编译原理第一次实验

学号：19335299

姓名：朱德朋

实验题目

词法分析扫描器的设计实现

实验目的

了解高级语言单词的分类；

了解状态图以及如何表示并识别单词规则，掌握状态图到识别程序的编程；

实验内容

使用LEX(FLEX)产生词法分析器的要求

用LEX生成XX(以C为例)语言的词法分析器

词法规则

- 了解所选择编程语言单词符号及其种别值

功能

- 输入一个C语言源程序文件demo.c
- 输出一个文件tokens.txt，该文件包括每一个单词及其种类枚举值，每行一个单词

提交6个文件

- 实验报告
- C语言的LEX源程序： clang.lex
- C语言词法分析程序C源程序：lex.yy.c
- C语言词法分析程序的可执行文件：clang.out/clang.exe
- C语言源程序文件：demo.c（实验输入）
- 词法分析及结果文件： tokens.txt（实验输出）

同时上传源码至Github

知识回顾

词法分析器的作用

词法分析是编译的第一阶段。词法分析器的功能输入源程序，按照构词规则分解成一系列单词符号。单词是语言中具有独立意义的最小单位，包括关键字、标识符、运算符、界符和常量等

1. 关键字 是由程序语言定义的具有固定意义的标识符。例如，Pascal 中的begin，end，if，while都是保留字。这些字通常不用作一般标识符。
2. 标识符 用来表示各种名字，如变量名，数组名，过程名等等。
3. 常数 常数的类型一般有整型、实型、布尔型、文字型等。
4. 运算符 如+、-、*、/等等。
5. 界符 如逗号、分号、括号、等等。

把词法分析安排为一个独立阶段的好处是，它可以使整个编译程序的结构更简洁、清晰和条理化。

正规表达式与有限自动机

对于字母表Σ，我们感兴趣的是它的一些特殊字集， 即所谓正规集。 我们将使用正规式这个概念来表示正规集。 下面是正规式和正规集的递归定义：

1. ϵ和ϕ都是Σ上的正规式，它们所表示的正规集分别为 $\{\epsilon\}$ 和ϕ；
2. 任何$a\in\Sigma$，a是Σ上的一个正规式，它所表示的正规集为 $\{a\}$ ；
3. 假定U和V都是Σ上的正规式，它们所表示的正规集分别为$L(U)$和$L(V)$，那么，(U)、$(U|V)$、$(U\cdot V)$和U^*也都是正规式,它们所表示的正规集分别为$L(U)\cup L(V)$、$L(U)L(V)$和$(L(U))^*$；

仅由有限次使用上述三步骤而得到的表达式才是 Σ上的正规式。 仅由这些正规式所表示的字集才是 Σ上的正规集。

正规式与有限自动机的等价性定理如下：

对任何有限自动机 M， 都存在一个正规式 r， 使得 $L(r)=L(M)$；

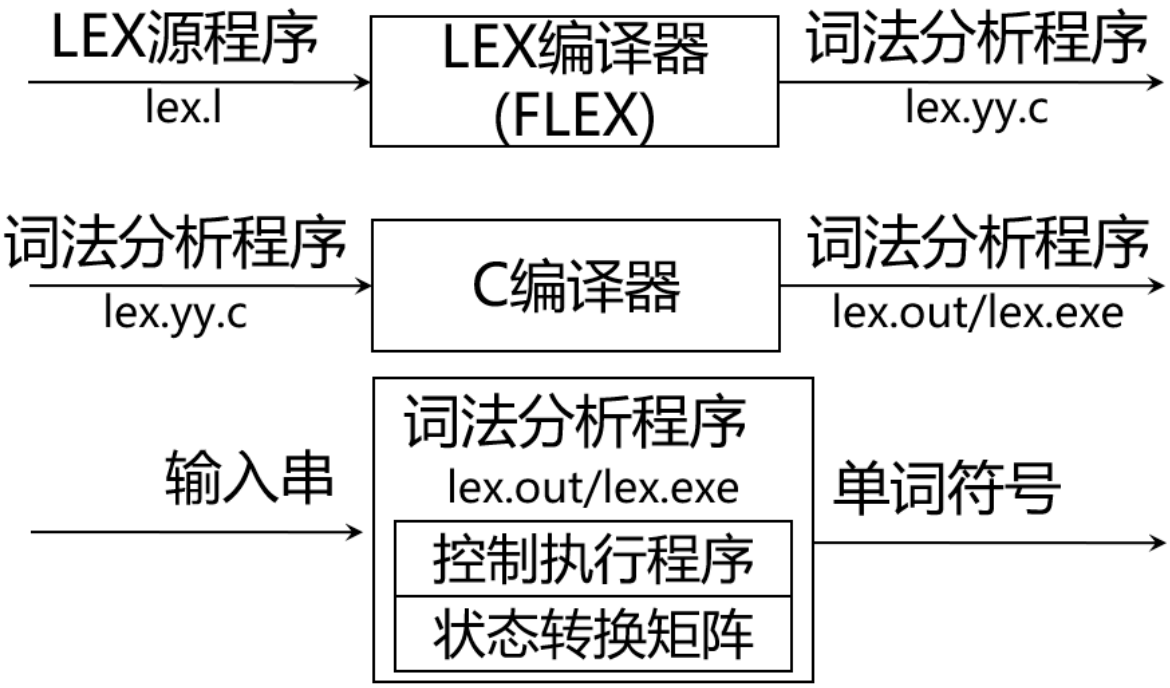
对任何正规式 r， 都存在一个有限自动机 M， 使得 $L(M)=L(r)$；

对于正规式r转换成确定有限自动机(DFA)大致分为下面几步：

1. 正规表达式r转化为非确定有限自动机(NFA)；
2. 非确定有限自动机(NFA)进一步确定化为确定有限自动机(DFA)；
3. 确定有限自动机(DFA)经过化简，得到最小化的确定有限自动机(DFA)；

词法分析器的自动产生

一个描述词法分析器的 LEX 程序由一组正规式以及与每个正规式相应的一个“ 动作”组成。 “动作”本身是一小段程序代码， 它指出了当按正规式识别出一个单词符号时应采取的行动。 将 LEX 程序被编译后所得的结果程序记为L,其作用就如同一个有限自动机一样， 可用来识别和产生单词符号。 结果程序含有一张状态转换表和一个控制程序。



一个LEX源程序主要包括两个部分，一部分使正规定义式，另一部分是识别规则。

如果 Σ 是一个字母表，Σ上的正规定义式是下述形式的定义序列：

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$d_i \rightarrow r_i$$

其中 d_i 表示不同的名字， 每个 r_i 是 $\Sigma \cup \{d_1, d_2, \cdots, d_{i-1}\}$ 上的符号所构成的正规式。r_i中不能含有$d_{i+1}, d_{i+2}, \cdots, d_n$，这样， 对任何 r_i，可以构成一个 Σ 上的正规表达式， 只要反复地将式中出现的名字代之以相应的正规式即可。 注意， 如果允许 r_i 中出现某些 d_j， $j \geq i$,那么这种替代过程将有可能不终止。

LEX 源程序中的识别规则是一串如下形式的LEX语句：

$$P_1 \quad \{A_1\}$$
$$P_2 \quad \{A_2\}$$
$$\cdots$$
$$P_m \quad \{A_m\}$$

其中， 每个 P_i 是一个正规式， 称为词形。 P_i 中除了出现 Σ 中的字符外， 还可以出现正规定义式左部所定义的任何简名d_i。 即，P_i 是$\Sigma \cup \{d_1, d_2, \cdots, d_n\}$ 上的--- 个正规式。 由于每个 d_i 最终都可化为纯粹 Σ 上的正规式， 因此， 每个 P_i 也同样如牝，每个 A_i 是一小段程序代码， 它指出了， 在识别出词形为 P_i 的单词之后， 间法分析器应采取的动作。 这些识别规则完全决定了词法分析器 I.的功能。 分析器I.只能识别具有词形$P_1, \cdots, P_m$的单词符号 .

至于LEX所产生的目标程序 L（词法分析器）的工作原理：L 逐一扫描输人串的每个字符， 寻找一个最长的子串匹配某个 P_i，将该子串截下来放在一个叫做TOKEN的缓冲区中（ 事实上， 这个 TOKEN 也可以只包含一对指示器， 它们分别指出这个子串在原输人缓冲区中的始末位置)。 然后， L 就调用动作子程序 A_i，当 A_i工作完后，L 就把所得的单词符号(由种别编码和属性值两部分构成)交给语法分析程序。 当 L 重新被调用时就从输人串中继上次截出的位置之后识别下一个单词符号。

实验过程

根据上面知识回顾，我们知道：LEX的基本工作原理为：由正规式生成NFA，将NFA变换成DFA，DFA经化简后，模拟生成词法分析器。

其中正规式由开发者使用LEX语言编写，其余部分由LEX翻译器完成.翻译器将LEX源程序翻译成一个名为`lex.yy.c`的C语言源文件，此文件含有两部分内容：一部分是根据正规式所构造的DFA状态转移表，另一部分是用来驱动该表的总控程序`yylex()`。当主程序需要从输入字符流中识别一个记号时，只需要调用一次`yylex()`就可以了。为了使用Lex所生成的词法分析器，我们需要将`lex.yy.c`程序用C编译器进行编译，并将相关支持库函数连入目标代码。
LEX源程序必须按照下面的语言规范来写，

```
    定义段
%%
    词法规则段
%%
    辅助函数段
```

注意，辅助函数并非必要的，并且每两个部分之间需要用 %% 分隔。Lex源程序中可以有注释，注释由/*和*/括起，但是请注意，注释的行首需要有前导空白。

定义段

定义段可以分为两部分：

第一部分需要写在%{和%}之中，里面以C语法写一些定义与声明。例如，一些头文件，函数声明，宏定义，以及一些全局变量和外部变量定义等等。

本次实验中该部分具体代码为

```
%{
#include <stdio.h>
#include <string.h>

const int key_num=32;
const char* keySet[]={
    "auto", "break", "case", "char", "const", "continue",
    "default", "do", "double", "else", "enum", "extern",
    "float", "for", "goto", "if", "int", "long", "register",
    "return", "short", "signed", "sizeof", "static", "struct",
    "switch", "typedef", "union", "unsigned", "void", "volatile",
    "while"
};

int iskey(char* str);

%}
```

这里定义了表示一个保留字的数量的变量`key_nun`，一个记录保留字的集合`keySet`，还定义了一个辅助函数`iskey`，用以判断遍历到的文本是否为保留字。本次实验中，我们只考虑了32种保留字，并未考虑C11标准中新添加的几种保留字，当然后续可以补充。

第二部分是一组正规定义和状态定义。正规定义是为了简化后面词法规则而给部分正规式定义的额名字。每条正规定义都要顶行写。当然这部分也可以省略。

本次实验中该部分具体代码为，

```
letter [A-Za-z]
char '[^']*'
digit [0-9]
preprocess ^#.*
string \"[^\"]*\"
id  ({letter}|_)({letter}|_|{digit})*
operater "!"|"%"|"^"|"&"|"*"|"("|")"|"-"|"+"|"="|"=="|"{"|"}"|"
["|"]"|"\\"|"|"|":"|";"|"<"|">"|","|"."|"/"|"?"|"!="|"
<="|">="|"&&"|"||"|"+="|"-="|"*="|"/="|"|%="|"<<="|">>="|"^="|"&="
int {digit}+
float ([+\-]?){digit}*(\.{digit}+)?((e|E)[+\-]?{digit}+)?
errorID {digit}({letter}|_|{digit})*({letter}|_)({letter}|_|{digit})*
%x BLOCK_COMMENT
```

利用正则表达式可以代表一系列的文本。其中`%x BLOCK_COMMENT`用来表示注释部分。

词法规则段

词法规则段列出的是词法分析器的需要匹配的正规式，以及匹配规则正规式后需要进行的相关动作。

在本次实验中具体代码如下，

```
%%

{preprocess} {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    fprintf(fp,"<%s,->\n",yytext);
    fclose(fp);
}

{string} {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    fprintf(fp,"<$string,%s>\n",yytext);
    fclose(fp);
}

{char} {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    fprintf(fp,"<$char,%s>\n",yytext);
    fclose(fp);
}

{id} {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    if(!iskey(yytext))
        fprintf(fp,"<$ID,%s>\n",yytext);
    else
        fprintf(fp,"<%s,->\n",yytext);
    fclose(fp);
}

```

```
{operater} {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    fprintf(fp,"<$operater,%s>\n",yytext);
    fclose(fp);
}

{int} {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    fprintf(fp,"<$int,%s>\n",yytext);
    fclose(fp);
}

{float} {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    fprintf(fp,"<$float,%s>\n",yytext);
    fclose(fp);
}

{errorID} {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    fprintf(fp,"There is a wrong id: %s.\n",yytext);
    fclose(fp);
}

"//".*\n {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    fprintf(fp,"single line comment.\n");
    fclose(fp);
}

"/*" {
    FILE *fp=NULL;
    fp=fopen("token.txt","a+");
    fprintf(fp,"multiple lines comment.\n");
    fclose(fp);
    BEGIN(BLOCK_COMMENT);
}

<BLOCK_COMMENT>"*/" {
    BEGIN(INITIAL);
}

%%
```

每行都是一条规则，该规则的前一部分是正规式，需要顶行首写，后一部分是匹配该正规式后需要进行的动作，这个动作是用C语法来写的，被包裹在{}之内，被Lex翻译器翻译后会被直接拷贝进`lex.yy.c`。正规式和语义动作之间要有空白隔开。其中用{}扩住的正规式表示正规定义的名字。

辅助函数段

辅助函数的写法为C语言写法，辅助函数一般是在词法规则段中用到的函数。这一部分一般会被直接拷贝到`lex.yy.c`中。

在本次实验中，我所使用的辅助函数如下

```
%%

int iskey(char* str){
    for(int i=0;i<key_num;i++){
        if(strcmp(keySet[i],str)==0){
            return i+1;
        }
    }
    return 0;
}

int main(){
    FILE *fp=NULL;
    fp=fopen("token.txt","w+");
    fprintf(fp,"Hello, this is a Lexer degisned by zhudp3.\n");
    fclose(fp);
    yylex();
    return 0;
}

```

一些变量和函数

`yyin`和`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。

`yytext`：指向当前识别的词法单元（词文）的指针

`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。

`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。

`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）。

实验结果

首先，对于LEX的使用，我们需要安装对应的环境，输入下面的命令配置环境，安装依赖

```
sudo apt install cpp lld flex bison zlib1g-dev \
  clang libclang-dev llvm-dev
```

之后，我们编写一个简单的C语言程序，来进行LEX源程序正确性的验证，

```c
#include <stdio.h>
#define N 10000

int main()
{
    int i;
    float j,k;
    i=10;
    j=1.0;
    k=-1.2e-10;
    char x='y';
    char y='x';
    //abcdef
    char *s="This is a Lexer designed by zhudp3!";
    /*Q2
    */
    return 0;
}
```

进一步的输入下面的命令是编译运行，

```
zhudp3@ubuntu:~/CP/LexAnalysis$ flex clang.lex
zhudp3@ubuntu:~/CP/LexAnalysis$ gcc -o clang.out lex.yy.c -lfl
zhudp3@ubuntu:~/CP/LexAnalysis$ ./clang.out <demo.c
```

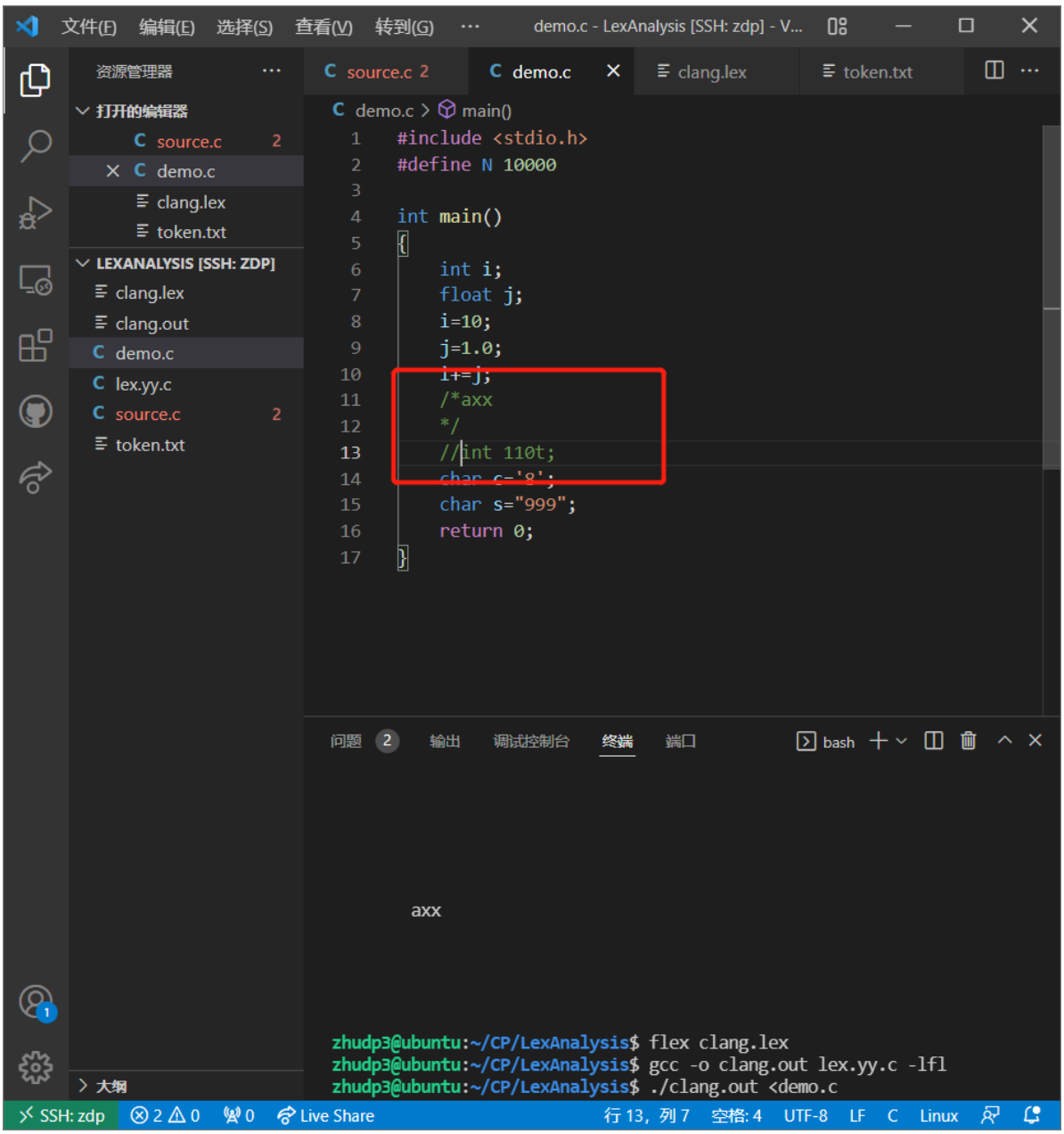
最终我们得到相应的输出，

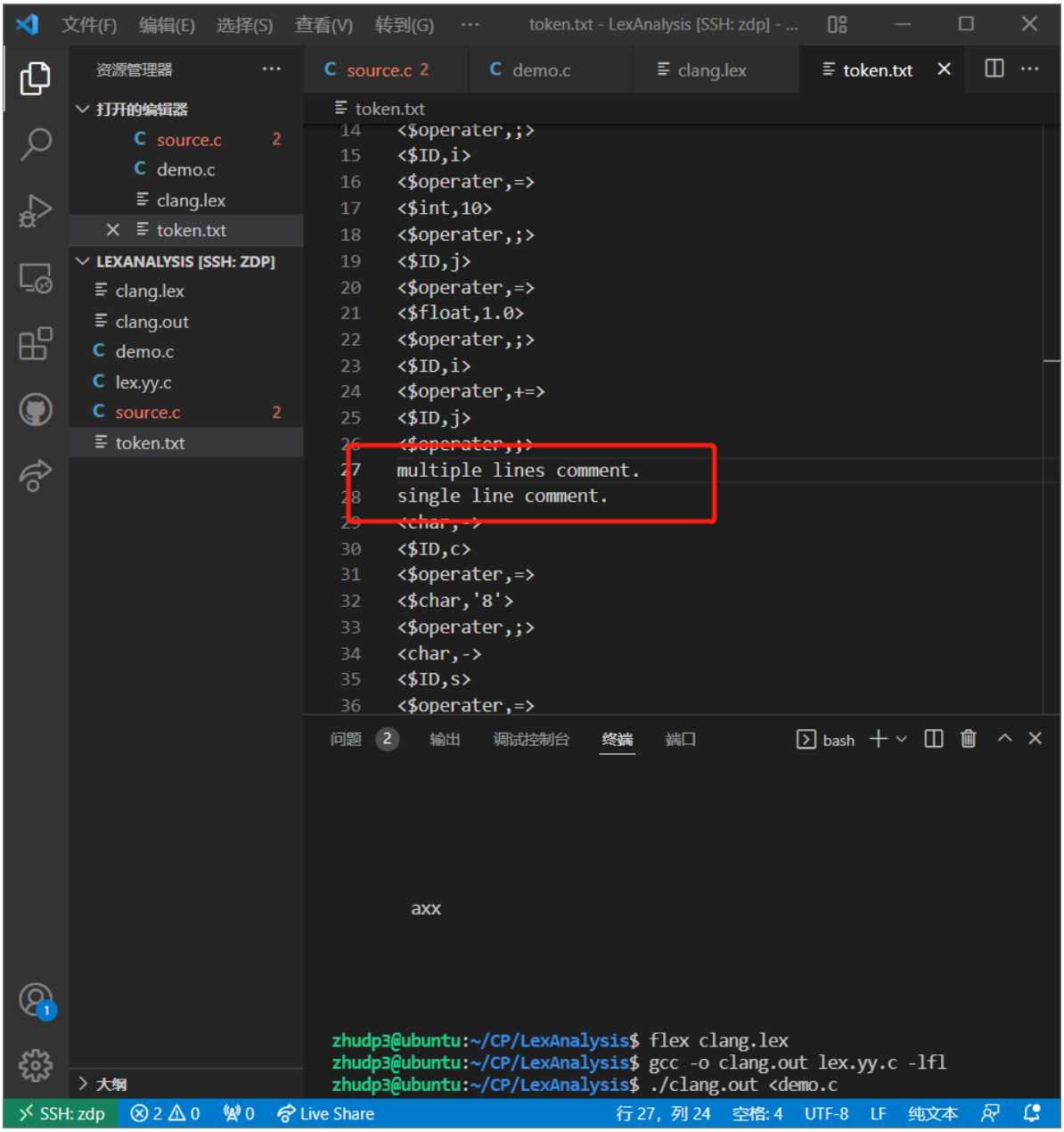
```
Hello, this is a Lexer degisned by zhudp3.
<#include <stdio.h>,->
<#define N 10000,->
<int,->
<$ID,main>
<$operater,(>
<$operater,)>
<$operater,{>
<int,->
<$ID,i>
<$operater,;>
<float,->
<$ID,j>
<$operater,,>
<$ID,k>
<$operater,;>
<$ID,i>
<$operater,=>
<$int,10>
<$operater,;>
<$ID,j>
<$operater,=>
<$float,1.0>
<$operater,;>
```

```
<$ID,k>
<$operater,=>
<$float,-1.2e-10>
<$operater,;>
<char,->
<$ID,x>
<$operater,=>
<$char,'y'>
<$operater,;>
<char,->
<$ID,y>
<$operater,=>
<$char,'x'>
<$operater,;>
single line comment.
<char,->
<$operater,*>
<$ID,s>
<$operater,=>
<$string,"This is a Lexer designed by zhudp3!">
<$operater,;>
multiple lines comment.
<return,->
<$int,0>
<$operater,;>
<$operater,}>
```

可以看到所有TOKEN都被正确输出，并且能够准确识别所有类型。特别的，在输出的第27行可以看到该程序可以识别出科学计数法。整型，浮点型，字符型和字符串都可以准确识别。

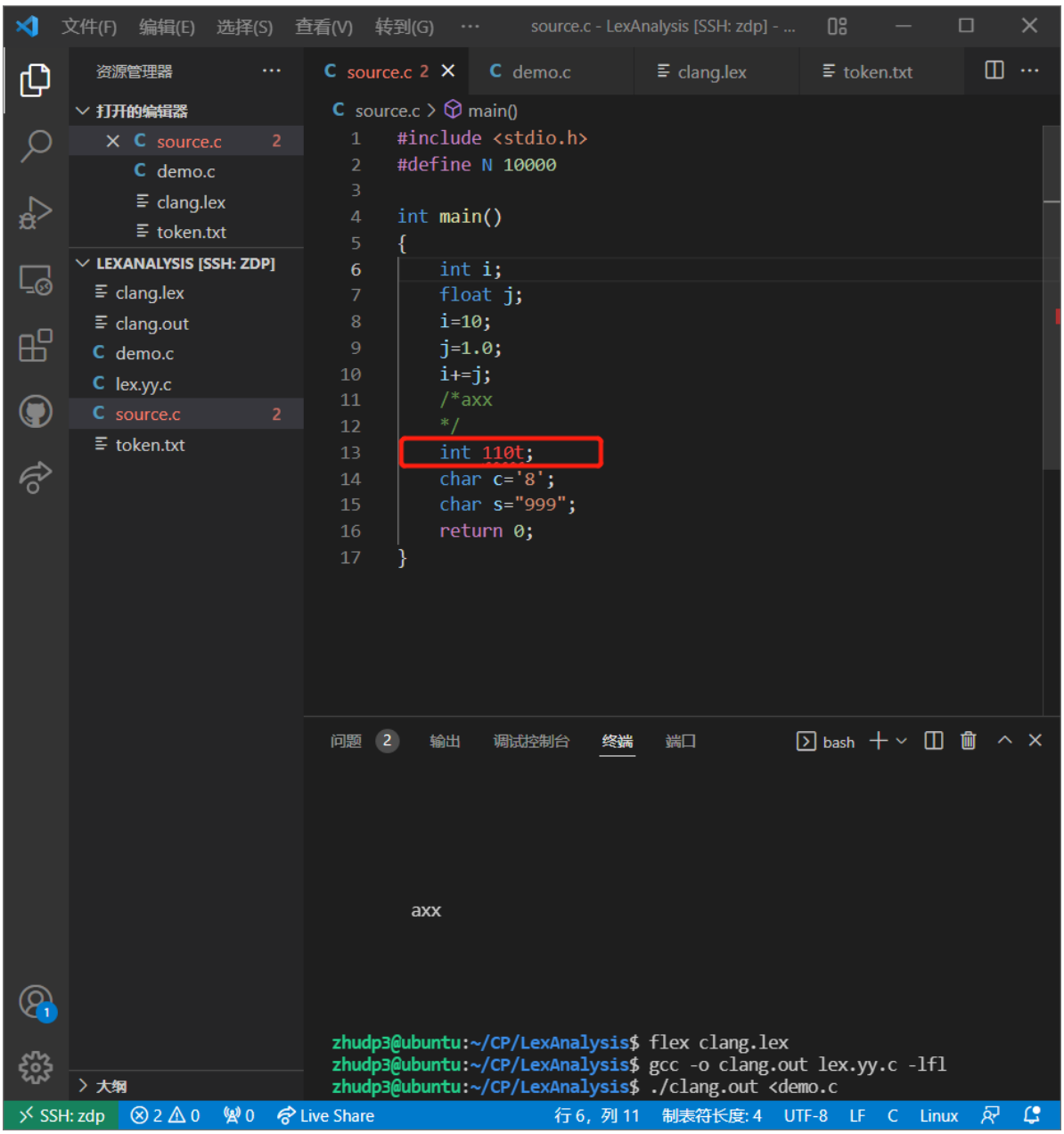
甚至单行注释和多行注释也可以准确识别，如下面两图

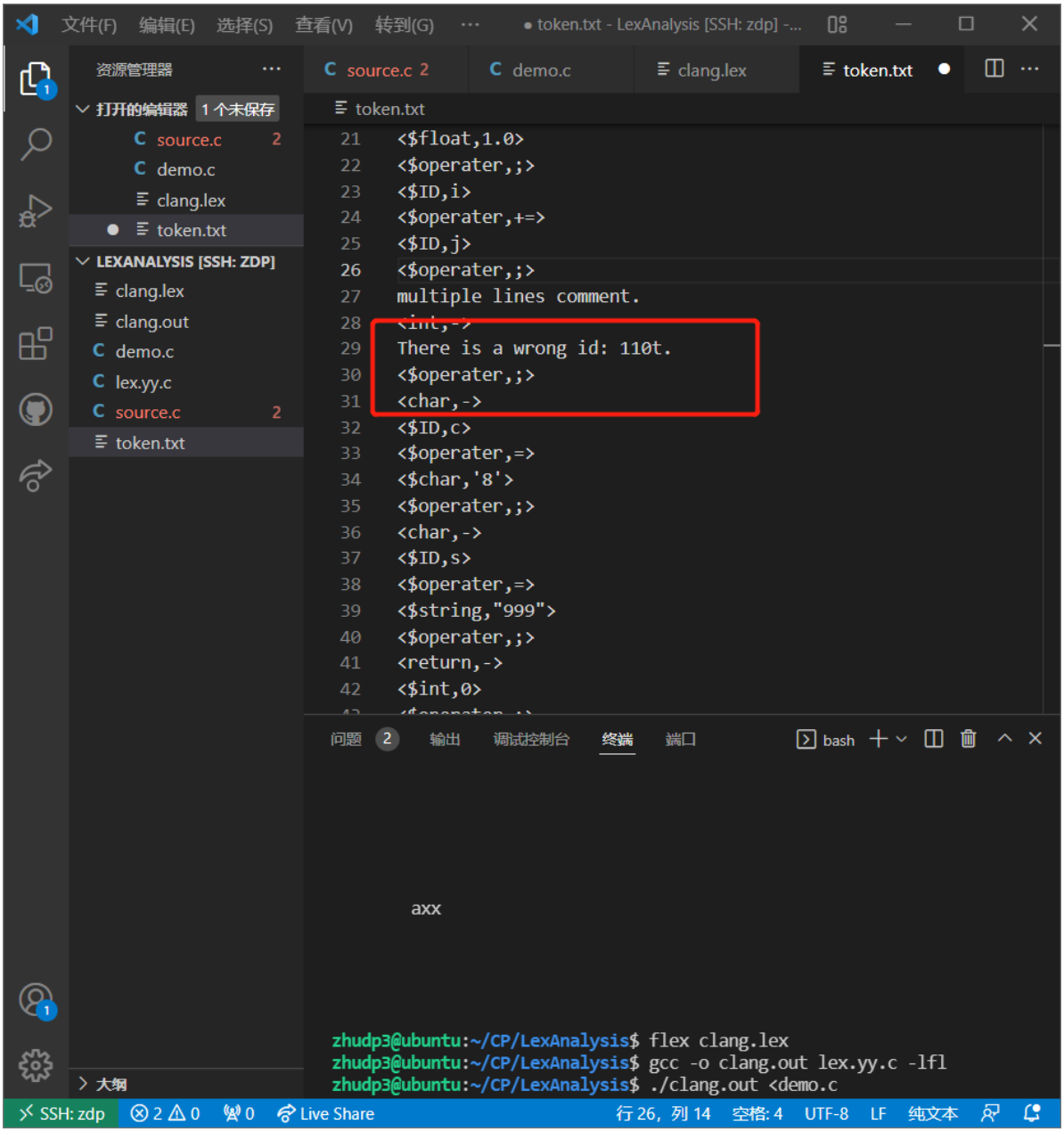

文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) ··· demo.c - LexAnalysis [SSH: zdp] - V...
资源管理器
打开的编辑器
source.c 2
demo.c
clang.lex
token.txt
LEXANALYSIS [SSH: ZDP]
clang.lex
clang.out
demo.c
lex.yy.c
source.c 2
token.txt
大纲
source.c 2
demo.c
clang.lex
token.txt
demo.c > main()
#include <stdio.h>
#define N 10000
int main()
{
int i;
float j;
i=10;
j=1.0;
i+=j;
/*axx
*/
//int 110t;
char c='8';
char s="999";
return 0;
}
问题 2
输出
调试控制台
终端
端口
bash
axx
zhudp3@ubuntu:~/CP/LexAnalysis$ flex clang.lex
zhudp3@ubuntu:~/CP/LexAnalysis$ gcc -o clang.out lex.yy.c -lfl
zhudp3@ubuntu:~/CP/LexAnalysis$ ./clang.out <demo.c
SSH: zdp
2 0
0
Live Share
行 13，列 7
空格: 4
UTF-8
LF
C
Linux



可以看到两种不同的注释都被分析成功。

除此之外，我还添加了一些报错方式，例如当变量名不符合C语言规范时，该词法分析器会对应的报错，如下面两图，


文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) ··· source.c - LexAnalysis [SSH: zdp] - ...
资源管理器
source.c 2
demo.c
clang.lex
token.txt
source.c > main()
#include <stdio.h>
#define N 10000
int main()
{
int i;
float j;
i=10;
j=1.0;
i+=j;
/*axx
*/
int 110t;
char c='8';
char s="999";
return 0;
}
打开的编辑器
LEXANALYSIS [SSH: ZDP]
clang.lex
clang.out
demo.c
lex.yy.c
source.c 2
token.txt
大纲
问题 2
输出
调试控制台
终端
端口
bash
axx
zhudp3@ubuntu:~/CP/LexAnalysis$ flex clang.lex
zhudp3@ubuntu:~/CP/LexAnalysis$ gcc -o clang.out lex.yy.c -lfl
zhudp3@ubuntu:~/CP/LexAnalysis$ ./clang.out <demo.c
SSH: zdp
2 0
0
Live Share
行 6，列 11
制表符长度: 4
UTF-8
LF
C
Linux



综上所述，我们可以基本认定实验成功。

实验总结

本次实验是编译原理第一次实验，也正是帮助我复习理论中正规式和有限自动机，以及其确定化，最小化的内容。在本次实验中，我选择使用LEX完成项目，这使得我学习到了LEX的相关语法。此外，我还惊叹于LEX的强大计算能力，更加提醒我要提高自己的学习能力和编程能力。

但是，我在本次实验中还存在一些不好的地方。例如，在文件写入和写出的时候，我使用的是`#include <stdio.h>`中的文件流，并没有用使用LEX中自带的`yyin`和`yyout`。这可能引发一些文件安全性的错误。