

编译原理实验报告

# 

学生姓名:

学 号: 201721XXXX

专 业: XXXXXXXX

班 级: 2017 级 2 班

指导教师: 唐益明、李宏芒

完成日期: 2019年10月31日

目 录

[编译原理实验报告 1](#_Toc29661874)

[实验一 词法分析设计 1](#_Toc29661876)

[1.1 实验目的 1](#_Toc29661877)

[1.2 实验内容 1](#_Toc29661878)

[1.3 实验环境 2](#_Toc29661879)

[1.4 词法分析实验设计思想及算法 3](#_Toc29661880)

[1.4.1 实验基本思路 3](#_Toc29661881)

[1.4.2 算法流程 3](#_Toc29661882)

[1.4.3 主要函数及其功能 6](#_Toc29661883)

[1.4.4 核心代码形式化描述及其实现 7](#_Toc29661884)

[1.5 程序运行截图 11](#_Toc29661888)

[实验二 LL(1)分析法 12](#_Toc29661889)

[2.1 实验目的 12](#_Toc29661890)

[2.2 实验内容 12](#_Toc29661891)

[2.3 实验环境 12](#_Toc29661892)

[2.4 LL(1)文法分析实验设计思想及算法 13](#_Toc29661893)

[2.4.1实验基本思路 13](#_Toc29661894)

[2.4.2算法流程 13](#_Toc29661895)

[2.4.3主要函数及其功能 14](#_Toc29661896)

[2.4.4核心代码形式化描述及其实现 16](#_Toc29661897)

[2.5程序运行截图 23](#_Toc29661902)

[实验三 LR(1)分析法 27](#_Toc29661903)

[3.1 实验目的 27](#_Toc29661904)

[3.2 实验内容 27](#_Toc29661905)

[3.3 实验环境 27](#_Toc29661906)

[3.4 LR(1)文法分析实验设计思想及算法 28](#_Toc29661907)

[3.4.1实验基本思路 28](#_Toc29661908)

[3.4.2算法流程 28](#_Toc29661909)

[3.4.3主要函数及其功能 29](#_Toc29661910)

[3.4.4核心代码形式化描述及其实现 30](#_Toc29661911)

[3.5程序运行截图 34](#_Toc29661912)

[参考文献 38](#_Toc29661913)

[附（实验代码链接） 39](#_Toc29661914)

# 实验一 词法分析设计

## 1.1 实验目的

通过本实验的编程实践,使学生了解词法分析的任务,掌握词法分析程序设计的原理和构造方法,使学生对编译的基本概念、原理和方法有完整的和清楚的理解,并能正确地、熟练地运用。

## 1.2 实验内容

用 VC++/VB/JAVA 语言实现对 C 语言子集的源程序进行词法分析。通过输入源程序从左到右对字符串进行扫描和分解,依次输出各个单词的内部编码及单词符号自身值;若遇到错误则显示“Error”,然后跳过错误部分继续显示 ;同时进行标识符登记符号表的管理。

以下是实现词法分析设计的主要工作:

（1）从源程序文件中读入字符。

（2）统计行数和列数用于错误单词的定位。

（3）删除空格类字符,包括回车、制表符空格。

（4）按拼写单词,并用(内码,属性)二元式表示。(属性值——token 的机内表示)

（5）如果发现错误则报告出错。

（6）根据需要是否填写标识符表供以后各阶段使用。

单词的基本分类：

关键字：由程序语言定义的具有固定意义的标识符。也称为保留字例如for、while、printf；单词种别码为1。

标识符：用以表示各种名字，如变量名、数组名、函数名；

常数: 任何数值常数。如：125、1、0.5、3.1416；

运算符：+、-、\*、/；

关系运算符： <、<=、= 、>、>=、<>；

分界符：;、,、(、)、[、]。

## 1.3 实验环境

硬件：

Dell G3 3579；

软件：

OS：Ubuntu 16.04.06；

开发工具：vim 8.0.0056、g++ 5.4.0；

编程语言：C++。

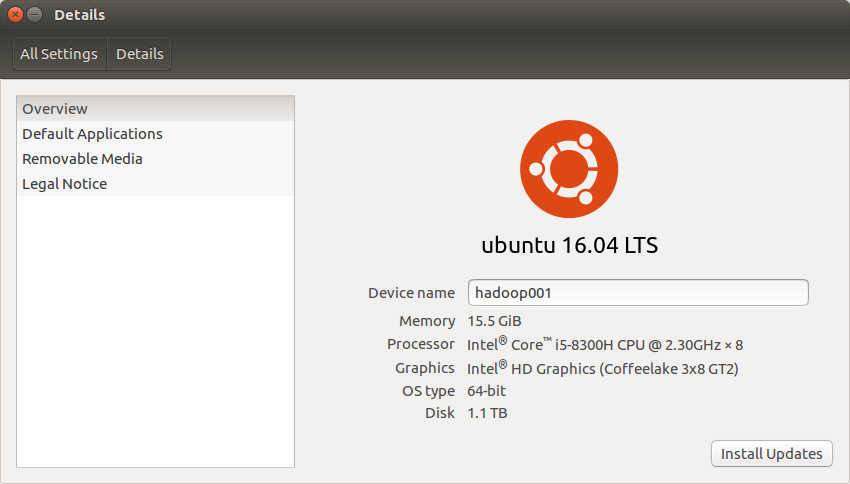


Figure 1 实验环境

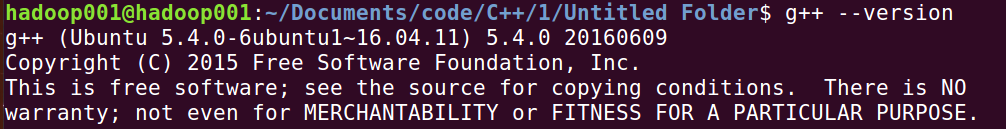


Figure 2 g++版本信息

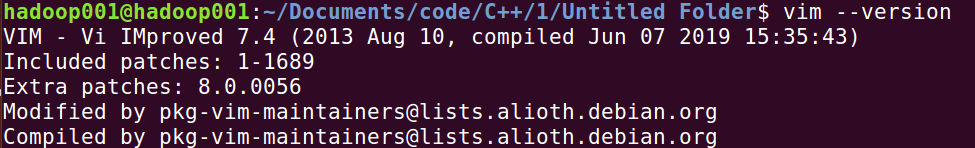


Figure 3 vim版本信息

## 1.4 词法分析实验设计思想及算法

### 1.4.1 实验基本思路

本实验的基本思想是读取代码中的输入字符并产生标记，词法分析器扫描程序的整个源代码，并一一识别每个令牌，值得注意的是，扫描程序通常仅在解析器请求时才生成令牌。基本步骤如下：

1. “获取下一个标记”是从解析器发送到词法分析器的命令；
2. 收到此命令后，词法分析器将扫描输入，直到找到下一个标记；
3. 它将令牌返回给解析器。

在创建这些标记时会跳过空格和注释。如果存在任何错误，那么词法分析器将把该错误与源文件和行号相关联。

### 1.4.2 算法流程

算法流程图如下：

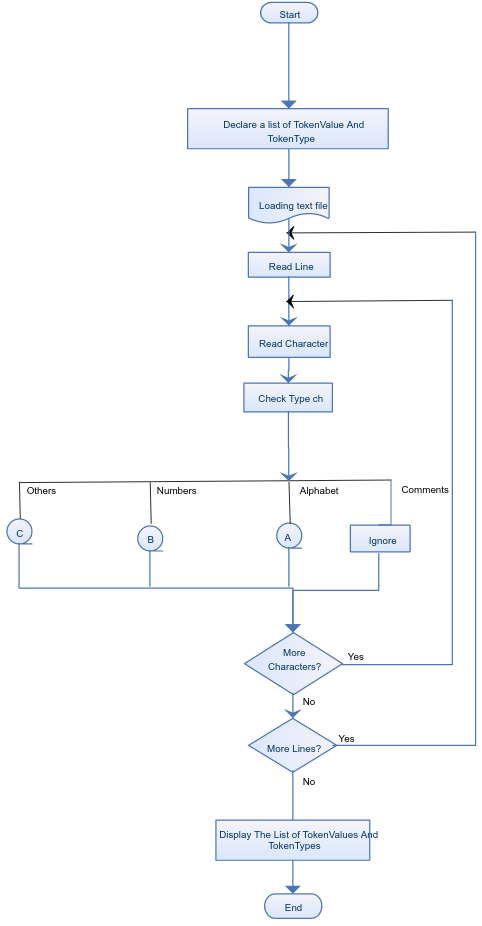


Figure 4 主控程序

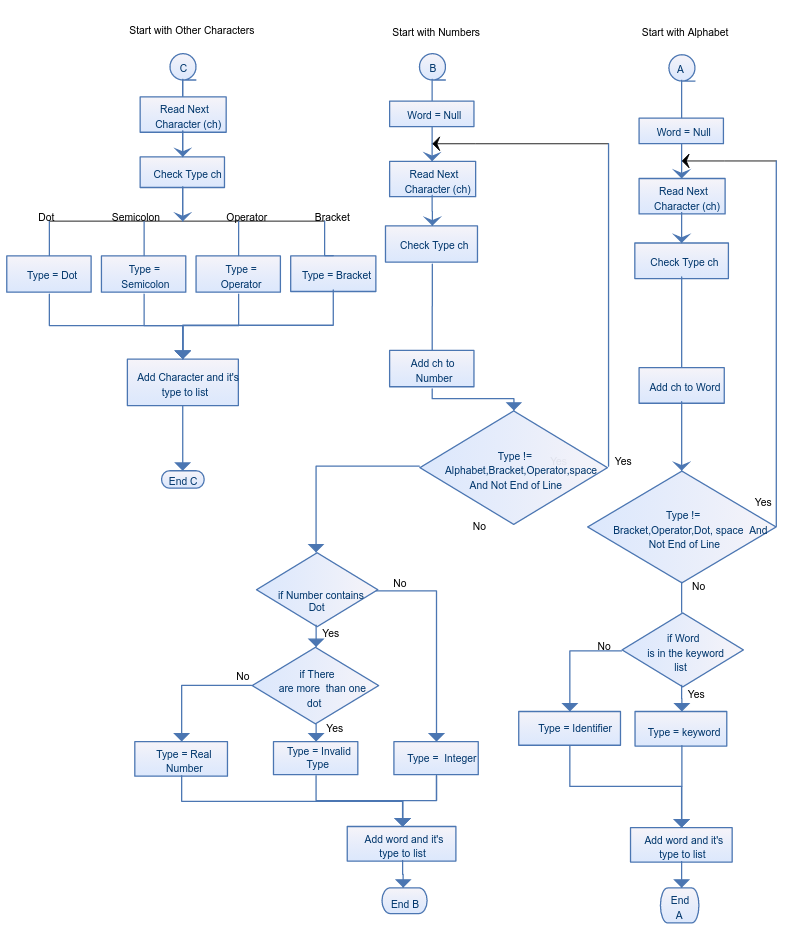


Figure 5 字符分析程序

状态转移图如图6所示。

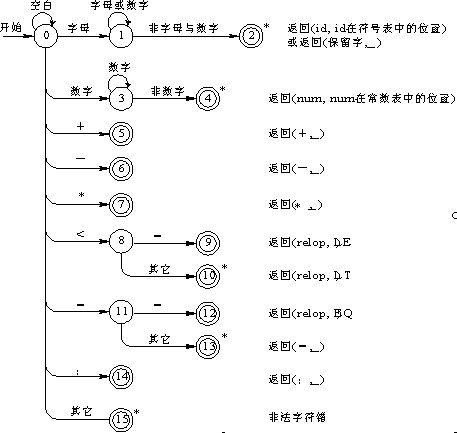


Figure 6 状态转移图

### 1.4.3 主要函数及其功能

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数名 | 参数类型 | 返回类型 | 功能 | 备注 |
| isKey | string | bool | 判断給定字符串 是否为关键字 | C++ |
| isLetter | char | bool | 判断給定字符 是否为字母 | C++ |
| isNumber | char | bool | 判断給定字符 是否为数字 | C++ |
| isS1 | string | bool | 判断給定字符串 是否在分界符表中 | C++ |
| isS2 | string | bool | 判断給定字符 串是否在算术运算符表中 | C++ |
| isS2 | char | bool | 判断給定字符 是否在算术运算符表中 | 函数重载 |
| isS3 | string | bool | 判断給定字符 是否在关系运算符表中 | C++ |
| insertID | string | int | 将給定字符串插入 标识符，并返回 在标识符表中位置 | C++ |
| insertCI | string | int | 将給定字符串插入 标识符，并返回 在常数表中位置 | C++ |
| analyse | FILE \* | void | 分析主函数 | C++ |

Table 1 主要函数概览

### 1.4.4 核心代码形式化描述及其实现

以下伪代码摘自教材，其实，本实验就是对该描述的直接C++实现。

int code, value;

strToken := " ";

GetChar(); GetBC();

if(Isletter())

begin

while (Isletter() or IsDigit())

begin

Concat (O); GetChar();

end

Retract();

code := Reserve();

if(code=0)

begin

value := Insertld(strToken);

return($ID, value);

end

else

return(code, -);

end

else if(IsDigit())

begin

while(Is Digit())

begin

Concat (); Getchar();

end

Retract();

value :=Insert Const(strToken);

return($INT, value);

end

else if(ch ='=')return($ASSICN, -);

else if(ch ='+')return($PLUS, -);

else if(ch＝'\*');

begin

GetChar();

if (ch='\*')return ($POWER，-);

Retract();return($STAR, -);

end

else if(ch = ';')return(SEMICOLON, -);

else if(ch ='(')return ($LPAR, -);

else if(ch =')')return($RPAR, -);

else if(ch ="')return($LBRACE, -);

else if(ch=')return($RBRACE, -);

else ProcEror();

### 

void analyse(FILE \*fpin)//分析主函数

{

    string instring;

    //是否到文件末尾或者出错

    while((ch=fgetc(fpin))!=EOF) {

        instring="";

        //判断空格换行

        if(ch==' '||ch=='\t'||ch=='\n') {

            if(ch=='\n') {

                line++;

                row=0;

            }

        }

        //判断是否为关键字标识符

        else if(isLetter(ch)) {

            while(isLetter(ch)||isNumber(ch)) {

                instring=instring+ch;

                ch=fgetc(fpin);

            }

            if(isKey(instring)) {

                row++;//识别一个关键字，增加一个单词位

                cout<<instring<<'\t'<<'\t'<<"(1,"<<instring<<")"<<'\t'<<'\t'<<"关键字"<<'\t'<<'\t'<<"("<<line<<","<<row<<")"<<endl;

            }

            else {

                row++;//识别一个标识符，增加一个单词位

                insertID(instring);

                cout<<instring<<'\t'<<'\t'<<"(6,"<<instring<<")"<<'\t'<<'\t'<<"标识符"<<'\t'<<'\t'<<"("<<line<<","<<row<<")"<<endl;

            }

            fseek(fpin,-1L,SEEK\_CUR);

        }

        //判断是否为常数

        else if(isNumber(ch)) {

            while(isNumber(ch)||ch=='.'&&isNumber(fgetc(fpin))) {

                instring=instring+ch;

                ch=fgetc(fpin);

            }

            if(isLetter(ch)) {

### 

while(isLetter(ch)||isNumber(ch)) {

                    instring=instring+ch;

                    ch=fgetc(fpin);

                }

                row++;

                cout<<instring<<'\t'<<'\t'<<"Error"<<'\t'<<'\t'<<"Error"<<'\t'<<'\t'<<"("<<line<<","<<row<<")"<<endl;

            }

            else {

                insertCI(instring);

                row++;

                cout<<instring<<'\t'<<'\t'<<"(5,"<<instring<<")"<<'\t'<<'\t'<<"常数"<<'\t'<<'\t'<<"("<<line<<","<<row<<")"<<endl;

            }

            fseek(fpin,-1L,SEEK\_CUR);

        }

        //判断是否为符号

        else {

            //int cnt;

            row++;

            instring=ch;

            //判断分界符

            if(isS1(instring)) {

                //cnt = 0;

                cout<<instring<<'\t'<<'\t'<<"(2,"<<instring<<")"<<'\t'<<'\t'<<"分界符"<<'\t'<<'\t'<<"("<<line<<","<<row<<")"<<endl;

            }

            //判断算术运算符

            else if(isS2(instring)) {

                ch=fgetc(fpin);

                fseek(fpin,-1L,SEEK\_CUR);

                char ch1=fgetc(fpin);

                if(isS2(ch)) {

                    instring=instring+ch;

                    cout<<instring<<'\t'<<'\t'<<"Error"<<'\t'<<'\t'<<"Error"<<'\t'<<'\t'<<"("<<line<<","<<row<<")"<<endl;

                }

                else if(ch == ch1) {

                    cout<<instring<<'\t'<<'\t'<<"Error"<<'\t'<<'\t'<<"Error"<<'\t'<<'\t'<<"("<<line<<","<<row<<")"<<endl;

                }

                else {

                    cout<<instring<<'\t'<<'\t'<<"(3,"<<instring<<")"<<'\t'<<'\t'<<"算术运算符"<<'\t'<<"("<<line<<","<<row<<")"<<endl;

                }

            }

### 

else if(isS3(instring)) {

                string si;

                ch=fgetc(fpin);

                si=instring+ch;

                if(isS3(si))

                    cout<<si<<'\t'<<'\t'<<"(3,"<<instring<<")"<<'\t'<<'\t'<<"关系运算符"<<'\t'<<"("<<line<<","<<row<<")"<<endl;

                else {

                    cout<<instring<<'\t'<<'\t'<<"(3,"<<instring<<")"<<'\t'<<'\t'<<"关系运算符"<<'\t'<<"("<<line<<","<<row<<")"<<endl;

                    fseek(fpin,-1L,SEEK\_CUR);

                }

            }

            else

                cout<<instring<<'\t'<<'\t'<<"Error"<<'\t'<<'\t'<<"Error"<<'\t'<<'\t'<<"("<<line<<","<<row<<")"<<endl;

        }

    }

}

## 1.5 程序运行截图



Figure 7 样例运行结果

# 实验二 LL(1)分析法

## 2.1 实验目的

通过完成预测分析法的语法分析程序,了解预测分析法和递归子程序法的区别和联系。使学生了解语法分析的功能,掌握语法分析程序设计的原理和构造方法,训练学生掌握开发应用程序的基本方法。有利于提高学生的专业素质,为培

养适应社会多方面需要的能力。

## 2.2 实验内容

1. 根据某一文法编制调试LL ( 1 )分析程序,以便对任意输入的符号串进行分析。
2. 构造预测分析表,并利用分析表和一个栈来实现对上述程序设计语言的分析程序。
3. 分析法的功能是利用LL(1)控制程序根据显示栈栈顶内容、向前看符号以及LL(1)分析表,对输入符号串自上而下的分析过程。

## 2.3 实验环境

硬件：

Dell G3 3579；

软件：

OS：Ubuntu 16.04.06；

IDE：IntelliJ IDEA Ultimate Edition（2019.1.3）；

编程语言：Scala、Java。

## 2.4 LL(1)文法分析实验设计思想及算法

### 2.4.1实验基本思路

首先给出LL(1)文法的定义[2]：

一个文法G被称为LL(1)文法，如果它满足以下条件：

1. 文法不含左递归。
2. 对于文法中每一个非终结符A的各个产生式的候选首字符集两两不相交。即，若

A→α1|α2|...|αn

则 FIRST(αi)∩FIRST(αj) = Φ (i≠j)

1. 对文法中的每个非终结符A，若它存在某个候选首字符集包含ε，则

FIRST(αi)∩FOLLOW(A) (i = 1, 2, , ..., n)

这里，LL(1)的第一个L表示从左向右扫描输入串，第二个L表示最左推导，1表示分析时每一步只需向前查看一个符号。

对于一个LL(1)文法，可以对其输入穿进行有效的无回溯的自上而下分析。假设要用非终结符A进行匹配，面临的输入符号为a，A的所有产生式为：

A→α1|α2|...|αn

1. 若a∈FIRST(αi)，则指派αi去执行匹配任务。
2. 若a不属于任何一个候选首符集，则：

①若ε属于某个FIRST(αi)且a∈FOLLOW(A)，则让A与ε自动匹配；

②否则，a的出现是一种错误。

综述，本实验对教材上的描述的几个算法进行了实现，成功达成了LL(1)文法分析，并进行了简单的测试。

### 2.4.2算法流程

LL(1)文法分析的架构如图8所示。

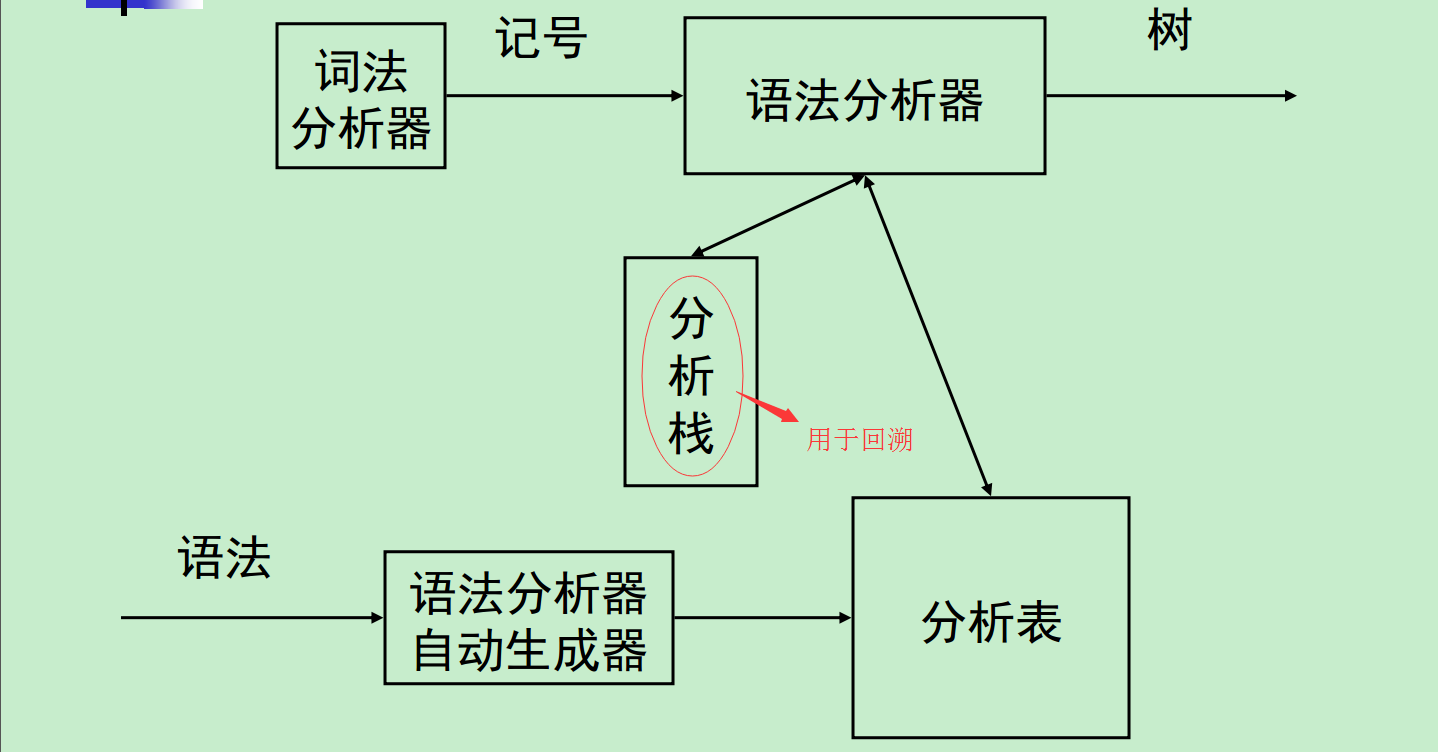


Figure 8 LL文法分析架构

### 2.4.3主要函数及其功能

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数名 | 参数类型 | 返回类型 | 功能 | 备注 |
| eliminateLeftRecursion | 无 | ArrayBuffer[(String,  String, String)] | 消除形式上的左递归 | Scala |
| FIRST | ArrayBuffer[(String,  String)] | Map[String, String] | 求解指定文法FIRST集 | 迭代求解，因此 代码较长，Scala |
| dfsFOLLOW | String | String | DFS寻找各个非终结符的 FOLLOW集元素 | Scala |
| FOLLOW | ArrayBuffer[(String,  String)] | Map[String, String] | 根据dfsFOLLOW函数， 获取各个非终结符的 FOLLOW集元素 | Scala |
| initiateMatrix | 无 | Array[Array[String]] | 初始化分析表 | Scala |
| createMatrix | 无 | Array[Array[String]] | 构造分析表 | Scala |
| analyse | String | Boolean | 对指定的字符串LL(1)分析 | Scala |
| GUI1 | 无 | 无 | 实现图形化界面展示 | 开始界面，Java |
| GUI2 | 无 | 无 | 实现图形化界面展示 | 分析界面，Java |

Table 2 LL(1)文法分析实验代码主要函数概览

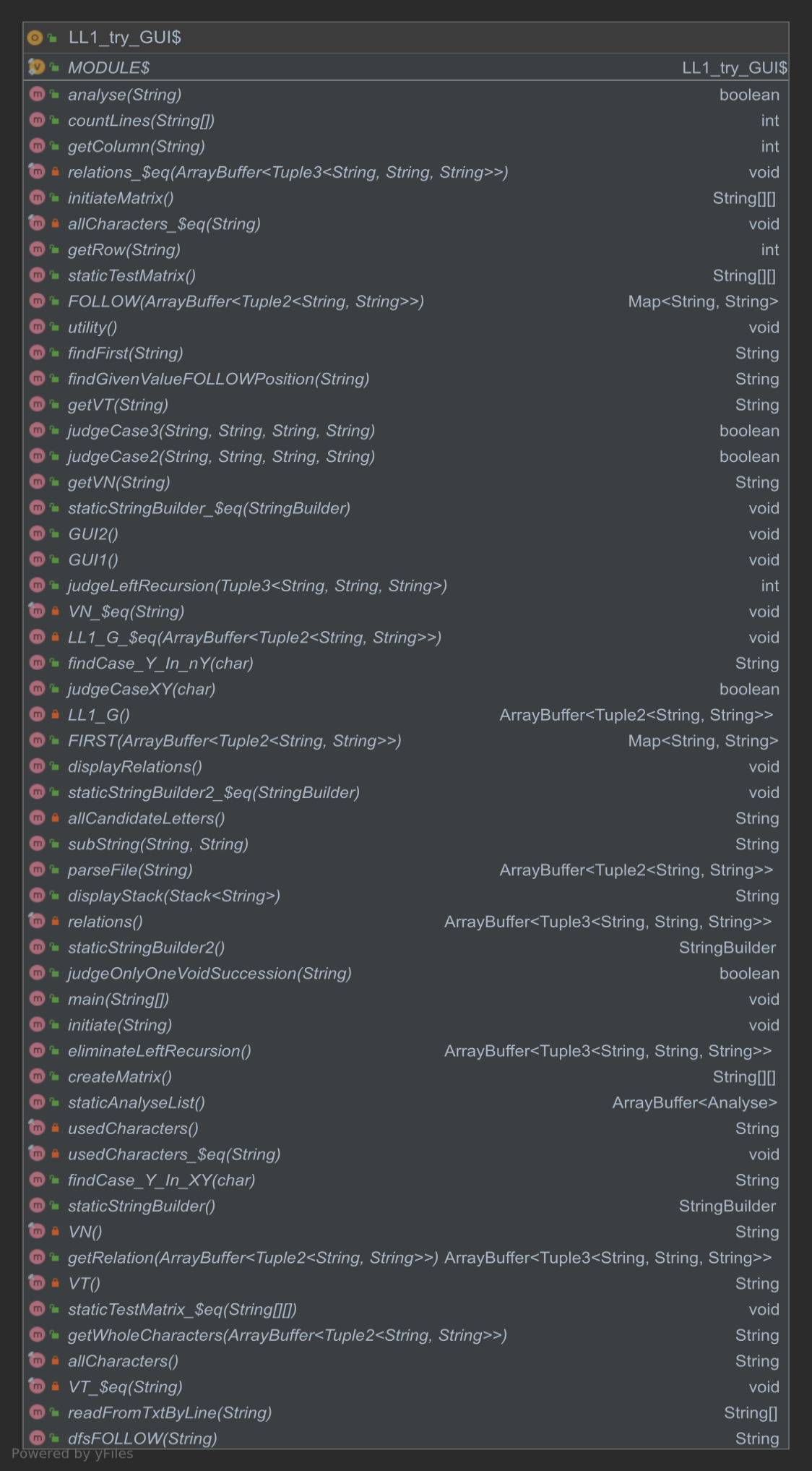


Figure 9 LL(1)文法分析实验代码完整函数

### 2.4.4核心代码形式化描述及其实现

实现LL(1)文法分析的一种有效方法是使用一张分析表和一个栈进行联合控制，而分析表的构造需要用到給定文法的FIRST集与FOLLOW集[2]。下面依次介绍预测分析程序的总控程序、FIRST集、FOLLOW集与分析表的构造流程。

预测分析程序的总控程序形式化描述[1]：

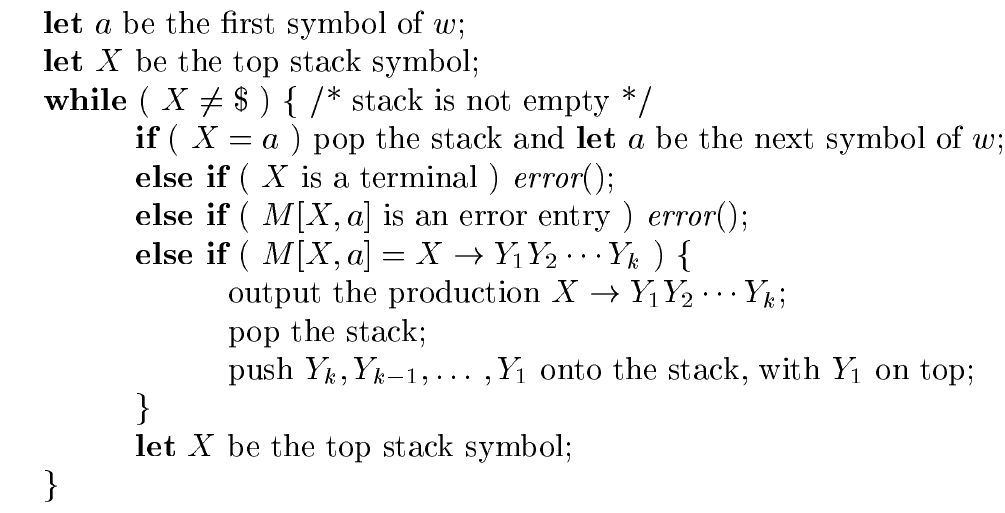


Figure 10 LL(1)预测分析总控形式化描述

FIRST集构造流程[1]：

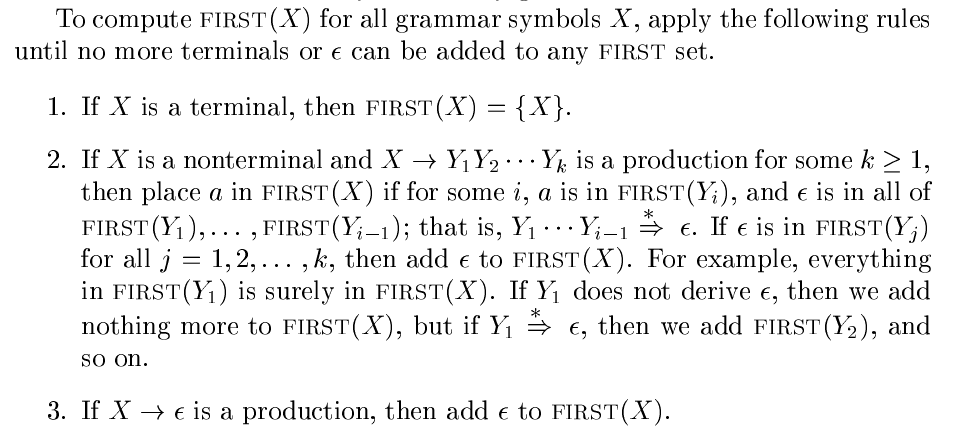


Figure 11 FIRST集构造流程

FOLLOW集构造流程[1]：

To compute FOLLOW ( A ) for all nonterminals A , apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW ( S ), where S is the start symbol, and $ is the input right endmarker.
2. If there is a production A → αBβ , then everything in FIRST (β) except ε is in FOLLOW ( B ).
3. If there is a production A → αB, or a production A → αBβ, where FIRST (β) contains ε, then everything in FOLLOW (A) is in FOLLOW (B).

Scala实现FIRST函数：

def FIRST( string: ArrayBuffer[ (String, String) ] ): Map[ String, String ] = {

val FIRST\_Group = Map[ String, String ]()

val wholeCharacters = allCharacters

val localVT = VT

val localVN = VN

for( character <- wholeCharacters ) {

// case 1

if( localVT.contains(character) ) {

//if there exist the original key that equals the current one

if( FIRST\_Group.contains(character.toString) == true ) {

val tmp = character.toString + FIRST\_Group(character.toString)

FIRST\_Group(character.toString) = tmp.distinct

}

//otherwise

else {

FIRST\_Group(character.toString) = character.toString

}

}

// case 2

if( localVN.contains(character.toString) == true ) {

// case 2.1

val value = findFirst(character.toString)

if ( value.length != 0 ) {

if ( FIRST\_Group.contains(character.toString) == true ) {

for( ch <- value ) {

val tmp = ch + FIRST\_Group(character.toString)

FIRST\_Group(character.toString) = tmp.distinct

}

}

### 

else {

FIRST\_Group(character.toString) = value.toString

}

}

// case 2.2

if( judgeOnlyOneVoidSuccession(character.toString) == true ) {

if ( FIRST\_Group.contains(character.toString) == true ) {

val tmp = "ε" + FIRST\_Group(character.toString)

FIRST\_Group(character.toString) = tmp.distinct

}

else {

FIRST\_Group(character.toString) = "ε"

}

}

}

for( character <- wholeCharacters ) {

// case 3.1

if( judgeCaseXY(character) == true ) {

val tmpReply = findCase\_Y\_In\_XY(character)

for( eachTmpReply <- tmpReply ) {

if( FIRST\_Group.contains(eachTmpReply.toString) == true ) {

for (ex <- FIRST\_Group(eachTmpReply.toString)) {

if (ex != 'ε') {

if (FIRST\_Group.contains(character.toString) == true) {

val tmp = ex.toString + FIRST\_Group(character.toString)

FIRST\_Group(character.toString) = tmp.distinct

}

else {

FIRST\_Group(character.toString) = ex.toString

}

}

}

}

}

}

// case 3.2

if( findCase\_Y\_In\_nY(character).length > 0 ) {

var flag = true

val tmpReply = findCase\_Y\_In\_nY(character)

for( ex <- tmpReply ) {

if( localVN.contains(ex.toString) && FIRST\_Group.contains(ex.toString) == true ) {

if( FIRST\_Group(ex.toString).contains("ε") == false ) {

flag = false

}

}

else flag = false

### 

if( flag == true ) {

if (FIRST\_Group.contains(character.toString) == true) {

val tmp = FIRST\_Group(ex.toString).replace( "ε", "" ) + FIRST\_Group(character.toString)

FIRST\_Group(character.toString) = tmp.distinct

}

else {

FIRST\_Group(character.toString) = FIRST\_Group(ex.toString).replace( "ε", "" )

}

}

}

}

// case 3.3

if( findCase\_Y\_In\_nY(character).length > 0 ) {

var flag = true

val tmpReply = findCase\_Y\_In\_nY(character)

for( ex <- tmpReply ) {

if( localVN.contains(ex.toString) && FIRST\_Group.contains(ex.toString) == true ) {

if( FIRST\_Group(ex.toString).contains("ε") == false ) {

flag = false

}

}

else {

flag = false

}

if( flag == true ) {

if (FIRST\_Group.contains(character.toString) == true) {

val tmp = "ε" + FIRST\_Group(character.toString)

FIRST\_Group(character.toString) = tmp.distinct

}

else {

FIRST\_Group(character.toString) = "ε"

}

}

}

}

}

}

FIRST\_Group

}

Scala实现FOLLOW、dfsFOLLOW与analyse函数：

def dfsFOLLOW( ch: String ): String = {

val FOLLOWPositions = Map[ String, String ]()

val FOLLOW\_Group = Map[ String, String ]()

val localLL1\_G = LL1\_G

val FIRST\_Group = FIRST(localLL1\_G)

val localVN = VN

for( ch <- localVN ) {

FOLLOWPositions(ch.toString) = findGivenValueFOLLOWPosition(ch.toString)

FOLLOW\_Group(ch.toString) = "#"

}

var result = ""

if( FOLLOWPositions(ch).length == 4 ) {

if( FOLLOWPositions(ch)(1).toString == "T" ) {

result += FIRST\_Group( FOLLOWPositions(ch)(0).toString )

FOLLOW\_Group(ch) += result.distinct

}

else if( FOLLOWPositions(ch)(3).toString == "T" ) {

result += FIRST\_Group( FOLLOWPositions(ch)(2).toString )

FOLLOW\_Group(ch) += result.distinct

}

if( FOLLOWPositions(ch)(1).toString == "W" ) {

result += dfsFOLLOW( FOLLOWPositions(ch)(0).toString )

FOLLOW\_Group(ch) = result.distinct

}

else if( FOLLOWPositions(ch)(3).toString == "W" ) {

result += dfsFOLLOW( FOLLOWPositions(ch)(2).toString )

FOLLOW\_Group(ch) = result.distinct

}

}

def FOLLOW( string: ArrayBuffer[ (String, String) ] ): Map[ String, String ] = {

val localVN = VN

val FOLLOW\_Group = Map[ String, String ]()

for( ch <- localVN ) {

FOLLOW\_Group(ch.toString) = dfsFOLLOW(ch.toString)

}

FOLLOW\_Group

}

## 

if( FOLLOWPositions(ch).length == 2 ) {

if( FOLLOWPositions(ch)(1).toString == "T" ) {

result += FIRST\_Group( FOLLOWPositions(ch)(0).toString )

FOLLOW\_Group(ch) = result.distinct

}

else if( FOLLOWPositions(ch)(1).toString == "W" ) {

result += dfsFOLLOW( FOLLOWPositions(ch)(0).toString )

FOLLOW\_Group(ch) = result.distinct

}

}

FOLLOW\_Group(ch).replace("ε", "")

}

## 

def analyse( expression: String ): Boolean = {

val stack = new mutable.Stack[String]()

var localExpression = expression

val table = createMatrix()

val localVT = VT

val localVN = VN

val localRelations = relations

stack.push("#")

stack.push( localRelations(0).\_1 )

var cnt = 0

staticAnalyseList.append(new Analyse("步骤","分析栈","剩余字符串","所用产生式","动作"));

staticAnalyseList.append(new Analyse(cnt.toString, displayStack(stack).reverse.toString,localExpression.toString,"","initiate"));

while( stack.isEmpty == false ) {

val stackTop = stack.top

stack.pop()

// 栈顶符号属于 非终结符

if( localVN.contains(stackTop) == true ) {

// 栈顶符号与表达式左端首字符 存在 关系

if( table( getRow(stackTop) )( getColumn( localExpression(0).toString ) ) != null ) {

val lastHalf = table( getRow(stackTop) )( getColumn( localExpression(0).toString ) ).split( "->", 2 ).last

val length = lastHalf.length

for( i <- 0 to (length - 1) ) {

if( lastHalf != "ε" ) {

stack.push(lastHalf(length - 1 - i).toString)

}

}

cnt += 1

if( lastHalf != "ε" ) {

staticAnalyseList.append(new Analyse(cnt.toString, displayStack(stack).reverse.toString, localExpression.toString, table(getRow(stackTop))(getColumn(localExpression(0).toString)), "POP, PUSH(" + lastHalf.reverse + ")"));

}

else {

staticAnalyseList.append(new Analyse(cnt.toString, displayStack(stack).reverse.toString, localExpression.toString, table(getRow(stackTop))(getColumn(localExpression(0).toString)), "POP"));

}

}

// 栈顶符号与表达式左端首字符 不存在 关系

else {

// 栈顶符号 等于 表达式左端首字符

if( stackTop == "#" && localExpression(0).toString == "#" ) {

println("11111")

return true

}

// 栈顶符号 不等于 表达式左端首字符

else {

println("1 - error")

staticAnalyseList.append(new Analyse(cnt.toString, displayStack(stack).reverse.toString,localExpression.toString,"",""));

return false

}

}

}

// 栈顶符号属于 终结符

if( localVT.contains(stackTop) == true ) {

// 栈顶符号 等于 表达式左端首字符

if( stackTop == localExpression(0).toString ) {

if( stackTop == localExpression(0).toString ) {

//stack.pop()

localExpression = localExpression.drop(1)

cnt += 1

staticAnalyseList.append(new Analyse(cnt.toString, displayStack(stack).reverse.toString,localExpression.toString,"","GETNEXT(" + stackTop + ")"));

}

// 栈顶符号 不等于 表达式左端首字符

else {

println("2 - error")

return false

}

}

}

}

true

}

## 2.5程序运行截图

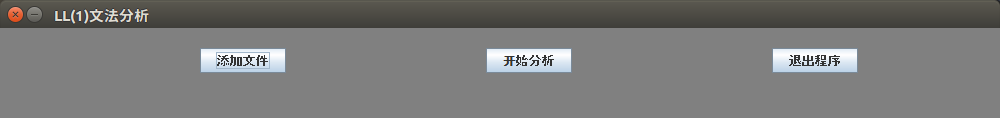


Figure 12 开始界面

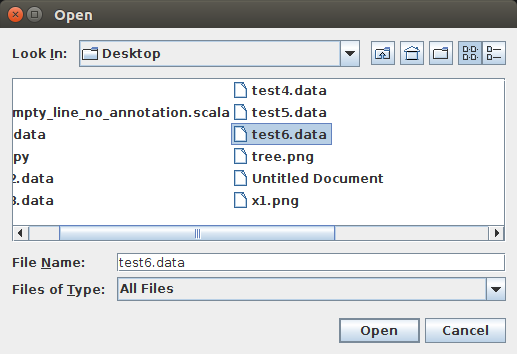


Figure 13 选择文件

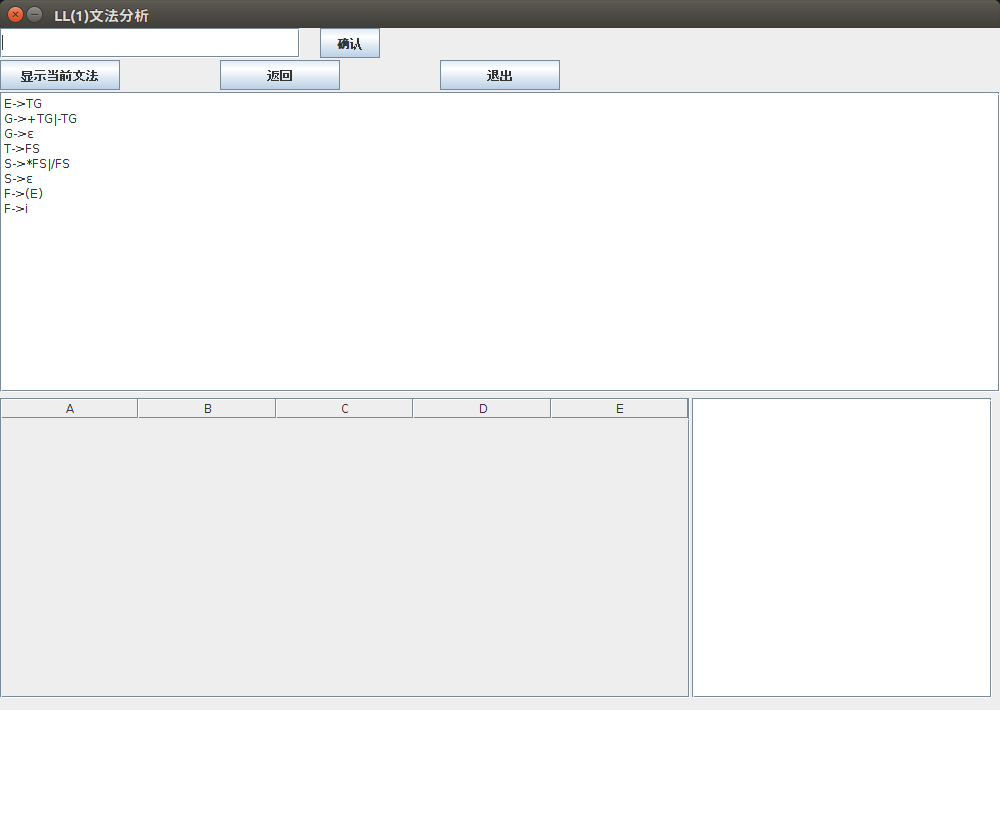


Figure 14 显示当前文法

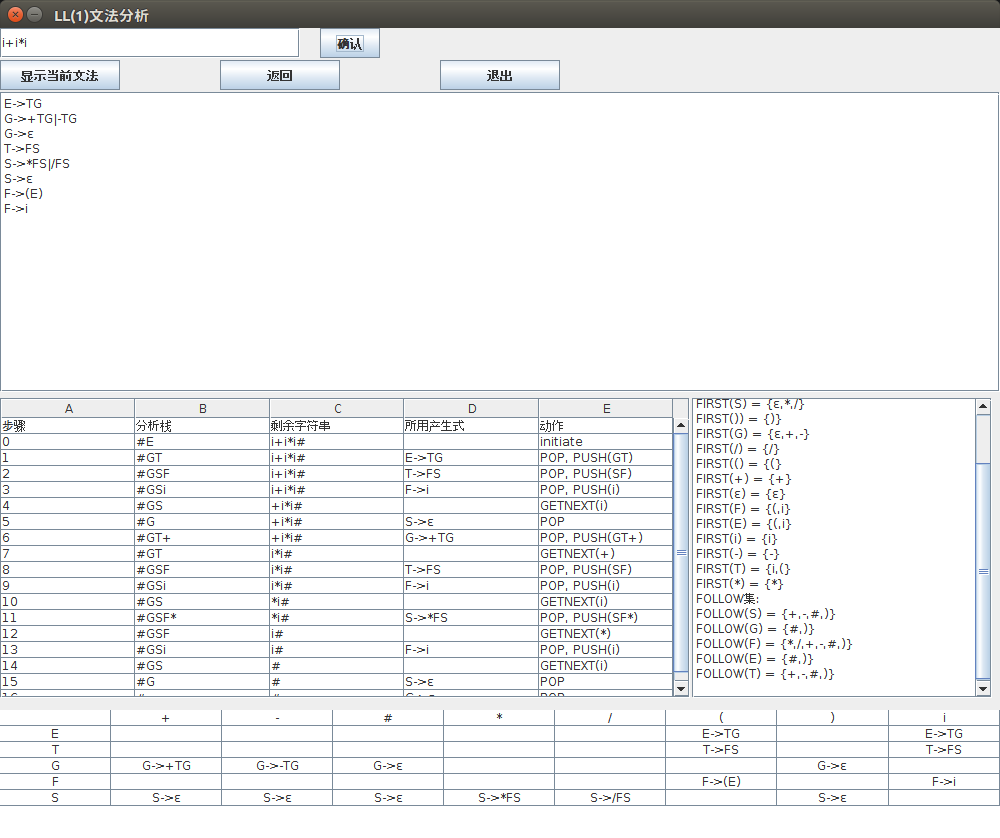


Figure 15 执行文法分析（输入表达式为“i+i\*i”）

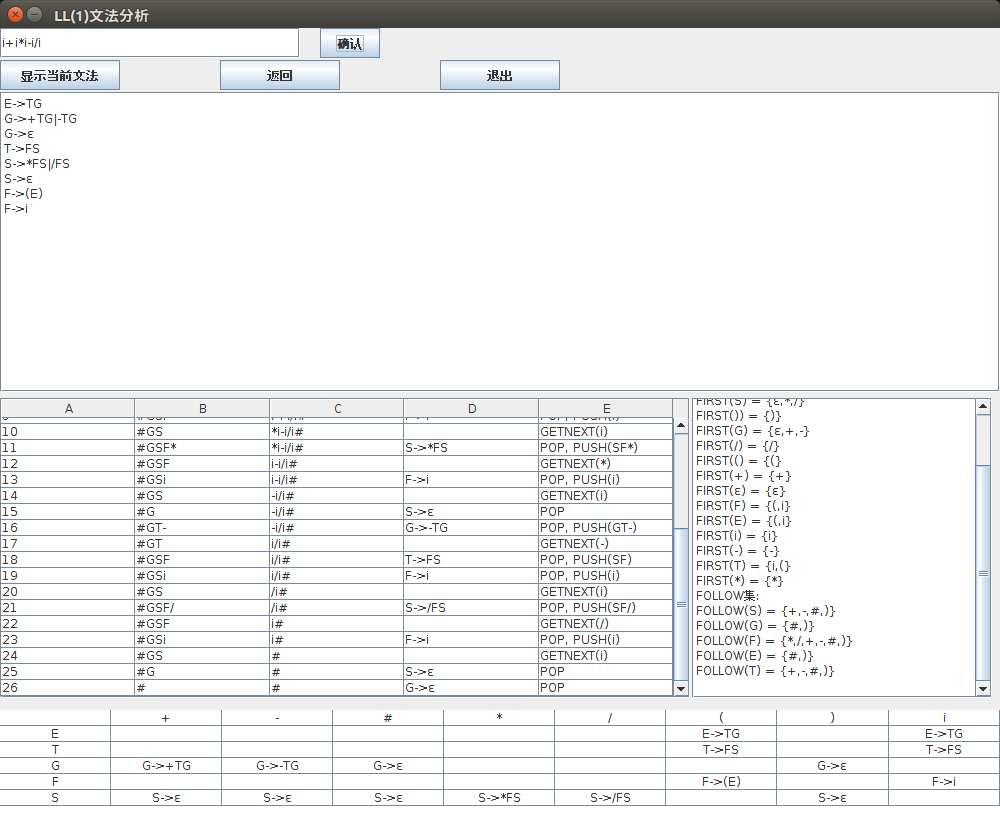


Figure 16 执行文法分析（输入表达式为“i+i\*i-i/i”）

# 实验三 LR(1)分析法

## 3.1 实验目的

构造LR(1)分析程序，利用它进行语法分析,判断给出的符号串是否为该文法识别的句子，了解LR(K)分析方法是严格的从左向右扫描，和自底向上的语法分析方法。

## 3.2 实验内容

对下列文法，用 LR(1)分析法对任意输入的符号串进行分析:

(1)E-> E+T

(2)E->T

(3)T-> T\*F

(4)T->F

(5)F-> (E)

(6)F-> i

## 3.3 实验环境

硬件：

Dell G3 3579；

软件：

OS：Ubuntu 16.04.06；

IDE：IntelliJ IDEA Ultimate Edition（2019.1.3）；

编程语言：Scala、Java。

## 3.4 LR(1)文法分析实验设计思想及算法

### 3.4.1实验基本思路[2]

LR文法的每个项目的一般形式是[A→α·β, a1a2...ak]，此处，A→α·β是一个LR(0)项目，每一个a都是终结符。这样的一个项目称为一个LR(k)项目。项目中的a1a2...ak称为它的向前搜索符串（或展望串）。向前搜索符串仅对规约项目[A→α·β, a1a2...ak]有意义。对于任何移进或待约项目[A→α·β, a1a2...ak]，β≠ε，搜索符串a1a2...ak没有作用。规约项目[A→α·, a1a2...ak]意味着：当它所属的状态呈现在栈顶且后续的k个输入符号为a1a2...ak时，才可以把栈顶上的α规约为A。我们只对k≤1的情形感兴趣，因为，对多数程序语言的语法来说，向前搜索（展望）一个符号就多半可以确定“移进”或“规约”。

综述，本实验对教材上的描述的几个算法进行了实现，成功达成了LR(1)文法分析，并进行了简单的测试。

### 3.4.2算法流程

LR文法分析的架构如图17所示。

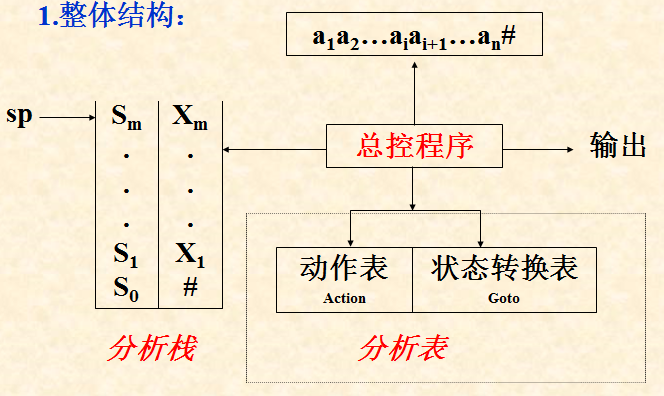


Figure 17 LR文法分析架构

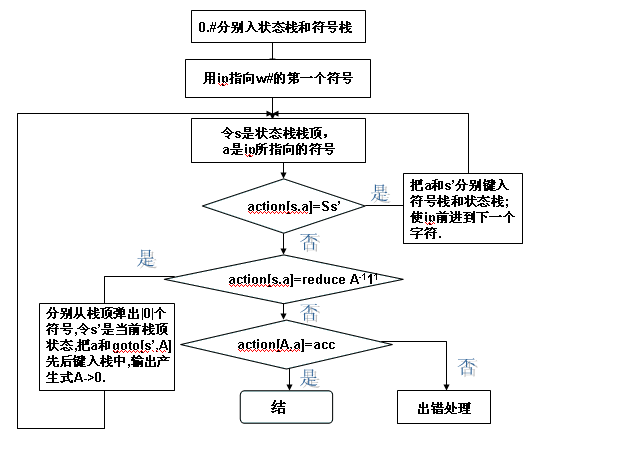


Figure 18 LR文法分析流程图

### 3.4.3主要函数及其功能

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数名 | 参数类型 | 返回类型 | 功能 | 备注 |
| FIRST | ArrayBuffer[(String,  String)] | Map[String, String] | 求解指定文法FIRST集 | 迭代求解，因此 代码较长，Scala |
| getClosure | ArrayBuffer[ (String,  String, String) ] | ArrayBuffer[ (String,  String, String) ] | 求給定项目集的闭包 | Scala |
| go | ArrayBuffer[ (String,  String, String), String | ArrayBuffer[ (String,  String, String) ] | 求給定项目对于特定 字符的下一状态 | Scala |
| createMatrix | 无 | Array[ Array[String] ] | 构造ACTION与GOTO分析表 | Scala |
| getItemGroup | 无 | 无 | 建立初始化的项目集 | Scala |
| analyse | String | Boolean | 对指定的字符串进行LR(1)分析 | Scala |
| GUI1 | 无 | 无 | 实现图形化界面展示 | 开始界面，Java |
| GUI2 | 无 | 无 | 实现图形化界面展示， | 分析界面，Java |

Table 3 LR(1)文法分析实验代码主要函数概览

### 3.4.4核心代码形式化描述及其实现

求解闭包伪代码[1]：

SetOfItems CLOSURE (I) {

repeat

for ( each item [A → α·Bβ, a] in I )

for ( each production B → γ in G’ )

for ( each terminal b in FIRST(βa) )

add [ B → ·γ, b ] to set I ;

until no more items are added to I ;

return I ;

}

GOTO函数伪代码[1]：

SetOfItems GOTO ( I, X ) {

initialize J to be the empty set;

for ( each item [A → α·Xβ, a] in I )

add item [A → α·Xβ, a] to set J;

return CLOSURE(J);

}

求解项目集族伪代码[1]：

void items ( G’ ) {

initialize C to { CLOSURE( { [S’ → ·S, $] } ) };

repeat

for ( each set of items I in C )

for ( each grammar symbol X )

if ( GOTO ( I, X ) is not empty and not in C )

add GOTO ( I, X ) to C;

until no new sets of items are added to C;

}

Scala实现getClosure函数：

def getClosure( items: ArrayBuffer[ (String, String, String) ] ): ArrayBuffer[ (String, String, String) ] = {

val result = new ArrayBuffer[ (String, String, String) ]()

result.appendAll(items)

val localFIRST = FIRST()

var addFlag = true

var cnt = 1

while (addFlag == true ) {

val originalResult = new ArrayBuffer[(String, String, String)]()

originalResult.appendAll(result)

for (ex <- result) {

val pointPosition = ex.\_2.indexOf("·")

//· 不在最右边

if (pointPosition < ex.\_2.length - 1) {

//B在 · 的右边

val B = ex.\_2(pointPosition + 1)

val a = ex.\_3

// case 1: β != Φ and a != # or

// case 2: β != Φ and a = #

if (pointPosition < ex.\_2.length - 2) {

val β = ex.\_2(pointPosition + 2)

// ξ

val rightExpressionsOfB = getRightExpressions(B.toString)

val FIRST\_Of\_βa = localFIRST(β.toString)

for (b <- FIRST\_Of\_βa) {

for (ksi <- rightExpressionsOfB) {

val tmp = ((B.toString, "·" + ksi, b.toString))

if (result.contains(tmp) == false) {

result += tmp

}

}

}

}

// case 3: β = Φ and a equals any character

if (pointPosition == ex.\_2.length - 2) {

val rightExpressionsOfB = getRightExpressions(B.toString)

val FIRST\_Of\_βa = localFIRST(a.toString)

for (b <- FIRST\_Of\_βa) {

for (ksi <- rightExpressionsOfB) {

val tmp = ((B.toString, "·" + ksi, b.toString))

if (result.contains(tmp) == false) {

result += tmp

}

}

}

}

}

}

if (result != originalResult) {

originalResult.remove(0, originalResult.length)

originalResult.appendAll(result)

cnt += 1

}

else {

addFlag = false

cnt += 1

}

}

result

}

Scala实现go函数：

def go( I: ArrayBuffer[ (String, String, String) ], X: String ): ArrayBuffer[ (String, String, String) ] = {

//GO(I, X) = CLOSURE(J)

//J = {任何形如[A->αX·β, a]的项目|[A->α·Xβ, a]∈I}

val ans = new ArrayBuffer[ (String, String, String) ]()

val items = new ArrayBuffer[ (String, String, String) ]()

for( ex <- I ) {

val pointPosition = ex.\_2.indexOf("·")

//· 不在最右边

if (pointPosition < ex.\_2.length - 1) {

val A = ex.\_1

val possibleX = ex.\_2( pointPosition + 1)

// αXβ

val noPointExpressionPart2 = ex.\_2.replace("·", "")

if( X == possibleX.toString ) {

// αX·β

val newPart2 = noPointExpressionPart2.substring(0, pointPosition + 1) + "·" +

noPointExpressionPart2.substring(pointPosition + 1, noPointExpressionPart2.length)

val a = ex.\_3

items += ( (A, newPart2, a) )

}

}

}

ans.appendAll( getClosure(items) )

ans

}

Scala实现getItemGroup函数：

def getItemGroup(): Unit = {

val ldx = ( relations(0).\_1, "·" + relations(0).\_2, "#" )

val I0 = getClosure( ArrayBuffer(ldx) )

val wholeCharacters = allCharacters

var tot = 0

itemGroup(I0) = tot

var appendFlag = true

while (appendFlag == true) {

var originalAns = Map[ ArrayBuffer[ (String, String, String) ], Int ]()

originalAns = itemGroup.clone()

//为什么用I作为遍历变量不行？！

for(item <- itemGroup.keys) {

for (ch <- wholeCharacters) {

val newItem = go(item, ch.toString).sorted

if (newItem.isEmpty == false && itemGroup.contains(newItem) == false) {

tot += 1

itemGroup(newItem) = tot

}

}

}

if( originalAns.equals(itemGroup) == true ) {

appendFlag = false

}

else {

originalAns.clear()

originalAns = itemGroup.clone()

}

}

}

Scala实现createMatrix函数：

def createMatrix(): Array[ Array[String] ] = {

val result = initiateMatrix()

val localVT = VT

val localVN = VN

case class getColumn( ch: String ) {

val matrix = initiateMatrix()

var ans = -1

for( j <- 0 to (columnLength - 1) ) {

if( matrix(0)(j) == ch ) {

ans = j

}

}

}

for( ex <- itemGroup ) {

for( tx <- ex.\_1 ) {

val pointPosition = tx.\_2.indexOf("·")

//· 不在最右边

//若项目[A->α·aβ] ∈ Ik，且GO(Ik, a) = Ij，a为终结符，则置ACTION[k, a]为“sj”

if (pointPosition < tx.\_2.length - 1) {

val a = tx.\_2( pointPosition + 1 )

if( localVT.contains(a) == true && findItemOrder(ex.\_1, a.toString) != -1 ) {

val j = findItemOrder(ex.\_1, a.toString)

var tmpRow = -1

tmpRow = ex.\_2 + 1

result(tmpRow)( getColumn(a.toString).ans ) = "S" + j.toString

}

}

if (pointPosition == tx.\_2.length - 1 ) {

val a = tx.\_3

var tmpRow = -1

tmpRow = ex.\_2 + 1

result(tmpRow)(getColumn(a).ans) = "r" + ( findRelationOrder( (tx.\_1,

tx.\_2.replace("·", "") ) ) )

}

if( tx.\_1 == relations(0).\_1 && tx.\_2 == relations(0).\_2 + "·" && tx.\_3 == "#" ) {

var tmpRow = -1

tmpRow = ex.\_2 + 1

result(tmpRow)( getColumn("#").ans ) = "acc"

}

}

for( ch <- localVN ) {

if( findItemOrder(ex.\_1, ch.toString) != -1 ) {

val gotoNumber = findItemOrder(ex.\_1, ch.toString)

var tmpRow = -1

tmpRow = ex.\_2 + 1

//A = ch

result(tmpRow)( getColumn(ch.toString).ans ) = gotoNumber.toString

}

}

}

result

}

## 3.5程序运行截图

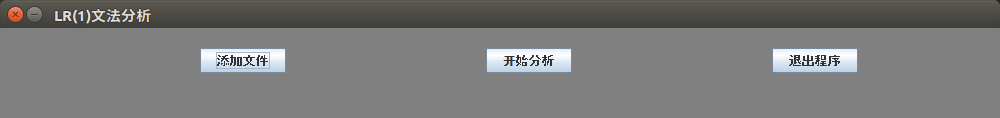


Figure 19 开始界面

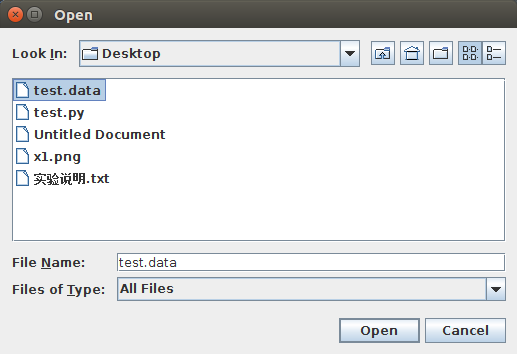


Figure 20 选择文件

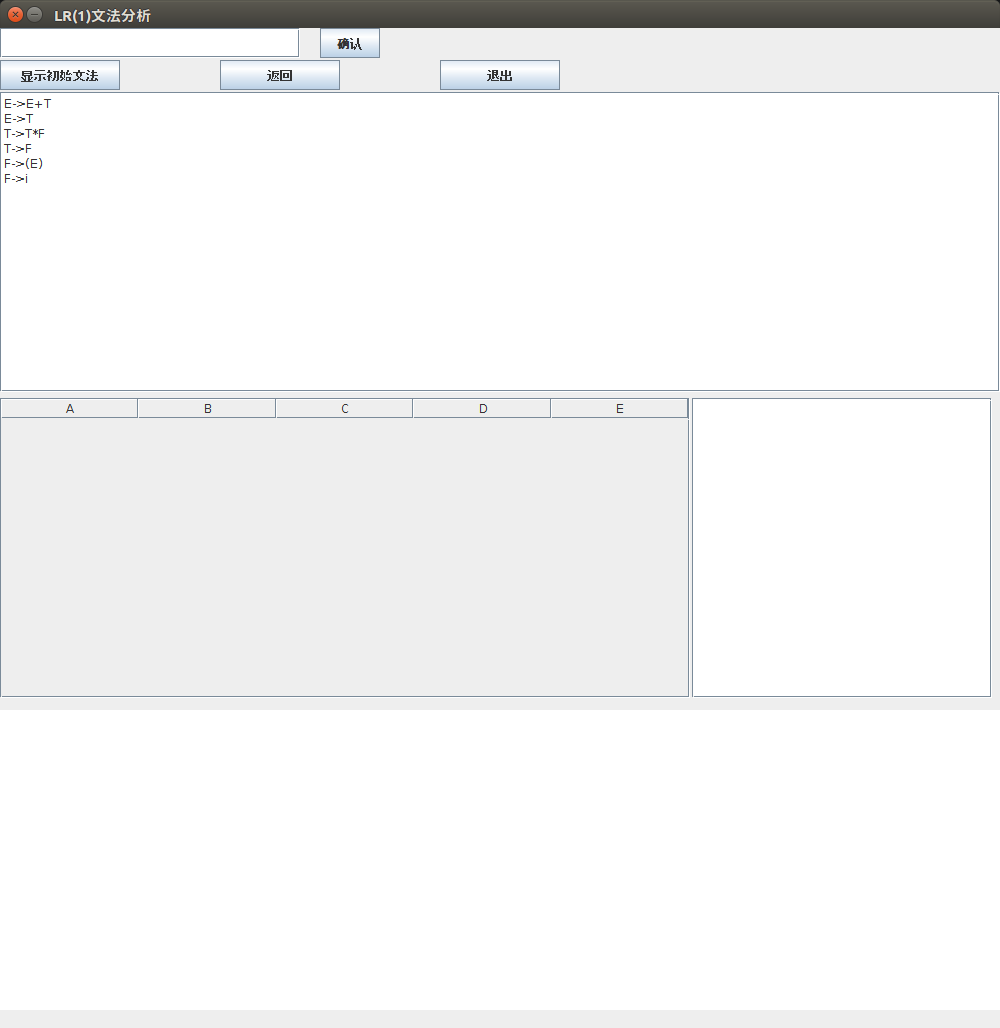


Figure 21 显示初始文法

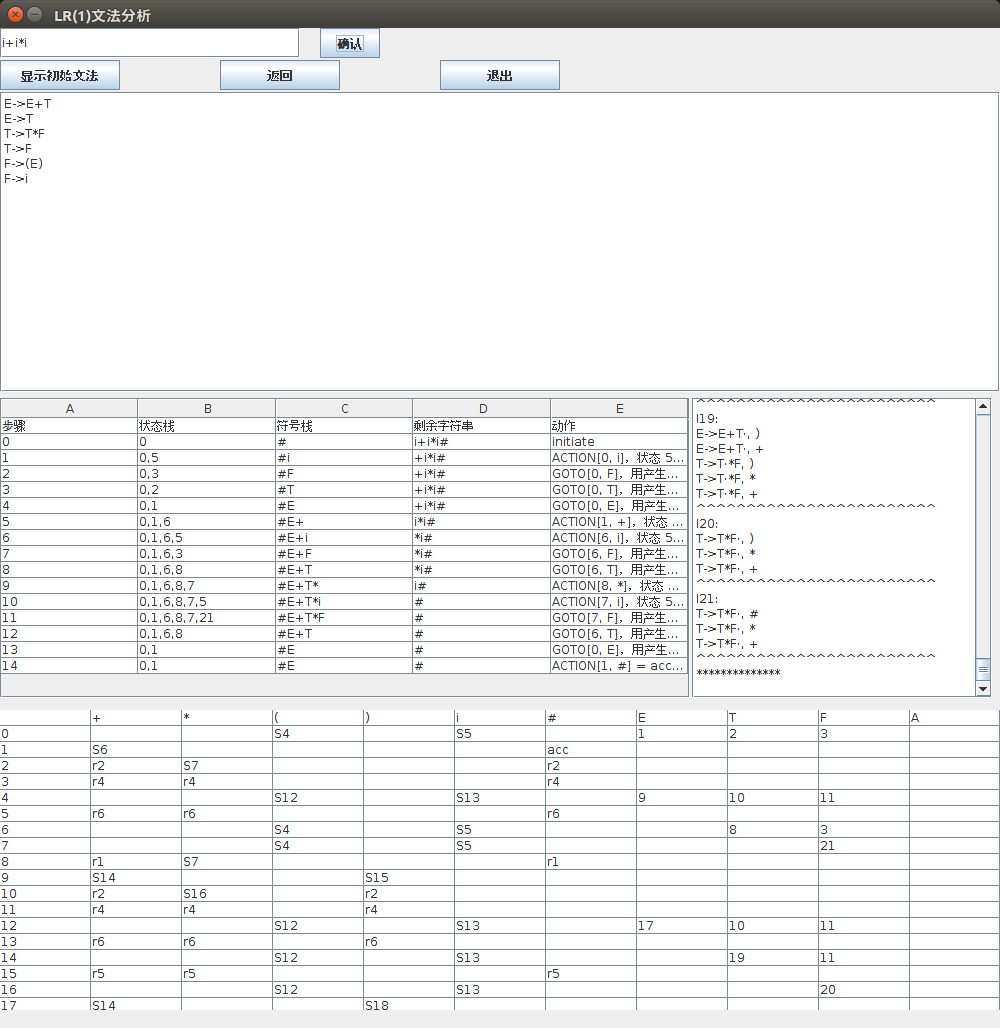


Figure 22 分析完成（输入表达死为“i+i\*i”）

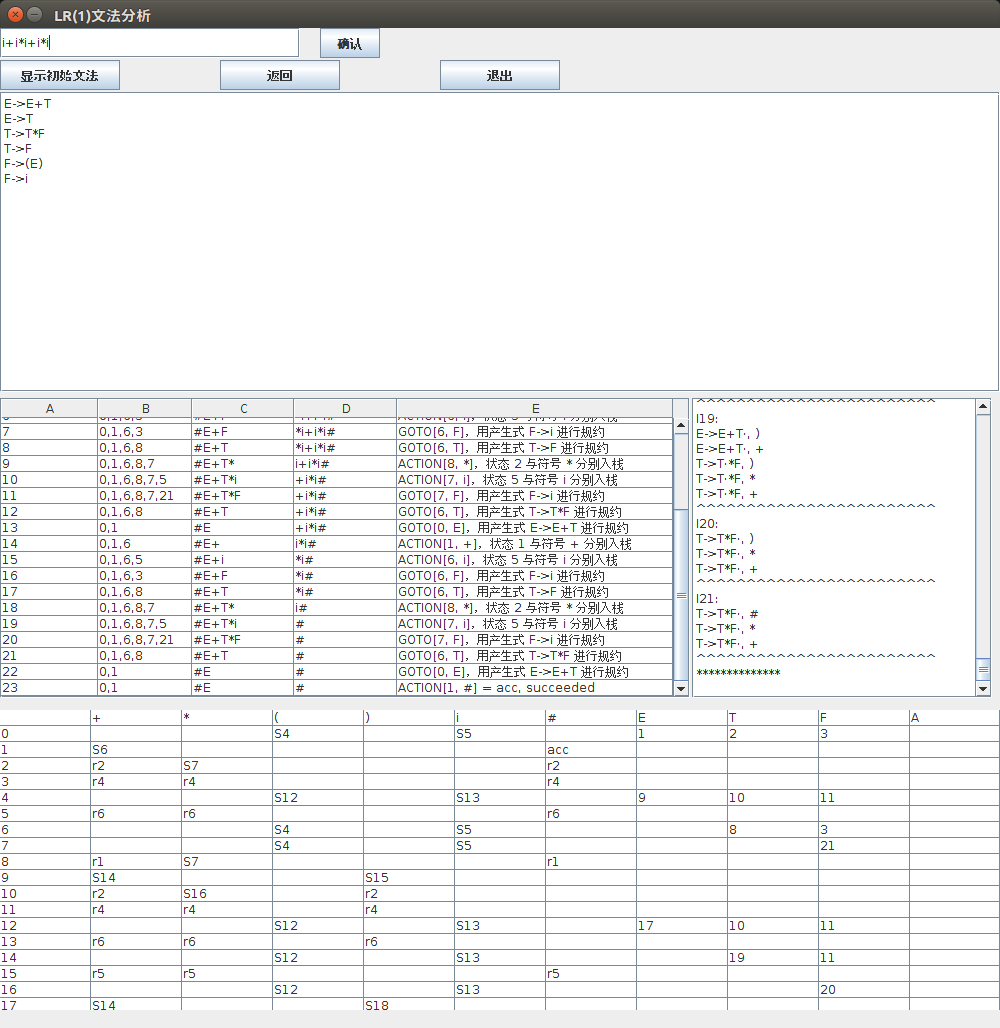


Figure 23 分析完成（输入表达死为“i+i\*i+i\*i”，单元格拉长）

# 参考文献

[1]Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers Principles, Techniques and Tools[M]. New York: Pearson Addison Wesley, 2006.

[2]陈火旺, 钱家骅, 孙永强. 程序设计语言编译原理（第3版）[M]. 北京：国防工业出版社, 1999.

# 附（实验代码链接）

实验1博客：

<https://blog.csdn.net/u25th_engineer/article/details/102458531>

实验2 github地址：

<https://github.com/25thengineer/Compile_Experiment_LL_1>

实验3 github地址：

<https://github.com/25thengineer/Compile_Experiment_LR_1>