**武汉大学计算机学院**

**本科生实验报告**

**解释器构造实践**

专 业 名 称 ：软件工程

课 程 名 称 ：解释器构造实践

指 导 教 师 ：刘纪平 职称 讲师

学 生 学 号 ：2017302580213

学 生 姓 名 ：彭英杰

二○一九年十月

**郑 重 声 明**

本人呈交的实验报告，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。

本人签名： 彭英杰 日期：

摘 要

本实验的实验目的是构造一个自定义CMM语言文法的完整解释器

实验内容主要包括：词法分析模块，语法分析模块以及语义分析模块、以及 使用JavaCC自动生成关于CMM词法、语法规则的词法、语法分析程序。

**关键词：**词法分析；语法分析；语义分析；JavaCC

**目 录**

**1** **实验目的**

1.1 实验目的 ………………………………………………………………………………6

**2** **词法分析**

2.1 概述 ………………………………………………………………………6

2.2实验原理与方案 ………………………………………………………… 6

2.2.1 符号设计 ………………………………………………………… 6

2.2.2 种类码表 ………………………………………………………… 7

2.3 程序结构 ……………………………………………………………………8

2.3.1 输入输出设计 ………………………………………………………8

2.3.2 符号种类码表实现…………………………………………………9

2.3.3 模块结构和具体算法………………………………………………10

2.3.4 错误处理……………………………………………………………11

2.4 运行与测试…………………………………………………………………12

2.4.1 测试用例……………………………………………………………12

2.4.2 测试结果……………………………………………………………13

**3** **语法分析**

3.1 概述 ………………………………………………………………………14

3.2实验原理与方案 ……………………………………………………… 14

3.2.1 实验原理………………………………………………………… 14

3.2.2 CMM文法定义…………………………………………………… 14

3.2.3 输入输出设计……………………………………………… 17

3.2.4 模块结构和具体算法……………………………………………… 17

3.2.5 错误处理…………………………………………………………… 25

3.3 运行与测试…………………………………………………………………………26

3.3.1 测试用例…………………………………………………………………26

3.3.2 测试结果…………………………………………………………………27

**4** **语义分析**

4.1 概述 ……………………………………………………………………… 29

4.2 实验原理与方案 ………………………………………………………… 29

4.2.1 实验原理…………………………………………………………… 30

4.2.2 语义规则…………………………………………………………… 30

4.3.1 模块结构和具体算法…………………………………………… 31

4.3.2 错误处理…………………………………………………………… 40

4.3 运行与测试………………………………………………………………… 40

4.3.1 测试用例……………………………………………………………………… 41

4.3.2 测试结果……………………………………………………………………… 42

**5** **JavaCC-CMM**

5.1 概述 ……………………………………………………………………… 43

5.2 实验原理与方案 ………………………………………………………… 43

5.2.1 JavaCC……………………………………………………………… 43

5.2.2 具体实现………………………………………………………… 43

5.3 运行与测试………………………………………………………………… 46

5.3.1 测试用例……………………………………………………………………… 46

5.3.2 测试结果……………………………………………………………………… 46

**结论** …………………………………………………………………………………… 47

**1 实验目的**

* 1. **实验目的**

本实验需要设计一个简化版的CMM语言文法并且实现针对该语言的解释器，要求能够解释执行出正确结果，实验步骤分为三个部分，词法分析，语法分析以及语义分析。除此之外，要求利用JavaCC自动生成针对CMM的词法分析器和语法分析器。本实验使用java高级语言与vscode环境实现

**2 词法分析**

**2.1** **概述**

词法分析是本次解释器构造实验的第一步，也是大致确定自定义的CMM语言的作用域与研究对象的第一步，该模块以源程序语言为输入，以识别后形成的词法单元token数组以及识别中发现的不合法的错误符号信息为输出。也为后一步骤，构建语法分析模块提供输入。

**2.2 实验原理与方案**

**2．2．1** **符号定义**

自定义的CMM语言主要在词法分析时主要识别以下几个作用域或词法单元

1.关键字

<关键字>-> int | bool | float | break |continue |while | for | if | else | return | scan | print | error

2.逻辑运算符

<运算符>-> < | > | <= | > | == | <>| && | ||

3.算数运算符

<运算符>->| + | - | \* | / | += | -= | /= | \*= | = | ++ | --

4.界符

<界符>-> ( | ) | { | } | [ | ]

5.注释

<注释>->//|/\*...\*/

6. 其他符号

<其他符号>-> ,| ; |.

7.标识符

letter -> A | B | ... | Z | a | b | ... | z | \_

digit -> 0 | 1 | ... | 9

identifier -> letter(letter\_|digit)\*

8.常数

<常数>-> floatVal | intVal | true | false

digit -> 0 | 1 | ... | 9

<floatVal>-> digit\*(.|digit)digit+

<intVal>->digit\*

**2．2．2** **种类码表**

此处给出了程序中使用的种类码表：

|  |  |  |  |
| --- | --- | --- | --- |
| **单词符号** | **种类码** | **单词符号** | **种类码** |
| **If** | 1 | **--** | 22 |
| **else** | 2 | **<** | 23 |
| **while** | 3 | **>** | 24 |
| **for** | 4 | **<=** | 25 |
| **int** | 5 | **>=** | 26 |
| **bool** | 6 | **==** | 27 |
| **float** | 7 | **<>** | 28 |
| **return** | 8 | **{** | 29 |
| **continue** | 9 | **}** | 30 |
| **break** | 10 | **[** | 31 |
| **scan** | 11 | **]** | 32 |
| **print** | 12 | **(** | 33 |
| **+** | 13 | **)** | 34 |
| **-** | 14 | **,** | 35 |
| **\*** | 15 | **;** | 36 |
| **/** | 16 | **error** | 37 |
| **+=** | 17 | **floatVal** | 38 |
| **-=** | 18 | **intVal** | 39 |
| **/=** | 19 | **true** | 40 |
| **\*=** | 20 | **false** | 41 |
| **++** | 21 | **identifier** | 42 |

**2.3**  **程序设计**

**2．3．1** **输入输出设计**

**输入设计**

为与真正成熟的解释器靠近，本实验采取类似的源程序输入方式，用户编写符合词法规范要求的源程序作为文本输入，词法分析器打开一个文件流，从对应的文件路径获得源程序文本，使用filereader对象将文件读入足够大的字符数组中。

**输出设计**

导入源程序代码文本之后，运行解释器，入口函数首先调用词法分析模块，将扫描文件得到的字符数组作为参数，词法分析模块经过一系列的匹配控制操作，得到定义的Token类型对象，并依次加入Tokenlist对象数组中，词法分析结束，循环遍历此数组，将其中token对象信息按顺序打印至控制台输出。

**2．3．2** **符号种类码实现**

为识别符合后续语法语义操作规范的词法单元，每个词法单元识别后应包含下列信息以方便进行错误处理以及后续语义语法操作。

     public int type;        *//词法单元种类*

    public String strval;   *//词法单元内容*

     public int line;        *//词法单元所在行*

     public String errorInfo; *//错误词法单元错误信息*

词法单元种类即规范的词法单元种类，定义一个final修饰的包含词法单元种类信息的类，根据之前定义的符号种类码表，在其中进行相应的类别信息定义

public static final int IF = 1;

public static final int ELSE = 2;

     public static final int WHILE = 3;

     public static final int FOR = 4;

     public static final int INT = 5;

**2．3．3** **模块结构和具体算法**

本次词法分析实现主要包含以下几个结构：

**WordAnalyze**类（词法分析核心算法具体实现的类）、**Token**类（定义词法单元的类）、**TokenType**类（定义词法单元类型的类）

**Token**类：成员变量为3.3.2提到的，构造函数分为普通类型构造函数（参数为type，strval，line）和错误类型构造函数（参数为type，strval，line，errorinfo），具体方法有tostring（）方法（主要作用为以'<'+""+type+"    "+strval+"    "+line+'>'形式打印词法单元，若为错误类型，则打印errorinfo错误信息）

**TokenType**类：成员变量为3.3.2提及的，数目与具体定义与符号码表一致。

**WordAnalyze**类：

在本词法分析模块中，对源程序转化得到的字符数组进行循环遍历扫描，使用if...else嵌套下的switch...case控制结构进行词法单元的扫描与匹配。根据不同词法单元的具体特性特性，其中对关键字、标识符、常量与界符、运算符等符号不同的词法单元进行分开识别。词法单元用Tokenlist列表存储。

为识别标识符、关键字、IntVal、FloatVal，建立匹配关键字、识别字母数字的四个工具函数。

isLetter(char curcharacter)、isDigit(char curcharacter) 、isKeyword(String word, int line)。返回类型均为boolean类型

主体识别部分为**wordAnalyze**(char[] words)函数，函数以源程序字符数组为参数，初始化一个变量用来标识当前字符所在行数。进入主体部分，循环遍历字符数组，每次初始化一个空字符串words用来存储词法单元的内容信息。

进入if...else条件语句。

第一个if语句用来识别标识符和关键字，因为两者结构类似，区别仅为关键字优先级更高。判断当前字符是否为字母，循环识别接下来的字符判断是否是字母或字符（根据情况也可加入下划线），循环中控制字符数组是否越界（越界则退出循环），将扫描过得字符加入字符串words中，循环条件不满足则退出循环，调用isKeyword函数，先将当前字符串与关键字进行匹配，匹配成功创建对应的token类型变量并加入tokenlist数组中。

第二个elseif语句处理常量（包括浮点数与整数）的识别，判断当前字符是否为数字，定义局部boolean变量标识小数点，和int类型变量flag标识小数点数目，以当前字符是数字或者是小数点为条件，循环遍历字符数组，将当前字符加入字符串，当当前字符为小数点但不是第一个小数点时判断为非法常数，退出循环，将扫描过的字符加入字符串words中，创建对应的token类型变量（token.type为error，且设置对应的errorinfo）并加入tokenlist数组中，回退至上一字符。当当前字符为其他字符时，退出循环，将扫描过的字符加入字符串words中，创建对应的token类型变量（token.type为intVal或floatVal（根据flag设置））并加入tokenlist数组中，回退至上一字符。

第三个else语句块使用switch...case语句块识别各种运算符、界符等其他字符，根据当前字符数组的值，进入对应的符号的case语句块，进行不同操作。case语句块有运算符（注释识别放在/运算符的识别中）、界符、空格符、回车符、换行符、分号、逗号等不同情况的的对应操作。我将以/、（、‘\n’、space为例进行详细介绍。

Case ‘/’，读取下一个字符，（未越界）如果为=，识别为/=，创建对应的token类型变量并加入tokenlist数组中；如果为/，直接循环遍历字符数组，以当前字符是否为换行符为条件，是则退出循环（循环遍历期间不进行任何操作，仅仅读取下一个字符）；如果为\*，直接循环遍历字符数组，以当前字符与下一字符是否为\*和/为条件，是则退出循环（循环遍历期间不进行任何操作，仅仅读取下一个字符）。不是上述几种情况则识别为/，将当前字符加入字符串words中，回退至上一字符，创建对应的token类型变量并加入tokenlist数组中

Case ‘（’，将当前字符加入字符串words中，创建对应的token类型变量并加入tokenlist数组中

Case ‘\n’ （程序中以对应的ASCII值出现），将记录当前字符行数的line局部变量+1。

Case space（程序中以对应的ASCII值出现），不做任何操作，除了每个case都执行的break语句。

Default，将当前字符加入字符串words中，创建对应的token类型变量（token.type为error，且设置对应的errorinfo为“未定义符号”）并加入tokenlist数组中。

当字符数组扫描完，使用printToken（）函数打印tokenlist中信息，具体实现为循环遍历tokenlist数组，调用每个token的tostring方法。

**2．3．4** **错误处理**

词法分析模块中我专门加入了一种词法单元，类型为error，且初始化该类型对象时加入对应的errorinfo信息。当输入源程序有不符合自定义的词法规范时，扫描完后，将打印出所有合法的词法单元以及所有的errorinfo错误信息、错误字符以及它们所在源程序行数

**2.4**  **运行与测试**

**2．4．1** **测试用例**

为了测试CMM语言词法分析分析的正确性，测试用例不仅包含正规的词法单元符号，还将包括部分未定义符号与错误的常量。测试用例中覆盖的检测点如下所示：

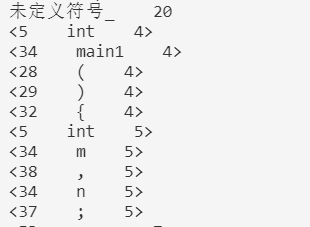
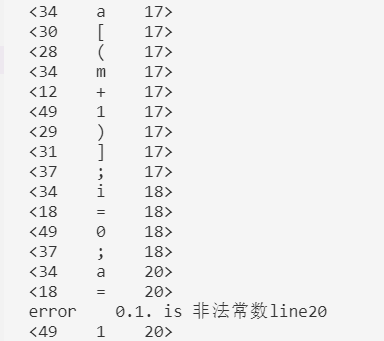
1. 未定义符号例如单独下划线
2. /\* \*/注释与//注释，注释中可出现任意字母以及数字和符号
3. 错误的常量，主要为错误的小数
4. 以及一些正常词法单元

如下是测试用例的代码：



**2．4．2** **测试结果**

如下是测试用例的运行结果。\_a 将\_识别为未定义符号，0.1.1识别为非法常数，另外/\*\*/块注释与//行注释全部省略。由于词法单元数较多，全部截图显得较冗余，故只贴上关键部分



**2．4．2** **实验中遇到问题**

未考虑换行符、回车符、空格符

考虑小数时，多个.情况的判别在众多词法单元的识别中较复杂

**3 语法分析**

**3.1** **概述**

经过词法分析模块分析之后，我们将获得输入源程序的词法单元数组形式，但此时这些词法单元并不具备意义，只是源程序的另一形式的表达，简单来说，是为方便进行语法检查构造的中间形式。因此，语法分析模块将是本次实验的核心，语义分析仅仅是对语法树的扫描以及操作。根据词法单元语法匹配的顺序，可分为自上而下语法分析与自下而上语法分析，自下而上语法分析使用分析表与总控程序进行匹配识别，重点是构造合适的分析表。而本实验将采用自上而下的递归子程序法来构造语法分析模块。

**3.2 实验原理与方案**

**3．2．1** **实验原理**

语法分析模块因不同的语言文法而异，因此语法分析的首要步骤是构建自定义的CMM文法，相对于完整的C语言文法，本次实验构造的CMM文法相对简单，算是C文法的子集的子集。构造好文法后，根据语法分析匹配识别的顺序，分为自上而下与自下而上语法分析法，本实验将采用自上而下的递归子程序法来构造语法分析模块。实验方案将按文法定义与输入输出设计、算法设计与实现、错误处理四部分顺序进行。

**3．2．2** **CMM文法定义**

为了减少左递归，回溯以及向前看的词法单元数量以及隐含的运算符优先级、结合性等诸多限制，在确定好文法后，还需要进行一定程度的改进，通过增加非终结符的数量，加深语法树的深度等方法，对文法进行了如下改进。$表示可为空。

以下是CMM的文法定义：

<程序> -> <函数定义或声明> <程序>

<程序> -> $

<函数定义或声明> -> <类型> <变量> ( <参数声明> ) <函数定义或声明闭包>

<函数定义或声明闭包> -> <复合函数块>

<函数定义或声明闭包> -> ;

<类型> -> int

<类型> -> bool

<类型> -> float

<变量> -> ID <数组下标>

<数组下标> -> [ <因式> ]

<数组下标> -> $

<因式> -> ( <算数表达式> )

<因式> -> -( <算数表达式> )

<因式> -> <变量>

<因式> -> -<变量>

<因式> -> -<常数>

<因式> -> <常数>

<因式> -> $

<算数表达式> -> <因子> <项>

<因子> -> <因式1> <因式递归>

<因式递归> -> \* <因式1> <因式递归>

<因式递归> -> / <因式1> <因式递归>

<因式递归> -> $

<因式1>->++<因式>

<因式1>->--<因式>

<因式1>-><因式>++

<因式1>-><因式>--

<因式1>-><因式>

<项> -> + <因子> <项>

<项> -> - <因子> <项>

<项> -> $

<参数声明> -> <声明> <声明闭包>

<参数声明> -> $

<声明> -> <类型> <变量> <赋初值>

<赋初值> -> = <右值>

<赋初值> -> $

<右值> -> <算数表达式>

<右值> -> <逻辑表达式>

<多个数据> -> <常数> <常数闭包>

<常数闭包> -> , <常数> <常数闭包>

<常数闭包> -> $

<声明闭包> -> , <声明> <声明闭包>

<声明闭包> -> $

<复合函数块> -> { <函数块> }

<函数块> -> <声明语句闭包> <函数块闭包>

<声明语句闭包> -> <声明语句> <声明语句闭包>

<声明语句闭包> -> $

<声明语句> -> <声明> ;

<声明语句> -> <参数声明>;

<函数块闭包> -> <赋值语句> <函数块>

<函数块闭包> -> <for循环> <函数块>

<函数块闭包> -> <while循环> <函数块>

<函数块闭包> -> <条件语句> <函数块>

<函数块闭包> -> <函数返回> <函数块>

<函数块闭包> -> <PRINT函数> <函数块>

<函数块闭包> -> <PRINTF函数> <函数块>

<函数块闭包> -> <SCAN函数> <函数块>

<函数块闭包> -> <SCANF函数> <函数块>

<函数块闭包> -> <CONTINUE> <函数块>

<函数块闭包> -> <BREAK> <函数块>

<函数块闭包> -> $

<赋值语句> -> <类型> <变量> <赋值>

<赋值语句> -> <变量> <赋值>

<赋值> -> = <右值> ;

<赋值> -> += <右值> ;

<赋值> -> -= <右值> ;

<赋值> -> \*= <右值> ;

<赋值> -> /= <右值> ;

<赋值> -> ++;

<赋值> -> --;

<参数>-> <变量>

<参数> -> <常数>

<for循环> -> for <for循环块>

<for循环块> -> ( <赋值语句> <逻辑表达式> ; <因式1> ) <复合函数块>

<for循环块> -> ( <赋值语句> <逻辑表达式> ; <因式1> ) <一条语句>

<逻辑表达式> -> <逻辑因式1>||<逻辑表达式>

<逻辑表达式> -> <逻辑因式1>

<逻辑因式1> -> <逻辑因式2> && <逻辑因式1>

<逻辑因式1> -> <逻辑因式2>

<逻辑因式2> ->true

<逻辑因式2> ->false

<逻辑因式2>-><算术表达式><逻辑运算符><算术表达式>

<逻辑因式2>->(逻辑表达式)

<逻辑运算符> -> <

<逻辑运算符> -> <=

<逻辑运算符> -> >

<逻辑运算符> -> >=

<逻辑运算符> -> <>

<逻辑运算符> -> ==

<逻辑运算符> -> ||

<逻辑运算符> -> &&

<while循环> -> while<while循环块>

<while循环块> -> ( <逻辑表达式> ) <复合函数块>

<while循环块> -> ( <逻辑表达式> ) <一条语句>

<条件语句> -> if <条件语句块>

<条件语句块> -> ( <逻辑表达式> ) <复合函数块> <否则语句>

<条件语句块> -> ( <逻辑表达式> ) <一条语句> <否则语句>

<否则语句> -> else <复合函数块>

<否则语句> -> else <一条语句>

<否则语句> -> $

<函数返回> -> return <因式> ;

<PRINT函数> -> print( <参数列表> ) ;

<SCAN函数> -> scan( <参数列表> )

<一条语句> -><赋值语句>

<一条语句> -><for循环>

<一条语句> -><while循环>

<一条语句> -><函数返回>

<一条语句> -><条件语句>

<一条语句> -><声明语句>

我将主要就以下文法主要几个方面在此详细解释：

1. 程序

程序也是整个语法分析识别的模块，由以上文法可看出，我将程序限制在了函数之内，即函数是我所定义文法的范围，任何函数外的语句本语法分析模块均无法识别。

1. 变量

变量包括标识符与数组两种形式，由于数组结构的特殊性，我在词法分析程序中并未将标识符与数组分开识别，而是在语法分析中识别一个标志符以及包含在[]中的因式（可以是算术表达式或常数或变量）。

1. 算术表达式

算术表达式即由+，-，\*，/，++，--等运算符以及（）组成的表达式，其中的与运算符组合的单元可以是数组、常数、变量。另外由于运算符隐含的结合性、优先级，如果不进行语法上的严格区分，而仅仅简单地使用如下表达

<算数表达式> -> <算数表达式><运算符><算数表达式>|$

这样不仅由于左递归极容易产生回溯，也会导致算术表达式的结果不唯一（稍微改变算术表达式内部顺序便获得不一样的结果）且与数的运算性质相违背（即使想构造另类的运算性质也需要进行优先级、结合性的区分）。在此，我使用C语言数的运算性质，即++、--优先级高于\*、/，\*、/优先级高于+、-。相同优先级则使用右结合性即右边的运算优先级更高。

改进后的文法形成的语法树更深、非终结符更多、文法结构也更复杂。如以上文法算术表达式部分。

1. 逻辑表达式

逻辑表达式与算术表达式类似，由||，&&，>=,<=,<>,==等逻辑运算符以及（）组成的表达式，其中的与运算符组合的单元为算数表达式。逻辑运算符同样也有优先级与结合性带来的问题，本次实验规定&&、||高于其他逻辑运算符，运算符同样是右结合的。故与算术表达式采用同样的增加文法深度的等方法实现。如以上文法逻辑表达式部分。

1. 函数块

我将所有语句块限制在函数中，也即是在{}复合函数块中，考虑到语句块的递归性与左递归易带来的回溯，将语句块的递归设置为右递归，并根据语句类型设置递归出口。

1. 赋值与声明语句

赋值语句与声明语句类似，但声明语句优先级较高，故识别定义初始化语句时，识别为声明语句。当语句开头没有类型时，识别为赋值语句。

1. 循环语句

循环语句可以是while循环以及for循环，文法定义如上

1. continue break

在词法分析中将两者作为词法单元识别，在语法分析中只对位于循环块中的该词法单元以及其后的；进行简单标注为continue语句或break语句。具体功能语义分析中实现

1. print，scan

为方便语义分析结果的输出，定义两个用来打印和读取键盘输入的词法单元，语法分析中将其后跟着的（参数）识别为print语句或scan语句。具体功能语义分析中实现

可能已经发现，在词法分析模块中，我并未进行负数的识别，这是因为负数前面的-符号，在无更多上下文信息时，不管是识别为-运算符还是负号标识都是可能不正确的，因此我将其放在需要分析更多上下文信息的语法分析中实现。以

<因式> -> -<常数>为例

此种情况下，将-运算符与其后常数识别为因式整体，语义分析中解释为0-<常数>。

**3．2．3 输入输出设计**

经过词法分析模块分析之后，我们将获得输入源程序的词法单元数组形式，将该词法单元数组作为语法分析模块的输入，进行顺序处理。语法分析后，我们需要得到含有语法分析信息的输出形式，在本模块中，以语法分析树作为分析结果，将语法分析树按树关系打印在控制台中。因此需要设计一个描述树结构的类。

**3．2．4** **模块结构与具体算法**

本语法分析模块实现主要包含以下几个结构：

**SyntaxAnalyze**类（语法分析核心算法具体实现的类）、**TreeNode**类（定义语法树节点单元的类）、**TreeNodeType**类（定义语法树节点类型的类）

**TreeNode**类：成员变量如下：

    private TreeNode parentNode;*//父节点*

    private ArrayList<TreeNode> children;*//孩子节点*

    private int type;*//节点类型*

    private Token value;*//终结符树节点的值*

    private String callName; *//节点名*

构造函数分为中间节点构造函数（参数为type， parentNode，name）和叶子节点构造函数（参数为type，token，parentNode，name），具体方法有一系列存取自身私有成员变量的get、set方法（Token中将成员变量设置为public，是一种不安全的使用方式，故在TreeNode类中以私有类型声明，外部类只能通过调用它的方法进行设置其私有属性）

**TreeNodeType**类：语法树节点类型即对应的非终结符对应的方法名或终结符类型，定义一个final修饰的包含节点类型信息的类，根据非终结符对应的方法名或终结符类型：

    public static int PROGRAM = 0;  *//程序*

    public static int TERMINAL\_SYMBOL = 1;  *//终结符*

    public static int declar\_stmt = 2;  *//函数定义或声明*

    public static int type = 3;      *//类型*

    public static int variable = 4;  *//变量*

    public static int arrayIndex = 5;  *//数组下标*

    public static int factor = 6;  *//因式*

    public static int factor1 = 36;  *//因式1*

    public static int arith\_expression = 7;  *//算数表达式*

**SyntaxAnalyze**类：

本语法分析模块采用递归下降的方法，需要为每个非终结符定义各自的函数，来递归调用其他函数或是自己。

非终结符即是上述CMM文法中文法表达式左边的<>,每个非终结符都有自己对应的函数(函数命名为非终结符的英文表达)，并且根据对应文法表示式右边的值在函数体中控制继续调用其他非终结符对应函数或匹配非终结符。若词法单元顺序不满足该调用控制，则记录对应错误信息并扫描结束时抛出。

故**SyntaxAnalyze**类成员变量需包含词法分析模块的词法单元数组输入，当前词法单元、错误信息、当前节点、语法树头结点信息。如下：

    private List<Token> tokenList;

    private Token curToken;

    private int tokenIndex = 0;

    private TreeNode curNode;

    private TreeNode headNode;

    private String errorInfo = "";

*//continue、break需在循环内，若在非循环块中出现，则不符合语法规范*

    private int loop = 0; *//标识循环*

为将当前node信息加入语法树、获得词法单元数组中当前词法单元的前后词法单元便于条件判断、以及区分表达式中括号属于逻辑表达式还是算数表达式创建了以下几个工具函数。

**addNonTerminal**(int type, String name)，**addTerminal**(Token token)

**getNextToken**()、 **getPreToken**()、**matchToken**()、**islogical**(int index)、

**printTree**（TreeNode headNode,String space）//将构建的语法树桉树结构打印出来

主分析函数为parse（），它将扫描词法单元数组的第一个词法单元，并构建语法树头节点为Program类型（即源程序头），根据文法表达式

进入右边表达式的第一个值<函数定义或声明>(对应函数为declare\_stmt())，从该函数返回后，进入右边表达式的第二个值即递归调用自己，返回后分析结束。语法分析模块大致递归子程序法算法流程及函数调用大致类似，以相应的文法表达式为根据，进行不同函数的调用或递归，若调用流程出错则说明该词法单元数组即源程序不满足本文法，记录错误信息。

下图为本语法分析模块中非终结符对应的所有函数：



由于各文法表达式递归调用流程大致类似，在此我只详细介绍4.2.2中列举的几个主要相关文法表达式的实现。

1. 变量

对应图中variable()方法，根据对应文法表达式：<变量> -> ID <数组下标>。

进入该方法后首先判断当前词法单元是否为标识符类型，是则调用**addNonTerminal**(TreeNodeType.variable, "变量")，该函数将变量这一非终结符创建对应树节点并将该节点的parentNode设为curNode(即上一节点)，将该节点加入curNode节点的孩子节点中，再将当前节点curNode置为新创建的节点。之后继续调用matchToken()函数，该将当前词法单元（因为当前词法单元为标识符，为终结符）传入**addTerminal**(curToken)函数中并获得下一词法单元，此函数作用与**addNonTerminal（）**相似，区别在于将标识符这一终结符创建对应叶子节点，加入curNode节点的孩子节点中且不更新curNode。matchToken函数返回后，调用arrayIndex（）函数，此函数为数组下标的识别，根据文法表达式

<数组下标> -> [ <因式> ]

<数组下标> -> $

进行类似识别。返回后，将当前节点curNode置为当前节点的parentNode（该步骤在每一个非终结符对应的函数最后都需要使用，来构造树节点之间的正确关系）。

1. 算术表达式

对应图中arith\_expression()方法，根据对应文法表达式：

<算数表达式> -> <因子> <项>

进入该方法后首先判断当前词法单元是否为*常数、标识符、(、++、--、-，*是则调用**add****NonTerminal**(TreeNodeType.arith\_expression, "算术表达式")。之后继续调用**factors**()函数（对应因子非终结符），之后继续调用**term**()（对应项非终结符）根据对应文法规则：

<因子> -> <因式1> <因式递归>

进**factors**()函数后，调用**addNonTerminal**(TreeNodeType.factors, “因子”),再调用**factor1**()（对应因式1非终结符）函数，再调用**factor\_recursion**()(对应因式递归非终结符)。根据文法表达式：

<因式1>->++<因式>|--<因式>|<因式>++|<因式>--|<因式>

进**factor1**()函数后，调用**addNonTerminal**(TreeNodeType.factor1, “因式1”)，判断当前词法单元是否为++、--，是则调用matchToken(),然后调用**factor**()函数（对应非终结符为因式1），不是则直接调用**factor**()函数，然后判断当前词法单元是否为++、--，是则调用matchToken()。根据文法表达式：

<因式递归> -> \* <因式1> <因式递归>| / <因式1> <因式递归>| $

进**factor\_recursion**()函数后，判断当前词法单元是否为/、\*，是则调用**addNonTerminal**(TreeNodeType.factor\_recursion, “因式1”)，

再调用matchToken（），再递归调用**factor1**()和**factor\_recursion**()。

根据文法表达式：

<项> -> + <因子> <项>| - <因子> <项>|$

进**term**()函数后，判断当前词法单元是否为+、-，是则调用**addNonTerminal**(TreeNodeType.term, “项”)，再调用matchToken（），再递归调用**factors**()和**term**()。

返回后，将当前节点curNode置为当前节点的parentNode（该步骤在每一个非终结符对应的函数最后都需要使用，来构造树节点之间的正确关系）

1. 逻辑表达式

对应图中**logical\_expression()**方法，根据对应文法表达式：

<逻辑表达式>-> <逻辑因式1>||<逻辑表达式> |<逻辑因式1>

进入该方法后首先判断当前词法单元是否为*常数、标识符、(、True、False，*是则调用**addNonTerminal**(TreeNodeType.logical\_expression, "逻辑表达式")。之后继续调用**logical\_factor1**()函数（对应逻辑因式1非终结符），之后判断当前词法单元是否为||，是则调用**matchToken()，**之后继续递归调用**logical\_expression**()。根据对应文法规则：

<逻辑因式1> -> <逻辑因式2> && <逻辑因式1> |<逻辑因式2>

进**logical\_factor1** ()函数后，判断当前词法单元是否为*常数、标识符、(、True、False，是则*调用**addNonTerminal**(TreeNodeType.logical\_factor1, “逻辑因式1”)。之后继续调用**logical\_factor2**()函数（对应逻辑因式2非终结符），之后判断当前词法单元是否为&&，是则调用**matchToken()，**之后继续递归调用**logical\_factor1**()。根据对应文法规则：

<逻辑因式2>-><算术表达式><逻辑运算符><算术表达式> | false| true

<逻辑因式2>->(逻辑表达式)

进**logical\_factor2** ()函数后判断当前词法单元是否为*常数、标识符、(、True、False，是则*调用**addNonTerminal**(TreeNodeType.logical\_factor2, “逻辑因式2”)。调用工具函数**islogical**(tokenIndex),若返回为真，则调用matchToken(),再调用logical\_expression(),之后判断当前词法单元是否为右括号，是则调用matchToken()。若返回为假，调用arith\_expression(),判断当前词法单元是否为<、<=、>、>=、<>、==、||、&&，是则调用matchToken(),再调用arith\_expression()。

返回后，将当前节点curNode置为当前节点的parentNode（该步骤在每一个非终结符对应的函数最后都需要使用，来构造树节点之间的正确关系）

1. 赋值语句

对应图中**assignment()**方法，根据对应文法表达式：

<赋值语句> -> <类型> <变量> <赋值>

<赋值语句> -> <变量> <赋值>

进入该方法后判断当前词法单元是否为int,bool,float,是则调用**addNonTerminal**(TreeNodeType.assignment, “赋值语句”)，再调用**type**()（对应类型非终结符）方法，再调用**variable**()（对应变量非终结符）方法，再调用**assign**()（对应赋值非终结符）方法。不是则调用**addNonTerminal**(TreeNodeType.assignment, “赋值语句”)，再调用**variable**()（对应变量非终结符）方法，再调用**assign**()（对应赋值非终结符）方法。根据对应文法规则：

<赋值> -> (=|+=|-=|/=|\*=) <右值> ;

<赋值> -> ++;

<赋值> -> --;

进入**assign**()方法后,判断当前词法单元是否为*=、+=、-=、\*=、/=*，是则调用**addNonTerminal**(TreeNodeType.assign, “赋值”)，再调用matchToken()，再调用**rightVal**()（对应右值非终结符），判断当前词法单元是否为分号，为分号调用matchToken()。不是*=、+=、-=、\*=、/=*则调用**addNonTerminal**(TreeNodeType.assign, “赋值”)，再调用matchToken()，判断当前词法单元是否为分号，为分号调用matchToken()。根据对应文法规则：



返回后，将当前节点curNode置为当前节点的parentNode（该步骤在每一个非终结符对应的函数最后都需要使用，来构造树节点之间的正确关系）

1. 复合函数块

对应图中**co\_function ()**方法，根据对应文法表达式：

<复合函数块> -> { <函数块> }

进入该方法后判断当前词法单元是否为{,是则调用**addNonTerminal**(TreeNodeType.co\_function, “复合函数块”)，再调用matchToken()，再调用**functionBlock**()（对应函数块非终结符）方法，判断当前词法单元是否为}，是则调用matchToken()。根据对应文法规则：

<函数块> -> <声明语句闭包> <函数块闭包>

进入**functionBlock** ()方法后,调用**addNonTerminal**(TreeNodeType.functionBlock, “函数块”)，再调用**stmt\_expression\_closure**()（对应声明语句闭包非终结符），再调用**functionBlock\_closure**()（对应函数块闭包非终结符）。根据对应文法规则：

<函数块闭包> -> (<赋值语句>| <for循环>| <while循环>|<条件语句>|<函数返回>|<PRINT函数>|<SCAN函数>| <CONTINUE>|<BREAK>) <函数块>|$

进入**functionBlock\_closure**()方法后，根据当前词法单元类型判断调用哪一函数。

返回后，将当前节点curNode置为当前节点的parentNode（该步骤在每一个非终结符对应的函数最后都需要使用，来构造树节点之间的正确关系）

1. 循环语句

循环语句可以是while循环以及for循环，以while循环为例，对应图中**whileloop ()**方法，根据对应文法表达式：

<while循环> -> while<while循环块>

进入**whileloop ()**方法后，判断当前词法单元是否为while，是则将类成员变量loop加一（用来判断是否位于循环块，用来识别正确的break，continue语句），再调用**addNonTerminal**(TreeNodeType.whileloop,“while循环”)，再调用matchToken(),再调用**whileblock**()（对应while循环块非终结符），返回后，将loop减一。根据对应文法规则：

<while循环块> -> ( <逻辑表达式> ) <复合函数块>

<while循环块> -> ( <逻辑表达式> ) <一条语句>

进入**whileblock**()方法后，判断当前词法单元是否为(，是则调用matchToken(),再调用logical\_expression(),再判断当前词法单元是否为)，是则调用matchToken()，再判断当前词法单元是否为{，是则调用**co\_functon**(),不是则调用**astament**()方法（对应一条语句非终结符）

1. continue break

在词法分析中将两者作为词法单元识别，在语法分析中只对位于循环块中的该词法单元以及其后的；进行简单标注为continue语句或break语句。具体功能语义分析中实现

1. print，scan

为方便语义分析结果的输出，定义两个用来打印和读取键盘输入的词法单元，语法分析中将其后跟着的（参数）识别为print语句或scan语句。具体功能语义分析中实现

**3．2．5** **错误处理**

在每个非终结符的递归函数内，通过当前词法单元是否为对应的文法定义中的非终结符，进行错误信息记录，当扫描完成后，输出错误信息。错误处理大致格式如下：

*//当前token为)时*

        if((curToken.type==TokenType.RIGHT\_PARENTHESIS)){

**matchToken**();

        }else{

            errorInfo += "error: line" + curToken.line + "," + "  " +

            curToken.strval +  "    expect: 缺少右括号;\n";

            }

**3.3**  **测试**

**3．3．1** **测试用例**

为了检验CMM语言语法分析的正确性，我们的测试用例应该尽量覆盖到所有的代码书写可能遇到的情况，不仅局限于正规的书写。测试用例中覆盖的检测点如下所示：

各种语句的组合：

声明语句，声明数组

while语句，复杂逻辑表达式并嵌套算术表达式，多重循环嵌套和条件语句嵌套

数组操作，计算

复杂算术表达式和自增减嵌套计算

测试用例如下：

int **main1**(){

   int m;

   int n;

   int v[n], int c[n]; *// value cost*

   int i = 0;

   int a[(m+1)];

   while(i < n){

     int j = m;

     while(j >= c[i]){

       if(a[j] < a[(j - c[i])] + v[i])

       {

         a[j] = a[(j - c[i])] + v[i];

       }

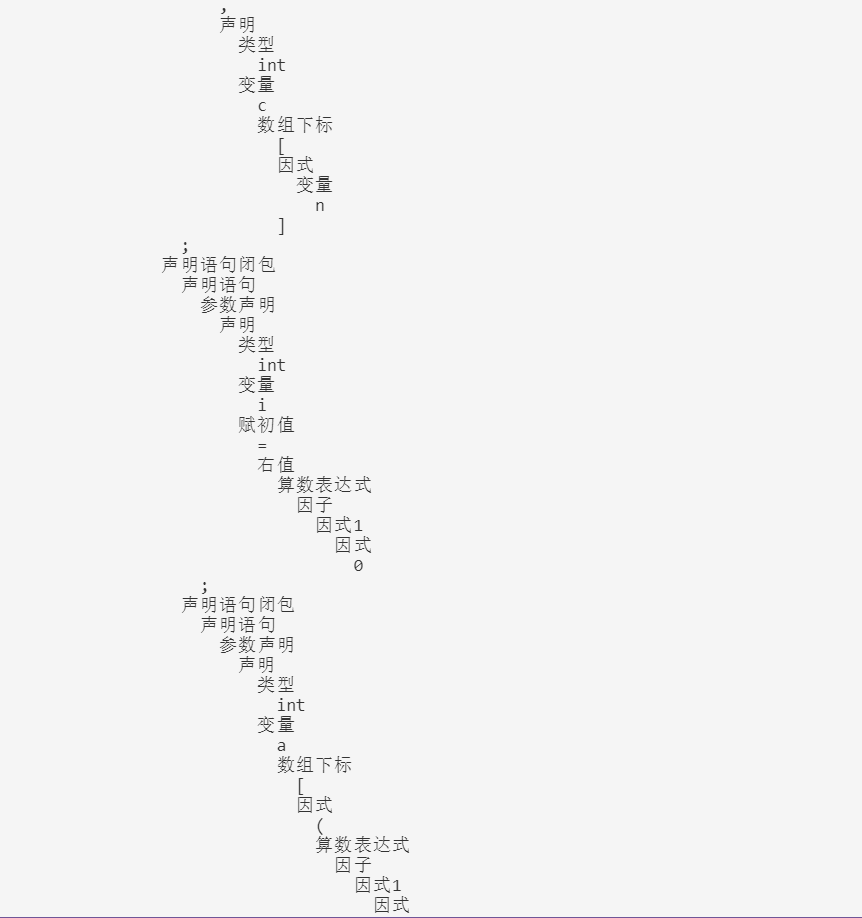
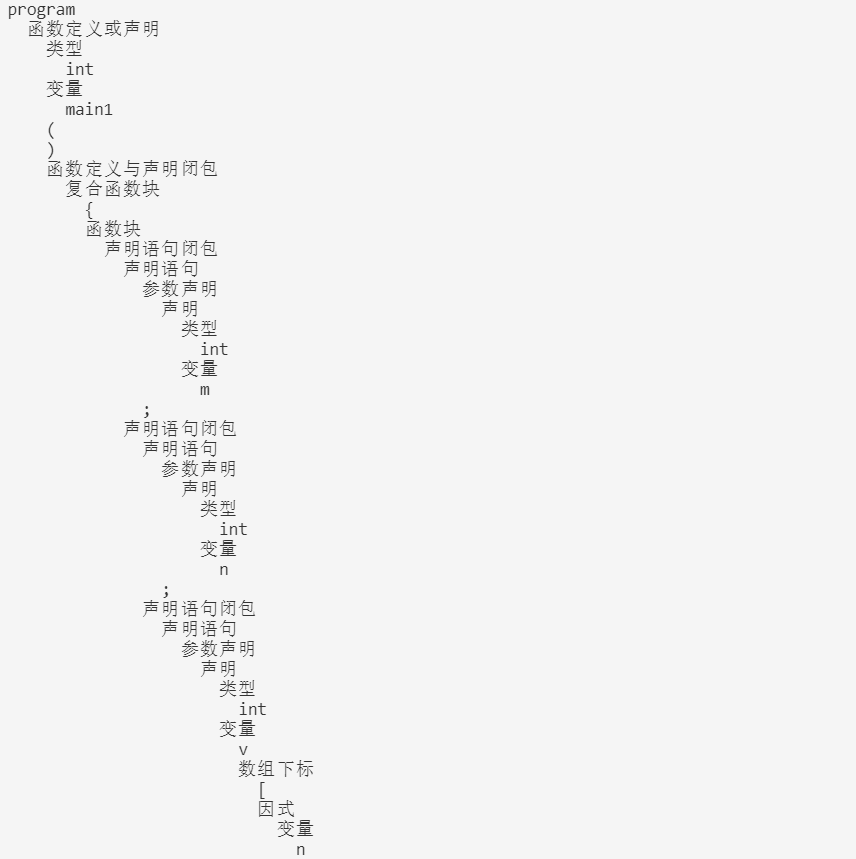
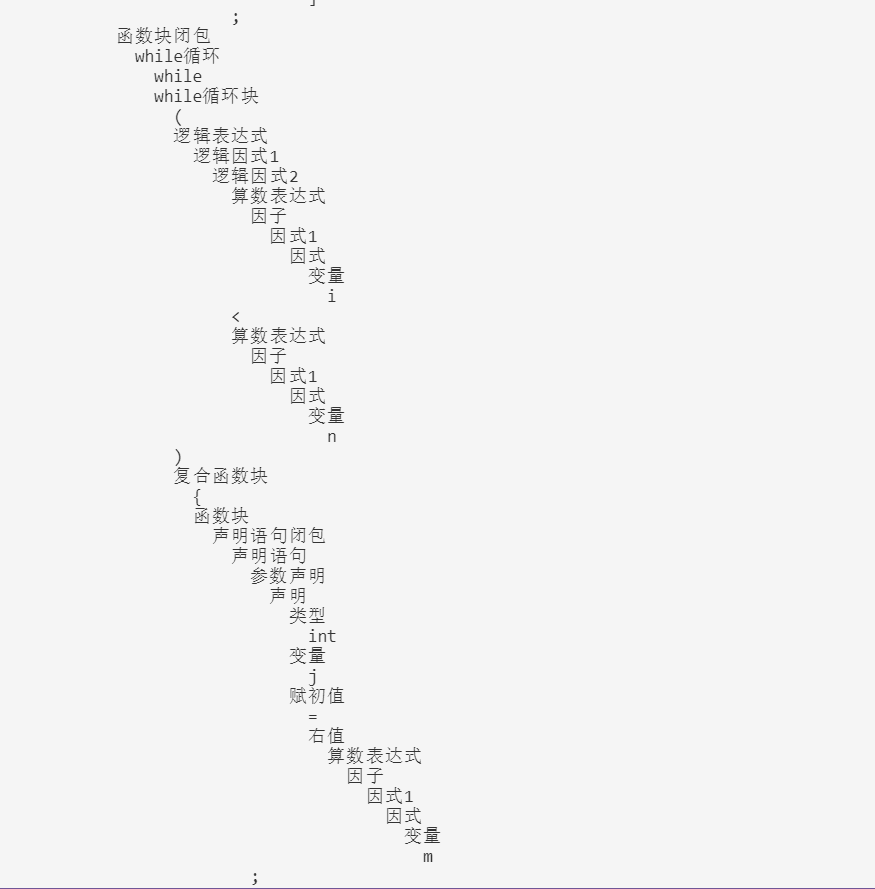
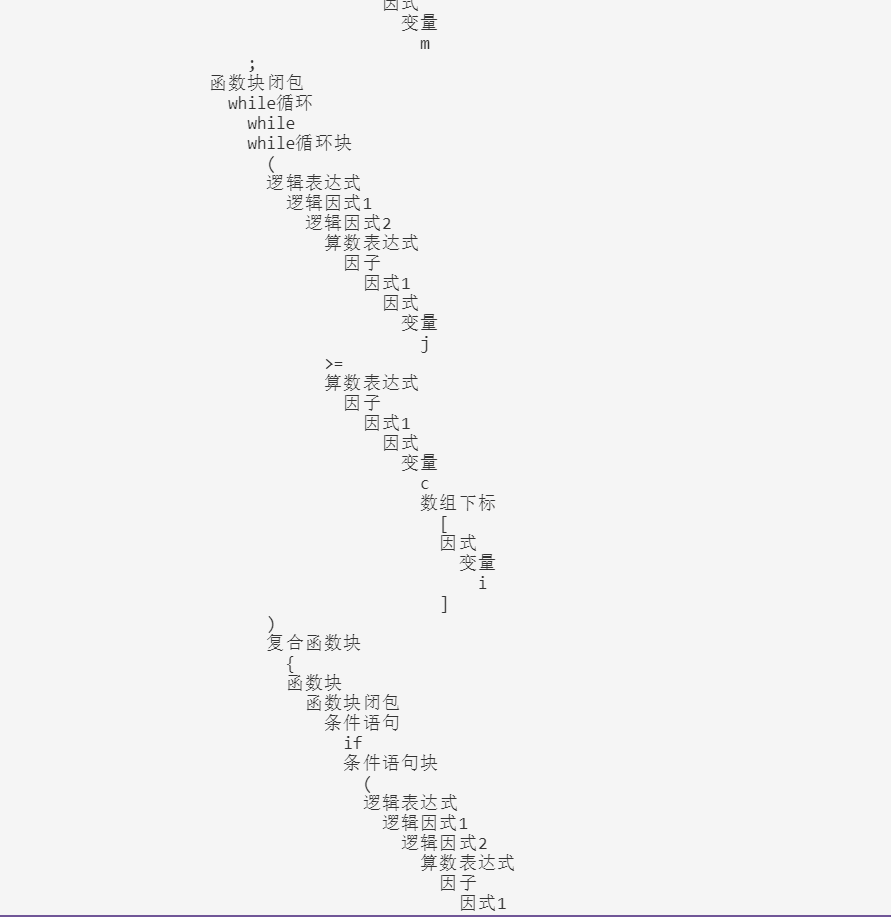
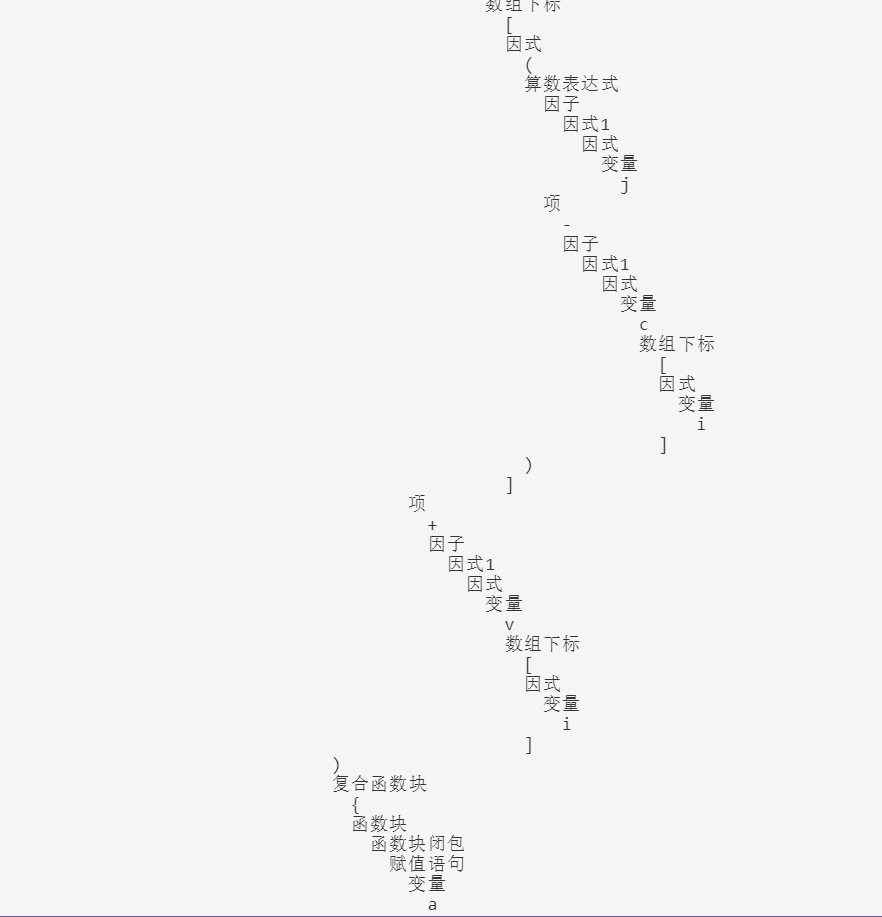
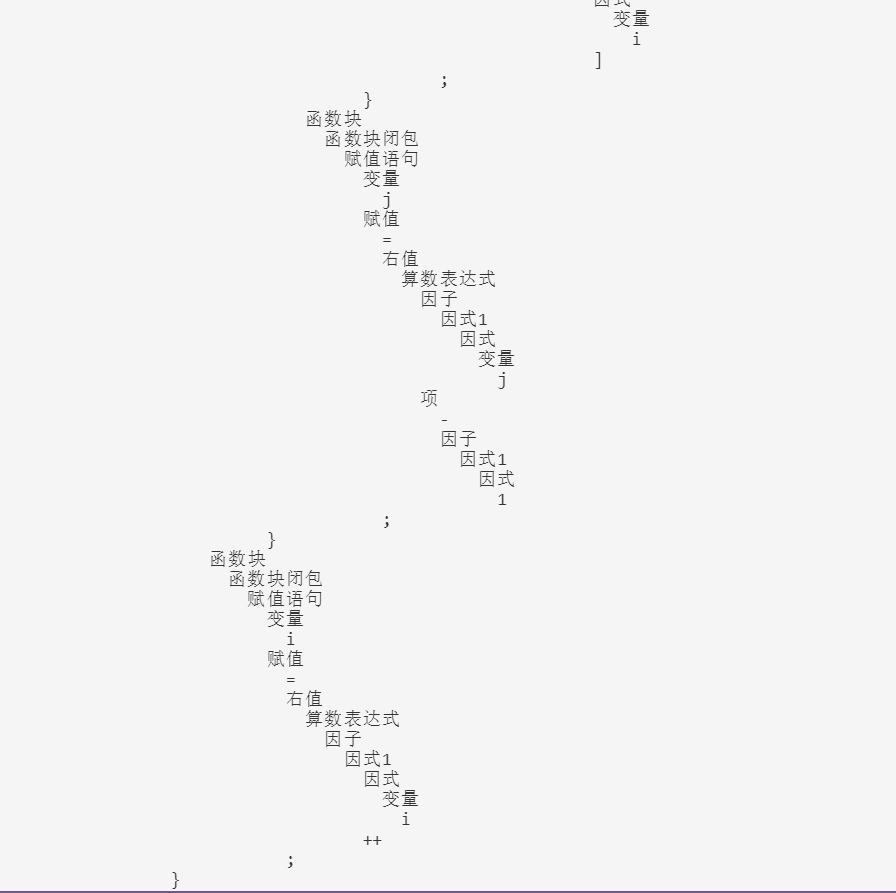
       j = j - 1;

     }

     i = i++;

     }

**3．3．2** **测试结果**

**4 语义分析**

**4.1** **概述**

经过语法分析模块分析之后，我们将获得输入源程序的语法树形式，但此时只是对源程序的初级解释，要让解释器做到我们需要它做到的事，还需要给它赋予含义，让解释器能完全理解源程序，也就是语义分析。因此，语义分析模块将是本次解释器构造的最后一步，但语义分析并没有统一的形式化描述，因此，我们借助编译原理学到的知识，使用语法制导翻译的方法来实现语义分析，而语法树已经有语法分析模块得到了，故我们需要对语法树进行扫描。

**4.2 实验原理与方案**

**4．2．1** **实验原理**

语法分析阶段构造的CMM文法最后得到了符合语法规范的语法树，但对于一个完整的解释器功能来说，还有所欠缺，需要对语法赋予语义，反应在文法层面上，就是扩充文法，将CMM文法进行语义规则上的扩充，因此，从这方面来说，语义分析只是另一次语法分析而已。 在语法分析的过程中，依随分析的过程，根据每个产生式添加的语义动作进行翻译。一旦某个产生式被选用于推导或归

约，就执行其后相应的语义动作，完成预定的翻译工作。语法分析与语义分析穿插进行，语法分析引导语义分析。因此我们沿用语法分析的递归下降法以及递归关系来进行语义分析

**4．2．2** **语义规则**

语义是上下文有关的，因此需要记录上下文的关系，我们将使用符号表的方法来对上下文内容进行存储匹配。

语义规则主要分为以下两类：

数据结构的含义——名字的含义(类型正确性检查)。

控制结构的含义——语言自身定义(形式化与非形式化)。

为了实现这些规则，我们需要对文法中的非终结符号或者终结符号引入一些属性，描述相应语言结构的语义值（性质）。为产生式附加语义子程序，用于计算文法符号的属性值（属性值的计算，由语法分析过程中产生的语法分析树相应结点

的环境推导出来。）或是查填符号表的操作、打印出错信息的操作、生成代码的操作等。

赋值语句语义规则：

<赋值语句> -> <类型> <变量> <赋值>

<赋值语句> -> <变量> <赋值>

<赋值> -> (=|+=|-=|/=|\*=) |<右值> ;

<赋值> -> ++;

<赋值> -> --;

将右值以某种运算关系处理后传给变量，若是自增减运算，则对变量进行相应加减一。对右值进行类型检查，以某种类型转换规则与变量类型相作用。

条件控制语句语义规则：

<条件语句> -> if <条件语句块>

<条件语句块> -> ( <逻辑表达式> ) <复合函数块> <否则语句>

<条件语句块> -> ( <逻辑表达式> ) <一条语句> <否则语句>

<否则语句> -> else <复合函数块>

计算逻辑表达式值，若为真执行复合函数块或一条语句，若为假执行否则语句中复合函数块中语句

循环控制语句语义规则：

<while循环> -> while<while循环块>

<while循环块> -> ( <逻辑表达式> ) <复合函数块>

<while循环块> -> ( <逻辑表达式> ) <一条语句>

<for循环> -> for <for循环块>

<for循环块> -> ( <赋值语句> <逻辑表达式> ; <因式1> ) <复合函数块>

<for循环块> -> ( <赋值语句> <逻辑表达式> ; <因式1> ) <一条语句>

对于while循环，先计算逻辑表达式值，若为假退出循环，若为真执行复合函数块或一条语句， 然后继续计算逻辑表达式值，重复。

对于for循环，先计算逻辑表达式值，若为假退出循环，若为真执行复合函数块或一条语句， 执行因式1，然后继续计算逻辑表达式值，重复。

Continue语句语义规则：遇到此语句，跳至本层循环结构条件判断逻辑表达式（while循环）或执行完因式1再进行逻辑表达式的判断（for循环），执行本层循环结构后一语句。

Break语句语义规则：遇到此语句，退出本层循环，执行本层循环结构后一语句。

负数语义规则：在词法分析、语法分析中并未对负数进行单独识别，在语义分析中赋予负数语义，-运算符与其后常数组合为语法因式时，对应操作为0-常数；-运算符与其后变量组合为语法因式时，对应操作为0-变量值；-运算符与其后算术表达式组合为语法因式时，对应操作为0-算术表达式结果

其他语义规则：每个变量作用范围仅在定义它的复合函数块内，当跳出该复合函数块时，将其在符号表中删去。

**4．2．3** **模块结构和具体算法**

**模块结构与具体算法**

本语义分析模块实现主要包含以下几个结构：

**SemanticAnalyze**类（词法分析核心算法具体实现的类）、**Record**类（定义记录（符号）的类）、**SymbolTable**类（实现符号表的类）

**Record**类：成员变量如下：

    private int level;*//变量的层次*

    private Token token;*//对应的标识符（不一定有）*

    private String name;

    private int type;*//数据类型*

    private int intVal;*//int类型数据的值*

    private boolean boolVal;*//boolean类型数据的值*

    private float floatVal;*//float类型数据的值*

    private int arrayNum;*//数组类型数据的大小*

    private int arrayIndex;*//数组下标,未赋值时为空*

    private int[] intArray;*//int类型数组*

    private boolean[] boolArray;*//boolean类型数据的值*

private float[] floatArray;*//float类型数组*

*//记录Record变量类型的类成员变量*

  public static int tInt = 0;

    public static int tbool = 1;

    public static int tFloat = 2;

    public static int tIntArray = 3;

    public static int tboolArray = 4;

    public static int tFloatArray = 5;

构造函数分为int变量构造函数（参数为level, token, type, name, intVal）、 bool变量构造函数（参数为level, token, type, name, boolVal）、float变量构造函数（参数为level, token, type, name, floatVal）、int数组变量构造函数（参数为level, token, type, name, intArray）、bool数组变量构造函数（参数为level, token, type, name, boolArray）、float数组变量构造函数（参数为level, token, type, name, floatArray）。具体方法有一系列存取自身私有成员变量的get、set方法（Token中将成员变量设置为public，是一种不安全的使用方式，故在Record类中以私有类型声明，外部类只能通过调用它的方法进行设置其私有属性）、还有对record类型值进行操作的一系列方法getvalue（）、setvalue（Record record）、addvalue（Record record）、substractvalue（Record record）、dividevalue（Record record）、mutiplyvalue（Record record）、

**SymbolTable**类：成员变量如下：

  private ArrayList<Record> table;*//符号表*

使用Record类型的数组列表实现符号表，该类方法全为对符号表的增删改查操作，判断记录是否已定义的**haveDefined**(String name)、清除内层变量的**s**(int level)方法，通过名字获取record的**getRecordByName**(String name)

**SemanticAnalyze**类（算法实现类）：

本语义分析模块采用递归下降的方法，并沿用语法分析的方法名与调用关系。

故**SemanticAnalyze**类成员变量需包含词法分析模块的词法单元数组输入，当前词法单元、错误信息、当前节点、语法树头结点信息。如下：

    private TreeNode headNode;  *//记录语法树的头结点*

    private SymbalTable symbalTable; *//符号表*

    private TreeNode curNode;

    private Token curToken;

    private int level = 0; *//变量所属层级*

    private String errorInfo = "";

    private boolean notcontinue = true; *//判断continue语句*

主分析函数为semantic（），它将扫描语法树的头结点即待解释源程序入口，根据文法规则与语法分析步骤：

进入 declare\_stmt()函数，将当前节点移至当前节点的第一个孩子节点，进行一系列函数调用，再将当前节点移至当前节点的parentNode，从该函数返回后，进入右边表达式的第二个值即递归调用自己，并将当前节点移至当前节点的第二个孩子节点，进行一系列函数调用，再将当前节点移至当前节点的parentNode，返回后分析结束。语义分析模块大致递归子程序法算法流程及函数调用大致类似，以相应的语法分析递归调用关系（也即文法规则）和语义规则为根据，进行不同函数的调用或递归，在相应的需要增加语义规则的地方使用语义规则（即对记录项进行操作），并记录错误信息。

由于各递归调用流程大致类似，在此我只详细介绍以下几个主要语法的语义分析相应实现。

1. 变量

对应variable()方法，根据对应文法表达式：<变量> -> ID <数组下标>。

进入该方法后首先将当前节点移至当前节点的第一个孩子节点，当前节点变为ID标识符，从符号表中根据当前节点的名字得到对应record项，若没有，则表示该变量还未声明，直接返回；若有，则判断当前节点的parentNode的子节点数量是否大于1（间接判断该变量是否为数组类型），若大于1，则调用factor（）（对应为因式节点）获得返回值设为当前数组下标值，对越界情况进行判断并错误抛出处理，返回record项；若小于1，则直接返回record项。在每个返回前，将当前节点置为进入函数时的对应节点。

1. 算术表达式

对应arith\_expression()方法，根据对应文法表达式：

<算数表达式> -> <因子> <项>

进入该方法后首先将当前节点移至当前节点的第一个孩子节点，当前节点为因子，调用**factors**()(因子节点对应扫描函数)根据对应文法规则：

<因子> -> <因式1> <因式递归>

进**factors**()函数后，将当前节点移至当前节点的第一个孩子节点，当前节点为因式1,再调用**factor1**()（对应因式1节点扫描函数）函数，再将当前节点移至当前节点的parentNode的第二个孩子节点，当前节点为因式递归,再调用**factor\_recursion**()(对应因式递归节点扫描函数)。根据文法表达式：

<因式1>->++<因式>|--<因式>|<因式>++|<因式>--|<因式>

进**factor1**()函数后，将当前节点移至当前节点的第一个孩子节点，判断若当前节点为因式，调用**factor**()(对应因式节点扫描函数)，得到record类型返回值var，再将当前节点移至当前节点的parentNode的第二个孩子节点，判断若当前节点为++，则将var对应的值加1，返回var，若当前节点为--，则将var对应的值减1，返回var，其他情况则直接返回var。若当前节点不为因式为++，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**factor**()，将得到的record类型返回值var的值加1，返回var，若当前节点不为因式为--，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**factor**()，将得到的record类型返回值var的值减1，返回var。根据文法表达式：

<因式递归> -> \* <因式1> <因式递归>| / <因式1> <因式递归>| $

进**factor\_recursion**(Record record)函数后，将当前节点移至当前节点的第一个孩子节点，判断当前节点的token值为\*，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**factor1**()，得到record类型返回值var，调用record.multiplyvalue(var)；判断当前节点的token值为/，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**factor1**()，得到record类型返回值var，调用record.dividealue(var)；再将当前节点移至当前节点的parentNode的第三个孩子节点，调用**factor\_recursion**(record)，返回record。在每个返回前，将当前节点置为进入函数时的对应节点。根据文法表达式：

<项> -> + <因子> <项>| - <因子> <项>|$

进**term**()函数后，将当前节点移至当前节点的第一个孩子节点，判断当前节点的token值为+，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**factors**()，得到record类型返回值var，调用record.multiplyvalue(var)；判断当前节点的token值为-，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**factors**()，得到record类型返回值var，调用record.dividealue(var)；再将当前节点移至当前节点的parentNode的第三个孩子节点，递归调用**term**(record)，返回record。

在每个返回前，将当前节点置为进入函数时的对应节点。

1. 逻辑表达式

对应**logical\_expression()**方法，根据对应文法表达式：

<逻辑表达式>-> <逻辑因式1>||<逻辑表达式> |<逻辑因式1>

进入该方法后首先将当前节点移至当前节点的第一个孩子节点，当前节点为逻辑因式，调用**logical\_factor1** ()(逻辑因式1节点对应扫描函数) ，获得返回值boolean类型left,若left为真，直接返回真。判断当前节点的parentNode的孩子节点个数大于1，若大于1，则再将当前节点移至当前节点的parentNode的第三个孩子节点，调用**logical\_expression**()，获得返回值boolean类型right, 若right为真，直接返回真。若不大于1，返回假根据对应文法规则：

<逻辑因式1> -> <逻辑因式2> && <逻辑因式1> |<逻辑因式2>

进**logical\_factor1** ()函数后，首先将当前节点移至当前节点的第一个孩子节点，当前节点为逻辑因式，调用**logical\_factor2** ()(逻辑因式2节点对应扫描函数) ，获得返回值boolean类型left,若left为假，直接返回假。判断当前节点的parentNode的孩子节点个数大于1，若大于1，则再将当前节点移至当前节点的parentNode的第三个孩子节点，调用**logical\_ factor1**()，获得返回值boolean类型right,返回right&&left。若不大于1，返回真。根据对应文法规则：

<逻辑因式2>-><算术表达式><逻辑运算符><算术表达式> | false| true

<逻辑因式2>->(逻辑表达式)

进**logical\_factor2** ()函数后首先将当前节点移至当前节点的第一个孩子节点，判断当前节点的token值是否为(,是则调用logical\_expression(),若是节点算术表达式，调用arith\_expression()。再将当前节点移至当前节点的parentNode的第三个孩子节点，调用arith\_expression ()，再将当前节点移至当前节点的parentNode的第二个孩子节点，判断当前节点的逻辑运算符是什么，进行对应运算，返回其boolean类型值。

在每个返回前，将当前节点置为进入函数时的对应节点。

1. 赋值语句

对应**assignment()**方法，根据对应文法表达式：

<赋值语句> -> <类型> <变量> <赋值>

<赋值语句> -> <变量> <赋值>

进入该方法后首先将当前节点移至当前节点的第一个孩子节点，判断当前节点是否为类型，是则调用**type** ()（对应类型节点扫描函数）方法，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**variable**()（对应变量节点扫描函数）方法，得到返回值Record类型variable，再将当前节点移至当前节点的parentNode的第三个孩子节点，再调用**assign**()（对应赋值节点扫描函数）方法，得到返回值Record类型record。当前节点不是为类型则调用**variable**(variable)（对应变量节点扫描函数）方法，得到返回值Record类型variable，再将当前节点移至当前节点的parentNode的第二个孩子节点，再调用**assign**(variable)方法。根据对应文法规则：

<赋值> -> (=|+=|-=|/=|\*=) <右值> ;

<赋值> -> ++;

<赋值> -> --;

进入**assign**(Record record)方法后, 首先将当前节点移至当前节点的第一个孩子节点，判断当前节点的token值为++,是则将record的值加1,若为--，则将record的值减1。若为=，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**rightVal()** (对应右值节点扫描函数)，得到返回值Record类型var，根据运算符的类型调用对应的record.setvalue(var); 若为+=，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**rightVal()** (对应右值节点扫描函数)，得到返回值Record类型var，根据运算符的类型调用对应的record.addvalue(var); 若为-=，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**rightVal()** (对应右值节点扫描函数)，得到返回值Record类型var，根据运算符的类型调用对应的record.substractvalue(var);若为/=，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**rightVal()**(对应右值节点扫描函数)，得到返回值Record类型var，根据运算符的类型调用对应的record.dividevalue(var); 若为\*=，再将当前节点移至当前节点的parentNode的第二个孩子节点，调用**rightVal()** (对应右值节点扫描函数)，得到返回值Record类型var，根据运算符的类型调用对应的record.multiplyvalue(var);。根据对应文法规则：



进入**rightVal()**方法后，首先将当前节点移至当前节点的第一个孩子节点，判断当前节点是否为逻辑表达式，是则，调用logical\_expression(),得到返回值Record类型var;不是则调用arith\_expression(),得到返回值Record类型var;返回var。

在每个返回前，将当前节点置为进入函数时的对应节点。

1. 复合函数块

对应**co\_function ()**方法，根据对应文法表达式：

<复合函数块> -> { <函数块> }

进入该方法后将当前节点移至当前节点的第二个孩子节点，调用**functionBlock**()（对应函数块非终结符节点扫描函数）方法。根据对应文法规则：

<函数块> -> <声明语句闭包> <函数块闭包>

进入**functionBlock** ()方法后, 将当前节点移至当前节点的第一个孩子节点，调用**stmt\_expression\_closure**()（对应声明语句闭包非终结符节点扫描函数），再将当前节点移至当前节点的parentNode的第二个孩子节点，再调用**functionBlock\_closure**()（对应函数块闭包非终结符节点扫描函数）。根据对应文法规则：

<函数块闭包> -> (<赋值语句>| <for循环>| <while循环>|<条件语句>|<函数返回>|<PRINT函数>|<SCAN函数>| <CONTINUE>|<BREAK>) <函数块>|$

进入**functionBlock\_closure**()方法后，将当前节点移至当前节点的第一个孩子节点，根据当前节点类型判断调用哪一函数。

在每个返回前，将当前节点置为进入函数时的对应节点。

1. 循环语句

循环语句可以是while循环以及for循环，以while循环为例，对应**whileloop ()**方法，根据对应文法表达式：

<while循环> -> while<while循环块>

进入**whileloop ()**方法后，首先将当前节点移至当前节点的第二个孩子节点，调用**whileblock**()(while循环块节点扫描函数)。根据对应文法规则：

<while循环块> -> ( <逻辑表达式> ) <复合函数块>

<while循环块> -> ( <逻辑表达式> ) <一条语句>

进入**whileblock**()方法后，首先将当前节点移至当前节点的第二个孩子节点，用logical\_expression()的返回值做为循环条件，若为真，将当前节点移至当前节点的第四个孩子节点，再判断当前节点是否为复合函数块，是则调用**co\_functon**(),不是则调用**astament**()方法（对应一条语句非终结符节点扫描函数），循环判断logical\_expression()的返回值。若为假，退出循环。

在每个返回前，将当前节点置为进入函数时的对应节点。

1. continue break

break 语句: 在functionBlock\_closure()中，判断当前节点为break时，进入下列语句：

else if (curNode.**getType**() == TreeNodeType.break\_statment) {

                if(loop>0){

                    notbreak = false;

                }

            }

因此，扫描到break时，函数块闭包结束，回退至复合函数块再退至循环块。

在各函数间传递boolean类型notbreak，将forblock()与whileblock()中的循环结构判断语句变为**logical\_expression**()&&notbreak。因此，语义分析器在检测到循环中的break语句时，回退至循环中，再退至循环外。

continue语句：在functionBlock\_closure()中，判断当前节点为continue时，进入下列语句：

else if (curNode.**getType**() == TreeNodeType.continue\_statment) {

                if(loop>0){

                    notcontinue = false;

                }

            }

判断当前节点为if语句时

else if (curNode.getType() == TreeNodeType.if\_statement) {

notbreak =if\_statement();

if(notbreak && notcontinue){

if(curNode.getParentNode().getChildren().size()>1){

curNode=curNode.getParentNode().getChildren().get(1);

notbreak =functionBlock();

}

}

因此，扫描到continue时，函数块闭包结束，回退至复合函数块再退至循环块。但并不直接退出循环。

当continue位于循环中的条件语句时，退至复合函数块，再退至条件语句块，再退出函数块闭包，再退至复合函数块，再退至循环块，继续判断条件进行循环或退出

**4．2．4** **错误处理**

在每个节点的递归函数内，进行错误信息记录，当扫描完成后，输出错误信息。例如数组越界错误处理、变量多次声明等情况

errorInfo += "重复声明变量：line " + curNode.**getChildren**().**get**(0).**getValue**().line

                + " "+ curNode.**getChildren**().**get**(0).**getValue**().strval + "\n";

**4.3 运行与测试**

**4．3．1** **测试用例**

为了检验CMM语言语义分析的正确性，测试用例中覆盖的要点如下：

各种语句的组合：

声明语句，声明数组，

for、while循环，复杂逻辑表达式嵌套算术表达式，复杂算术表达式和自增减嵌套计算，多重循环嵌套和条件语句嵌套

数组操作，计算

打印和输入

测试用例一如下

int **main1**(){

  int N;

**scan**(N);

*//  if(N == 0)*

*//    return;*

  int text[N];

  int k = 0;

 while(k < N){

**scan**(text[k]);

    k = k + 1;

  }

  N = 2\*N +1; *//Position count*

  int L[N]; *//LPS Length Array*

  L[0] = 0;

  L[1] = 1;

   int i = 0;

  int C = 1; *//centerPosition*

  int R = 2; *//centerRightPosition*

*//currentRightPosition*

  int iMirror; *//currentLeftPosition*

  int maxLPSLength = 0;

  int maxLPSCenterPosition = 0;

  int start = -1;  *///-1*

  int end = -1;

  int diff = -1;

  i = 2;

while(i < N)

    {

      iMirror  = 2\*C-i;

      L[i] = 0;

      diff = R - i;

      if(diff > 0)

        if(L[iMirror] > diff)

          L[i] = diff;

        else L[i] = L[iMirror];

      while ( ((i + L[i]) < N && (i - L[i]) > 0) &&

              ( ((i + L[i] + 1) - (i + L[i] + 1) / 2 \* 2 == 0) ||

             (text[((i + L[i] + 1)/2)] == text[((i - L[i] - 1)/2)] )))

        {

          L[i]++;     }

      if(L[i] > maxLPSLength)

        {

          maxLPSLength = L[i];

          maxLPSCenterPosition = i;

        }

      if (i + L[i] > R)

        {

          C = i;

          R = i + L[i];

        }

      i = i + 1;

    }

  start = (maxLPSCenterPosition - maxLPSLength)/2;

  end = start + maxLPSLength - 1;

i = start;

while(i <= end){

**print**(text[i]);

  i = i + 1;

 }

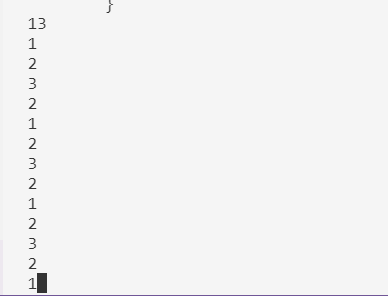
}

**4．3．2** **运行结果**

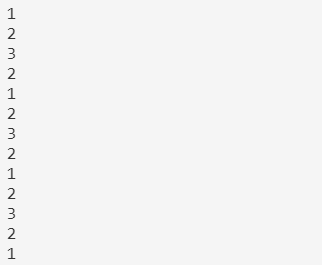
输入为：

13

1 2 3 2 1 2 3 2 1 2 3 2 1



输出



**5 JavaCC**

**5.1** **概述**

本实验为第四部分，也是最后一步，要求使用JavaCC自动生成符合自定义CMM语言文法的词法分析和语法分析程序。

**5.2 实验原理与方案**

**5．2．1** **JavaCC**

JavaCC是一个词法分析器和语法分析器的生成器，用于java领域，根据指定格式的输入文件生成java源码的词法、语法分析程序。 其特性如下：

1. 使用递归下降语法解析，LL(k)。其中，第一个L表示从左到右扫描输入；第二个L表示每次都进行最左推导(在推导语法树的过程中每次都替换句型中最左的非终结符为终结符。类似还有最右推导)；k表示的是每次向前探索(lookahead)k个终结符
2. 词法规则，语法规则定义在同一文件中，就是.jj文件。
3. 更向前一步解决移进规约。当文法本身存在二义性的时候有时候通过设置lookahead为k能解决问题，带来的问题就是增加编译时间，所以最好的方法是修改二义性文法为无二义性文法。

**5．2．2具体实现**

javaCC能根据我们定义的词法、语法规则自动生成对应的词法、语法分析程序，而词法规则，语法规则都需定义在.jj文件，因此整个javaCC的实现也就是.jj文件的完善与实现。

具体来看，jj文件格式如下：

javacc\_options 参数列表

PARSER\_BEGIN( <IDENTIFIER> )

java\_compilation\_unit JAVA编辑单元

PARSER\_END( <IDENTIFIER> )

( production )\* 相关产物

<EOF>

参数列表接下来的是一个处在”PARSER\_BEGIN(name)”和”PARSER\_END(name)”中间的编译单元。这个编译单元可以是任意的复杂。在这个编译单元中唯一的限制就是它必须定一个一个叫”name”的类——与PARSER\_BEGIN和PARSER\_END的参数的相同。这个”name”被用作语法分析产生器生成的java文件的前缀。

其中(production)\*相关产物是我们最关心也是最重要的，又可分为JAVA代码产物、BNF范式产物、正则表达式产物和符号管理器声明四种类型，对于自动构造编译器而言，BNF范式是至关重要的，所生成的编译器就是根据所写的BNF范式构造而来。简要来说，jj文件中的所有BNF范式描述了一种语法，即描述了一种语言，JavaCC根据所写的BNF范式生成识别这种语言的编译器。需要做的只是将定义某种语言的BNF范式写入jj文件，即可生成能识别相应语言的编译器。

下面将详细描述我的词法分析器和语法分析器构造。我的词法分析器分为两个主要部分，即SKIP 和 TOKEN，

**SKIP** :

**{**

" "

**|** "\r"

**|** "\t"

**|** "\n"

**|** **<**"//" **(~[**"\n","\r"**])\*** **(**"\n"**|**"\r"**)>**

**|** **<**"//" **(~[**"\n","\r"**])\*** **(**"\n" **|** "\r" **|** "\r\n"**)>**

**|** **<**"/\*" **(~[**"\*"**])\*** "\*" **(**"\*" **|** **~[**"\*","/"**]** **(~[**"\*"**])\*** "\*"**)\*** "/"**>**

**}**

SKIP里的内容也就是我们所生成的词法分析器里自动跳过的内容，内容为跳过回车符、制表符、换行符以及//和/\* \*/注释（这两个正则表达式是系统自动生成的）。

为了识别CMM关键字、标识符、运算符、界符等所有合法符号，我们需要在token当中对这些符号进行定义，根据2.2.2中符号码表进行相关定义。其中关键字部分Token定义如下：

//关键字

**TOKEN** :

**{**

**<**BOOL : "bool" **>**

**|** **<**CHAR : "char" **>**

**|** **<**INT: "int"**>**

**|** **<**FLOAT: "float"**>**

**|** **<**IF: "if"**>**

**|** **<**ELSE: "else"**>**

**|** **<**WHILE: "while"**>**

**|** **<**FOR: "for"**>**

**|** **<**BREAK: "break"**>**

**|** **<**CONTINUE: "continue"**>**

**|** **<**RETURN: "return"**>**

**|** **<**TRUE: "true"**>**

**|** **<**FALSE: "false"**>**

**}**

至于其他的例如符号、标识符等定义大致类似，不在此列出

JavaCC把文法识别完全地做到了函数过程中，而且javaCC使用递归子程序法来构造语法分析程序，但相对前面自己动手构造的语法分析程序，在javaCC中只需要利用正则表达式来描述下级的递归函数以及各种组合和可能，就能自动生成功能相同的语法分析程序。对应的递归调用关系与前面的语法语义分析类似，但函数内部更加简单，以下列举生成识别表达式（包括逻辑表达式和算数表达式）、循环语句、条件语句的函数。

逻辑表达式

**void** logical\_expression():**{}{**

logical\_factor1()**(**"||"logical\_expression()**)?**

**}**

**void** logical\_factor1():**{}{**

logical\_factor2()**(**"&&"logical\_factor1()**)?**

**}**

**void** logical\_factor2():**{}{**

**(**"true"**)**

**|(**"false"**)**

**|(**arith\_expression()logical\_symbol()arith\_expression()**)**

**|(**"("logical\_expression()")"**)**

**}**

for循环语句

**void** forloop():**{}{**

"for"forblock()

**}**

**void** forblock():**{}{**

**(**"("assignment()";"factor1()**)(**co\_function()**|**astament()**)**

**}**

条件语句

**void** if\_statement():**{}{**

"if"ifblock()

**}**

**void** ifblock():**{}{**

**(**"("logical\_expression()")"**)(**co\_function()**|**astament()**)(**else\_statement()**)?**

**}**

**void** else\_statement():**{}{**

"else"**(**co\_function()**|**astament()**)**

**}**

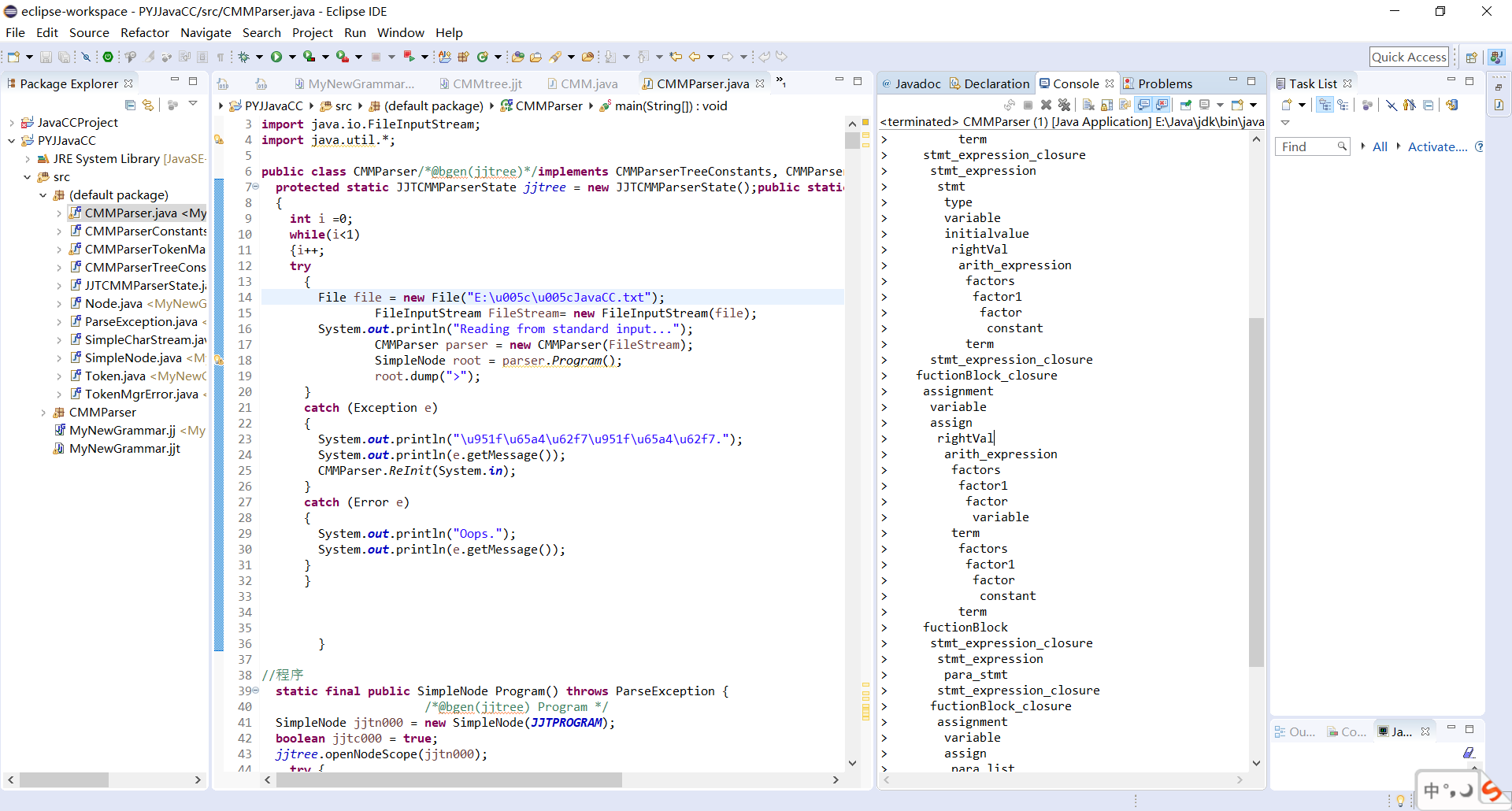
可以看到各函数的文法规则的具体实现相对3.3.3中的函数说明，简洁明了太多。

**5.3 运行与测试**

**5．3．1** **测试用例**

本实验模块的测试用例使用语法分析阶段的测试用例，见3.4.1

**5．3．2** **运行结果**

****

**结论**

在进行本次实验之前，上学期经过了一整个学期100多个学时编译原理理论课学习，虽然也经过了考试，分数也算不坏，但说实话只是刻板生硬的知识记忆，不仅容易忘记，理解也并不深刻，而这也直接导致了刚刚开始实验时的摸不着头脑，不知从何下手，但正所谓实践出真知，更何况之前已经有过不少理论积累，在经过了一番尝试与网上博客的实验指导之后，一上手就渐渐越做越懂，但难题也越做越多，当然也有许多之前理论学习时不曾出现或并不在意的细节给实验带来了极大阻碍，例如词法分析时忽视了换行回车空格符等、词法分析阶段负数实现的不现实、复杂大型文法的构造（例如左递归的消除在少而简单的文法时比较容易，但文法复杂后容易出现使得构造后文法可读性差，直接导致实现递归子程序法时出现错漏而不自知）、语义分析时break与continue的实现（虽然在理论以及语法树层面看起来相当简单，但落实到复杂，调用关系纠缠的递归子程序算法对应的代码段中时，需要厘清函数的一级级调用以及各种相关情况）等等。总之，经过这次实验，我才算对语法语义有了一个更深的认识，而不仅仅停留在想象中，不仅如此，在这次实验过程中，我慢慢养成了模块化开发的软件开发理念，这都是bug逼出来的，随着程序结构的复杂（特别是此实验中设计大量的递归调用，任何细微的问题都会被放到最大），从前的那种一次编写，慢慢修改的写代码方式给我带来了极大的麻烦，有时候一个bug需要改上大半天，而且也并不一定能一次改好，可能带来其他的问题，经过几次痛苦的修改后，几乎与我最初写的代码面目全非，本以为是提高效率的方式，但因为自身代码能力不够，难以一次性写出高质量的代码，反而造成效率低下且代码繁杂的问题，痛定思痛，我渐渐开始一个模块一个模块的开发调试，很快我发现这样虽然看上去慢，但是模块出错率低，最后集成后bug极少。总的来说，这次实验真可谓对我意义深重，帮助良多。

教师评语评分

评语：

评分：

评阅人：

年 月

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）