编译原理课程设计

题　　目: 词法、语法、语义分析

姓　　名: 李南骏、高博文、席卓

学　　号: 19220208、11120227、9203020720

学　　院: 人工智能学院

专　　业: 计算机科学与技术

班　　级: 计科202

指导教师: 黄 芬

成绩评定:

2022年12月18日

目录

词法、语法、语义分析

摘要：编译程序是现代计算机系统的基本组成部分之一,而且多数计算机系统都配有不止一种高级语言的编译程序,对有些高级语言甚至配置了几个不同性能的编译程序。从功能上看，一个编译程序就是一个语言翻译程序。“编译原理”的知识结构贯穿程序设计语言、系统环境以及体系结构，能以相对独立的视角推向从软件到硬件协同的整机概念。本次课程设计主要就是根据上课和书中所学的理论知识，通过上机实践上机实践对相应的理论知识进行验证、并进一步体会编译程序进行词法分析和语法分析的过程并实现编译过程的可视化。

关键词：编译原理、词法分析、语法分析、JAVA

Lexical, Grammatical, Semantic analysis

Abstract: The compiler is one of the basic components of modern computer systems, and most computer systems are equipped with more than one high-level language compiler, some of the high-level languages even configured with several different performance of the compiler. Functionally, a compiler is a language translator. The knowledge structure of "compilation principle" runs through programming language, system environment and architecture, and can promote the concept of the whole system from software to hardware collaboration from a relatively independent perspective. This course design is mainly based on the theoretical knowledge learned in the class and the book, through the computer practice on the computer practice to verify the corresponding theoretical knowledge, and further understand the compiler for lexical analysis and grammar analysis process and realize the visualization of the compilation process.

Key words: Compiler principles, lexical analysis, syntax analysis, JAVA

引言：这次课程设计在允许使用面向对象语言开发的前提下，将题目给定的文法进行设计，分别实现了词法分析、语法分析和语义分析几个功能，并设计了用户操作界面。程序从文件读取字符串，并根据单词规则拆分成单词，随后进行语法分析和语法指导的翻译方案。

一、工作目标及介绍

1、本次编译原理课程的设计，旨在通过词法、语法分析工具的设计，加深我们对编辑原理课程理论的理解，更好地理解计算机科学技术这门课程。同时也锻炼我们的逻辑思考能力。

2、采用 SLR(1)语法分析方法，设计、开发所选文法描述语言的语法分析程序，加深课堂相关理论教学内容的理解，提高语法分析方法的实践能力；

3、参照书中相应文法的翻译方案，采用语法制导的翻译技术，编写语义分析程序，实现对任务 3 所选择文法的语义分析，输出四元式中间代码，加深课堂相关理论教学内容的理解，提高语法分析方法的实践能力。

4、用 JAVA 程序语言来设计这门课设。通过对理论知识的实践，加深对词法和语法分析的理解。编译器是现代计算机系统中最基本的组成部分之一，编译器是一种语言翻译程序，它把高级语言程序翻译成低级语言程序，编译器的重要性体现在它使大多数计算机用户不需要考虑与机器相关的繁琐细节，对理解编译器对软件开发人员能力的提高有很大帮助。

本次课设选题，我们选择的是文法三控制结构语句的词法分析+语法分析+语义分析，对于文法三是三个文法中较为困难的文法，包括算术表达式、复合语

句、赋值语句、布尔表达式等。

二、设计思路

2.1 编译程序概述

编译程序完成从源程序到目标程序的翻译工作，是一个复杂的整体的过程。从概念上来讲，一个编译程序的整个工作过程是划分成阶段进行的，每个阶段将源程序的一种表示形式转换成另一种表示形式，各个阶段进行的操作在逻辑上是紧密连接在一起的。一般一个编译过程划分成词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成六个阶段。

上述编译过程的六个阶段的任务，可以分别由六个模块完成，它们称作词法分析程序、语法分析程序、语义分析程序、中间代码生成程序、代码优化程序和目标代码生成程序。

2.1.1 系统框架图

图 2-1 系统框架图

图 2-1 编译程序结构框图

2.2 总体设计

对于课设整体的设计思路我大致遵循以下的流程。在进行理论分析后，大致分为三个阶段，第一是从文件中读取含有测试语句的文件，接下来就是对读进来的语句进行词法分析，将语句分割成单词并生成一个识别序列，将词法分割生成的序列作为输入串按照 SLR(1)分析表过程进行语法分析，同时对语法分析过的语句进行语法制导的翻译方案，即语义分析。

图 2-2 编译程序结构框图

首先是读取文件部分的设计，我设计读取文件的主要目的是为了方便进行批量的测试，而不用直接修改源程序。其次是规范编程，提供标准的输入输出测试，便于源程序的维护。将文法相应的测试语句(正确和不正确都可以)从文件中读取，对语句进行词法分析，若读入的词句有误，则程序会报错。

在做词法、语法分析之前需要进行 DFA 构造，并根据绘制的 DFA 绘制 SLR(1)分析表，并且根据文法以及运算符优先级来解决冲突，生成无冲突的 SLR(1)分析表。

在词法分析过程中，需要根据读入的文件进行单词的分割，最后根据ACTION 表中的终结符比如：begin、end、if、then 等与分割的单词进行比对，来生成语句的对应终结符的数字串，作为语法分析的输入串。

在语法分析过程中，需要用到 SLR(1)分析表、对词法分析的输入串进行分析，步骤就是书上和上课讲过的理论步骤，但是实际执行的时候还是有一点难度的。语法分析的简单设计步骤就是根据 SLR(1)分析表和输入串使用状态栈、符号栈、以及 ACTION、GOTO 等来识别活前缀，并及时即可规约的活前缀进行规约。这也是我们构造的 DFA 的功能就是识别活前缀。最后对输出输入串进行识别和分析，识别成功或者报错。

语义分析我们小组在理论知识学习完成后，未能成功编程实现可视化，这是我们小组最大的遗憾。

2.3 词法分析设计

词法分析程序的功能是从左到右逐字逐句扫描源程序，识别来自字符流中的源程序中的单词，实现一些功能，如：过滤源程序中的注释和空格，记录新输入字符行的信息，识别关键字、标识符、常数、运算符、界符等，辅助完成语法分析，将所发现的错误信息与源程序联系起来。在本次课程设计中，对关键字、标识符、界符、运算符、数字等进行识别。

其设计思想如下：程序可以实现从文本文件中逐行读取字符串，对每一行的字符串进行分割，将字符串转换为数组，对每个字符逐个进行判断，并将其字符按照识别规则分割为独立的字符或者组成特定的字符串，比如组成标识符和begin、end 等，使用集合类来存放暂时存放识别组成的字符或字符串，并将其传递到识别函数 compare 中，识别函数可以判断这是关键字、标识符、操作符等等，然后将识别的类别序号保存在一个 orderList 集合中，每个类别对应于一个数字，随后对输入串的处理也按照这个类别进行语法分析。在此基础上，设计词法纠错的功能，识别出错误的符号，或以数字开头的标识符进行报错。关键在于字符的准确分割和判断。

2.3.1 词法分析设计图

图 2-3 词法分析步骤

2.3.2 关键字和标识符

程序设计时因输入序列较长，因此我们组选择，将关键字和标识符转化为数字，关键字和标识符所对应的数字如下：

0:begin
1:end
2:if
3:then
4:id
5:;
6:=
7:+
8:*
9:-
10:(
11:)
12:or
13:and
14:not
15:rop
16:true
17:false
18:#
19：S
20：C
21:L
22:A
23:B
24:K
25:E

2.2.3 相关词法规则

<标识符>::=<字母>
<标识符>::=<标识符><字母>
<标识符>::=<标识符><数字>
<常量>::＝<无符号整数>
<无符号整数>::=<数字序列>
<数字序列>::=<数字序列><数字>
<数字序列>::=<数字>
<字母>::=a|b|c|……|x|y|z
<数字>::=0|1|2|3|4|5|6|7|8|9
<加法运算符>::＝+|-
<乘法运算符>::＝*
<关系运算符>::＝<|>|!=|>=|<=|==
<赋值运算符>::＝:=

<分界符>::=;|(|)
<保留字>::=begin|end|if|then|or|and|not|true|false|id

词法分析阶段是编译过程的第一个阶段，是编译的基础。这个阶段的任务是从左到右一个字符一个字符地读入源程序，即对构成源程序的字符流进行扫描然后根据构词规则识别单词(也称单词符号或符号)。词法分析程序实现这个任务。

2.4 语法分析

语法分析的程序功能就是将词法分析得到的数字序列(输入串)按照构造好的 SLR(1)表的分析进行分析。

原理如下：首先有状态栈和符号栈分别存放的是状态序列和符号序列，输入串序列跟随移进规约过程不断的减少，还有 ACTION 和 GOTO 表示当前步骤的动作和将要转向的状态。

首先读状态栈的栈顶元素，作为 SLR(1)分析表的横坐标，然后读输入串的首个元素(区别栈顶元素)，作为 SLR(1)分析表的纵坐标，然后去读 SLR(1)分析表，如果读到的是 0 那么就表示当前输出串的首个元素移进到符号栈中，组成可识别活前缀，如果读到的是 100 那么表示当前符号栈中已经有了可规约的活前缀，只需将按照产生式将其规约即可。

同时规约之后查找 SLR(1)分析表可以得到规约后转移到的状态，加入到状态栈，成为栈顶。如果读到的是空(SLR(1)表中没有对应的项)那么此时程序将报错，分析错误程序种植。重复上述步骤即可完成基于 SLR(1)分析表的语法分析。程序设计的时候也是大体按照这样的理论思想来设计的，有所不同和较为困难的是在规约的时候，我们手工状态下可以知道选择哪个产生式进行规约以及对应的状态栈和符号栈需要出栈几位，但是程序是不知道的，需要我们在程序设计的时候按照每条产生式来判断规约的时候出栈几次。

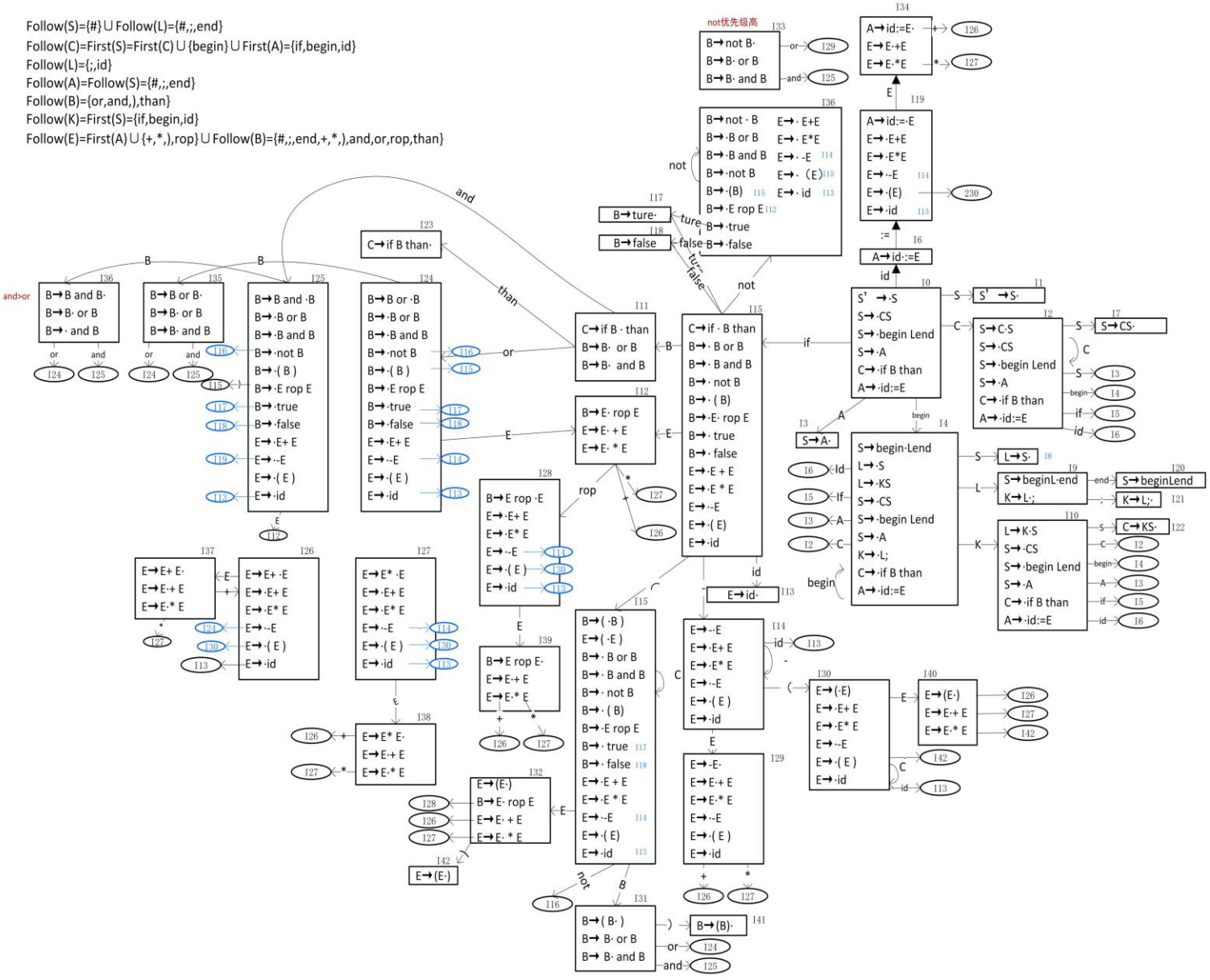
2.4.1 语法分析流程图

图 2-4 语法分析流程图

2.5 文法产生式以及 DFA 状态

S'->S	0
S->CS	1
S->begin L end	2
S->A	3
C->if B then	4
L->S	5
L->K S	6
K->L;	7
A->id:=E	8
E->E+E	9
E->E*E	10
E->-E	11
E->(E)	12
E->id	13
B->B or B	14
B-> B and B	15
B->not B	16
B->(B)	17
B->E rop E	18
B->true	19
B->false	20

其中：S-语句,L 复合语句,A 赋值语句,E 算术表达式,B 布尔表达式



I0:

Sta S' -> .S
S -> .CS
S -> .beginLend
S -> .A
C -> .ifBthen
A -> .id:=E

I1:

S' -> S.

I2:

S -> C.S
S -> .CS
S -> .beginLend
S -> .A
C -> .ifBthen
A -> .id:=E

I3:

S -> A.

I4:

S -> begin.Lend
L -> .S
L -> .KS
S -> .CS
S -> .beginLend
S -> .A
K -> .L;

C -> .ifBthen
A -> .id:=E

I5:
C -> if.Bthen
B -> .BorB
B -> .BandB
B -> .notB
B -> .(B)

I6:
A -> id.:=E

I7:
S -> CS.

I8:
L -> S.

I9:
S -> beginL.end
K -> L.;

I10:
L -> K.S
S -> .CS
S -> .beginLend
S -> .A
C -> .ifBthen
A -> .id:=E

I11:
C -> ifB.then
B -> B.orB
B -> B.andB

I12:
B -> E.ropE
E -> E.+E
E -> E.*E

I13:
E -> id.

I14:
E -> -.E
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I15:
B -> (.B)
E -> (.E)
B -> .BorB
B -> .BandB
B -> .notB
B -> .(B)
B -> .EropE
B -> .true
B -> .false
E -> .E+E
B -> .EropE
B -> .true
B -> .false
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I16:
B -> not.B
B -> .BorB
B -> .BandB
B -> .notB
B -> .(B)
B -> .EropE
B -> .true
B -> .false
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I17:
B -> true.

I18:
B -> false.

I19:
A -> id:=.E
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I20:
S -> beginLend.

I21:
K -> L;.

I22:
L -> KS.

I23:
C -> ifBthen.

I24:
B -> Bor.B
B -> .BorB
B -> .BandB
B -> .notB
B -> .(B)
B -> .EropE

B -> .true
B -> .false
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I25:

B -> Band.B
B -> .BorB
B -> .BandB
B -> .notB
B -> .(B)
B -> .EropE
B -> .true
B -> .false
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I26:

E -> E+.E
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I27:

E -> E*.E
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I28:

B -> Erop.E
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I29:

E -> -E.
E -> E.+E
E -> E.*E

I30:

E -> (.E)
E -> .E+E
E -> .E*E
E -> .-E
E -> .(E)
E -> .id

I31:

B -> (B.)
B -> B.orB
B -> B.andB

I32:

E -> (E.)
B -> E.ropE
E -> E.+E
E -> E.*E

I33:

B -> notB.
B -> B.orB
B -> B.andB

I34:

A -> id:=E.
E -> E.+E
E -> E.*E

I35:

B -> BorB.
B -> B.orB
B -> B.andB

I36:

B -> BandB.
B -> B.orB
B -> B.andB

I37:

E -> E+E.
E -> E.+E
E -> E.*E

I38:

E -> E*E.
E -> E.+E
E -> E.*E

I39:

B -> EropE.
E -> E.+E
E -> E.*E

I40:

E -> (E.)
E -> E.+E
E -> E.*E

I41:

B -> (B).

I42:

E -> (E).

2.6 SLR(1)分析表

2.6.1 ACTION 表

	begin	end	if	then	id	;	:=	+	*	-	(	)	or	and	not	rop	true	false	#
9		s20				s21													
10	s4		s5		s6														
11				s23									s24	s25					
12								s26	s27							s28			
13		r13		r13		r13		r13	r13			r13	r13	r13		r13			r13
14					s13					s14	s30								
15					s13					s14	s15				s16		s17	s18	
16					s13					s14	s15				s16		s17	s18	
17				r19								r19	r19	r19					
18				r20								r20	r20	r20					
19					s13					s14	s30								
20		r2				r2													r2
21	r7		r7		r7														
22		r6				r6													
23	r4		r4		r4														
24					s13					s14	s15				s16		s17	s18	
25					s13					s14	s15				s16		s17	s18	
26					s13					s14	s30								
27					s13					s14	s30								
28					s13					s14	s30								
29		r11		r11		r11		r11	r11			r11	r11	r11		r11			r11
30					s13					s14	s30								
31												s41	s24	s25					
32								s26	s27			s42				s28			
33				r16								r16	r16	r16					
34		r8				r8		s26	s27										r8
35				r14								r14	r14	s25					
36				r15								r15	r15	r15					
37		r9		r9		r9		r9	s27			r9	r9	r9		r9			r9
38		r10		r10		r10		r10	r10			r10	r10	r10		r10			r10
39				r18				s26	s27			r18	r18	r18					
40								s26	s27			s42							
41				r17								r17	r17	r17					
42		r12		r12		r12		r12	r12			r12	r12	r12		r12			r12

图 2-6 ACTION 表

2.6.2 GOTO 表

S	C	L	A	B	K	E
1	2		3			
7	2		3			
8	2	9	3		10	
				11		12
22	2		3			
						29
				31		32
				33		12
						34
				35		12
				36		12
						37
						38
						39
						40

图 2-6 GOTO 表

三、核心代码

3.1 文件读取模块(伪代码)

为方便修改和测试，我选择将测试的内容放在文件中，程序从文件中读取测试内容进行测试。技术的实现主要是用 JAVA 的 File 相关类实现对文件的读取，并对文件中的内容进行词法分析。

```
public void readfile() { //  读文件
        try {
            创建文件 1 用来输出文件内容
            创建文件 2 用来进行词法分析
            BufferedReader br = new BufferedReader(fr),br1=new BufferedReader(fr1);
            String s,s1;
            System.out.println("当前分析语句为:");
            while (文件 1 没有读完) {
                读出文件
            }
            while (文件二没有读完的时候) { //  读取一行不为空
                lexi_analysis(s);//  进行词法分析
```

```
            }
            关闭文件 1
            关闭文件 2
        } catch (Exception e) {
            // TODO 自动生成的 catch 块
            e.printStackTrace();
        }
    }
```

3.2 词法分析模块(伪代码)

词法分析的主要目的就是将读入的语句，进行单词的分割，并与 SLR(1)表中的保留字进行对比确定属于文法中的那个成分，比如 begin a>b end# 就要把 begin end 等分割出来，然后就行对比确定属于 SLR(1)分析表中的那个部分，最后得到一个分析后的单词序列。

```
public void lexi_analysis(String s) { // 词法分析
        将字符串转为字符数组
        List<Character> list = new ArrayList<Character>();//建立临时 List 用于存放字符串
        for (对每个字符进行分析) {
            if (这个字符是字母);// 加入 list
                // 如果字母后面是数字字母下划线_ 美元符号$ 组成的 继续拼接
                while (字母后面是数字、字母下划线等合法符号) {
                    加入 list
                } // 循环结束 得到字母开头的标识符
                --i;
                调用 compare 函数，与已知标识符进行对比
                清空 list ，注意每执行一次操作都需要清空一次 List,便于再次读取存储
            } else if (是数字) {
                while (后面还是数字)) { // 数字后面还是数字合法
                    list.add(a[i++]);// 加入 list
                } // 循环结束 要么是长度到了 要么就是后面不是数字
                if (数字后面是字母) {
                    // 数字 后面是字母 是非法的
                    报错;
                } else {
                    --i;
                    调用 compare 函数，与已知标识符进行对比
```

```
            清空 list
        }


} else if (字符是'#') {
        加入 list
        调用 compare 函数，与已知标识符进行对比
        清空 list
} else if (字符是'>') { // >  号
        加入 list
        if (字符是  '=') {
            将这个  =  加入
        }
        调用 compare 函数，与已知标识符进行对比
        清空 list
} else if (字符是  '<') {
        加入 list
        if (如果后面是  =  号) {
            将这个  =  加入
        }
        调用 compare 函数，与已知标识符进行对比
        清空 list
} else if (字符是'!') {
        if (a[i + 1] == '=') {//  是  !=
            ！ 加入 list
            =  加入 list
        }
        调用 compare 函数，与已知标识符进行对比
        清空 list
} else if (字符是  '=') {
        if (a[i + 1] == '=') {//  是  ==
            =  加入 list
            =  加入 list
        }
        调用 compare 函数，与已知标识符进行对比
```

```
        清空 list
    } else if (字符是':') { //  识别赋值号
        if (a[i + 1] == '=') {//  是  :=
            :  加入 list
            =  加入 list
        }
        调用 compare 函数，与已知标识符进行对比
        清空 list
    } else if (字符是  ';') {
        加入 list
        调用 compare 函数，与已知标识符进行对比
        清空 list
    } else if (字符是  '+') {
        加入 list
        调用 compare 函数，与已知标识符进行对比
        清空 list
    } else if (字符是  '*') {
        加入 list
        调用 compare 函数，与已知标识符进行对比
        清空 list
    } else if (字符是'-') {
        加入 list
        调用 compare 函数，与已知标识符进行对比
        清空 list
    } else if (字符是  '(') {
        加入 list
        调用 compare 函数，与已知标识符进行对比
        清空 list
    } else if (字符是  ')') {
        加入 list
        调用 compare 函数，与已知标识符进行对比
        清空 list
    } else if (字符是  '@' || a[i] == '$' || a[i] == '%' || a[i] == '^' || a[i] == '\\' || a[i] == '/'
            || a[i] == '.' || a[i] == ','等其他字符) {
```

```
                报错
                System.exit(0);
            }
        }
    }
```

3.3 语法分析（伪代码）

语法分析就是按照书上的理论知识，来编程实现，主要是进行规约步骤的时候较为复杂。需要确定适用哪条产生式进行分析。

```
public void gram_analysis() { //  语法分析
        状态栈中栈顶元素作为遍历 ACTION 表的 x;
        输入串的第一个符号(数字)作为遍历 ACTION 表的 y;
        while (不是 acc 接受) { //不是 acc 的时候循环
            状态栈中栈顶元素作为遍历 ACTION 表的 x;
            输入串的第一个符号(数字)作为遍历 ACTION 表的 y;
            显示状态栈
            显示符号栈
            显示剩余串
            显示 ACTION 动作
            if (ACTION[x][y] < 0) { //  表示规约  需要知道按照哪条产生式规约  几个符号出栈
                找到回退几位  并出栈
                显示 goto 动作
                规约后的非终结符加入符号栈
            } else { // ACTION[x][y]>0  移进项目
                新状态进栈
                移进后  输入串-1
            }
        }//循环正常结束  表示 acc
        System.out.println("语法分析结束，输入串格式正确，acc。");
    }
```

其中显示 ACTION 动作的时候需要根据 ACTION 表中的内容确定是移进还是规约，我这里使用的方法是，将移进动作设置为正数，那么当读到的数字>0 时就会输出 Si，表示移进动作，如果读到的<0，就会输出 ri，表示规约动作。

四、用例测试与结果分析

测试样例中共提交了 8 个可供测试的 txt 文件，这里选取 4 个比较典型的样

例说明测试结果的合理性。

4.1 测试用例 1

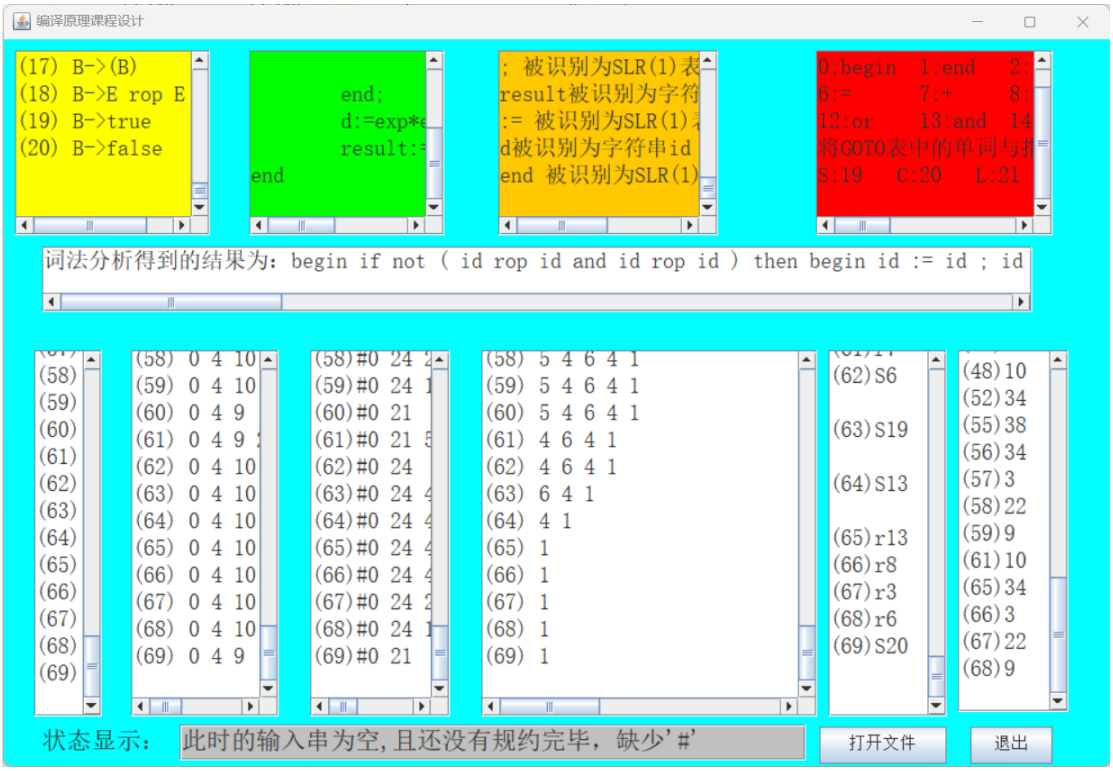
```
begin
    if a<b then
    begin
        a:=(a+b)*c;
        b:=-c
    end;
    d:=exp*exp;
    result:=a+b+d
end#
```

测试结果：

结果显示输入正确。

4.2 测试用例 2（测试用例 1 去掉了结束符’#’）

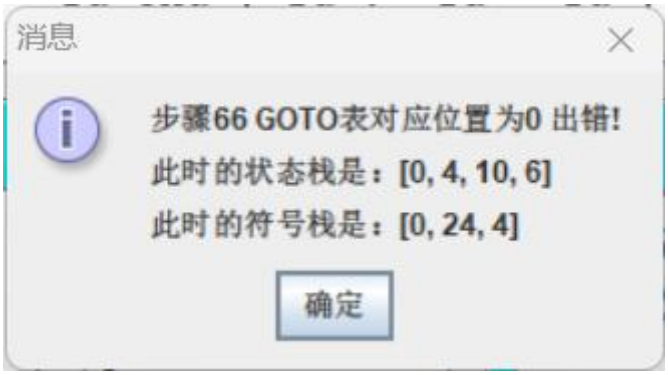
```
begin
    if a<b then
    begin
        a:=(a+b)*c;
        b:=-c
    end;
    d:=exp*exp;
    result:=a+b+d
end
```

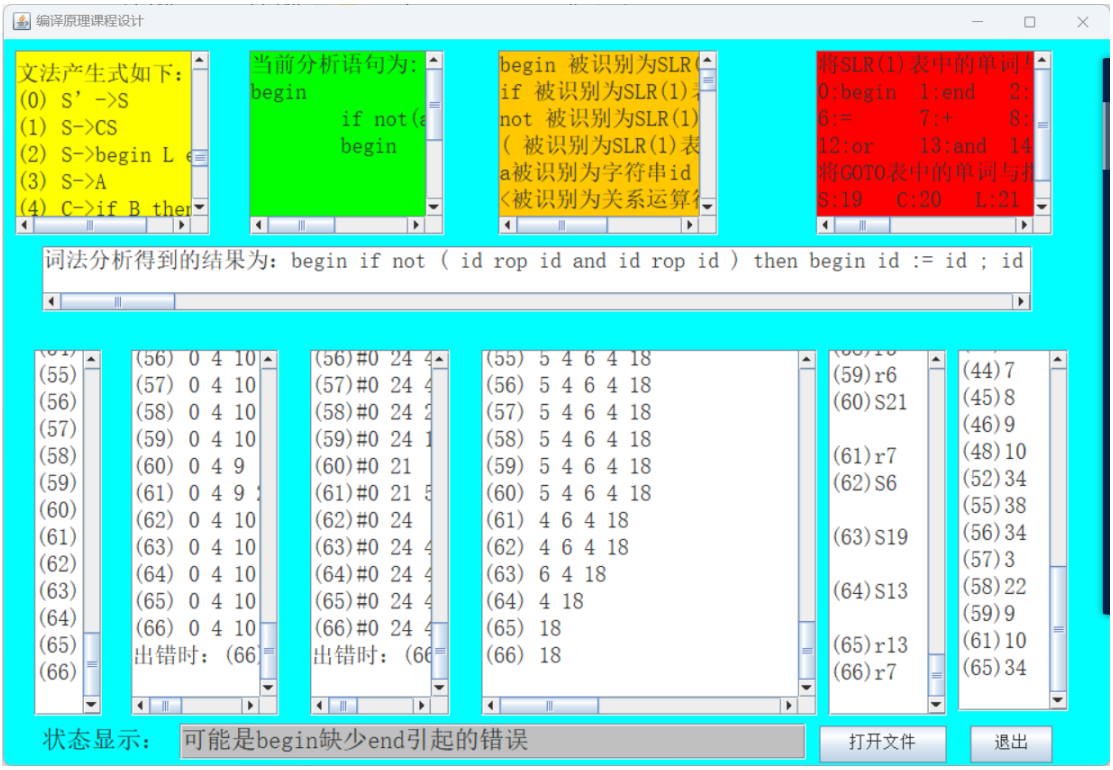


从结果中可以看到，程序检测到分析有误，并弹窗提示是由于缺少#引起的，程序也停在了发生错误的第 69 步。

4.3 测试用例 3（**测试用例 1 去掉了一个** end）

```
begin
    if a<b then
    begin
        a:=(a+b)*c;
        b:=-c
    end;
    d:=exp*exp;
    result:=a+b+d
#
```



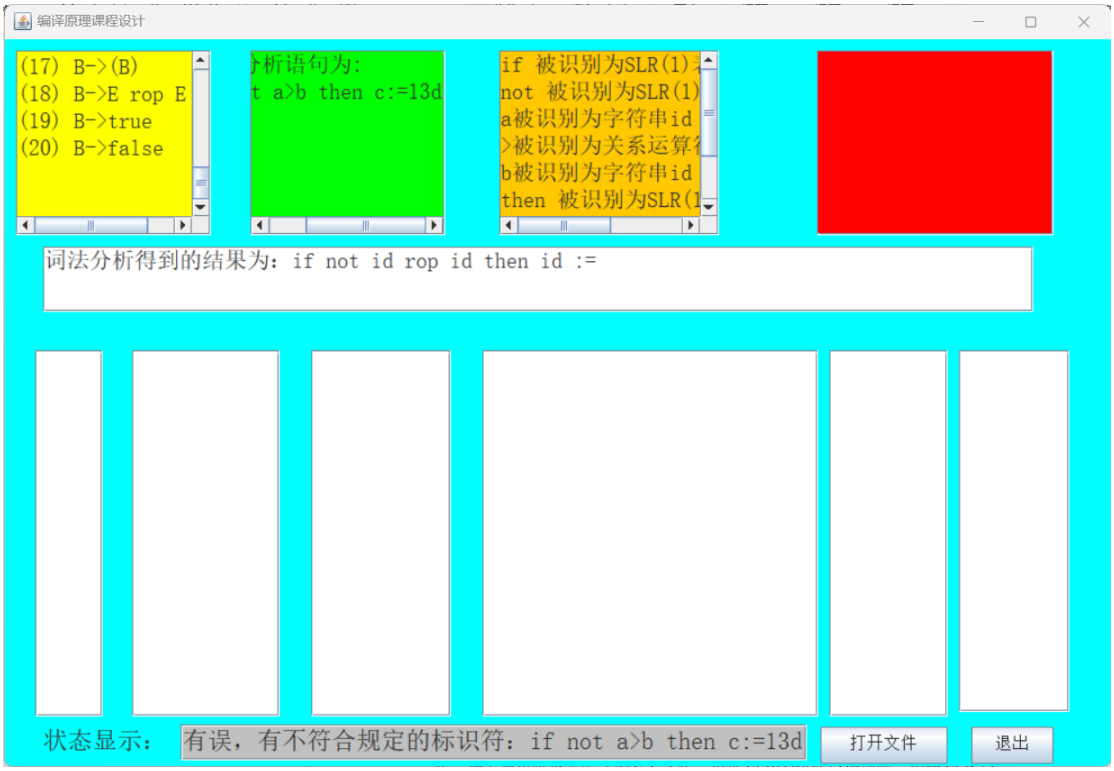


可以看到程序分析出文法有误，并提示可能是 begin 后面缺少 end 引起的。

4.4 测试用例 3

if not a>b then c:=13d

结果如下：



在词法分析阶段出现问题，出现不合规定的标识符（13d），直接停止分析，与语法分析相关的模块都没有输出。

五、遇到问题及解决

5.1 SLR(1)表的构建

本次课设由于产生式规则较多，构造 DFA 及绘制 SLR（1）表时工作量稍大。我们组首先找到各个非终结符的 First 集，然后找到非终结符和终结符的 Follow 集，接着得出每个产生式的 Select 集。根据 SLR(1)语法分析规则，构造 SLR 分析表。中途几经波折，但我们最终确定了 43 个状态，画出了 DFA 图。

5.2 代码的杂糅

在编程过程中，我们组原本打算在一个 gui 的 java 类中实现所有功能，但在编程实现的过程中，代码看起来非常混乱，最终我们选择新增一个 test 类用于辅助，对界面的相关操作可从 test 类中调用。

5.3 语法分析规约问题

这个问题主要是理论转变成的问题。当我们在按照分析表进行语法分析的时候，当某个步骤需要规约的时候，我们根据状态栈栈顶元素和输入串首字母，去查 ACTION 表读到 r_i，表示使用第 i 条产生式进行规约，规约的时候把活前缀规约成了产生式左边的非终结符。这里便会产生问题，编程实现的时候，是无法直接根据产生式来判断需要回退几位的。

最终我们组选择的解决办法是，根据产生式，将每条语句退栈的位数存到了一个固定的数组里面，并与产生式的序号对应，这样在实现规约的时候，就可以直接读出规约的位数，降低了编程的工作量。

六、心得体会

在本次课设的完成过程中，我们小组 3 人合作，完成了控制结构语句的词法分析和语法分析的任务，把相关的理论知识重新学习了一遍，在编程实现的过程中更是加深了理解，也加强了实践能力。在构造 DFA 和编程实现的时候都遇到了不少问题，或是改变思路，或是相互交流，最终将问题解决，这是在本次课设中的最大收获。实现编译过程的可视化，本身也是对我们编程能力的考验，需要我们不断地调试和修改代码。总体上，本次课程设计，我们通过编译原理理论的学习与实践，对于将源程序从编写，到编译、链接、生成中间文件等过程有了更深层次的了解。

但是，本次课设也存在着不足，由于编程能力的欠缺与理论学习的不足，语义分析最终没有完成，留下了不小的遗憾。今后将加强编程能力，完成更健壮，更丰富的代码。

工作量排序	姓名	学号
1	李南骏	**19220208**
2	高博文	**11120227**
3	席卓	**9203020720**