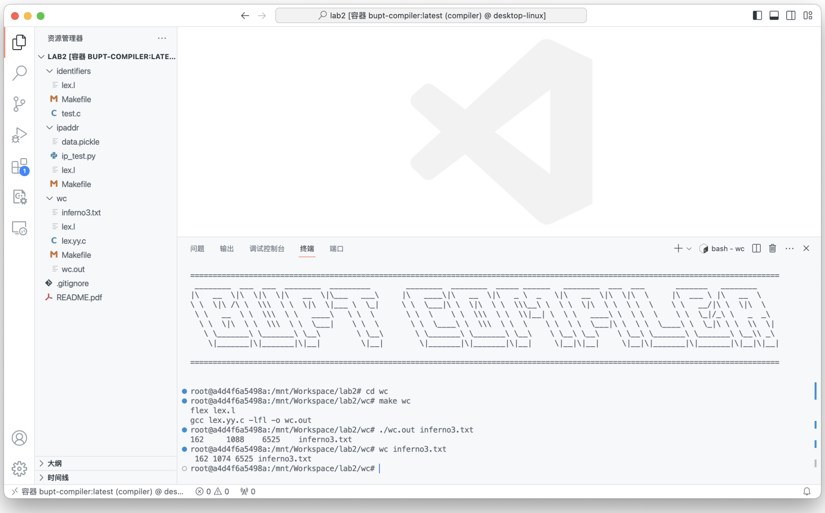
NAME: Jinyi Xia
STUDENT ID: 2021212057
CLASS NUMBER: 2021211802
CONTAINER NUMBER: a4d4f6a5498a4e912a4d1a76a7fa36cf23af64b416688d222a8968a38b29a711

REPORT ON LAB 2

1 Running the `wc` program example

Follow the instructions in the lab manual to build and run the `wc` program example as is shown in fig. 1.



```
root@a4d4f6a5498a:/mnt/Workspace/lab2# cd wc
root@a4d4f6a5498a:/mnt/Workspace/lab2/wc# make wc
flex lex.l
gcc lex.yy.c -lfl -o wc.out
root@a4d4f6a5498a:/mnt/Workspace/lab2/wc# ./wc.out inferno3.txt
162     1088    6525    inferno3.txt
root@a4d4f6a5498a:/mnt/Workspace/lab2/wc# wc inferno3.txt
 162 1074 6525 inferno3.txt
root@a4d4f6a5498a:/mnt/Workspace/lab2/wc# 
```

Figure 1: Running the `wc` program example

Obviously, the example's `wc` differs from the one of Linux systems in the result of word numbers. Linux's `wc` treats a sequence of characters separated by spaces, tabs, or newlines as a word; while the example's `wc` treats a sequence of letters separated by other characters as a word.

2 Flex exercise: identifiers

Two changes are made to the `lex.l` file:

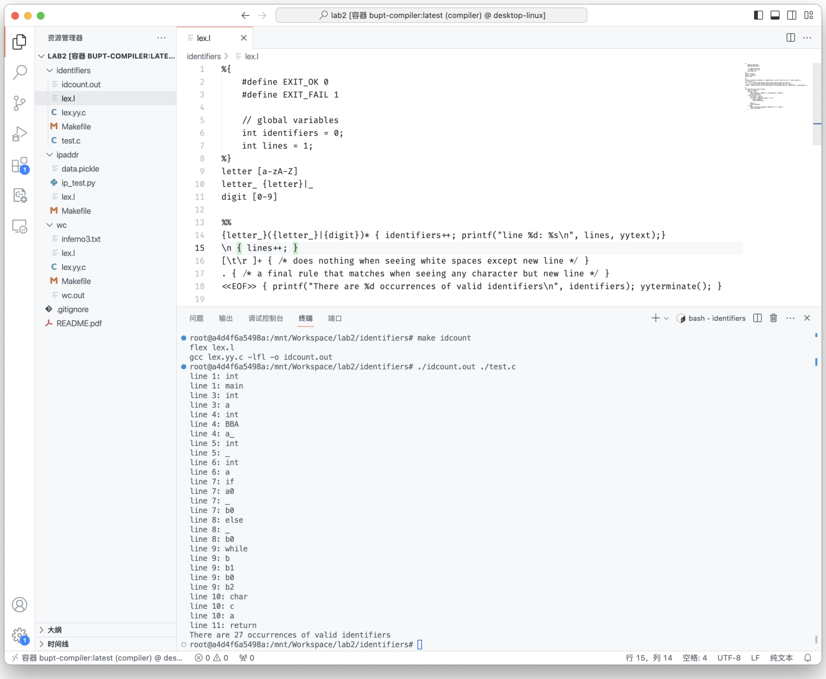
- Line 7:

  Change the initial value of `lines` to `1`.

- Line 15:

  Change this line to `\n { lines++; }`.

The result is shown in fig. 2.

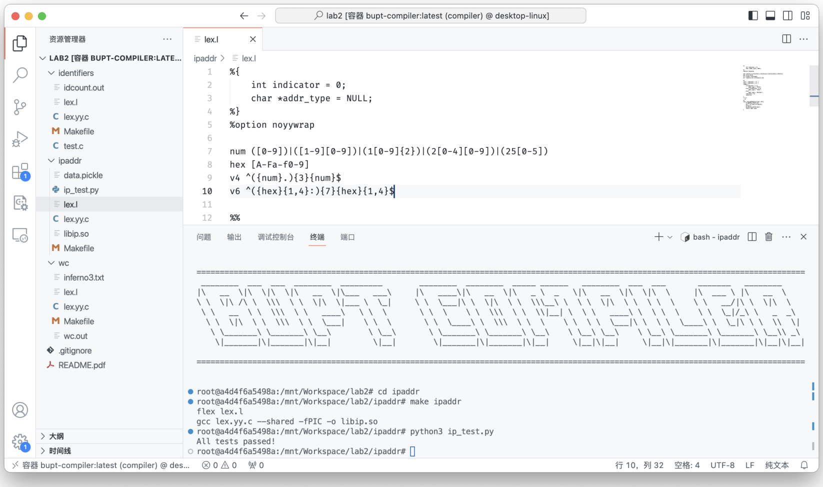


```
%{
    #define EXIT_OK 0
    #define EXIT_FAIL 1

    // global variables
    int identifiers = 0;
    int lines = 1;
%}
letter [a-zA-Z]
letter_ {letter}|_
digit [0-9]

%%
{letter_}({letter_}|{digit})* { identifiers++; printf("line %d: %s\n", lines, yytext);}
\n { lines++; }
[\t\r ]+ { /* does nothing when seeing white spaces except new line */ }
. { /* a final rule that matches when seeing any character but new line */ }
<<EOF>> { printf("There are %d occurrences of valid identifiers\n", identifiers); yyterminate(); }
```

```
root@a4d4f6a5498a:/mnt/Workspace/lab2/identifiers# make idcount
flex lex.l
gcc lex.yy.c -lfl -o idcount.out
root@a4d4f6a5498a:/mnt/Workspace/lab2/identifiers# ./idcount.out ./test.c
line 1: int
line 1: main
line 3: int
line 3: a
line 4: int
line 4: BBA
line 4: a_
line 5: int
line 5: _
line 6: int
line 6: a
line 7: if
line 7: a0
line 7: _
line 7: b0
line 8: else
line 8: _
line 8: b0
line 9: while
line 9: b
line 9: b1
line 9: b0
line 9: b2
line 10: char
line 10: c
line 10: a
line 11: return
There are 27 occurrences of valid identifiers
root@a4d4f6a5498a:/mnt/Workspace/lab2/identifiers# 
```

Figure 2: Result for the exercise on identifiers

3 Flex exercise: ipaddr



```
%{
    int indicator = 0;
    char *addr_type = NULL;
%}
%option noyywrap

num ([0-9])|([1-9][0-9])|(1[0-9]{2})|(2[0-4][0-9])|(25[0-5])
hex [A-Fa-f0-9]
v4 ^({num}.){3}{num}$
v6 ^({hex}{1,4}:){7}{hex}{1,4}$

%%
```

```
root@a4d4f6a5498a:/mnt/Workspace/lab2# cd ipaddr
root@a4d4f6a5498a:/mnt/Workspace/lab2/ipaddr# make ipaddr
flex lex.l
gcc lex.yy.c --shared -fPIC -o libip.so
root@a4d4f6a5498a:/mnt/Workspace/lab2/ipaddr# python3 ip_test.py
All tests passed!
root@a4d4f6a5498a:/mnt/Workspace/lab2/ipaddr# 
```

Figure 3: Result for the exercise on ipaddr

The key part of my code is as follows.

```
num ([0-9])|([1-9][0-9])|(1[0-9]{2})|(2[0-4][0-9])|(25[0-5])
hex [A-Fa-f0-9]
v4 ^({num}.){3}{num}$
```

```
v6 ^({hex}{1,4}:){7}{hex}{1,4}$
```

In this part, num defines the pattern of decimal numbers from 0 to 255. hex defines the pattern of hexadecimal numbers from 0 to f. v4 defines the pattern of an IPv4 address. v6 defines the pattern of an IPv6 address.