**电子科技大学 信息与软件工程 学院**

**标 准 实 验 报 告**

**（实验）课程名称编译技术**

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名： 李艳超 学 号： 2022090902007**

**指导教师： 肖堃**

**实验时间： 2023.10.12 实验地点： 信软 303**

**一、实验室名称： 信软 303**

**二、实验项目名称： 词法分析器**

**三、实验学时： 4学时**

**四、实验内容及步骤：**

实验内容：根据词法分析实验中给定的文法，利用flex设计一词法分析器，该分析器从标准输入读入源代码后，输出单词的类别编号及附加信息

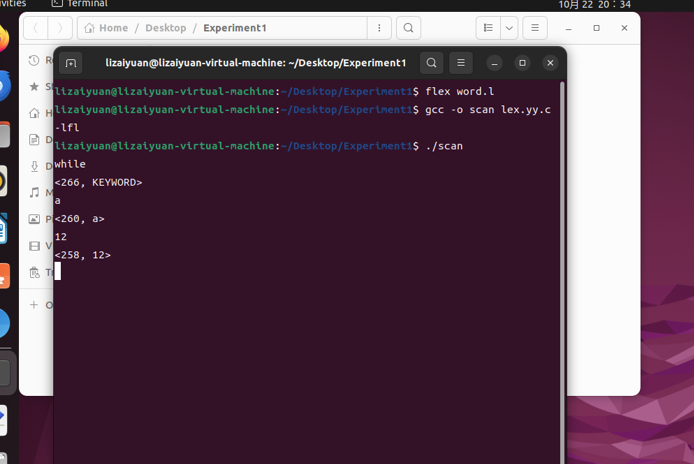
实验步骤：依次构建文件中的**Declarations Definitions Rule User functions**

**需要注意的是其中部分内容是可选的，只有Rule部分必须填充，但我们通常都会填写**Definition Section(定义段) Definition Section(定义段) 和C code section （用户子程序段）这三部分。

在最后的分析总结步骤中，我也会从这三方面来处理

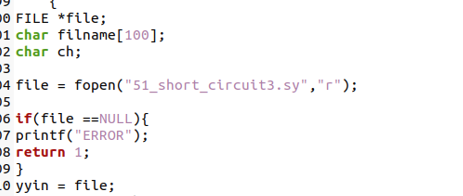
**五、实验运行结果及测试：**

我在编写好基础的word.l文件后，进行了这样的测试



注意这里是需要不断输入，然后会逐个字段地输出相应结果

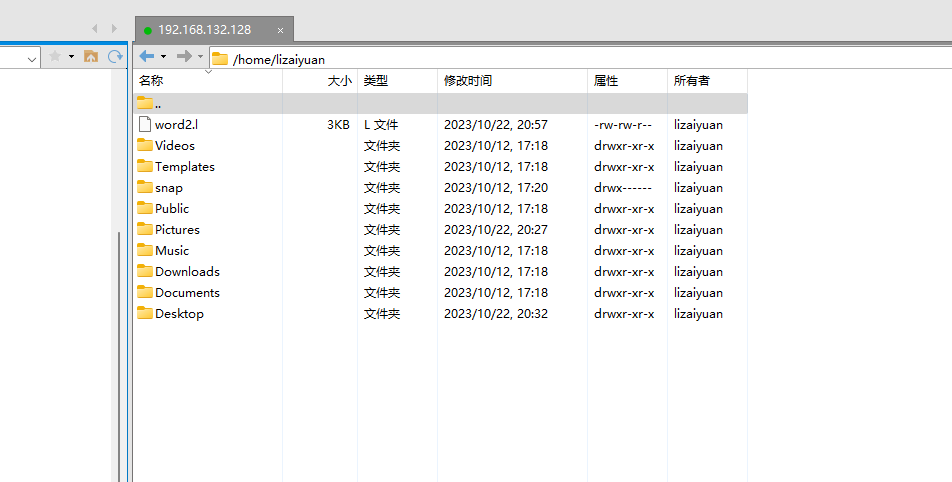
这样的测试是片面的，我挑选了一个字符颇多的文件来测试，利用xftp传入后，修改word.l文件，具体来说是将其与测试文件链接



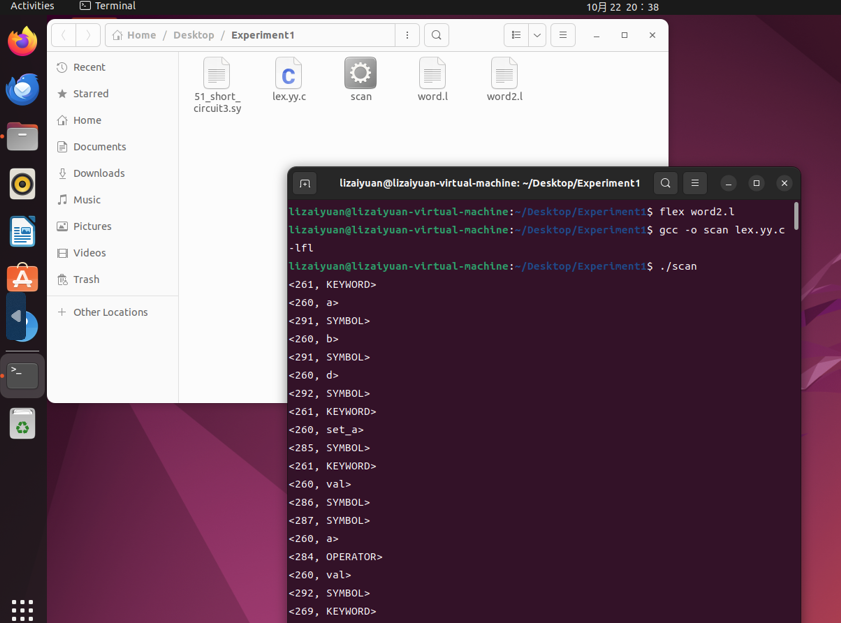
这样处理后，我再重复上述测试，这次我们便可以直接输出测试文件的结果了

具体步骤如下：

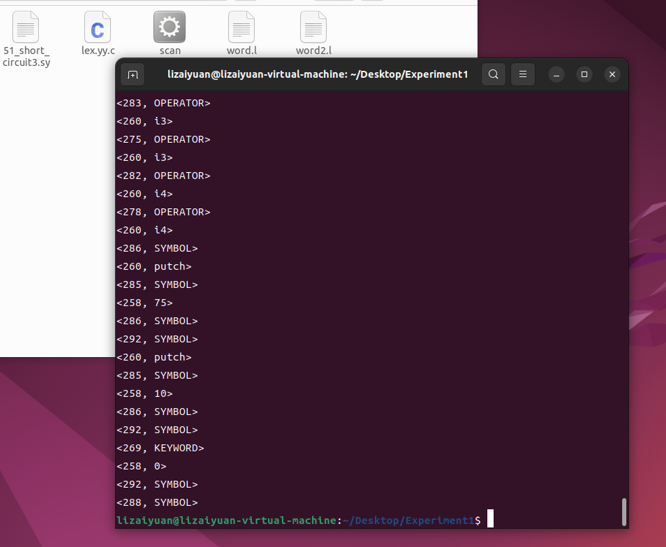
传入测试文件



然后生成



末尾截图



**六、实验结论与总结：**

我在实验课上完成了本次实验的大部分工作，但都比较简单，比如类似于

"//".\* {} // 表示忽略单行注释。

[ \t] { /\* ignore whitespace \*/ }//表示忽略空格和制"int" { return Y\_INT;}

//...

"void" {return Y\_VOID;}

"+" { return Y\_ADD; }

//...

"-" { return Y\_SUB; }表符等空白字符

\n {}// 表示忽略换行符。

欠缺的是一些关键性的细节处理和主函数的进一步优化，具体来讲，前者包括标识符 整数 浮点数的处理，也就是下面这段代码

//表示识别整数，并将其转换为 num\_INT 类型的词法单元，并将其存储在 yylval.int\_value 中

-?[0-9]+ { yylval.int\_value = atoi(yytext); return num\_INT; }

//表示识别浮点数，并将其转换为 num\_FLOAT 类型的词法单元，并将其存储在 yylval.float\_value 中。

-?[0-9]\*\.[0-9]+ { yylval.float\_value = atof(yytext); return num\_FLOAT; }

// 表示识别标识符，并将其转换为 Y\_ID 类型的词法单元，并将其存储在 yylval.id\_name 中。

[a-zA-Z\_][a-zA-Z0-9\_]\* { yylval.id\_name = strdup(yytext); return Y\_ID; }

// 表示识别十六进制数，并将其转换为 num\_INT 类型的词法单元，并将其存储在 yylval.int\_value 中。

0x[0-9a-fA-F]+ { yylval.int\_value = strtol(yytext, NULL, 16); return num\_INT; }

还有注释，空格等需要忽略部分

至于优化部分，我把自己原来的switch-case纯数字分类进行了转换，使用范围来确定输出，虽然还是用到了switch-case，但与if-else结合中省略了篇幅

"//".\* {} // 表示忽略单行注释。

[ \t] { /\* ignore whitespace \*/ }//表示忽略空格和制表符等空白字符

\n {}// 表示忽略换行符。

switch (token)

{

case num\_INT:

printf("%d>\n",ivalue);

break;

case num\_FLOAT:

printf("%f>\n",fvalue);

break;

case Y\_ID:

printf("%s>\n",idname);

break;

case Y\_FLOAT:

printf("KEYWORD>\n");

break;

default:

if(token>=Y\_INT&&token<=Y\_RETURN)

{

printf("KEYWORD>\n");

}

else if(token>=Y\_ADD&&token<=Y\_ASSIGN)

{

printf("OPERATOR>\n");

}

else if(token>=Y\_LPAR&&token<=Y\_SEMICOLON)

{

printf("SYMBOL>\n");

}

break;

}

进一步总结：

.l文件的结构可以概括为

Definition section(定义段)

%%

Rules section（规则段）

%%

C code section （用户子程序段）

首先是Definition Section(定义段)

可以放C语言的各种各种include，define等声明语句，但是要用%{ %}括起来。可以放预定义的正则表达式：minus “-” 还要放token的定义，方法是：代号 正则表达式。然后到了\*\*规则段\*\*就可以通过{代号} 来引用正则表达式，可以这样表达

%{

#include <stdio.h> // include

int wordCount = 0; // 定义全局变量 单词计数器

%}

// 预定义代号表

//定义一个名为 chars 的模式，该模式匹配包括字母、下划线、单引号、句点和双引号在内的一些字符

chars [A-za-z\\_\'\.\"]

numbers ([0-9])+

delim [" "\n\t]

whitespace {delim}+ // 使用前面的代号 来 定义代号

words {chars}+ // 使用前面的代号 来 定义代号

%%

下面是Rules section（规则段）

在这里放置的rules就是每个正则表达式要对应的动作，一般是返回一个token

这里的动作都是用{}扩起来的，用C语言来描述，这些代码可以做任何想要做的事情

//注意这里的%%在上面已经出现过了

{words} { wordCount++; /\* wordCount 加1 \*/ }

{whitespace} { /\* 空白，什么也不做\*/ }

{numbers} { /\* 这里可以加入如果遇到数字的处理逻辑\*/ }

%%

最后是C code section （用户子程序段）

Lex 编程的第三段，也就是最后一段覆盖了 C 的函数声明（有时是主函数）。注意这一段必须包括 yywrap() 函数。 Lex 有一套可供使用的函数和变量。 其中之一就是 yywrap。

/\* main函数 \*/

void main()

{

yylex(); //这一函数开始分析。 它由 Lex 自动生成。

printf(" No of words: %d\n", wordCount);

}

/\*这一函数在文件（或输入）的末尾调用。 如果函数的返回值是1，就停止解析。

因此它可以用来解析多个文件。 代码可以写在第三段，这就能够解析多个文件。

方法是使用 yyin 文件指针指向不同的文件，直到所有的文件都被解析。

最后，yywrap() 可以返回 1 来表示解析的结束。\*/

int yywrap()

{

return 1;

}

补充：在Flex中有一些常用的全局变量和宏

**FILE \*yyin:** **FILE \*yyout：** 这是Lex中本身已定义的输入和输出文件指针。这两个变量指明了lex生成的词法分析器从哪里获得输入和输出到哪里。默认：键盘输入，屏幕输出。

**char \*yytext：**指向当前识别的词法单元（词文）的指针

**int yyleng：**当前词法单元的长度。

**ECHO：**Lex中预定义的宏，可以出现在动作中，相当于fprintf(yyout, “%s”,yytext)，即输出当前匹配的词法单元。

**REJECT：** 指示简析器对当前规则不做处理，而是采用第二匹配规则。 因为解析器在通常情况下，每个被匹配的对象只会对一个动作生效。REJECT指示解析器，会寻找下一个最配的规则来做处理。下面的规则会把输入的"abcd"处理后输出"abcd|abc|ab|a|abcd"。

**BEGIN** 开始一个条件处理块

**int yylex()：**词法分析器驱动程序，用Lex翻译器生成的lex.yy.c内必然含有这个函数。它自动移动文件指针 yyin 和 yyout 。 **在定义模式动作时，可以用 return 语句来结束 yylex分析函数。return 需要返回一个整数。** 由于 yylex() 函数的运行环境都是以全局变量的方式来保存，因此在下一次调用 yylex() 时，yylex()可以从上次扫描的断点处继续扫描。若用户未定义相应的return语句，则yylex()继续分析被扫描的文件，直到碰到文件结束标识符EOF。 在读取到EOF时，yylex() 函数调用 int yywrap() 函数，若yywrap()返回非0值，则yylex() 函数结束返回0；否则，yylex()继续对yyin指向的文件扫描。

**int yywrap()：**词法分析器遇到文件结尾时会调用yywrap()来决定下一步怎么做： 若yywrap()返回0，则继续扫描 若返回1，则返回报告文件结尾的0标记。 由于词法分析器总会调用yywrap，因此辅助函数中最好提供yywrap，如果不提供，则在用C编译器编译lex.yy.c时，需要链接相应的库，库中会给出标准的yywrap函数（标准函数返回1）。

**yymore()**这一函数告诉 Lexer 将下一个标记附加到当前标记后。 `yymore()' 告诉解析器下一次匹配的规则，满足的部分将会添加到当前yytext值得后面而不是替换它。 例如，指定的输入"mega-kludge"经过下面的程序处理后将会输出"mega-mega-kludge"。

**yyless(int n)** 返回当前匹配项除了开始的n个字符内的所有的内容到输入缓存区，解析器处理下一个匹配时，它们将会被重新解析。yyless将会导致yytext与yyleng的调整。（yyleng将会等于=n） 如输入"createtable"被下面的程序处理后，将会输出"createtableatetable". 因为前n=3个字符foo外的字符atetable被重新返回到输入缓存区，再次解析。

**unput(c)** 将字符c放回到输入流中，该字符可以重新被解析。下面的动作将当前的匹配值附上括号后重新进行匹配。注意: 由于每次unput()将指定的字符添加到输入源的开头，所以将字符串添加到输入源开头必须从后道前处理。一个比较重要的潜在问题是使用unput()的时候，如果采用了%pointer指针模式保存yytext,unput会破坏yytext的内容，从最右边的字符开始将会破坏左边的一个字符。如果在unput()后要用到yytext,你首先必须复制一份yytext,或者用%array模式来保存yytext. 最后不能尝试放一个EOF标志输入流的结束。

**input()** 从输入源中读取下一个字符。

以上主要来自于我的学习笔记，在最后提交过程中我会尽量将之一同提交，以免出现纰漏

以上就是我的本次实验报告的所有内容，感谢老师的评阅！

**报告评分：**

**指导教师签字：**