CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

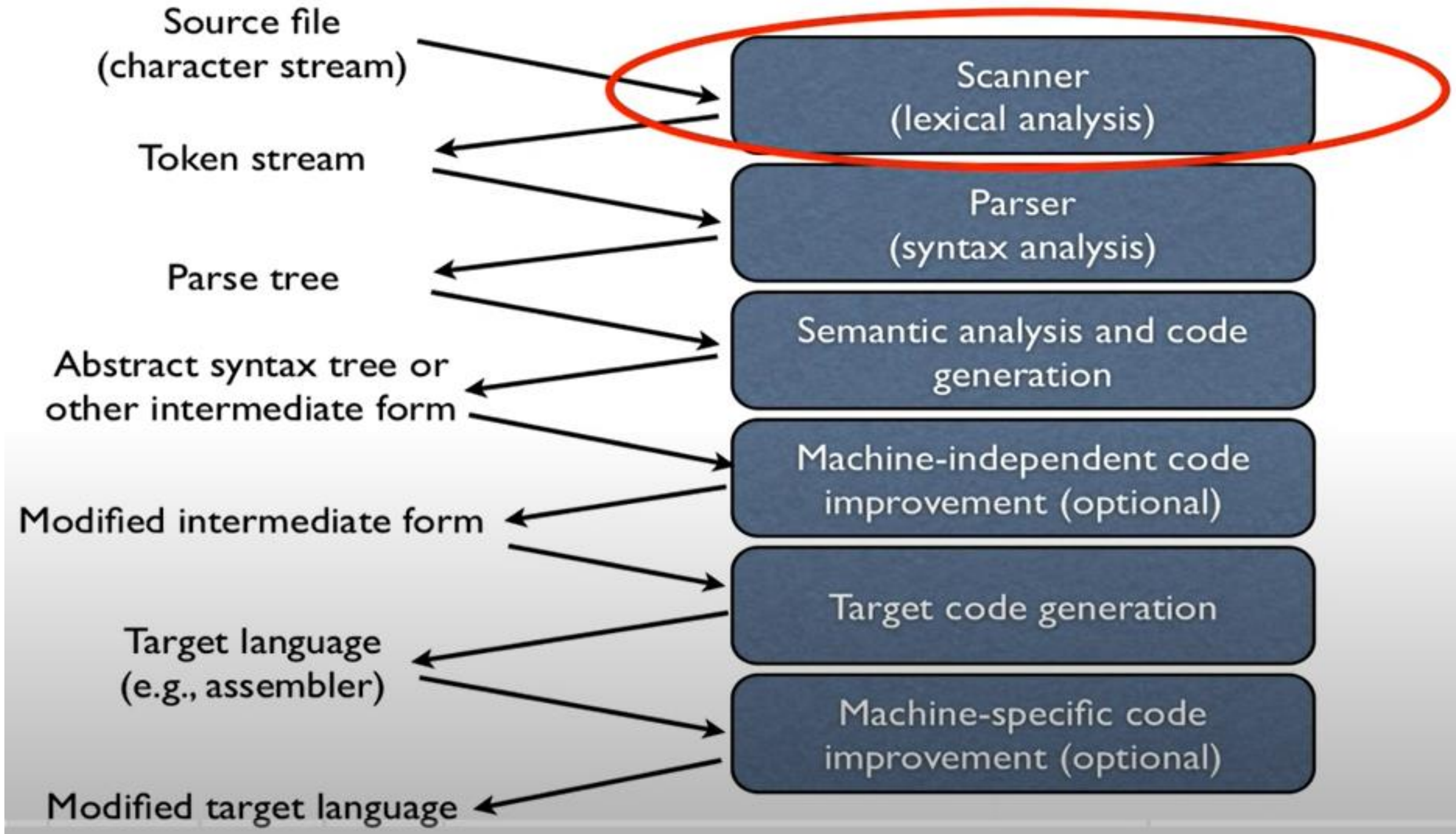
- Introduction to lexical analysis (Review)
    - Language Processing
    - Lexical Analysis
- What is lex?
    - lex / flex

- How does it work?
  - lex input
  - lex input file_1
  - Using lex
- A sample program (a hands-on demo)
  - myscanner.l
  - myscanner.c

Language Processing

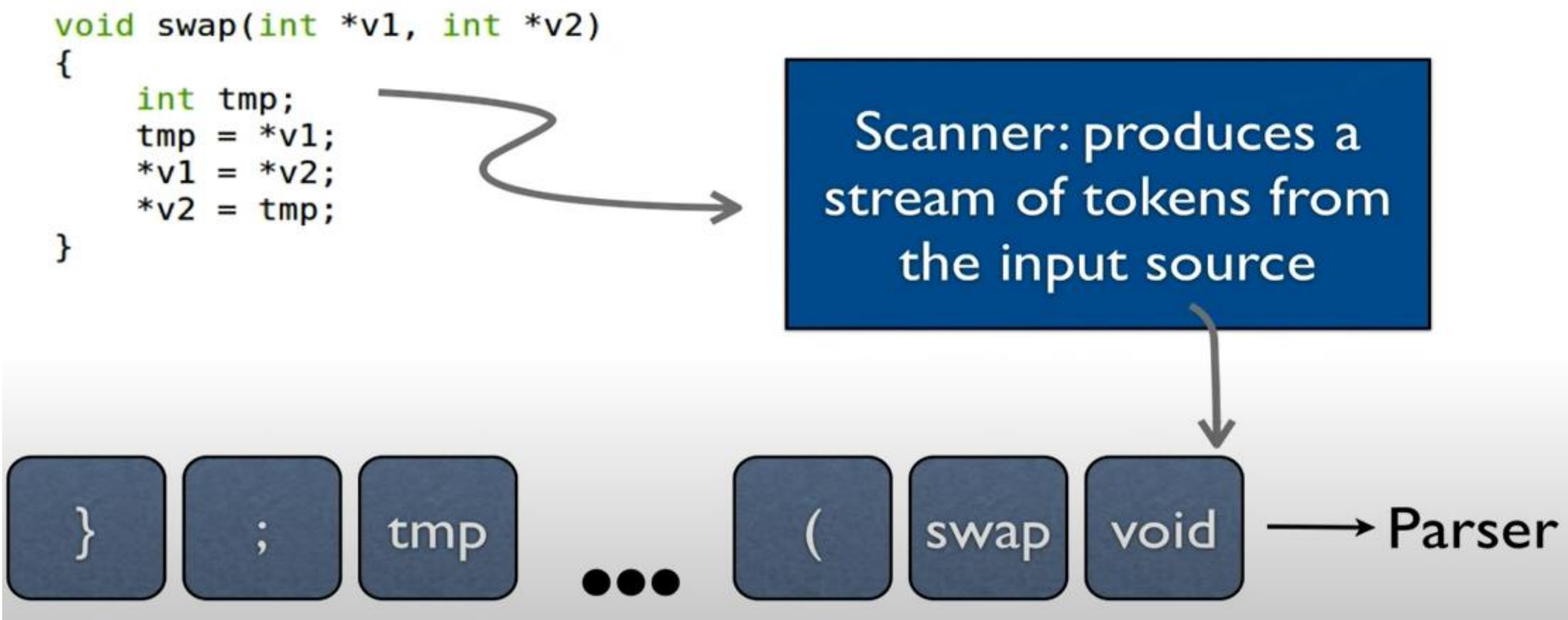
- Let’s review our stacked chart

Our job today is to understand the 1st phase and this is our objective to introduce the ***lex***


Source file (character stream)
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Target language (e.g., assembler)
Machine-specific code improvement (optional)
Modified target language

Lexical Analysis

- The first 2 tokens are “void” and “swap”
- All of these streams of tokens are sent to the Parser to build up the parse tree



```
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
```

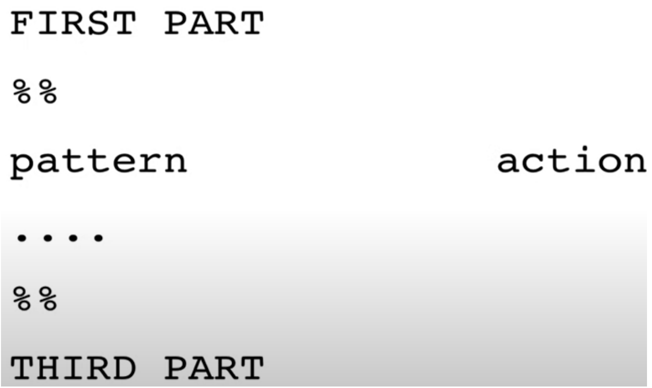
Scanner: produces a stream of tokens from the input source

lex / flex

- Lex is a scanner **generator**
    - Writing a scanner by hand (programming) is not very hard to do if it is a small language
    - It could be very hard if we have larger set of tokens
    - **Input** is a set of **regular expressions** and associated **actions** (in our example, actions are written in C language) (The input file is a .l file)
    - **Output**: when we run lex on input file, lex will generate a table driven scanner (lex.yy.c)
- Flex: **an open source implementation** of the original Unix ***lex*** utility
- When the people are talking about the lex / flex, they are essentially talking about the same utility

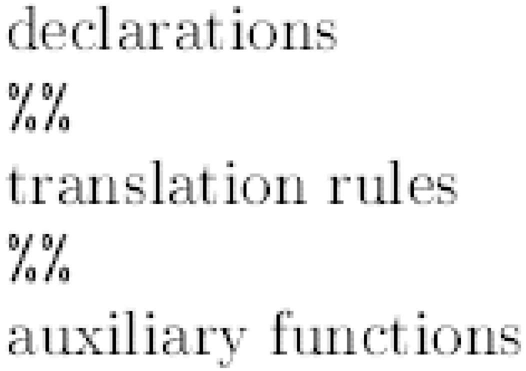
lex input

- lex input is fairly straightforward
- The input file has 3 parts
- The 1st part is optional, we will take a look at that in the examples



```
FIRST PART
%%
pattern                    action
....
%%
THIRD PART
```

This is briefly reviewed in the previous classes

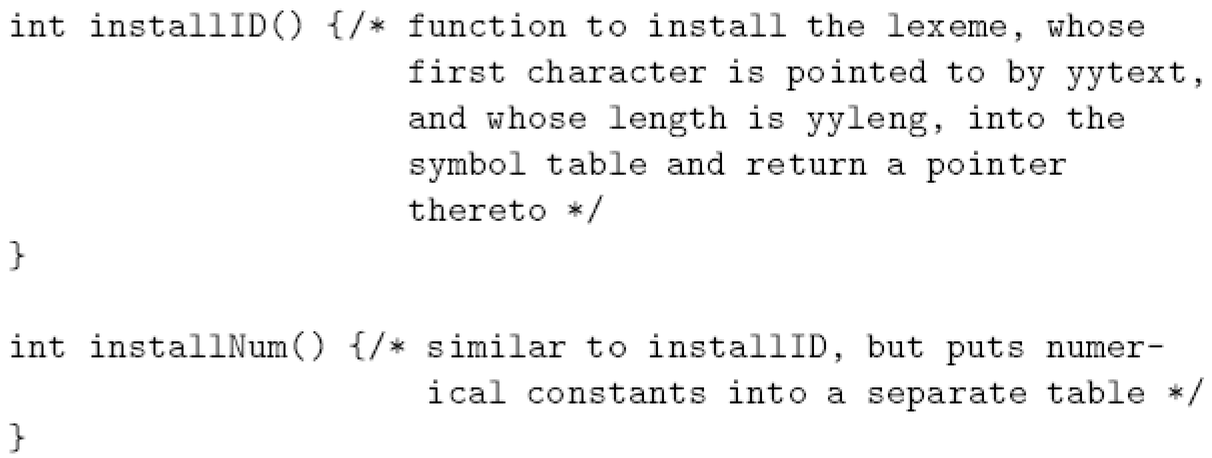


```
declarations
%%
translation rules
%%
auxiliary functions
```

- The 2nd part is a list of the **regular expression** pattern followed by some white space, then **action**
  - The action could be a single C statement (if we have only one thing to do) or a block of statement with {…}
  - The rule of thumb in this part is that, we might have a list of patterns. But Lex is always like to choose the **lexeme** with the “**longest matching prefix**” and the **pattern** (use for matching) is listed **first**
  - Why we need **patterns**? Matching! Of course, we need to look for tokens

lex input

- The 3rd part is also **optional**. In the book, we introduced installID() and installNum() these 2 function.
  - The previous one is to install the lexeme into the symbol table
  - The later one is to install numerical constants into a separate table



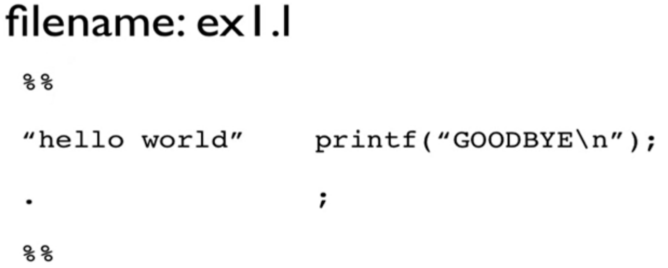
```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

lex input file_1

- The RHS is the whole workflow provided in the book
- This one is a very small example of our ***lex***

code. See? No 2nd and 3rd part.

- It is just the **pattern – action** pair



filename: ex1.l

```
%%
“hello world”    printf(“GOODBYE\n”);
.                ;
%%
```

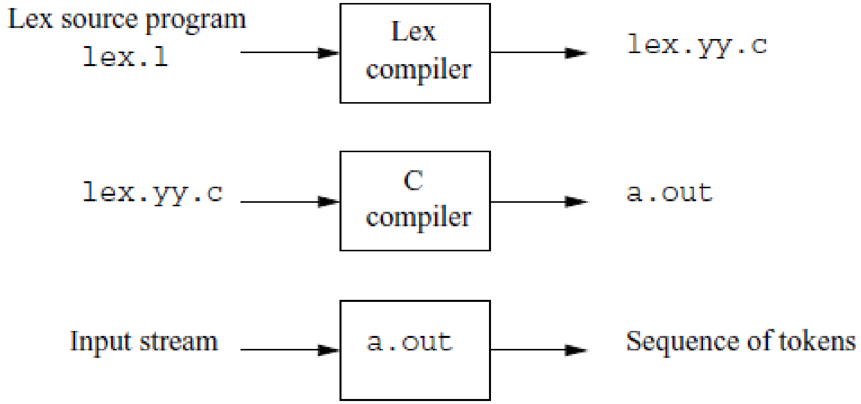
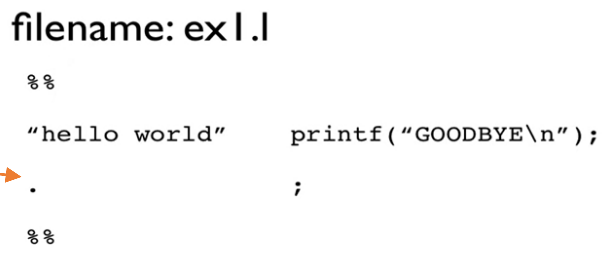


Figure 3.22: Creating a lexical analyzer with Lex

- Believe it or not, “hello world” is a valid regular expression
    - That means, to look for the token “hello world”, the action is to print something
    - If the “hello world” hello world is found, it simply print out a “GOODBYE”

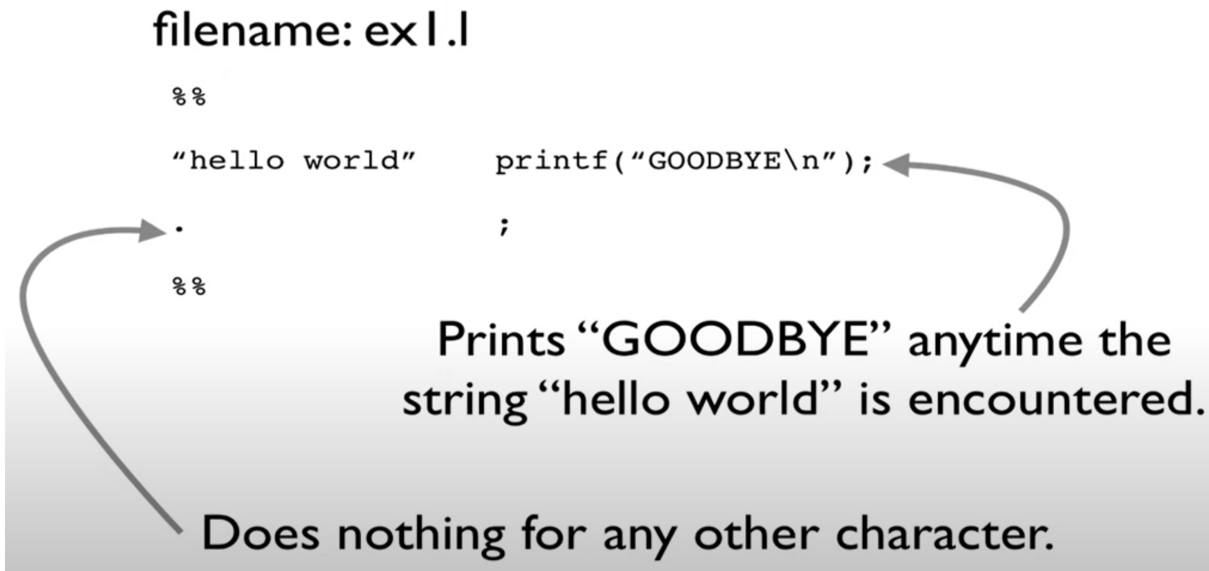
lex input file_1

- What about this, a dot?
  - Match any character, and the action is just **empty C statement. A semi-colon**
- In conclusion, it is saying we only focus on “hello world”.
- For others, we don’t need to care about



filename: ex1.l

```
%%
“hello world”     printf(“GOODBYE\n”);
.                 ;
%%
```



filename: ex1.l

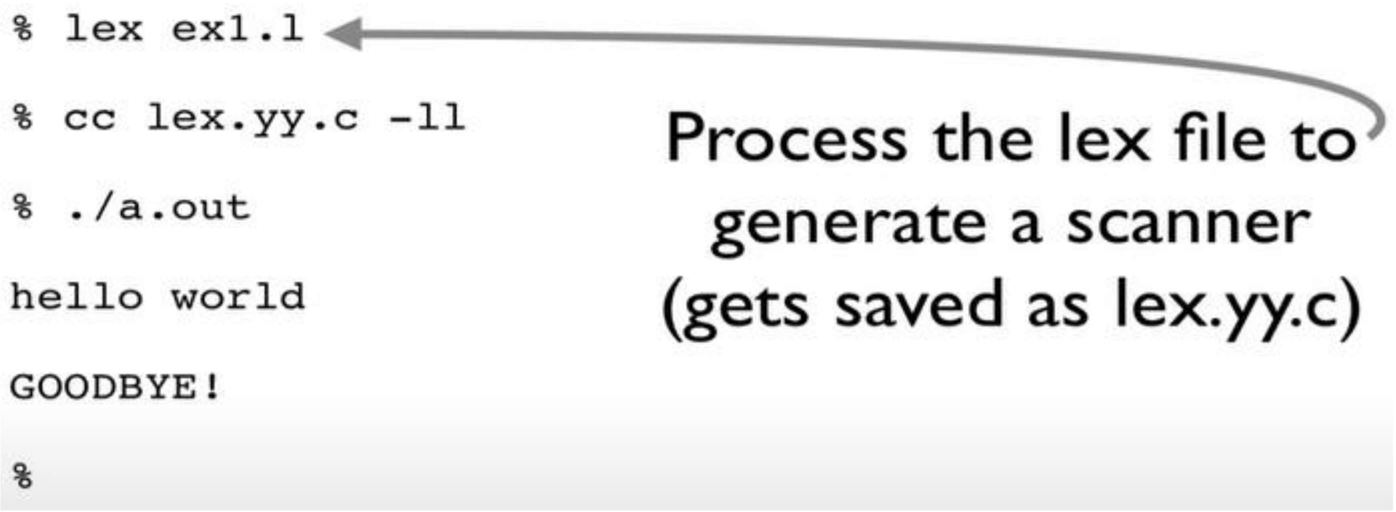
```
%%
“hello world”     printf(“GOODBYE\n”);
.                 ;
%%
```

Prints “GOODBYE” anytime the string “hello world” is encountered.

Does nothing for any other character.

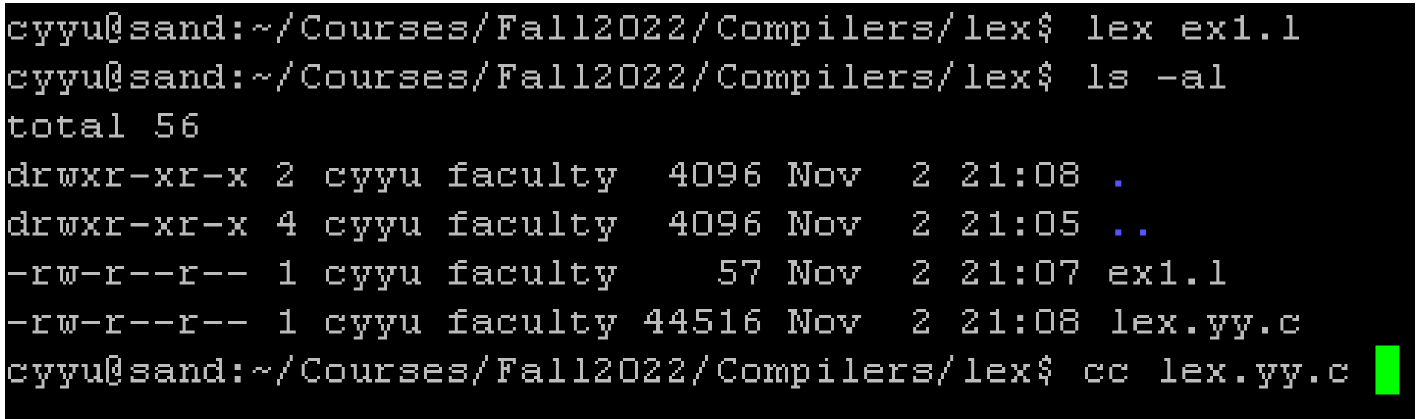
Using lex

- We can run it by first process the ex1.l file
- It generates lex.yy.c file
  - The **generated** scanner / tokenizer file!
- Then, I’m going to use cc to compile the scanner and grab main() from the lex library (-ll option)



```
% lex ex1.l

% cc lex.yy.c -ll

% ./a.out

hello world

GOODBYE!

%
```

Process the lex file to generate a scanner (gets saved as lex.yy.c)

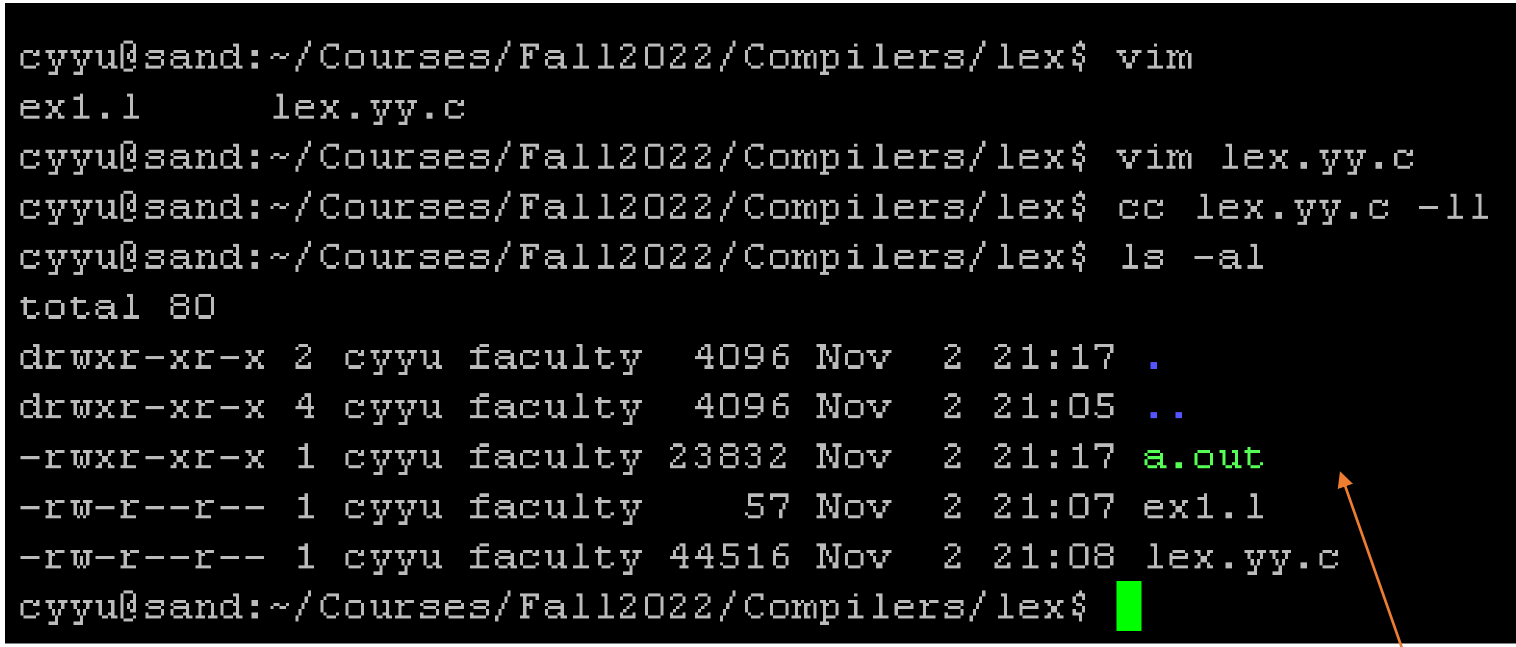


```
cyyu@sand:~/Courses/Fall2022/Compilers/lex$ lex ex1.l
cyyu@sand:~/Courses/Fall2022/Compilers/lex$ ls -al
total 56
drwxr-xr-x 2 cyyu faculty  4096 Nov  2 21:08 .
drwxr-xr-x 4 cyyu faculty  4096 Nov  2 21:05 ..
-rw-r--r-- 1 cyyu faculty    57 Nov  2 21:07 ex1.l
-rw-r--r-- 1 cyyu faculty 44516 Nov  2 21:08 lex.yy.c
cyyu@sand:~/Courses/Fall2022/Compilers/lex$ cc lex.yy.c
```

Using lex

- You can see the “a.out” after the execution of cc
- Now, if I directly run the a.out?

(See next page for detail)



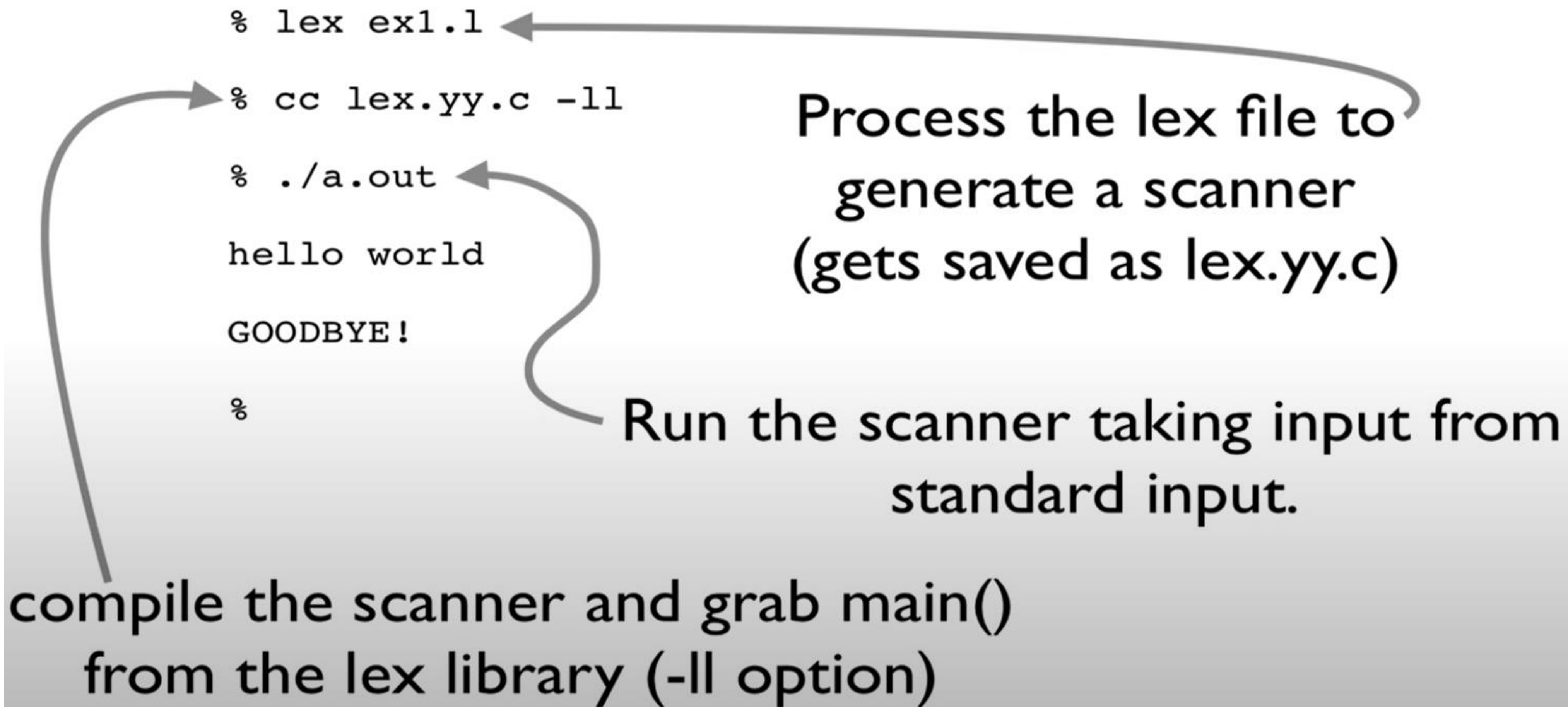
```
cyyu@sand:~/Courses/Fall2022/Compilers/lex$ vim
ex1.l     lex.yy.c
cyyu@sand:~/Courses/Fall2022/Compilers/lex$ vim lex.yy.c
cyyu@sand:~/Courses/Fall2022/Compilers/lex$ cc lex.yy.c -ll
cyyu@sand:~/Courses/Fall2022/Compilers/lex$ ls -al
total 80
drwxr-xr-x 2 cyyu faculty  4096 Nov  2 21:17 .
drwxr-xr-x 4 cyyu faculty  4096 Nov  2 21:05 ..
-rwxr-xr-x 1 cyyu faculty 23832 Nov  2 21:17 a.out
-rw-r--r-- 1 cyyu faculty    57 Nov  2 21:07 ex1.l
-rw-r--r-- 1 cyyu faculty 44516 Nov  2 21:08 lex.yy.c
cyyu@sand:~/Courses/Fall2022/Compilers/lex$ 
```

Using lex

- Note that if you run ./a.out, the program will be waiting there for user’s input.
- Now we can just input “hello world”, then press [Enter]
    - After a quick match, it will print out a GOODBYE

Using lex

- A very simple example to generate a scanner. Define it, compile it and run it



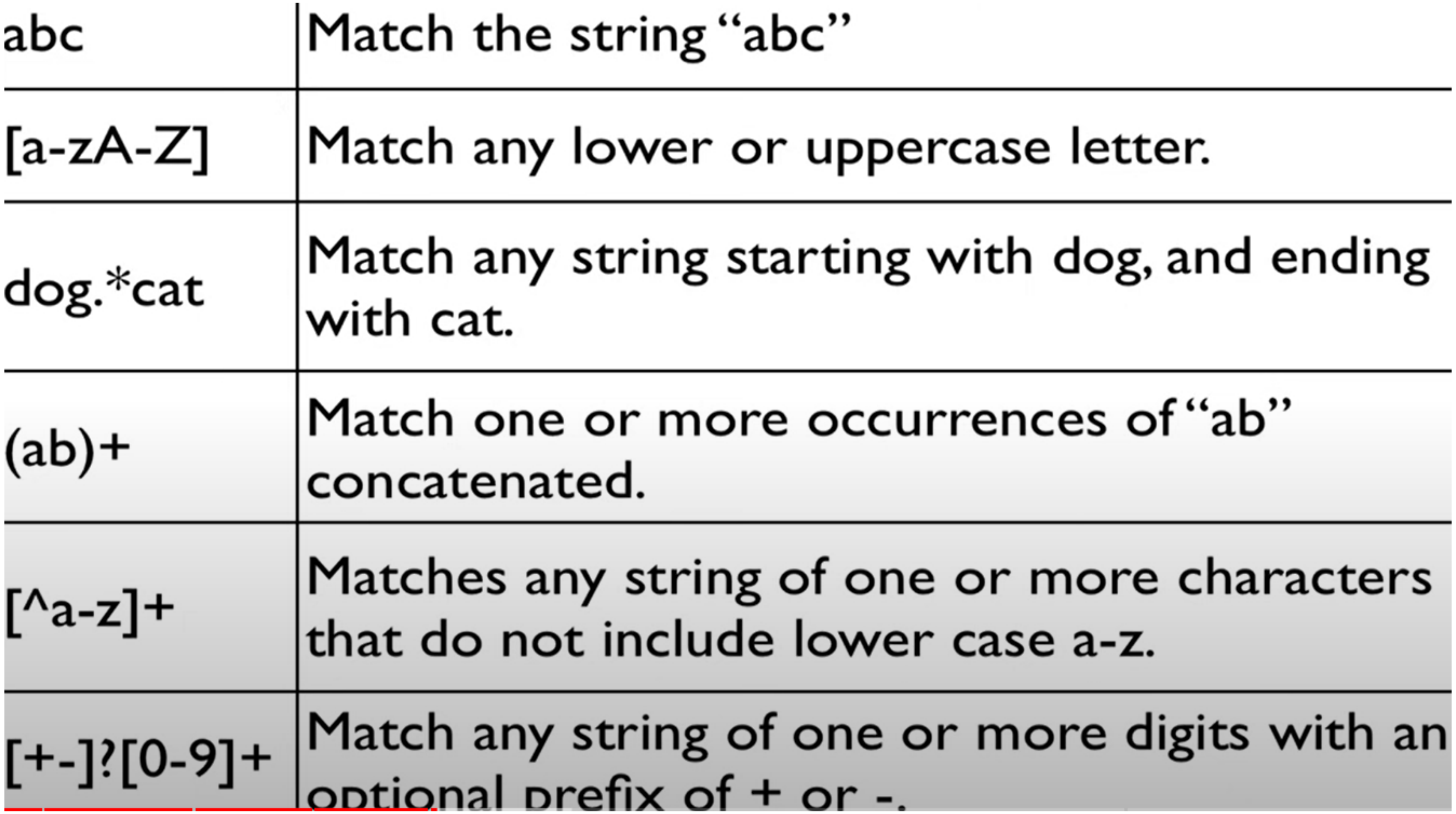
```
% lex ex1.l

% cc lex.yy.c -ll

% ./a.out

hello world

GOODBYE!

%
```

Process the lex file to generate a scanner (gets saved as lex.yy.c)

Run the scanner taking input from standard input.

compile the scanner and grab main() from the lex library (-ll option)

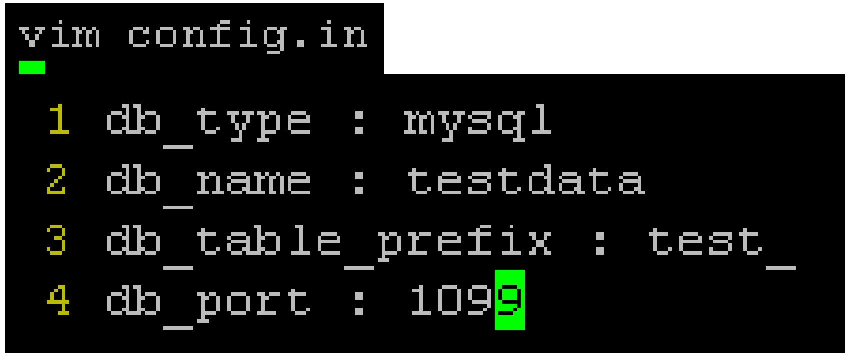
Lex pattern examples



abc	Match the string "abc"
[a-zA-Z]	Match any lower or uppercase letter.
dog.*cat	Match any string starting with dog, and ending with cat.
(ab)+	Match one or more occurrences of "ab" concatenated.
[^a-z]+	Matches any string of one or more characters that do not include lower case a-z.
[+-]?[0-9]+	Match any string of one or more digits with an optional prefix of + or -.

A sample program (a hands-on demo)

- In this example, I’m going to process a textual configuration file
  - config.in
- We are just trying to show you that how to use *lex.* In the realistic case, it could be very complicated

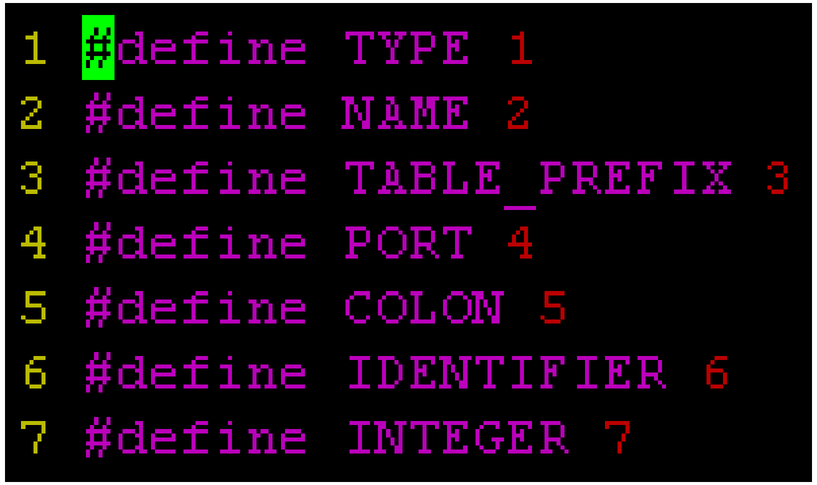


```
vim config.in
```

```
db_type : mysql
db_name : testdata
db_table_prefix : test_
db_port : 1099
```

A sample program (a hands-on demo)

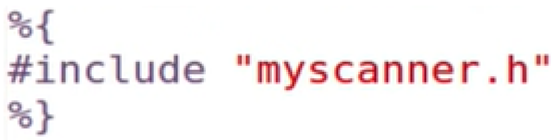
- Let’s go create the scanner called myscanner.h
- What I’m going to find a symbol for each type of token that is in my config file --- config.in
- Here’s the content of myscanner.h
- Those are the **types** of the **tokens** that “myscanner” has to recognize
- Next step --- write our own lex file



```
#define TYPE 1
#define NAME 2
#define TABLE_PREFIX 3
#define PORT 4
#define COLON 5
#define IDENTIFIER 6
#define INTEGER 7
```

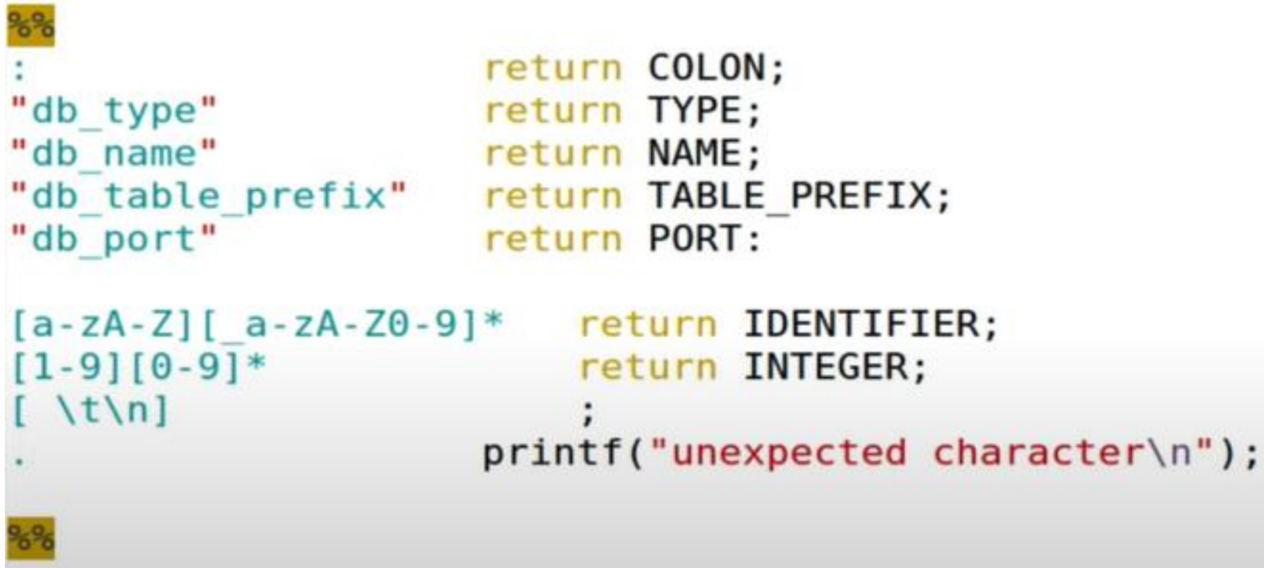
A sample program (a hands-on demo)

- The 1st part of my lex file (myscanner.l)
- The 2nd part is the patterns and actions
  - A list of tokens I'm going to take care of
  - I also used regular expressions to recognize ID as well as INTEGERS
  - I ignore white spaces, tabs, line feeds
  - Any other characters will be invalid



```
%{
#include "myscanner.h"
%}
```

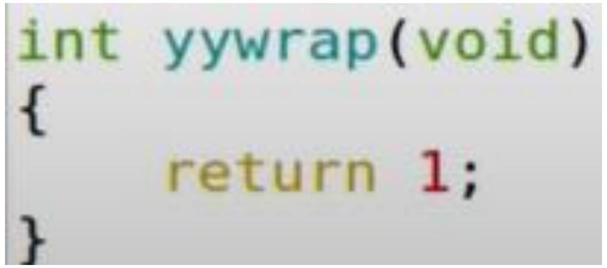
- We make the db_type in the list in front of the db_name makes sense.
- We sometimes need to understand the file structure. If we switched the order of db_type and db_name, we might never get db_type matched



```
%%
:                      return COLON;
"db_type"              return TYPE;
"db_name"              return NAME;
"db_table_prefix"      return TABLE_PREFIX;
"db_port"              return PORT:

[a-zA-Z][_a-zA-Z0-9]*     return IDENTIFIER;
[1-9][0-9]*               return INTEGER;
[ \t\n]                   ;
.                      printf("unexpected character\n");

%%
```

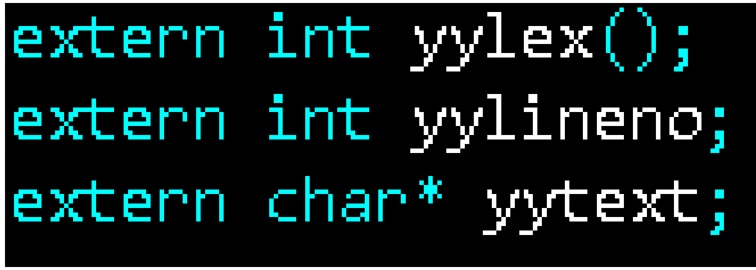
A sample program (a hands-on demo)

- Since we are going to incorporate that into a C program, we need to define a yywrap() function in our **3rd part**



```
int yywrap(void)
{
        return 1;
}
```

- Let’s see if this can compile?
    - lex myscanner.l
    - Looks good, no errors at all.
    - lex.yy.c is generated
- The last thing we need to do is to write a simple C program to utilize this – **myscanner.c**
    - We will need something in the beginning of this program!
    - yyline number is to give us more information when we are parsing to generate helpful error message
    - yytext? Check the slide #8
    - Tell the compiler, these are 3 things defined in other module Please link and connect with those things from the “external” files

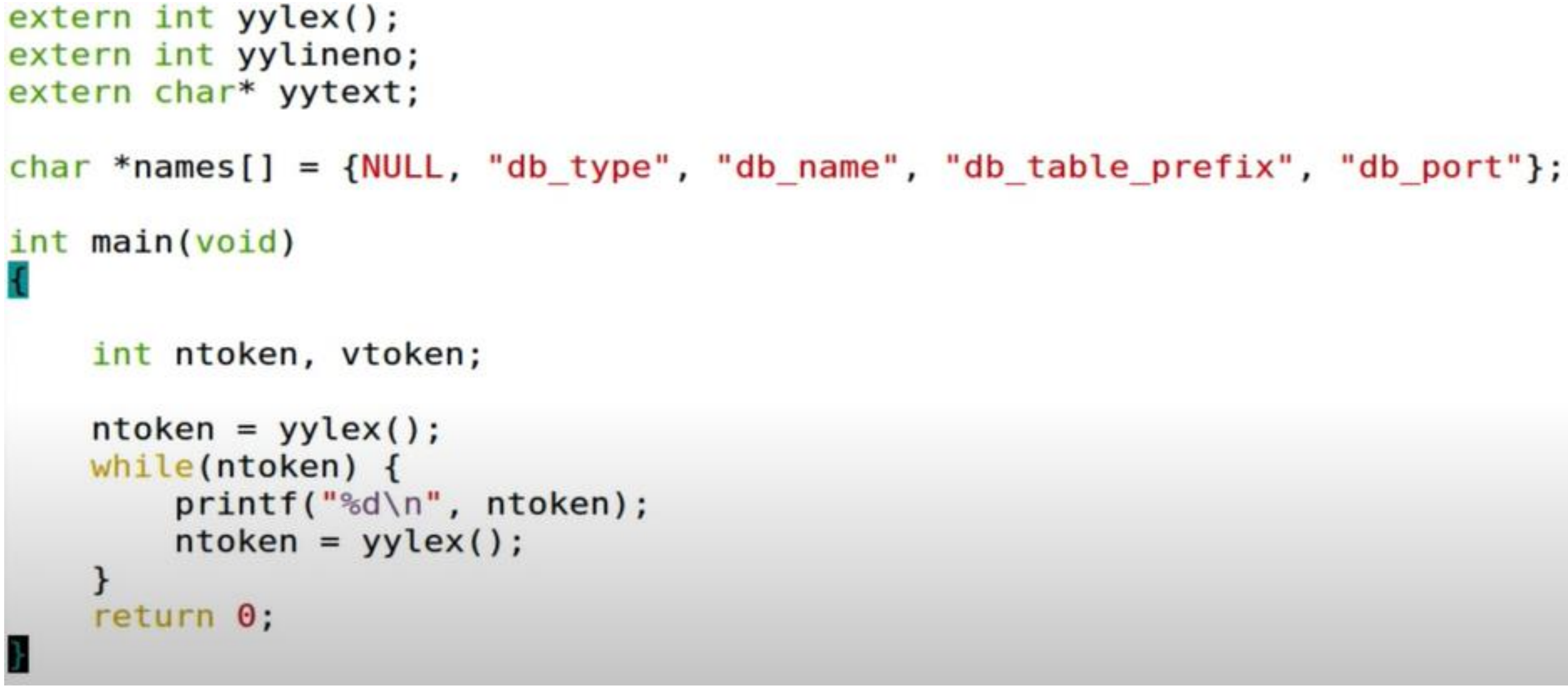


```
extern int yylex();
extern int yylineno;
extern char* yytext;
```

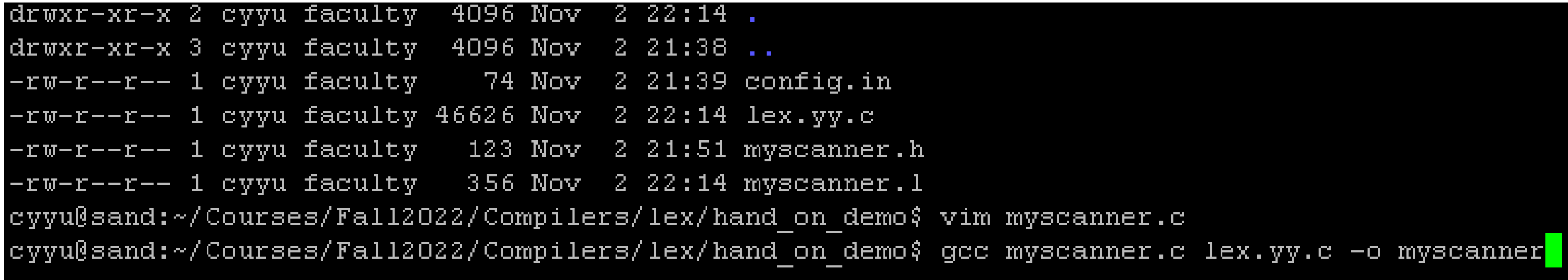
A sample program (a hands-on demo)

- **See if this run as expected?**

(myscanner.c)



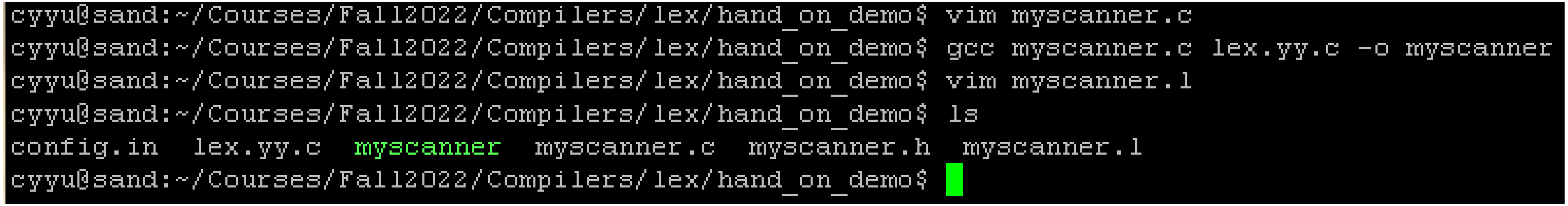
```c
extern int yylex();
extern int yylineno;
extern char* yytext;

char *names[] = {NULL, "db_type", "db_name", "db_table_prefix", "db_port"};

int main(void)
{

    int ntoken, vtoken;

    ntoken = yylex();
    while(ntoken) {
        printf("%d\n", ntoken);
        ntoken = yylex();
    }
    return 0;
}
```



```
drwxr-xr-x 2 cyyu faculty  4096 Nov  2 22:14 .
drwxr-xr-x 3 cyyu faculty  4096 Nov  2 21:38 ..
-rw-r--r-- 1 cyyu faculty    74 Nov  2 21:39 config.in
-rw-r--r-- 1 cyyu faculty 46626 Nov  2 22:14 lex.yy.c
-rw-r--r-- 1 cyyu faculty   123 Nov  2 21:51 myscanner.h
-rw-r--r-- 1 cyyu faculty   356 Nov  2 22:14 myscanner.l
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ vim myscanner.c
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ gcc myscanner.c lex.yy.c -o myscanner
```

A sample program (a hands-on demo)

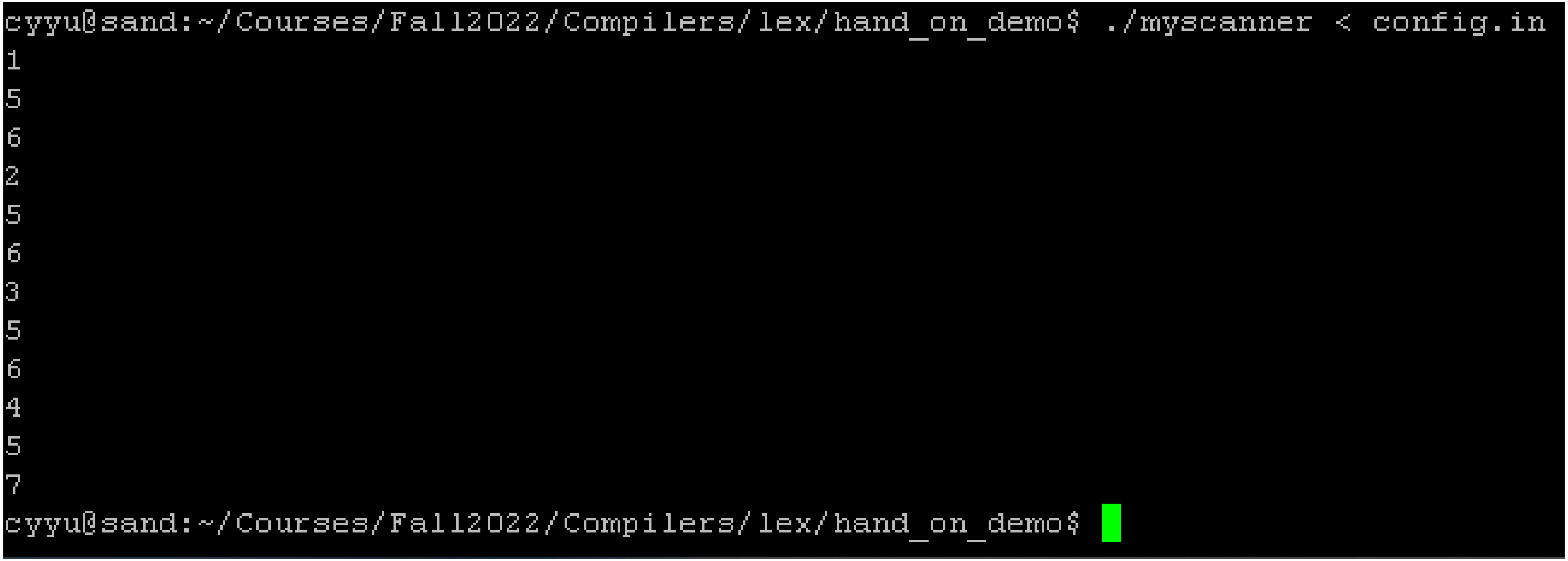
- It looks good!



```
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ vim myscanner.c
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ gcc myscanner.c lex.yy.c -o myscanner
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ vim myscanner.l
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ ls
config.in  lex.yy.c  myscanner  myscanner.c  myscanner.h  myscanner.l
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$
```

- We should be able to run the program and feed it by our input file
  - ./myscanner < config.in

A sample program (a hands-on demo)



```
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ ./myscanner < config.in
1
5
6
2
5
6
3
5
6
4
5
7
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ 
```

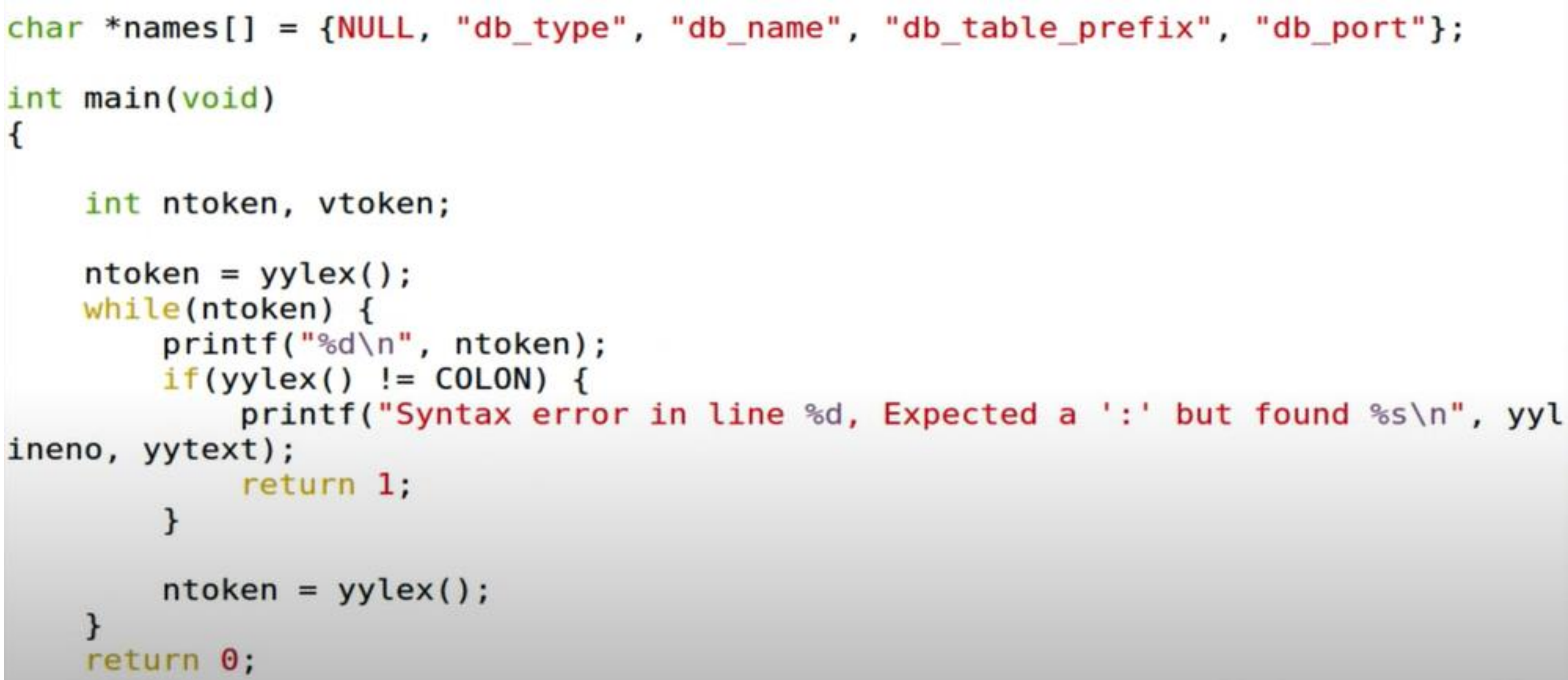
It returns a stream of tokens --- the integers

A sample program (a hands-on demo)

- So, let’s just polish our program a little bit more
- Check our config.in file:
  - If we got a name, the next thing, it has to be a colon →:
  - If it is not a colon, we can print out informative error message
- We expect colon but we found something else, we can actually retrieve this value by using “**yylineno**” (the line # the scanner is currently on), and “**yytext**” is the textual representation of the token it returned us.
- If we had an error, we **exit** right away
- (see next page for our “polishes”)

A sample program (a hands-on demo)

- If it is not a colon, we are going to print out an error

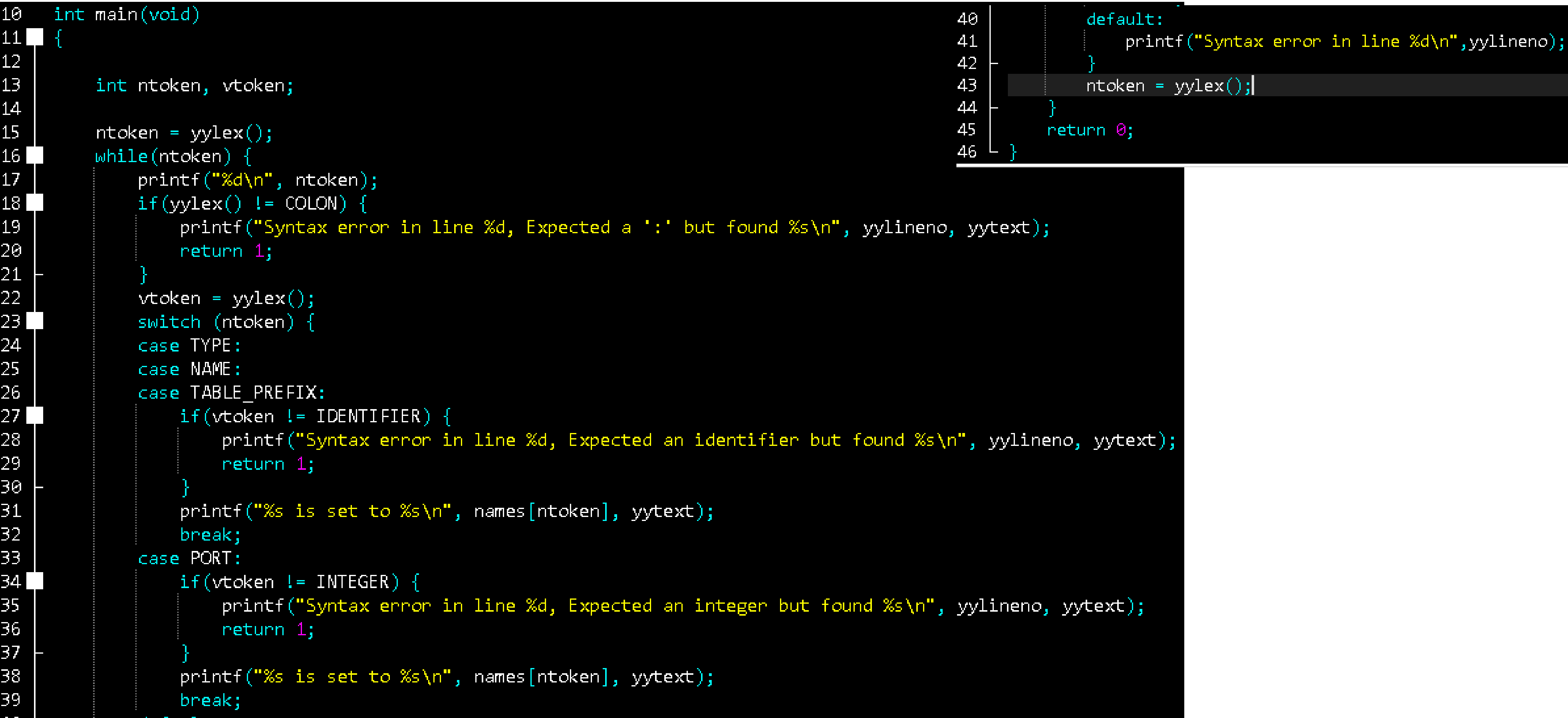


```
char *names[] = {NULL, "db_type", "db_name", "db_table_prefix", "db_port"};

int main(void)
{

    int ntoken, vtoken;

    ntoken = yylex();
    while(ntoken) {
        printf("%d\n", ntoken);
        if(yylex() != COLON) {
            printf("Syntax error in line %d, Expected a ':' but found %s\n", yyl
ineno, yytext);
            return 1;
        }

        ntoken = yylex();
    }
    return 0;
```

A sample program (a hands-on demo)

- Now, we are going to process the **value** token
    - We can do something meaningful for these values
    - **Switch** on the name token
    - For TABLE_PREFIX, if the value token is not an ID, we print out some errors
        - If it is an ID, we print out some message saying it is legal.
    - Same thing, if it is a port, we actually expect a number
- See next page for our modified main()

A sample program (a hands-on demo)

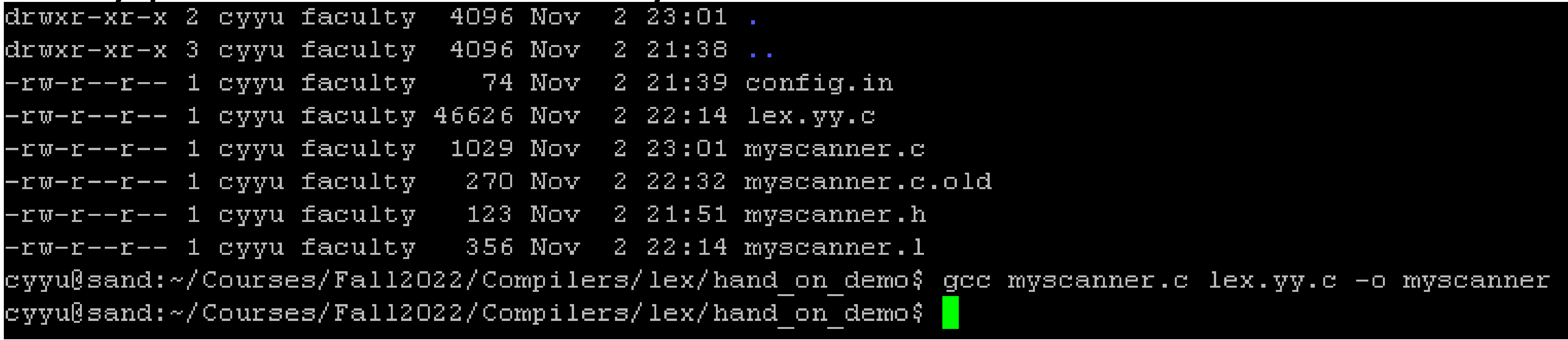


```c
int main(void)
{

    int ntoken, vtoken;

    ntoken = yylex();
    while(ntoken) {
        printf("%d\n", ntoken);
        if(yylex() != COLON) {
            printf("Syntax error in line %d, Expected a ':' but found %s\n", yylineno, yytext);
            return 1;
        }
        vtoken = yylex();
        switch (ntoken) {
        case TYPE:
        case NAME:
        case TABLE_PREFIX:
            if(vtoken != IDENTIFIER) {
                printf("Syntax error in line %d, Expected an identifier but found %s\n", yylineno, yytext);
                return 1;
            }
            printf("%s is set to %s\n", names[ntoken], yytext);
            break;
        case PORT:
            if(vtoken != INTEGER) {
                printf("Syntax error in line %d, Expected an integer but found %s\n", yylineno, yytext);
                return 1;
            }
            printf("%s is set to %s\n", names[ntoken], yytext);
            break;
```

```c
        default:
            printf("Syntax error in line %d\n",yylineno);
        }
        ntoken = yylex();
    }
    return 0;
}
```

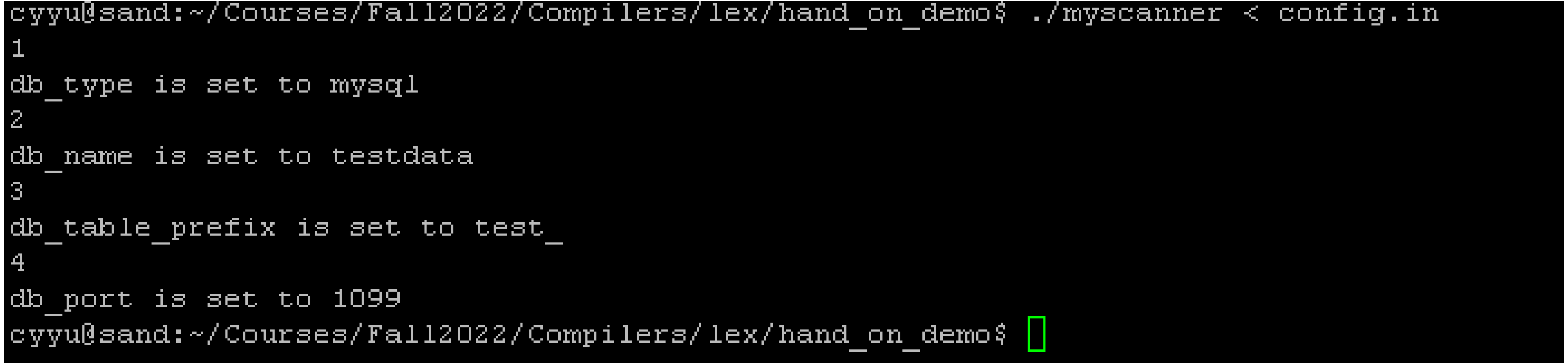
A sample program (a hands-on demo)

- OK. Compile and run it!
- I just move my previous version as myscanner.old



```
drwxr-xr-x 2 cyyu faculty  4096 Nov  2 23:01 .
drwxr-xr-x 3 cyyu faculty  4096 Nov  2 21:38 ..
-rw-r--r-- 1 cyyu faculty    74 Nov  2 21:39 config.in
-rw-r--r-- 1 cyyu faculty 46626 Nov  2 22:14 lex.yy.c
-rw-r--r-- 1 cyyu faculty  1029 Nov  2 23:01 myscanner.c
-rw-r--r-- 1 cyyu faculty   270 Nov  2 22:32 myscanner.c.old
-rw-r--r-- 1 cyyu faculty   123 Nov  2 21:51 myscanner.h
-rw-r--r-- 1 cyyu faculty   356 Nov  2 22:14 myscanner.l
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ gcc myscanner.c lex.yy.c -o myscanner
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$
```

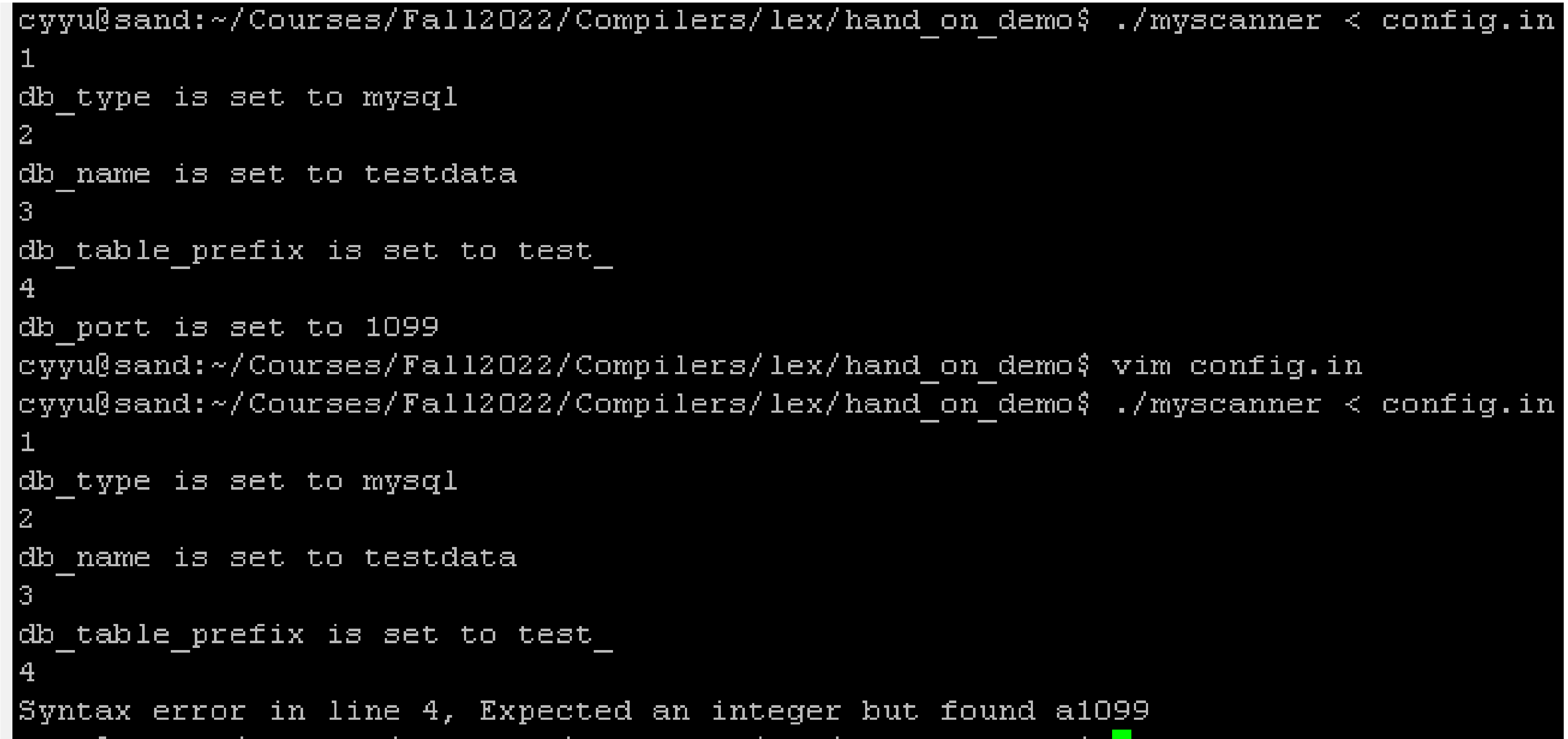
- Run it!



```
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ ./myscanner < config.in
1
db_type is set to mysql
2
db_name is set to testdata
3
db_table_prefix is set to test_
4
db_port is set to 1099
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$
```

A sample program (a hands-on demo)

- Test our error handling
- If we change this to the port to a1099, the error will be caught!



```
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ ./myscanner < config.in
1
db_type is set to mysql
2
db_name is set to testdata
3
db_table_prefix is set to test_
4
db_port is set to 1099
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ vim config.in
cyyu@sand:~/Courses/Fall2022/Compilers/lex/hand_on_demo$ ./myscanner < config.in
1
db_type is set to mysql
2
db_name is set to testdata
3
db_table_prefix is set to test_
4
Syntax error in line 4, Expected an integer but found a1099
```

A sample program (a hands-on demo)

- A very brief introduction to use the *lex,* to generate the tokenizer and incorporate into our C program

Thank you