**武汉大学计算机学院**

**本科生实验报告**

**解释器构造实验**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：解释器构造实践

指 导 教 师 ：刘纪平

学 生 学 号 ：2017302580256

学 生 姓 名 ：黄煜坤

二○一九年十二月

**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： 黄煜坤 日期： 2019.12.12

摘 要

本实验的实验目的是设计并编制调试一个分析单词的词法分析器，加深对词法分析原理的理解；设计并编制调试一个分析CMM程序结构的语法分析器，加深对语法分析原理的理解；设计并编制调试一个实现CMM程序结构的解释器，加深对编译技术的理解和综合应用；为CMM语言编写JavaCC的模板文件，使用JavaCC自动生成CMM语言的词法和语法分析程序。

**关键词：**解释器;词法分析;语法分析;语义分析;JavaCC

**目 录**

# 1 词法分析

## 1.1 形式化表述

### 1.1.1 关键词

这类单词在本程序设计语言当中有固定的意义，又称为保留字 例如：if，while，return，def，else，while，for，break，continue，print，scan，int，real，char等

### 1.1.2 标识符

它的作用是为某个实体起一个名字。

### 1.1.3 字面量

是指直接以其字面值所表示的常量，可以是字符串、字符，也可以是个数值

### 1.1.4 特殊符号

类似于自然语言中的标点符号，每个符号在程序设计语言中都有特殊的用途。例如：+，-，\*，/，//，/\*，\*/，(，)，<，>等。

### 1.1.5 种别码表

表1.1 单词种别码表

|  |  |
| --- | --- |
| 单词符号 | 种别码 |
| if |  |
| else |  |
| while |  |
| for |  |
| break |  |
| continue |  |
| print |  |
| scan |  |
| + |  |
| - |  |
| \* |  |
| / |  |
| // |  |
| /\* |  |
| \*/ |  |
| \t |  |
| \n |  |
| \r |  |
| ( |  |
| ) |  |
| < |  |
| > |  |
| = |  |
| == |  |
| <> |  |
| <= |  |
| >= |  |
| ; |  |
| ‘ |  |
| “ |  |
| int |  |
| real |  |
| char |  |
| [ |  |
| ] |  |
| { |  |
| } |  |
| && |  |
| || |  |
| , |  |
| identifier |  |
| inumber |  |
| fnumber |  |
| def |  |
| return |  |

## 1.2 程序结构

### 1.2.1 程序总体结构

词法分析程序位于CMM.LA包中，根据功能，分为FileExecutor.java, JudgeTools.java 以及 LAnalyzer.java三个文件。其中，FileExecutor主要负责代码文件的读取并对于读取得到的字符串进行简单的处理，例如替换换行退格等间隔符，同时，对于单词进行切分。JudgeTools负责对于单词类别的判断识别，而LAnalyzer综合以上两个类中的方法，完成单词种类识别的过程，并将识别的单词与对应种别保存在数组中。IDE中，文件结构如图1.1所示。

此外，定义相应的数据结构Word.java，由于该类型会出现在解释器运行的整个流程中，因此将其存放在model包中。

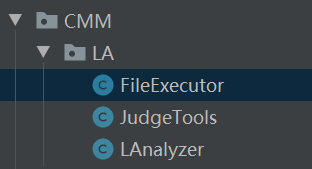


图1.1 词法分析程序总体结构

### 1.2.2 数据结构

对于识别的单词，定义Word类型存放该单词的字符串、类别、出现在代码中的行数与位置，结构如图1.2所示。将属性设为public以便供词法、语法、语义分析使用。

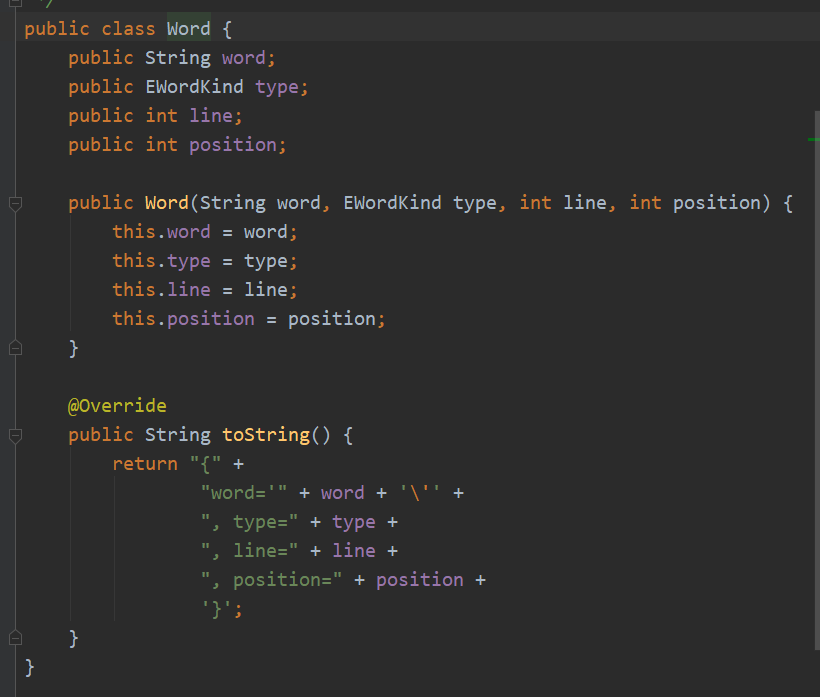
****

图1.2 Word类型结构

### 1.2.3 算法说明

#### 1.2.3.1 单词获取算法

本实验的单词分割算法参考python语言的语言特性，逐行进行读取，因此逐行读取代码文件，并对于每行代码进行单词切割。切割过程采用状态自动机思想，将待识别字符看作一个个的识别状态，识别过程中，根据读取的字符进行状态转移，由于本实验中，绝大多数待识别单词的前缀区别只有1个字符，部分单词需要在识别切割字符后再移入1个字符进行识别，因此该问题可退化为利用设定好的切割字符对于字符串进行单词分割的问题。

首先定义切割字符，认为空格、+、-、\*、/等非数字非字母字符为切割字符，当识别到切割字符时，将前面移入的所有字符规约为一个单词，并针对切割字符类型，尝试判断后续字符能否与切割字符再次规约。切割字符的设定如图1.3.



图1.3 切割字符的设置

在单词分割时，同时还需要处理字符串与字符识别的问题。对于字符串识别，初始化时，设置字符串开关为假，当读取到一个字符串，将字符串开关取反。识别过程中，当字符串开关为真时，归纳移入字符为字符串中的字符。每行读取结束前，若字符串开关为真，则说明存在未闭合的字符串，此时抛出错误；对于字符识别，则当读取到“’”字符时，检查后面三位字符，若第二位字符为“’”或是三位字符为“\t’”、“\n’”、“\r’”，则将这几位字符识别为一个单词。以上所有过程对应代码如图1.4所示。

ArrayList<String> words = new ArrayList<>();  
StringBuffer all = new StringBuffer(s);  
StringBuffer word = new StringBuffer("");  
boolean isStr=false;  
for(int i=0;i<all.Length();i++){  
 char cur=all.charAt(i);  
 //字符串开关  
 if(cur=='"'){  
 isStr=!isStr;  
 if(!isStr)  
 word.append(cur);  
 //切割前部收入词组  
 if(!"".equals(word.toString())){  
 words.add(word.toString());  
 word = new StringBuffer("");  
 }  
 if(isStr)  
 word.append(cur);  
 continue;  
 }  
  
 if(isStr) {  
 //字符串开启  
 word.append(cur);  
 }else if(isOptsOrEnd(cur)){  
 //若当前字符为分割字符，切割单词  
 //切割前部收入词组  
 if(!"".equals(word.toString())){  
 words.add(word.toString());  
 word = new StringBuffer("");  
 }  
  
 //判断切割字符是否为关键字  
 if(isOptsOrEnd(cur)){  
 word.append(cur);  
 //将后一位字符加入，进行关键字判断  
 if(i+1<all.Length()){  
 word.append(all.charAt(i+1));  
 //关键字判断失败，回退一位，若成功，则将这两个看作一个单词收入词库  
 if(JudgeTools.isKeyWord(word.toString())== EWordKind.WRONG) {  
 word.deleteCharAt(1);  
 i--;  
 }  
 //成功，不回退  
 i++;  
 }  
 if(cur=='\''){  
 if(i+2<all.Length()&&all.charAt(i+2)=='\''){  
 word.append(all.charAt(i+1));  
 word.append(all.charAt(i+2));  
 i+=2;  
 }else if(i+3<all.Length()&&all.charAt(i+3)=='\''&&all.charAt(i+1)=='\\'&&  
 (all.charAt(i+2)=='r'||all.charAt(i+2)=='t'||all.charAt(i+2)=='n')){  
 word.append(all.charAt(i+1));  
 word.append(all.charAt(i+2));  
 word.append(all.charAt(i+3));  
 i+=3;  
 }  
 }  
 words.add(word.toString());  
 word=new StringBuffer("");  
 }  
 }else {  
 word.append(cur);  
 }  
}  
//尾部单词  
if(!"".equals(word.toString())){  
 words.add(word.toString());  
}  
if(isStr) {  
 ErrorExecutor.showError("LA","识别到未闭合的字符串结构",line+1,0);  
 words.clear();  
}

#### 1.2.3.2 单词识别

在识别单词时，对于单词按照关键词--数字--变量名--字符串顺序判断，若都不满足，则进行标记为错。

关键词识别时，逐个检查单词字符，若存在非法字符或是开头结尾不符合要求则报错。如下所示。

public static EWordKind isIdentifier(String word) {

for (int i = 0; i < word.Length(); i++) {

char c = word.charAt(i);

if (i == 0 &&(c < 'a' || c > 'z') && (c < 'A' || c > 'Z')) return EWordKind.WRONG;

if (!((c <= '9' && c >= '0') || (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || c == '\_'))

return EWordKind.WRONG;

if(i==word.Length()-1&&c=='\_')

return EWordKind.WRONG;

}

return EWordKind.IDENTIFIER;

}

数字识别时，采用正则表达式方法，对于整数数字、实数数字进行检查判断。代码如下所示。

public static EWordKind isFNumeric(String str) {

Pattern pattern = Pattern.compile("[0-9]+\\.[0-9]+");

String bigStr;

try {

bigStr = new BigDecimal(str).toString();

} catch (Exception e) {

return EWordKind.WRONG;//异常 说明包含非数字。

}

Matcher isNum = pattern.matcher(bigStr); // matcher是全匹配

if (!isNum.matches()) {

return EWordKind.WRONG;

}

return EWordKind.FNUMBER;

}

public static EWordKind isINumeric(String str) {

Pattern pattern = Pattern.compile("[0-9]+");

String bigStr;

try {

bigStr = new BigDecimal(str).toString();

} catch (Exception e) {

return EWordKind.WRONG;//异常 说明包含非数字。

}

Matcher isNum = pattern.matcher(bigStr); // matcher是全匹配

if (!isNum.matches()) {

return EWordKind.WRONG;

}

return EWordKind.INUMBER;

}

关键词识别则是将识别单词与种码表中的关键词进行字符串匹配，若满足要求，则返回对应类型。

字符串识别则是检查字符串头尾是否为双引号，并将字符串中的转义字符进行替换，例如“\t”替换为’\t’。代码如下所示。

word = word.replaceAll("\\\\t","\t");

word = word.replaceAll("\\\\r","\r");

word = word.replaceAll("\\\\n","\n");

若以上类型都不满足，则将该单词识别为WRONG类型，否则，创建Word类型并将该实例存入数组中。

在单词识别时，同时还要处理多行注释、单行注释的问题。对于多行注释，我将其看作为读取开关，当识别到左多行注释时，将开关数加一，当识别到右多行注释时，将开关数减一。读取时，检查开关是否大于0，若大于0，则跳过该单词的识别，同时检查开关大小是否小于0，若小于0，则说明存在未闭合右多行注释，此时抛出错误。处理单行注释时，若识别单词为单行注释，则跳过该行单词的检测。代码如图1.4所示。

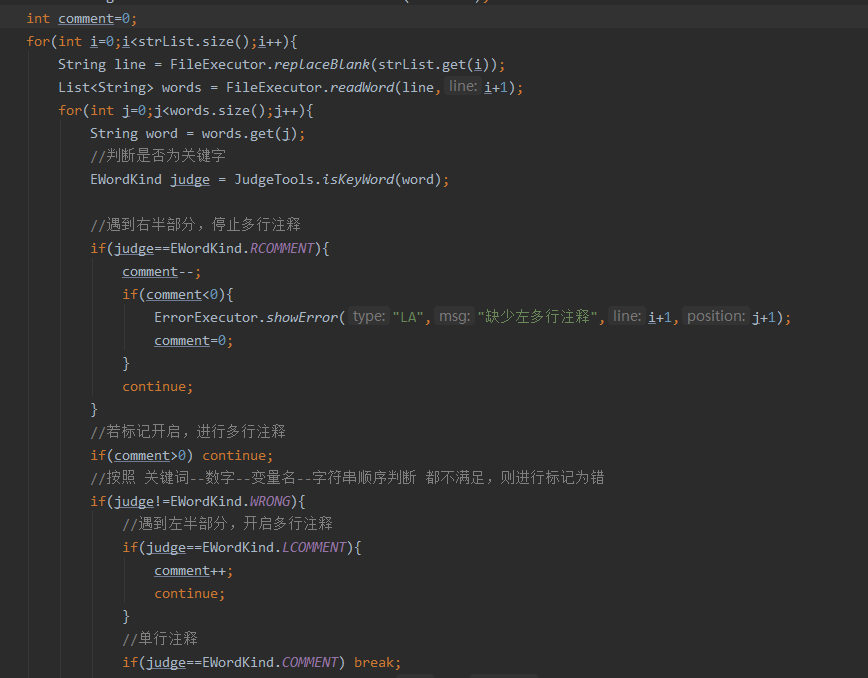


图1.4 多行注释及单行注释处理

## 1.3 测试用例及结果说明

### 1.3.1 注释测试

**测试样例：**

int

// undefined operation for half comment. So you can treat the remaining as comment or throw error for "'\*/' not found"

/\* the error line /\*

int x;

\*/

print(x);

**识别结果：**

{word='int', type=INT, line=1, position=1}

**说明：**

成功跳过了单行注释部分，对于未闭合的多行注释，跳过后续所有的未闭合部分

### 1.3.2 数字测试

**测试样例：**

// PART TWO: basic number detection

// right form

12.6

0.3

0123456789

-20.000

-02346

+23476

+0234

0239.120

// error form

0.0.2.2.5

23.-3

.34 // may regard as right form ~

0x123 // may regard as right form ~

123..

4..3245

234e-3 // may regard as right from ~

23. // may regard as right form ~

234\_234 // may regard as right form in JAVA

**识别结果：**

{word='12.6', type=FNUMBER, line=3, position=1}

{word='0.3', type=FNUMBER, line=4, position=1}

{word='0123456789', type=INUMBER, line=5, position=1}

{word='-', type=MINUS, line=6, position=1}

{word='20.000', type=FNUMBER, line=6, position=2}

{word='-', type=MINUS, line=7, position=1}

{word='02346', type=INUMBER, line=7, position=2}

{word='+', type=PLUS, line=8, position=1}

{word='23476', type=INUMBER, line=8, position=2}

{word='+', type=PLUS, line=9, position=1}

{word='0234', type=INUMBER, line=9, position=2}

{word='0239.120', type=FNUMBER, line=10, position=1}

{word='0.0.2.2.5', type=WRONG, line=12, position=1}

{word='23.', type=INUMBER, line=13, position=1}

{word='-', type=MINUS, line=13, position=2}

{word='3', type=INUMBER, line=13, position=3}

{word='.34', type=FNUMBER, line=14, position=1}

{word='0x123', type=WRONG, line=15, position=1}

{word='123..', type=WRONG, line=16, position=1}

{word='4..3245', type=WRONG, line=17, position=1}

{word='234e', type=WRONG, line=18, position=1}

{word='23.', type=INUMBER, line=19, position=1}

{word='234\_234', type=WRONG, line=20, position=1}

**说明：**

实数数字可省略小数点前或者小数点后的数字部分，对于数字中出现的正负号，识别为加减号，对于正负号的处理将放置于语法处理中；对于科学表达式、十六进制以及java数字表示不支持。

### 1.3.3 基本符号测试

**测试样例：**

// PART THREE: basic sign detection

// right form

;

<

>

==

<>

>= // expected to implement

<= // expected to implement

&& // expected to implement

|| // expected to implement

(

}

)

[

}

]

=

' // expected to implement

" // expected to implement

\n // expected to implement

\r // expected to implement

\t // expected to implement

/

+

-

\*

// wrong form

@

#

$

^

\*\* // may regard as right form in python

-- // may regard as right form

\

! // may regard as right form in Java

, // may regard as right form

**识别结果：**

LA错误：识别到未闭合的字符串结构，错误发生在第23行 第0处

{word=';', type=BRANCH, line=3, position=1}

{word='<', type=SMALLER, line=4, position=1}

{word='>', type=BIGGER, line=5, position=1}

{word='==', type=EQUAL, line=6, position=1}

{word='<>', type=NEQUAL, line=7, position=1}

{word='>=', type=BIGGERE, line=8, position=1}

{word='<=', type=SMALLERE, line=9, position=1}

{word='&&', type=AND, line=10, position=1}

{word='||', type=OR, line=11, position=1}

{word='(', type=LBRACKET, line=14, position=1}

{word='}', type=RBRACES, line=15, position=1}

{word=')', type=RBRACKET, line=16, position=1}

{word='[', type=LARRAY, line=17, position=1}

{word='}', type=RBRACES, line=18, position=1}

{word=']', type=RARRAY, line=19, position=1}

{word='=', type=ASSIGN, line=20, position=1}

{word=''', type=SQUOTE, line=21, position=1}

{word='\n', type=NEWLINE, line=23, position=1}

{word='\r', type=ENTER, line=24, position=1}

{word='\t', type=TAB, line=25, position=1}

{word='/', type=DIVIDE, line=27, position=1}

{word='+', type=PLUS, line=28, position=1}

{word='-', type=MINUS, line=29, position=1}

{word='\*', type=ASTERISK, line=30, position=1}

{word='@', type=WRONG, line=32, position=1}

{word='#', type=WRONG, line=33, position=1}

{word='$', type=WRONG, line=34, position=1}

{word='^', type=WRONG, line=35, position=1}

{word='\*', type=ASTERISK, line=36, position=1}

{word='\*', type=ASTERISK, line=36, position=2}

{word='-', type=MINUS, line=37, position=1}

{word='-', type=MINUS, line=37, position=2}

{word='\', type=WRONG, line=38, position=1}

{word='!', type=WRONG, line=39, position=1}

{word=',', type=COMMA, line=40, position=1}

**说明：**

由于该部分第22行出现双引号后，未闭合，因此报错为：未闭合的字符串结构。此外由于对于正负号的处理模仿Python语言，因此，此处并不支持自增或者自减。

### 1.3.4 标识符识别

**测试样例：**

// PART FOUR: basic identifier dectection

// right form

asdf

1DSF3

DSF

SD2DF23as23s

a\_\_\_\_\_\_\_\_3

s0\_234\_4\_23ss

zxcvbnmlkjhggfdsapoiuytrewqqwertyuiopasdfghjklmnbvcxzzxcvbnmasdfghjklqwertyuiop // may be regard as wrong form due to the Length limit(64) of variable name.

no2nOFN23

// wrong form

\_w234 // may regard as right form

\_234

gfdff\_

23asdfc

sdf23\_\_\_\_\_

3jk3n23kl3n4nl6

**识别结果：**

{word='asdf', type=IDENTIFIER, line=3, position=1}

{word='1DSF3', type=WRONG, line=4, position=1}

{word='DSF', type=IDENTIFIER, line=5, position=1}

{word='SD2DF23as23s', type=IDENTIFIER, line=6, position=1}

{word='a\_\_\_\_\_\_\_\_3', type=IDENTIFIER, line=7, position=1}

{word='s0\_234\_4\_23ss', type=IDENTIFIER, line=8, position=1}

{word='zxcvbnmlkjhggfdsapoiuytrewqqwertyuiopasdfghjklmnbvcxzzxcvbnmasdfghjklqwertyuiop', type=WRONG, line=9, position=1}

{word='no2nOFN23', type=IDENTIFIER, line=10, position=1}

{word='\_w234', type=WRONG, line=12, position=1}

{word='\_234', type=WRONG, line=13, position=1}

{word='gfdff\_', type=WRONG, line=14, position=1}

{word='23asdfc', type=WRONG, line=15, position=1}

{word='sdf23\_\_\_\_\_', type=WRONG, line=16, position=1}

{word='3jk3n23kl3n4nl6', type=WRONG, line=17, position=1}

**说明：**

正确识别合法标识符，并对于过长的标识符（例如第九行）判断为非法标识符。

### 1.3.5 基本关键字测试

**测试样例：**

// PART FIVE: basic keyword detection

// type

int

real // depend on yourself

char // expect to implement

// structure

while

for

break // expect to implement

continue // expect to implement

if

else

// input,output is REQUIRED in PPT

print // depend on yourself

scan // depend on yourself

// you may treat the below as reserved word or identifier

define // may regard as right form

goto // may regard as right form

switch // may regard as right form

include // may regard as right form

typedef // may regard as right form

**识别结果：**

{word='int', type=INT, line=3, position=1}

{word='real', type=REAL, line=4, position=1}

{word='char', type=CHAR, line=5, position=1}

{word='while', type=WHILE, line=8, position=1}

{word='for', type=FOR, line=9, position=1}

{word='break', type=BREAK, line=10, position=1}

{word='continue', type=CONTINUE, line=11, position=1}

{word='if', type=IF, line=12, position=1}

{word='else', type=ELSE, line=13, position=1}

{word='print', type=PRINT, line=16, position=1}

{word='scan', type=SCAN, line=17, position=1}

{word='define', type=IDENTIFIER, line=20, position=1}

{word='goto', type=IDENTIFIER, line=21, position=1}

{word='switch', type=IDENTIFIER, line=22, position=1}

{word='include', type=IDENTIFIER, line=23, position=1}

{word='typedef', type=IDENTIFIER, line=24, position=1}

**说明：**

成功识别本解释器中设定的关键字，对于define、goto等非关键字，识别为标识符。

### 1.3.6 综合测试

**测试样例：**

// PART SIX: comprehensive analysis

// here are some code adapted from C which is completely lexically legal.

// piece one

int a[3];

a = 234;

b = sdf\_234df

// piece two

{

int a = 3<>5;

real b=-23;

int a[3] = { 1,3,,b,nono }

else print (b);

if( a>b){ print(a );

}

{

236 \* 1+ 345/(23 \* 34) - 2346/ (31- 31) =xx

}

}

// piece three

{

int a scan(a);real b =0.234;

{

{ print(a\_b\_c\_d\_e\_\_\_\_1) == 0126.120

while (a >+234){

b = b - 1 - a;

while( b )

{

real c = 123;

2343

= b - 23;

b =

- 14.1238675;

print(a)

}

}

}

**识别结果：**

{word='int', type=INT, line=4, position=1}

{word='a', type=IDENTIFIER, line=4, position=2}

{word='[', type=LARRAY, line=4, position=3}

{word='3', type=INUMBER, line=4, position=4}

{word=']', type=RARRAY, line=4, position=5}

{word=';', type=BRANCH, line=4, position=6}

{word='a', type=IDENTIFIER, line=5, position=1}

{word='=', type=ASSIGN, line=5, position=2}

{word='234', type=INUMBER, line=5, position=3}

{word=';', type=BRANCH, line=5, position=4}

{word='b', type=IDENTIFIER, line=6, position=1}

{word='=', type=ASSIGN, line=6, position=2}

{word='sdf\_234df', type=IDENTIFIER, line=6, position=3}

{word='{', type=LBRACES, line=9, position=1}

{word='int', type=INT, line=10, position=1}

{word='a', type=IDENTIFIER, line=10, position=2}

{word='=', type=ASSIGN, line=10, position=3}

{word='3', type=INUMBER, line=10, position=4}

{word='<>', type=NEQUAL, line=10, position=5}

{word='5', type=INUMBER, line=10, position=6}

{word=';', type=BRANCH, line=10, position=7}

{word='real', type=REAL, line=11, position=1}

{word='b', type=IDENTIFIER, line=11, position=2}

{word='=', type=ASSIGN, line=11, position=3}

{word='-', type=MINUS, line=11, position=4}

{word='23', type=INUMBER, line=11, position=5}

{word=';', type=BRANCH, line=11, position=6}

{word='int', type=INT, line=12, position=1}

{word='a', type=IDENTIFIER, line=12, position=2}

{word='[', type=LARRAY, line=12, position=3}

{word='3', type=INUMBER, line=12, position=4}

{word=']', type=RARRAY, line=12, position=5}

{word='=', type=ASSIGN, line=12, position=6}

{word='{', type=LBRACES, line=12, position=7}

{word='1', type=INUMBER, line=12, position=8}

{word=',', type=COMMA, line=12, position=9}

{word='3', type=INUMBER, line=12, position=10}

{word=',', type=COMMA, line=12, position=11}

{word=',', type=COMMA, line=12, position=12}

{word='b', type=IDENTIFIER, line=12, position=13}

{word=',', type=COMMA, line=12, position=14}

{word='nono', type=IDENTIFIER, line=12, position=15}

{word='}', type=RBRACES, line=12, position=16}

{word='else', type=ELSE, line=13, position=1}

{word='print', type=PRINT, line=13, position=2}

{word='(', type=LBRACKET, line=13, position=3}

{word='b', type=IDENTIFIER, line=13, position=4}

{word=')', type=RBRACKET, line=13, position=5}

{word=';', type=BRANCH, line=13, position=6}

{word='if', type=IF, line=14, position=1}

{word='(', type=LBRACKET, line=14, position=2}

{word='a', type=IDENTIFIER, line=14, position=3}

{word='>', type=BIGGER, line=14, position=4}

{word='b', type=IDENTIFIER, line=14, position=5}

{word=')', type=RBRACKET, line=14, position=6}

{word='{', type=LBRACES, line=14, position=7}

{word='print', type=PRINT, line=14, position=8}

{word='(', type=LBRACKET, line=14, position=9}

{word='a', type=IDENTIFIER, line=14, position=10}

{word=')', type=RBRACKET, line=14, position=11}

{word=';', type=BRANCH, line=14, position=12}

{word='}', type=RBRACES, line=15, position=1}

{word='{', type=LBRACES, line=16, position=1}

{word='236', type=INUMBER, line=17, position=1}

{word='\*', type=ASTERISK, line=17, position=2}

{word='1', type=INUMBER, line=17, position=3}

{word='+', type=PLUS, line=17, position=4}

{word='345', type=INUMBER, line=17, position=5}

{word='/', type=DIVIDE, line=17, position=6}

{word='(', type=LBRACKET, line=17, position=7}

{word='23', type=INUMBER, line=17, position=8}

{word='\*', type=ASTERISK, line=17, position=9}

{word='34', type=INUMBER, line=17, position=10}

{word=')', type=RBRACKET, line=17, position=11}

{word='-', type=MINUS, line=17, position=12}

{word='2346', type=INUMBER, line=17, position=13}

{word='/', type=DIVIDE, line=17, position=14}

{word='(', type=LBRACKET, line=17, position=15}

{word='31', type=INUMBER, line=17, position=16}

{word='-', type=MINUS, line=17, position=17}

{word='31', type=INUMBER, line=17, position=18}

{word=')', type=RBRACKET, line=17, position=19}

{word='=', type=ASSIGN, line=17, position=20}

{word='xx', type=IDENTIFIER, line=17, position=21}

{word='}', type=RBRACES, line=18, position=1}

{word='}', type=RBRACES, line=19, position=1}

{word='{', type=LBRACES, line=22, position=1}

{word='int', type=INT, line=23, position=1}

{word='a', type=IDENTIFIER, line=23, position=2}

{word='scan', type=SCAN, line=23, position=3}

{word='(', type=LBRACKET, line=23, position=4}

{word='a', type=IDENTIFIER, line=23, position=5}

{word=')', type=RBRACKET, line=23, position=6}

{word=';', type=BRANCH, line=23, position=7}

{word='real', type=REAL, line=23, position=8}

{word='b', type=IDENTIFIER, line=23, position=9}

{word='=', type=ASSIGN, line=23, position=10}

{word='0.234', type=FNUMBER, line=23, position=11}

{word=';', type=BRANCH, line=23, position=12}

{word='{', type=LBRACES, line=24, position=1}

{word='{', type=LBRACES, line=25, position=1}

{word='print', type=PRINT, line=25, position=2}

{word='(', type=LBRACKET, line=25, position=3}

{word='a\_b\_c\_d\_e\_\_\_\_1', type=IDENTIFIER, line=25, position=4}

{word=')', type=RBRACKET, line=25, position=5}

{word='==', type=EQUAL, line=25, position=6}

{word='0126.120', type=FNUMBER, line=25, position=7}

{word='while', type=WHILE, line=26, position=1}

{word='(', type=LBRACKET, line=26, position=2}

{word='a', type=IDENTIFIER, line=26, position=3}

{word='>', type=BIGGER, line=26, position=4}

{word='+', type=PLUS, line=26, position=5}

{word='234', type=INUMBER, line=26, position=6}

{word=')', type=RBRACKET, line=26, position=7}

{word='{', type=LBRACES, line=26, position=8}

{word='b', type=IDENTIFIER, line=27, position=1}

{word='=', type=ASSIGN, line=27, position=2}

{word='b', type=IDENTIFIER, line=27, position=3}

{word='-', type=MINUS, line=27, position=4}

{word='1', type=INUMBER, line=27, position=5}

{word='-', type=MINUS, line=27, position=6}

{word='a', type=IDENTIFIER, line=27, position=7}

{word=';', type=BRANCH, line=27, position=8}

{word='while', type=WHILE, line=28, position=1}

{word='(', type=LBRACKET, line=28, position=2}

{word='b', type=IDENTIFIER, line=28, position=3}

{word=')', type=RBRACKET, line=28, position=4}

{word='{', type=LBRACES, line=29, position=1}

{word='real', type=REAL, line=30, position=1}

{word='c', type=IDENTIFIER, line=30, position=2}

{word='=', type=ASSIGN, line=30, position=3}

{word='123', type=INUMBER, line=30, position=4}

{word=';', type=BRANCH, line=30, position=5}

{word='2343', type=INUMBER, line=31, position=1}

{word='=', type=ASSIGN, line=32, position=1}

{word='b', type=IDENTIFIER, line=32, position=2}

{word='-', type=MINUS, line=32, position=3}

{word='23', type=INUMBER, line=32, position=4}

{word=';', type=BRANCH, line=32, position=5}

{word='b', type=IDENTIFIER, line=33, position=1}

{word='=', type=ASSIGN, line=33, position=2}

{word='-', type=MINUS, line=34, position=1}

{word='14.1238675', type=FNUMBER, line=34, position=2}

{word=';', type=BRANCH, line=34, position=3}

{word='print', type=PRINT, line=35, position=1}

{word='(', type=LBRACKET, line=35, position=2}

{word='a', type=IDENTIFIER, line=35, position=3}

{word=')', type=RBRACKET, line=35, position=4}

{word='}', type=RBRACES, line=36, position=1}

{word='}', type=RBRACES, line=37, position=1}

{word='}', type=RBRACES, line=38, position=1}

**说明：**

均识别正确

## 1.4 实验总结

### 1.4.1 问题与解决方案

#### 1.4.1.1 连续加减运算符

对于数字的正负号处理上，我参考了c++, java, python三种语言的做法，对于c++和java来说，正负号不允许连续并拢，这是为了处理自增或者自减运算，但是允许正负号交替出现，此时，运算中的正负号将进行乘法运算，根据运算结果取加或者取减。而在python中，正负号允许连续并拢或是交替出现，此时，运算中的正负号将进行乘法运算，根据运算结果取加或者取减，因此，python中也不存在自增或是自减操作符。由于本实验为解释器构造，因此在设计时，尽可能参考解释型语言，因此选择将所有正负号识别为加减号，并在语法分析与语义执行时，逐步计算加减号连续出现时的运算语义。

#### 1.4.1.2 字符串识别与转义字符

起初，字符串识别后，由于读取时java对于读取字符串自动进行转义处理，因此，对于字符串中包含的转义字符无法获取正确的语义。因此，在识别阶段，将单词判定为字符串后，使用replace方法将字符串中的转义字符替换为非转义字符。

### 1.4.2 实验的优缺点

词法实验的重点主要在于单词规约上，在这一步中，最容易想到的就是利用状态自动机，逐字符读取并根据字符进行状态的跳转与单词规约。但是，由于本次实验设计的词法之间具有较好的分割特性与较少的前缀重合部分，因此采用分割单词的方法，对于读入的语句进行切分。一方面，这种简化使得程序结构复杂度降低；另一方面，这种简化也带来了新的问题：当日后扩展功能时，若新增识别单词与现有词法中的多个单词存在较长重合前缀，则需要对目前的程序设计进行较大改动。在扩展性上的表现不如原始的状态自动机。

# 2 语法分析

## 2.1语法说明

### 2.1.1 文法说明

<program> ::= { <unit> }

<unit>::= <statement>| <def-func>

<statement> ::= <while-loop> | <if-else> | ( (<assignment> | <declaration> | <in> | <out> | <jump>|<return>|<call-func>)**;**)

<while-loop> ::= **while (** <condition> **)** <block>

<if-else> ::= **if (** <condition> **)**  <block> （**else**  <block>）

<block> ::= **{** { <statement> } **}**

<out> ::= **print(**<expression>|<string>**)**

<in> ::= **scan(**<variable >**)**

<assignment> ::= <variable> **=** <expression>

<declaration> ::= <type> <variable> <rest-variable>

<rest-variable> ::= **,**<variable>< rest-variable >|空

<variable> ::= <identifier> ( **[** <inumber> **]** )

<expression> ::= <calculation> | <character>

<calculation> ::= <item>((**+**|**-**)<calculation >) | <item>

<item> ::= (<factor> (**\***|**-**)<item>) | <factor>

<factor> ::= (<variable > | <number> | <call-func>) | (**(** <calculation> **)**)

<condition> ::= (<sub-con> (**&&** | **||**) <condition>) | <sub-con>

<sub-con> ::= <expression><relation key><expression>

<number> ::= (+|-)<number> | (<inumber> | <fnumber>)

<relation key> ::= **<** | **>** | **<=** | **>=** | **==** | **<>**

<type> ::= **int** | **real** | **char**

<jump> ::= **break** | **continue**

<string>::=“{<character> } “

<return>::= **return** (<expression>|<blank>)

<def-func>::=**def** <identifier> **(**<def-val-list>**)** <block>

<def-val-list>::={ (<declaration>**;**) }

<call-func>::=<identifier>**(** <val-list> **)**

<val-list>::=<variable> <rest-variable>|<blank>

### 2.1.2 语法分析过程

实验采用递归下降分析方法，具体实现方法概述如下：

1）每个非终结符对应一个解析函数；

2）产生式右侧为该产生式左侧非终结符所对应解析函数的“函数体”；

3）产生式右侧终结符对应从输入串中“消耗”该终结符的操作；

4）产生式中的‘|’对应函数体中的“if-else”语句；

5）产生式中的（）形成子产生式；

6）？符匹配子产生式1次或者0次；

7）对于扩展的BNF文法，产生式中的‘{}’对应函数体中的“while”语句；

8）产生式中粗体字符表示终结符。

## 2.2 程序结构

### 2.2.1 程序总体结构

语法分析程序位于CMM.SA包中，根据功能，分为SAnalyzer.java, SATreeNode.java 以及 SAException.java三个文件。其中，SAnalyzer包含了在语法分析中，所有非终结符对应的解析函数。SATreeNode定义了语法树节点。SAException 定义了语法分析语法异常，供解析过程中对于语法错误的处理。IDE中，文件结构如图2.1所示。

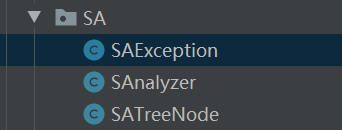


图2.1 语法分析程序总体结构

### 2.2.2 数据结构

定义的语法树节点需要包含节点对应的语法类型、该语法对应的单词以及它下面的子节点，对应的数据结构如图2.2所示。

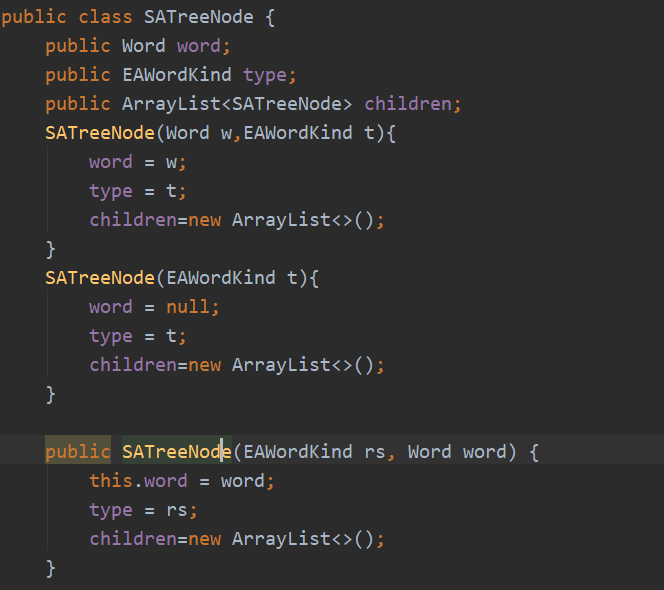
****

图2.2 SATreeNode类型数据结构

### 2.2.3 算法说明

语法分析针对于词法分析结束后获取的单词序列。对于不同类型的识别，存在不同的识别策略。由于大多数类型的识别过程相同，以下算法仅选取特别个例进行算法说明。

#### 2.2.3.1 非终结符识别算法

设计非终结符识别函数时，需要读取一个该非终结符识别的起始位置以及包含该非终结符的父节点，返回值为非终结符识别完成时的识别位置，若过程中出现不可忽略的识别错误，则返回起始位置。识别过程中，首先创建节点对应该非终结符。接着判断该起始位置是否超出单词识别范围，若超出范围，则退出该识别过程，并返回起始位置。之后根据文法结构对于该非终结符的子结构进行识别。若识别对象为非终结符，则传入此时的识别位置与该非终结符节点，获取非终结符识别结束后的识别位置，并判断该识别位置是否等于传入的识别位置，若相等，则说明识别失败，否则成功识别子结构中的非终结符，结束后判断新的识别位置是否超出识别范围；若识别终结符，则判断待识别单词的类型是否满足要求。完成识别后，将该非终结符节点加入父节点的子节点序列中，最后返回整个识别完成后，单词识别的位置。以下示例以非终结符“Unit”为例，如图2.3所示。

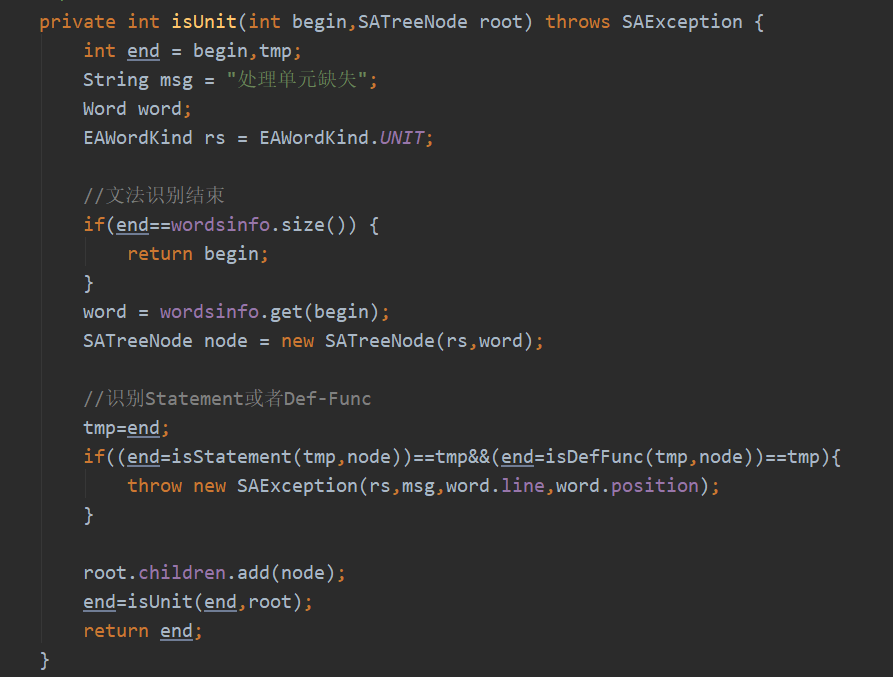


图2.3 非终结符“Unit”识别过程

识别Unit时，先判断文法识别是否结束，接着，创建Unit节点，并识别Unit下可能的子结构“Statement”或“Def-Func”，若两者均不能完成识别，则抛出异常，若识别完成，则将Unit节点加入父节点的子节点序列中。由于非终结符Program对于子节点Unit进行连续识别，则在加入父节点后，继续递归调用isUnit函数，对于非终结符Unit进行识别。当识别结束或不能继续满足识别要求时，返回此时的识别结束位置。

#### 2.2.3.2 终结符识别算法

识别终结符分为两种，一种为尝试识别，一种则是直接识别。如图2.4，2.5所示。

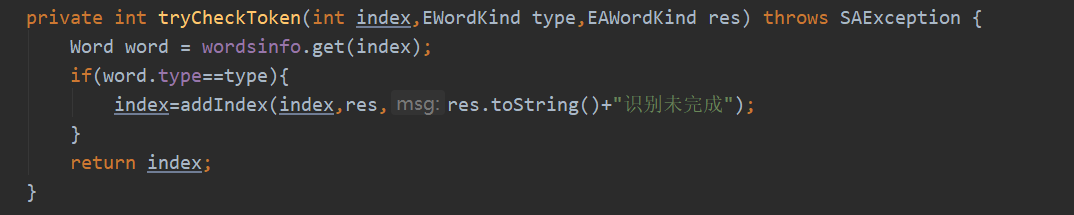


图2.4终结符尝试识别

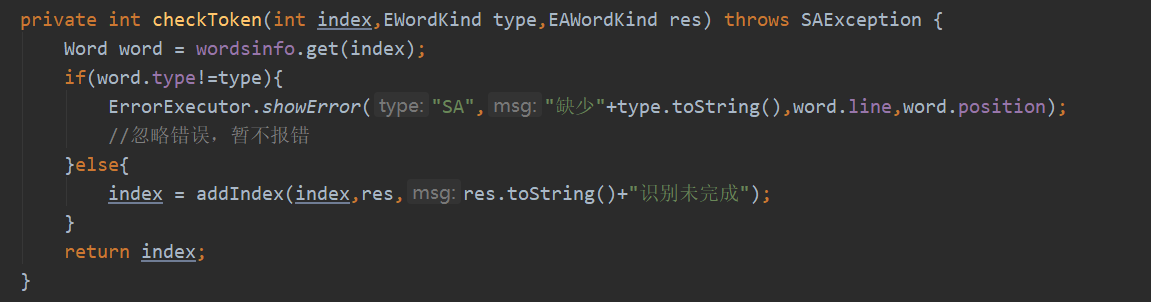


图2.5 终结符直接识别

尝试识别终结符时，需要传入识别的起始位置，待识别的非终结符类型以及该次识别对应的父非终结符结构。识别时，根据传入的识别起始位置，获取单词，判断单词的类型是否为待识别的终结符类型，若满足要求，则将识别位置加一。

直接识别终结符时，需要传入识别的起始位置，待识别的非终结符类型以及该次识别对应的父非终结符结构。识别时，根据传入的识别起始位置，获取单词，判断单词的类型是否为待识别的终结符类型，若满足要求，则将识别位置加一。若不满足，则输出错误信息。

这两种终结符的识别对应不同的识别需求，例如非终结符If-Else子结构，在识别完if部分后，需要尝试识别后续的else部分，因此使用tryCheckToken对终结符Else进行识别，不进行识别失败的错误报告。而对于if部分，则是使用checkToken进行直接识别，若不满足要求，则打印错误信息。

#### 2.2.3.3 错误处理

针对于语法识别中的错误，分为两种类型进行处理。对于一般的结构缺失，若不会对语义产生影响，则仅打印错误信息，对于语法继续进行识别，如图2.6所示。若识别错误发生在语义影响的部分，则抛出异常，中止识别，如图2.7所示。

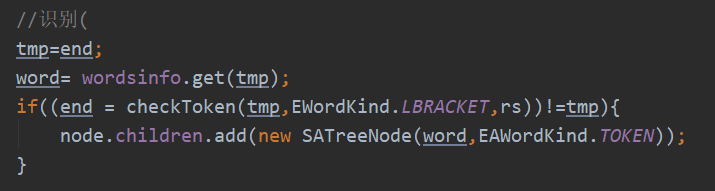


图2.6 处理一般的结构缺失

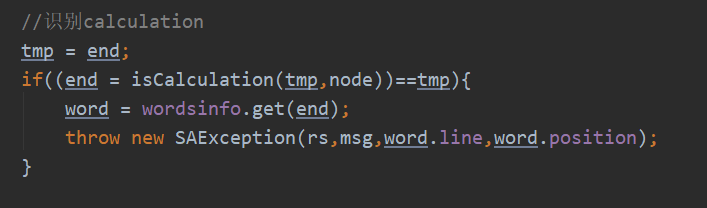


图2.7 处理影响语义的结构缺失

## 2.3 测试用例及结果说明

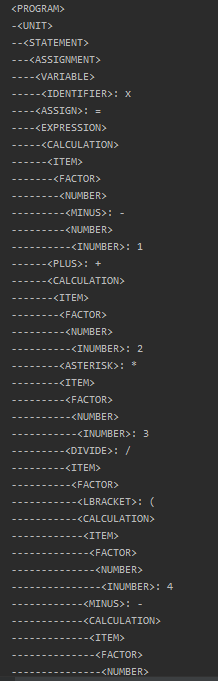
由于语法树过长无法完全展示，以下测试中，识别结果仅仅截取语法树的部分结构，语法分析下的所有测试将在语义执行测试时再次进行，以运行结果证明语法阶段的正确识别。

### 2.3.1 复杂算术表达式测试

**测试样例：**

x=-1+2\*3/(4-3)+(+3)-(-2)/(4 \* 4 + 2)/2.0;

**识别结果：**



**说明：**

成功完成赋值语句的识别，且该语句包含不同优先级运算嵌套。

### 2.3.2 逻辑运算测试

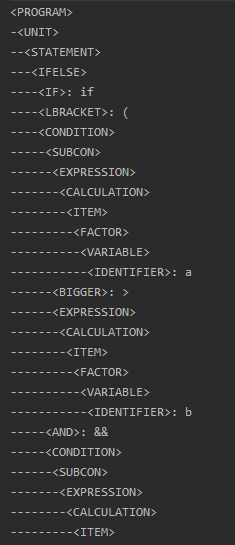
**测试样例：**

if (a>b&&a==2+3||b<>(5-a)){

print(a);

}

**识别结果：**



**说明：**

成功完成逻辑运算的识别。

### 2.3.3 结构语句嵌套

**测试样例：**

if (a>b&&a==2+3||b<>(5-a)){

while(a<b){

if(c<>a){

while(c>12&&c<>b){

print(c);

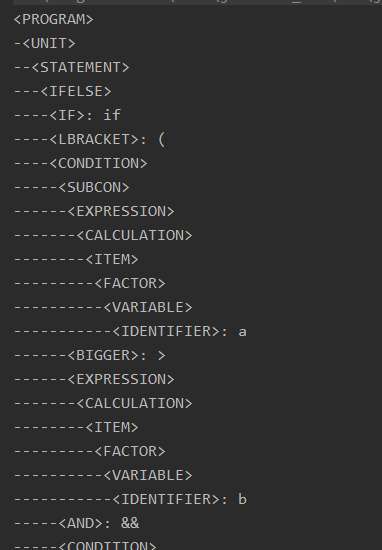
}

}

}

}

**识别结果：**



**说明：**

成功识别嵌套下的结构语句，正确生成语法树。

### 2.3.4 数组定义与使用

**测试样例：**

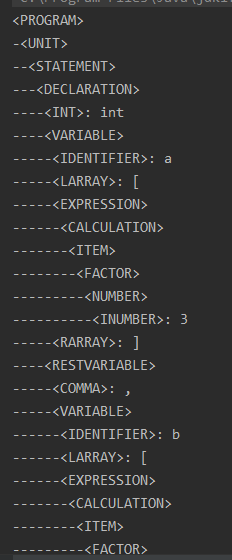
int a[3],b[4],c[3],d[-1],e[2.0];

real f;

char g;

a[0]=b[2]+c[0]+d[-1]+f;

**识别结果：**



**说明：**

正确识别数组的定义与使用，需要注意的是，我将数组索引的判断放置于语义识别部分，此处的数组名直接使用或是下标的越界、下标不符合类型不会引起错误。

### 2.3.5 函数定义与使用

**测试样例：**

def func(){

return;

}

def func(int a;){

}

def func(int a,b;char c;){

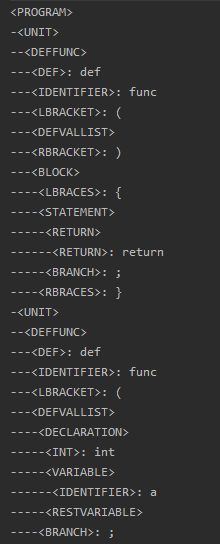
return a;

}

func(a,b,c,d);

func();

**识别结果：**



**说明：**

成功识别带参数以及不带参数的函数定义，并且成功识别带参数或不带参数的函数调用。

### 2.3.6 字符与字符串

**测试样例：**

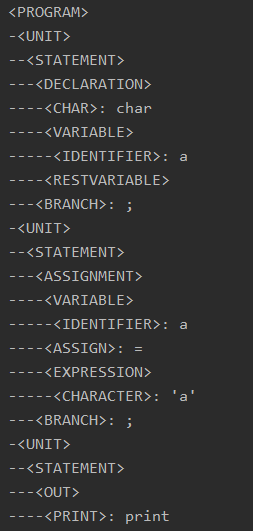
char a;

a = 'a';

print(a);

print("The character a is printed as 'a'.");

**识别结果：**



**说明：**

成功识别字符的赋值以及字符串的打印。

### 2.3.7非关键语义语法类型缺失错误

**测试样例：**

while(a<b{

print a-b)

if c<d {

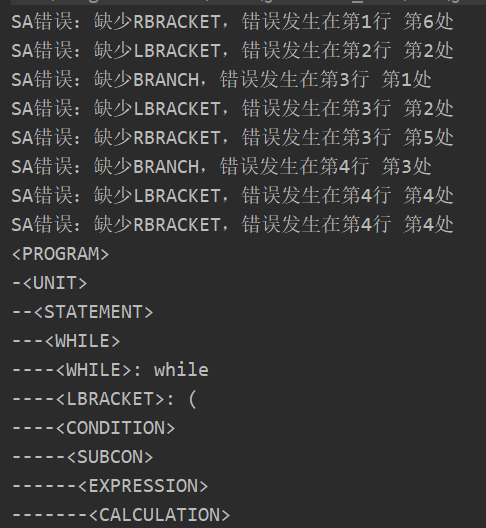
int c d;

c=12;

}

}

**识别结果：**



**说明：**

对于语法类型不影响语义的错误，例如这里的小括号、逗号分号等非关键语法，仅报出错误信息，但继续进行语法识别。

### 2.3.8 重要语义语法类型缺失错误

**测试样例：**

while(a<{

print a-b)

if c<d {

int c d;

c=12;

}

}

**识别结果：**



**说明：**

对于语法类型影响语义的错误，例如此处的条件表达式中的子条件语句缺失，由于缺失后，后续语法可能产生歧义，继续识别可能无法按照正常语法规则进行，因此在此处中断识别过程，报出错误。

## 2.4 实验总结

### 2.4.1 问题与解决方案

#### 2.4.1.1 语法与处理单元

完成语法分析实验前，思考最久的一个问题是，对于类似python的解释语言，最小的独立语法处理单元应该包含什么。起初，设计文法时，所有语法的最高规约为Statement，该语法的子语法包含while循环、if-else条件结构语句、赋值语句、遍历声明语句、输出语句、输入语句、跳转语句。但是在加入函数设计后，函数定义的语法应当与该处理单元并行，因此，在Statement之上，使用Unit进行规约，该单元为目前的最小处理单元。与现有的语言进行对比，发现这种设计与Python语言的语法类似。在执行过程中，从上到下逐行执行处理单元，既可以定义函数封装操作，也可以单独执行某一步操作。不需要定义函数入口，每一行或每一个函数定义即为可执行单元。

#### 2.4.1.2 错误处理

在识别语法过程中，最大的问题就是如何如何处理错误。首先需要明确，什么错误仅仅需要报出错误信息，而什么错误需要进行识别中断处理。对于语法识别错误，主要是以语法结构缺失为主。因此，我将缺失的结构分为两类，一类对于后续的语法语义没有太大影响，仅仅会影响此处的语法完整性；另一类，则是缺失后，可能会产生较大歧义，对于后续的语法识别也会产生影响。显然，分号、括号等语法对于语义的影响不大，因此归于第一类问题；其他语法结构含有较强的语义，因此归于第二类。当然，除了直接抛出错误外，也可以进行忽略某一段的语法，跳过错误语法部分继续进行识别，但是在本语法设计中，对于忽略界限的判定加强了对于后续某一结构的依赖，同样会带来新的结构错误，总体而言，这种抛出策略对于本文法的语法识别过程而言，带来的识别效果的提升并不大，因此并没有考虑。

### 2.4.2 实验的优缺点

使用LL文法，进行递归下降分析，这使得程序理解性更强，同时，在书写过程中，由于能够将每个语法分析策略转化为对应的函数，因此在错误恢复上，更加灵活。

但是相比于LR文法而言，LL文法支持的上下文无关文法数量更少，LL实现时，必须进行左因子提取并且消除左递归，在后续添加新的文法时，必须重新调整文法结构，在完成实验中，我反复修改语法分析部分以便适应于新的文法需求。

# 3 语义解释

## 3.1 解释方法说明

### 3.1.1 语义分析方法

语义分析方法使用语法制导翻译，翻译过程中，定义语法属性。其中包括：跳转类型(JumpType)、数值（Val）、变量列表（ValList）、变量长度（Length）、变量名数组（NameArray）、变量名（Name）、布尔类型（Bool）、函数入口点（EnterPoint）。在翻译过程中，定义全局变量Rax储存函数返回值、funcList储存函数定义、LocalScope储存局域变量表。在3.1.2小节中，按照语法规则以及对应翻译方法，给出语法制导翻译方案。

### 3.1.2 语法制导翻译方案

<statement> ::= <while-loop>

statement.returnTyep = while-loop.returnTyep

<statement> ::= <if-else>

statement.returnTyep = if-else.JumpType

<statement> ::= <return>**;**

statement.returnTyep ::= return.JumpType

<statement> ::= <jump>**;**

statement.returnTyep = jump.returnTyep

<statement> ::= <assignment>**;** | <declaration>**;** | <in>**;** | <out>**;**|<call-func>**;**

statement.returnTyep = **NOJUMP**

<while-loop> ::= **while (** <condition> **)** <block>

block.EnterPoint = loop;

loop:

if(condition.Bool!=**True**) goto out;

<block>

while-loop.JumpType = block.JumpType

if(while-loop.JumpType==**break**||while-loop.JumpType==**return**) goto out

goto loop;

out:;

<if-else> ::= **if (** <condition> **)**  <block1> {**else**  <block2>}

if(condition.Bool!=True) goto else;

<block1>

goto out;

else:

<block2>

out:;

<block> ::= **{** {<statement>} **}**

**for** i in Statements

block.JumpType = statement.return.Type

if(block.JumpType==**continue**||block.JumpType==**break**) return block.JumpType

<out> ::= **print(**<expression>|<string>**)**

out.Val = expression.Val | string.Val

<in> ::= **scan(**<variable >**)**

variable.Val = input.Val

<assignment> ::= <variable> **=** <expression>

variable.Val = expression.Val

<declaration> ::= <type> <variable> <rest-variable>

for i in variable.Length

variables.NameArray.add(variable.Name)

variables = variables.NameArray+rest-variable.NameArray

declaration.ValList = ValList(variables,type.type)

LocalScope.add(declaration.ValList)

<rest-variable1> ::= **,**<variable>< rest-variable2>|空

rest-variable1.NameArray.add(variable.Name)

rest-variable1.NameArray = rest-variable2.NameArray+ rest-variable1.NameArray

<variable> ::= <identifier> { **[** <inumber> **]** }

variable.Name=identifier.String

variable.Length = inumber.Val

<expression> ::= <calculation> | <character>

expression.Val = calculation.Val | character.Val

<calculation> ::= <item>((**+**|**-**)<calculation >) | <item>

calculation.Val = item.Val +|- calculation.Val | item.Val

<item> ::= (<factor> (**\***|**-**)<item>) | <factor>

item.Val = factor.Val \*|/ item.Val | factor.Val

<factor> ::= (<variable > | <number> | <call-func>) | **(** <calculation> **)**

factor.Val = variable.Val | number.Val | call-func.Val | calculation.Val

<condition> ::= (<sub-con> (**&&** | **||**) <condition>) | <sub-con>

condition.Bool = sub-con.Bool (**&&**|**||**) condition.Bool | sub-con.Bool

<sub-con> ::= <expression1><relation-key><expression2>

if(!(relation-key.operator(expression1.Val, expression2.Val))) goto False;

sub-con.Bool = **True**

goto out

False: sub-con.Bool = **False**

out:;

<number> ::= (+|-)<number> | (<inumber> | <fnumber>)

number.Val = (+|-)number.Val | inumber.Val | fnumber.Val

<relation-key> ::= **<** | **>** | **<=** | **>=** | **==** | **<>**

relation-key.operator = **<** | **>** | **<=** | **>=** | **==** | **<>**

<type> ::= **int** | **real | char**

type.type = **int** | **real | char**

<jump> ::= **break** | **continue**

jump.JumpType = **break** | **continue**

<return>::= **return** (<expression>|<blank>)

return.JumpType = **RETURN**

rax.Val = expression.Val | NULL

<def-func>::=**def** <identifier> **(**<def-val-list>**)** <block>

def-func.ValList = def-val-list.ValList

def-func.Name = identifier

def-func.EnterPoint = block.Node

funcList.add(def-func)

<def-val-list>::=(<declaration>**;**)\*|<blank>

for i in declarations

def-val-list.ValList = declaration.ValList + def-val-list.ValList

<call-func>::=<identifier>**(**<val-list>**)**

def-func = funcList.get(identifier)

def-func.ValList.Val = val-list.ValList.Val

goto def-func.EnterPoint

<val-list>::=<variable> <rest-variable>|<blank>

val-list.ValList.add(variable.Val)

val-list.ValList = val-list.ValList+rest-variable.ValList

### 3.1.3 中间代码设计及输出

使用上述定义的翻译方法，将执行过程中的实际语法属性代入语法对应的翻译语法进行输出。对于数组等复杂变量属性，则仅打印数组地址与属性类型。在程序执行过程中，将这些中间过程用ArrayList储存，在需要调试时可以打印。

### 3.1.4 优化方案

针对于上述设计的语法，进行以下优化：

对于带逻辑运算的条件表达式，进行短路处理，例如，条件表达式为逻辑求和运算时，从前往后计算子表达式布尔值，若不满足，则条件表达式对应布尔值判断为假，后续子表达式跳过计算。

## 3.2 程序结构

### 3.2.1 程序总体结构

语法分析程序位于CMM.SEA包中，根据功能，分为SEAnalyzer.java, FuncDef.java 以及 Val.java三个文件。其中，SEAnalyzer包含了在语义分析中，针对于语法树中不同语法结构的解释函数。FuncDef定义了函数定义数据结构。Val定义了数值属性值。IDE中，文件结构如图2.1所示。

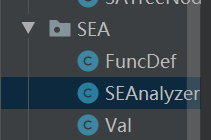


图3.1 语义分析程序总体结构

### 3.2.2 数据结构

定义的函数定义数据结构如图3.2所示。其中，valList保存函数签名中的变量表，name对应函数的名称，funcNode对应函数调用的入口节点。

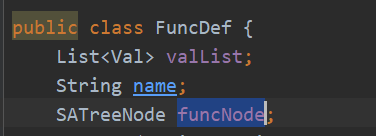


图3.2 FuncDef类型数据结构

定义的数值型属性数据结构如图3.3所示。若该数值型属性属于变量，则name对应该变量名称否则置为空，type对应该数值型属性的数据类型，例如int，real等，value对应数值型属性的值。

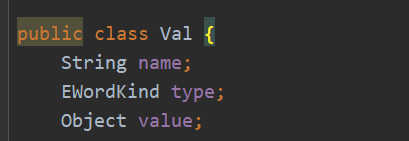


图3.3 Val类型数据结构

### 3.2.3 算法说明

一般的语法解释方法按照3.1中的解释方法进行，以下讨论的算法说明针对于解释方案中的未说明部分。

#### 3.2.3.1 函数定义

获取函数标识符、函数参数列表、以及函数入口节点。并将函数标识符与函数的参数列表与函数定义表中的函数签名进行匹配。若匹配结果为真，则说明存在重复的函数定义。函数定义代码如图3.4所示。

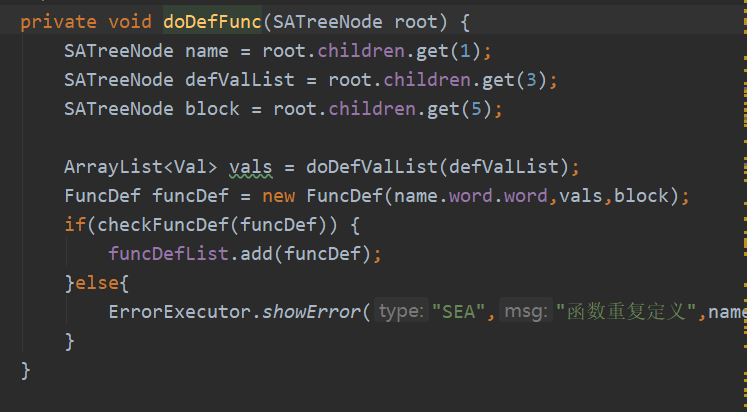


图3.4 函数定义语义解释代码

#### 3.2.3.2 函数调用

函数调用首先获取函数签名，将签名与函数定义表进行匹配，若匹配结果为假，则不存在对应的函数。匹配结果为真，则获取函数定义表中的函数定义，执行函数前，将函数作用域入栈，通过函数的入口节点，解释执行该函数的代码块，结束后，弹出该函数作用域。函数调用代码如图3.5所示。

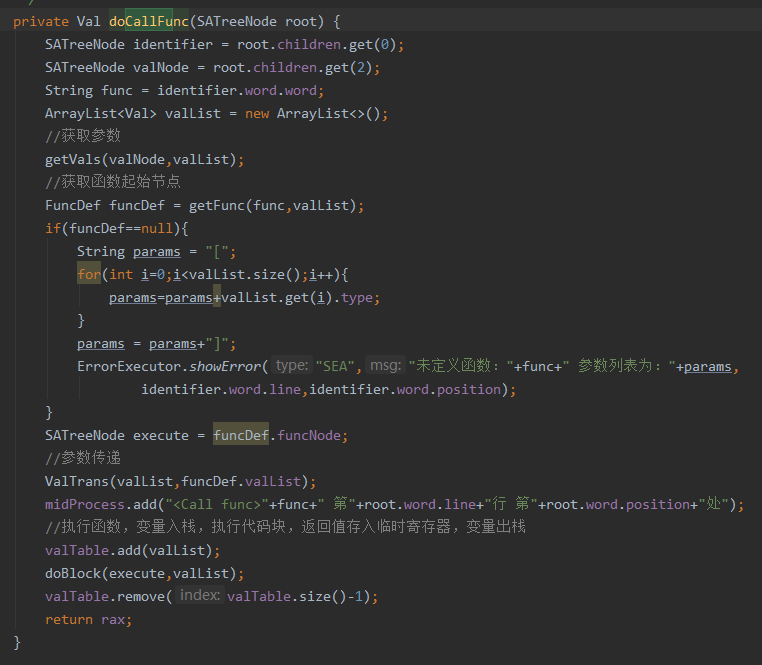


图3.5 函数调用语义解释代码

#### 3.2.3.3 变量查找

变量修改或变量获取都需要从变量表中，查找该变量的储存位置。设计符号表存放各个作用域的变量表，其中，在代码解释初，将全局作用域对应的变量表存入符号表，该域在符号表中下标为0。进行变量获取和变量修改时，根据变量名在符号表中最顶层作用域的变量表中查找，若不存在，则在全局域中查找。变量获取和变量修改代码如图3.6，图3.7所示。



图3.6 变量获取解释代码

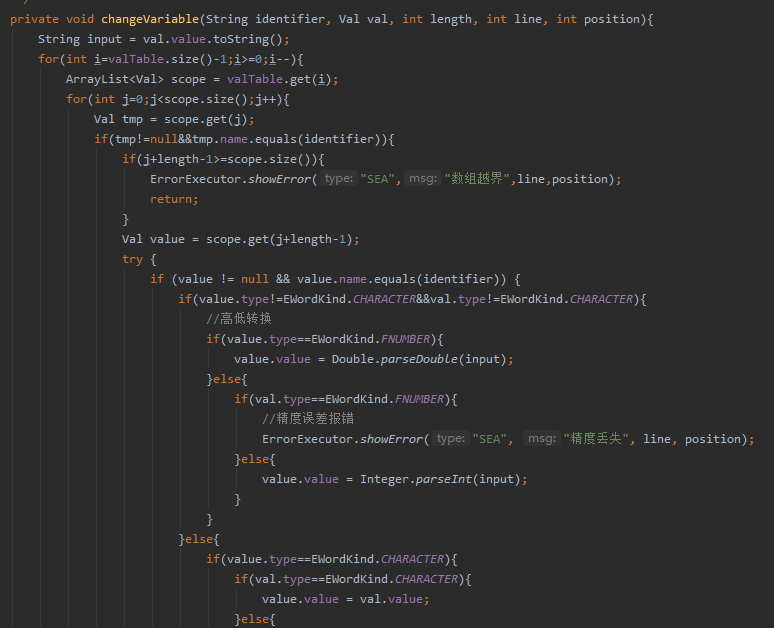


图3.7 变量修改解释代码

#### 3.2.3.4 类型转换

在进行计算、修改时，对于不同类型的变量进行类型转换。定义字符类型char与数值类型int、real。不允许字符类型与数值类型相互转换，不允许real向int类型转换。相关处理如图3.8。

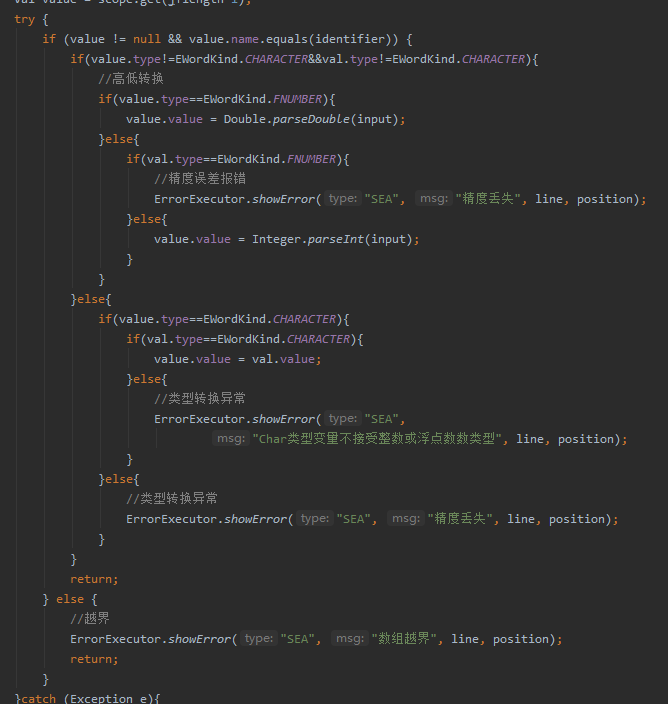


图3.8 数值类型转换代码

## 3.3 测试用例及结果说明

### 3.3.1 复杂算术表达式测试

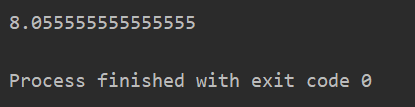
**测试样例：**

real x;

x=-1+2\*3/(4-3)+(+3)-((-2.0)/(4 \* 4 + 2))/2.0;

print(x);

**执行结果：**



**说明：**

成功完成计算且结果正确。

### 3.3.2 逻辑运算测试

**测试样例：**

int a,b;

a=3;

b=4;

if (a>b&&a==2+3||b<>(5-a)){

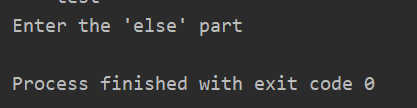
print("Enter the 'if' part");

}else{

print("Enter the 'else' part");

}

**执行结果：**



**说明：**

a>b为假，a==2+3为假，b<>（5-a）为真，运算时，先计算a>b，为假，则直接进入else部分。

### 3.3.3 结构语句嵌套

**测试样例：**

int a,b,c;

a=3;

b=4;

c=7;

if (a<b&&a==2+3||b<>(5-a)){

while(a<b){

if(c<>a){

while(c>3&&c<>b){

print(c);

c=c-1;

}

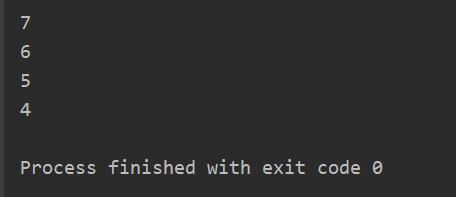
}

a=a+1;

}

}

**执行结果：**



**说明：**

a<b为真，a==2+3为假，b<>（5-a）为真，运算时，先计算a<b，为真，则继续计算子表达式，由于子表达式是或连接，则由于b<>(5-a)为真，子表达式为真，最终if条件为真，进入while循环，满足条件，则进入循环内容，由于 c<>a为真，进入if语句，继续内层循环，打印c的值，最终由于c==a=3跳出循环，进入外层循环后，a=a+1,a<b条件不成立，则循环结束。

### 3.3.4 数组定义与使用

**测试样例：**

int a[12],i,x;

real b[12];

x=1;

char str[100];

real sum;

while(i<10){

a[i] = i;

b[i] = (a[i]+1)\*i+3;

print(b[i]);

sum=sum+b[i];

print(sum);

if(sum>33&&b[i]<=33){

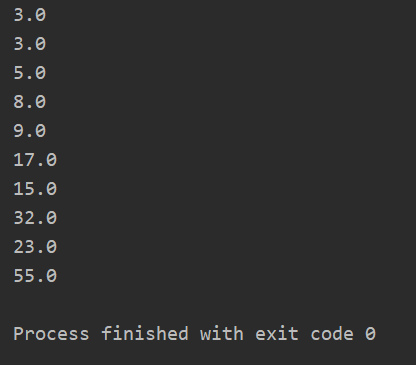
break;

}

i=i+1;

}

**执行结果：**

****

**说明：**

正确执行循环，打印数组中对应的元素

### 3.3.5 函数定义与使用

**测试样例：**

//函数

def abs(int x;){

if(x<0){

x=0-x;

}

return x;

}

//测试函数匹配

int x1;

real x2;

char x3;

x1 = -2;

x2 = -2.0;

x3 = 'a';

print(abs(x1));

//print(abs(x2));

//print(abs());

//print(abs(x3));

def abs(real x;){

if(x<0){

x=0-x;

}

return x;

}

print(abs(x2));

//测试函数定义域

char x;

x='a';

print("The x in outside is: ");

print(x);

def testField1(){

print(x);

return;

}

testField1();

def testField2(){

int x;

x=12;

print(x);

return;

}

print("The x in inside is:");

testField2();

//测试函数递归

int i,time;

i=0;

time=10;

def fibo(int cnt;){

if(cnt<=1){

return 1;

}

int a1,a2;

a1=cnt-1;

a2=cnt-2;

return fibo(a1)+fibo(a2);

}

print("fibo function output is as below:");

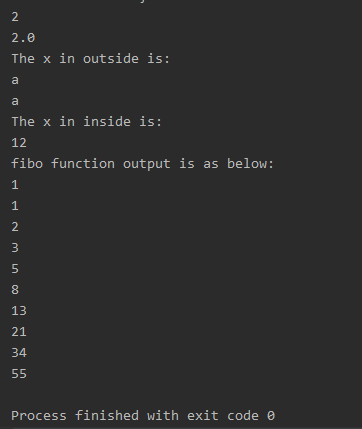
while(i<time){

print(fibo(i));

i=i+1;

}

**执行结果：**



**说明：**

成功执行函数定义，并能区分函数定义域与全局定义域，并且能够成功调用递归函数完成斐波那契数列的打印。

### 3.3.6 字符与字符串

**测试样例：**

print("This is a test for string.\n\tHowever, the string only support ASCII code.");

char a[12];

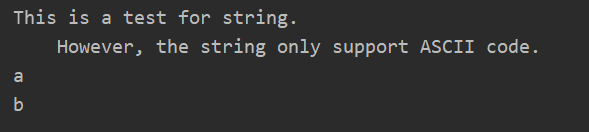
a[0]='a';

a[1]='b';

print(a[0]);

print(a[1]);

**执行结果：**

****

**说明：**

成功执行字符串打印与赋值，对于字符串中的转义字符也能进行识别、执行。

### 3.3.7 除数为0报错

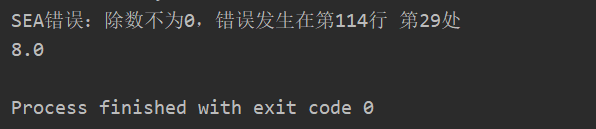
**测试样例：**

real x;

x=-1+(2\*3)/(4-3)+(+3)-((-2.0)/(4 \* 4 + 2-18))/2.0;

print(x);

**执行结果：**

****

**说明：**

成功报出表达式中(4 \* 4 + 2-18)部分为0的错误，报出错误后，不停止运行，忽略错误部分，将其余正确部分结果赋值，继续下面的运行。

### 3.3.8 变量重复定义

**测试样例：**

real x;

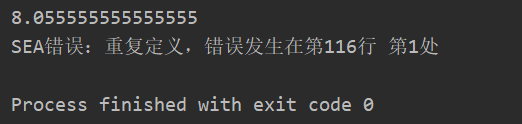
x=-1+(2\*3)/(4-3)+(+3)-((-2.0)/(4 \* 4 + 2))/2.0;

print(x);

real x;

x=0;

**执行结果：**

****

**说明：**

成功报出表达式中real x部分重复定义的错误，报出错误后，不停止运行，忽略错误部分，继续下面的运行。

### 3.3.9 函数重载

**测试样例：**

//函数

def abs(int x;){

if(x<0){

x=0-x;

}

return x;

}

//测试函数匹配

int x1;

real x2;

char x3;

x1 = -2;

x2 = -2.0;

x3 = 'a';

def abs(real x;){

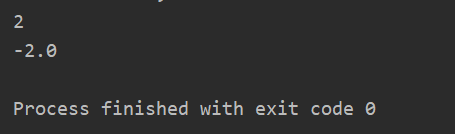
return x;

}

print(abs(x1));

print(abs(x2));

**执行结果：**

****

**说明：**

成功匹配不同函数签名的函数，第一个abs()返回传入值的绝对值，第二个abs()直接返回传入值。

### 3.3.10 函数定义域

**测试样例：**

//测试函数定义域

char x;

x='a';

print("The x in outside is: ");

print(x);

def testField1(){

print(x);

return;

}

testField1();

def testField2(){

int x;

x=12;

print(x);

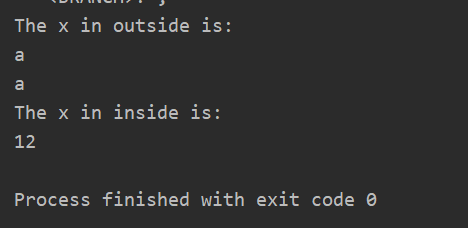
return;

}

print("The x in inside is:");

testField2();

**执行结果：**

****

**说明：**

成功区分不同作用域内的同名变量，变量的查找顺序为本函数作用域->全局作用域。

### 3.3.11 中间过程输出

**测试样例：**

char x;

x='a';

print("The x in outside is: ");

print(x);

def testField1(){

print(x);

return;

}

testField1();

def testField2(){

int x;

x=12;

print(x);

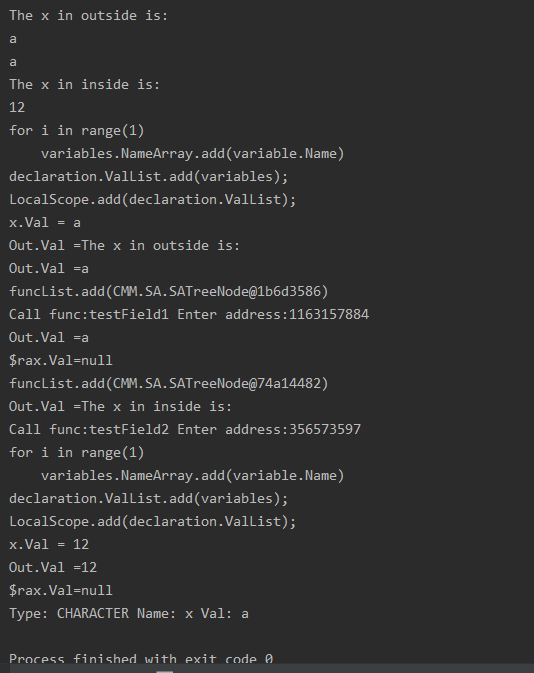
return;

}

print("The x in inside is:");

testField2();

**执行结果：**

****

**说明：**

在程序结束后，打印输出执行时的中间过程，该中间过程有助于后续进行调试以及掌握程序执行的流程。

### 3.3.12 符号表

**测试样例：**

char x;

x='a';

print("The x in outside is: ");

print(x);

def testField1(){

print(x);

return;

}

testField1();

def testField2(){

int x;

x=12;

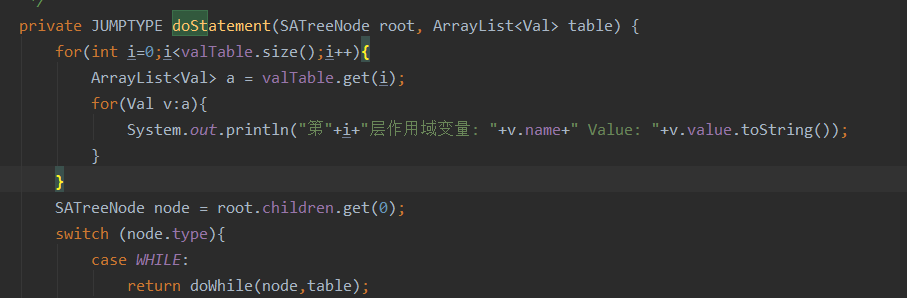
print(x);

return;

}

print("The x in inside is:");

testField2();



**执行结果：**

****

**说明：**

可在需要的地方变量符号表（valTable），打印输出符号表的值。此处我在Unit语法执行函数打印输出符号表中的值，可以看到，此时仅存在全局变量x，类型为CHARACTER。当进入函数后，符号表添加函数作用域，打印出局部变量x，类型为INT。

### 3.3.13 数组下标越界

**测试样例：**

//第一个运行测试

int a[1];

print(a[-1]);

//第二个运行测试

int a[1];

print(a[3]);

**执行结果：**

****

****

**说明：**

对于数组下标，进行界限判断，若为负数或者超出数组大小，则报错。

### 3.3.14 数组下标非法

**测试样例：**

int a[2.0];

**执行结果：**

****

**说明：**

对于数组下标，进行类型判断，要求为整数类型，若为其他类型，则报错处理。

## 3.4 实验总结

### 3.4.1 问题与解决方案

#### 3.4.1.1 函数返回

起初设计语法解释过程时，没有考虑函数的存在，因此没有设计函数返回途径。添加函数语法后，为解释return语句，要求各个语法执行函数返回类型为RETURNTYPE的返回值。当语法树执行到return语句时，向上返回RETURN，父节点识别到RETURN类型的返回值时，直接跳过后续的待执行子节点，继续将RETURN类型向上层返回。此外，对于return语句的返回值，使用全局变量rax储存，执行时，将返回变量赋予给rax，调用者执行完函数过程后，从rax中读取返回变量。此处参考编译过程中的rax寄存器，专门用来储存返回值。

#### 3.4.1.2 错误处理

该部分曾在2.4中讨论，此处针对于语义执行过程中的错误处理进行说明。在语法翻译过程中，我认为此时的语法结构已经在语法分析中得到了完整性的保证，此时仅需要确保各个处理单元能够完成正确的语法解释。在解释过程中，对于错误的语义，选择忽略该处理单元中的错误中断，仅仅报出错误信息。因为我认为各个处理单元之间的关联性相比于语法树中的各个语法单元的关联性要小，即使前面的处理单元出错，后面的处理单元仍有大概率可以继续执行。因此，这里尽可能不进行中断，将更多的语义错误报出。

### 3.4.2 实验的优缺点

语义执行过程中，我并没有使用编译中间代码再对中间代码执行的流程，而是直接借助语法执行过程中的语法树，进行解释执行。执行过程中，底层数据使用Java语言的封装数据进行，语义动作使用Java语言中对应的操作完成。从实现过程上来看，整个解释流程简单易懂，而且实现方便，不需要再对中间代码进行执行操作。同时，底层数据使用封装对象有利于程序的安全性，这一点上继承了实验所用的Java语言的特性。

但是相比于中间代码编译的语义执行方法来说，直接利用语法树进行解释操作无法生成通用的中间代码。因此，对于不同的编译平台需求，无法动态修改后端的执行过程。相应的，需要针对性的设计解释器进行解释过程。此外，在执行效率上，编译型语言的可执行程序产生的是直接执行机器指令，而解释型语言的每一句源代码都要经过解释器解释为可以执行的机器指令，相比之下解释型语言的执行效率会低一些。

# 4 JavaCC的使用

## 4.1模板说明

### 4.1.1 options

options{}部分：这个部分对产生的语法分析器的特性进行说明，例如向前看的token的个数（用来解除冲突）。这一部分是可以省略的，因为每一个选项都有默认值，当我们没有对某个选项进行说明时，它就采用默认值。也可以把这些选项作为javacc命令的参数来启动javacc，可以达到同样的效果。

options {

STATIC = false;

}

### 4.1.2 解析器

解析器类的声明：这个部分指定了分析器类的名字，以及其他类中成员的声明。这个部分是必须有的。这个部分的声明如下：

PARSER\_BEGIN(Adder)

import java.io.\*;

public class Adder {

public static void main(String[] args) {

for (String arg : args) {

try {

System.out.println(evaluate(arg));

// return(evaluate(arg));

} catch (ParseException ex) {

System.err.println(ex.getMessage());

}

}

}

public static long evaluate(String src) throws ParseException {

Reader reader = new StringReader(src);

return new Adder(reader).expr();

}

}

PARSER\_END(Adder)

### 4.1.3 词法部分

词法部分声明：这里面有四类：SKIP、TOKEN、SPECIAL\_TOKEN、MORE。其中，SKIP用来说明被忽略的串，代码如下：

SKIP: { <[" ", "\t", "\r", "\n"]> }

### 4.1.4 Token

TOKEN用来说明在词法层次上识别的token，代码如下：

TOKEN: {

<INTEGER: (["0"-"9"])+>

}

### 4.1.5 语法声明和动作代码

这一部分生成的代码会直接插入解析器类声明的结束括号之前。

long expr():

{

Token x, y;

}

{

x=<INTEGER> "+" y=<INTEGER> <EOF>

{

return Long.parseLong(x.image) + Long.parseLong(y.image);

}

}

## 4.2 生成步骤

### 4.2.1 使用javacc命令处理.jj文件

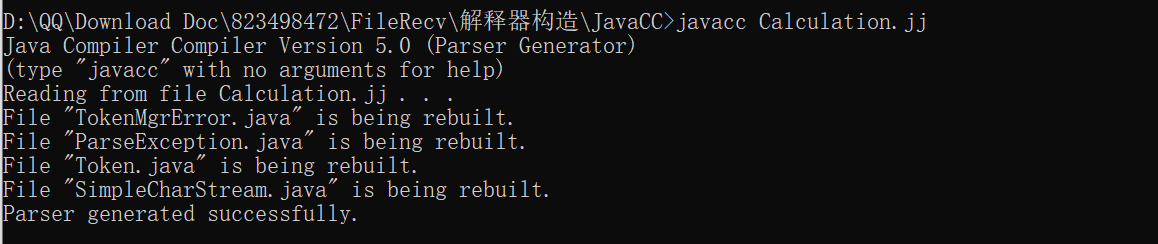


图4.1 javacc处理.jj文件

### 4.2.2 编译生成的Adder.java

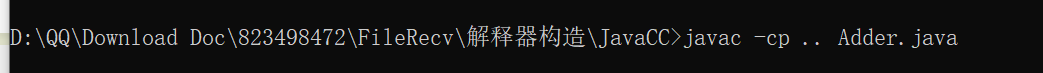
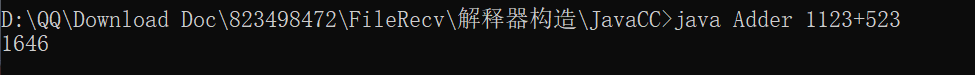


图4.2 编译生成的Adder.java文件

### 4.2.3 测试运行



## 4.3 使用感受

JavaCC是一个词法分析生成器和语法分析生成器。词法分析和语法分析是处理输入字符序列的软件组件，编译器和解释器协同词法分析和语法分析来解码程序文件。词法分析器可以把一连串的字符序列划分成一个一个 “Token”子序列，同时它也可以把这些Token分类。这些Token序列将会传送给语法分析器以供其决定程序的结构。

从开发角度来看，使用JavaCC让用户自定义Token等词法序列，方便了词法分析阶段的开发。并且，JavaCC产生自顶向下的语法分析器，采用自顶向下的分析方法允许更通用的语法，并且易于调试。总的来说，如果使用JavaCC，解释器构造实验中的词法分析与语法分析部分的实现可能会更加轻松。