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

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++。

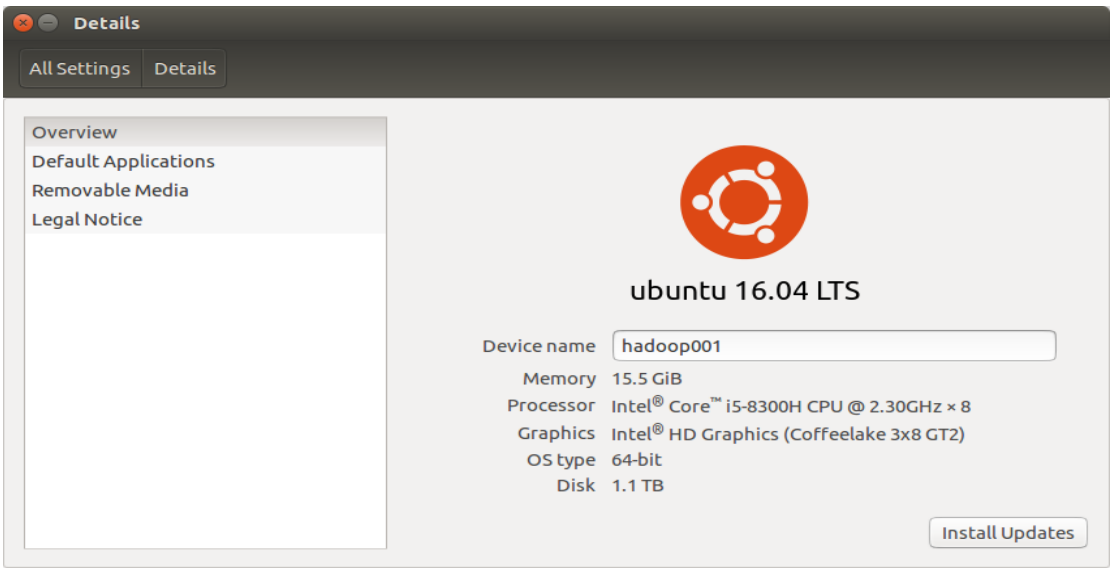


Figure 1 实验环境

```
hadoop001@hadoop001:~/Documents/code/C++/1/Untitled Folder$ g++ --version
g++ (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Figure 2 g++版本信息

```
hadoop001@hadoop001:~/Documents/code/C++/1/Untitled Folder$ vim --version
VIM - Vi IMproved 7.4 (2013 Aug 10, compiled Jun 07 2019 15:35:43)
Included patches: 1-1689
Extra patches: 8.0.0056
Modified by pkg-vim-maintainers@lists.alioth.debian.org
Compiled by pkg-vim-maintainers@lists.alioth.debian.org
```

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：

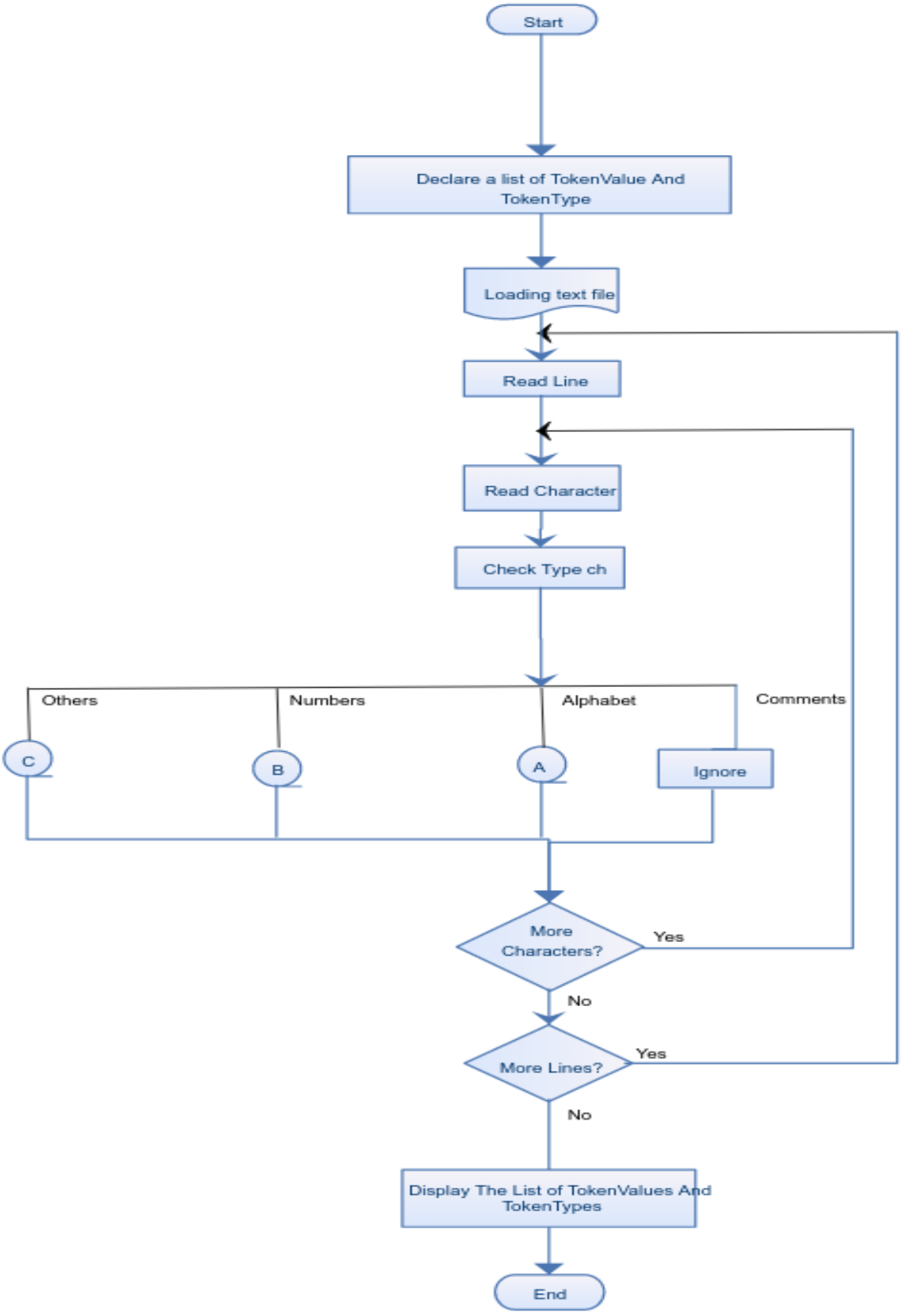


Figure 4 主控程序

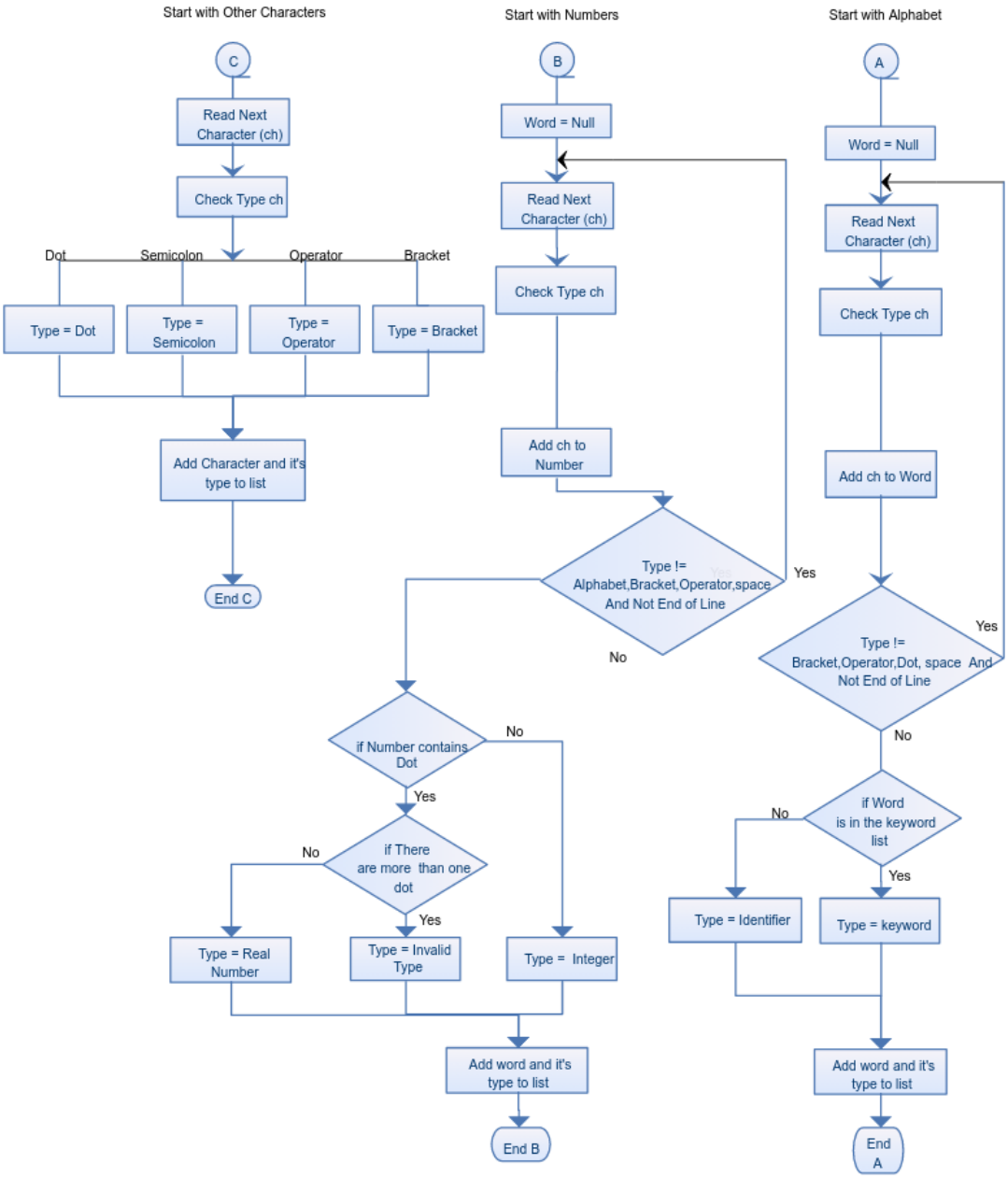


Figure 5 字符分析程序

状态转移图如图 6 所示。

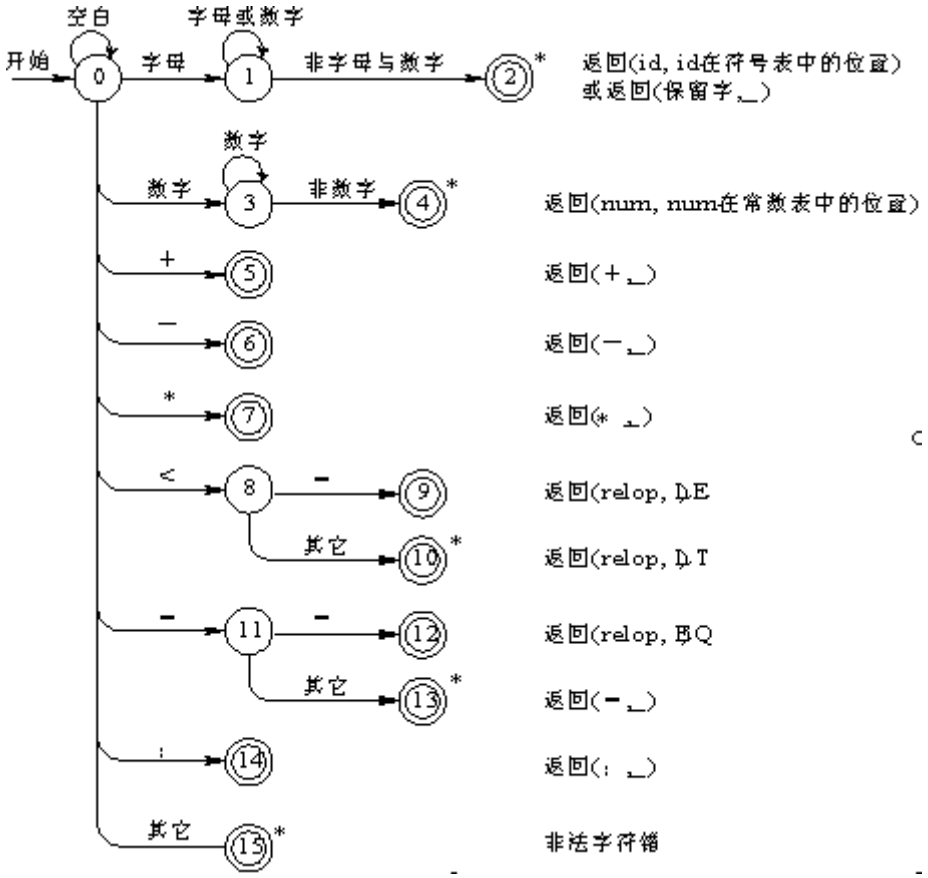


Figure 6 状态转移图

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
```

```
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

```
hadoop001@hadoop001:~/Documents/code/C++/1/Untitled Folder$ ./test.out
单词     二元序列          类型              位置
if       ( 1, if )         关键字            ( 1,1 )
i        ( 6, i )          标识符            ( 1,2 )
=        ( 4, = )          关系运算符        ( 1,3 )
0        ( 5, 0 )          无符号数          ( 1,4 )
then     ( 6, then )       标识符            ( 1,5 )
n        ( 6, n )          标识符            ( 1,6 )
++       ( 3, ++ )         算术运算符        ( 1,7 )
;        ( 2, ; )          分界符            ( 1,8 )
a        ( 6, a )          标识符            ( 2,1 )
<=       ( 4, <= )         关系运算符        ( 2,2 )
3b        error            error             ( 2,3 )
%        ( 3, % )          算术运算符        ( 2,4 )
)        ( 2, ) )          分界符            ( 2,5 )
;        ( 2, ; )          分界符            ( 2,6 )
```

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

$$A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$$

则 $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \Phi \quad (i \neq j)$

（3）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

$$FIRST(\alpha_i) \cap FOLLOW(A) \quad (i = 1, 2, , ..., n)$$

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：

$$A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$$

（1） 若 $a \in FIRST(\alpha_i)$，则指派 α_i 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 $FIRST(\alpha_i)$且 $a \in FOLLOW(A)$，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。

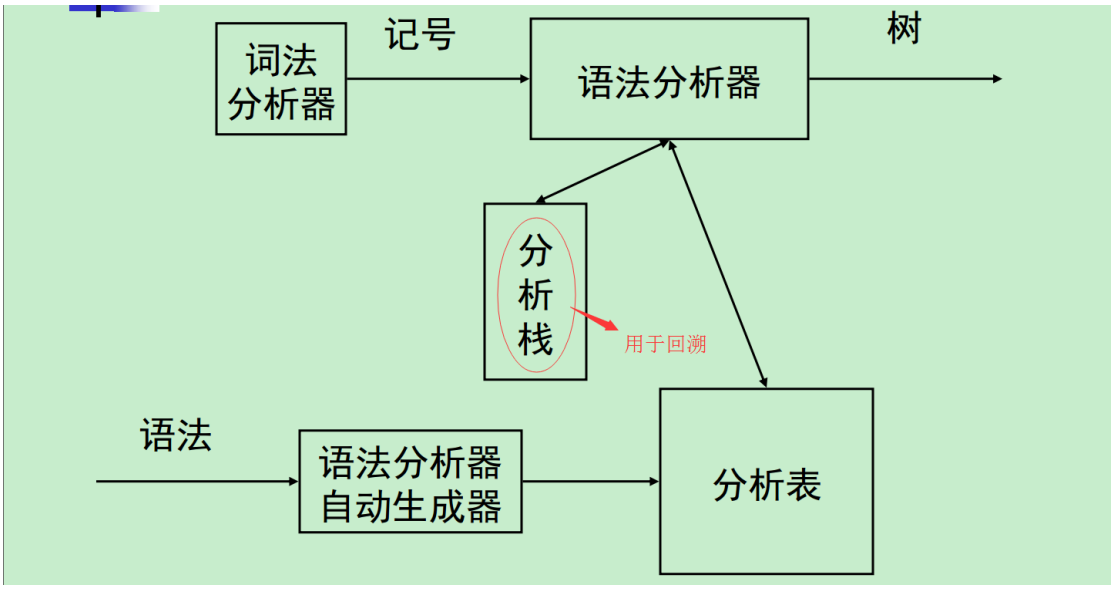


Figure 8 LL 文法分析架构

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

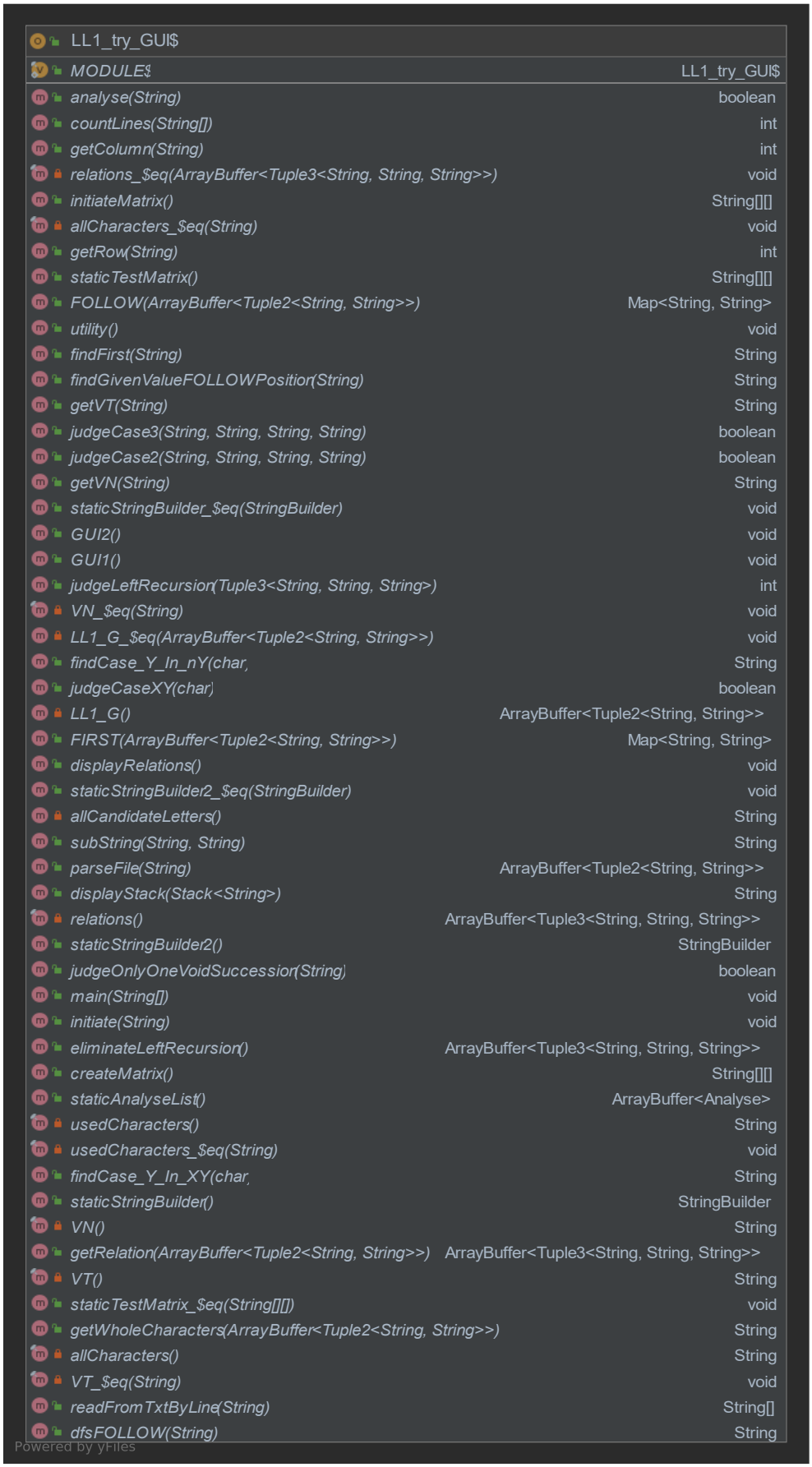


Figure 9 LL(1)文法分析实验代码完整函数

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]：

let a be the first symbol of w;
let X be the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if ($X = a$) pop the stack and **let** a be the next symbol of w;
    else if (X is a terminal) *error*();
    else if ($M[X, a]$ is an error entry) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
    }
    let X be the top stack symbol;
}

Figure 10 LL(1)预测分析总控形式化描述

FIRST 集构造流程[1]：

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = $\{X\}$.
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$), and so on.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

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 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
    }
```

```
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
            }
        }
```

```
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

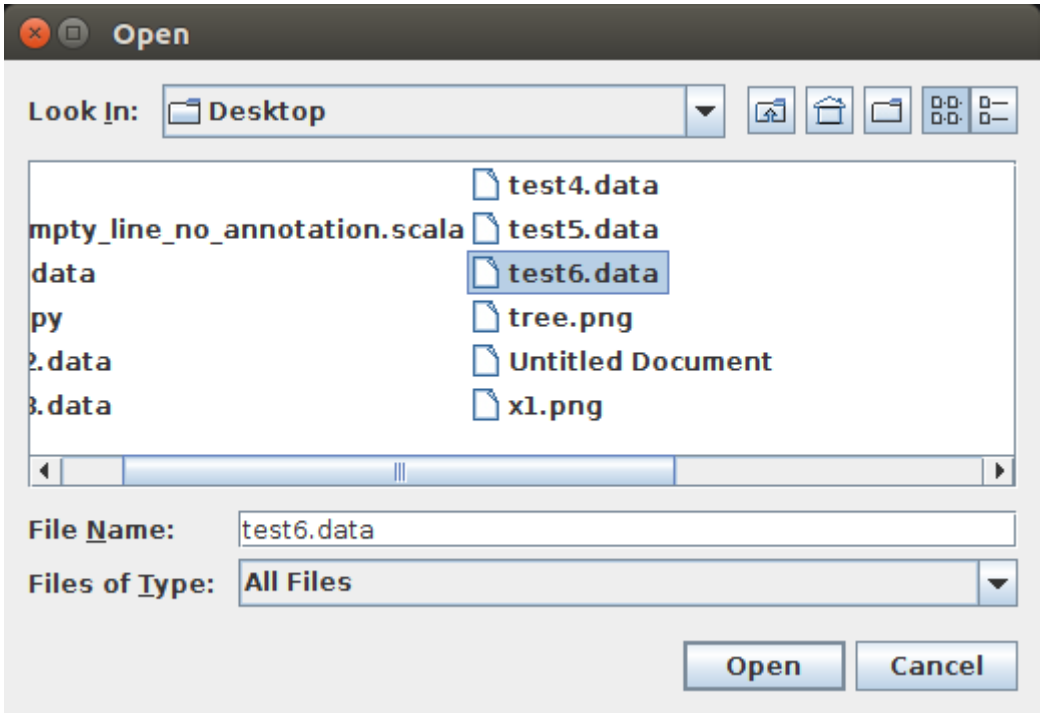


Figure 13 选择文件

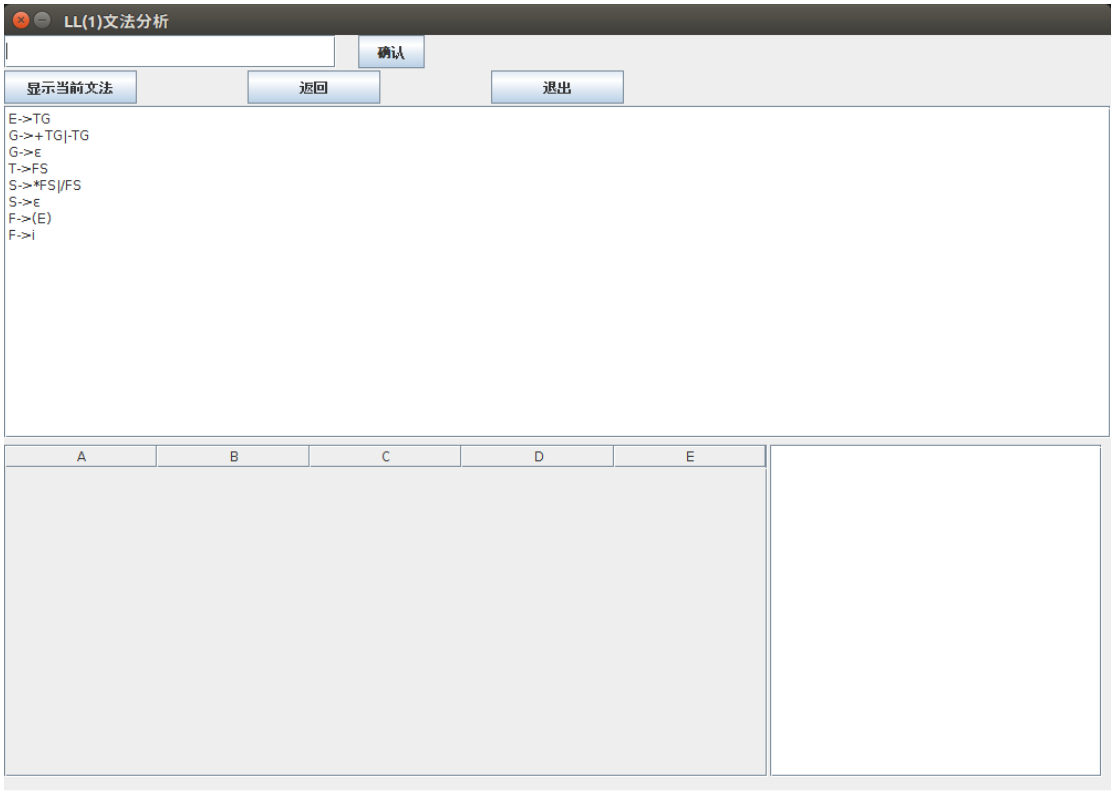
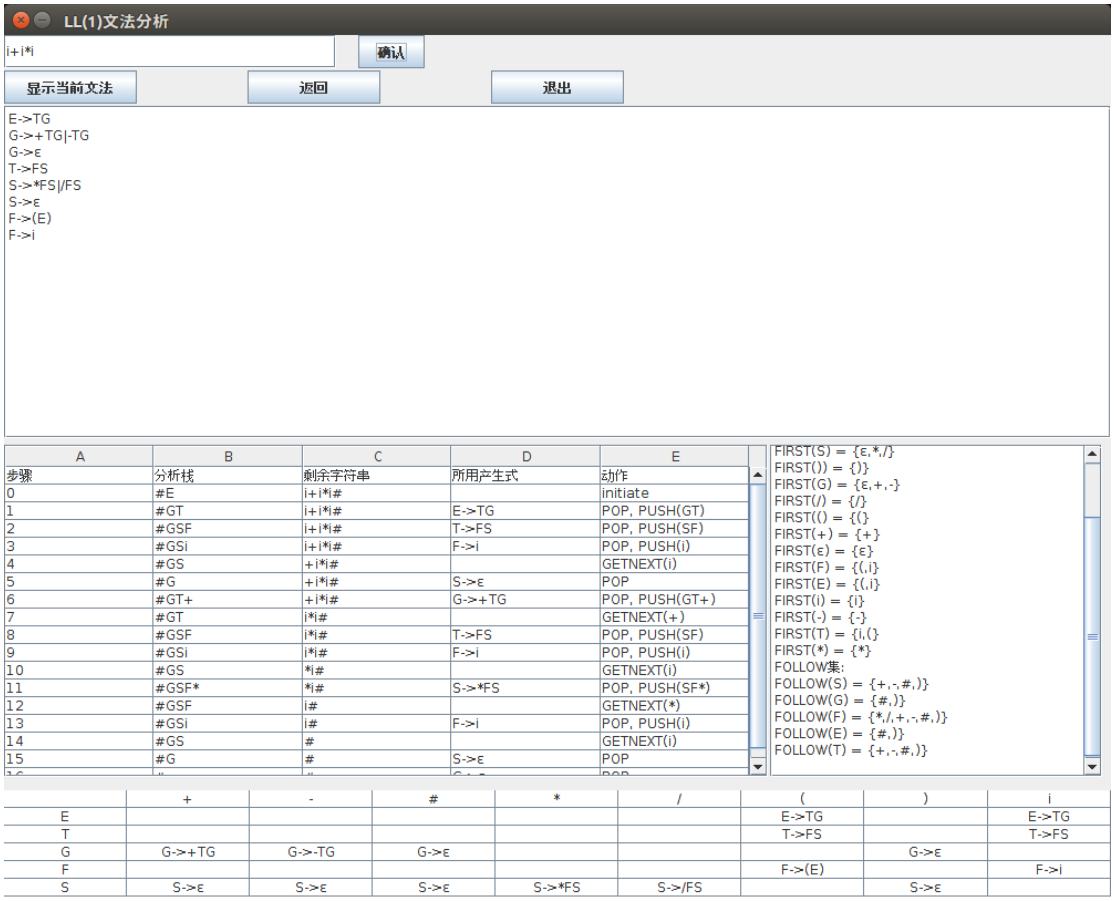


Figure 14 显示当前文法



A	B	C	D	E
步骤	分析栈	剩余字符串	所用产生式	动作
0	#E	i+i*i#		initiate
1	#GT	i+i*i#	E->TG	POP, PUSH(GT)
2	#GSF	i+i*i#	T->FS	POP, PUSH(SF)
3	#GSi	i+i*i#	F->i	POP, PUSH(i)
4	#GS	+i*i#		GETNEXT(i)
5	#G	+i*i#	S->ε	POP
6	#GT+	+i*i#	G->+TG	POP, PUSH(GT+)
7	#GT	i*i#		GETNEXT(+)
8	#GSF	i*i#	T->FS	POP, PUSH(SF)
9	#GSi	i*i#	F->i	POP, PUSH(i)
10	#GS	*i#		GETNEXT(i)
11	#GSF*	*i#	S->*FS	POP, PUSH(SF*)
12	#GSF	i#		GETNEXT(*)
13	#GSi	i#	F->i	POP, PUSH(i)
14	#GS	#		GETNEXT(i)
15	#G	#	S->ε	POP

	+	-	#	*	/	(	)	i
E						E->TG		E->TG
T						T->FS		T->FS
G	G->+TG	G->-TG	G->ε				G->ε	
F						F->(E)		F->i
S	S->ε	S->ε	S->ε	S->*FS	S->/FS		S->ε	

Figure 15 执行文法分析（输入表达式为“i+i*i”）

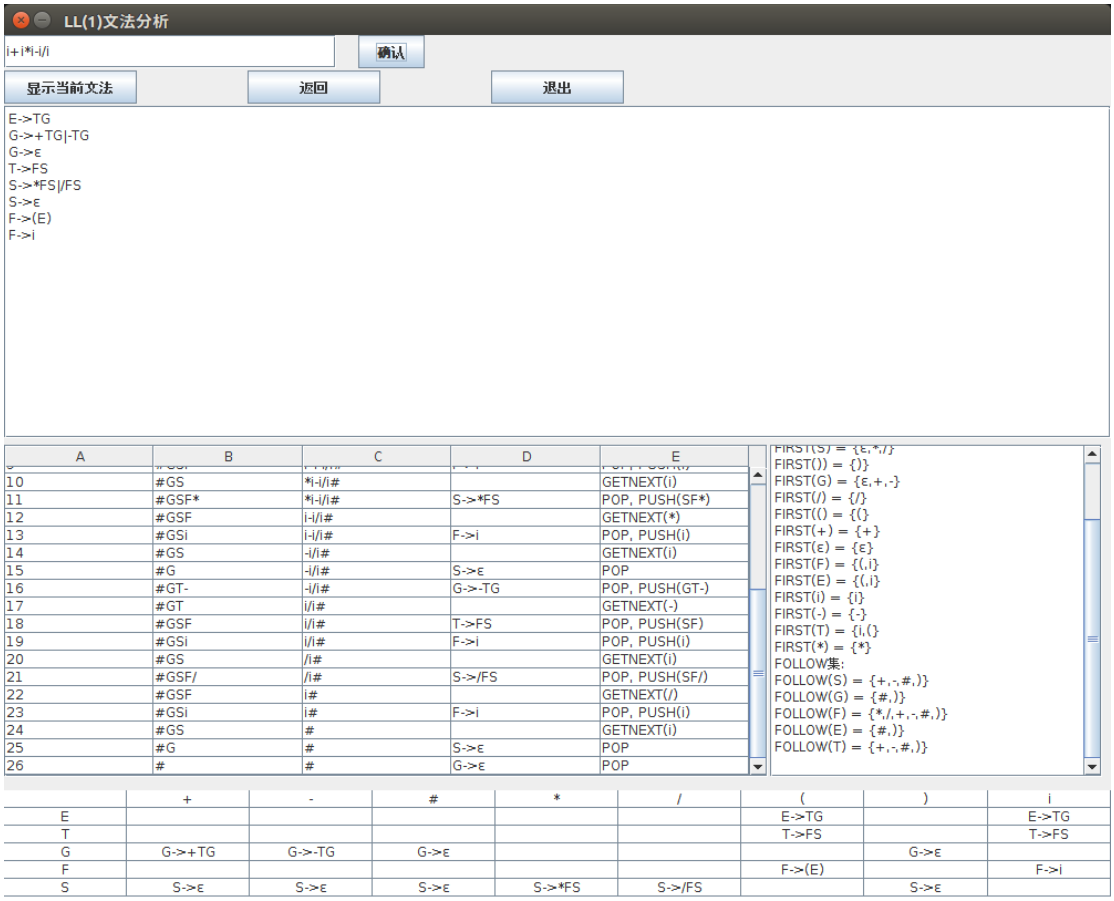


Figure 16 执行文法分析（输入表达式为“i+i*i-i/i”）

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, $a_1a_2...a_k$]，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 $a_1a_2...a_k$ 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, $a_1a_2...a_k$]有意义。对于任何移进或待约项目[A→α·β, $a_1a_2...a_k$]，β≠ε，搜索符串 $a_1a_2...a_k$ 没有作用。规约项目[A→α·, $a_1a_2...a_k$]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 $a_1a_2...a_k$ 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。

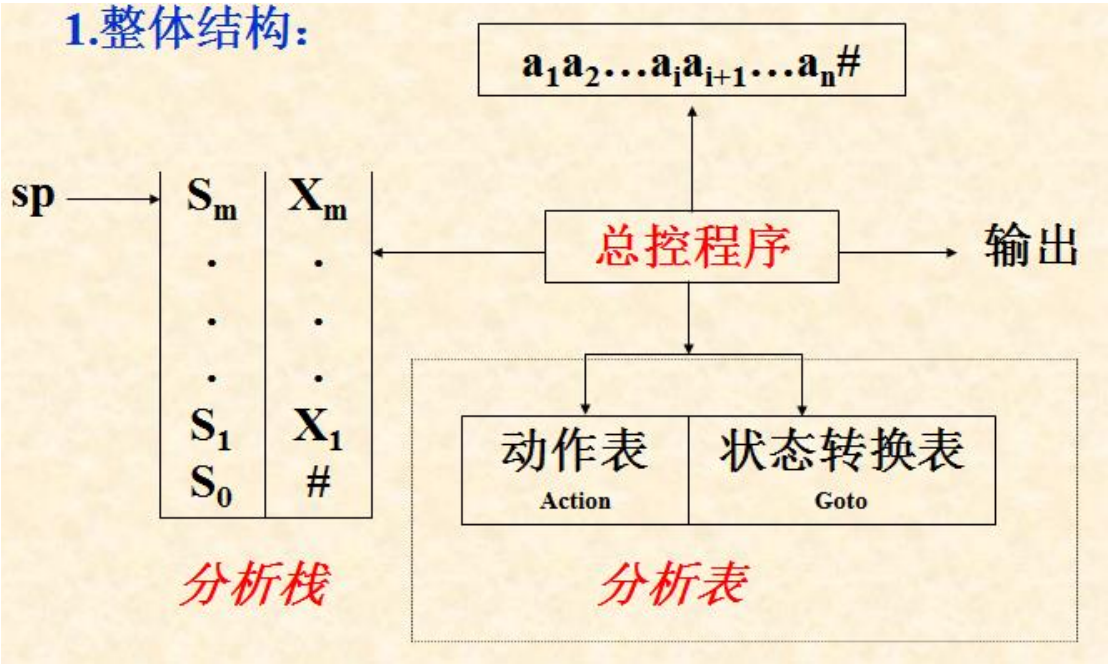


Figure 17 LR 文法分析架构

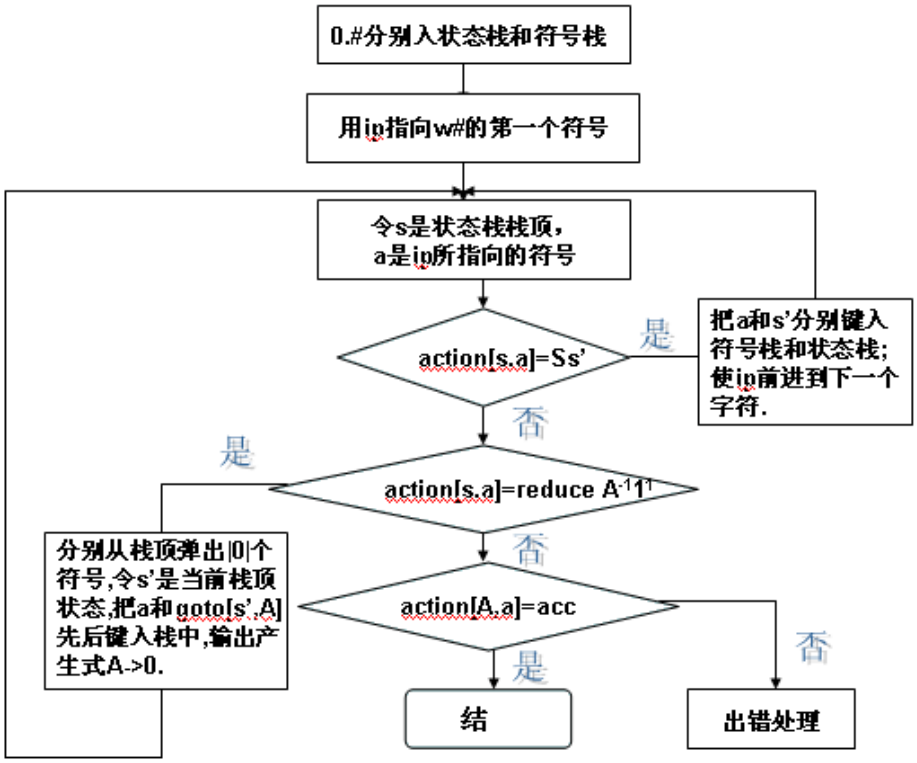


Figure 18 LR 文法分析流程图

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;
}
```

```
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->α X • β , a]的项目 | [A->α • X β , 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
    }
```

```
def getItemGroup(): Unit = {
        val Idx = ( relations(0)._1, "·" + relations(0)._2, "#" )
        val I0 = getClosure( ArrayBuffer(Idx) )
        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

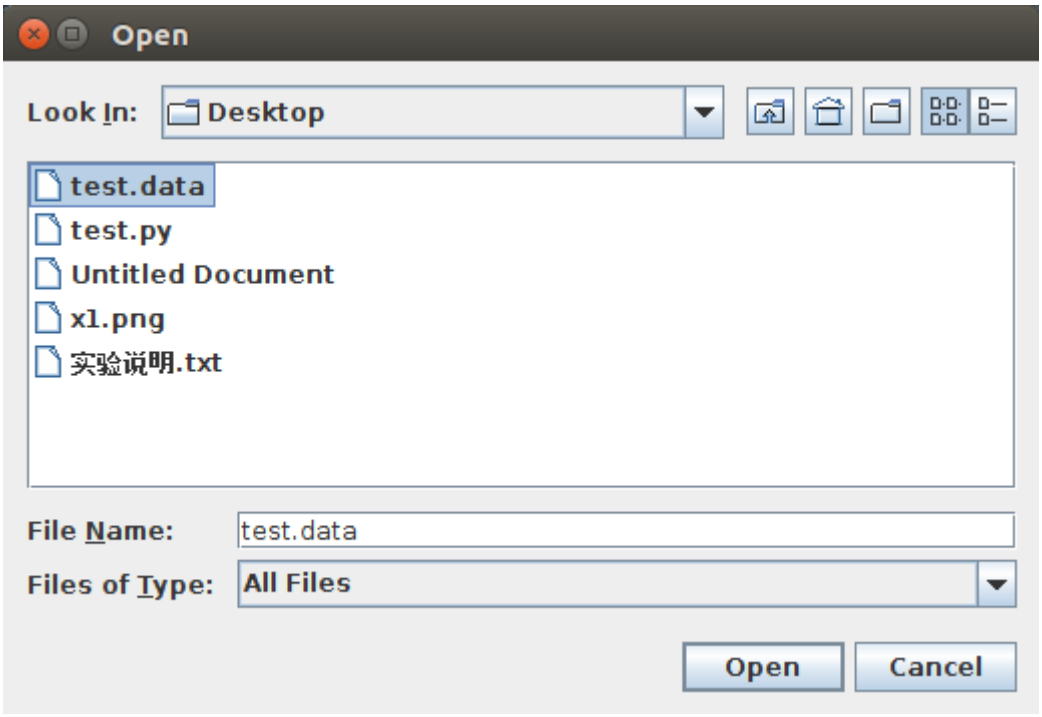


Figure 20 选择文件

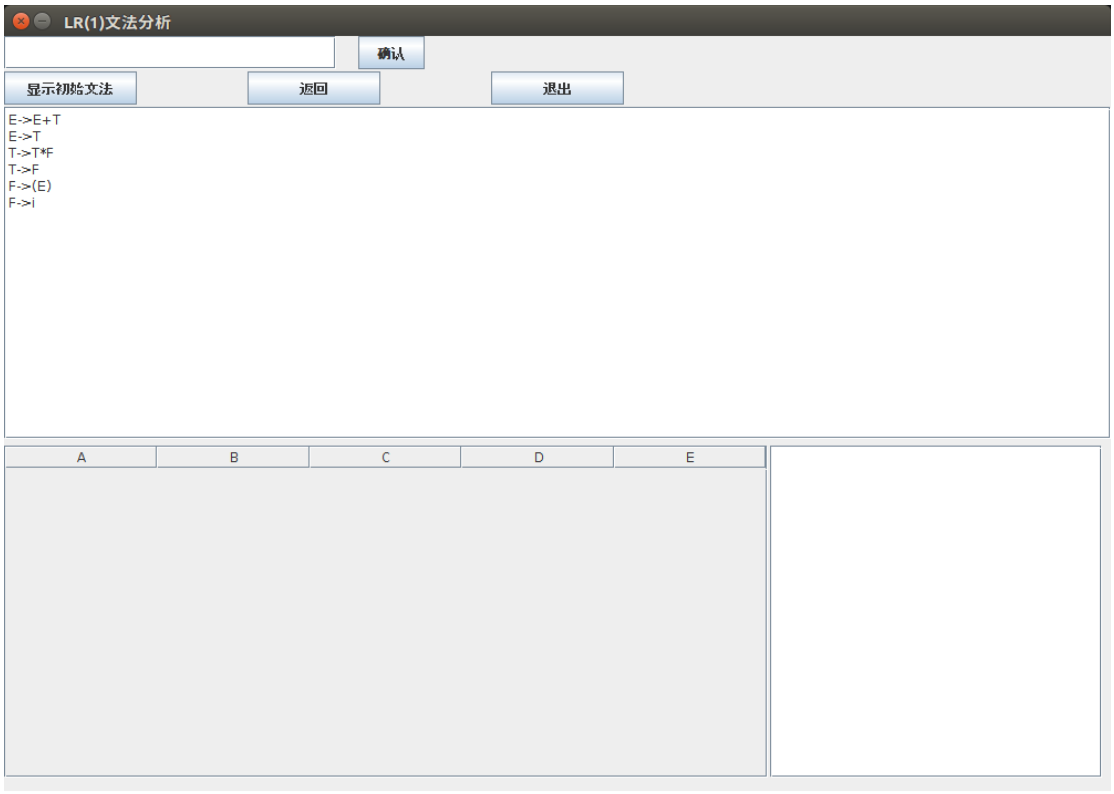


Figure 21 显示初始文法

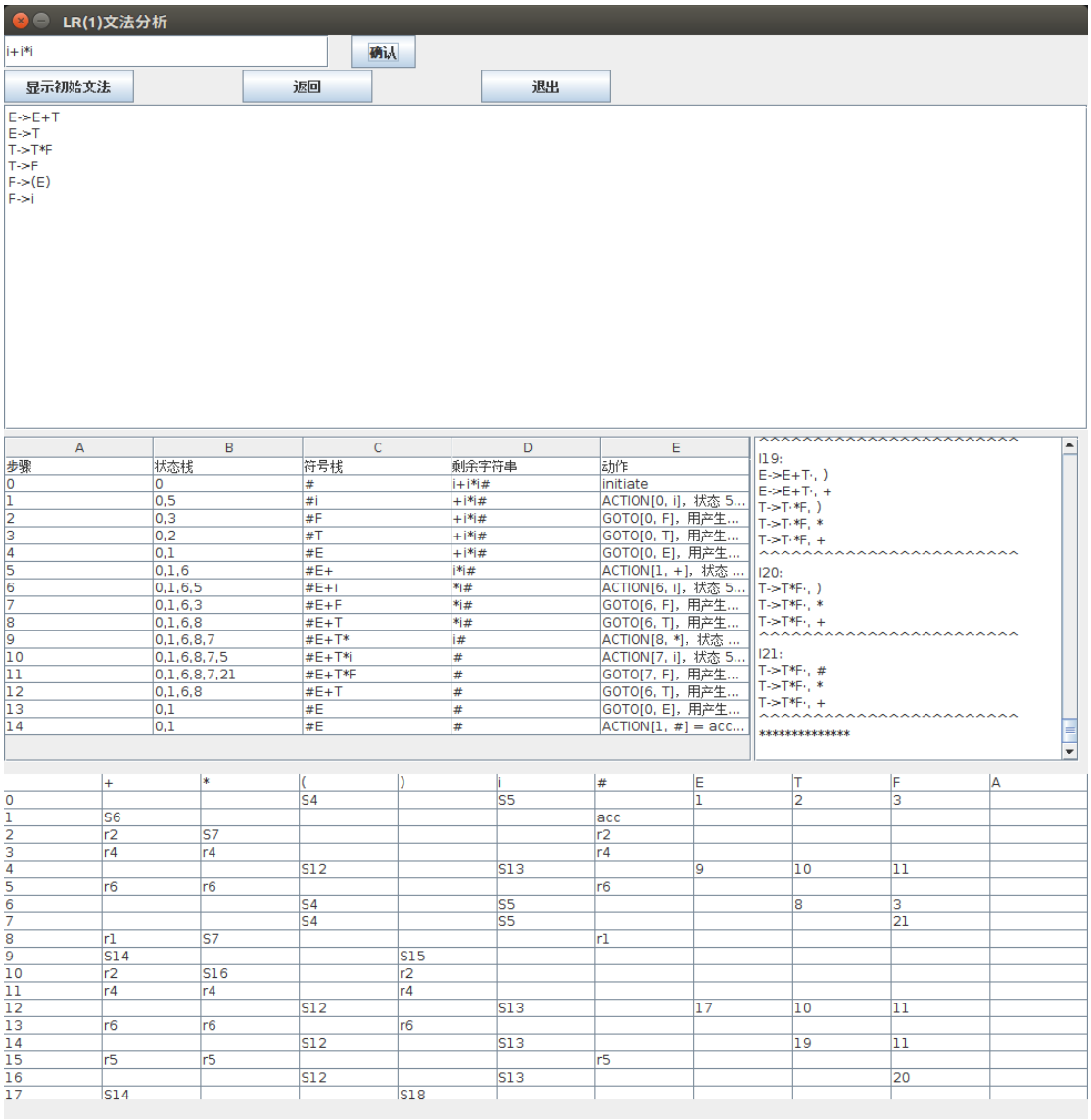


Figure 22 分析完成（输入表达死为“i+i*i”）

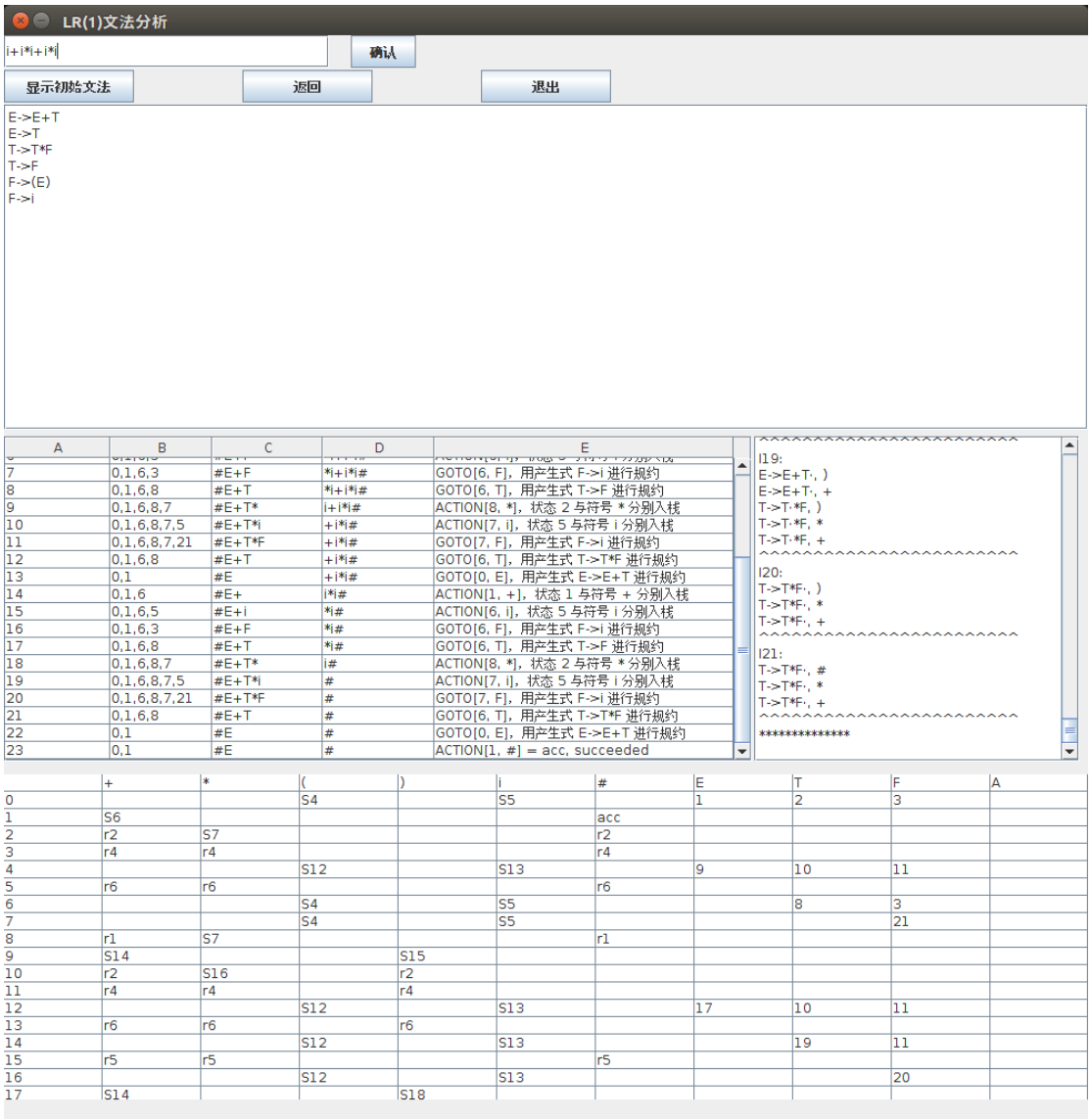


Figure 23 分析完成（输入表达死为“i+i*i+i*i”，单元格拉长）

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