编译原理词法分析实验报告

一、实验题目与要求

> 题目一：认识LEX/FLEX及其语法结构，并完成简单规则的书写
>
> 题目二：利用FLEX工具生成PL语言的词法分析器，在题目一的基础上进一步扩展
>
> 题目三：在题目一、题目二都掌握的基础上，设计并实现一个C语言的词法分析程序

实验一的要求与解析：

1.用户编程要求

根据提示，在右侧编辑器补充代码，实现对以小写字母 `ab` 结尾的字符串（只包含大小写字母）的识别，如 `Helloab` 和 `Goab`。

注意，你只需要保证合法的输入（以ab结尾的字符串）有结果，不合法的输入将会包含在 `.` 规则中。

图1 实验一编程要求

由题目要求（图1）可知，我们只需要实现对于全字母的字符串，结尾为ab的辨别即可，其模式为：[a-zA-Z]* ab

2.LEX/FLEX语法结构（实验二三略去）

通过处理其源文件来生词法和语法分析器的，源文件的扩展名为 `.l`，其语法被分为三个部分，用 `%%` 进行分离，如图2所示：

```
/* 定义段 */
%{
%}
%%
/* 规则段 */
%%
/* 用户子程序段 */
```

图2 LEX/FLEX语法结构示例

1)定义段：

这一部分一般是一些声明及选项设置等。C 语言的注释、头文件包含等放在 `%{ %}` 之间，这一部分的内容会被直接复制到输出文件的开头部分；

2)规则段：

为一系列匹配模式和动作，模式一般使用正则表达式书写，动作部分为C代码：`模式1 {动作1(C代码)}`，在输入和 `模式 1` 匹配的时候，执行动作部分的代码；其中，模式内的合法符号需要满

足如图3的要求：

符号	含义
|	或
[]	括号中的字符取其一
-	a-z表示ascii码中介于a-z包括a.z的字符
\	转义（flex不能识别除字母外的字符）
*	0或多个字符
?	0或1个字符
+	1或多个字符
^	除此之外的其余字符
.	除\n外的所有字符，等价于^\n

图3 模式串中，各个合法符号的含义

3)用户子程序段：

仅含 `C` 代码，会被原样复制到输出文件中，一般这里定义一些辅助函数等，如动作代码中使用到的辅助函数。

3.测试程序示例

测试输入：`Helloab` , `Goab` , `HiAb` , `ab` , `AB` ;

预期输出：

```
Helloab: Hit!
Goab: Hit!
ab: Hit!
```

图4 实验一测试程序示例

如图4所示，我们可以从中得到输出格式，即 `printf("%s: Hit!\n",yytext)` 语句。

4.实验通过截图

```
/* 简单词法分析器 */
/* 功能：能够识别出以小写字母ab结尾的所有字符串（仅含大小写字母）并给打印'Hit!' */
/* 说明：在下面的begin和end之间添加代码，注意格式 */
/* 提示：你只需要保证合法的输入（以ab结尾的字符串）有结果，不合法的输入将会包含在.规则中～ */
%{
#include <stdio.h>
%}

%%
/* begin */
[a-zA-Z]*ab    {printf("%s: Hit!\n",yytext);}
/* end */

\n              {}
.               {}
%%
int yywrap() { return 1; }
int main(int argc, char **argv)
{
    if (argc > 1) {
        if (!(yyin = fopen(argv[1], "r"))) {
            perror(argv[1]);
            return 1;
        }
    }
    while (yylex());
    return 0;
}
```

图5 笔者编写的实验一程序

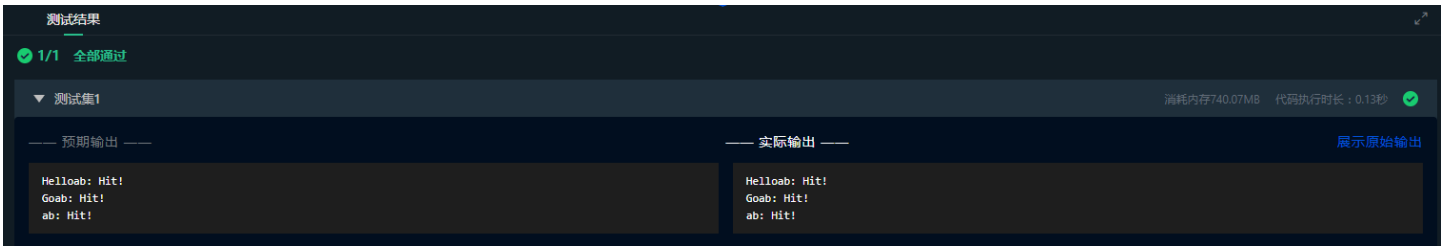


图6 实验一通过截图

笔者代码如图5所示，实验结果如图6所示。

实验二的要求与解析：

1.用户编程要求

根据提示，在右侧编辑器补充代码，设计识别PL语言单词符号的词法分析器。

PL语言单词符号及其种别值

单词符号	种别枚举值	单词符号	种别枚举值
标识符	IDENT	]	RBRACK
整常量	INTCON	(	LPAREN
字符常量	CHARCON	)	RPAREN
+	PLUS	,	COMMA
-	MINUS	;	SEMICOLON
*	TIMES	.	PERIOD
/	DIVSYM	:=	BECOME
=	EQL	:	COLON
<>	NEQ	begin	BEGINSYM
<	LSS	end	ENDSYM
<=	LEQ	if	IFSYM
>	GTR	then	THENSYM
>=	GEQ	else	ELSESYM
of	OFSYM	while	WHILESYM
array	ARRAYSYM	do	DOSYM
program	PROGRAMSYM	call	CALLSYM
mod	MODSYM	const	CONSTSYM
and	ANDSYM	type	TYPESYM
or	ORSYM	var	VARSYM
not	NOTSYM	procedure	PROCSYM
[	LBRACK		

注意，这里还有一个类别ERROR，包括不是出现在字符串中的非法字符，有~!@#$%^&_\ 当词法分析器读到非法字符时，应该输出ERROR作为种别值。

图7 PL语言的单词类别

如图7所示，我们需要根据PL(Programming Language)语言的合法词元，构造可以识别PL语言单词符号的词法分析器。

2.状态图与自动机

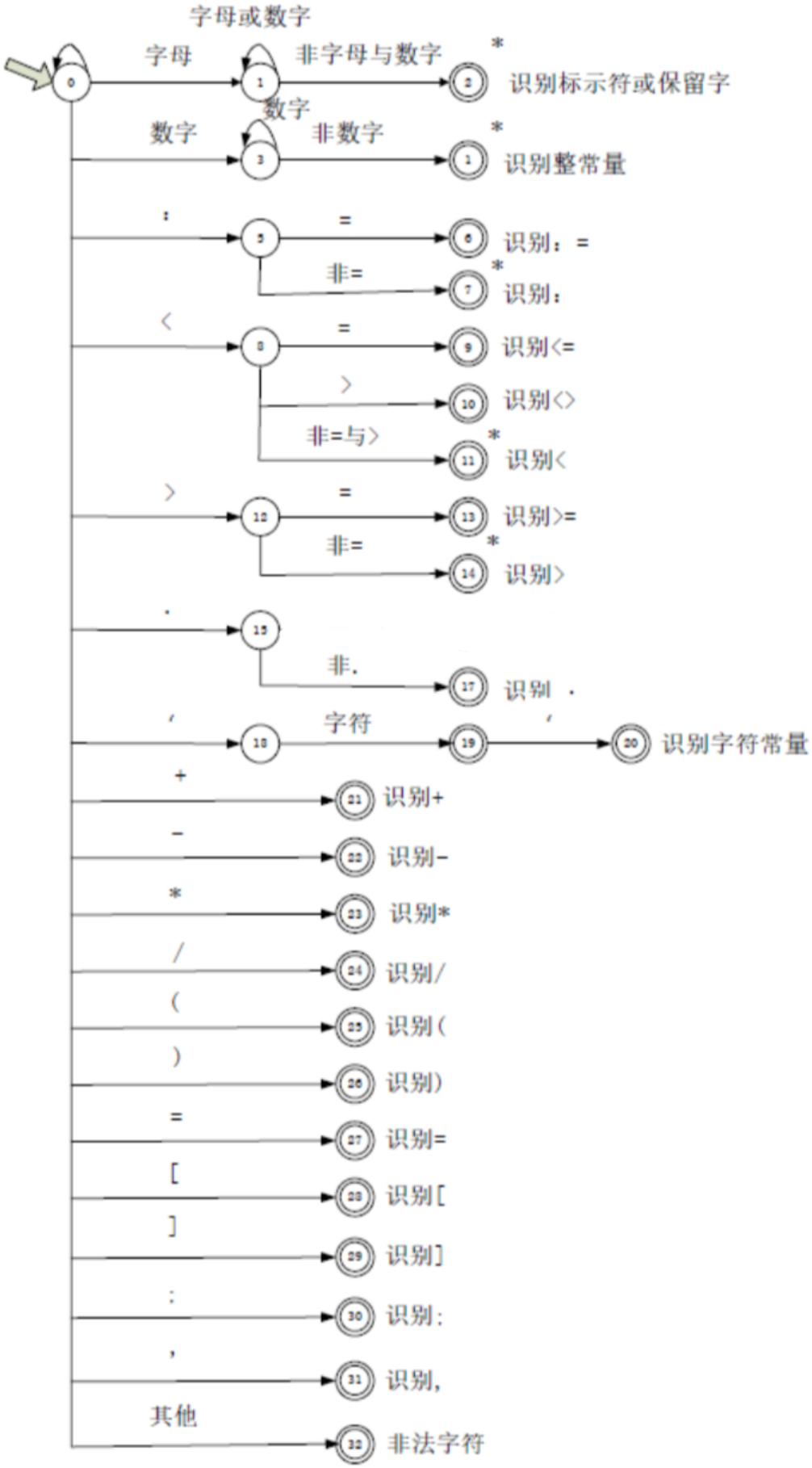


图8 PL语言的单词识别自动机

如图8所示，我们以此自动机构造模式串和识别顺序优先级，需要注意的是，我们并不需要识别符号".."，其在所有样例当中不存在，此外，按照普遍规律而言，PL语言当中很少用到此单词符号。

3.测试程序示例

测试输入1：`Test2_1.PLS`；
这个测试集主要测试对算术运算符的识别。
需要完成的种别有：IDENT，INTCON，PLUS，MINUS，TIMES，DIVSYM，BECOME。

测试输入2：`Test2_2.PLS`
这个测试集主要测试对字符常量的识别。
需要完成的种别有：CHARCON。

测试输入3：`Test2_3.PLS`
这个测试集主要测试对比较运算符的识别。
需要完成的种别有：IDENT，EQL，NEQ，LSS，LEQ，GTR，GEQ。

测试输入4：`Test2_4.PLS`
这个测试集是一个完整PL源程序。
需要完成的种别有：所有PL可以识别的种别。

测试输入5：`Test2_5.PLS`
这个测试集是一个完整PL源程序中间插入了非法字符，需要将其识别出来。
需要完成的种别有：所有PL可以识别的种别+ERROR。

图9 实验二测试程序示例

如图9，我们可以得知除去以上的分支检测以外，还需要额外注意ERROR错误检测，在此笔者赶忙修补了错误分支的检测，添加了对字符`~!@#$%^&_\`的匹配检测语句。

4.实验通过截图

```
 1  /* PL词法分析器 */
 2  /* 功能：能够识别出PL支持的所有单词符号并给出种别值 */
 3  /* 说明：在下面的begin和end之间添加代码，已经实现了标识符和整常量的识别，你需要完成剩下的部分，加油吧！ */
 4  /* 提示：因为是顺序匹配，即从上至下依次匹配规则，所以需要合理安排顺序~ */
 5  %{
 6  #include <stdio.h>
 7  %}
 8  /* begin */
 9  INTCON          [\-]?[1-9][0-9]*|0
10  IDENT           [A-Za-z][A-Za-z0-9]*
11  CHARCON         ['][^']*[']
12
13  /* end */
14
15  %%
16  /* begin */
17
18  {CHARCON}       {printf("%s: CHARCON\n", yytext);}
19  [\+]            {printf("%s: PLUS\n", yytext);}
20  [\-]            {printf("%s: MINUS\n", yytext);}
21  [\*]            {printf("%s: TIMES\n", yytext);}
22  [\/]            {printf("%s: DIVSYM\n", yytext);}
```

```
[\=]             {printf("%s: EQL\n", yytext);}
[\<][\>]         {printf("%s: NEQ\n", yytext);}
[\<][\=]         {printf("%s: LEQ\n", yytext);}
[\<]             {printf("%s: LSS\n", yytext);}
[\>][\=]         {printf("%s: GEQ\n", yytext);}
[\>]             {printf("%s: GTR\n", yytext);}
[o][f]             {printf("%s: OFSYM\n", yytext);}
[a][r][r][a][y]          {printf("%s: ARRAYSYM\n", yytext);}
[p][r][o][g][r][a][m]        {printf("%s: PROGRAMSYM\n", yytext);}
[m][o][d]            {printf("%s: MODSYM\n", yytext);}
[a][n][d]            {printf("%s: ANDSYM\n", yytext);}
[o][r]             {printf("%s: ORSYM\n", yytext);}
[n][o][t]            {printf("%s: NOTSYM\n", yytext);}
[\[]             {printf("%s: LBRACK\n", yytext);}
[\]]             {printf("%s: RBRACK\n", yytext);}
[\(]             {printf("%s: LPAREN\n", yytext);}
[\)]             {printf("%s: RPAREN\n", yytext);}
[\,]             {printf("%s: COMMA\n", yytext);}
[\;]             {printf("%s: SEMICOLON\n", yytext);}
[\.]             {printf("%s: PERIOD\n", yytext);}
[\:][\=]         {printf("%s: BECOME\n", yytext);}
[\:]             {printf("%s: COLON\n", yytext);}
[b][e][g][i][n]          {printf("%s: BEGINSYM\n", yytext);}
[e][n][d]            {printf("%s: ENDSYM\n", yytext);}
[i][f]             {printf("%s: IFSYM\n", yytext);}
[t][h][e][n]           {printf("%s: THENSYM\n", yytext);}
[e][l][s][e]           {printf("%s: ELSESYM\n", yytext);}
[w][h][i][l][e]          {printf("%s: WHILESYM\n", yytext);}
[d][o]             {printf("%s: DOSYM\n", yytext);}
[c][a][l][l]           {printf("%s: CALLSYM\n", yytext);}
[c][o][n][s][t]          {printf("%s: CONSTSYM\n", yytext);}
[t][y][p][e]           {printf("%s: TYPESYM\n", yytext);}
[v][a][r]            {printf("%s: VARSYM\n", yytext);}
[p][r][o][c][e][d][u][r][e]      {printf("%s: PROCSYM\n", yytext);}
{INTCON}         {printf("%s: INTCON\n", yytext);}
{IDENT}          {printf("%s: IDENT\n", yytext);}
[\~]|[\!]|[\@]|[\#]|[\$]|[\%]|[\^]|[\&]|[\_]|[\\]              {printf("%s: ERROR\n", yytext);}

 /* end */

\n               {}
.                {}
%%
int yywrap() { return 1; }
int main(int argc, char **argv)
{
    if (argc > 1) {
        if (!(yyin = fopen(argv[1], "r"))) {
            perror(argv[1]);
            return 1;
        }
    }
    while (yylex());
    return 0;
}
```

图10 笔者编写的实验二程序

图11 实验二通过截图

笔者代码如图10所示，实验结果如图11所示。

实验三的要求与解析：

1.用户编程要求

设计并实现一个C语言的词法分析程序。
要求:

1. 识别单词符号并以记号形式输出，并标出该单词符号所在行数。应考虑的单词符号见输出格式；
2. 能够识别并跳过注释；
3. 能够检查到错误的词法；
4. 能够统计行数、各个单词符号的类别数，以及词法错误数。

注：子任务中未提及的部分可由同学们自行考虑合理的输出逻辑（包括单词符号的识别与错误处理等），评判所用测试集不能代表词法分析的所有任务。

图12 实验三编程要求

实验要求如图12，我们可以得知，题目希望我们实现一个基于C11标准的C语言词法分析器，可以**识别**任意一段合法的C语言代码，还能够给出程序的**信息**和检出的**错误**数量；此外任务还提示我们可以在完成任务之外，额外选配一些功能，为我们留足了**发挥空间**。

2.平台环境说明

编译器版本：gcc7.3.0
OS版本：Debian GNU/Linux 9

图13 实验三环境说明

由图13我们可以得知，平台的实验环境为gcc7+Linux9的结合，也不采用非标准库，故正常编写即可。

3.状态图与自动机

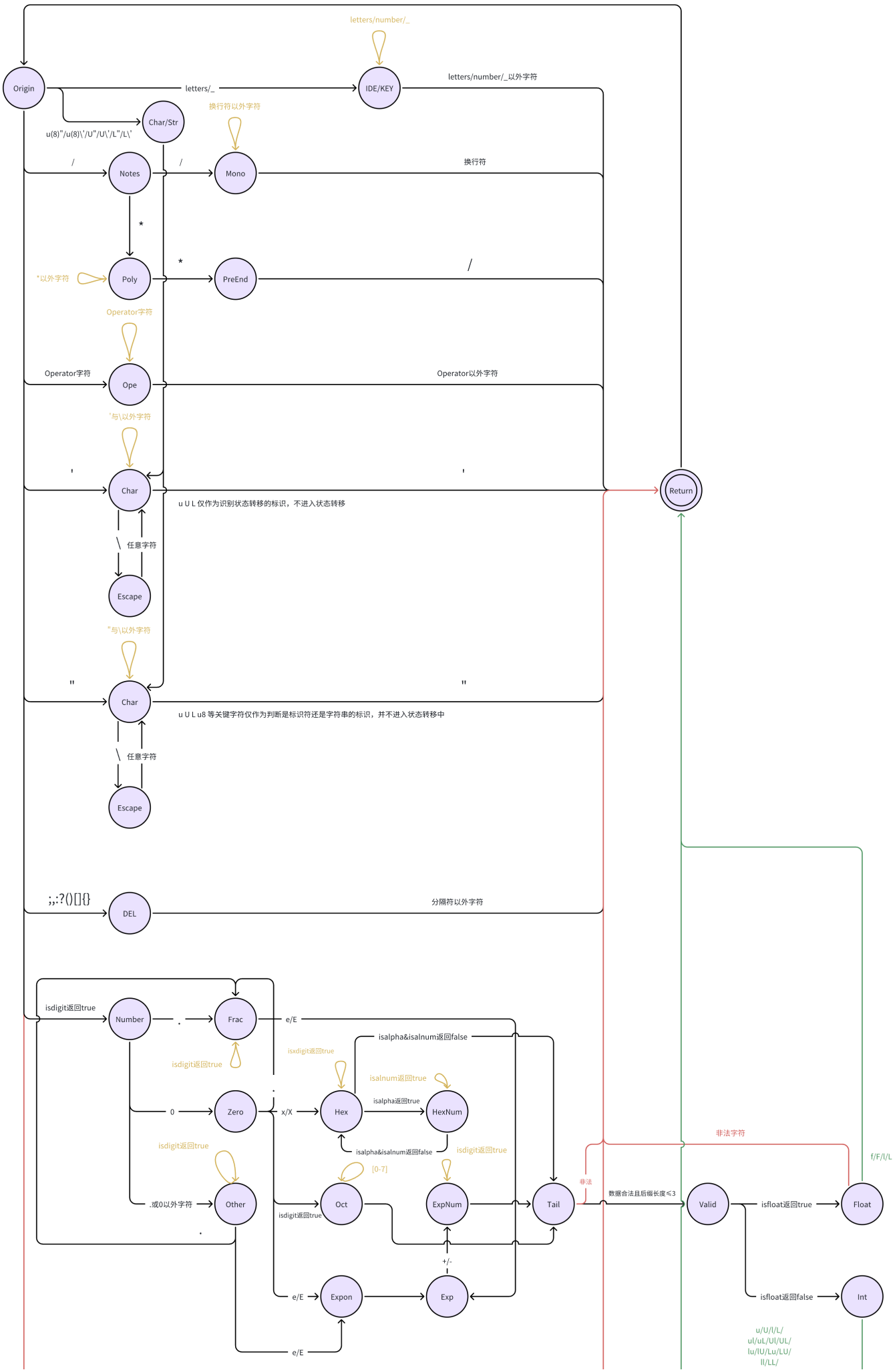

letters/number/_
Origin
letters/_
IDE/KEY
letters/number/_以外字符
Char/Str
u(8)"/u(8)\'/U"/U\'/L"/L\'
换行符以外字符
/
Notes
/
Mono
换行符
*
*以外字符
Poly
*
PreEnd
/
Operator字符
Operator字符
Ope
Operator以外字符
'与\以外字符
'
Char
'
Return
u U L 仅作为识别状态转移的标识，不进入状态转移
\
任意字符
Escape
"与\以外字符
"
Char
"
u U L u8 等关键字符仅作为判断是标识符还是字符串的标识，并不进入状态转移中
\
任意字符
Escape
;,:?()[]{}
DEL
分隔符以外字符
isdigit返回true
Number
.
Frac
e/E
isalpha&isalnum返回false
isdigit返回true
isxdigit返回true
isalnum返回true
0
Zero
x/X
Hex
isalpha返回true
HexNum
isalpha&isalnum返回false
非法字符
f/F/l/L
isdigit返回true
isdigit返回true
[0-7]
非法
.或0以外字符
Other
Oct
ExpNum
Tail
数据合法且后缀长度≤3
Valid
isfloat返回true
Float
isdigit返回true
+/-
e/E
Expon
Exp
isfloat返回false
Int
e/E
u/U/l/L/
ul/uL/Ul/UL/
lu/lU/Lu/LU/
ll/LL/

图14 实验三词法分析器自动机

针对C语言的C11标准，先按照八种单词符号类型进行初步识别（关键字、标识符、运算符、分隔符、字符常量、字符串常量、数值常量以及错误），在每种类型的分支下，针对每一个范围内的字符做判断和细分识别，遇到复杂的函数和状态变量，就先封装再细化，最终构造出如上图的自动机。

4.测试程序示例

1）输入格式：

从文件流等读入输入文件：给定待读取文件的文件名，利用代码中的字符串变量(它的值就是文件名)，完成输入数据的读取。

文件内容为一段C语言程序源代码，或者为每行一个单词符号的文本。

文件换行格式为 `LF`，即换行只需要考虑 `\n` 一个字符。

2）输出格式：

输出到标准输出（使用 `cout` / `printf` 等输出）中。

假设对一个C语言程序进行一次扫描，共得到n个单词符号，则输出共 `n+3` 行。

第1行到第n行：按从头到尾扫描的顺序，每行输出空格分隔的一个整数和一个 <记号,属性> 形式的单词符号表示。

第n+1行：输出1个整数，代表统计的总行数。

第n+2行：输出空格分隔的7个整数，按顺序分别代表：关键字(KEYWORD)、标识符(IDENTIFIER)、运算符(OPERATOR)、分隔符(DELIMITER)、字符常量(CHARCON)、字符串(STRING)和数值常量(NUMBER)的统计总数。

第n+3行：输出1个整数，代表词法错误(ERROR)的词数。

3）记号属性表：

属性 为原始单词符号对应的字符串；

所有的 **记号** 都是由大写字母构成的字符串，具体如图15所示：

记号类别	值
关键字	KEYWORD
标识符	IDENTIFIER
运算符	OPERATOR
分隔符	DELIMITER
字符常量	CHARCON
字符串	STRING
数值常量	NUMBER

图15 记号类型表

对于词法错误的单词符号，将其记号归类为ERROR，如图16所示：

记号类别	值
词法错误	ERROR

图16 错误类型

4）简单测试样例：

```
while
if
default
```

输入

```
1 <KEYWORD,while>
2 <KEYWORD,if>
3 <KEYWORD,default>
3
3 0 0 0 0 0 0
0
```

输出

5）简单样例分析：

样例的输入中，包含3个关键字。

样例的前3行输出中，前面的整数代表单词符号在程序中的位置(行数)，后面的则为对应单词符号的输出形式。

第4行输出为 `3` ，是因为读取的内容共三行。

第5行输出为 `3 0 0 0 0 0 0` ，是因为只有3个关键字类型的单词符号，其余的种类数为 `0` ，这里单词种类自左向右依次对应关键字、标识符、运算符、分隔符、字符常量、字符串常量以及数值常量。

第6行输出为错误单词符号数，也是 `0` 。

针对实验三的具体设计，我们将会在第二部分——程序设计构思中详细说明

5.实验通过截图

源代码可见第二部分的第6点

图17 实验三通过截图

二、程序设计构思

💡 **目标一：完成基本词法分析功能**

目标二：完成扩展功能，尝试更好的识别能力

1.题目解析

设计并实现 `C` 语言的词法分析程序，要求如下。

1. 可以识别出用 `C` 语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。
2. 可以识别并跳过源程序中的注释。
3. 可以统计源程序汇总的语句行数、单词个数和字符个数，并输出统计结果。
4. 检查源程序中存在的错误，并可以报告错误所在的位置。
5. 发现源程序中存在的错误后，进行适当的恢复，使词法分析可以继续进行，对源程序进行一次扫描，即可检查并报告出源程序中存在的词法错误。 采用 `C` 作为实现语言，手工编写词法分析程序。

2.本地实验环境

- Microsoft Windows 11
- Microsoft Visual Studio Community 2022 (64 位) v17.8.3
- Microsoft .NET Framework v4.8.04161

3.模块设计

1）模块划分

各模块及其关系如图18所示，其中绿色代表函数群的入口，红色代表出口，紫色代表主要函数类，蓝色代表重要辅助函数，黄色代表语句模块：

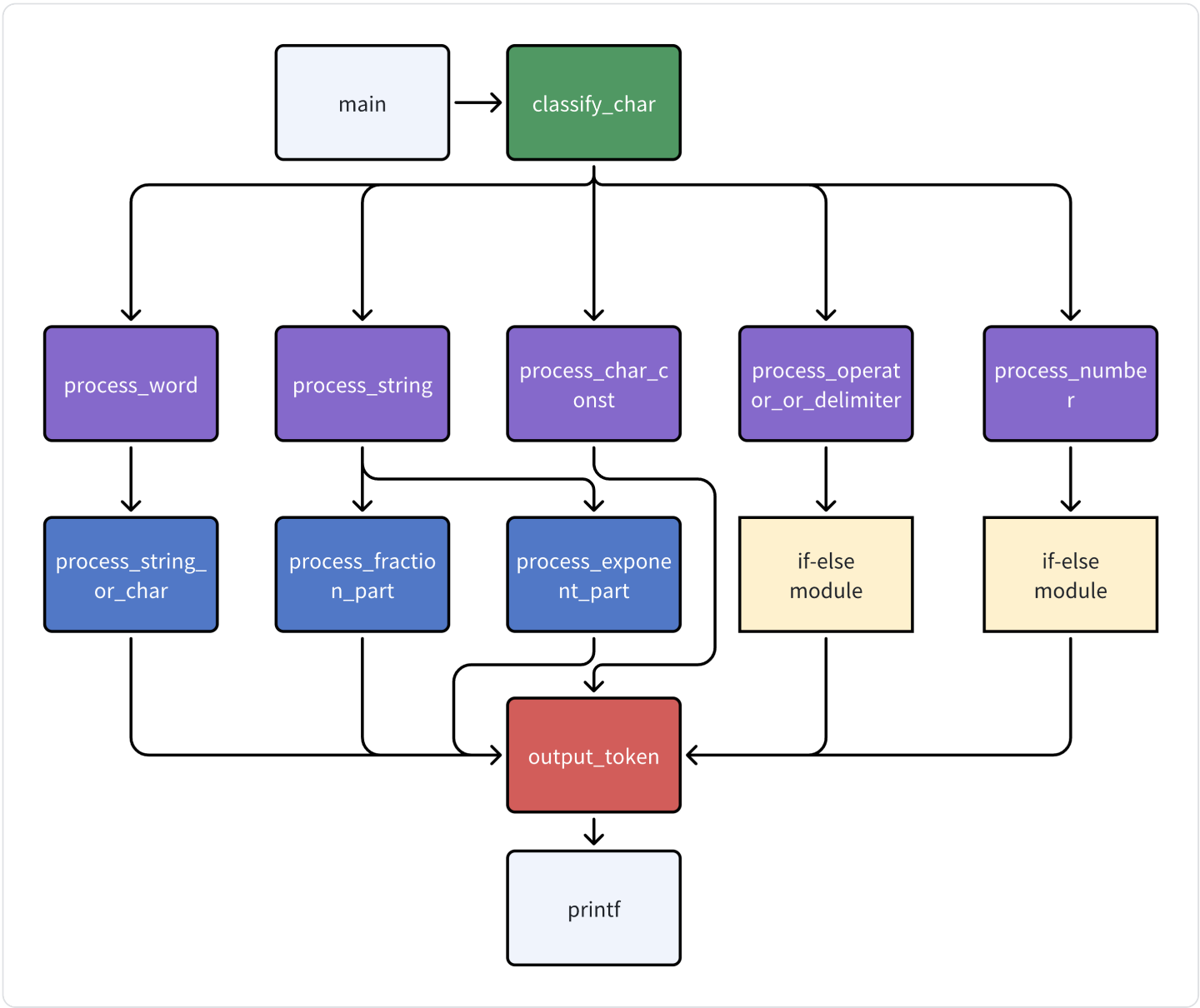


图18 函数模块设计以及调用关系

2）记号表

我将记号分为以下八类：

1.KEYWORD，关键字

在C11标准中共有44个，这里我们选择常用的32个关键字进行识别："char", "double", "enum", "float", "int", "long","short", "signed", "struct", "union", "unsigned", "void","for", "do", "while", "break", "continue", "if", "else","goto", "switch", "case", "default", "return", "auto","extern", "register", "static", "const", "sizeof", "typedef","volatile"

补充：C11其他关键字为 `_Bool`、`_Complex`、`_Imaginary`、`inline`、`restrict`、`_Alignas`、`_Alignof`、`_Atomic`、`_Generic`、`_Noreturn`、`_Static_assert`、`_Thread_local`，很多都属于系统调用/编译关键字，在书写程序中不常用，但仍然推荐加入到考核标准中。

2.IDENTIFIER，标识符

是由数字、下划线、小写及大写拉丁字母和以 `\u` 及 `\U` 转义记号指定的 Unicode 字符组成的任意长度序列。合法的标识符必须以非数字字符开始。

3.OPERATOR，运算符

C11常用运算符如下表所示

算术运算符	+	-	*	/	%	++	--
关系运算符	==	!=	>=	>	<=	<	
逻辑运算符	&&	||	!				
位运算符	&	|	^	~	<<	>>	
赋值运算符	=	+=	-=	*=	/=	%=	<<=
	>>=	&=	|=	^=			
成员运算符	.	->					

表1 运算符的具体划分

4.DELIMITER，分隔符

C11常用的分隔符包括：`;`、`,`、`:`、`?`、`(`、`)`、`[`、`]`、`{`、`}`

5.CHARCON，字符常量

a. 是由一对单引号括起来的一个字符序列，可以带有一个前缀，不带前缀的字符常量默认类型为int，这类字符常量称为整数字符常量。

b. 如果字符常量的前缀是L，则其类型为wchar_t；如果前缀是u，类型是char16_t；如果前缀为U，类型为char32_t。wchar_t、char16_t和char32_t等类型的字符常量通称为宽字符常量。

c. 如果字符常量的前缀是u8，则其类型为char8_t（C23起），该类型与unsigned char相同。

6.STRING，字符串常量

是由一对双引号括起来的一个字符序列，并且把空字符‘\0’自动附加到字符串的尾部作为字符串的结束标志，在函数设计当中需要额外注意这一点。

7.NUMBER，数值常量

包括如下类型：

- **前缀（八进制，十进制、16进制以及浮点的数值常量）**

a. 16进制：由0x或者0X开头，由0～9，A~F或a~f组成。

b. 八进制：由数字0开头，0～7组成。

c. 十进制：开头可以使用+-号，0~9组成。

d. 浮点型：开头可以使用+-号，数字0～9及点号组成。

- **中缀（科学记数法）**

a. 计数法中的e可大写可小写；

b. e前后必须有数字，可正可负可为0，不可省略；

c. e后面必须为整数，不能存在变量(可正可负，正号可以省略)。

- **后缀（`u`、`l`、`ul`、`f`后缀；`l`放在小数后面表示该数值为double型）**

a. u或者l（l,u,f大小写均可）在整形数值后面分别表示unsigned int 和long int;同样ul(大写的UL亦可)组合起来表示unsigned long，而不是默认int型。

b. 同样f或者F作为后缀，表示的是float型。而不是默认double。

c. 如果l放在小数后面表示该数值为double型。

- **还有纯数字**

8.ERROR，错误

若词法不属于以上七种类型，则统一归为错误类型，这里挑选测试集的一些进行举例：

a. 字符`@`，其不会出现在关键字、标识符、数字的识别过程中，所以需要在读入过程中直接加入对此字符的识别；

b. 没有闭合的字符常量与字符串常量统一被归为错误类型；

c. 格式错误的关键字（例如数字开头的）；

d. 格式错误的数值常量（科学计数法没有指数项）；

e. 未知的字符。

3）数据结构定义

1.词元标记类型定义

```
// 枚举定义标记类型
typedef enum {
    KEYWORD = 0,
    IDENTIFIER,
    OPERATOR,
    DELIMITER,
    CHARCON,
```

```
    STRING,
    NUMBER,
    ERROR,
    TOKEN_TYPE_COUNT
} TokenType;
```

2.词元字符类型定义

```
typedef enum {
    CHAR_LETTER = 0,
    CHAR_DIGIT,
    CHAR_SINGLE_QUOTE,
    CHAR_DOUBLE_QUOTE,
    CHAR_OTHER
} CharType;
```

3.关键字列表定义

我们为了能够避免某些关键字是其他关键字的前缀导致识别错误的情况（见同一部分第5点下"5)"小项），在这里提前进行了排序，是其他关键字前缀的关键字尽可能靠后，例如下表中的`double`排在`do`前：

```
const char* keywords[] = {
    "char", "double", "enum", "float", "int", "long",
    "short", "signed", "struct", "union", "unsigned", "void",
    "for", "do", "while", "break", "continue", "if", "else",
    "goto", "switch", "case", "default", "return", "auto",
    "extern", "register", "static", "const", "sizeof", "typedef",
    "volatile"
};
```

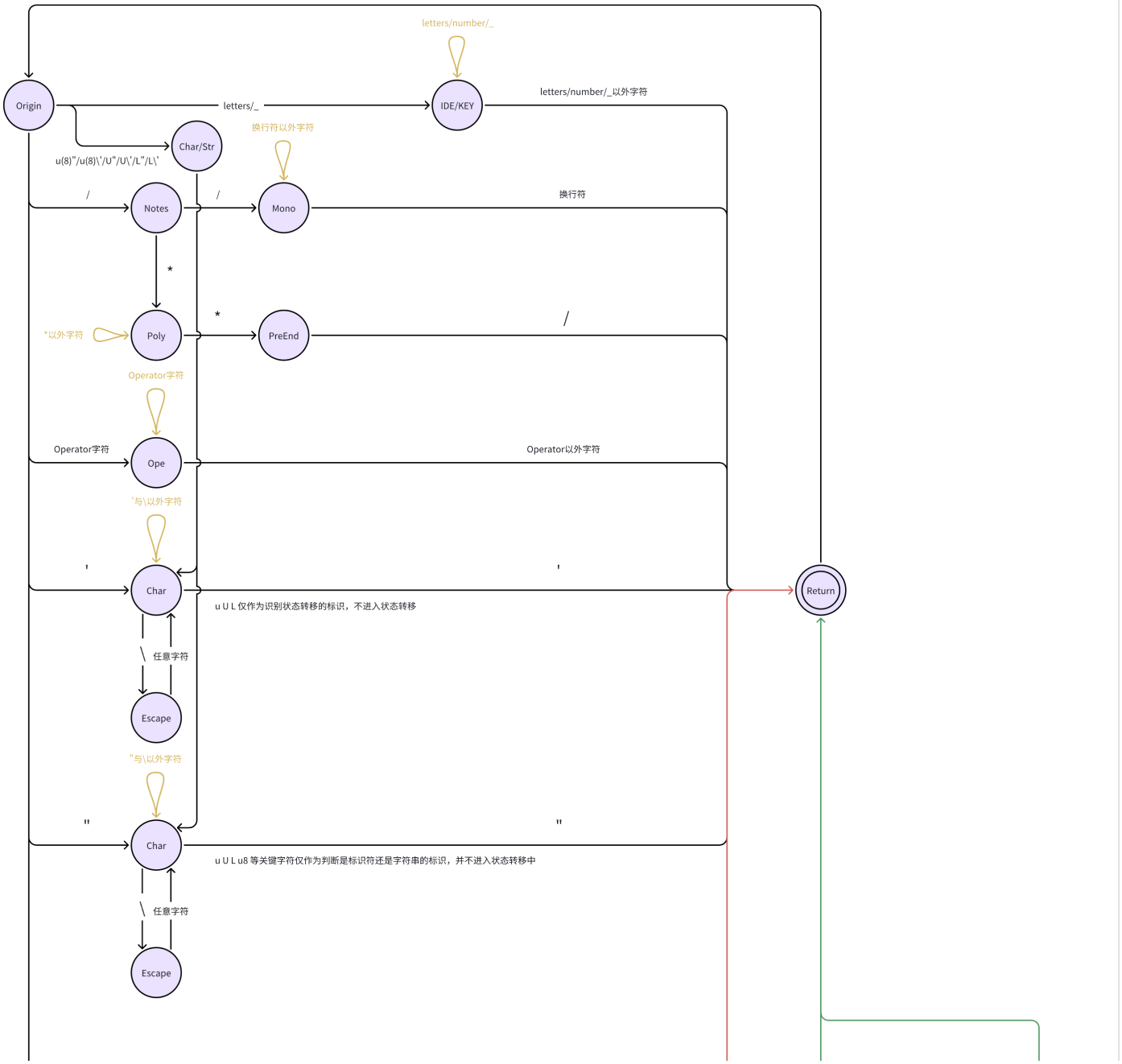
4.词法分析器定义

```
typedef struct {
    FILE* file;
    int line_number;
    long long token_counts[TOKEN_TYPE_COUNT];
    char* lexeme;
    int lexeme_length;
    int lexeme_capacity;
} LexerState;
```

4）特殊情况

- 需要格外注意数值常量的前缀中缀后缀，相对来说其格式过于复杂，识别过程最为困难；
- 字符串常量范围中，会出现夹杂转义引号的情况，需要考虑在内；
- 字符常量可以不是单个字符，此外还有着非常丰富的前缀，也需要考虑识别的实现；
- 读入字符时，例如块注释可能在其中包括多个开始/结束符号，需要考虑建立缓冲区，在难以判断的时候可以回退已读入的字符，方便寻求最优解；

4.自动机设计



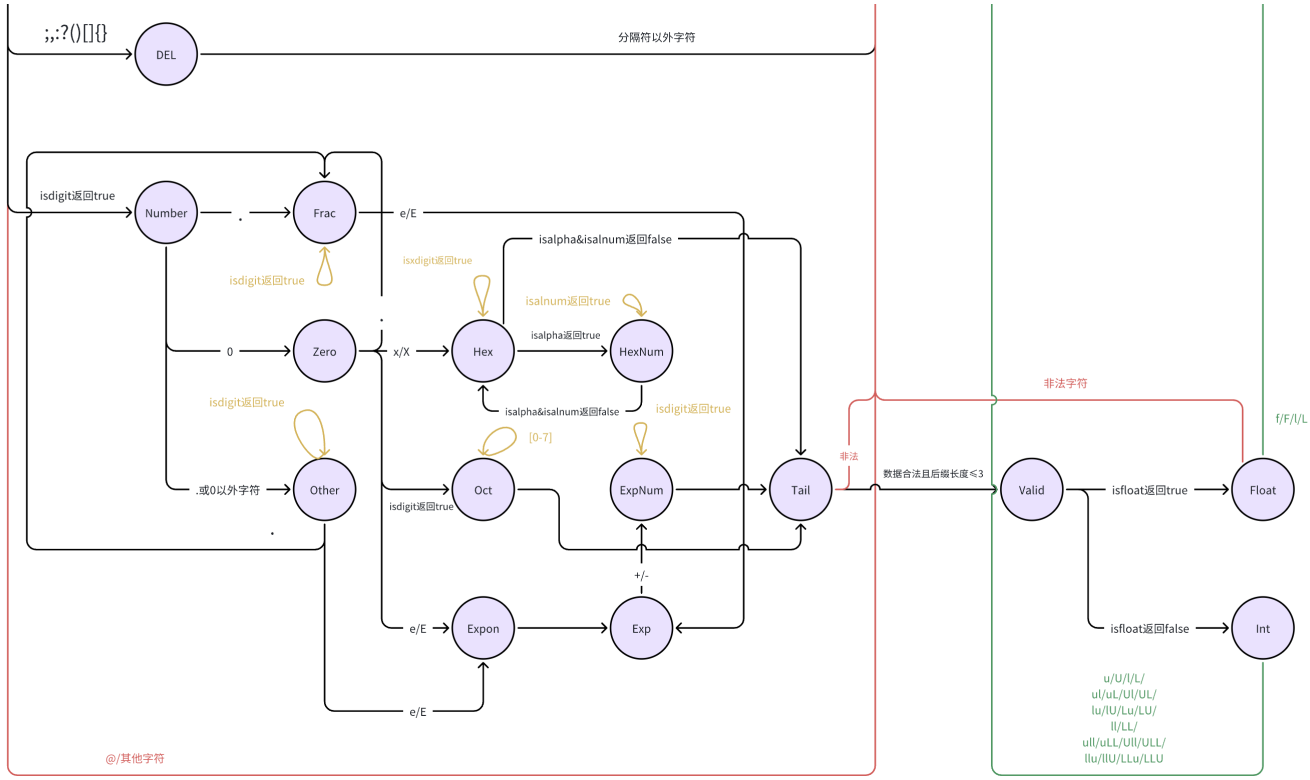


图19 实验三词法分析器自动机

5.函数设计与实现

1）处理可能的前缀、关键字、标识符

标识符是由字母、下划线或字母与数字的组合构成的字符串，用于表示变量名、函数名等。

根据我们上文设计的模块与自动机，当词法分析器读取到符号 `_` 或者大小写字母时，会进入标识符处理函数 `process_word`。词法分析器会继续读取连续的包含下划线和字母数字的字符，并将其加入词素缓冲区中。

读取完成后，词法分析器会检查该字符串是否为关键字，例如 `if`、`for`、`while` 等；如果是关键字，则返回关键字记号，否则返回标识符记号。此外，对于前缀如 `u8`、`L`、`U` 等，词法分析器会判断它们是否为字符串或字符常量的前缀，并进行相应处理，以确保语言中特殊标识符的正确解析。

这种处理方式可以正确区分标识符与关键字，同时支持编程语言中的多种标识符形式。

```
// 处理标识符或关键字，处理字符串和字符常量的前缀
void process_word(LexerState* state, int ch) {
    append_char(state, ch);
    int next_ch;

    // 检查前缀
    if (ch == 'u') {
```

```
        if (next_ch == '8') {
            int peek_ch = read_char(state);
            if (peek_ch == '"' || peek_ch == '\'') {
                process_string_or_char(state, state->lexeme);
            }
        }
        else if (next_ch == '"' || next_ch == '\'') {
            process_string_or_char(state, state->lexeme);
        }
    }
    else if (ch == 'U' || ch == 'L') {
        if (next_ch == '"' || next_ch == '\'') {
            process_string_or_char(state, state->lexeme);
        }
    }

    // 继续读取标识符
    while ((next_ch = read_char(state)), isalnum(next_ch) || next_ch == '_') {
        append_char(state, next_ch);
    }
    unread_char(state, next_ch);

    // 循环对比检查是否为关键字
    for (size_t i = 0; i < keyword_count; ++i) {...}

    output_token(state, is_keyword ? KEYWORD : IDENTIFIER, state->lexeme);
}
```

以上 `process_word` 函数调用带前缀的字符常量/字符串常量处理函数 `process_string_or_char`，其实现方法如下：

```
// 处理带前缀的字符串或字符常量
void process_string_or_char(LexerState* state, const char* prefix) {
    int ch = state->lexeme[state->lexeme_length - 1]; // 已经读取了引号
    int is_string = (ch == '"');
    int is_valid = 1;

    while ((ch = read_char(state)) != EOF && ch != '\n') {
        append_char(state, ch);
        if (ch == '\\') {
            ch = read_char(state);
            if (ch == EOF || ch == '\n') {
                is_valid = 0;
                break;
            }
```

```
            append_char(state, ch);
        }
        else if ((is_string && ch == '"') || (!is_string && ch == '\'')) {
            // 结束引号
            break;
        }
    }

    ...

    if (is_valid) {
        output_token(state, is_string ? STRING : CHARCON, state->lexeme);
    }
    else {
        output_token(state, ERROR, state->lexeme);
        if (ch == '\n') {
            // 读取到换行符后再增加行号
            state->line_number++;
        }
    }
}
```

2）处理数值常量

当词法分析器遇到数字或者符号 `.` 后跟数字时，会进入数字常量处理函数 `process_number`。数字常量的处理分为以下几种情况：

- **十进制整数**：如果数字是十进制整数，词法分析器会处理连续的数字部分，直到遇到其他非数字字符为止。若遇到小数点 `.` 则转为浮点数处理，若遇到 `e/E` 则处理为科学计数法表示的浮点数。
- **八进制整数**：以 `0` 开头的数字被视为八进制数，后续只能包含 `0-7` 之间的数字。如果遇到非法字符如 `8` 或 `9`，则返回错误。八进制整数的解析会持续读取直到遇到非八进制数字的字符。
- **十六进制整数**：以 `0x` 或 `0X` 开头的数字被视为十六进制数，后续可以包含 `0-9`、`a-f` 或 `A-F` 的字符。如果遇到小数点，则需要确保小数点后至少有一个十六进制数字，否则返回错误。此外，十六进制数至少需要包含一个有效的十六进制数字，否则会返回错误。
- **浮点数**：浮点数的解析分为两部分：小数部分和指数部分。若遇到小数点 `.`，则会读取小数部分的数字；若遇到 `e/E`，则会读取科学计数法表示的指数部分，指数部分可以包含 `+` 或 `-` 符号，之后必须跟随至少一个数字。浮点数也可以包含合法的后缀（如 `f`、`l` 等）。
- **特别注意：**遇到以数字开头的标识符，属于ERROR，当程序进入数字识别状态时，它会检查遇到的每个字符。如果遇到一个不属于数字但可以包含在标识符中的字符，程序就会判断为发生了标识符开头错误。

识别数值常量的过程较复杂，这里我们再次梳理出一个DFA：

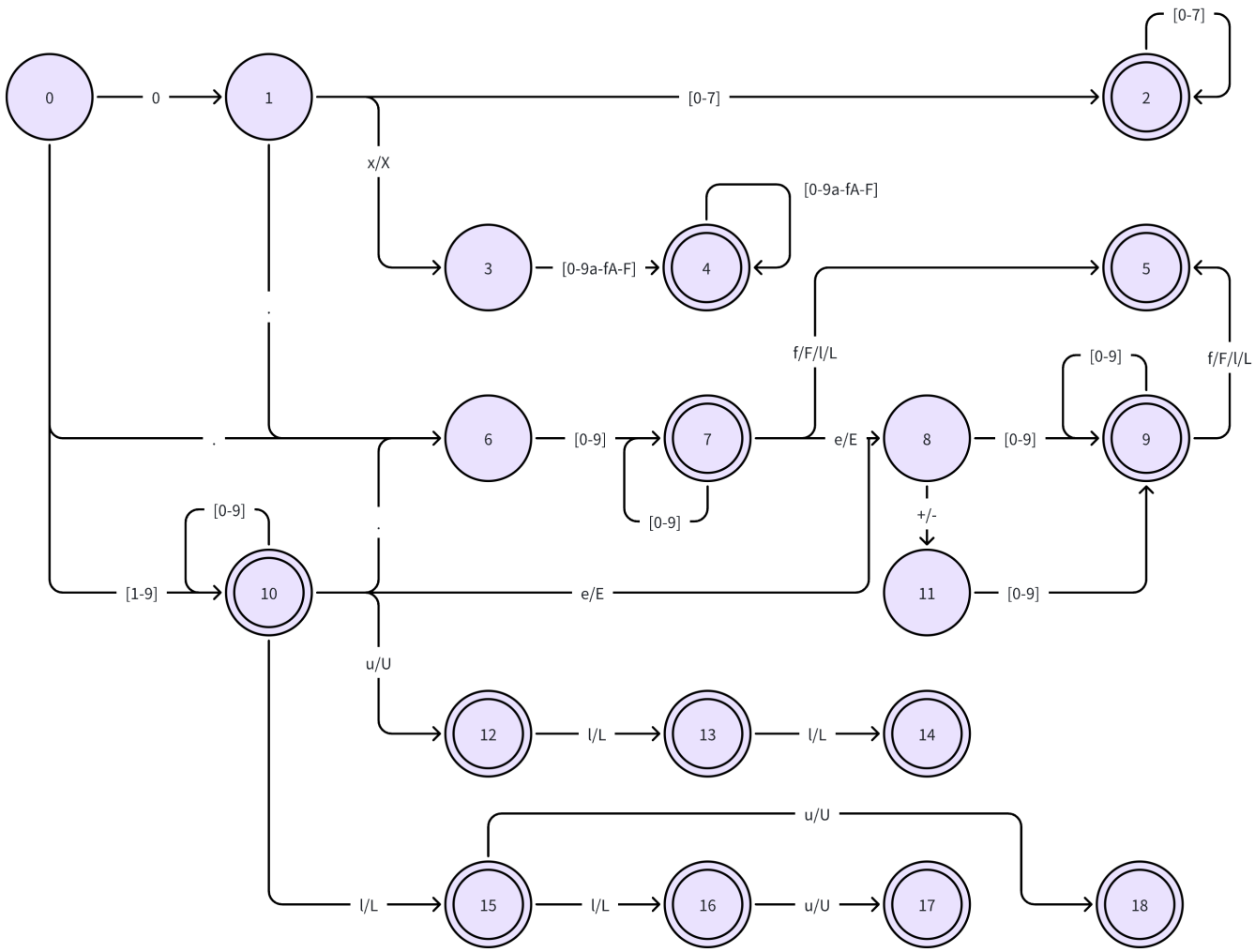


图20 数值常量识别自动机

以下均为处理函数实现：

```
// 处理数字，包括整数、浮点数、十六进制、八进制等
void process_number(LexerState* state, int ch) {
    if (ch == '.') {
        // 处理以 . 开头的浮点数
        is_valid = process_fraction_part(state);
    }
    else if (ch == '0') {
        if (next_ch == 'x' || next_ch == 'X') {
            // 处理十六进制数
            ...
        }
        else if (isdigit(next_ch) && next_ch != '8' && next_ch != '9') {
            // 处理八进制数
            ...
        }
```

```
        else if (next_ch == '.') {
            // 处理浮点数，如 0.5
            process_fraction_part(state);
        }
        else if (next_ch == 'e' || next_ch == 'E') {
            // 处理科学计数法，如 0e10
            process_exponent_part(state);
        }
    }
    else {
        // 处理十进制数或浮点数
        ...
    }

    // 处理数字后缀
    if (is_valid) {
        char suffix[4] = { 0 }; // 最大后缀长度为3
        ...
        if (is_float) {
            // 检查浮点数合法后缀
            valid_suffix = is_valid_float_suffix(suffix);
        }
        else {
            // 检查整数合法后缀
            valid_suffix = is_valid_integer_suffix(suffix);
        }

    }
    //输出函数
}
```

以上 `process_word` 函数调用针对小数和指数部分的判断函数分别为 `process_fraction_part` 和 `process_exponent_part`，实现方法如下：

```
// 处理小数部分
int process_fraction_part(LexerState* state) {
    if (next_ch == 'e' || next_ch == 'E') { // 检查指数后缀
        append_char(state, next_ch);
        if (!process_exponent_part(state)) {
            return 0;
        }
    }
    return has_digits;
}
```

```
// 处理科学计数法的指数部分
int process_exponent_part(LexerState* state) {
    if (next_ch == '+' || next_ch == '-') {
        ...
    }
    while (isdigit(next_ch)) {
        ...
    }
    ...
}
```

除此之外还有检测后缀的函数`is_valid_integer_suffix`和`is_valid_float_suffix`，实现如下：

```
// 检查是否为有效的整数后缀
int is_valid_integer_suffix(const char* suffix) {
    const char* valid_suffixes[] = {
        "u", "U", "l", "L",
        "ul", "uL", "Ul", "UL",
        "lu", "lU", "Lu", "LU",
        "ll", "LL",
        "ull", "uLL", "Ull", "ULL",
        "llu", "llU", "LLu", "LLU"
    };
    ...
    return suffix[0] == '\0'; // 空后缀也是合法的
}
```

```
// 检查是否为有效的浮点数后缀
int is_valid_float_suffix(const char* suffix) {
    const char* valid_suffixes[] = { "f", "F", "l", "L" };
    ...
    return suffix[0] == '\0'; // 空后缀也是合法的
}
```

3）处理字符串字面量

字符串字面量是由双引号包围的字符序列。

当词法分析器读取到符号 `"` 或者字母 `L` 紧跟一个 `"` 时，会进入字符串字面量处理函数 `process_string`。与字符常量处理类似，词法分析器会不断读取字符，直到遇到结束符号 `"`，并将这些字符加入词素缓冲区。如果字符串未闭合（即未找到匹配的结束符号 `"`），则返回未闭合字符串错误。

字符串字面量中也可以包含转义字符（例如 `\tab` 表示制表符），词法分析器会正确处理这些转义字符，以确保字符串的内容被正确解析。特别注意到，字符常量和字符串常量中的转义字符 `\` 的处理。这是因为转义字符的存在使得我们在解析过程中遇到的单引号 `'` 和双引号 `"` 可能并不代表常量的结束，而是常量内容的一部分。

为了准确处理这种情况，每当程序在解析过程中遇到反斜杠 `\` 时，我让程序自动跳过对下一个字符的常规判断。相反，程序会直接将该字符及其后的字符写入到当前正在解析的常量内容中。这样，即使常量中包含了引号，程序也不会错误地将它们识别为常量的结束标志。

```
// 处理字符串字面量
void process_string(LexerState* state) {
    while ((ch = read_char(state)) != EOF && ch != '\n') {
        append_char(state, ch);
        if (ch == '\\') {
            ch = read_char(state);
            if (ch == EOF || ch == '\n') {
                is_valid = 0;
                break;
            }
            append_char(state, ch);
        }
        else if (ch == '"') {
            break;
        }
    }
}
```

4）处理字符常量

字符常量是由单引号包围的字符或字符序列。

当词法分析器读取到符号 `'` 或者字母 `L` 紧跟一个 `'` 时，会进入字符常量处理函数 `process_char_const`。在字符常量处理中，词法分析器会不断读取字符，直到遇到结束符号 `'`，并将这些字符加入词素缓冲区。如果字符常量未闭合（即未找到匹配的结束符号 `'`），则返回未闭合字符常量错误。

此外，字符常量中可以包含转义字符（例如 `\n` 表示换行符），这些转义字符也会被正确解析和处理，在上方的"3)"小项内已经解释过。

```
// 处理字符常量，允许单引号内有多个字符
void process_char_const(LexerState* state, int ch) {
    while ((ch = read_char(state)) != EOF && ch != '\n') {
        if (ch == '\\') { // 检查转义符号
            ...
        }
        else if (ch == '\'') { // 检查句内单引号
            break;
        }
    }

    if (ch != '\'') {
        is_valid = 0;
    }
}
```

5）处理运算符与分隔符

一般符号：

运算符与分隔符在编程语言中非常重要，用于分隔不同的语句，参与组成不同的表达式。

词法分析器在遇到符号时，会进入 `process_operator_or_delimiter` 函数进行处理。该函数负责区分标点符号的类型，并根据符号的不同进行匹配。符号的匹配过程是尽可能贪心的，即优先匹配更长的符号。例如，对于表达式 `i+++=+` ，词法分析器会将其解析为五个记号，分别是 `i` 、`++` 、 `==` 和 `+` 。像这里，有些运算符 `+` 可能是其他运算符 `+=` 的前缀，如果处理不当，就会导致错误的识别。首先检查最长的运算符，然后才是较短的运算符。这样做的原因是，如果一个较长的运算符被错误地识别为一个较短的运算符，那么就会丢失一些重要的信息。这种贪心匹配有助于正确识别多字符运算符，从而确保代码的语义正确解析。

特殊情况：

单行注释处理

当词法分析器在代码中遇到符号 `//` 时，会进入单行注释处理阶段。在 `process_operator_or_delimiter` 函数中，当识别到 `//` 后，词法分析器会继续读取后续的所有字符，直到遇到换行符或者文件末尾为止，标志着注释的结束。在此过程中，注释中的所有字符都会被忽略，不会生成任何标记。这样做的目的是为了确保注释内容不会对代码的执行产生影响，同时保留注释作为代码的说明性文字。

块注释处理

当词法分析器匹配到符号 `/*` 后，会进入块注释处理阶段。在这种情况下，词法分析器会不断读取后续字符，直到遇到符号 `*/` 或者文件末尾。如果在文件末尾之前没有遇到 `*/`，则说明块注释未闭合，此时会返回块注释未闭合错误。块注释的处理同样在 `process_operator_or_delimiter` 函数中实现，与单行注释不同的是，块注释可以跨越多行，因此词法分析器需要特别处理换行符以正确跟踪代码行数。

错误符号处理：

在词法分析的过程中，可能会遇到无法识别的字符或符号，此时词法分析器会进入错误处理阶段。

在 `process_operator_or_delimiter` 函数中，如果遇到非法的符号，例如 `@`，则会将该符号识别为错误，并输出相应的错误标记。错误符号处理的目的是为了确保代码中的非法字符不会影响正常的词法分析过程，同时能够明确指出代码中存在的问题，方便开发者进行调试和修改。

对于错误符号，词法分析器会将其加入词素缓冲区，并在生成错误标记后继续分析下一个字符，以保持对代码的完整扫描。

根据我们上文设计的模块与自动机，处理运算符和分隔符的函数 `process_operator_or_delimiter` 实现如下：

```
// 处理运算符和分隔符
void process_operator_or_delimiter(LexerState* state, int ch) {
    int next_ch = read_char(state);

    if (ch == '.' && isdigit(next_ch)) {
        // 处理浮点数，如 .5
        process_number(state, ch);
    }

    append_char(state, ch);

    // 分隔符处理
    if (strchr(";,:?[](){}", ch)) {
        ...
    }

    // 处理多字符运算符
    if (ch == '+' && (next_ch == '+' || next_ch == '=')) {
        ...
    }
    else if (ch == '-' && (next_ch == '-' || next_ch == '=' || next_ch ==
'>')) {
```

```
        ...
    }
    else if (ch == '*' && next_ch == '=') {
        ...
    }
    else if (ch == '/' && next_ch == '=') {
        ...
    }
    else if ((ch == '%' || ch == '^' || ch == '&' || ch == '|') && next_ch == '=') {
        ...
    }
    else if ((ch == '<' || ch == '>') && (next_ch == '=' || next_ch == ch)) {
        append_char(state, next_ch);
        if (next_ch == ch) {
            // 可能是 <<= 或 >>= 等
            ...
        }
    }
    else if ((ch == '=' || ch == '!') && next_ch == '=') {
        ...
    }
    else if ((ch == '&' && next_ch == '&') || (ch == '|' && next_ch == '|')) {
        ...
    }
    else if (ch == '/' && (next_ch == '/' || next_ch == '*')) {
        // 处理注释
        if (next_ch == '/') {
            // 单行注释
            ...
        }
        else {
            // 多行注释
            ...
        }
    }
    else if (ispunct(ch) && strchr("~!@#$%^&*-+=|\\:;\"'<>./?", ch)) {
        // 单字符运算符或分隔符
        if (ch == '@') {
            // 处理非法字符
            output_token(state, ERROR, state->lexeme);
        }
        else {
            output_token(state, OPERATOR, state->lexeme);
        }
        unread_char(state, next_ch);
    }
```

```
68      else {
69          // 处理未识别的字符
70          ...
71      }
72  }
```

三、词法分析实验总结

本次实验中，我亲手编写了一个C语言的词法分析程序，这让我对词法分析的流程有了更深刻的理解，并且加深了我对相关知识点的掌握。

为了实现C语言的词法分析，我首先参考了C11的**ISO标准**，严格按照标准中定义的词法规则来编写程序。标准中对各类记号的文法定义非常清晰，这在一定程度上降低了我编程的难度。然而，在实现过程中，我也发现了许多需要注意的细节。在第一次编写程序时，我遇到了一些**问题**，例如缓冲区的回退异常、数值常量后缀识别缺陷、字符串前缀识别缺陷、错误符号识别不全等，通过多次调整文法自动机和精心面向样例进行调试，我最终解决了上面提到的所有bug。

在本次实验中，我将**自动机**的思想**融入**到了代码中，通过不同的处理函数来实现对不同种类记号的处理，这样的代码风格更易读一些。例如，我编写了 `process_word` 、 `process_number` 、 `process_string` 、 `process_char_const` 和 `process_operator_or_delimiter` 等函数，分别用于处理标识符与关键字、数字、字符串、字符常量，以及操作符和分隔符。这些函数通过逐字符地分析输入，根据C语言的词法规则，识别并分类不同的记号。我没有完全照搬教材上的实现方式，也没有显示地实现一个DFA（确定性有限自动机），也没有明显的展示状态转移过程，这样的实现过程较为死板与冗杂，大量的中间状态和条件判断实在不**简洁**。

此外，我在缓冲区的实现上也做了一些创新。我没有使用教材上的方式，而是利用了C语言的动态内存分配特性，使得维护缓冲区变得更加容易，大大减少了编程的复杂度。例如，我使用 `malloc` 和 `realloc` 来动态地分配和调整缓冲区的大小，这样可以更**灵活**地处理不同长度的记号。

在实现过程中，我还大量运用了 `ctype.h` 库中的 `is` 族函数来辅助判断字符的类型，如 `isalpha` 、 `isdigit` 等。使用这些函数，**简化**了我的代码，提高了程序的可读性和可维护性。

通过本次实验，我从中收获颇丰，不仅对课内**词法分析**有关的知识有了更多的认识，而且我的C语言编程能力和英文文献阅读能力也得到了提高，特别是在理解**C语言词法规则**和处理各种编程细节方面。

四、（附录）测试报告

此部分为实验详情，属于实验报告的一部分。

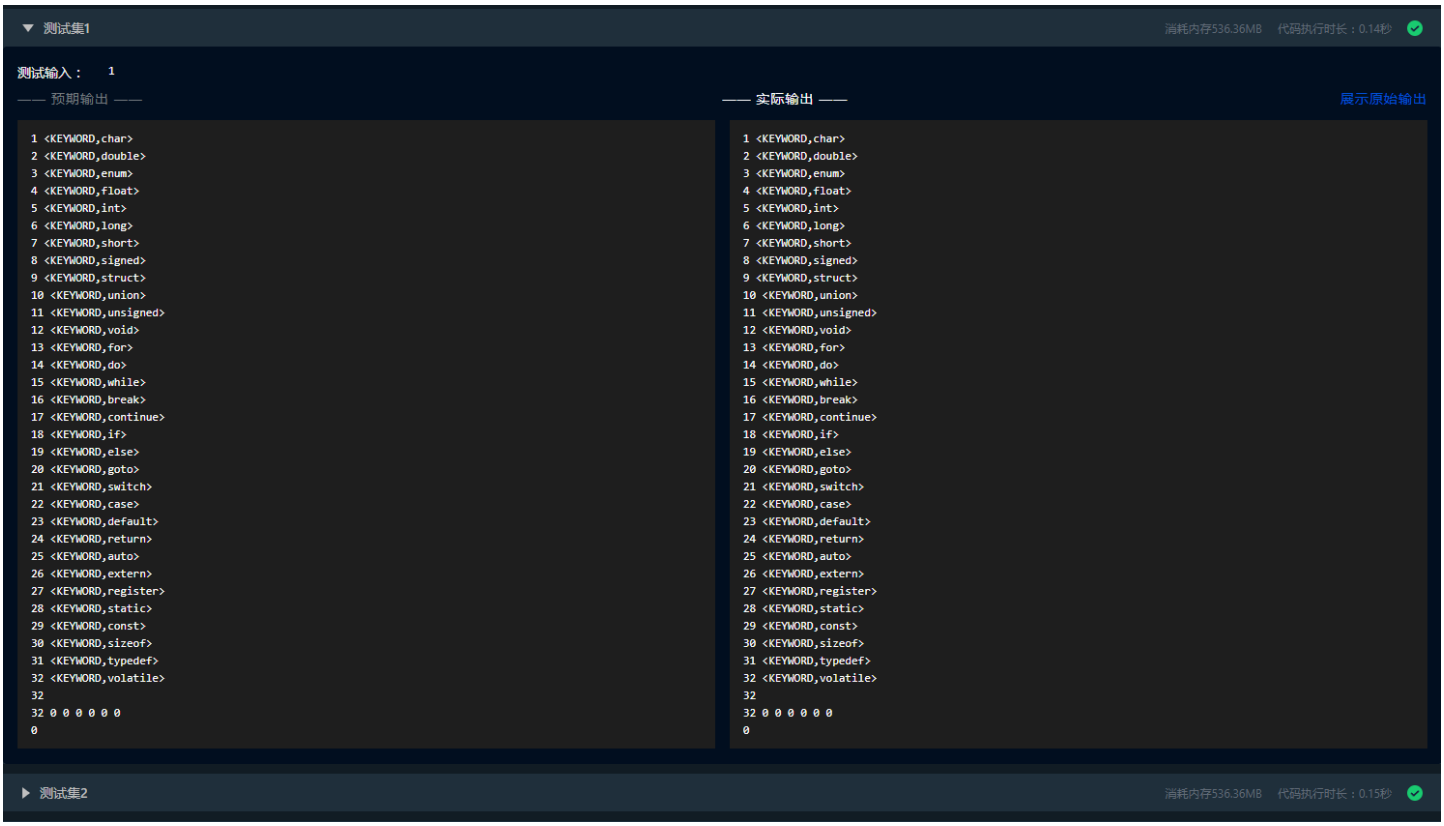
在这里，笔者扒出样例输入与输出，并逐一在本地测试，出错则写明错误原因和修改思路，最终提交。

1.关键字识别

1.样例输入

```
char
double
enum
float
int
long
short
signed
struct
union
unsigned
void
for
do
while
break
continue
if
else
goto
switch
case
default
return
auto
extern
register
static
const
sizeof
typedef
volatile
```

2.运行结果与截图



3.分析说明

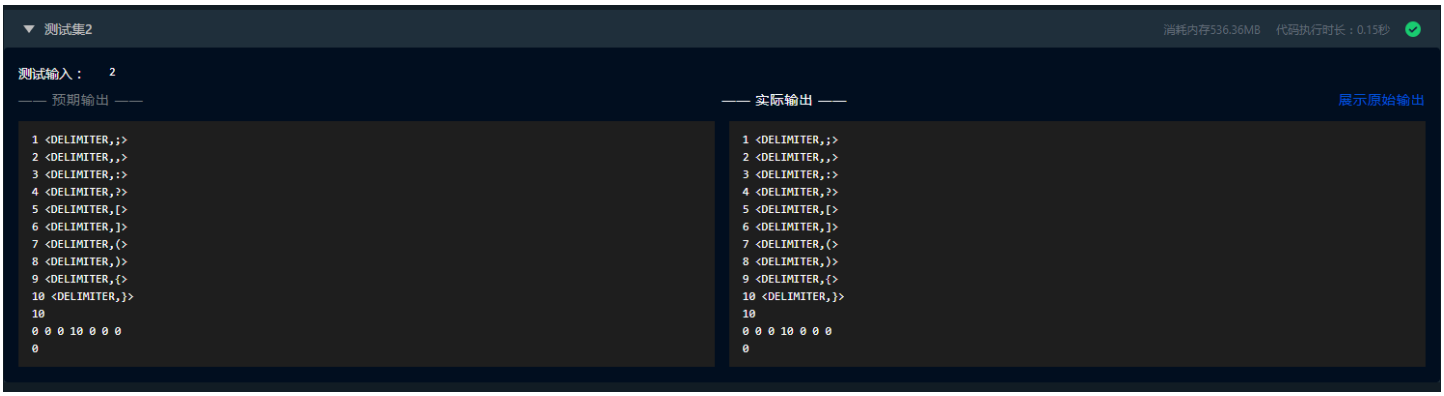
32种常用关键字完全可以识别

2.分隔符识别

1.样例输入

```
;
,
:
?
[
]
(
)
{
}
```

2.运行结果与截图



3.分析说明

10种分隔符完全可以识别

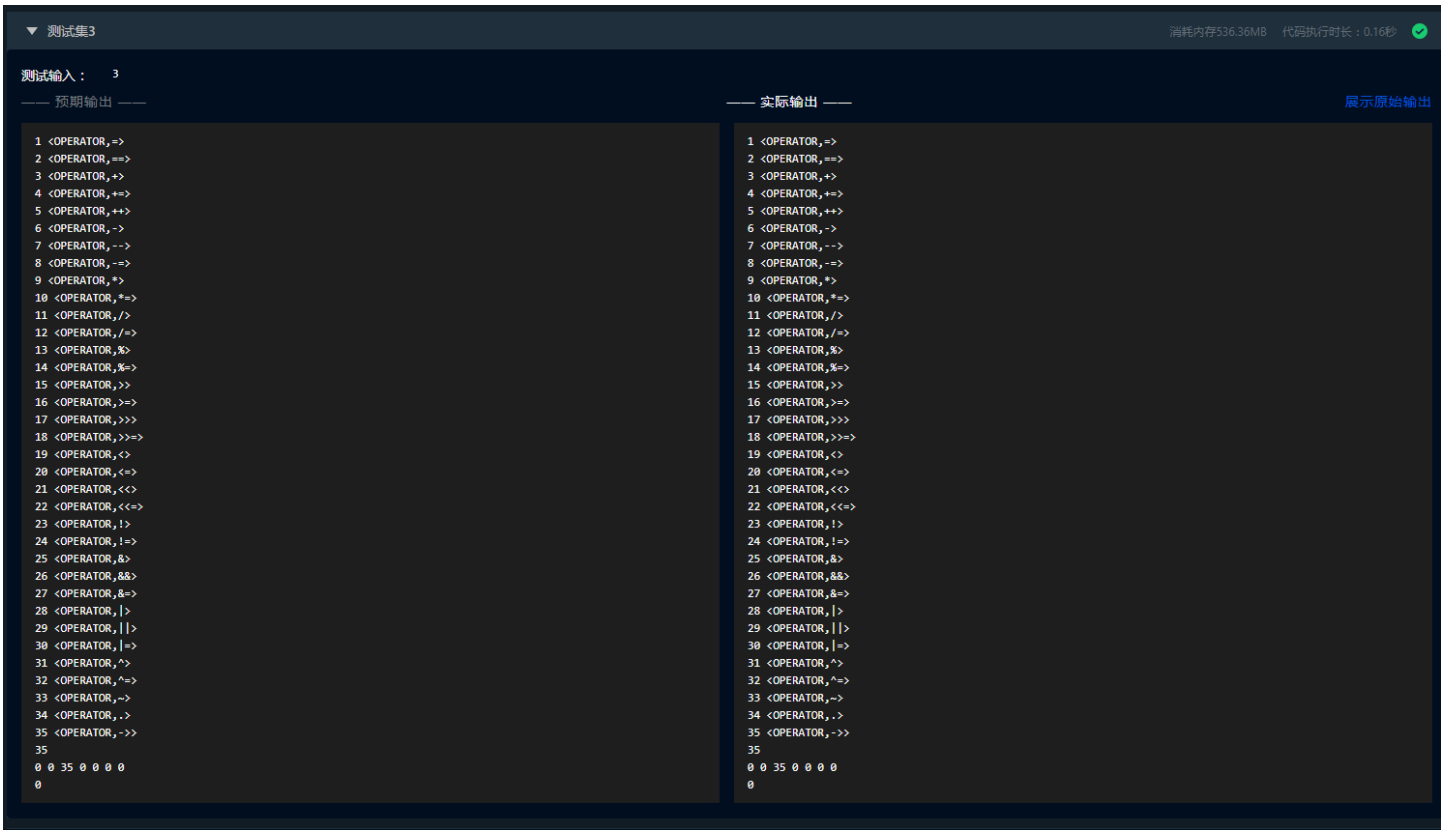
3.分隔符识别

1.样例输入

```
=
==
+
+=
++
-
--
-=
*
*=
/
/=
%
%=
>
>=
>>
>>=
<
<=
<<
<<=
!
!=
&
&&
&=
```

```
|
||
|=
^
^=
~
.
->
```

2.运行结果与截图



3.分析说明

35种常用运算符完全可以识别

4.标识符识别

1.样例输入

```
fa8f7wayhnf74y6t16
fasjsiahnuwfnuw
w_9j9a_k90wjm
_38tj93nwe_
__a8
```

```
6 dodo
7 charif
8 int3
```

2.运行结果与截图

3.分析说明

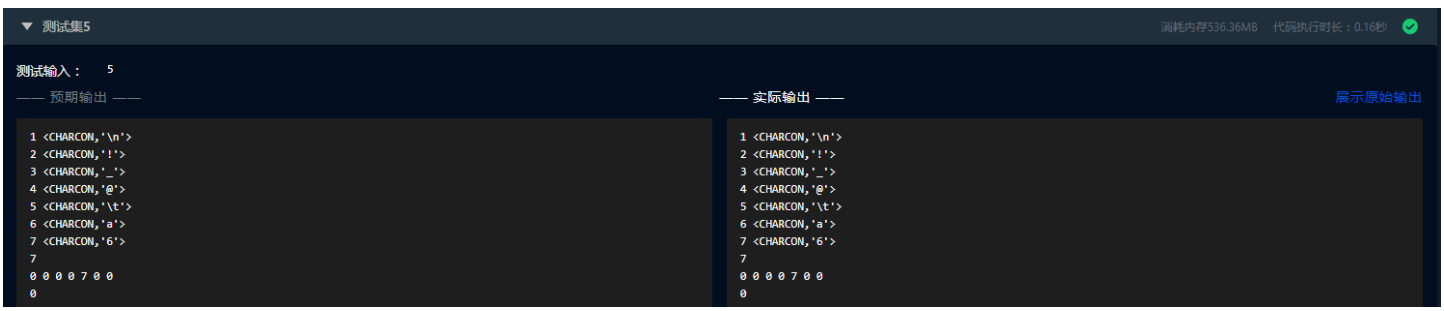
标识符完全可以识别

5.字符常量识别

1.样例输入

```
1 '\n'
2 '!'
3 '_'
4 '@'
5 '\t'
6 'a'
7 '6'
```

2.运行结果与截图



3.分析说明

字符常量完全可以识别

6.字符串常量识别

1.样例输入

```
"1234+abc"
"@"
"'\n'"
"a\""
"//"
"<0>"
"::"
"{[1,2,3]}"
```

2.运行结果与截图

3.分析说明

字符串常量完全可以识别

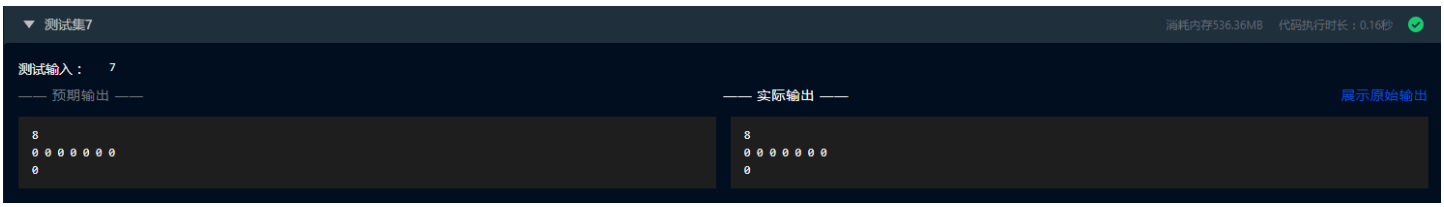
7.注释识别

1.样例输入

```
//test
/*char if @1 */
/***a/****/
/*char;//t
1+;
"'\n'"
```

```
7 {*/
8 /*//*/
```

2.运行结果与截图



3.分析说明

注释完全可以识别，且其中的字符不输出

8.整型识别

1.样例输入

```
1 1
2 1234567890
3 5555
4 10000
5 0
```

2.运行结果与截图

3.分析说明

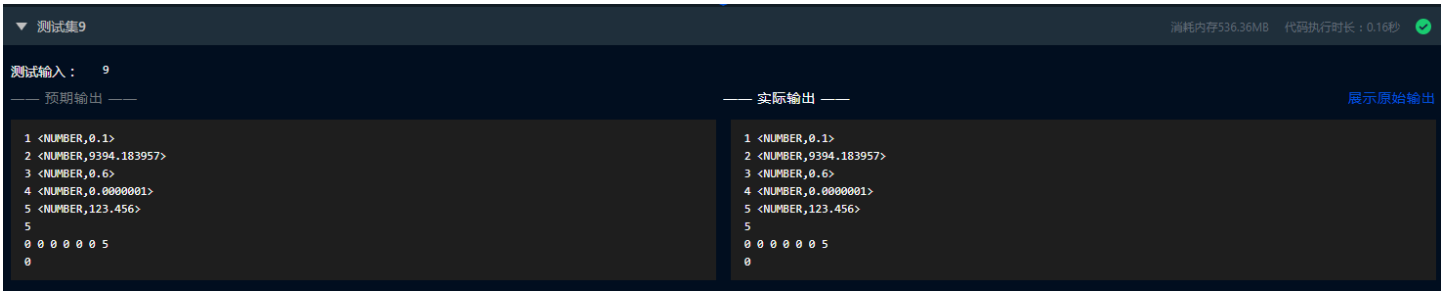
整型数值常量完全可以识别

9.正常浮点数识别

1.样例输入

```
0.1
9394.183957
0.6
0.0000001
123.456
```

2.运行结果与截图



3.分析说明

正常浮点数数值常量完全可以识别

10.指数浮点数识别

1.样例输入

```
6e-10
6E92
0.6e2
5135E+5
9.22E-100
1.0e0
```

2.运行结果与截图

3.分析说明

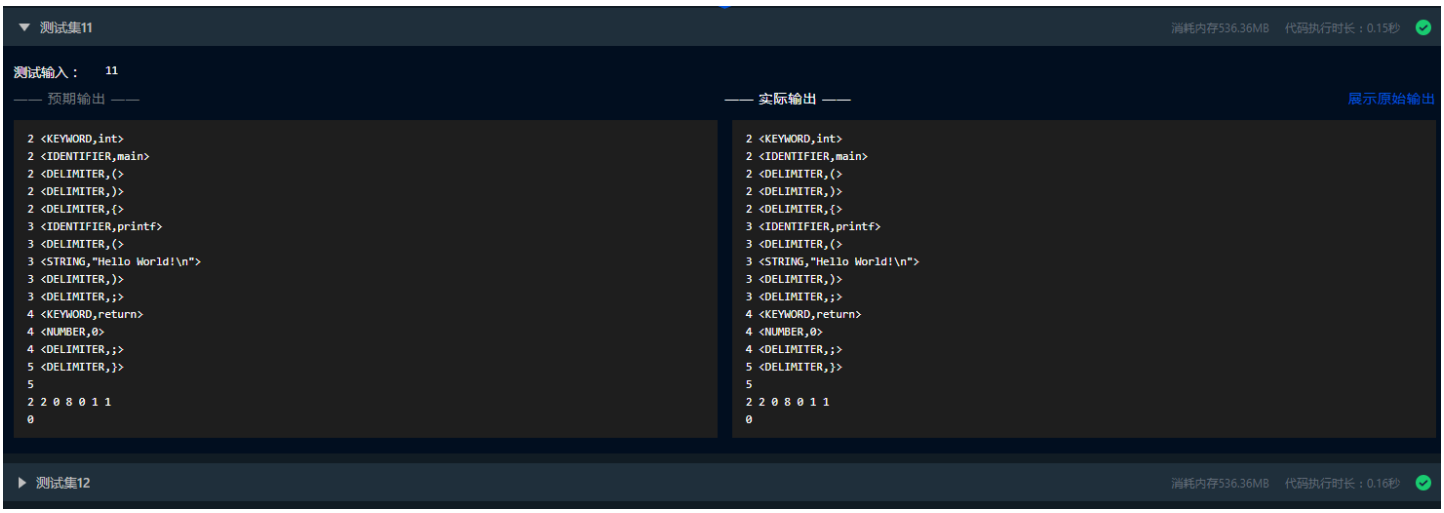
指数浮点数数值常量完全可以识别

11.简单代码块识别

1.样例输入

```

int main(){
        printf("Hello World!\n");
        return 0;
}
```

2.运行结果与截图



3.分析说明

简单代码块完全可以识别

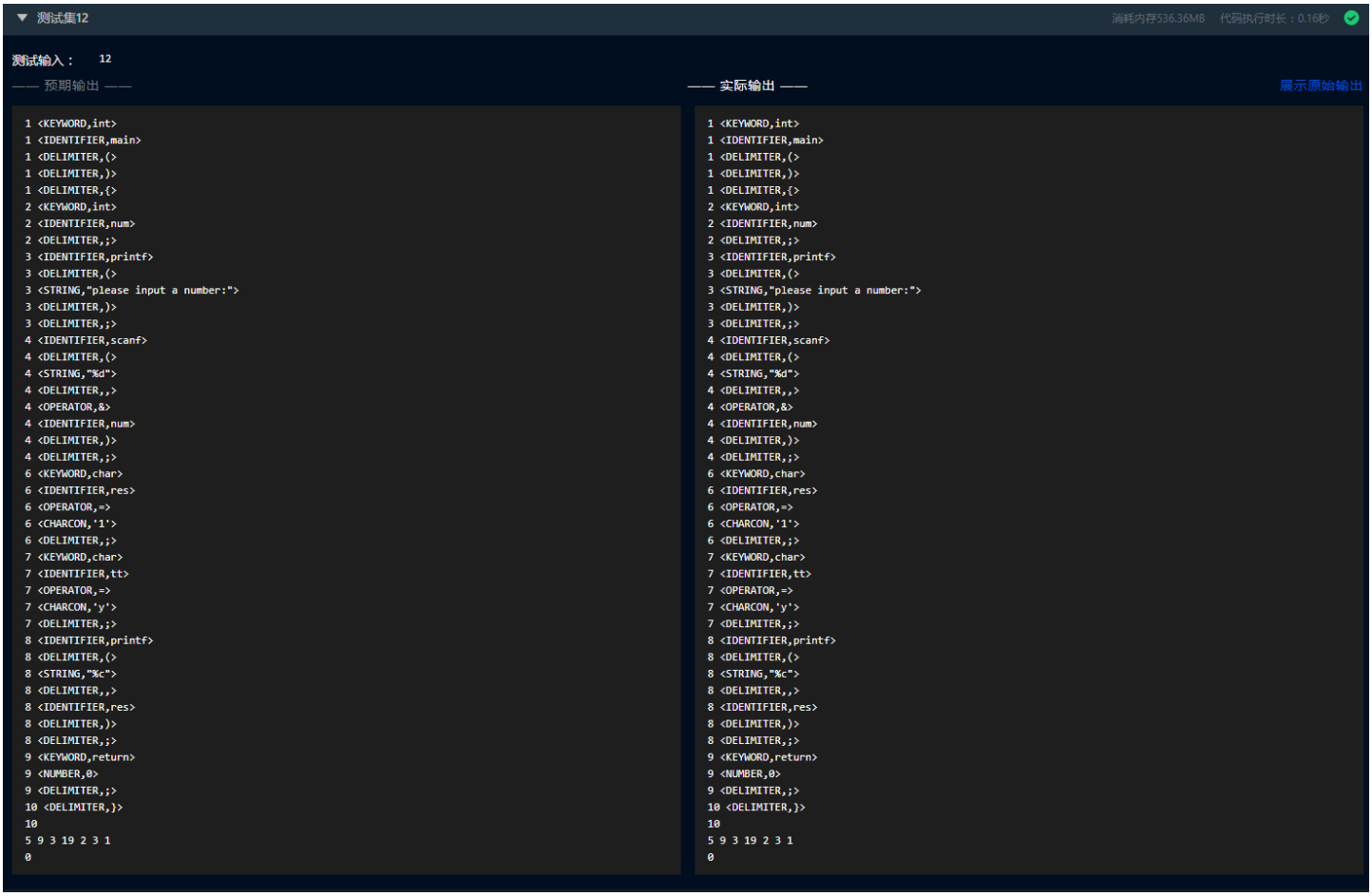
12.简单代码块识别二

1.样例输入

```
int main() {
        int num;
        printf("please input a number:");
        scanf("%d", &num);

        char res = '1';
```

```
        char tt = 'y';
        printf("%c", res);
    return 0;
}
```

2.运行结果与截图



3.分析说明

简单代码块完全可以识别

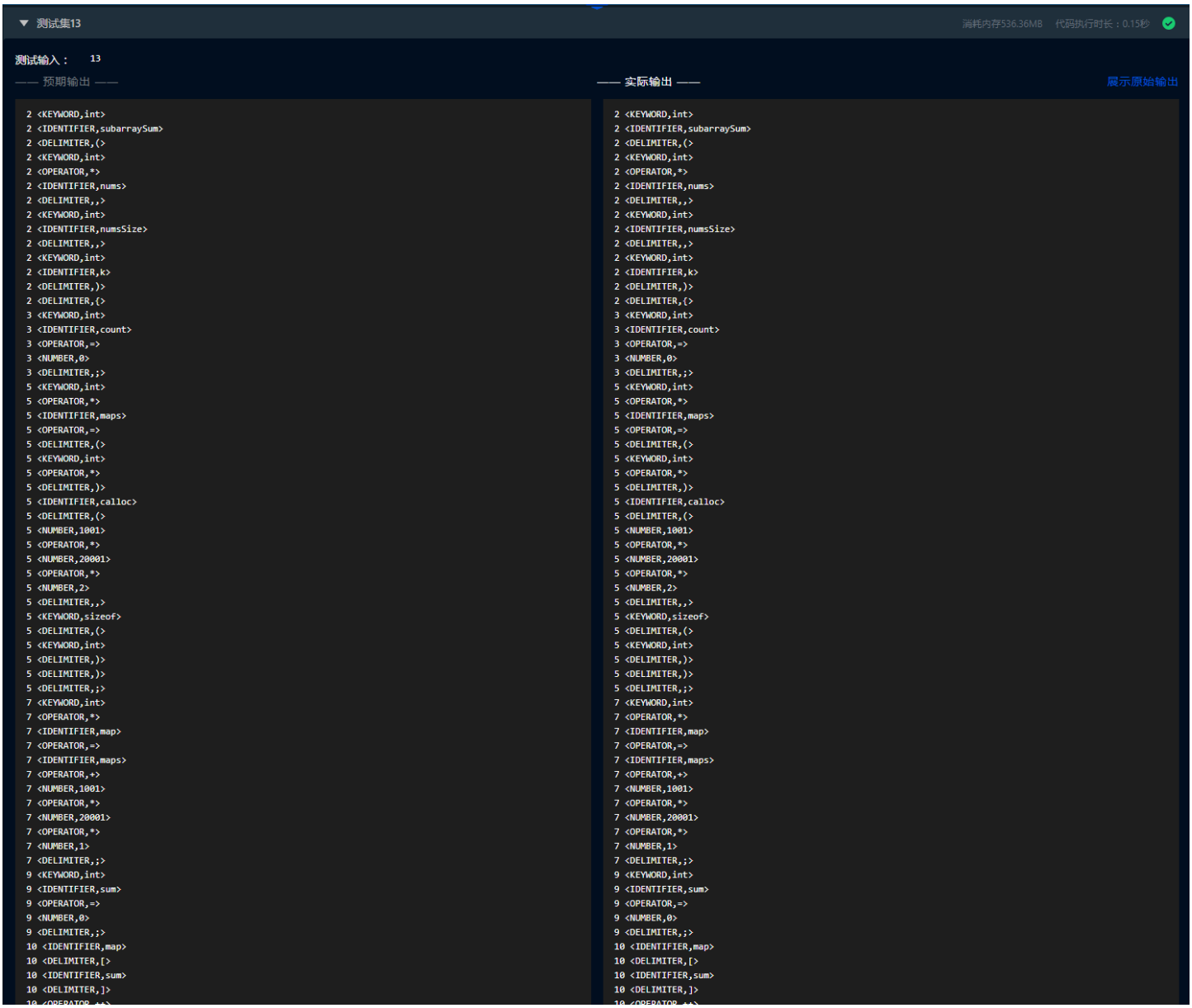
13.含注释的简单代码块识别

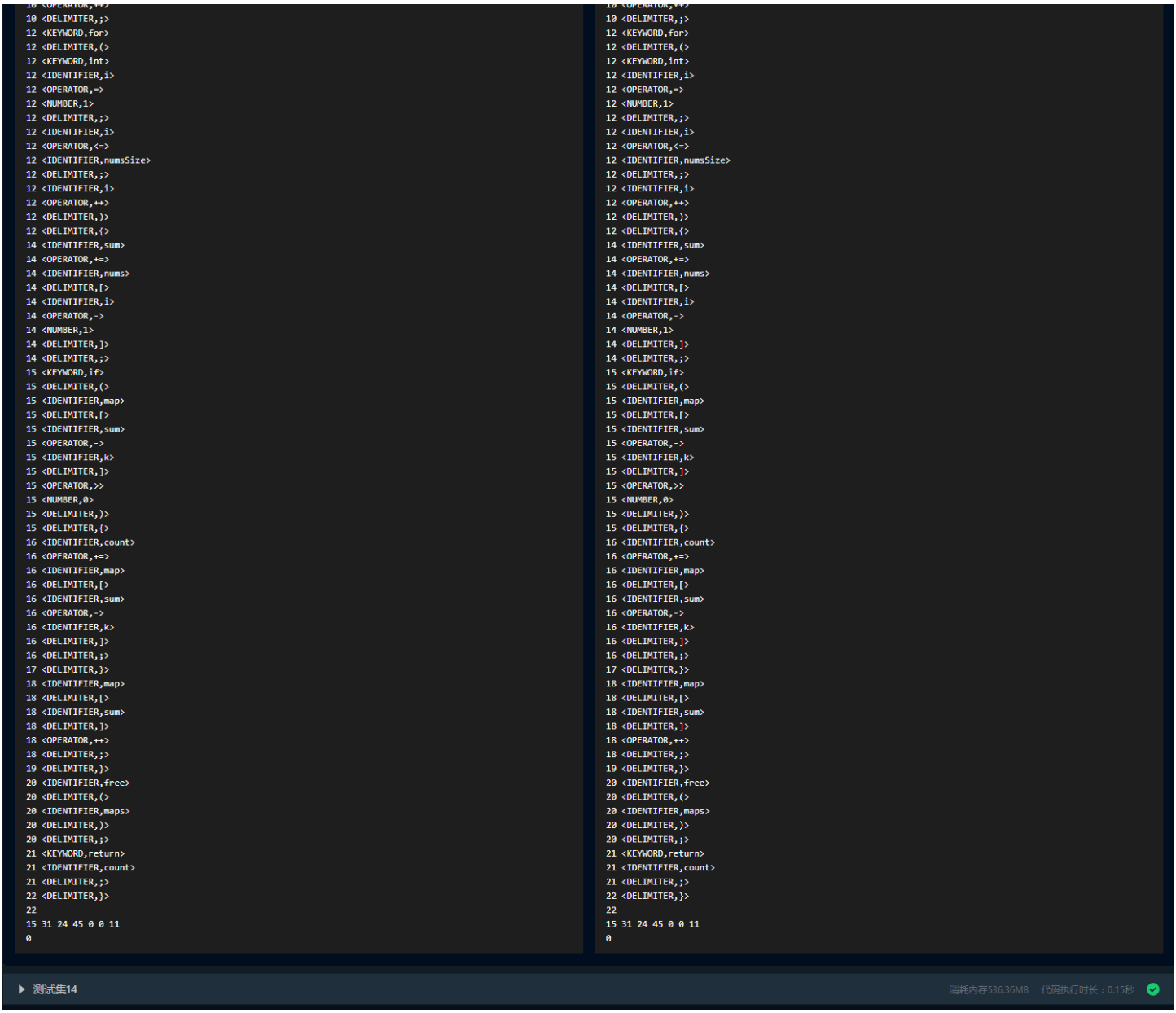
1.样例输入

```
// 暴力美学：20行C代码
int subarraySum(int *nums, int numsSize, int k) {
    int count = 0;
    // 弄个大数组做个暴力的Hash表，大概4*20M*2=160M。用calloc初始化为全零。
    int *maps = (int *)calloc(1001 * 20001 * 2, sizeof(int));
    // 前缀和可能是负数，把指针放到中间位置
```

```
    int *map = maps + 1001 * 20001 * 1;
    // 补一个最前面的前缀和，用了下标0
    int sum = 0;
    map[sum]++;
    // 下标从1开始，注意=
    for (int i = 1; i <= numsSize; i++) {
        // 注意-1
        sum += nums[i - 1];
        if (map[sum - k] > 0) {
            count += map[sum - k];
        }
        map[sum]++;
    }
    free(maps);
    return count;
}
```

2.运行结果与截图





3.分析说明

含注释的简单代码块完全可以识别

14.含注释的乱序代码块识别

1.样例输入

```
/*101-200
2sqrt
*/
/**/prime(int m){int
i,k,h=0,leap=1;/**/printf("\n");k=sqrt(m+1);for(i=2;i<=k;i++)if(m%i==0)
{leap=0;break;}if(leap)
```

```
    {printf("%-4d",m);h++;if(h%10==0)printf("\n");}leap=1;printf("\nThe total is
    %d",h);}
```

2.运行结果与截图

测试集14　　消耗内存536.36MB　代码执行时长：0.15秒

测试输入：　14

—— 预期输出 ——

```
4 <IDENTIFIER,prime>
4 <DELIMITER,(>
4 <KEYWORD,int>
4 <IDENTIFIER,m>
4 <DELIMITER,)>
4 <DELIMITER,{>
4 <KEYWORD,int>
4 <IDENTIFIER,i>
4 <DELIMITER,,>
4 <IDENTIFIER,k>
4 <DELIMITER,,>
4 <IDENTIFIER,h>
4 <OPERATOR,=>
4 <NUMBER,0>
4 <DELIMITER,,>
4 <IDENTIFIER,leap>
4 <OPERATOR,=>
4 <NUMBER,1>
4 <DELIMITER,;>
4 <IDENTIFIER,printf>
4 <DELIMITER,(>
4 <STRING,"\n">
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <IDENTIFIER,k>
4 <OPERATOR,=>
4 <IDENTIFIER,sqrt>
4 <DELIMITER,(>
4 <IDENTIFIER,m>
4 <OPERATOR,+>
4 <NUMBER,1>
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <KEYWORD,for>
4 <DELIMITER,(>
4 <IDENTIFIER,i>
4 <OPERATOR,=>
4 <NUMBER,2>
4 <DELIMITER,;>
4 <IDENTIFIER,i>
4 <OPERATOR,<=>
4 <IDENTIFIER,k>
4 <DELIMITER,;>
4 <IDENTIFIER,i>
4 <OPERATOR,++>
4 <DELIMITER,)>
4 <KEYWORD,if>
4 <DELIMITER,(>
4 <IDENTIFIER,m>
4 <OPERATOR,%>
4 <IDENTIFIER,i>
4 <OPERATOR,==>
4 <NUMBER,0>
4 <DELIMITER,)>
4 <DELIMITER,{>
4 <IDENTIFIER,leap>
4 <OPERATOR,=>
4 <NUMBER,0>
4 <DELIMITER,;>
4 <KEYWORD,break>
4 <DELIMITER,;>
4 <DELIMITER,}>
4 <KEYWORD,if>
4 <DELIMITER,(>
4 <IDENTIFIER,leap>
4 <DELIMITER,)>
4 <DELIMITER,{>
4 <IDENTIFIER,printf>
4 <DELIMITER,(>
4 <STRING,"%-4d">
4 <DELIMITER,,>
4 <IDENTIFIER,m>
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <IDENTIFIER,h>
4 <OPERATOR,++>
4 <DELIMITER,;>
4 <KEYWORD,if>
4 <DELIMITER,(>
4 <IDENTIFIER,h>
4 <OPERATOR,%>
4 <NUMBER,10>
4 <OPERATOR,==>
4 <NUMBER,0>
4 <DELIMITER,)>
4 <IDENTIFIER,printf>
4 <DELIMITER,(>
4 <STRING,"\n">
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <DELIMITER,}>
4 <IDENTIFIER,leap>
4 <OPERATOR,=>
4 <NUMBER,1>
4 <DELIMITER,;>
4 <IDENTIFIER,printf>
4 <DELIMITER,(>
4 <STRING,"\nThe total is %d">
4 <DELIMITER,,>
4 <IDENTIFIER,h>
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <DELIMITER,}>
4
7 26 14 43 0 4 9
0
```

—— 实际输出 ——　　展示原始输出

```
4 <IDENTIFIER,prime>
4 <DELIMITER,(>
4 <KEYWORD,int>
4 <IDENTIFIER,m>
4 <DELIMITER,)>
4 <DELIMITER,{>
4 <KEYWORD,int>
4 <IDENTIFIER,i>
4 <DELIMITER,,>
4 <IDENTIFIER,k>
4 <DELIMITER,,>
4 <IDENTIFIER,h>
4 <OPERATOR,=>
4 <NUMBER,0>
4 <DELIMITER,,>
4 <IDENTIFIER,leap>
4 <OPERATOR,=>
4 <NUMBER,1>
4 <DELIMITER,;>
4 <IDENTIFIER,printf>
4 <DELIMITER,(>
4 <STRING,"\n">
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <IDENTIFIER,k>
4 <OPERATOR,=>
4 <IDENTIFIER,sqrt>
4 <DELIMITER,(>
4 <IDENTIFIER,m>
4 <OPERATOR,+>
4 <NUMBER,1>
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <KEYWORD,for>
4 <DELIMITER,(>
4 <IDENTIFIER,i>
4 <OPERATOR,=>
4 <NUMBER,2>
4 <DELIMITER,;>
4 <IDENTIFIER,i>
4 <OPERATOR,<=>
4 <IDENTIFIER,k>
4 <DELIMITER,;>
4 <IDENTIFIER,i>
4 <OPERATOR,++>
4 <DELIMITER,)>
4 <KEYWORD,if>
4 <DELIMITER,(>
4 <IDENTIFIER,m>
4 <OPERATOR,%>
4 <IDENTIFIER,i>
4 <OPERATOR,==>
4 <NUMBER,0>
4 <DELIMITER,)>
4 <DELIMITER,{>
4 <IDENTIFIER,leap>
4 <OPERATOR,=>
4 <NUMBER,0>
4 <DELIMITER,;>
4 <KEYWORD,break>
4 <DELIMITER,;>
4 <DELIMITER,}>
4 <KEYWORD,if>
4 <DELIMITER,(>
4 <IDENTIFIER,leap>
4 <DELIMITER,)>
4 <DELIMITER,{>
4 <IDENTIFIER,printf>
4 <DELIMITER,(>
4 <STRING,"%-4d">
4 <DELIMITER,,>
4 <IDENTIFIER,m>
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <IDENTIFIER,h>
4 <OPERATOR,++>
4 <DELIMITER,;>
4 <KEYWORD,if>
4 <DELIMITER,(>
4 <IDENTIFIER,h>
4 <OPERATOR,%>
4 <NUMBER,10>
4 <OPERATOR,==>
4 <NUMBER,0>
4 <DELIMITER,)>
4 <IDENTIFIER,printf>
4 <DELIMITER,(>
4 <STRING,"\n">
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <DELIMITER,}>
4 <IDENTIFIER,leap>
4 <OPERATOR,=>
4 <NUMBER,1>
4 <DELIMITER,;>
4 <IDENTIFIER,printf>
4 <DELIMITER,(>
4 <STRING,"\nThe total is %d">
4 <DELIMITER,,>
4 <IDENTIFIER,h>
4 <DELIMITER,)>
4 <DELIMITER,;>
4 <DELIMITER,}>
4
7 26 14 43 0 4 9
0
```

3.分析说明

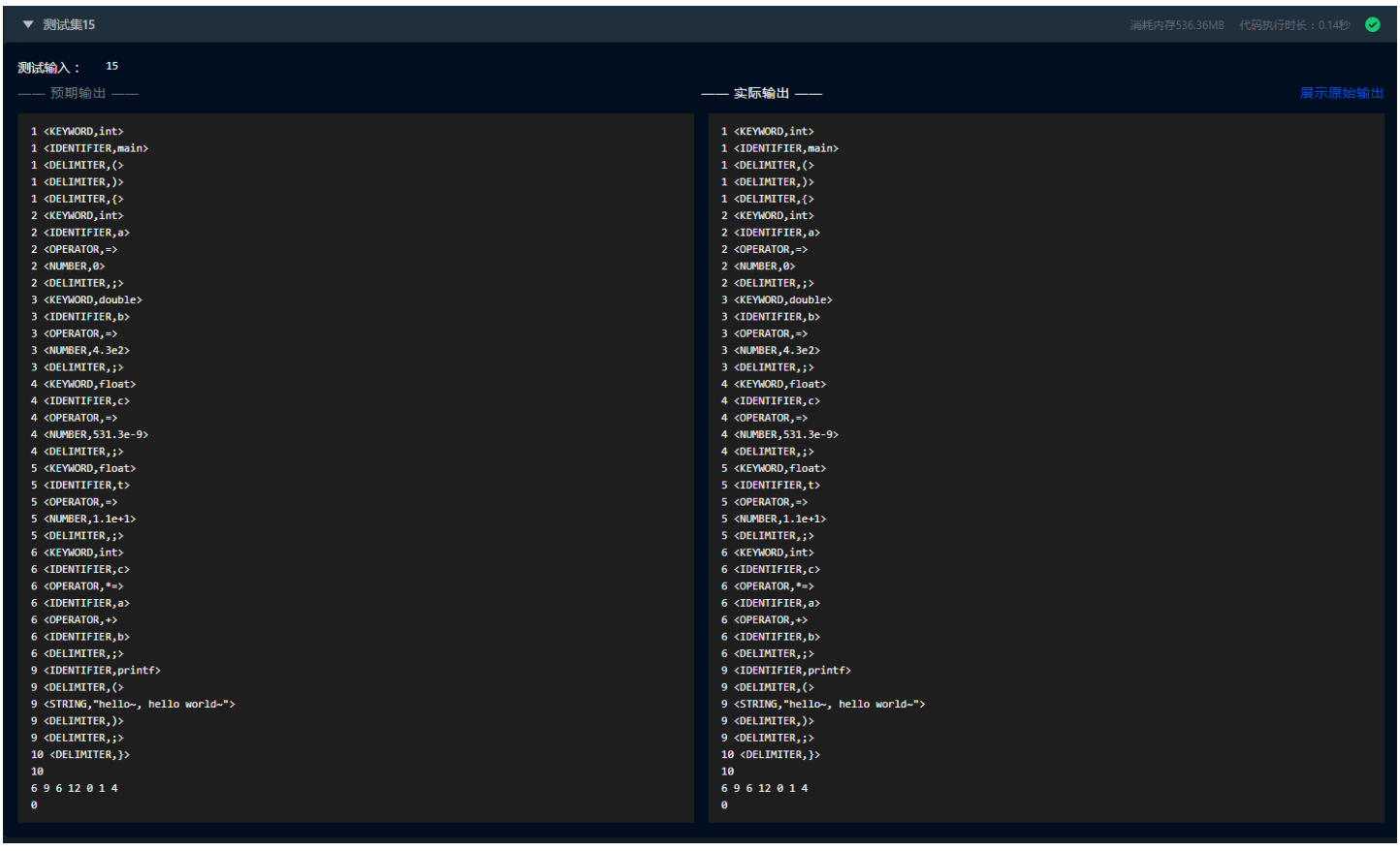
含注释的乱序代码块完全可以识别

15.含大量数值常量的简单代码块识别

1.样例输入

```
int main() {
 int a= 0; // a = 0
 double b = 4.3e2;
 float c = 531.3e-9;
 float t = 1.1e+1;
 int c*= a + b; /*
 adgegasg
 */
 printf("hello~, hello world~");
}
```

2.运行结果与截图



3.分析说明

含大量数值常量的简单代码块完全可以识别

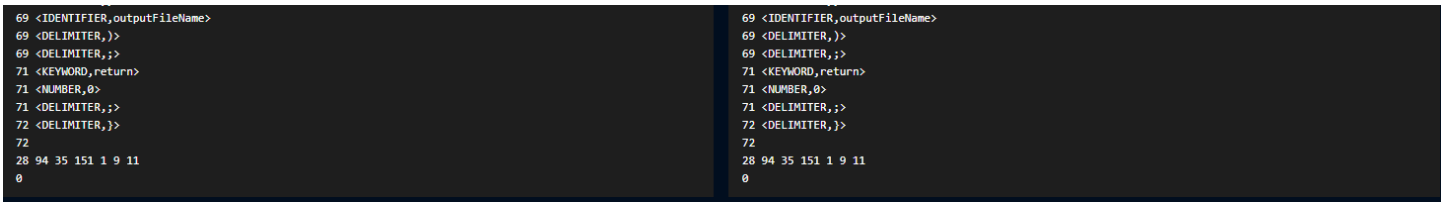
16.复杂代码块识别

1.样例输入

```
typedef struct {
    char word[MAX_WORD_LENGTH];
    int count;
} WordCount;

int main() {
    char inputFileName[] = "input.txt";
    char outputFileName[] = "output.txt";

    // 打开输入文件
    FILE *inputFile = fopen(inputFileName, "r");
    if (inputFile == NULL) {
        printf("无法打开输入文件 %s\n", inputFileName);
        return 1;
    }

    // 打开输出文件
    FILE *outputFile = fopen(outputFileName, "w");
    if (outputFile == NULL) {
        printf("无法创建输出文件 %s\n", outputFileName);
        fclose(inputFile);
        return 1;
    }

    // 初始化单词计数数组
    WordCount wordCounts[MAX_WORD_LENGTH];
    int numWords = 0;

    // 读取输入文件并统计单词
    char word[MAX_WORD_LENGTH];
    while (fscanf(inputFile, "%s", word) != EOF) {
        // 去除单词中的标点符号和转换为小写
        int len = strlen(word);
        for (int i = 0; i < len; i++) {
            if (ispunct(word[i])) {
                word[i] = '\0'; // 去除标点符号
                break;
            }
```

```
            word[i] = tolower(word[i]); // 转换为小写
        }

        // 查找单词是否已存在
        int found = 0;
        for (int i = 0; i < numWords; i++) {
            if (strcmp(wordCounts[i].word, word) == 0) {
                wordCounts[i].count++;
                found = 1;
                break;
            }
        }

        // 如果单词未找到，则添加到数组中
        if (!found) {
            strcpy(wordCounts[numWords].word, word);
            wordCounts[numWords].count = 1;
            numWords++;
        }
    }

    // 写入单词统计结果到输出文件
    for (int i = 0; i < numWords; i++) {
        fprintf(outputFile, "%s: %d\n", wordCounts[i].word,
wordCounts[i].count);
    }

    // 关闭文件
    fclose(inputFile);
    fclose(outputFile);

    printf("单词统计已完成，结果已写入 %s\n", outputFileName);

    return 0;
}
```

2.运行结果与截图



3.分析说明

复杂代码块完全可以识别

17.trie树代码块识别

1.样例输入

```
struct trie_node *new_node(int level)
{
        struct trie_node *p = (struct trie_node *)malloc(sizeof(struct trie_node));
        memset(p, 0, sizeof(struct trie_node));
        p->level = level + 1;
        return p;
}
// 增加节点
void trie_add(struct trie_node *node, unsigned char *pattern)
{
        unsigned int cnt = pattern[0];
        if (node->type[cnt] == TRIR_CHILD_TYPE_NULL)
        {
                if (cnt)
                        set_child_str(node, cnt, pattern);
                else
                        set_child_str(node, cnt, NULL);
                return;
        }
        else if (node->type[cnt] == TRIR_CHILD_TYPE_NODE)
        {
                trie_add((struct trie_node *)(node->child[cnt]), pattern + 1);
                return;
        }
        else if (node->type[cnt] == TRIR_CHILD_TYPE_LEAF) //分裂问题
        {
                if (node->child[cnt] == NULL)
                return;
                unsigned char *leaf = (unsigned char *)(node->child[cnt]);
                struct trie_node *child_node = new_node(node->level);
```

```
31                  add_child_node(node, cnt, child_node);
32                  trie_add(child_node, leaf + 1);
33                  trie_add(child_node, pattern + 1);
34                  return;
35          }
36  }
37  BOOL trie_find(struct trie_node *node, unsigned char *p)
38  {
39          int index = p[0];
40          if (node->type[index] == TRIR_CHILD_TYPE_NULL)
41                  return FALSE;
42          else if (node->type[index] == TRIR_CHILD_TYPE_LEAF)
43          {
44                  if (node->child[index] == 0 && p[0] == 0)
45                          return TRUE;
46                  else if (node->child[index] != 0 && p[0] != 0) //都不为 0，则判断
    是否相同
47                          return strcmp((char *)(p), (char *)(node-
    >child[index])) ==0;
48                  else
49                          return FALSE;
50          }
51          return trie_find((node->child[index]), p + 1);
52  }
53  int main()
54  {
55          FILE *f = fopen("pattern-gbk", "rt");
56          if (f == NULL)
57          {
58                  printf("can not open file 'pattern'");
59                  return 1;
60          }
61          struct trie_node *root = new_node(0);
62          char buffer[256];
63          while (fgets(buffer, sizeof(buffer), f) != NULL)
64          {
65                  char *p = trim_str(buffer);
66                  if (*p == 0)
67                  continue;
68                  trie_add(root, (unsigned char *)strdup(p));
69          }
70          fclose(f);
71          FILE *fData = fopen("input-gbk", "rt");
72          if (fData == NULL)
73          {
74                  printf("can not open file 'input'");
75                  return 1;
```

```
        }
        FILE *fwrite = fopen("yipipei.txt", "w");
        if (fwrite == NULL)
        {
                printf("can not open file 'pattern'");
                return 1;
        }
        int index = 0;
        int count = 0;
        while (fgets(buffer, sizeof(buffer), fData) != NULL)
        {
                index++;
                char *p = trim_str(buffer);
                if (*p == 0)
                continue;
                if (trie_find(root, (unsigned char *)p))
                {
                        count++;
                        printf("%d: %s yes", index, p);
                        fputs(p,fwrite);
                        fprintf(fwrite," ");
                }
                else
                        printf("%d: %s no", index, p);
        }
        printf("read %d lines, found %d", index, count);
        fclose(fData);
        fclose(fwrite);
        return 0;
}
```

2.运行结果与截图

3.分析说明

trie树代码块完全可以识别

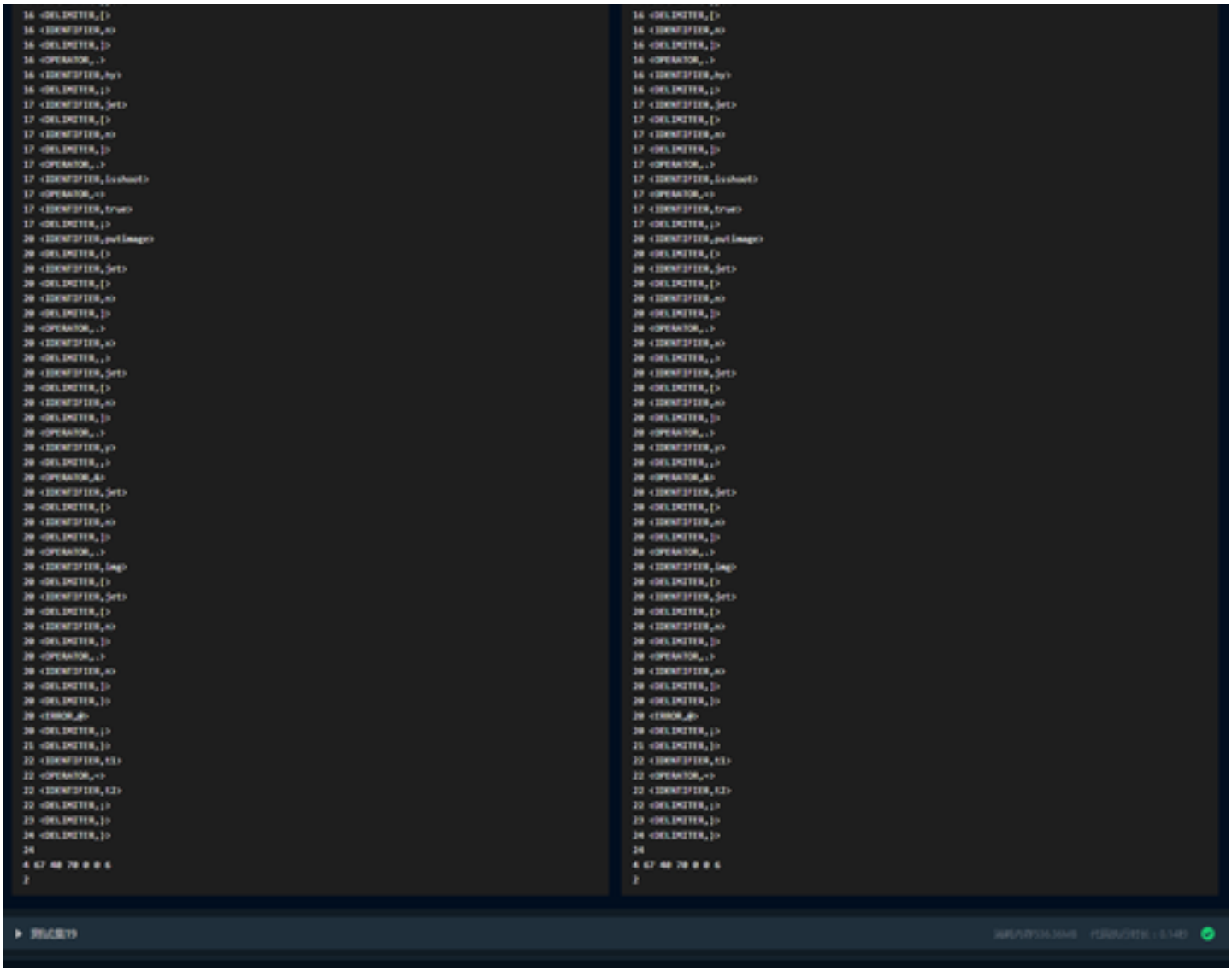
18.含简单错误的代码块识别

1.样例输入

```
@
void choice(DWORD& t1)
{
	DWORD t2 = timeGetTime();

	if (t2 - t1 > 100)	//100ms产生一个烟花弹
	{
		int n = rand() % 20;	//0-19
		if (n < NUM && jet[n].isshoot == false && fire[n].show == false)
		{
			//重置烟花弹
			jet[n].x = rand() % (WND_WIDTH - 20);
			jet[n].y = rand() % 100 + 400;	//450-549
			jet[n].hx = jet[n].x;
			jet[n].hy = rand() % 400;		//0-399
			jet[n].height = jet[n].y - jet[n].hy;
			jet[n].isshoot = true;

			//n
			putimage(jet[n].x, jet[n].y, &jet[n].img[jet[n].n])@;
		}
		t1 = t2;
	}
}
```

2.运行结果与截图

```
4 <DELIMETER,(>
4 <DELIMETER,)>
4 <DELIMETER,;>
6 <KEYWORD,if>
6 <DELIMETER,(>
6 <IDENTIFIER,t2>
6 <OPERATOR,->
6 <IDENTIFIER,t1>
6 <OPERATOR,>>
6 <NUMBER,100>
6 <DELIMETER,)>
7 <DELIMETER,{>
8 <KEYWORD,int>
8 <IDENTIFIER,x>
8 <OPERATOR,=>
8 <IDENTIFIER,rand>
8 <DELIMETER,(>
8 <DELIMETER,)>
8 <OPERATOR,%>
8 <NUMBER,20>
8 <DELIMETER,;>
9 <KEYWORD,if>
9 <DELIMETER,(>
9 <IDENTIFIER,x>
9 <OPERATOR,<>
9 <IDENTIFIER,NUM>
9 <OPERATOR,&&>
9 <IDENTIFIER,jet>
9 <DELIMETER,[>
9 <IDENTIFIER,x>
9 <DELIMETER,]>
9 <OPERATOR,.>
9 <IDENTIFIER,isshoot>
9 <OPERATOR,==>
9 <IDENTIFIER,false>
9 <OPERATOR,&&>
9 <IDENTIFIER,fire>
9 <DELIMETER,[>
9 <IDENTIFIER,x>
9 <DELIMETER,]>
9 <OPERATOR,.>
9 <IDENTIFIER,show>
9 <OPERATOR,==>
9 <IDENTIFIER,false>
9 <DELIMETER,)>
10 <DELIMETER,{>
12 <IDENTIFIER,jet>
12 <DELIMETER,[>
12 <IDENTIFIER,x>
12 <DELIMETER,]>
12 <OPERATOR,.>
12 <IDENTIFIER,x>
12 <OPERATOR,->
12 <IDENTIFIER,rand>
12 <DELIMETER,(>
12 <DELIMETER,)>
12 <OPERATOR,%>
12 <DELIMETER,(>
12 <IDENTIFIER,WND_WIDTH>
12 <OPERATOR,->
12 <NUMBER,20>
12 <DELIMETER,)>
12 <DELIMETER,;>
13 <IDENTIFIER,jet>
13 <DELIMETER,[>
13 <IDENTIFIER,x>
13 <DELIMETER,]>
13 <OPERATOR,.>
13 <IDENTIFIER,y>
13 <OPERATOR,=>
13 <IDENTIFIER,rand>
13 <DELIMETER,(>
13 <DELIMETER,)>
13 <OPERATOR,%>
13 <NUMBER,100>
13 <OPERATOR,+>
13 <NUMBER,400>
13 <DELIMETER,;>
14 <IDENTIFIER,jet>
14 <DELIMETER,[>
14 <IDENTIFIER,x>
14 <DELIMETER,]>
14 <OPERATOR,.>
14 <IDENTIFIER,hx>
14 <OPERATOR,=>
14 <IDENTIFIER,jet>
14 <DELIMETER,[>
14 <IDENTIFIER,x>
14 <DELIMETER,]>
14 <OPERATOR,.>
14 <IDENTIFIER,x>
14 <DELIMETER,;>
15 <IDENTIFIER,jet>
15 <DELIMETER,[>
15 <IDENTIFIER,x>
15 <DELIMETER,]>
15 <OPERATOR,.>
15 <IDENTIFIER,hy>
15 <OPERATOR,=>
15 <IDENTIFIER,rand>
15 <DELIMETER,(>
15 <DELIMETER,)>
15 <OPERATOR,%>
15 <NUMBER,400>
15 <DELIMETER,;>
16 <IDENTIFIER,jet>
16 <DELIMETER,[>
16 <IDENTIFIER,x>
16 <DELIMETER,]>
16 <OPERATOR,.>
16 <IDENTIFIER,height>
16 <OPERATOR,=>
16 <IDENTIFIER,jet>
16 <DELIMETER,[>
16 <IDENTIFIER,x>
16 <DELIMETER,]>
16 <OPERATOR,.>
16 <IDENTIFIER,y>
16 <OPERATOR,->
16 <IDENTIFIER,jet>
```



3.分析说明

含简单错误的代码块完全可以识别

19.含有未闭合字符常量/字符串常量的代码块识别**

此题目曾出错

我们可以看到程序段在正确的代码前几乎都增加了一个 `'` 或者 `"` 符号。

在最开始测试的时候，我的词法分析器总是讲上一个引号与下一行的引号串联在一起当成正确的CHARSON或者STRING类型了。也就是说，此时的词法分析器遇到换行符无法识别。

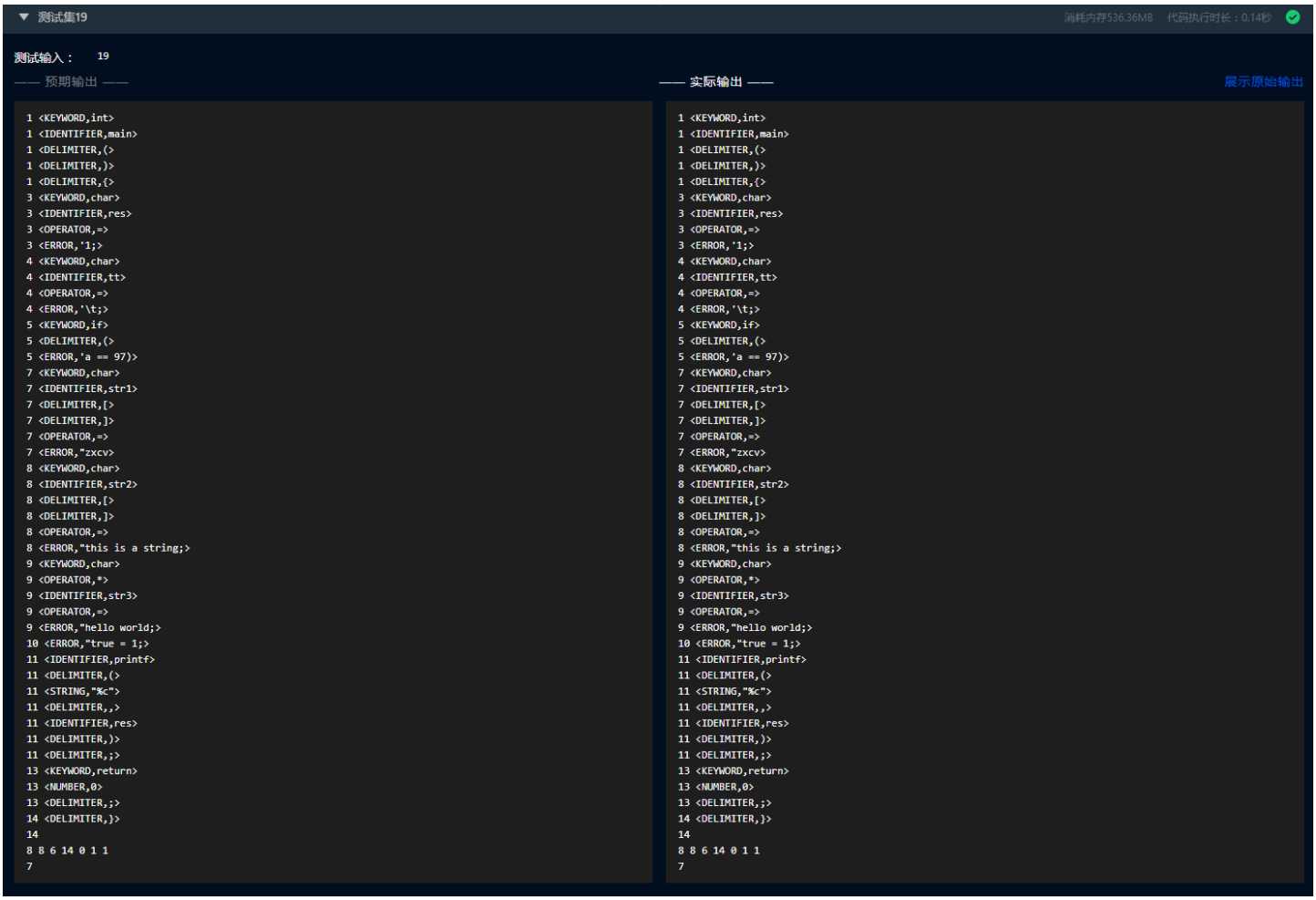
故去修改 `process_string` 与 `process_char_const` 函数，读入的字符如果包括非转义的换行符，且此时引号没有闭合，将优先判定为ERROR类型，之后问题得以解决。

1.样例输入

```
1 int main() {
```

```
2
3           char res = '1;
4           char tt = '\t;
5           if('a == 97)
6
7           char str1[] = "zxcv
8           char str2[] = "this is a string;
9           char *str3  = "hello world;
10          "true = 1;
11          printf("%c", res);
12
13      return 0;
14  }
```

2.运行结果与截图



3.分析说明

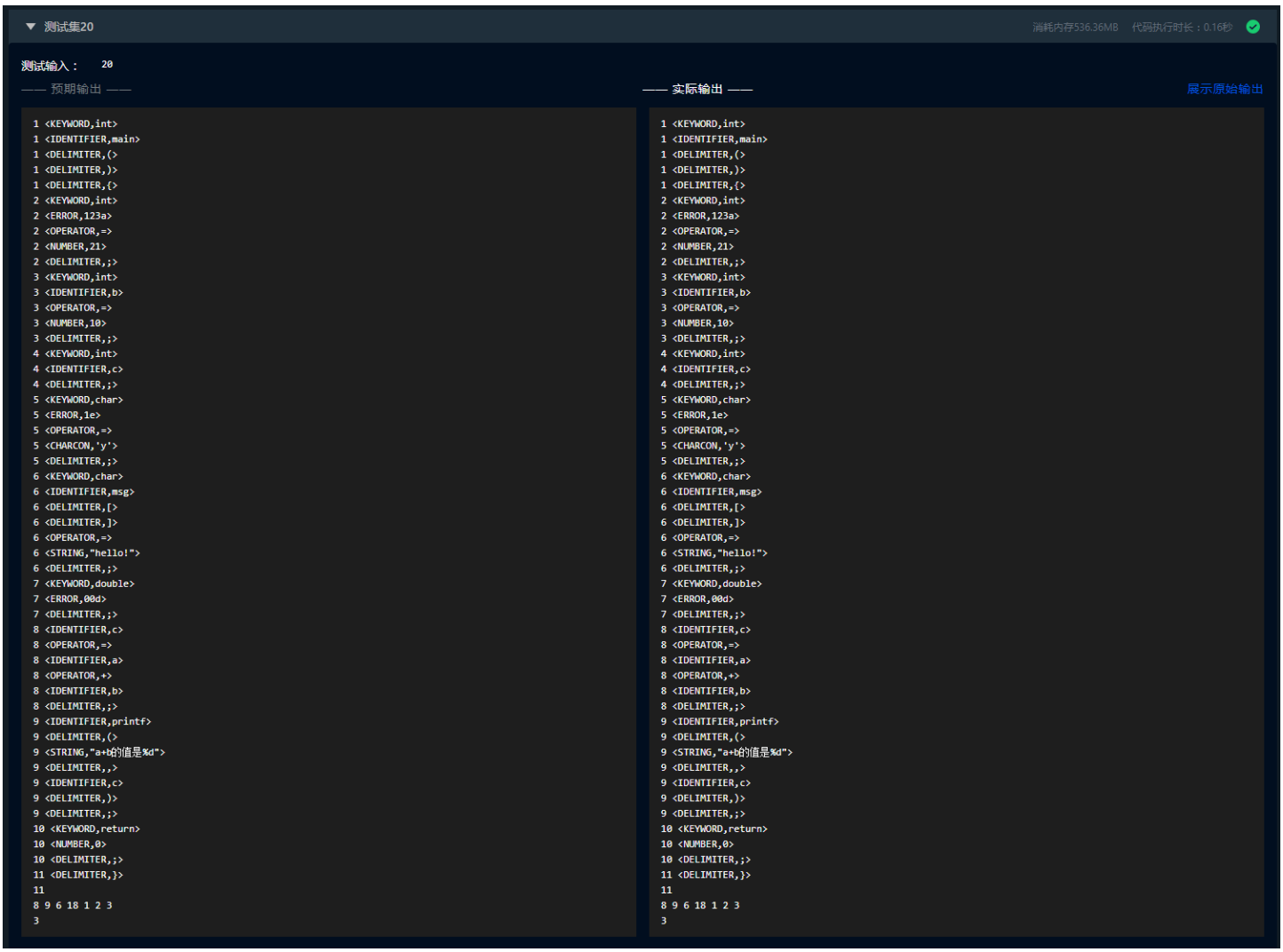
含有未闭合字符常量/字符串常量的代码块完全可以识别

20.含有隐晦错误的代码块识别

1.样例输入

```c
int main() {
        int 123a = 21;
        int b = 10;
        int c;
        char 1e='y';
        char msg[] = "hello!";
    double 00d;
        c = a + b;
        printf("a+b的值是%d", c);
        return 0;
}//#
```

2.运行结果与截图



3.分析说明

含有隐晦错误的代码块完全可以识别

21.含有十六进制的代码块识别**

此题目曾出错

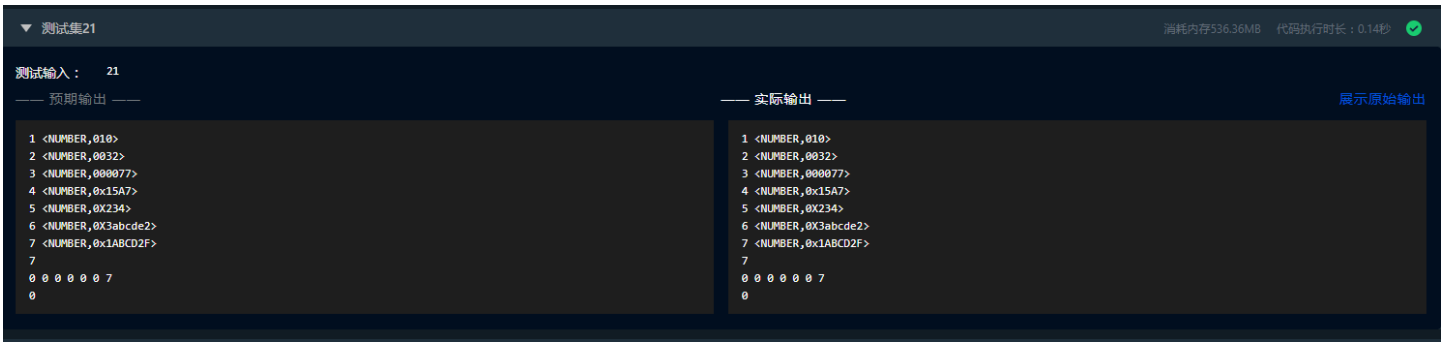
样例输入中含有大量的0前缀和十六进制，在最初的词法分析器中，多个前缀0并没有考虑到识别范围内，十六进制识别中也错误地没有将F算入合法字符中。

之后我针对性地添加了完备的前缀识别函数，十六进制的识别方法也改用了`isxdigit`函数，解决了这一问题。

1.样例输入

```
010
0032
000077
0x15A7
0X234
0X3abcde2
0x1ABCD2F
```

2.运行结果与截图



3.分析说明

含有十六进制的代码块完全可以识别

22.含有后缀十进制整数的代码块识别**

此题目曾出错

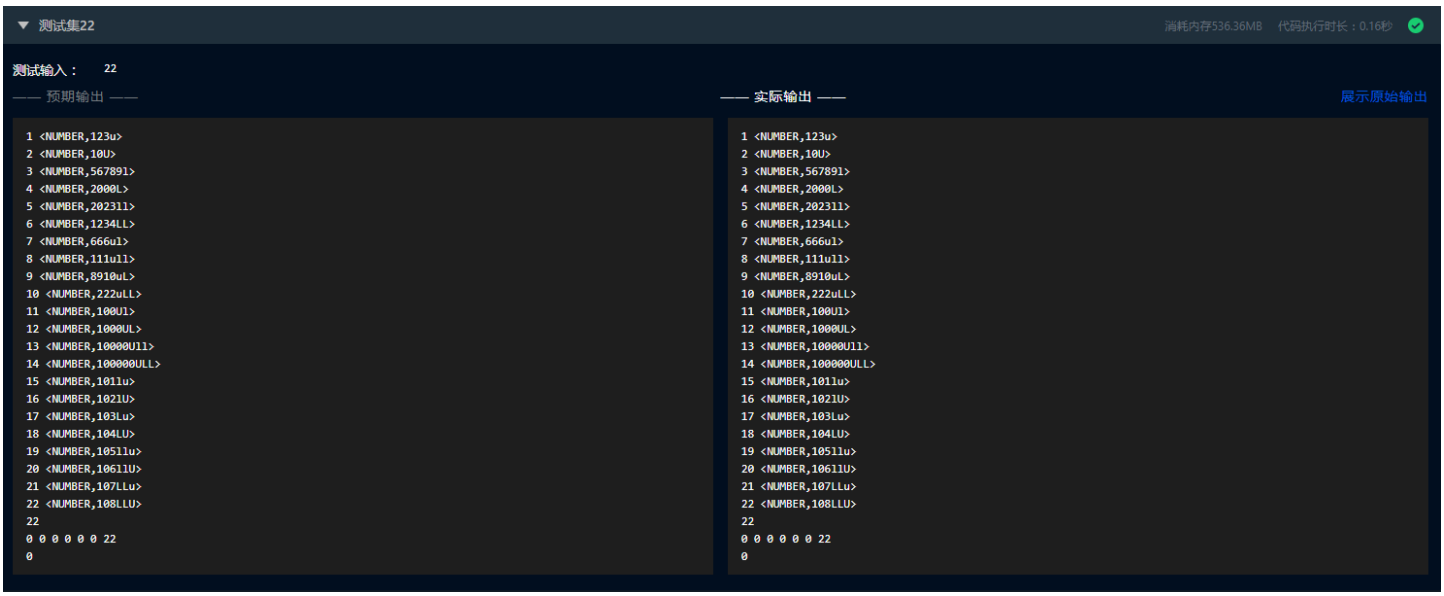
样例输入中包含了十进制整数的所有合法后缀，而在最开始的词法分析器实现中，后缀识别函数没有枚举完全大小写状况，导致返回大量ERROR类型。

之后我选择针对性地写一个合法后缀识别函数，循环地将所有合法后缀与读入的后缀一一对比，相对高效的解决了这一问题。

1.样例输入

```
123u
10U
56789l
2000L
2023ll
1234LL
666ul
111ull
8910uL
222uLL
100Ul
1000UL
10000Ull
100000ULL
101lu
102lU
103Lu
104LU
105llu
106llU
107LLu
108LLU
```

2.运行结果与截图



3.分析说明

含有后缀十进制整数的代码块完全可以识别

23.含有完整的浮点型常量的代码块识别

1.样例输入

```
1.11
.23
0.1f
1e-1
3.14E+2
3.0e1l
4.0E2F
5.0L
.001e2f
```

2.运行结果与截图

3.分析说明

含有完整的浮点型常量的代码块完全可以识别

24.含有前缀的字符串常量的代码块识别**

此题目曾出错

样例输入中，三种格式的字符串常量排在一行，最初的词法分析器遇到 `\"` 与换行符的时候没有识别正确，返回了大量的ERROR类型。

之后修改了`process_string`与`process_char_const`函数，在遇到`\`的时候直接跳过下一字符，遇到换行符的时候结束这一行的识别，重新读入，解决了这一问题。

1.样例输入

```
u8"ABC" u"!!"        U"\""
L""
```

2.运行结果与截图

3.分析说明

含有前缀的字符串常量的代码块完全可以识别

25.含有完整的字符常量的代码块识别**

此题目曾出错

样例输入中，含有`\\`和`\'`转义字符，最初的词法分析器遇到`\\`和`\'`的时候没有识别正确，返回了大量的ERROR类型。

之后修改了`process_string`与`process_char_const`函数，在遇到`\`的时候直接跳过下一字符，解决了这一问题。

1.样例输入

```
'abc'
'\''
L'1'
u'\\'
U'\010'
'\x210'
'\010'
'\11'
```

2.运行结果与截图

3.分析说明

含有完整的字符常量的代码块完全可以识别

26.源代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_TOKEN_LENGTH 1024

// 枚举定义标记类型
typedef enum {
    KEYWORD = 0,
    IDENTIFIER,
    OPERATOR,
    DELIMITER,
    CHARCON,
    STRING,
    NUMBER,
    ERROR,
    TOKEN_TYPE_COUNT
} TokenType;

// 枚举定义字符类型
typedef enum {
    CHAR_LETTER = 0,
    CHAR_DIGIT,
    CHAR_SINGLE_QUOTE,
    CHAR_DOUBLE_QUOTE,
```

```
    CHAR_OTHER
} CharType;

// 关键字列表
const char* keywords[] = {
    "char", "double", "enum", "float", "int", "long",
    "short", "signed", "struct", "union", "unsigned", "void",
    "for", "do", "while", "break", "continue", "if", "else",
    "goto", "switch", "case", "default", "return", "auto",
    "extern", "register", "static", "const", "sizeof", "typedef",
    "volatile"
};
const size_t keyword_count = sizeof(keywords) / sizeof(keywords[0]);

// 词法分析器状态结构体
typedef struct {
    FILE* file;
    int line_number;
    long long token_counts[TOKEN_TYPE_COUNT];
    char* lexeme;
    int lexeme_length;
    int lexeme_capacity;
} LexerState;

// 函数声明
int read_char(LexerState* state);
void unread_char(LexerState* state, int ch);
CharType classify_char(int ch);
void append_char(LexerState* state, int ch);
void reset_lexeme(LexerState* state);
void output_token(LexerState* state, TokenType type, const char* value);
void process_word(LexerState* state, int ch);
void process_number(LexerState* state, int ch);
void process_string(LexerState* state);
void process_char_const(LexerState* state, int ch);
void process_operator_or_delimiter(LexerState* state, int ch);
int process_fraction_part(LexerState* state);
int process_exponent_part(LexerState* state);
void process_string_or_char(LexerState* state, const char* prefix);
int is_valid_integer_suffix(const char* suffix);
int is_valid_float_suffix(const char* suffix);

int main(int argc, char* argv[]) {
    if (argc < 2) {
        fprintf(stderr, "用法: %s <源文件名>\n", argv[0]);
        return EXIT_FAILURE;
    }
```

```
74
75      // 初始化词法分析器状态
76      LexerState state;
77      state.file = fopen(argv[1], "r");
78      if (!state.file) {
79          perror("文件打开失败");
80          return EXIT_FAILURE;
81      }
82      state.line_number = 1;
83      memset(state.token_counts, 0, sizeof(state.token_counts));
84      state.lexeme_capacity = MAX_TOKEN_LENGTH;
85      state.lexeme = (char*)malloc(state.lexeme_capacity);
86      state.lexeme_length = 0;
87
88      int ch;
89      while ((ch = read_char(&state)) != EOF) {
90          if (ch == '\n') {
91              state.line_number++;
92          }
93          if (isspace(ch)) {
94              continue;
95          }
96
97          CharType char_type = classify_char(ch);
98          switch (char_type) {
99          case CHAR_LETTER:
100             process_word(&state, ch);
101             break;
102         case CHAR_DIGIT:
103             process_number(&state, ch);
104             break;
105         case CHAR_SINGLE_QUOTE:
106             process_char_const(&state, ch);
107             break;
108         case CHAR_DOUBLE_QUOTE:
109             process_string(&state);
110             break;
111         case CHAR_OTHER:
112             process_operator_or_delimiter(&state, ch);
113             break;
114         }
115     }
116
117     fclose(state.file);
118
119     // 输出总行数
120     printf("%d\n", state.line_number);
```

```
121
122     // 输出各标记类型的计数
123     for (int i = 0; i < TOKEN_TYPE_COUNT - 1; ++i) {
124         printf("%lld", state.token_counts[i]);
125         if (i == NUMBER) {
126             putchar('\n');
127         }
128         else {
129             putchar(' ');
130         }
131     }
132     printf("%lld", state.token_counts[ERROR]); // 输出结束后不再输出换行符
133
134     free(state.lexeme);
135     return EXIT_SUCCESS;
136 }
137
138 // 从文件读取下一个字符
139 int read_char(LexerState* state) {
140     return fgetc(state->file);
141 }
142
143 // 将字符放回输入流
144 void unread_char(LexerState* state, int ch) {
145     if (ch != EOF) {
146         ungetc(ch, state->file);
147     }
148 }
149
150 // 分类字符类型
151 CharType classify_char(int ch) {
152     if (isalpha(ch) || ch == '_') {
153         return CHAR_LETTER;
154     }
155     else if (isdigit(ch)) {
156         return CHAR_DIGIT;
157     }
158     else if (ch == '\'') {
159         return CHAR_SINGLE_QUOTE;
160     }
161     else if (ch == '"') {
162         return CHAR_DOUBLE_QUOTE;
163     }
164     else {
165         return CHAR_OTHER;
166     }
167 }
```

```
168
169 // 将字符添加到词素缓冲区
170 void append_char(LexerState* state, int ch) {
171     if (state->lexeme_length >= state->lexeme_capacity - 1) {
172         state->lexeme_capacity *= 2;
173         state->lexeme = (char*)realloc(state->lexeme, state->lexeme_capacity);
174     }
175     state->lexeme[state->lexeme_length++] = (char)ch;
176     state->lexeme[state->lexeme_length] = '\0';
177 }
178
179 // 重置词素缓冲区
180 void reset_lexeme(LexerState* state) {
181     state->lexeme_length = 0;
182     state->lexeme[0] = '\0';
183 }
184
185 // 输出标记，按照 v0 的格式
186 void output_token(LexerState* state, TokenType type, const char* value) {
187     const char* type_names[] = {
188         "KEYWORD", "IDENTIFIER", "OPERATOR", "DELIMITER",
189         "CHARCON", "STRING", "NUMBER", "ERROR"
190     };
191     printf("%d <%s,%s>\n", state->line_number, type_names[type], value);
192     state->token_counts[type]++;
193     reset_lexeme(state);
194 }
195
196 // 处理标识符或关键字，处理字符串和字符常量的前缀
197 void process_word(LexerState* state, int ch) {
198     append_char(state, ch);
199     int next_ch;
200
201     // 检查前缀
202     if (ch == 'u') {
203         next_ch = read_char(state);
204         if (next_ch == '8') {
205             append_char(state, next_ch);
206             int peek_ch = read_char(state);
207             if (peek_ch == '"' || peek_ch == '\'') {
208                 append_char(state, peek_ch);
209                 process_string_or_char(state, state->lexeme);
210                 return;
211             }
212             else {
213                 unread_char(state, peek_ch);
214             }
```

```
215         }
216         else if (next_ch == '"' || next_ch == '\'') {
217             append_char(state, next_ch);
218             process_string_or_char(state, state->lexeme);
219             return;
220         }
221         else {
222             unread_char(state, next_ch);
223         }
224     }
225     else if (ch == 'U' || ch == 'L') {
226         next_ch = read_char(state);
227         if (next_ch == '"' || next_ch == '\'') {
228             append_char(state, next_ch);
229             process_string_or_char(state, state->lexeme);
230             return;
231         }
232         else {
233             unread_char(state, next_ch);
234         }
235     }
236 
237     // 继续读取标识符
238     while ((next_ch = read_char(state)), isalnum(next_ch) || next_ch == '_') {
239         append_char(state, next_ch);
240     }
241     unread_char(state, next_ch);
242 
243     // 循环对比检查是否为关键字
244     int is_keyword = 0;
245     for (size_t i = 0; i < keyword_count; ++i) {
246         if (strcmp(state->lexeme, keywords[i]) == 0) {
247             is_keyword = 1;
248             break;
249         }
250     }
251 
252     output_token(state, is_keyword ? KEYWORD : IDENTIFIER, state->lexeme);
253 }
254 
255 // 处理带前缀的字符串或字符常量
256 void process_string_or_char(LexerState* state, const char* prefix) {
257     int ch = state->lexeme[state->lexeme_length - 1]; // 已经读取了引号
258     int is_string = (ch == '"');
259     int is_valid = 1;
260 
261     while ((ch = read_char(state)) != EOF && ch != '\n') {
```

```
262         append_char(state, ch);
263         if (ch == '\\') {
264             ch = read_char(state);
265             if (ch == EOF || ch == '\n') {
266                 is_valid = 0;
267                 break;
268             }
269             append_char(state, ch);
270         }
271         else if ((is_string && ch == '"') || (!is_string && ch == '\'')) {
272             // 结束引号
273             break;
274         }
275     }
276 
277     if ((is_string && ch != '"') || (!is_string && ch != '\'')) {
278         is_valid = 0;
279     }
280 
281     if (is_valid) {
282         output_token(state, is_string ? STRING : CHARCON, state->lexeme);
283     }
284     else {
285         output_token(state, ERROR, state->lexeme);
286         if (ch == '\n') {
287             // 读取到换行符后再增加行号
288             state->line_number++;
289         }
290     }
291 }
292 
293 // 处理数字，包括整数、浮点数、十六进制、八进制等
294 void process_number(LexerState* state, int ch) {
295     append_char(state, ch);
296     int next_ch;
297     int is_valid = 1;
298     int is_float = 0;
299 
300     if (ch == '.') {
301         // 处理以 . 开头的浮点数
302         is_float = 1;
303         is_valid = process_fraction_part(state);
304     }
305     else if (ch == '0') {
306         next_ch = read_char(state);
307         if (next_ch == 'x' || next_ch == 'X') {
308             // 处理十六进制数
```

```
309             append_char(state, next_ch);
310             while ((next_ch = read_char(state)), isxdigit(next_ch)) {
311                 append_char(state, next_ch);
312             }
313             if (isalpha(next_ch)) {
314                 is_valid = 0;
315                 while (isalnum(next_ch)) {
316                     append_char(state, next_ch);
317                     next_ch = read_char(state);
318                 }
319             }
320             unread_char(state, next_ch);
321         }
322         else if (isdigit(next_ch) && next_ch != '8' && next_ch != '9') {
323             // 处理八进制数
324             append_char(state, next_ch);
325             while ((next_ch = read_char(state)), isdigit(next_ch)) {
326                 if (next_ch >= '0' && next_ch <= '7') {
327                     append_char(state, next_ch);
328                 }
329                 else {
330                     is_valid = 0;
331                     append_char(state, next_ch);
332                 }
333             }
334             unread_char(state, next_ch);
335         }
336         else if (next_ch == '.') {
337             // 处理浮点数，如 0.5
338             append_char(state, next_ch);
339             is_float = 1;
340             is_valid = process_fraction_part(state);
341         }
342         else if (next_ch == 'e' || next_ch == 'E') {
343             // 处理科学计数法，如 0e10
344             append_char(state, next_ch);
345             is_float = 1;
346             is_valid = process_exponent_part(state);
347         }
348         else {
349             unread_char(state, next_ch);
350         }
351     }
352     else {
353         // 处理十进制数或浮点数
354         while ((next_ch = read_char(state)), isdigit(next_ch)) {
355             append_char(state, next_ch);
```

```
356         }
357         if (next_ch == '.') {
358             append_char(state, next_ch);
359             is_float = 1;
360             is_valid = process_fraction_part(state);
361         }
362         else if (next_ch == 'e' || next_ch == 'E') {
363             append_char(state, next_ch);
364             is_float = 1;
365             is_valid = process_exponent_part(state);
366         }
367         else {
368             unread_char(state, next_ch);
369         }
370     }
371 
372     // 处理数字后缀
373     if (is_valid) {
374         char suffix[4] = { 0 }; // 最大后缀长度为3
375         int suffix_len = 0;
376         int suffix_ch;
377 
378         while (suffix_len < 3) {
379             suffix_ch = read_char(state);
380             if (isalpha(suffix_ch)) {
381                 append_char(state, suffix_ch);
382                 suffix[suffix_len++] = (char)suffix_ch;
383             }
384             else {
385                 unread_char(state, suffix_ch);
386                 break;
387             }
388         }
389         suffix[suffix_len] = '\0';
390 
391         int valid_suffix = 0;
392         if (is_float) {
393             // 检查浮点数合法后缀
394             valid_suffix = is_valid_float_suffix(suffix);
395         }
396         else {
397             // 检查整数合法后缀
398             valid_suffix = is_valid_integer_suffix(suffix);
399         }
400 
401         if (!valid_suffix) {
402             is_valid = 0;
```

```
403         }
404     }
405 
406     if (is_valid) {
407         output_token(state, NUMBER, state->lexeme);
408     }
409     else {
410         output_token(state, ERROR, state->lexeme);
411     }
412 }
413 
414 // 处理小数部分
415 int process_fraction_part(LexerState* state) {
416     int next_ch;
417     int has_digits = 0;
418 
419     while ((next_ch = read_char(state)), isdigit(next_ch)) {
420         has_digits = 1;
421         append_char(state, next_ch);
422     }
423     if (next_ch == 'e' || next_ch == 'E') {
424         append_char(state, next_ch);
425         if (!process_exponent_part(state)) {
426             return 0;
427         }
428     }
429     else {
430         unread_char(state, next_ch);
431     }
432     return has_digits;
433 }
434 
435 // 处理科学计数法的指数部分
436 int process_exponent_part(LexerState* state) {
437     int next_ch = read_char(state);
438     if (next_ch == '+' || next_ch == '-') {
439         append_char(state, next_ch);
440         next_ch = read_char(state);
441     }
442     if (!isdigit(next_ch)) {
443         unread_char(state, next_ch);
444         return 0;
445     }
446     while (isdigit(next_ch)) {
447         append_char(state, next_ch);
448         next_ch = read_char(state);
449     }
```

```
450     unread_char(state, next_ch);
451     return 1;
452 }
453 
454 // 检查是否为有效的整数后缀
455 int is_valid_integer_suffix(const char* suffix) {
456     const char* valid_suffixes[] = {
457         "u", "U", "l", "L",
458         "ul", "uL", "Ul", "UL",
459         "lu", "lU", "Lu", "LU",
460         "ll", "LL",
461         "ull", "uLL", "Ull", "ULL",
462         "llu", "llU", "LLu", "LLU"
463     };
464     size_t num_suffixes = sizeof(valid_suffixes) / sizeof(valid_suffixes[0]);
465 
466     for (size_t i = 0; i < num_suffixes; ++i) {
467         if (strcmp(suffix, valid_suffixes[i]) == 0) {
468             return 1;
469         }
470     }
471     return suffix[0] == '\0'; // 空后缀也是合法的
472 }
473 
474 // 检查是否为有效的浮点数后缀
475 int is_valid_float_suffix(const char* suffix) {
476     const char* valid_suffixes[] = { "f", "F", "l", "L" };
477     size_t num_suffixes = sizeof(valid_suffixes) / sizeof(valid_suffixes[0]);
478 
479     for (size_t i = 0; i < num_suffixes; ++i) {
480         if (strcmp(suffix, valid_suffixes[i]) == 0) {
481             return 1;
482         }
483     }
484     return suffix[0] == '\0'; // 空后缀也是合法的
485 }
486 
487 // 处理字符串字面量
488 void process_string(LexerState* state) {
489     append_char(state, '"');
490     int ch;
491     int is_valid = 1;
492 
493     while ((ch = read_char(state)) != EOF && ch != '\n') {
494         append_char(state, ch);
495         if (ch == '\\') {
496             ch = read_char(state);
```

```
            if (ch == EOF || ch == '\n') {
                is_valid = 0;
                break;
            }
            append_char(state, ch);
        }
        else if (ch == '"') {
            break;
        }
    }

    if (ch != '"') {
        is_valid = 0;
    }

    if (is_valid) {
        output_token(state, STRING, state->lexeme);
    }
    else {
        output_token(state, ERROR, state->lexeme);
        if (ch == '\n') {
            // 读取到换行符后再增加行号
            state->line_number++;
        }
    }
}

// 处理字符常量，允许单引号内有多个字符
void process_char_const(LexerState* state, int ch) {
    append_char(state, ch);
    int is_valid = 1;

    while ((ch = read_char(state)) != EOF && ch != '\n') {
        append_char(state, ch);
        if (ch == '\\') {
            ch = read_char(state);
            if (ch == EOF || ch == '\n') {
                is_valid = 0;
                break;
            }
            append_char(state, ch);
        }
        else if (ch == '\'') {
            break;
        }
    }

```

```
544     if (ch != '\'') {
545         is_valid = 0;
546     }
547 
548     if (is_valid) {
549         output_token(state, CHARCON, state->lexeme);
550     }
551     else {
552         output_token(state, ERROR, state->lexeme);
553         if (ch == '\n') {
554             // 读取到换行符后再增加行号
555             state->line_number++;
556         }
557     }
558 }
559 
560 // 处理运算符和分隔符
561 void process_operator_or_delimiter(LexerState* state, int ch) {
562     int next_ch = read_char(state);
563 
564     if (ch == '.' && isdigit(next_ch)) {
565         // 处理浮点数，如 .5
566         unread_char(state, next_ch);
567         process_number(state, ch);
568         return;
569     }
570 
571     append_char(state, ch);
572 
573     // 分隔符处理
574     if (strchr(";,:?[](){}", ch)) {
575         output_token(state, DELIMITER, state->lexeme);
576         unread_char(state, next_ch);
577         return;
578     }
579 
580     // 处理多字符运算符
581     if (ch == '+' && (next_ch == '+' || next_ch == '=')) {
582         append_char(state, next_ch);
583         output_token(state, OPERATOR, state->lexeme);
584     }
585     else if (ch == '-' && (next_ch == '-' || next_ch == '=' || next_ch ==
    '>')) {
586         append_char(state, next_ch);
587         output_token(state, OPERATOR, state->lexeme);
588     }
589     else if (ch == '*' && next_ch == '=') {
```

```
590         append_char(state, next_ch);
591         output_token(state, OPERATOR, state->lexeme);
592     }
593     else if (ch == '/' && next_ch == '=') {
594         append_char(state, next_ch);
595         output_token(state, OPERATOR, state->lexeme);
596     }
597     else if ((ch == '%' || ch == '^' || ch == '&' || ch == '|') && next_ch ==
    '=') {
598         append_char(state, next_ch);
599         output_token(state, OPERATOR, state->lexeme);
600     }
601     else if ((ch == '<' || ch == '>') && (next_ch == '=' || next_ch == ch)) {
602         append_char(state, next_ch);
603         if (next_ch == ch) {
604             // 可能是 <<= 或 >>= 等
605             int third_ch = read_char(state);
606             if (third_ch == '=') {
607                 append_char(state, third_ch);
608             }
609             else {
610                 unread_char(state, third_ch);
611             }
612         }
613         output_token(state, OPERATOR, state->lexeme);
614     }
615     else if ((ch == '=' || ch == '!') && next_ch == '=') {
616         append_char(state, next_ch);
617         output_token(state, OPERATOR, state->lexeme);
618     }
619     else if ((ch == '&' && next_ch == '&') || (ch == '|' && next_ch == '|')) {
620         append_char(state, next_ch);
621         output_token(state, OPERATOR, state->lexeme);
622     }
623     else if (ch == '/' && (next_ch == '/' || next_ch == '*')) {
624         // 处理注释
625         if (next_ch == '/') {
626             // 单行注释
627             while ((ch = read_char(state)) != EOF && ch != '\n');
628             if (ch == '\n') {
629                 state->line_number++;
630             }
631             reset_lexeme(state);
632         }
633         else {
634             // 多行注释
635             int prev_ch = 0;
```

```
636             while ((ch = read_char(state)) != EOF) {
637                 if (ch == '\n') {
638                     state->line_number++;
639                 }
640                 if (prev_ch == '*' && ch == '/') {
641                     break;
642                 }
643                 prev_ch = ch;
644             }
645             reset_lexeme(state);
646         }
647     }
648     else if (ispunct(ch) && strchr("~!@#$%^&*-+=|\\:;\"'<>./?", ch)) {
649         // 单字符运算符或分隔符
650         if (ch == '@') {
651             // 处理非法字符
652             output_token(state, ERROR, state->lexeme);
653         }
654         else {
655             output_token(state, OPERATOR, state->lexeme);
656         }
657         unread_char(state, next_ch);
658     }
659     else {
660         // 处理未识别的字符
661         output_token(state, ERROR, state->lexeme);
662         unread_char(state, next_ch);
663     }
664 }
```