介绍 7](#_Toc345285526)

[分析方法: 递归下降法 7](#_Toc345285527)

[3.1 所需结构 8](#_Toc345285528)

[3.1.1 Tokens类: 管理从词法分析模块得到的所有token 8](#_Toc345285529)

[3.1.2 GrammerTreeNode结构 9](#_Toc345285530)

[3.1.3 模块对外接口 9](#_Toc345285531)

[3.2 具体实现 10](#_Toc345285532)

[3.2.1程序由类或者结构体组成 11](#_Toc345285533)

[} 11](#_Toc345285534)

[3.2.2 类由成员函数或者数据成员构成 11](#_Toc345285535)

[} 11](#_Toc345285536)

[3.2.3 结构体由成员定义组成 12](#_Toc345285537)

[3.2.4 复合语句 12](#_Toc345285538)

[3.2.5 声明语句 13](#_Toc345285539)

[3.2.6 循环语句 14](#_Toc345285540)

[3.2.7 条件语句 14](#_Toc345285541)

[3.2.8 读语句 15](#_Toc345285542)

[3.2.9 写语句 15](#_Toc345285543)

[3.2.10 赋值语句 16](#_Toc345285544)

[3.2.11 表达式(包含逻辑表达式、相等性表达式和算术表达式 16](#_Toc345285545)

[4 实例展示 21](#_Toc345285546)

# 1 整体概述

Toureter语法分析模块的主要功能是将词法分析所得到的token串，通过扫描并且分类生成符合CMM语法的语法树，以便编译器后续的分析。并且在词法分析过程中对token串进行语法扫描，如果用户的语法有误，将提醒用户进行语法更正。

整体情况而言:

1. Toureter设计的拓展CMM语法, 引入面向对象的思想。程序通过class进行组织,同时允许纯数据段(没有函数段)的struct存在。

2. 拓展CMM语法中，Class中以函数和成员定义的组合而成。语法允许复杂的逻辑表达式,函数调用,任意维度的动态数组等。

3. 可以进行错误识别，并给出纠正建议。比如这个位置期望得到哪些类型的token.

# 2 实现概念介绍

## 2.1 语法分析对象介绍

### 2.1.1 CMM语言简介

词法分析的对象是CMM语言的源程序，源程序的叙述如下：

CMM语言为C语言的一个子集：

* + 语言结构：顺序结构（赋值语句、输入、输出）、选择语句（if-else）、循环结构（while）。这些语句结构和C语言的结构一样，允许嵌套。
  + 表达式局限于关系表达式和算术表达式，运算的优先级为：算术运算、关系运算，并服从左结合规则。
  + 算术表达式包括整数和实数上的运算、变量以及“（）”、“\*”、“+”、“-”、“/”，运算符的优先级顺序为：“（）”大于“\*”和“/”大于“+”和“-”。
  + 关系运算符包括：“<”、“==”、“<>”。
  + 一条语句以“；”结束；程序由一条语句或者由“{”和“}”嵌套表达的复合语句。
  + 支持多行注释（使用“/\*”和“ \*/ ”）
  + 支持数组运算，数组的下标必须是正整数，使用“[”和“]”表示数组下标。
  + 变量的使用之前需要先声明，声明的方式和C语言一样。

|  |  |  |
| --- | --- | --- |
| 保留字 | 特殊符号 | 其他 |
| if | + | 十进制的整数与实数 |
| else | - |
| while | \* |
| read | / | 标识符（由数字、字母和下划线组成的串，但必须以字母开头、且不能以下划线结尾的串） |
| write | = |
| int | < |
| real | == |  |
| struct | <> |  |
| class | ( |  |
|  | ) |  |
|  | ; |  |
|  | { |  |
|  | } |  |
|  | /\* |  |
|  | \*/ |  |
|  | [ |  |
|  | ] |  |
|  | /\* |  |
|  | \*/ |  |
|  | && |  |
|  | || |  |
|  | ! |  |

### 2.1.2 语法分析对象简介

经过词法分析，得到的符号类型简介如下：

|  |  |
| --- | --- |
| 符号类型 | 符号 |
| Keyword | if，else，while，int，real，read，write |
| Sign | +，-，\*，/，/\*，\*/，=，==，<，<>，[，]，(，)，{，}，;，,， |
| int | 十进制整数，由数字串组成 |
| real | 十进制实数，由数字和小数点组成 |
| identifier | 由字母开头，由数字、字母和\_组成，并且不以下划线结尾的字符串 |

## 2.3 总体流程图

词法分析的总体流程图如下图所示：

## 2.4 拓展CMM语法规则总结

**注意: 大写字母为非终结符(Nonterminal), 小写为终结符(Terminal),有颜色的字体为保留字或者运算符.语法用BNF范式表示**

### 2.4.1 程序

程序由class和struct组成

Program ->( class|struct)\*

* + 1. **类**

CLASS-> class identifier { (DECLARE\_STMT |FUNCTION)\* } //类定义由ID和声明语句,函数组合而成

* + 1. **结构体**

STRUCT-> struct identifier { (DECLARE\_STMT)\* } //结构体由ID和声明语句组成

* + 1. **函数**

FUNCTION-> TYPE identifier (( DECLARE\_STMT)\*) STMT\_SEQUENCE

* + 1. **语句序列**

STMT\_SEQUENCE -> { STATEMENT}\* // 语句序列由多条语句组成

* + 1. **语句**

STATEMENT-> ASSGIN\_STMT|

WRITE\_STMT |

READ\_STMT |

IF\_STMT |

WHILE\_STMT |

DECLARE\_STMT|

CALL\_STMT

* + 1. **循环语句**

WHILE\_STMT -> while ( EXPR\* ) ({ STMT\_SEQUENCE})|(STATEMENT) //while语句

* + 1. **赋值语句**

ASSGIN\_STMT -> identifier = EXPR

identifier -> identifier (“[“ EXPR“]” )\*

* + 1. **条件语句**

If\_STMT -> if ( EXPR\* ) ({ STMT\_SEQUENCE})|(STMT) (else ({STMT\_SEQUENCE})|(STATEMENT))?

* + 1. **读语句**

READ\_STMT -> read identifier; //读语句

* + 1. **写语句**

WRITE\_STMT -> write EXPR; //写语句

* + 1. **表达式语句**

EXPR->LOGIC\_AND\_EXPR

LOGIC\_AND\_EXPR -> LOGIC\_OR\_EXPR (&& LOGIC\_OR\_EPXR)?

LOGIC\_OR\_EXPR ->LOGIC\_NOT\_EXPR (|| LOGIC\_NOT\_EXPR)?

LOGIC\_NOT\_EXPR ->(!)? LOGIC\_ELEMENT\_EXPR

LOGIC\_ELEMENT\_EXPR-> true|false|EQUALITY\_EXPR

EQUALITY\_EXPR -> ARITHM\_EXPR (<|<=|>|>=|==|<> ARITHM\_EXPR)?

ARTHM\_EXPR ->TERM\_EXPR (+|- TERM\_EXPR)?

TERM\_EXPR ->UNARY\_EXPR (\*|/|% UNARY\_EXPR)?

UNARY\_EXPR ->(-)? ELMENT

ELMENT -> const|CALL\_STMT|identifier|(EXPR)

# 实现方法介绍

## 分析方法: 递归下降法

分析发现,要对以上语法进行分析,至少需要LL(3)分析法。因为首符集的冲突,最多需要预读3个token才能判定用哪条规约。比如变量定义语句和函数定义语句就是一个例子。

递归下降法的思路比较简单，只需要一直往下，每次consume一个终结符或者调用子程序(可能涉及到递归),获得子程序的结果就行了。按照上面设计的文法，一路敲下来基本不需要debug。

需要注意的是良好地设计一个Token的管理类。我弄了一个继承于List<Token>的类Tokens来进行分析。在确定走利用哪条文法进行分析的时候，对于接下来一个需要识别的Token,只需要每次调用Tokens::Consume(token,ExpectedTokenType);如果该token不是ExpectedTokenType(看成一个数组)的元素,那么就说明源代码出错,此时专门由Tokens::Consume()方法抛出异常（ParseException）. 对于catch到异常的代码段,因为知道此时错误token,以及期望的类型。就可以做出很好的错误识别。

## 3.1 所需结构

### 3.1.1 Tokens类: 管理从词法分析模块得到的所有token

internal sealed class Tokens:List<Token>,ICloneable{

public Tokens(IEnumerable<Token> tokens=null){

if(tokens!=null)

this.AddRange(tokens);

}

public Token Consume(TokenKind tokenKind) {

Token first = this.First();

if (tokenKind == first.Kind) {

this.RemoveAt(0);

index = index + 1 >= JDYCompiler.LookAhead ? 0 : index + 1;

this.buffer[index] = first;

return first;

}

throw new ParseException(first, tokenKind);

}

public Token Consume(TokenKind[] tokenKinds) {

Token first = this.First();

foreach (TokenKind tokenKind in tokenKinds)

if (tokenKind == first.Kind)

{

this.RemoveAt(0);

index = index + 1 >= JDYCompiler.LookAhead ? 0 : index + 1;

this.buffer[index] = first;

return first;

}

throw new ParseException(first, tokenKinds);

}

public bool TestNextToken(TokenKind[] tokenKinds, int lookAhead = 0)

{ //nonconsume

Token first = this[lookAhead];

foreach (TokenKind tokenKind in tokenKinds)

if (tokenKind == first.Kind)

{

return true;

}

return false;

}

… …//省略部分内容

}

需要关注的是Tokens类既可以匹配一个token后把该token从Tokens自身中删除,也可以单纯以测试的形式检查是否能匹配到一个token.这种方式可以实现对前面之后几个还没有遇到的token进行预先检查.在首符集冲突的时候,这种形式可以避免做出判断。

在分析的时候，如果确定下一个token必须是什么,否则就错的话（这种情况非常多，比如if后面一个Token必须是’(’,即TokenKind.LPARENT）,就需要使用Consume方法。如果Consume没有匹配到需要的类型,就会抛出一个异常，说明CMM程序源代码错误。

### 3.1.2 GrammerTreeNode结构

internal sealed class GrammerTreeNode {

public List<GrammerTreeNode> Children { get;private set; }

public TreeNodeType Type { get; set; } //is terminal

public NonterminalType NonterminalType { get; set; }

public Token TerminalType { get; set; }

public float RealValue { get;private set; }

public int IntValue { get; private set; }

public int Dimension { get;private set; }

……//已经删除非数据段之外的内容

}

其中:

internal enum TreeNodeType {TERMINAL,NONTERMINAL}

internal enum NonterminalType {

START,/\*表示开始的非终结符\*/

ARRAY,STRUCT,CLASS,/\*代表结构体,类或者函数的非终结符,这类对象做参数时,按照引用传递\*/

STMT\_SEQUENCE,STMT, DECLARE\_STMT, IF\_STMT, WHILE\_STMT, ASSGIN\_STMT, RETURN\_STMT, WRITE\_STMT,READ\_STMT,FUNCTION\_CALL\_STMT, FUNCTION\_DEFINE\_STMT,/\*语句类型的非终结符\*/

EXPR,ANDEXPR,OREXPR,LOGICNOTEXPR, EQUALEXPR, ARITHMEXPR, TERM, UNARY, ELEMENT/\*表达式类型的非终结符\*/

}

树节点可以通过TreeNodeType Type字段指明该节点是一个终结符还是非终结符.如果是非终结符,那么就只能在NonterminalType中枚举。这部分和前面语法解释模块基本一致。可以参考那边。

### 3.1.3 模块对外接口

public static GrammerTreeNode Parse(Tokens tokens) {

sTokens = tokens;

GrammerTreeNode root=start();

return root;

}

## 3.2 具体实现

模块的思路基本上就是（其实后面的子程序处理方法也是同样的思路）： 比如我现在要识别一句话，如果这句话有几种可能（如可以是一个函数调用，也可能是一个赋值语句），那么先通过sTokens(外界传入过来的Tokens)TestNextToken方法,看看接下来的Token是否是’(’.如果是,那么这就是一个函数调用;反之,这就是一个赋值。

假设现在判断是一个赋值语句。那么我们知道接下来第一个token必须是一个**终结符**identifier,所以我可以放心大胆地sToken.Consume(TokenKind.IDENTIFIER),如果成功,那么就会返回一个GrammerTreeNode节点；否则，会抛出一个异常。

假设我们匹配（Consume）完赋值语句的“=”(**终结符**)号了,接下来是=号右边的内容。右边应该是一个表达式，此时是一个**非终结符**，我们直接调用专门处理非终结符的子程序expr(),如果源代码在=号右边都是正确的,那么expr()就应该返回一个识别好的expr树的根节点。

而expr()中具体识别过程是和刚才一样的过程。

为了更加清晰地展示语法分析过程中整个递归下降的过程，下面给出整个语法分析类的函数签名:

public static GrammerTreeNode Parse(Tokens tokens)

private static GrammerTreeNode start()

private static GrammerTreeNode classDeclareStmt()

private static GrammerTreeNode structDeclareStmt()

private static List<GrammerTreeNode> declareStmt()

private static GrammerTreeNode functionStmt()

private static GrammerTreeNode stmtSequence()

private static GrammerTreeNode stmt()

private static GrammerTreeNode assginStmt()

private static GrammerTreeNode returnStmt()

private static GrammerTreeNode writeStmt()

private static GrammerTreeNode readStmt()

private static GrammerTreeNode callStmt()

private static GrammerTreeNode whileStmt()

private static GrammerTreeNode ifStmt()

private static GrammerTreeNode expression()

private static GrammerTreeNode logicalOrExpression()

private static GrammerTreeNode logicalAndExpression()

private static GrammerTreeNode logicalNotExpression()

private static GrammerTreeNode logicalElement()

private static GrammerTreeNode equalityExpression()

private static GrammerTreeNode arithmaticExpression()

private static GrammerTreeNode termExpression()

private static GrammerTreeNode unaryExpression()

private static GrammerTreeNode element()

### 3.2.1程序由类或者结构体组成

private static GrammerTreeNode start(){

GrammerTreeNode root = new GrammerTreeNode(){Type=TreeNodeType.NONTERMINAL,NonterminalType=NonterminalType.START};

while(!sTokens.TestNextToken(TokenKind.EOF)){

if(sTokens.TestNextToken(TokenKind.STRUCT)){

root.Add(structDeclareStmt());

}

else if (sTokens.TestNextToken(TokenKind.LPARENT, 2))

{

root.Children.Add(functionStmt());

}

else

{

root.Children.AddRange(declareStmt());

}

}

return root;

}

### 3.2.2 类由成员函数或者数据成员构成

private static GrammerTreeNode classDeclareStmt()

{

GrammerTreeNode class\_stmt = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.CLASS };

sTokens.Consume(TokenKind.CLASS);

class\_stmt.TerminalType = (sTokens.Consume(TokenKind.IDENTIFIER));

sTokens.Consume(TokenKind.OBRACE);

while (!sTokens.TestNextToken(TokenKind.CBRACE))

{

if (sTokens.TestNextToken(TokenKind.LPARENT, 2))

class\_stmt.Children.Add(functionStmt());

else

class\_stmt.Children.AddRange(declareStmt());

}

sTokens.Consume(TokenKind.CBRACE);

return class\_stmt;

}

### 3.2.3 结构体由成员定义组成

private static GrammerTreeNode structDeclareStmt()

{

GrammerTreeNode struct\_stmt = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.STRUCT };

sTokens.Consume(TokenKind.STRUCT);

struct\_stmt.TerminalType = (sTokens.Consume(TokenKind.IDENTIFIER));

sTokens.Consume(TokenKind.OBRACE);

while (!sTokens.TestNextToken(TokenKind.CBRACE))

struct\_stmt.Add(declareStmt());

sTokens.Consume(TokenKind.CBRACE);

return struct\_stmt;

}

### 3.2.4 复合语句

private static GrammerTreeNode stmtSequence() {

GrammerTreeNode stmt\_sequence = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.STMT\_SEQUENCE };

while (!sTokens.TestNextToken(TokenKind.CBRACE))

{

stmt\_sequence.Add(stmt());

}

return stmt\_sequence;

}

private static GrammerTreeNode stmt() {

GrammerTreeNode stmt = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.STMT };

if (sTokens.TestNextToken(nonVoidTypeToken))

stmt.Add(declareStmt());

else if (sTokens.TestNextToken(TokenKind.IF))

stmt.Add(ifStmt());

else if (sTokens.TestNextToken(TokenKind.WHILE))

stmt.Add(whileStmt());

else if (sTokens.TestNextToken(TokenKind.READ))

stmt.Add(readStmt());

else if (sTokens.TestNextToken(TokenKind.WRITE))

stmt.Add(writeStmt());

else if (sTokens.TestNextToken(TokenKind.RETURN))

stmt.Add(returnStmt());

else if (sTokens.TestNextToken(TokenKind.SEMI))//空白语句直接忽略.

sTokens.Consume(TokenKind.SEMI);

else if (sTokens.TestNextToken(TokenKind.IDENTIFIER)) //函数调用或者是赋值语句

{

if (sTokens.TestNextToken(TokenKind.LPARENT, 1)) //lookahead

stmt.Add(callStmt());

else if (sTokens.TestNextToken(TokenKind.IDENTIFIER, 1))

stmt.Add(declareStmt());

else

stmt.Add(assginStmt());

}

return stmt;

}

### 3.2.5 声明语句

private static List<GrammerTreeNode> declareStmt()

{

List<GrammerTreeNode> declares = new List<GrammerTreeNode>();

Token type= sTokens.Consume(typeToken);

while (true)

{

GrammerTreeNode declare = new GrammerTreeNode() {Type=TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.DECLARE\_STMT };

declare.Add(type);

declare.Children.First().NonterminalType = NonterminalType.STRUCT;

GrammerTreeNode element = declare.Add(sTokens.Consume(TokenKind.IDENTIFIER));

if (sTokens.TestNextToken(TokenKind.ASSIGN))

{

sTokens.Consume(TokenKind.ASSIGN);

declare.Add(expression());

}

else if (sTokens.TestNextToken(TokenKind.OBRAKET))//n dimentional array

{

do

{

sTokens.Consume(TokenKind.OBRAKET);

element.DimensionAdd();

sTokens.Consume(TokenKind.CBRAKET);

} while (sTokens.TestNextToken(TokenKind.OBRAKET));

}

declares.Add(declare);

if (sTokens.TestNextToken(TokenKind.COMMA))

sTokens.Consume(TokenKind.COMMA);

else

break;

}

sTokens.Consume(TokenKind.SEMI);

return declares;

}

### 3.2.6 循环语句

private static GrammerTreeNode writeStmt(){

GrammerTreeNode write\_stmt = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.WRITE\_STMT };

sTokens.Consume(TokenKind.WRITE);

write\_stmt.Add(expression());

sTokens.Consume(TokenKind.SEMI);

return write\_stmt;

}

### 3.2.7 条件语句

private static GrammerTreeNode ifStmt() {

GrammerTreeNode if\_stmt = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.IF\_STMT };

sTokens.Consume(TokenKind.IF);

sTokens.Consume(TokenKind.LPARENT);

if\_stmt.Add(expression());

sTokens.Consume(TokenKind.RPARENT);

if (sTokens.TestNextToken(TokenKind.OBRACE))

{

sTokens.Consume(TokenKind.OBRACE);

if\_stmt.Add(stmtSequence());

sTokens.Consume(TokenKind.CBRACE);

}

else {

if\_stmt.Add(stmt());

}

if (sTokens.TestNextToken(TokenKind.ELSE)) {

sTokens.Consume(TokenKind.ELSE);

if (sTokens.TestNextToken(TokenKind.OBRACE))

{

sTokens.Consume(TokenKind.OBRACE);

if\_stmt.Add(stmtSequence());

sTokens.Consume(TokenKind.CBRACE);

}

else

{

if\_stmt.Add(stmt());

}

}

return if\_stmt;

}

### 3.2.8 读语句

private static GrammerTreeNode readStmt() {

GrammerTreeNode read\_stmt = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.READ\_STMT };

sTokens.Consume(TokenKind.READ);

GrammerTreeNode element = read\_stmt.Add(sTokens.Consume(TokenKind.IDENTIFIER));

while (sTokens.TestNextToken(TokenKind.OBRAKET)) {

sTokens.Consume(TokenKind.OBRAKET);

element.Add(arithmaticExpression());

element.DimensionAdd();

sTokens.Consume(TokenKind.CBRAKET);

}

sTokens.Consume(TokenKind.SEMI);

return read\_stmt;

}

### 

### 3.2.9 写语句

private static GrammerTreeNode writeStmt(){

GrammerTreeNode write\_stmt = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.WRITE\_STMT };

sTokens.Consume(TokenKind.WRITE);

write\_stmt.Add(expression());

sTokens.Consume(TokenKind.SEMI);

return write\_stmt;

}

### 

### 3.2.10 赋值语句

private static GrammerTreeNode assginStmt() {

GrammerTreeNode assgin\_stmt = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.ASSGIN\_STMT };

assgin\_stmt.Add(assginableElement());

sTokens.Consume(TokenKind.ASSIGN); //consume "="

assgin\_stmt.Add(expression()); //right part of "="(expr)

sTokens.Consume(TokenKind.SEMI);

return assgin\_stmt;

}

### 3.2.11 表达式(包含逻辑表达式、相等性表达式和算术表达式)

private static GrammerTreeNode expression() {

return logicalOrExpression();

}

/// <summary>

///

/// </summary>

/// <returns></returns>

private static GrammerTreeNode logicalOrExpression() {

GrammerTreeNode logic\_or\_expr = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.OREXPR };

logic\_or\_expr.Add(logicalAndExpression());

if (sTokens.TestNextToken(TokenKind.OR))

{

logic\_or\_expr.Add(sTokens.Consume(TokenKind.OR));

logic\_or\_expr.Add(logicalOrExpression());

}

else {

logic\_or\_expr = logic\_or\_expr.Children.First();//no operator, trim the node

}

return logic\_or\_expr;

}

/// <summary>

/// tree node sturct

/// --------------------------------

/// logic\_not\_expr

/// logic\_and\_expr and(&&)

/// logic\_not\_expr

///

/// however no operator(&&) exist, it will be trimed like:

/// logic\_not\_expr

/// </summary>

/// <returns></returns>

private static GrammerTreeNode logicalAndExpression() {

GrammerTreeNode logic\_and\_expr = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.ANDEXPR };

logic\_and\_expr.Add(logicalNotExpression());

if (sTokens.TestNextToken(TokenKind.AND))

{

logic\_and\_expr.Add(sTokens.Consume(TokenKind.AND)); //can be simplified

logic\_and\_expr.Add(logicalAndExpression());

}

else {

logic\_and\_expr = logic\_and\_expr.Children.First();//no operator, trim the node

}

return logic\_and\_expr;

}

/// <summary>

/// not(optional)

/// logic\_not\_expr logic\_element

///

/// however if trimed, it look like:

///

/// logic\_element

///

/// </summary>

/// <returns></returns>

private static GrammerTreeNode logicalNotExpression() {

GrammerTreeNode logic\_not\_expr = new GrammerTreeNode() { Type=TreeNodeType.NONTERMINAL,NonterminalType=NonterminalType.LOGICNOTEXPR};

if (sTokens.TestNextToken(TokenKind.NOT))

{

logic\_not\_expr.Add(sTokens.Consume(TokenKind.NOT));

logic\_not\_expr.Add(logicalElement());

}

else {

logic\_not\_expr = logicalElement();

}

return logic\_not\_expr;

}

/// <summary>

/// logic\_element true|false

///

/// or trimed like:

/// equality\_expr

/// </summary>

/// <returns></returns>

private static GrammerTreeNode logicalElement() {

GrammerTreeNode logic\_elemt = null;

if (sTokens.TestNextToken(boolElemtToken))

{

logic\_elemt = new GrammerTreeNode() { Type = TreeNodeType.TERMINAL, TerminalType = sTokens.Consume(boolElemtToken) };

}

else

{

logic\_elemt =equalityExpression();

}

return logic\_elemt;

}

private static GrammerTreeNode equalityExpression() {

GrammerTreeNode equal\_expr= new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.EQUALEXPR };

equal\_expr.Add(arithmaticExpression());

if (sTokens.TestNextToken(equalityToken))

{

equal\_expr.Add(sTokens.Consume(equalityToken));

equal\_expr.Add(arithmaticExpression());

}

else {

equal\_expr = equal\_expr.Children.First();

}

return equal\_expr;

}

private static GrammerTreeNode arithmaticExpression() {

GrammerTreeNode arithm\_expr = new GrammerTreeNode() { Type=TreeNodeType.NONTERMINAL,NonterminalType=NonterminalType.ARITHMEXPR};

arithm\_expr.Add(termExpression());

if (sTokens.TestNextToken(plusOrMinusToken))

{

arithm\_expr.Add(sTokens.Consume(plusOrMinusToken));

arithm\_expr.Add(arithmaticExpression());

}

else

{

arithm\_expr = arithm\_expr.Children.First();

}

return arithm\_expr;

}

private static GrammerTreeNode termExpression() {

GrammerTreeNode term\_expr = new GrammerTreeNode() { Type=TreeNodeType.NONTERMINAL,NonterminalType=NonterminalType.TERM};

term\_expr.Add(unaryExpression());

if (sTokens.TestNextToken(multiOrDividToken))

{

term\_expr.Add(sTokens.Consume(multiOrDividToken));

term\_expr.Add(termExpression());

}

else {

term\_expr = term\_expr.Children.First();

}

return term\_expr;

}

/// <summary>

///

/// mius(optional)

/// unary\_expr element

///

/// </summary>

/// <returns></returns>

private static GrammerTreeNode unaryExpression() {

GrammerTreeNode unary\_expr = new GrammerTreeNode() { Type=TreeNodeType.NONTERMINAL,NonterminalType=NonterminalType.UNARY};

if (sTokens.TestNextToken(TokenKind.MINUS))

{

unary\_expr.Add(sTokens.Consume(TokenKind.MINUS));

unary\_expr.Add(element());

}

else {

unary\_expr = element();

}

return unary\_expr;

}

private static GrammerTreeNode element() {

GrammerTreeNode elem = null;

if (sTokens.TestNextToken(constElemToken)) {

elem = constElement();

}

else if (sTokens.TestNextToken(TokenKind.IDENTIFIER))

{

if (sTokens.TestNextToken(TokenKind.LPARENT, 1))

{//function call

elem = new GrammerTreeNode() { Type = TreeNodeType.NONTERMINAL, NonterminalType = NonterminalType.FUNCTION\_CALL\_STMT };

elem.Add(sTokens.Consume(TokenKind.IDENTIFIER));

sTokens.Consume(TokenKind.LPARENT);

while (!sTokens.TestNextToken(TokenKind.RPARENT))

{//param list

elem.Add(expression());

if (sTokens.TestNextToken(TokenKind.COMMA))

sTokens.Consume(TokenKind.COMMA);

}

sTokens.Consume(TokenKind.RPARENT);

}

else

{

elem = assginableElement();

}

}

else if (sTokens.TestNextToken(TokenKind.LPARENT))

{ //括号表达式...

sTokens.Consume(TokenKind.LPARENT);

elem=expression();

sTokens.Consume(TokenKind.RPARENT);

}

else { //trigger a exception

sTokens.Consume(constElemToken.Union(new TokenKind[] {TokenKind.IDENTIFIER,TokenKind.LPARENT }).ToArray());

}

return elem;

}

private static GrammerTreeNode constElement() {

return new GrammerTreeNode(sTokens.Consume(constElemToken));

}

private static GrammerTreeNode assginableElement()

{

GrammerTreeNode elem = null;

//ID element

elem = new GrammerTreeNode() {

Type = TreeNodeType.TERMINAL,

TerminalType = sTokens.Consume(TokenKind.IDENTIFIER)

};

GrammerTreeNode tmp = elem;

while (true)

{

if (sTokens.TestNextToken(TokenKind.DOT))

{

tmp.Add(sTokens.Consume(TokenKind.DOT));

tmp.Add(sTokens.Consume(TokenKind.IDENTIFIER));

// if (sTokens.TestNextToken(TokenKind.OBRAKET))

tmp = tmp.Children.Last();

}

else if (sTokens.TestNextToken(TokenKind.OBRAKET))

{

//array element

while (sTokens.TestNextToken(TokenKind.OBRAKET))

{

sTokens.Consume(TokenKind.OBRAKET);

tmp.Add(arithmaticExpression());

tmp.DimensionAdd();

sTokens.Consume(TokenKind.CBRAKET);

}

}

else break;

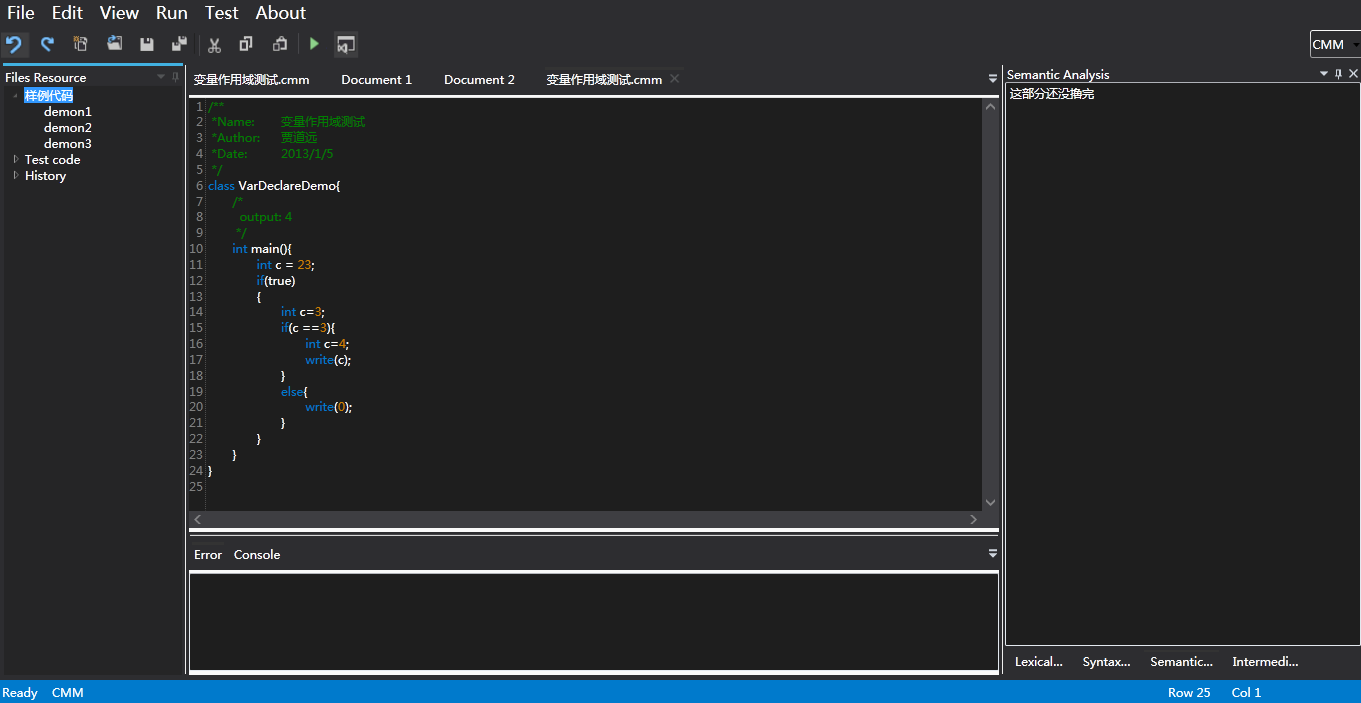
}

return elem;

}

# 4 实例展示

输入以下内容：



得到如下结果，完成词法分析过程：

