编译系统课程实验报告

实验 1：词法分析

<table>
<tr><td>姓名</td><td>杨富祥
王天一
白镇北</td><td>院系</td><td colspan="2">计算机科学与技术</td><td>学号</td><td>1171800323
1170300220
1170301005</td></tr>
<tr><td>任课教师</td><td colspan="2">陈鄞</td><td>指导教师</td><td colspan="3">文荟俨</td></tr>
<tr><td>实验地点</td><td colspan="2">软件学院三楼</td><td>实验时间</td><td colspan="3">2020-04-05　13: 45 – 15: 30</td></tr>
<tr><td rowspan="2">实验课表现</td><td>出勤、表现得分</td><td></td><td rowspan="2">实验报告得分</td><td rowspan="2"></td><td rowspan="2">实验总分</td><td rowspan="2"></td></tr>
<tr><td>操作结果得分</td><td></td></tr>
<tr><td colspan="7">组内分工情况说明：

杨富祥：
词法规则设计、DFA 汇总图设计、转换表输入文件构造、数据结构设计、GUI、报告修改

王天一：
核心算法实现、错误处理、数据结构设计、DFA 汇总图修改、报告修改

白镇北：
DFA 转换表构造、数据结构设计完善、实验报告撰写、GUI

小组群内讨论的截图：</td></tr>
</table>

一、需求分析	得分	

要求：阐述词法分析系统所要完成的功能

（1）作为对程序语言进行编译的第一阶段，词法分析是计算机科学中将字符序列转换为标记（Token）序列的过程。进行词法分析的程序或者函数叫做词法分析器，也叫扫描器。词法分析器一般以函数的形式存在，供语法分析器调用。词法分析的主要任务是读入源程序的输入字符、将它们组成词素，生成并输出一个词法单元，即（Token）序列，这个词法单元

序列被输入到语法分析器进行语法分析。

（2）这里的标记是一个字符串，是构成源代码的最小单位。从输入字符流中生成标记的过程叫做标记化，在这个过程中，词法分析器还会对标记进行分类。词法分析器通常不会关心标记之间的关系（这属于语法分析的范畴），举例来说：词法分析器能够将括号识别为标记，但并不保证括号是否匹配。

二、文法设计	得分	

（1） 给出各类单词的词法规则描述（正则文法或正则表达式）

a）标识符：

digit → [0 – 9]

letter_ → [A - Z a - z _]

id → letter_ (letter_ | digit) *

b）无符号数：

digits → digit digit *

optionalFraction → . digits | ε

optionalExponent → (E (+ | - | ε) digits) | ε

number → digits optionalFraction optionalExponent

c）运算符：

relational_op → == | != | > | < | <= | >=

logical_op → && | ! | ||

arithmetic_op → + | - | * | / | % | ++ | - -

d）界符：

delimiter → = | , | ; | (|) | [|] | { | }

e）注释：

comment → / * other * /

f）8 进制：

OCT → 0 [0 – 7] +

g）16 进制：

hex → 0x [0 – 9 a – f A – F] +

h）字符常数：

char → ' [a – z A – Z] '

i）关键字：

keyword → int | float | char | struct | bool | true | false | if | else | do | while | for | break | continue | proc | call | return

（2） DFA 转换图

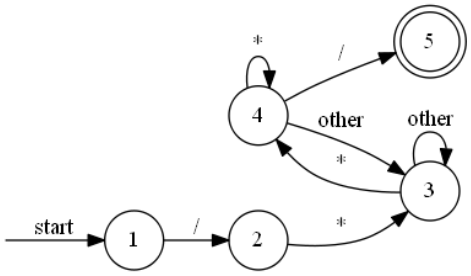


图 1：识别注释的 DFA

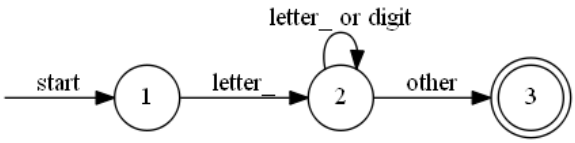


图 2：识别标识符的 DFA

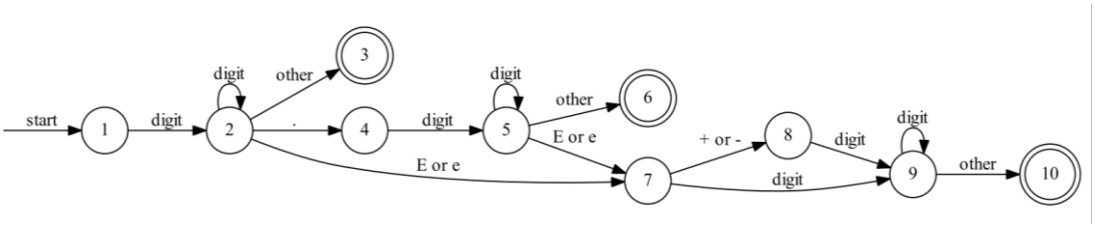


图 3：识别无符号数的 DFA

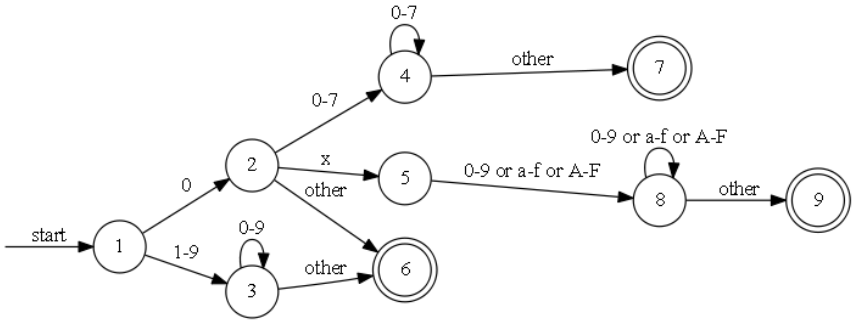


图 4：识别十进制（6）、八进制（7）和十六进制（9）

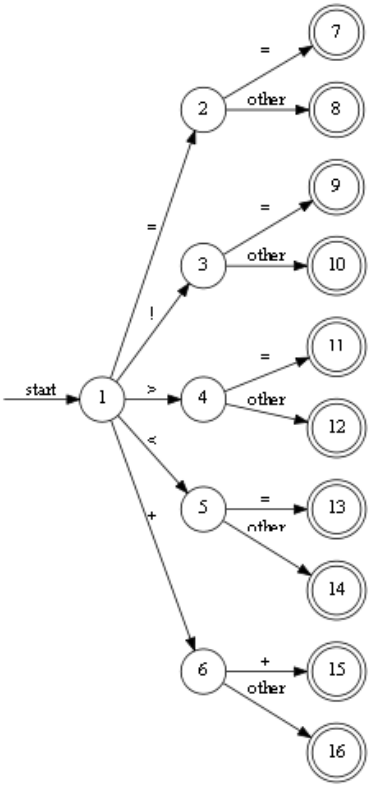


图 5：识别部分运算符

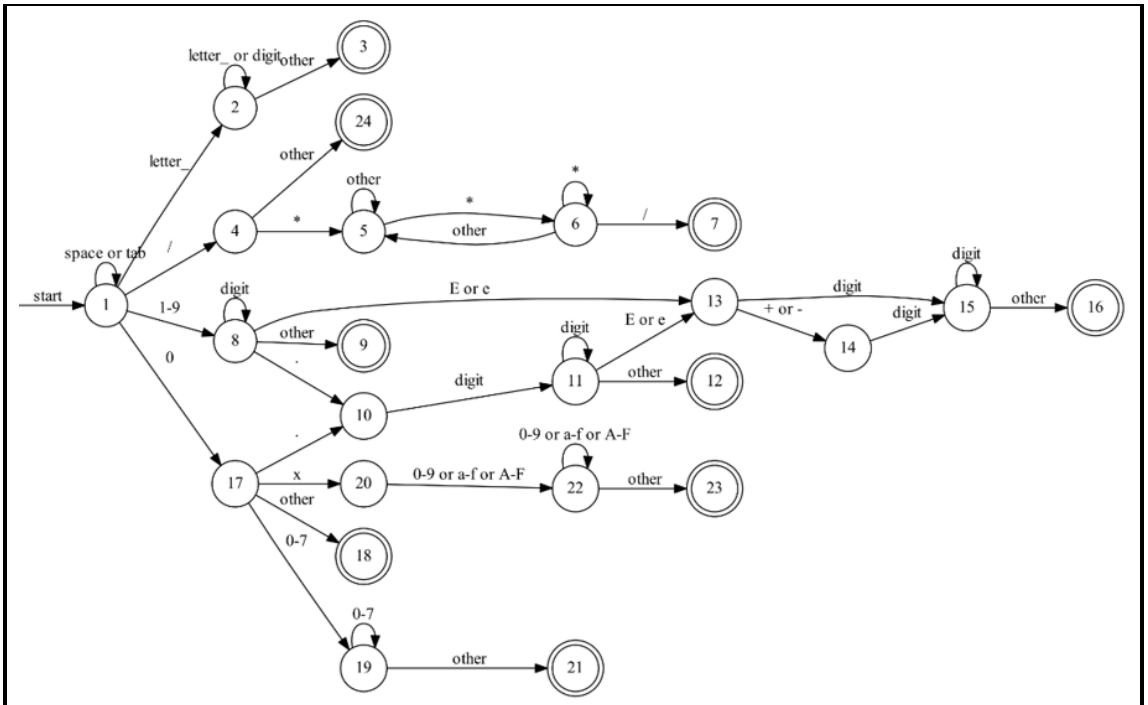
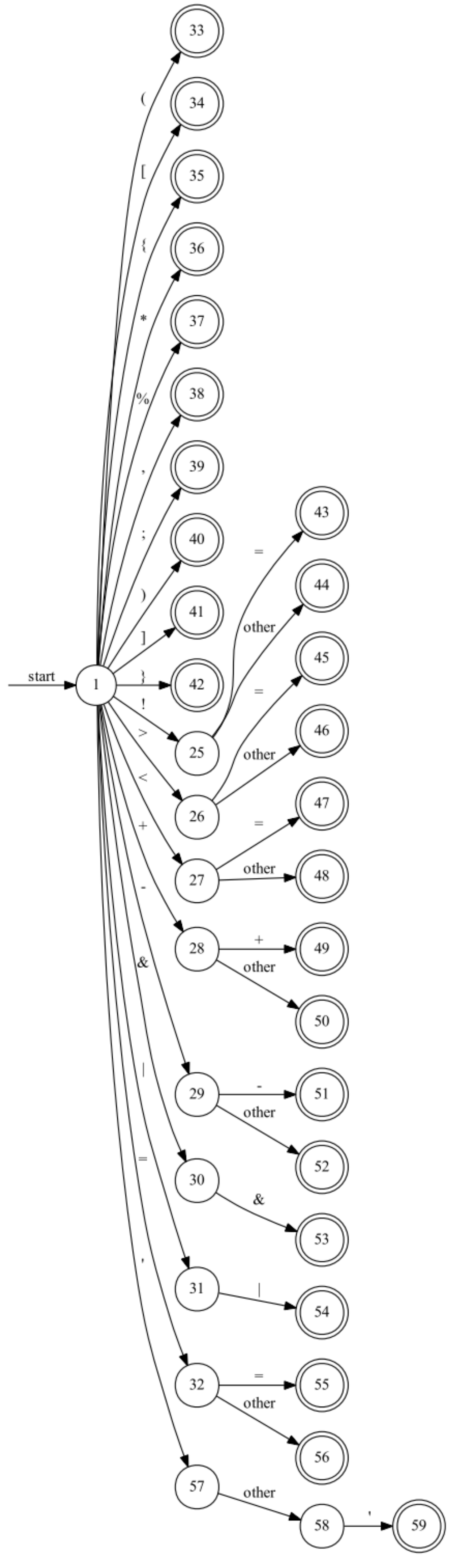

start
space or tab
letter_
/
1-9
0
digit
other
.
E or e
+ or -
digit
other
letter_ or digit
other
other
*
other
*
/
digit
E or e
digit
other
.
x
other
0-7
0-7
other
0-9 or a-f or A-F
0-9 or a-f or A-F
other
digit
other
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24

图 6：汇总后的 DFA（1）



start
1
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
25
26
27
28
29
30
31
32
(
[
{
*
%
,
;
)
]
}
!
>
<
+
-
&
|
=
'
other

图 7：汇总后的 DFA（2）

三、系统设计	得分	

（1）系统概要设计：

下图为词法分析器部分的 UML：

Lexer 为分析器的主类；

Token 作为词素在其中引用了 Tag 种所枚举的词语类型，且根据词素作为各进制整型、浮点型还是关键字或字符型设计了 Num, Real, Word 三种方案来继承它；

Graph 中以有向图的形式保存了由外部输入所构建的 DFA，Edge 作为边集保存了由 DFA 的一个状态转换为另一个状态的过程；

MapRuler 中完成了由所给输入构建 DFA 边的权重到输入文件的映射规则；

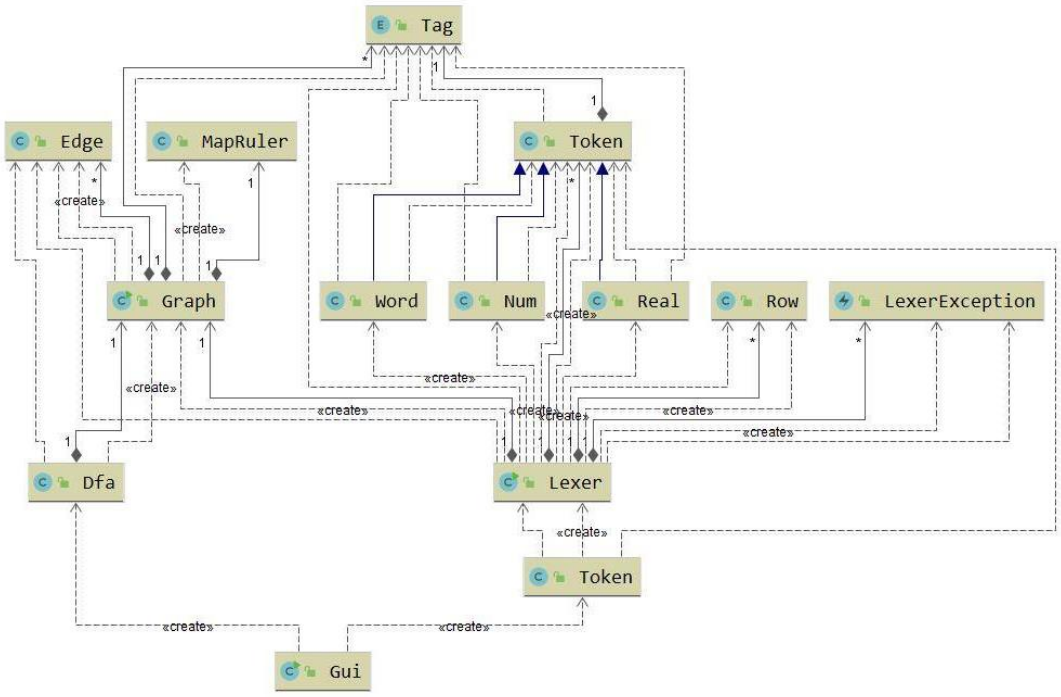


图 8：词法分析器的 UML

（2）系统详细设计：

1、核心数据结构的设计

a）Graph 类：

由边集 Edge 组成，通过读入给定的文件，以 List<Edge>的形式完成了 DFA 的构建，其内部主要实现了给定起点和输入符号，输出可行终点的 getTarget() 方法。

b）Edge 类：

有向边集，将自动机的状态以整型变量保存为边的两个顶点，两个状态间的转换符号用字符串型式的 weight 保存。

c）Token 类：

给运算符、界符等一词一码的词素使用。

2、主要功能函数说明

Lexer. findTokens()

最主要的生成 token 的函数，这里面对每一行的每一个字符进行识别判断，调用了 move 方法，简单来说，从头开始一步一步走，当到达到接收状态时，就把接收状态，和缓存的字符串传递到 addToken 中，然后状态归一，temp 清空。

Lexer.addToken(String symbol, int state)

Symbol 就是接受的字符串，state 就是接收状态，根据接收状态和接受字符串构建一个 Token.

Lexer.readFile(String filename)

读取测试用例的方法。

Graph.getTarget(int source, char ch)

move 方法，给定一个状态和一个输入，通过遍历输入的 dfa 文件，找到当前状态，及当前输入的字符，返回他对应的下一个状态。

3、程序核心部分的程序流程图：

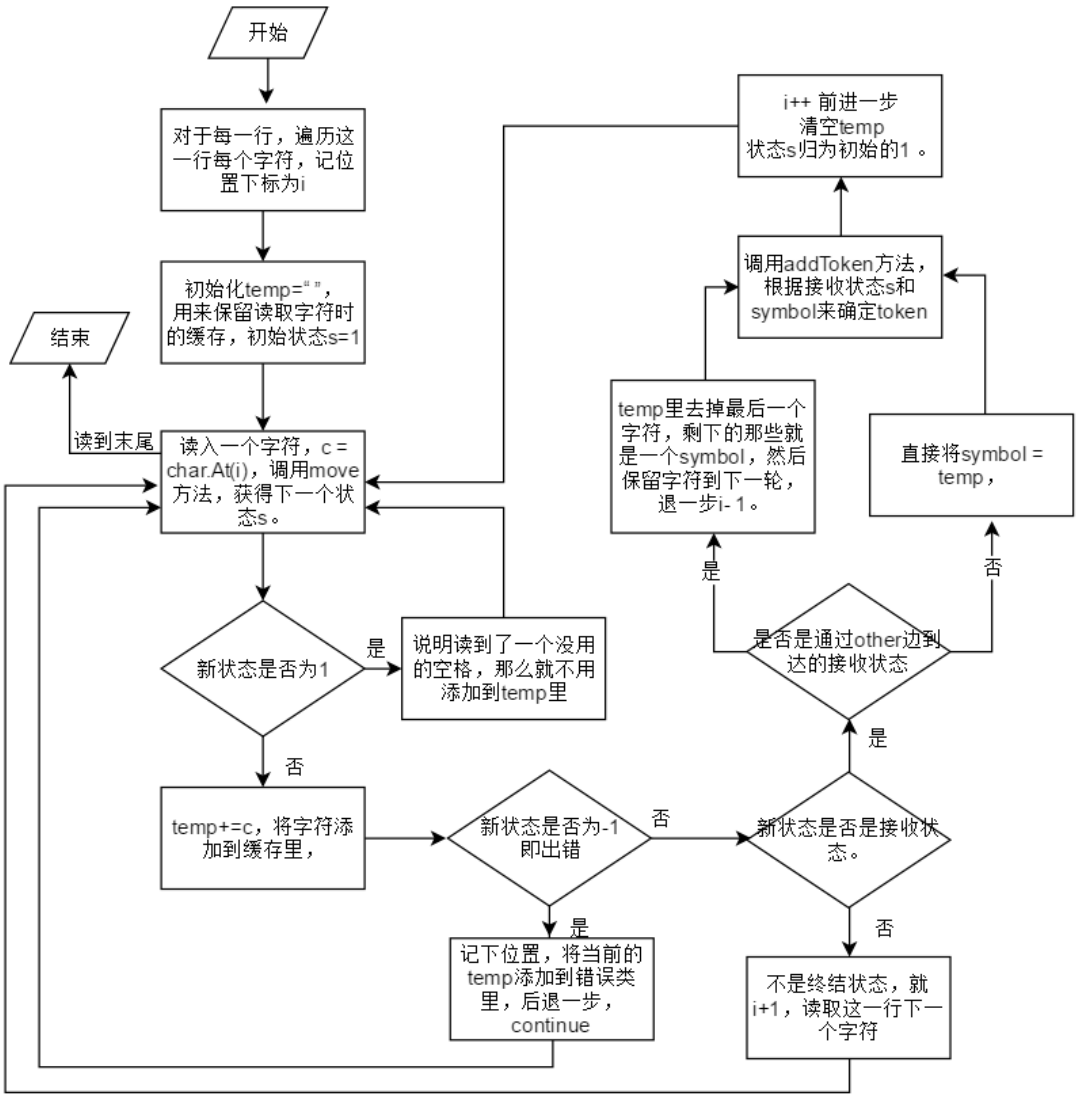


图 9：关键函数流程图

四、系统实现及结果分析

得分

（1） 系统实现过程中遇到的问题；

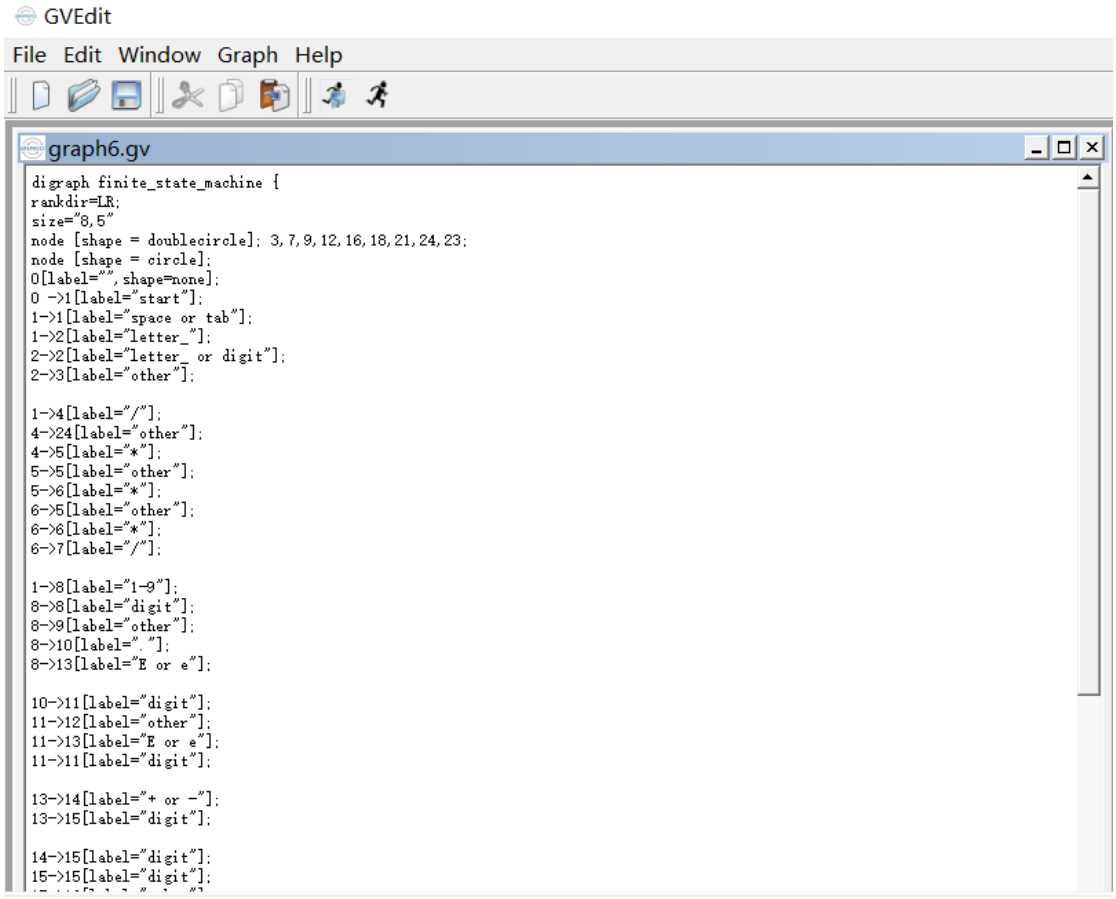
1)、对于有些符号，其本应该被识别成功的，但是却被作为错误字符处理了，又会回到初始状态，可能在识别过程中会产生无限循环的问题，但也很容易解决。

2)、在写核心的 findTokens 方法时，因为在 DFA 转换图中有的接收状态是通过 other 实现的，有的不需要，比如 int，他就是一个标识符，1 遇到 I,=>2，2 遇到 n=>2，2 遇到 t=>2，接下来 2 再读到一个 other 才会=>3，所以这个时候，temp 里面只能加入 int，不能把这个 other 字符读进去。

所以在到达接收状态时候，判断是否是通过 other 到达的，如果不是，就正常接受，把读到的全加到 temp 里，然后就调用 addToken 方法。

如果是通过 other 到达的，所以解决办法是本次步幅 i 不变，这样保证下次的循环是从 other 开始的，就不会多加或漏掉任何的字符。

3）、DFA 汇总图的制作比较耗时耗力，尤其需要注意八进制、十六进制和无符号数它们各自 DFA 图合并时些许冲突问题。由于图状态较多，建议采用 GVEdit 这个软件绘制。



图：使用代码绘制 DFA 图

4）、首先我们设计出了各类单词的词法规则，然后会根据词法规则制作 DFA 转换图。需要设计数据结构存储这个转换图，便于查询。另外，要从文件读入该图，文件应当设计得当，便于程序读取。

5）、DFA 转换图的边有时会是一个字符，如“*”，但有时会是多个字符，如所有数字。当然，这些字符都可以单独成边，但是过于杂多。实际上可以用“digit”、“letter_”等字符串表示，这样它们实际上代表符号的集合。于是，需要一个映射关系，将集合名称和集合元素一一对应。本次实验，我们将之写在了程序内部：MapRuler.java.

6）、Tag 类中枚举了所有接收单词，这里需要仔细推敲。实际上我们进行了若干次增删

才最终确定结果。

7）、lexer.Token 类中属性为单词种别码，如果接收的单词是一词一码的运算符、界符等可以使用；Word 类、Num 类、Real 类都分别继承了 lexer.Token 类，其中 Word 负责注释、ID、关键字、char 字符，Num 负责十进制数、八进制和十六进制数，Real 负责浮点数和科学计数法。

7）、实际上，如果识别到了一个单词，如“int”，要返回 token 序列，我们需要知道应该是返回 new Word()，还是 new Num()、new Real()或者 new Token().并且要能从这个单词对应到它的种别码。为此，我们设计了 public static Tag fromString(String tag)函数和 private void addToken(String symbol, int state)函数。addToken 函数根据接收单词和接收状态就能利用 switch 语句判断应该返回那个类型的 Token.

而且，这样可以和接受状态解耦合，接收状态号变了，如“3”变成了“4”，仍然能够正确构建 token.

```
/**
 * 根据接收状态和接受字符串构建Token
 *
 * @param symbol 接收单词
 * @param state  接收状态
 */
private void addToken(String symbol, int state) {

    Tag tag = graph.getEndStates().get(state);
    switch (tag) {
        case ID:
            if (graph.isKeyWord(symbol)) {
                tokens.add(new Token(Tag.fromString(symbol)));
                return;
            }
        case NOTE:
        case CHARACTER:
            tokens.add(new Word(symbol, tag));
            break;
        case NUM:
        case OCT:
        case HEX:
            tokens.add(new Num(parseToNum(symbol), tag));
            break;
        case REAL:
            tokens.add(new Real(parseToReal(symbol)));
            break;
        default:
            tokens.add(new Token(tag));
            break;
    }
}
```

图：确定返回特定类型的 token

8）、DFA 转换图中出现了 other 边，经过 other 边会到达接收状态，识别该单词，回到初始状态，应该再从 other 边这个字符读起，如“int a = 10;”，读到分号会从 other 边进入接收十进制数的状态，并回到初始状态，从分号这个字符读起，不能丢掉分号。

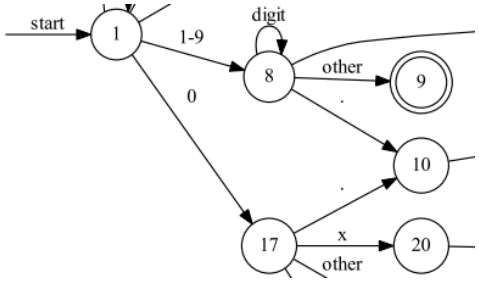
还有错误处理时，也要注意字符丢失的处理。

具体来说，从一个状态出发，经过的边会有合法字符、“other”。但并不是都有“other”，

如果没有“other”边，会进行错误处理，回到初始状态；如果有“other”边并且该字符是非法的，会前接收该字符前的单词，跳回初始状态。会有无限循环。

如“10#”，读入“1”，进入状态 8，读入“0”，进入状态 8，读入非法字符“#”，会经过“other”边进入状态 9，接收整型数 10，然后跳回状态 1。如果回退一个字符再从“#”开始读，又会跳到状态 1，造成无限循环。而如果是“10;”，则回退一个字符从分号读起，会接收该分号。

为了解决该问题，可以加一个小判断，如果原状态是 1，遇到非法字符不要回退即可。



图：DFA 转换图部分

（2） 测试用例如下：

1.

```
struct student{
    int id ;
    char[] name ;
};
    boolen flag = false  ;
    int a = 011 #;
    int b = 0xff ;
    int c = 0  错误测试 */ ;
    float d = 99.9;
    float e  = 9.7e-2;
    char ch = ' ;
    char cch = 'a'; & ;
    while(c<10){
        for (int i = 1 ; i<5;i++){
            c += 1;
            if(c = 9)
                break;
        }
    }
    /*函数声明*/
    proc int getSum(int x,int y){
        return x+y;
    }
    /*函数调用*/
    call getNum(3,5);
```

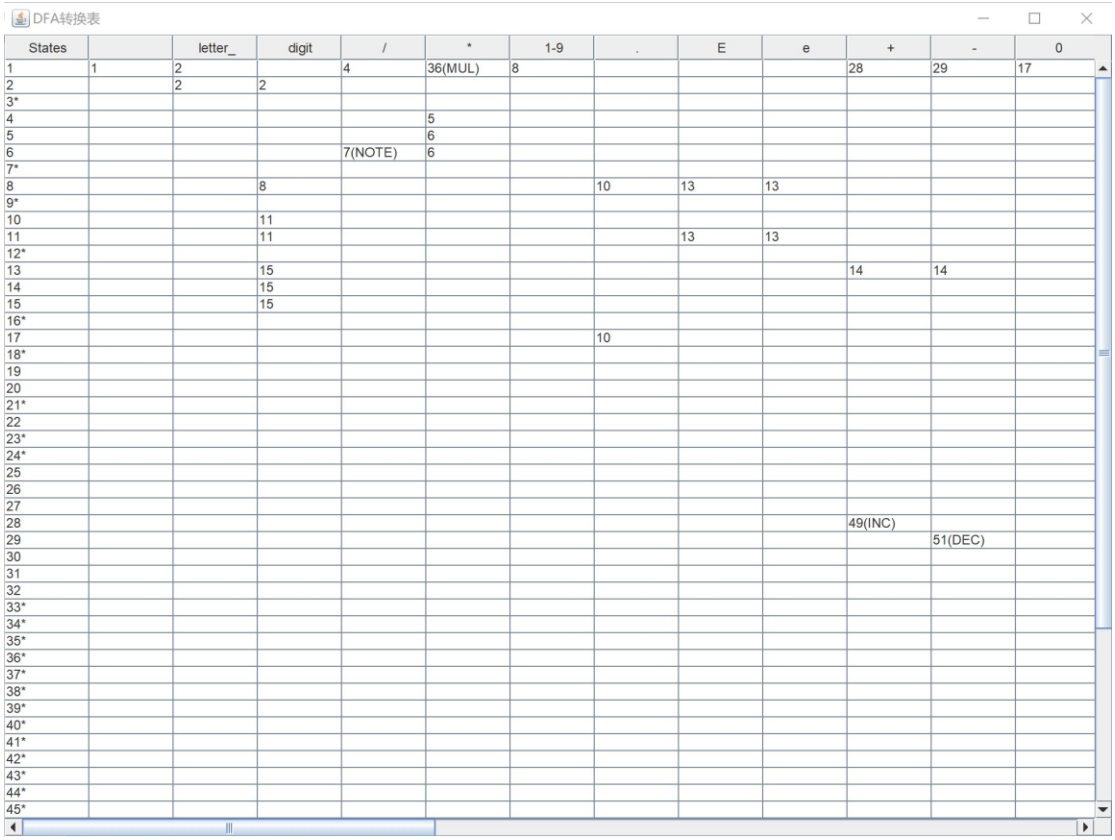
图：测试用例

（3） 构建的 DFA 转换表（部分）：

在文件第一行声明了关键字，接下来的以 endstates 开头的就是所有的接收状态，再下面就是具体的状态转换了。以第七行为例，表示状态 1 遇到 letter_会跳转到状态 2.

```
keyword:int float char struct bool true false if else while do for break continue proc call return
endstate:(3#id) (24#/) (7#note) (9#num) (12#real) (16#real) (23#hex) (18#num) (21#oct)
endstate:(33#() (34#[) (35#{) (36#*) (37#%) (38#,) (39#;) (40#)) (41#]) (42#})
endstate:(43#!=) (44#!) (45#>=) (46#>) (47#<=) (48#<) (49#++) (50#+) (51#--) (52#-)
endstate:(53#&&) (54#||) (55#==) (56#=) (59#character)
1#1#
1#2#letter_
2#2#letter_
2#2#digit
2#3#other1
1#4#/
4#24#other2
4#5#*
5#5#other3
5#6#*
6#5#other4
6#6#*
6#7#/
1#8#1-9
8#9#other5
8#8#digit
8#10#.
8#13#E
8#13#e
10#11#digit
11#11#digit
11#12#other6
11#13#E
11#13#e
13#14#+
13#14#-
13#15#digit
14#15#digit
15#15#digit
15#16#other7
1#17#0
17#10#.
17#20#x
17#18#other8
17#19#0-7
19#19#0-7
19#21#other9
20#22#0-9 or a-f or A-F
22#22#0-9 or a-f or A-F
22#23#other10
1#33#(
1#34#[
1#35#{
```

图：输入的用以构建 DFA 的序列



States		letter_	digit	/	*	1-9	.	E	e	+	-	0
1	1	2		4	36(MUL)	8				28	29	17
2		2	2									
3*												
4					5							
5					6							
6				7(NOTE)	6							
7*												
8			8				10	13	13			
9*												
10			11									
11			11					13	13			
12*												
13			15							14	14	
14			15									
15			15									
16*												
17							10					
18*												
19												
20												
21*												
22												
23*												
24*												
25												
26												
27												
28										49(INC)		
29											51(DEC)	
30												
31												
32												
33*												
34*												
35*												
36*												
37*												
38*												
39*												
40*												
41*												
42*												
43*												
44*												
45*												

图：DFA 转换表

（4） 词法分析结果：Token 序列

```
<STRUCT, _>
<ID, student>
<LB, _>
<INT, _>
<ID, id>
<SEMI, _>
<CHAR, _>
<LM, _>
<RM, _>
<ID, name>
<SEMI, _>
<RB, _>
<SEMI, _>
<ID, boolen>
<ID, flag>
<ASSIGN, _>
<FALSE, _>
<SEMI, _>
<INT, _>
<ID, a>
<ASSIGN, _>
<OCT, 9>
<SEMI, _>
<INT, _>
<ID, b>
<ASSIGN, _>
<HEX, 255>
<SEMI, _>
<INT, _>
<ID, c>
<ASSIGN, _>
<NUM, 0>
<MUL, _>
<DIV, _>
<SEMI, _>
<FLOAT, _>
<ID, d>
<ASSIGN, _>
<REAL, 99.9>
<SEMI, _>
<FLOAT, _>
<ID, e>
<ASSIGN, _>
<REAL, 0.097>
<SEMI, _>
<CHAR, _>
<ID, ch>
<ASSIGN, _>
<SEMI, _>
<CHAR, _>
<ID, cch>
<ASSIGN, _>
<CHARACTER, 'a'>
<SEMI, _>
<SEMI, _>
<WHILE, _>
<LS, _>
<ID, c>
<LT, _>
<NUM, 10>
<RS, _>
<LB, _>
<FOR, _>
<LS, _>
<INT, _>
<ID, i>
<ASSIGN, _>
<NUM, 1>
<SEMI, _>
<ID, i>
<LT, _>
<NUM, 5>
<SEMI, _>
<ID, i>
<INC, _>
<RS, _>
<LB, _>
<ID, c>
<ADD, _>
<ASSIGN, _>
<NUM, 1>
<SEMI, _>
<IF, _>
<LS, _>
<ID, c>
<ASSIGN, _>
<NUM, 9>
<RS, _>
<BREAK, _>
<SEMI, _>
<RB, _>
<RB, _>
<NOTE, /*函数声明*/>
<PROC, _>
<INT, _>
<ID, getSum>
<LS, _>
<INT, _>
<ID, x>
<COMMA, _>
<INT, _>
<ID, y>
<RS, _>
<LB, _>
<RETURN, _>
<ID, x>
<ADD, _>
<ID, y>
<SEMI, _>
<RB, _>
<NOTE, /*函数调用*/>
<CALL, _>
<ID, getNum>
<LS, _>
<NUM, 3>
<COMMA, _>
<NUM, 5>
<RS, _>
<SEMI, _>
```

图：输出的 Token 序列

（5） 输出针对此测试程序对应的词法错误报告；

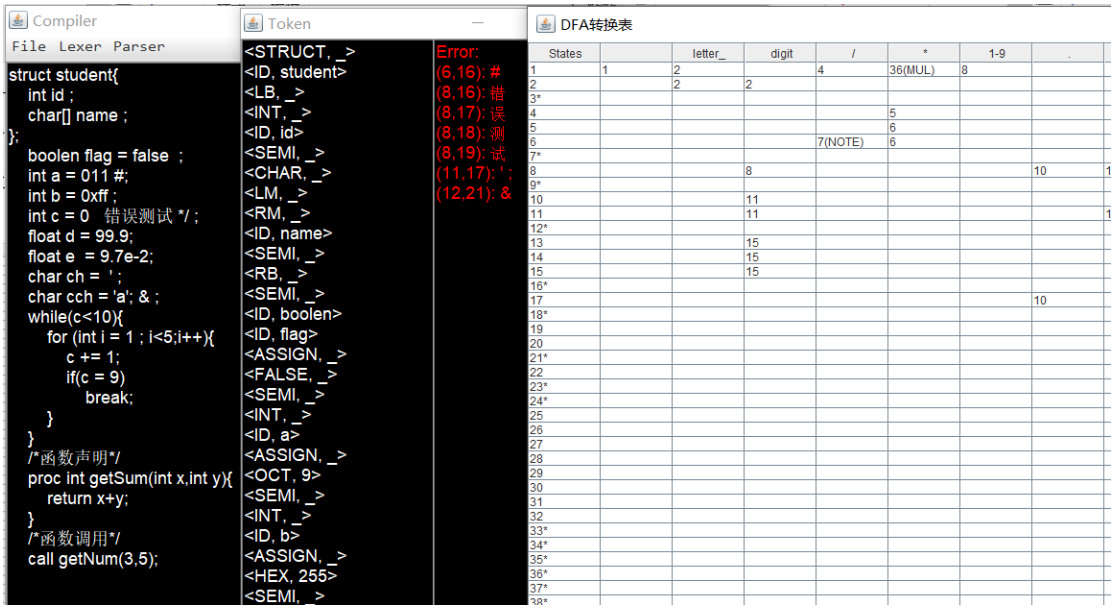
图：错误报告

上图表示在第六行，第 16 个字符处出现错误。

（6） 对实验结果进行分析。

对于这段源程序，其中包含各类关键字、算符、界符、标识符、常数以及注释，在最终的词法分析的 token 序列中可以将各类单词区分开，并且将注释和空白符（空格

及回车）过滤掉。对于错误信息，输出为错误的位置，精确到具体的行数字符数。



图：程序运行界面（部分）

指导教师评语：

日期：