Table of Contents

[Tanvir Anjom Siddique 2](#_Toc167655670)

[Lab 1: 2](#_Toc167655671)

[Setup Terminal: 2](#_Toc167655672)

[Compile Code: 3](#_Toc167655673)

[Preprocess: 3](#_Toc167655674)

[Create Assembly file (.i to .s): 4](#_Toc167655675)

[\*\*\* All Steps Of Compiling C Program \*\*\* 6](#_Toc167655676)

[Explanation 6](#_Toc167655677)

[Summary 8](#_Toc167655678)

[# Build System: 8](#_Toc167655679)

[\*\*Makefile (compile multiple file): 8](#_Toc167655680)

[Flex (lexical analysis): [Identify ID , VAR, STRING, OP] 10](#_Toc167655681)

[\*\*\* Highlight cal.l > bison & lex (extension install) 11](#_Toc167655682)

[Lab 02: Lexical & Syntax Analysis 14](#_Toc167655683)

[Identify Token 14](#_Toc167655684)

[Read input from a file 15](#_Toc167655685)

[Use RE rules: 15](#_Toc167655686)

[This don’t check syntax but token only 17](#_Toc167655687)

[BISON: 18](#_Toc167655688)

[Parse Program line: int a=10; 20](#_Toc167655689)

[Parse While loop: 23](#_Toc167655690)

[\*\*\* Parser (cal.y ->cal.tab.c, cal.tab.h ) , Lexical Analyzer (cal.l -> lex.yy.c) all commands Makefile\*\*\* 26](#_Toc167655691)

[1. **Generate Parser**: 26](#_Toc167655692)

[2. **Generate Lexical Analyzer**: 27](#_Toc167655693)

[3. **Compile Lexer and Parser**: 27](#_Toc167655694)

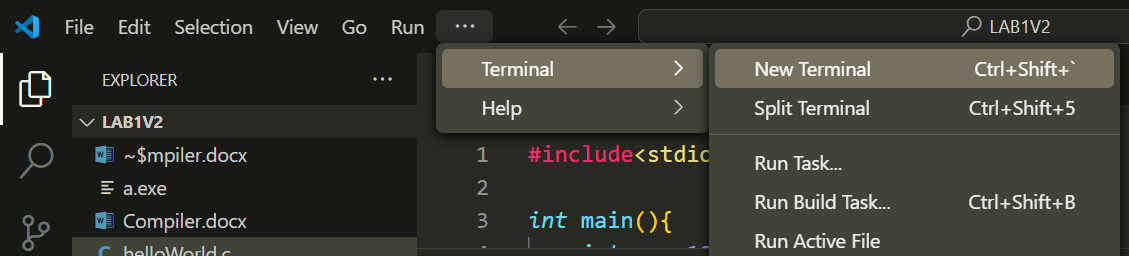
[4. **Run the Executable**: 27](#_Toc167655695)

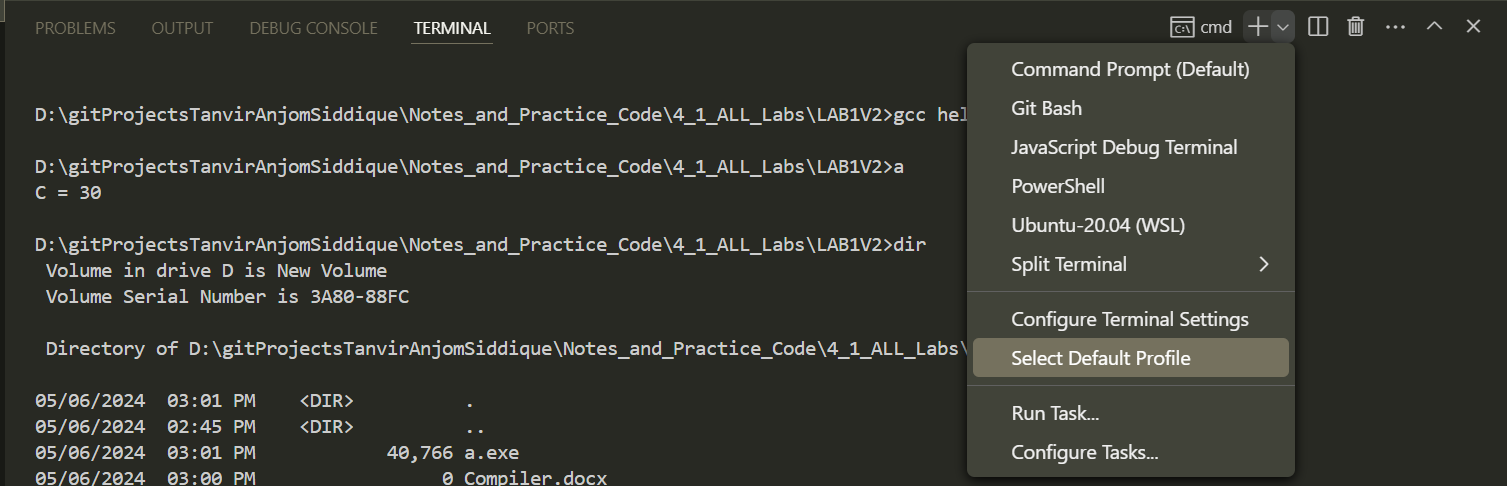
[\*\*\* Parse & Lexically Analyze “While loop” in short: \*\*\* 27](#_Toc167655696)

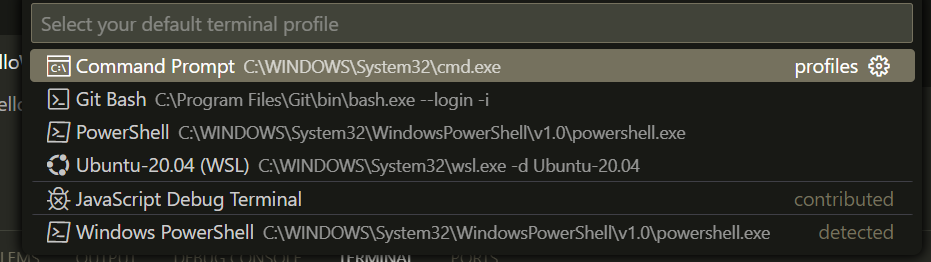
# Tanvir Anjom Siddique

# Lab 1:

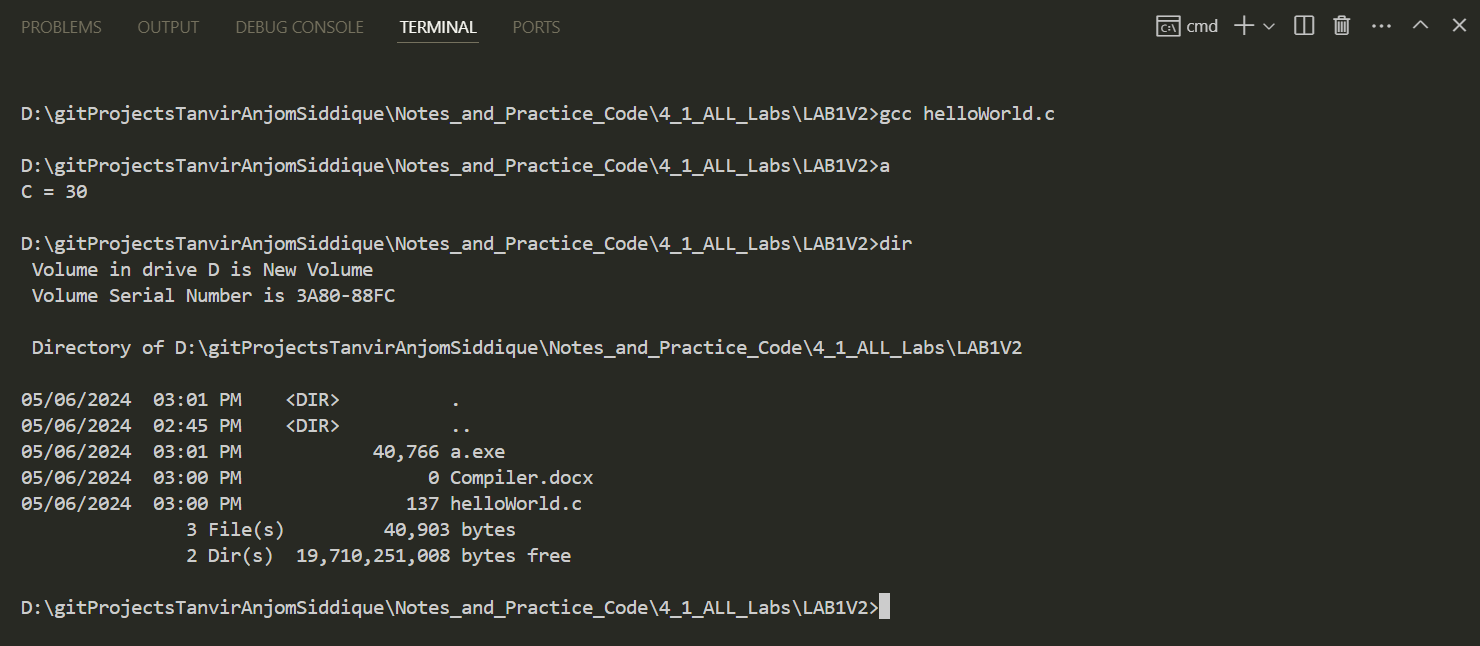
## Setup Terminal:

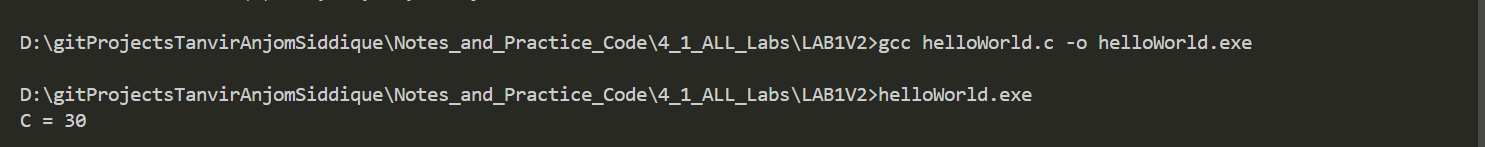


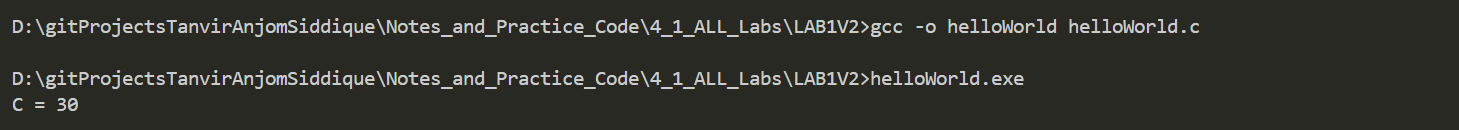




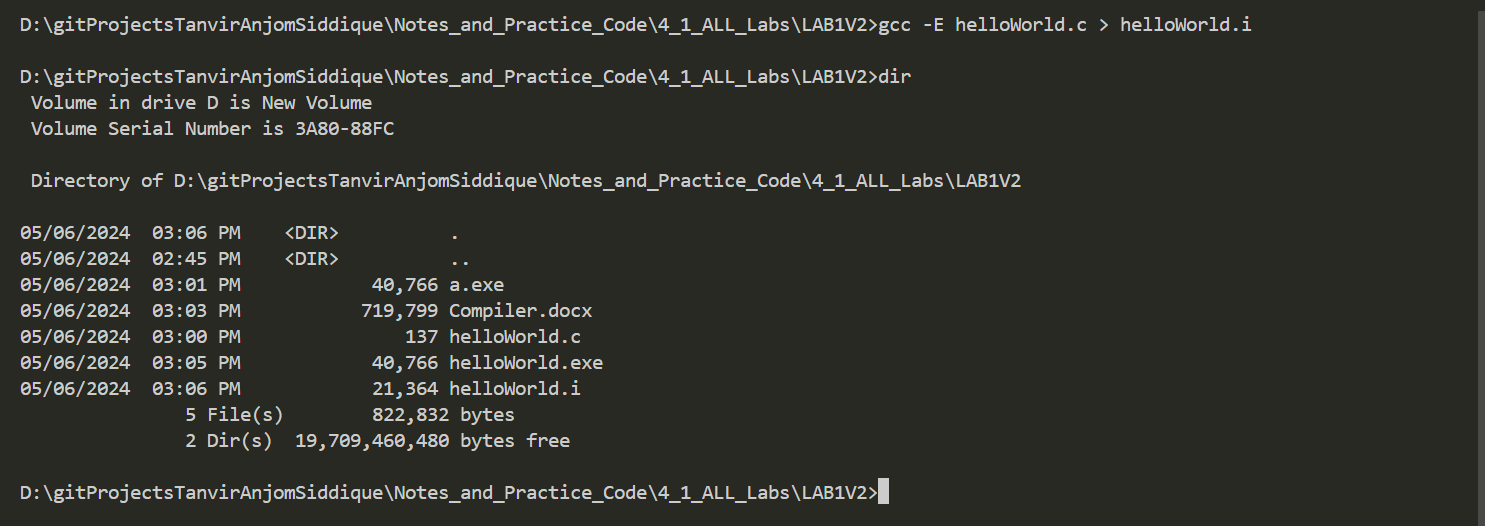
## Compile Code:





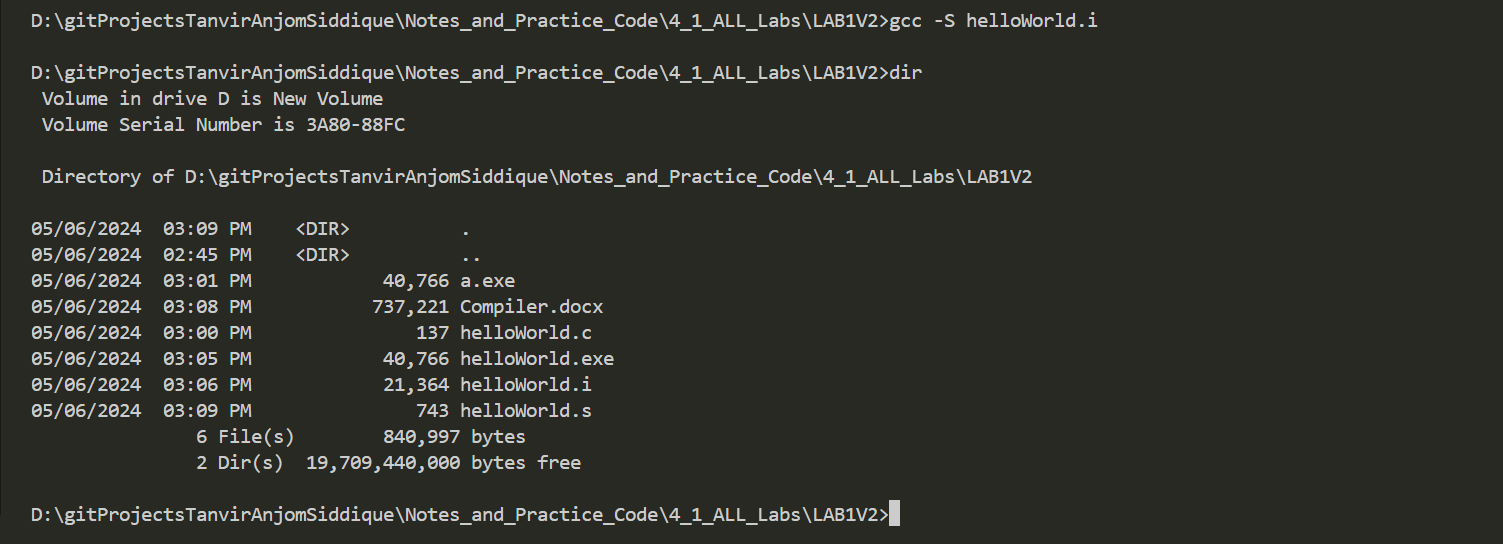


## Preprocess:



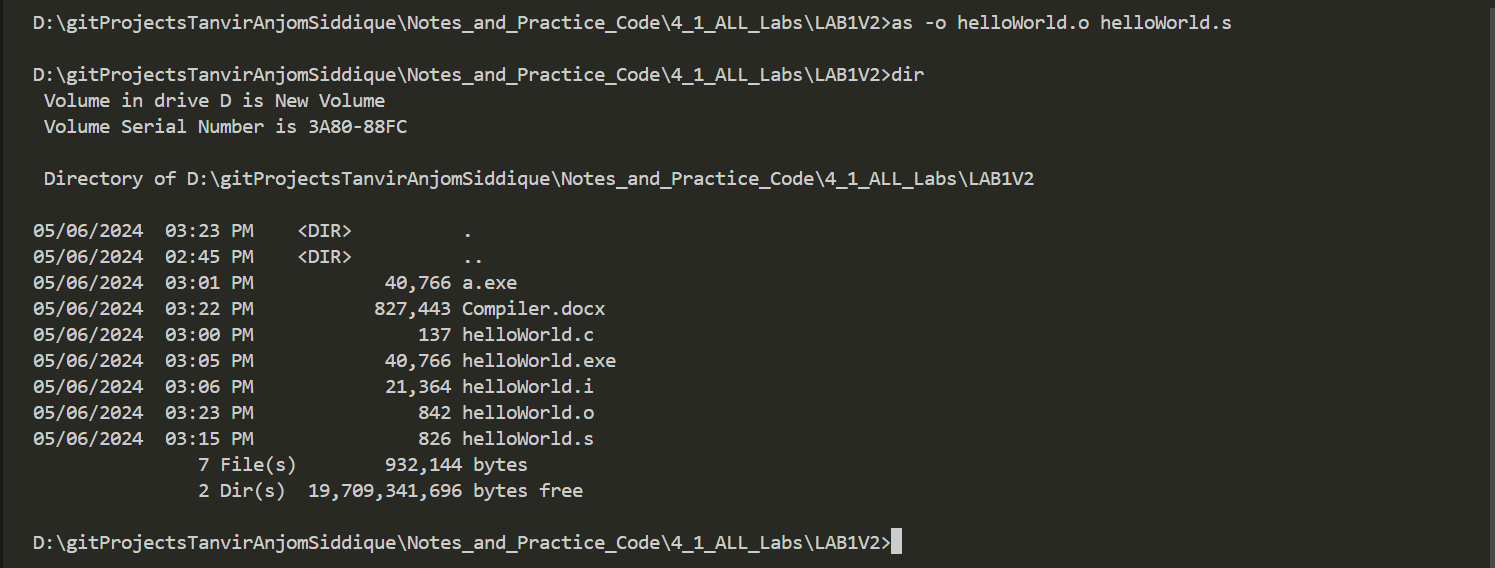
.i file 🡪 Code of header files are pasted before your main .c code

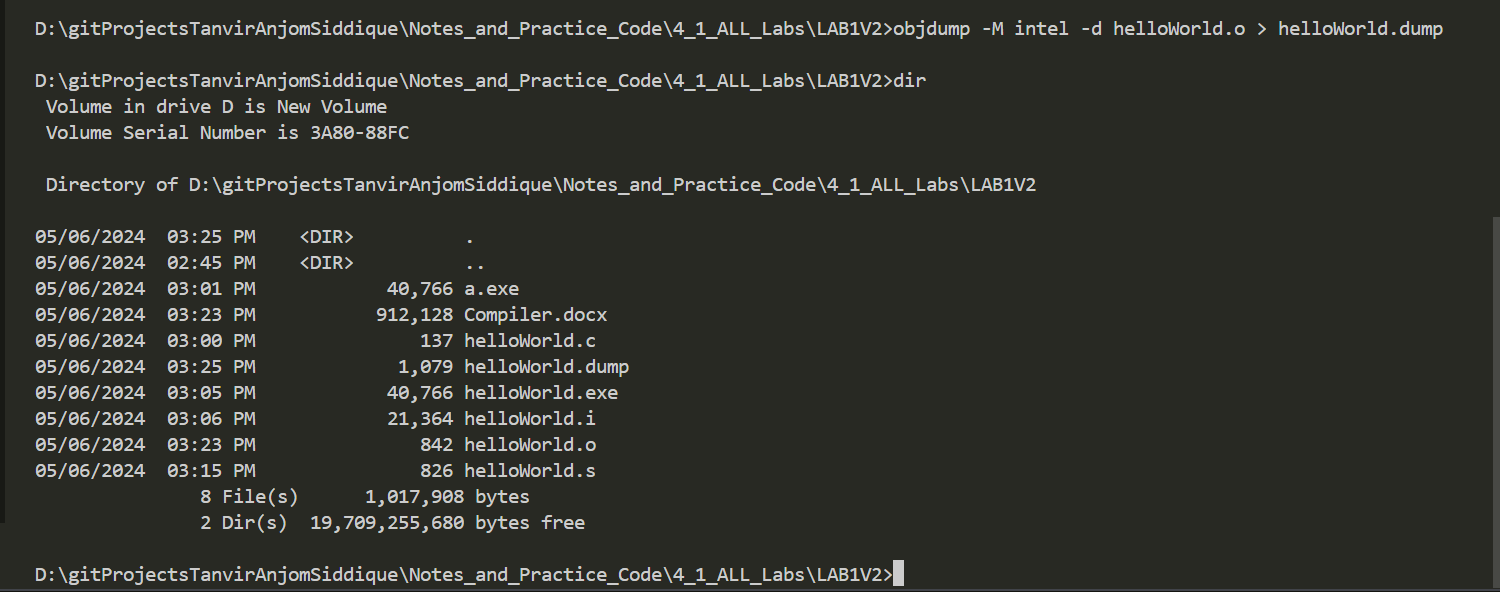
## Create Assembly file (.i to .s):



Assembly language for masm Intel:



Create .s to .o: 

Create .dump file (Machine code that you can read): 

|  |
| --- |
| helloWorld.o: file format pe-i386  Disassembly of section .text:  00000000 <\_main>:  0: 55 push ebp  1: 89 e5 mov ebp,esp  3: 83 e4 f0 and esp,0xfffffff0  6: 83 ec 20 sub esp,0x20  9: e8 00 00 00 00 call e <\_main+0xe>  e: c7 44 24 1c 0a 00 00 mov DWORD PTR [esp+0x1c],0xa  15: 00  16: c7 44 24 18 14 00 00 mov DWORD PTR [esp+0x18],0x14  1d: 00  1e: 8b 54 24 1c mov edx,DWORD PTR [esp+0x1c]  22: 8b 44 24 18 mov eax,DWORD PTR [esp+0x18]  26: 01 d0 add eax,edx  28: 89 44 24 14 mov DWORD PTR [esp+0x14],eax  2c: 8b 44 24 14 mov eax,DWORD PTR [esp+0x14]  30: 89 44 24 04 mov DWORD PTR [esp+0x4],eax  34: c7 04 24 00 00 00 00 mov DWORD PTR [esp],0x0  3b: e8 00 00 00 00 call 40 <\_main+0x40>  40: b8 00 00 00 00 mov eax,0x0  45: c9 leave  46: c3 ret  47: 90 nop |

Here size of each line different 🡪 CISC

Add Linker/ Loader: 

Size of Hellowrold2.dump is greater than helloworld.dump as library functions are linked here.

## \*\*\* All Steps Of Compiling C Program \*\*\*

|  |
| --- |
| Makefile |
| main:      gcc -E helloWorld.c > helloWorld.i      gcc -S helloWorld.i      gcc -S -masm=intel helloWorld.i      as -o helloWorld.o helloWorld.s      objdump -M intel -d helloWorld.o > helloWorld.dump      gcc -o helloWorld.exe helloWorld.o      helloWorld.exe      objdump -M intel -d helloWorld.exe > helloWorld.txt |

### Explanation

1. **Target**:
   * **main:** is the target in this Makefile. When you run **make main**, all the commands under this target will be executed sequentially.
2. **Preprocessing**:
   * **gcc -E helloWorld.c > helloWorld.i**
     + The **gcc -E** command preprocesses the **helloWorld.c** file. The preprocessing step includes handling directives like **#include** and **#define**. The output is saved in **helloWorld.i**.
3. **Compilation to Assembly (AT&T syntax)**:
   * **gcc -S helloWorld.i**
     + The **gcc -S** command compiles the preprocessed file **helloWorld.i** into assembly language code in AT&T syntax. The output is **helloWorld.s**.
4. **Compilation to Assembly (Intel syntax)**:
   * **gcc -S -masm=intel helloWorld.i**
     + This command is similar to the previous one, but it generates assembly code in Intel syntax instead of AT&T syntax. It will overwrite the **helloWorld.s** file with the Intel syntax version.
5. **Assembling**:
   * **as -o helloWorld.o helloWorld.s**
     + The **as** command assembles the **helloWorld.s** file (which is in assembly language) into an object file **helloWorld.o**.
6. **Disassembling the Object File**:
   * **objdump -M intel -d helloWorld.o > helloWorld.dump**
     + The **objdump** command disassembles the **helloWorld.o** object file. The **-M intel** option specifies that the disassembly output should be in Intel syntax. The output is saved to **helloWorld.dump**.
7. **Linking**:
   * **gcc -o helloWorld.exe helloWorld.o**
     + This command links the object file **helloWorld.o** to create the executable **helloWorld.exe**.
8. **Running the Executable**:
   * **helloWorld.exe**
     + This runs the compiled executable **helloWorld.exe**.
9. **Disassembling the Executable**:
   * **objdump -M intel -d helloWorld.exe > helloWorld.txt**
     + The **objdump** command disassembles the **helloWorld.exe** executable file. The **-M intel** option specifies that the disassembly should be in Intel syntax. The output is saved to **helloWorld.txt**.

### Summary

* The Makefile automates the process of compiling a C program (**helloWorld.c**) into an executable (**helloWorld.exe**).
* It includes steps to preprocess the C code, compile it to assembly, assemble it into an object file, link the object file into an executable, and disassemble both the object file and the executable for inspection.
* The output files at each step allow you to see the intermediate stages: preprocessed code (**helloWorld.i**), assembly code (**helloWorld.s**), object file (**helloWorld.o**), and disassembled output (**helloWorld.dump** and **helloWorld.txt**).

## # Build System:

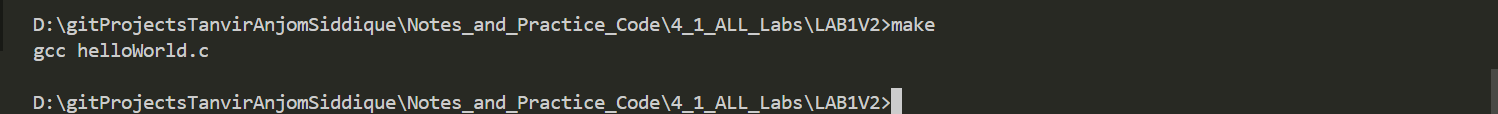
Write 1 command compile many files.

Create a file: (this name is fixed)

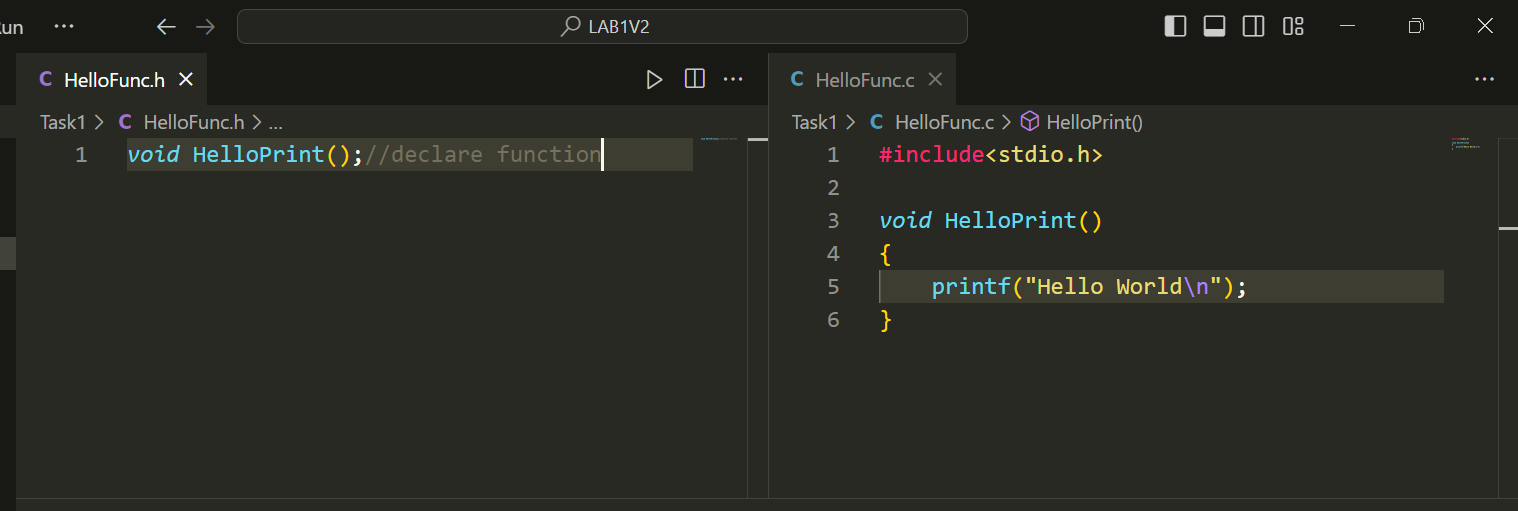
|  |
| --- |
| Makefile |
|  |

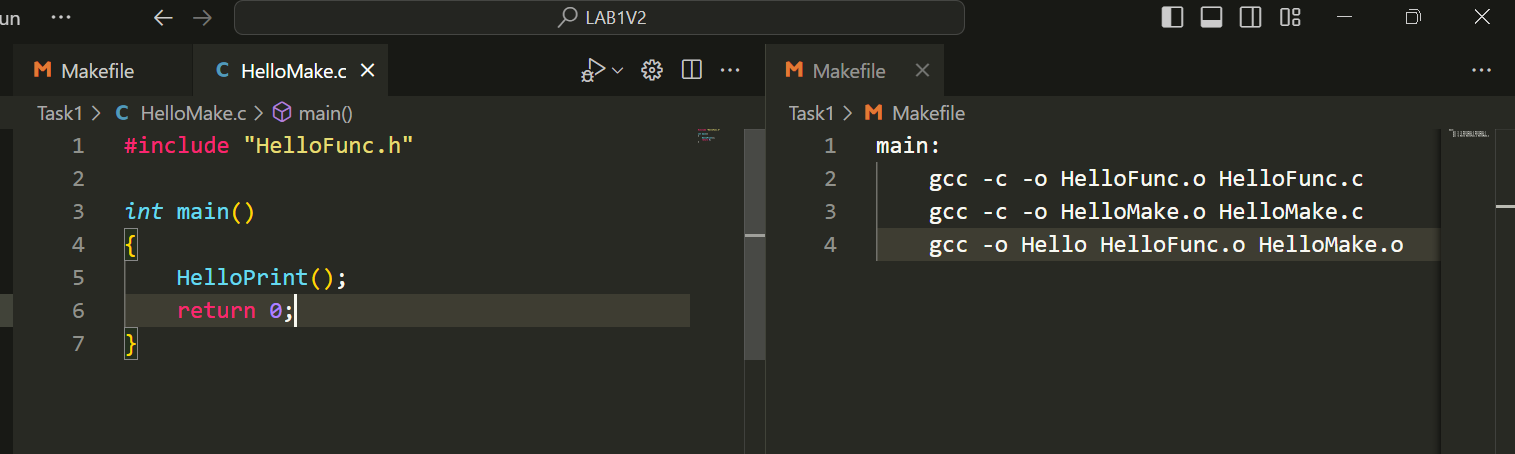
We can execute multiple commands here

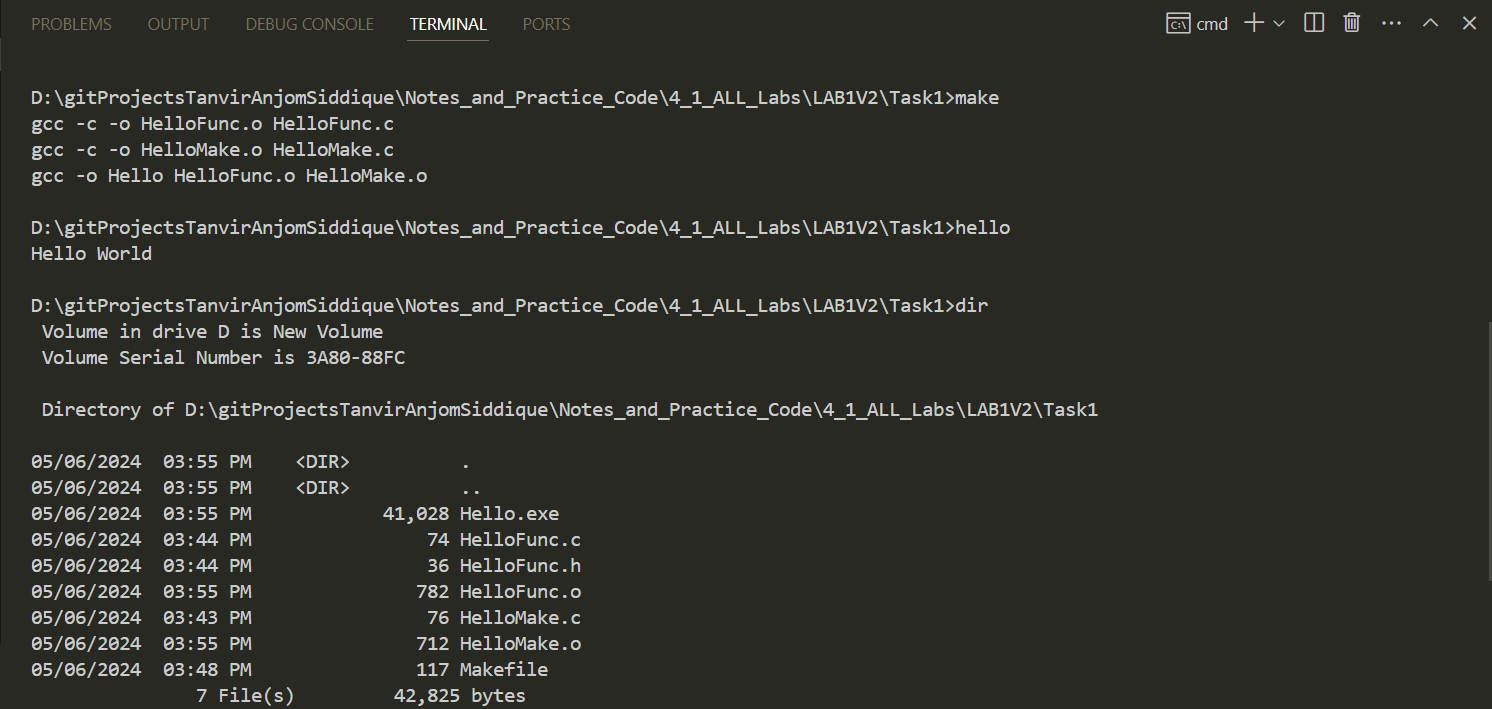
Now in terminal:

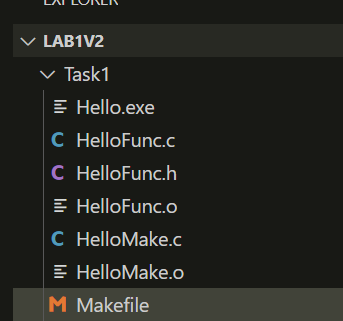


### \*\*Makefile (compile multiple file):



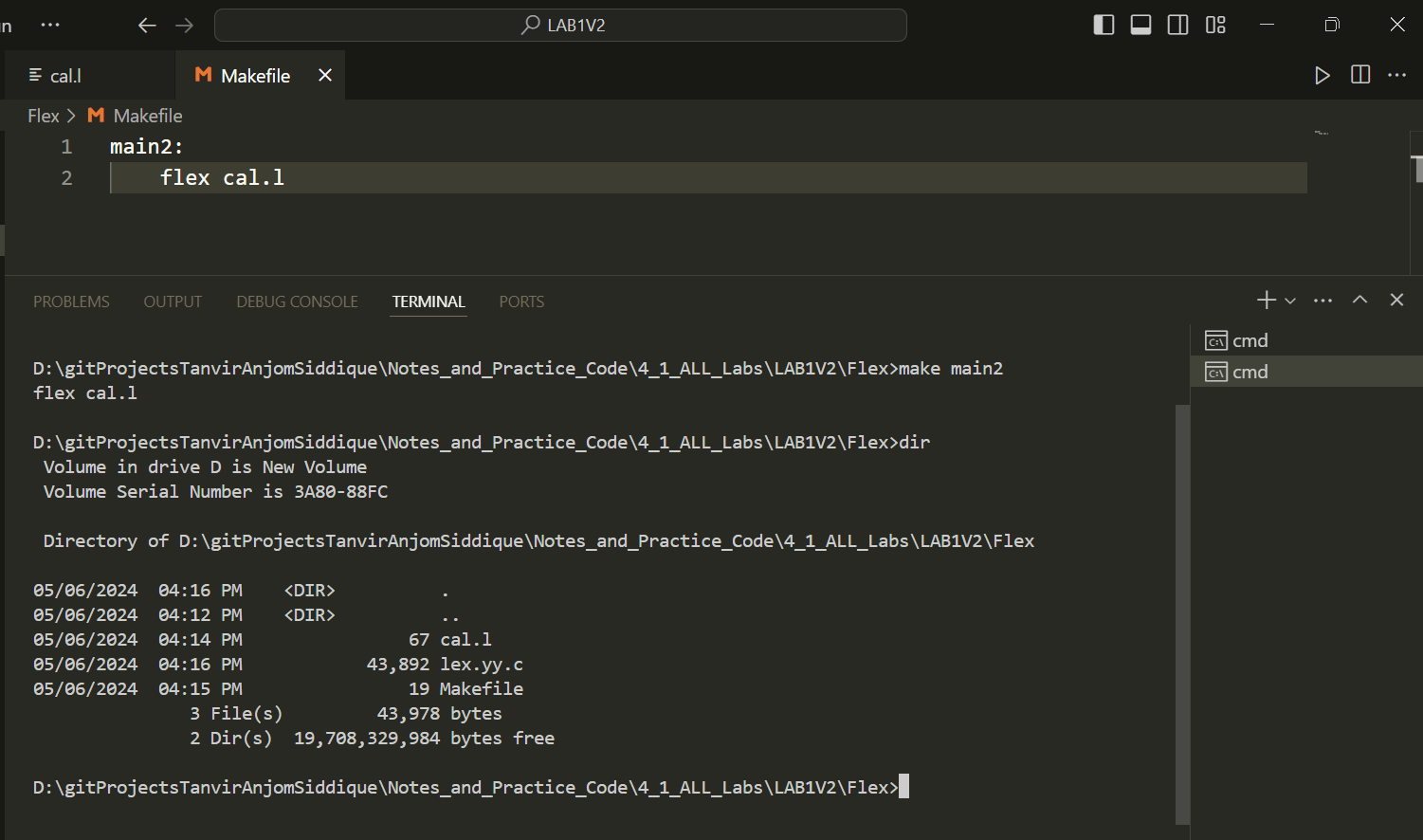


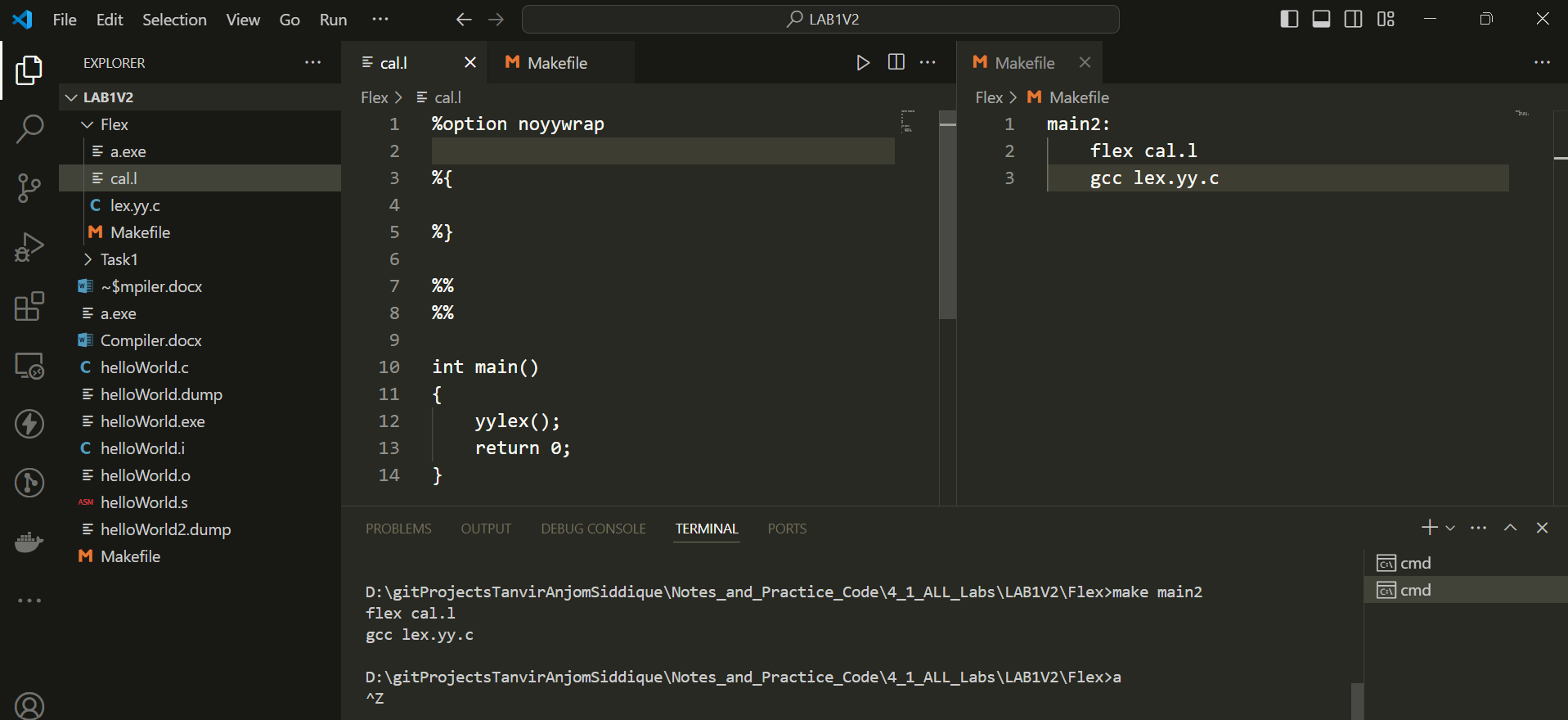




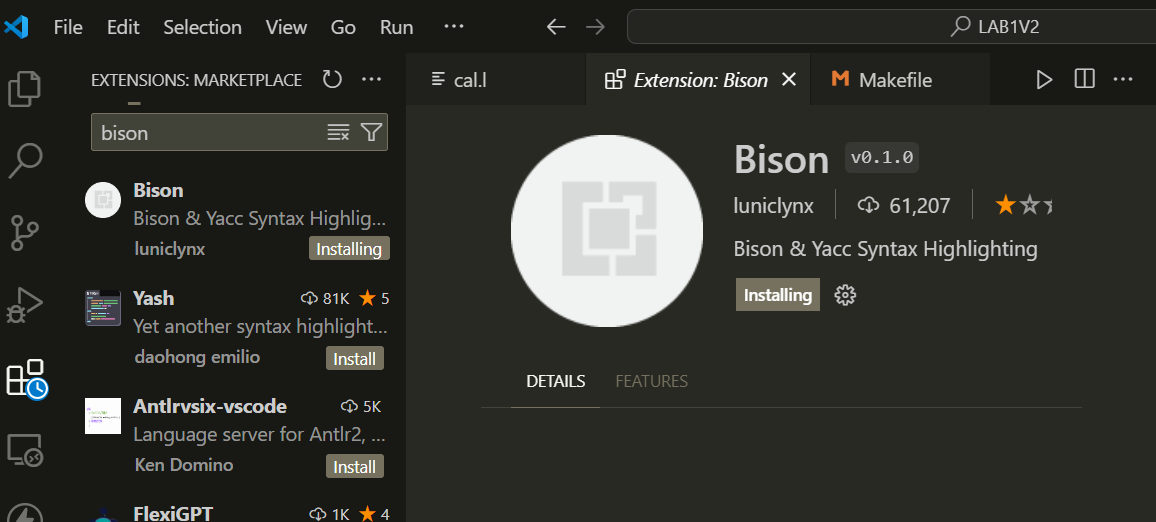
Files in different folders

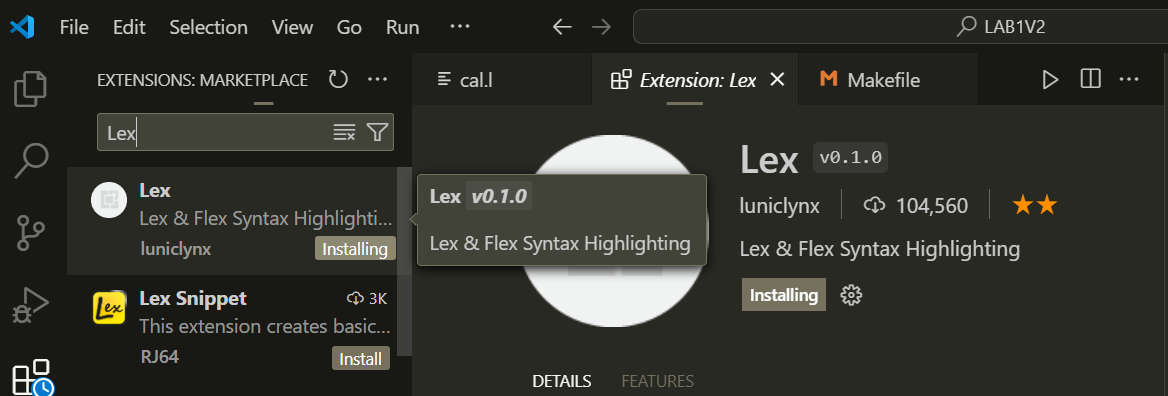
## Flex (lexical analysis): [Identify ID , VAR, STRING, OP]

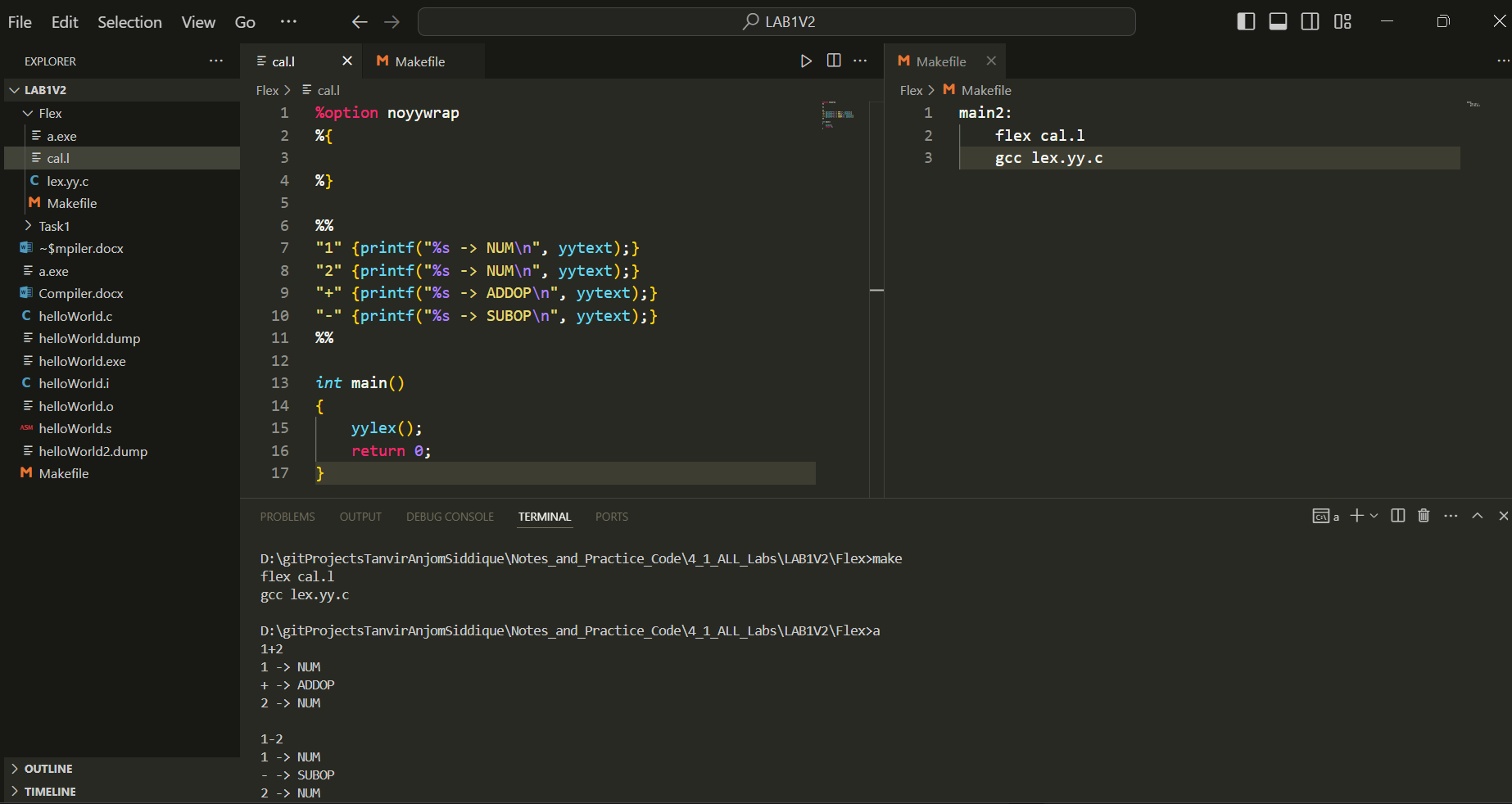


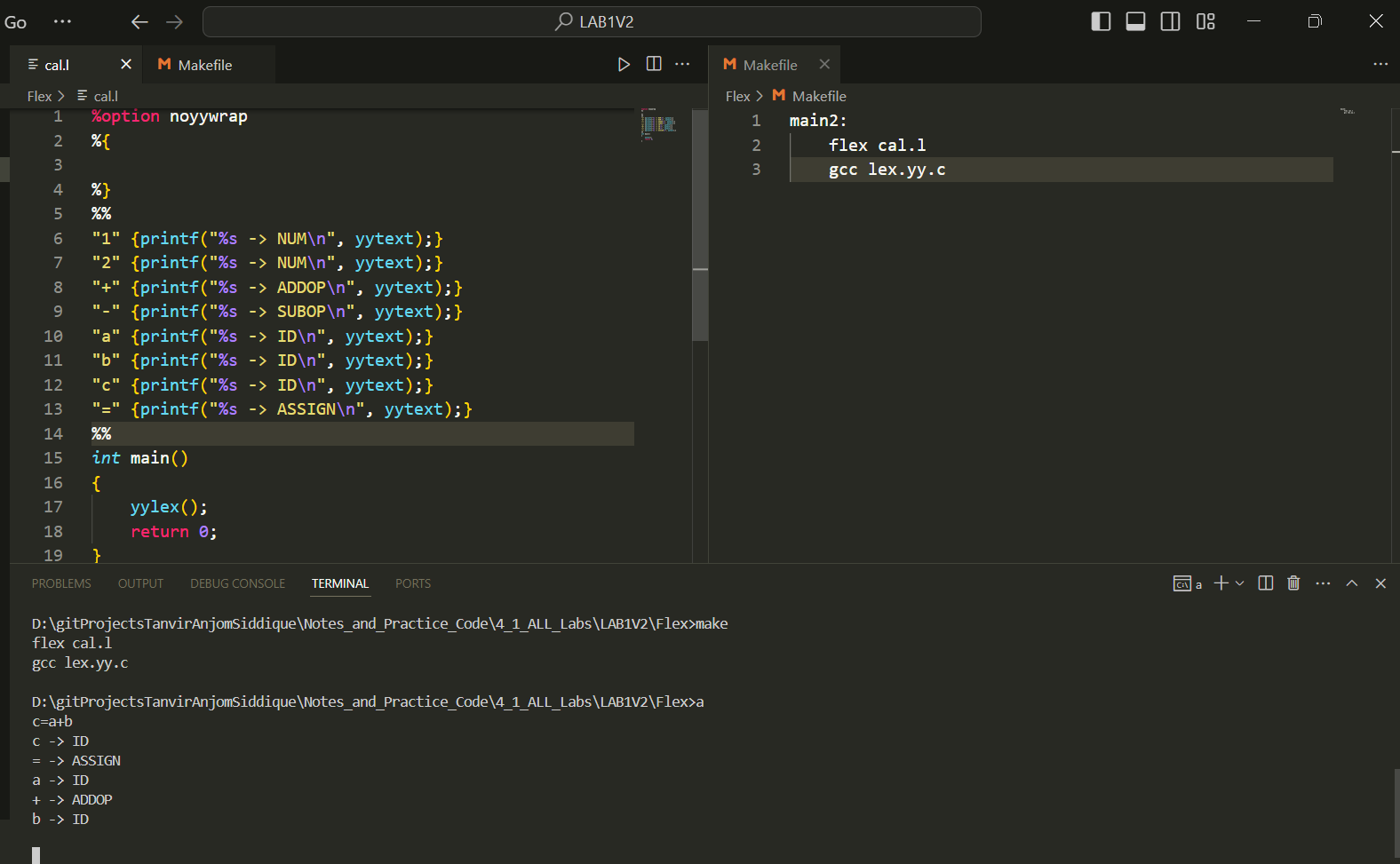


## \*\*\* Highlight cal.l > bison & lex (extension install)



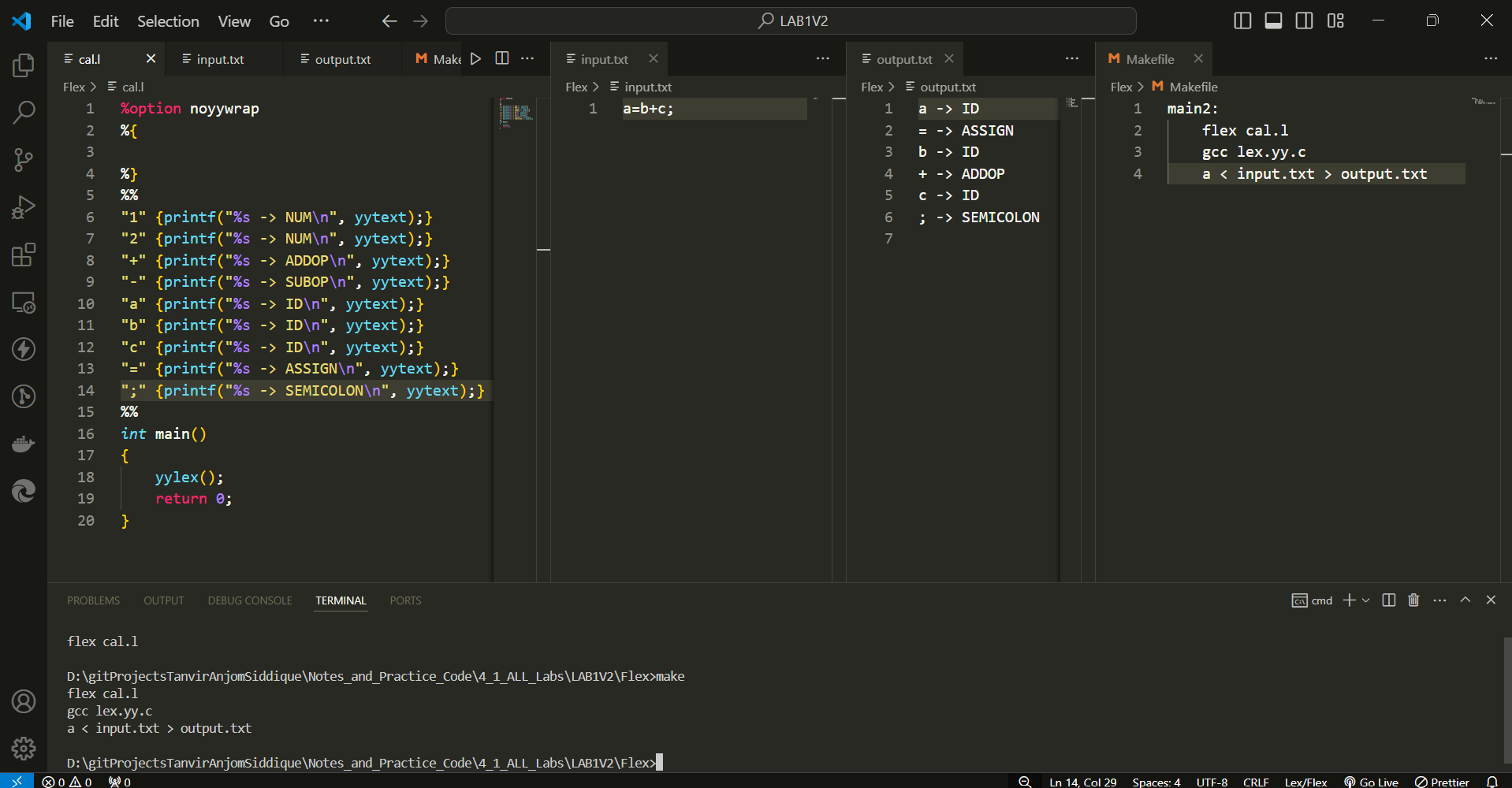






|  |
| --- |
| cal.l |
| %option noyywrap  %{  %}  %%  "1" {printf("%s -> NUM\n", yytext);}  "2" {printf("%s -> NUM\n", yytext);}  "+" {printf("%s -> ADDOP\n", yytext);}  "-" {printf("%s -> SUBOP\n", yytext);}  "a" {printf("%s -> ID\n", yytext);}  "b" {printf("%s -> ID\n", yytext);}  "c" {printf("%s -> ID\n", yytext);}  "=" {printf("%s -> ASSIGN\n", yytext);}  %%  *int* main()  {      yylex();      return 0;  } |

Take input from “input.txt” show ouput in “output.txt”



# Lab 02: Lexical & Syntax Analysis

## Identify Token

cal.l

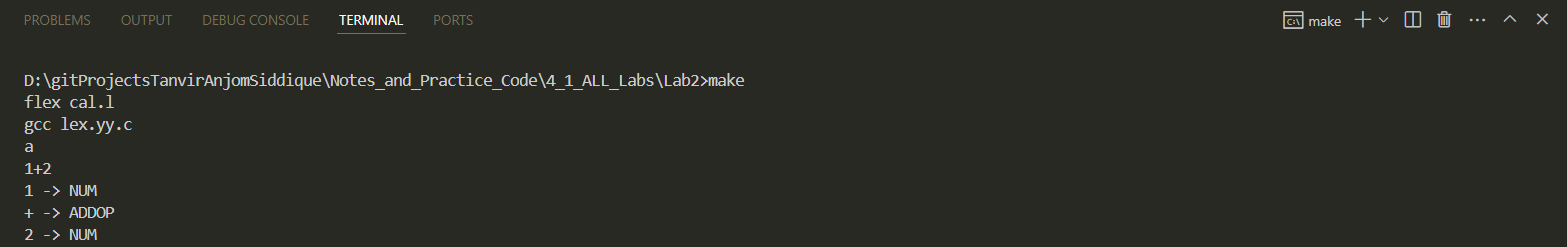
|  |
| --- |
| %option noyywrap  %{  %}  %%  "1" {printf("%s -> NUM\n", yytext);}  "2" {printf("%s -> NUM\n", yytext);}  "+" {printf("%s -> ADDOP\n", yytext);}  "-" {printf("%s -> SUBOP\n", yytext);}  "a" {printf("%s -> ID\n", yytext);}  "b" {printf("%s -> ID\n", yytext);}  "c" {printf("%s -> ID\n", yytext);}  "=" {printf("%s -> ASSIGN\n", yytext);}  ";" {printf("%s -> SEMICOLON\n", yytext);}  %%  *int* main()  {      yylex();      return 0;  } |

Makefile

|  |
| --- |
| main:      flex cal.l      gcc lex.yy.c      a |

|  |
| --- |
| D:\gitProjectsTanvirAnjomSiddique\Notes\_and\_Practice\_Code\4\_1\_ALL\_Labs\Lab2>make  flex cal.l  gcc lex.yy.c  a  1+2  1 -> NUM  + -> ADDOP  2 -> NUM |

lex.yy.c will be created then, we will compile it with gcc and run it a.exe



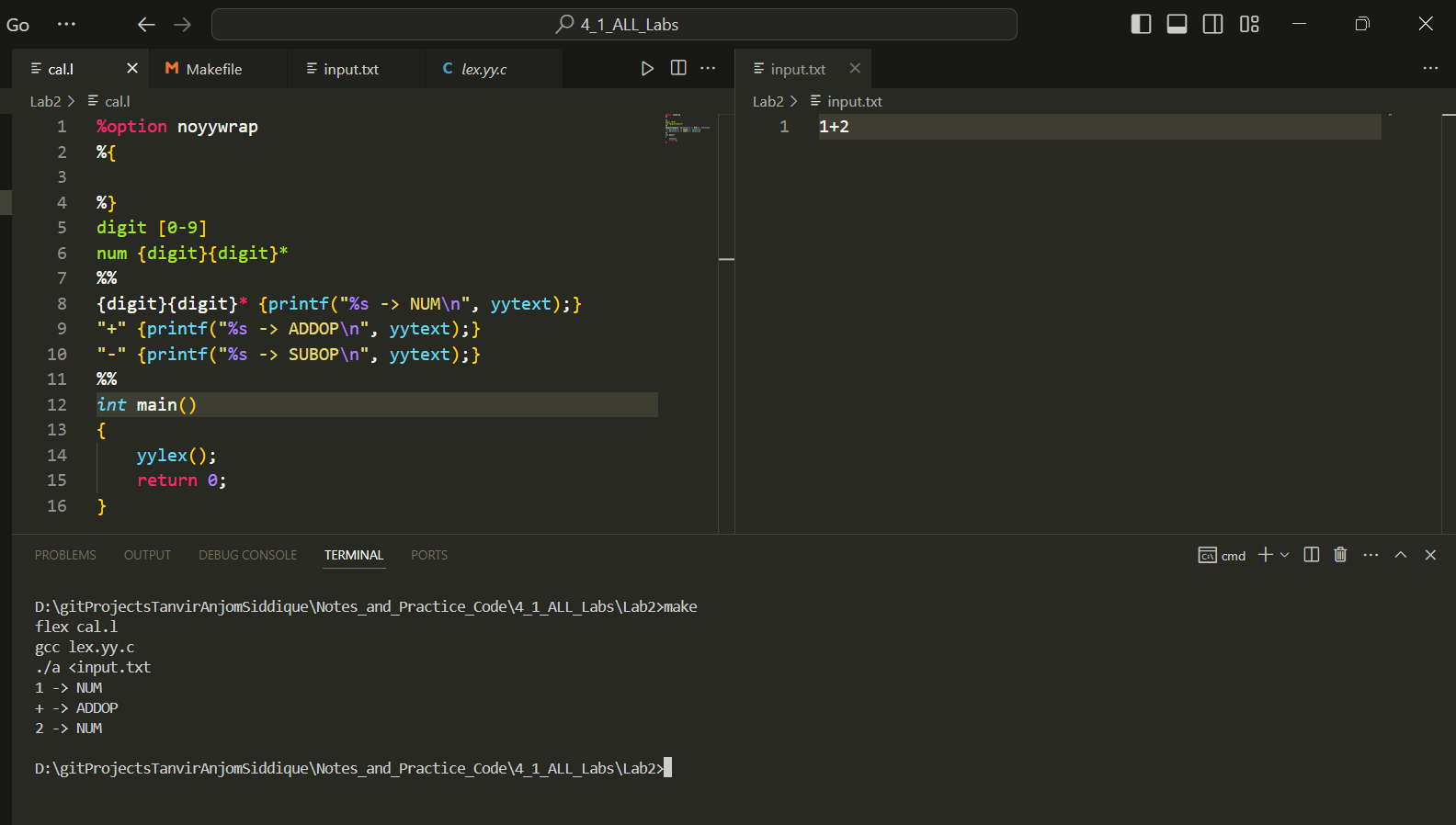
### Read input from a file

|  |
| --- |
| Makefile |
| main:      flex cal.l      gcc lex.yy.c      ./a <input.txt |

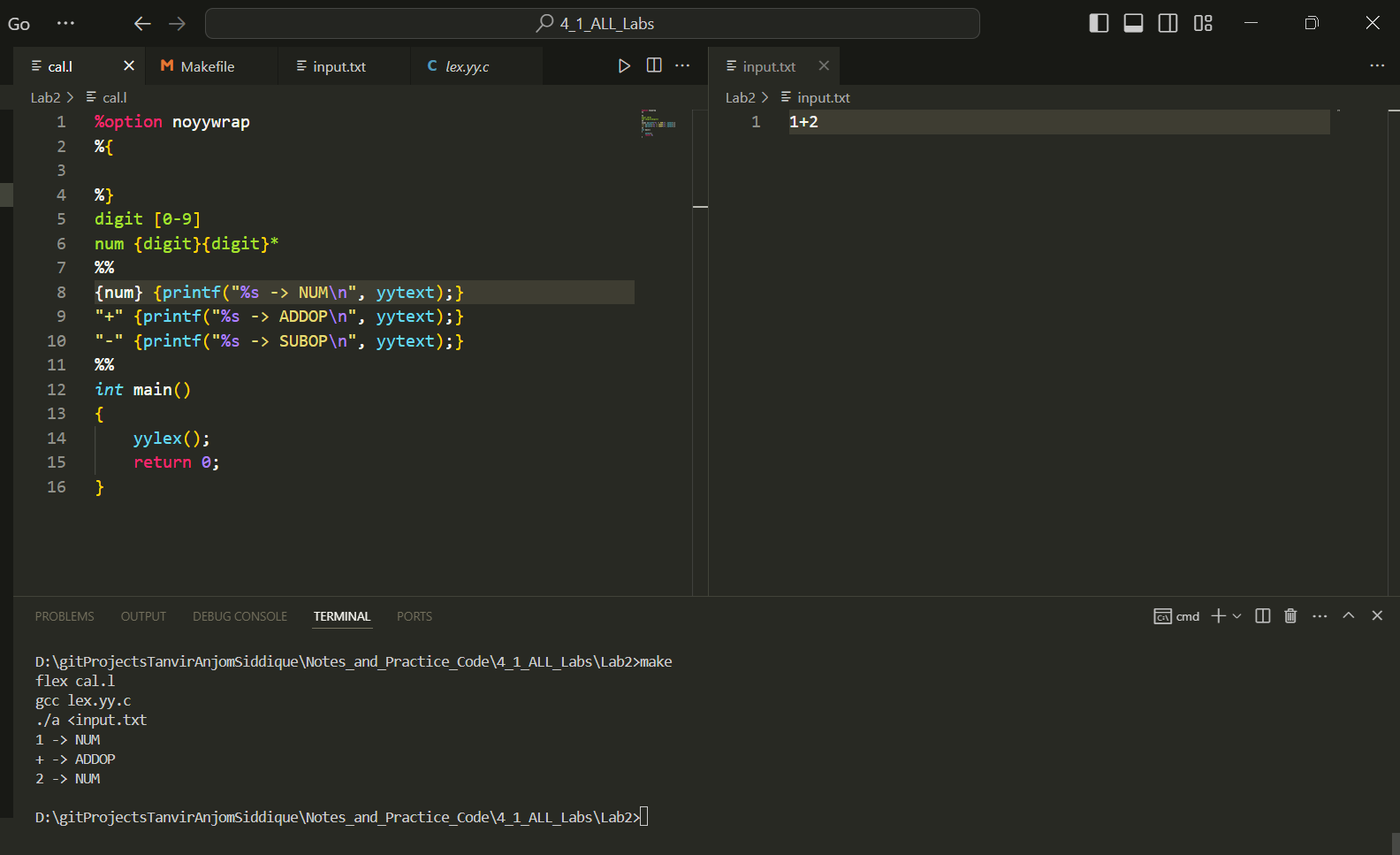
|  |
| --- |
| D:\gitProjectsTanvirAnjomSiddique\Notes\_and\_Practice\_Code\4\_1\_ALL\_Labs\Lab2>make  flex cal.l  gcc lex.yy.c  ./a <input.txt  1 -> NUM  + -> ADDOP  2 -> NUM |

### Use RE rules:

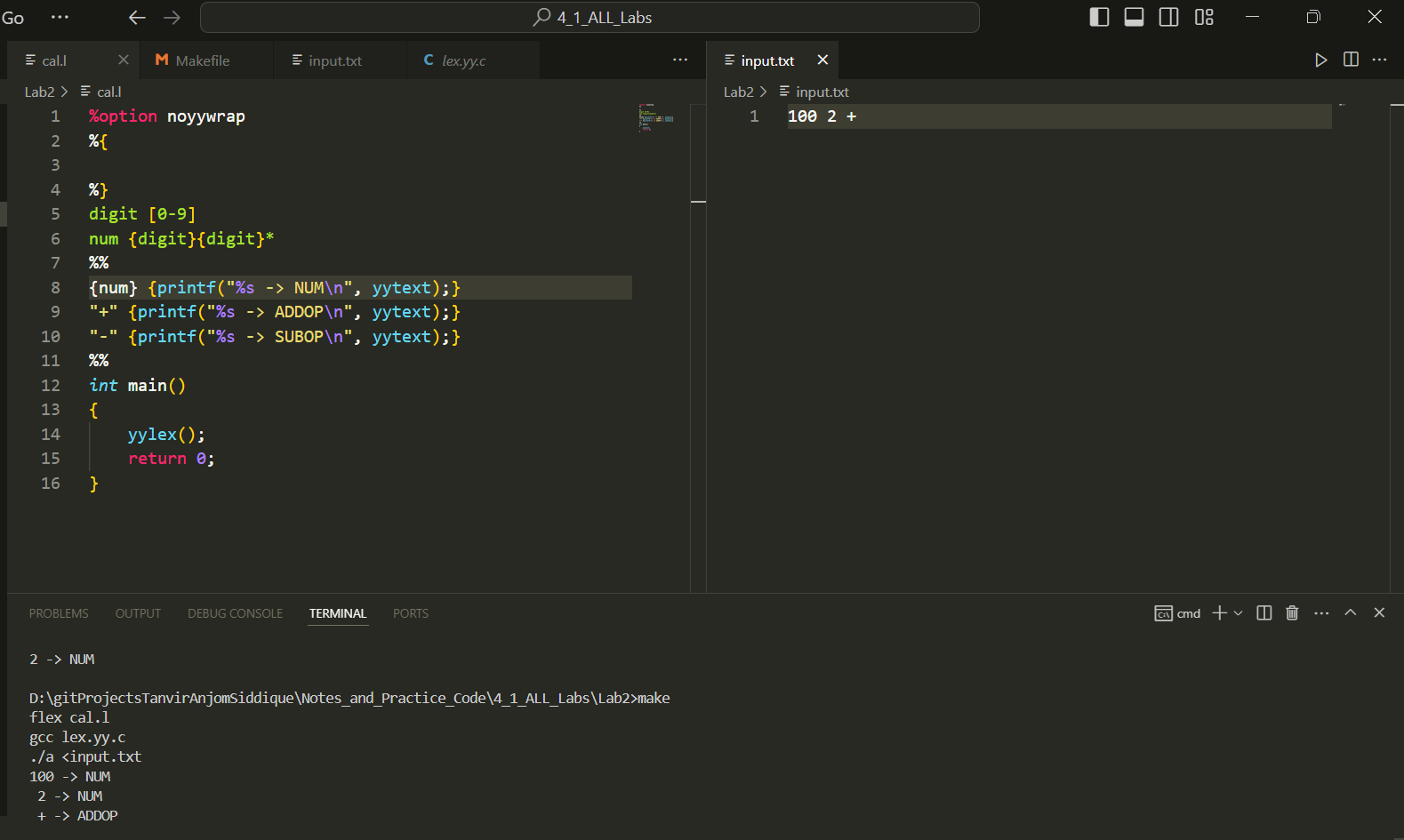
|  |
| --- |
| cal.l |
| %option noyywrap  %{  %}  digit [0-9]  num {digit}{digit}\*  %%  {digit}{digit}\* {printf("%s -> NUM\n", yytext);}  "+" {printf("%s -> ADDOP\n", yytext);}  "-" {printf("%s -> SUBOP\n", yytext);}  %%  *int* main()  {      yylex();      return 0;  } |



Or,

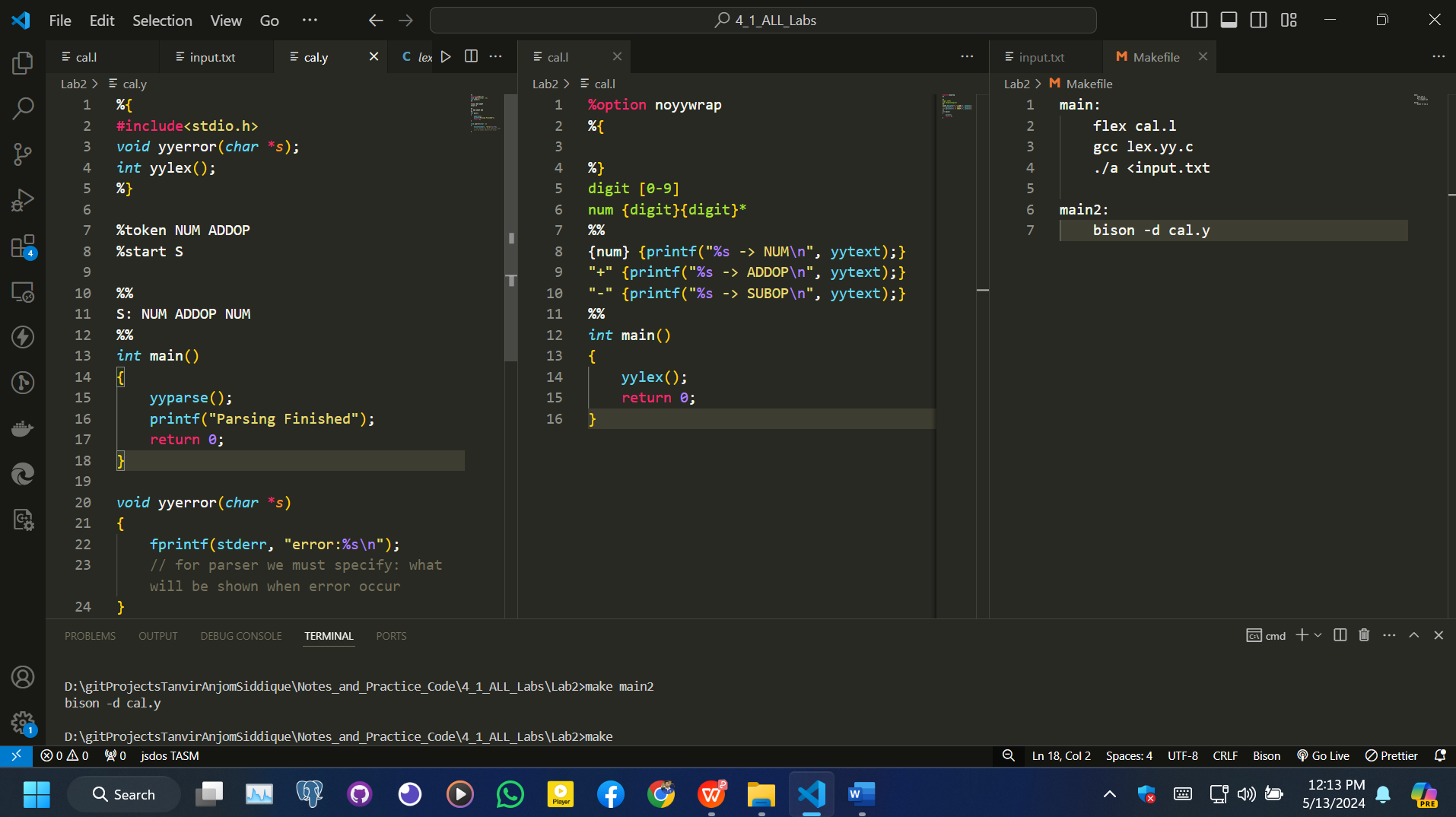


### This don’t check syntax but token only



## BISON:

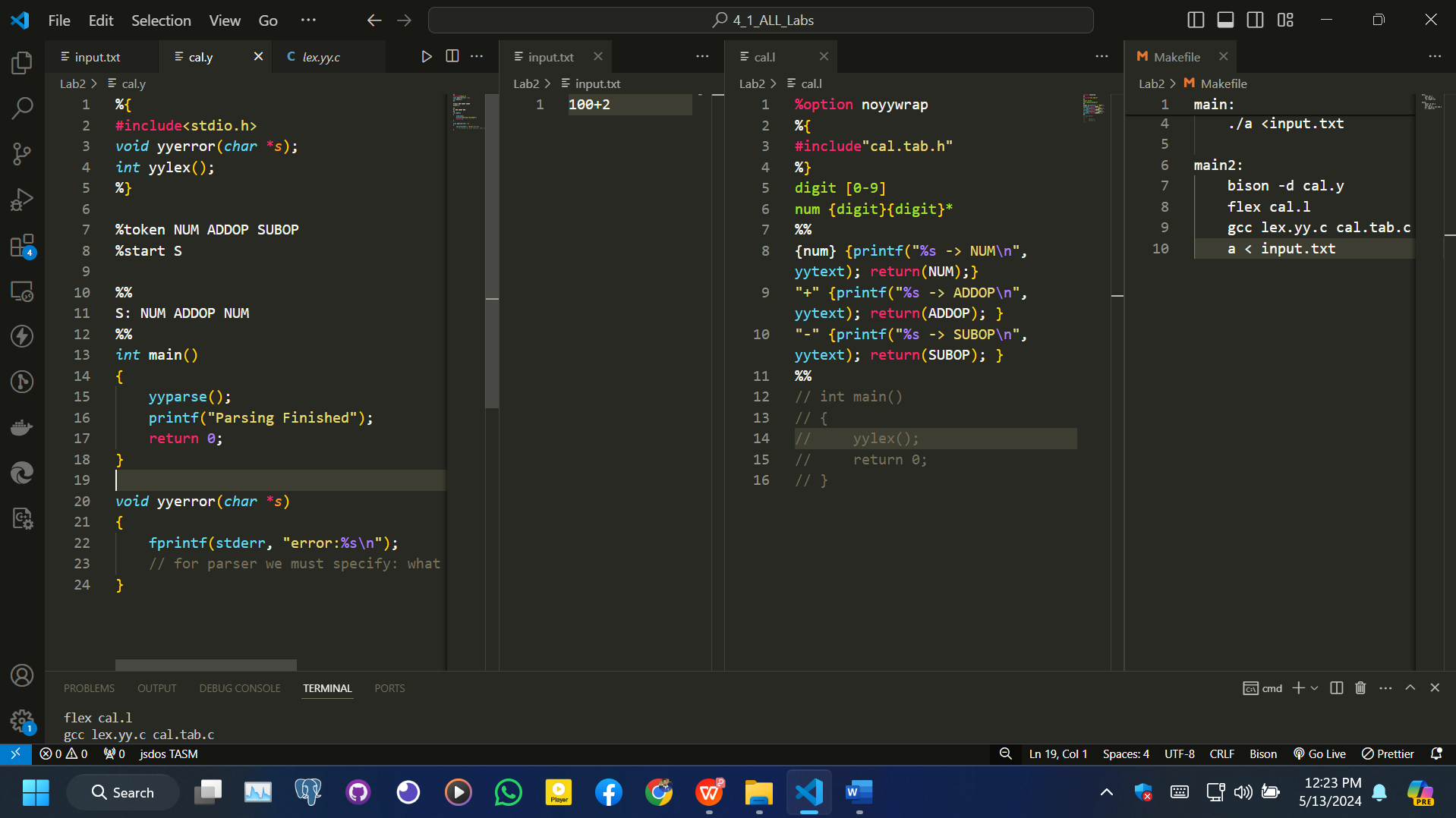
|  |
| --- |
| cal.y |
| %{  #include<stdio.h>  *void* yyerror(*char* \**s*);  *int* yylex();  %}  %token NUM ADDOP  %start S  %%  S: NUM ADDOP NUM  %%  *int* main()  {      yyparse();      printf("Parsing Finished");      return 0;  }  *void* yyerror(*char* \**s*)  {      fprintf(stderr, "error:%s\n");      // for parser we must specify: what will be shown when error occur  } |



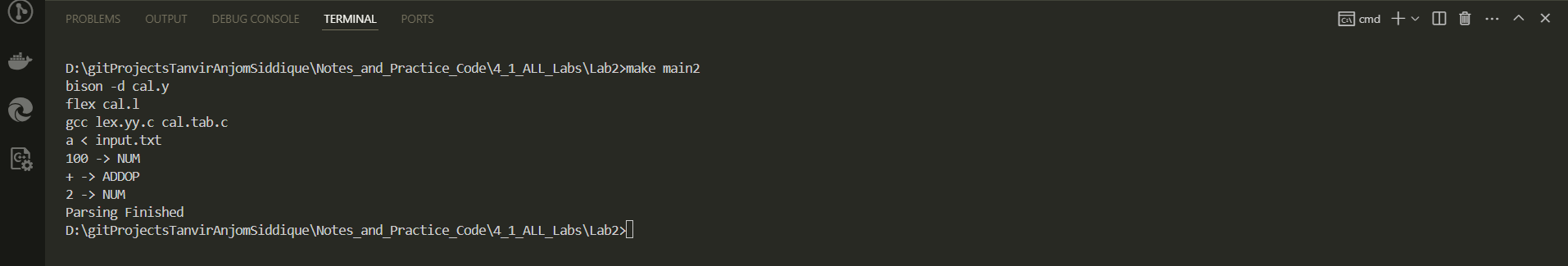
cal.tab.c

and

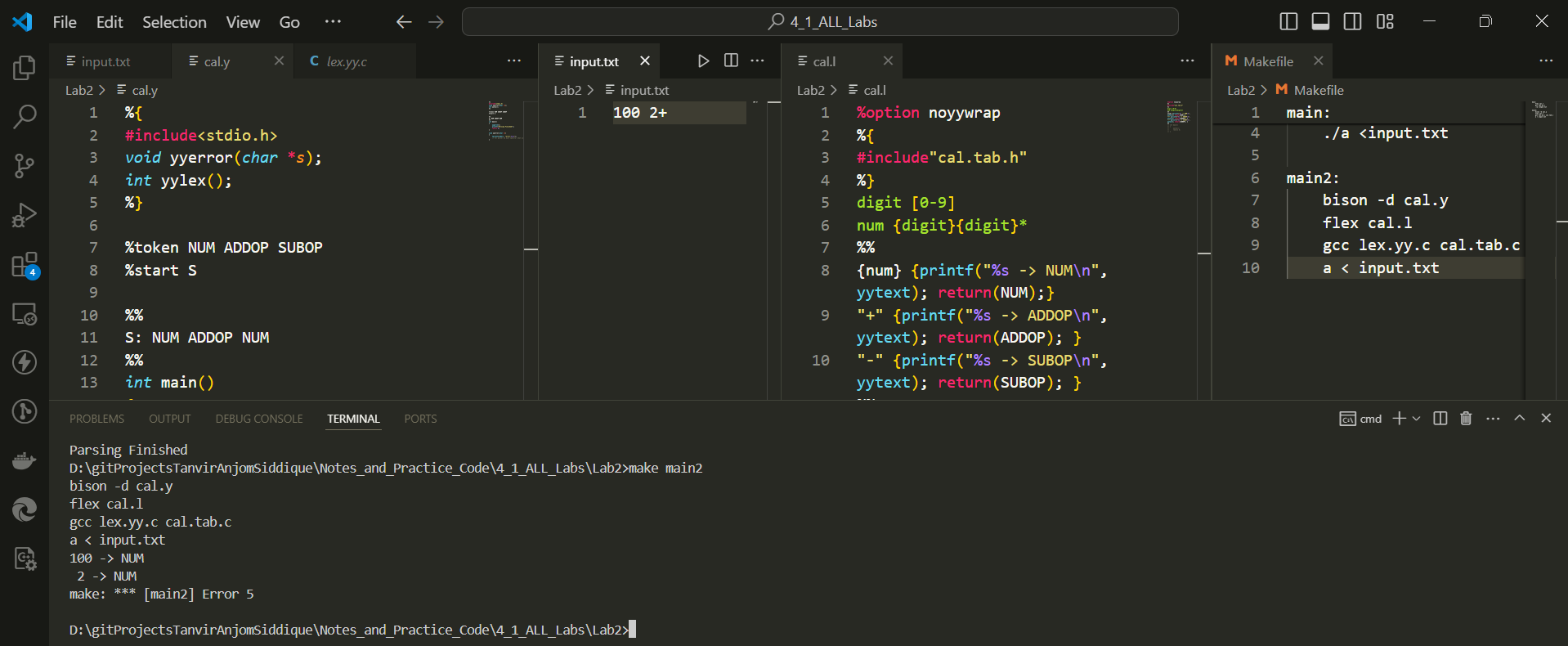
cal.tab.h will be created



|  |
| --- |
| Terminal:  make main2 |



If there is error in input’s syntax in file:



### Parse Program line: int a=10;

Copy cal.l to prog1.l

Copy cal.y to prog1.y

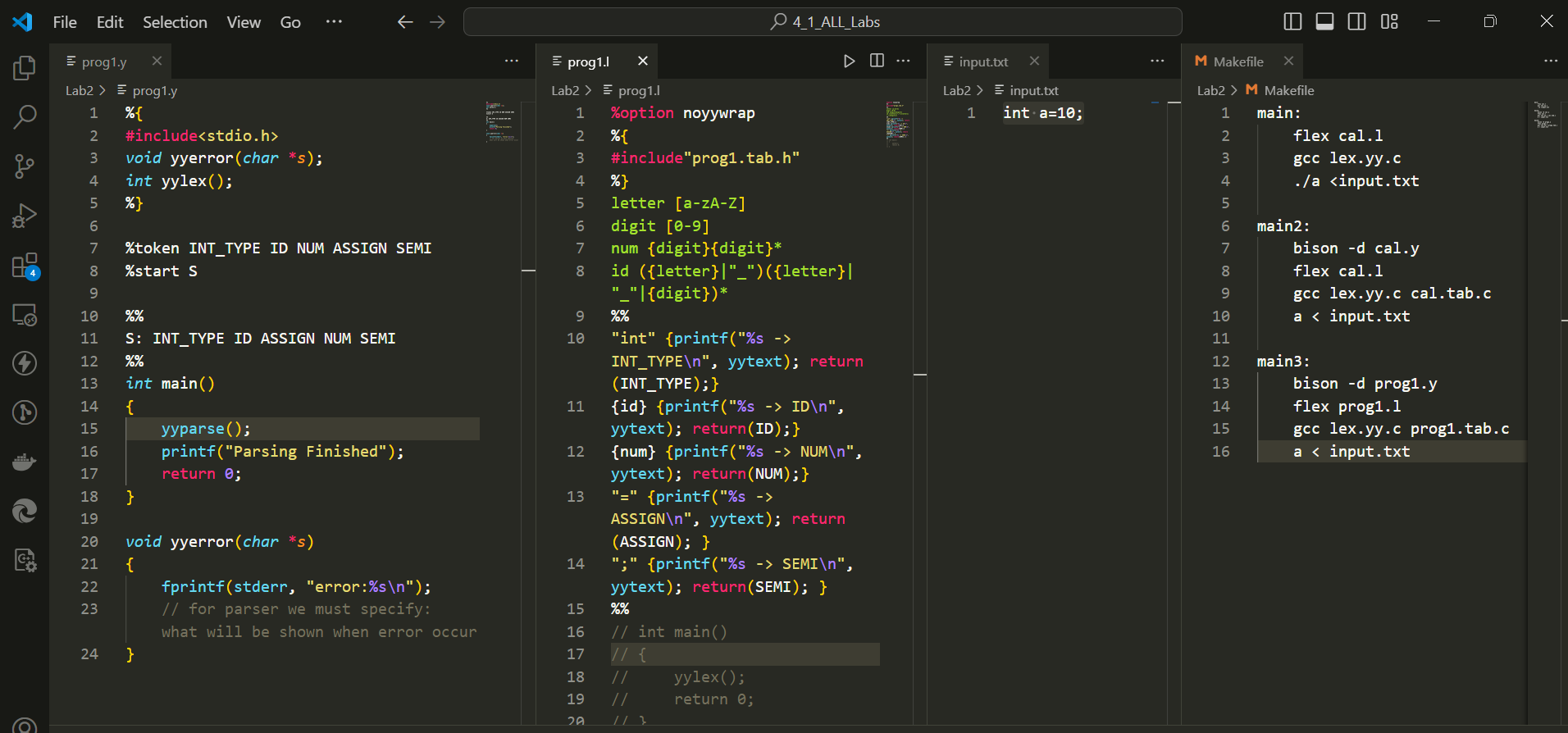
|  |
| --- |
| prog1.l |
| %option noyywrap  %{  #include"prog1.tab.h"  %}  letter [a-zA-Z]  digit [0-9]  num {digit}{digit}\*  id ({letter}|"\_")({letter}|"\_"|{digit})\*  %%  "int" {printf("%s -> INT\_TYPE\n", yytext); return(INT\_TYPE);}  {id} {printf("%s -> ID\n", yytext); return(ID);}  {num} {printf("%s -> NUM\n", yytext); return(NUM);}  "=" {printf("%s -> ASSIGN\n", yytext); return(ASSIGN); }  ";" {printf("%s -> SEMI\n", yytext); return(SEMI); }  %%  // int main()  // {  //     yylex();  //     return 0;  // } |

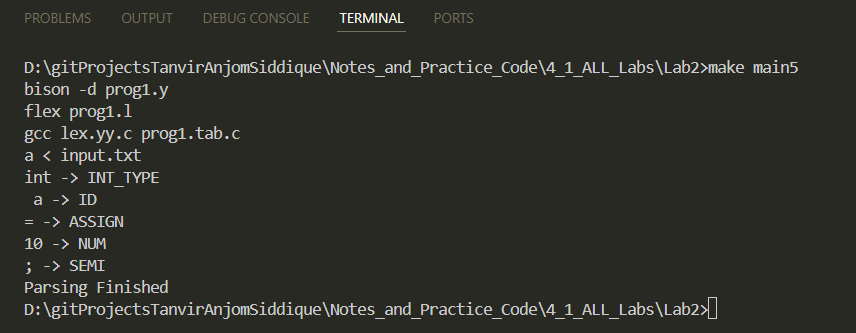
|  |
| --- |
| prog1.y |
| %{  #include<stdio.h>  *void* yyerror(*char* \**s*);  *int* yylex();  %}  %token INT\_TYPE ID NUM ASSIGN SEMI  %start S  %%  S: INT\_TYPE ID ASSIGN NUM SEMI  %%  *int* main()  {      yyparse();      printf("Parsing Finished");      return 0;  }  *void* yyerror(*char* \**s*)  {      fprintf(stderr, "error:%s\n");      // for parser we must specify: what will be shown when error occur  } |

|  |
| --- |
| input.txt |
| int a=10; |

|  |
| --- |
| Makefile |
| main:      flex cal.l      gcc lex.yy.c      ./a <input.txt  main2:      bison -d cal.y      flex cal.l      gcc lex.yy.c cal.tab.c      a < input.txt  main3:      bison -d prog1.y      flex prog1.l      gcc lex.yy.c prog1.tab.c      a < input.txt |

|  |
| --- |
| Terminal |
| D:\gitProjectsTanvirAnjomSiddique\Notes\_and\_Practice\_Code\4\_1\_ALL\_Labs\Lab2>make main5  bison -d prog1.y  flex prog1.l  gcc lex.yy.c prog1.tab.c  a < input.txt  int -> INT\_TYPE  a -> ID  = -> ASSIGN  10 -> NUM  ; -> SEMI  Parsing Finished |





### Parse While loop:

#### Parser: prog1.y 🡪 prog1.tab.c , prog1.tab.h

|  |
| --- |
| prog1.y |
| %{  #include<stdio.h>  *void* yyerror(*char* \**s*);  *int* yylex();  %}  %token INT\_TYPE ID NUM ASSIGN SEMI WHILE LP LT RP LB INCREMENT ADDOP RB  %start S  %%  S: declr while\_loop;  declr: INT\_TYPE ID ASSIGN NUM SEMI  while\_loop: WHILE LP ID LT NUM RP LB ID INCREMENT SEMI RB  %%  *int* main()  {      yyparse();      printf("Parsing Finished");      return 0;  }  *void* yyerror(*char* \**s*)  {      fprintf(stderr, "error:%s\n");      // for parser we must specify: what will be shown when error occur  } |

#### Lexical Analyzer: prog1.l

|  |
| --- |
| prog1.l |
| %option noyywrap  %{  #include"prog1.tab.h"  %}  letter [a-zA-Z]  digit [0-9]  num {digit}{digit}\*  id ({letter}|"\_")({letter}|"\_"|{digit})\*  delim [ \n]  %%  {delim} {}  "int" {printf("%s -> INT\_TYPE\n", yytext); return(INT\_TYPE);}  "while" {printf("%s -> WHILE\n", yytext); return(WHILE); }  {id} {printf("%s -> ID\n", yytext); yylval=atoi(yytext); return(ID);}  {num} {printf("%s -> NUM\n", yytext); return(NUM);}  "=" {printf("%s -> ASSIGN\n", yytext); return(ASSIGN); }  ";" {printf("%s -> SEMI\n", yytext); return(SEMI); }  "(" {printf("%s -> LP\n", yytext); return(LP); }  "<" {printf("%s -> LT\n", yytext); return(LT); }  ")" {printf("%s -> RP\n", yytext); return(RP); }  "{" {printf("%s -> LB\n", yytext); return(LB); }  "++" {printf("%s -> INCREMENT\n", yytext); return(INCREMENT); }  "+" {printf("%s -> ADDOP\n", yytext); return(ADDOP); }  "}" {printf("%s -> RB\n", yytext); return(RB); }  %%  // int main()  // {  //     // while and int --> match with id --> so write their rule before id  //     // {delim} {}  --> space or \n (newline)  no operation {}  //     yylex();  //     return 0;  // } |

#### Input: input.txt

|  |
| --- |
| input.txt |
| int i=0;  while ( i<5 ){  i++;  } |

#### Makefile

|  |
| --- |
| Makefile |
| main:      flex cal.l      gcc lex.yy.c      ./a <input.txt  main2:      bison -d cal.y      flex cal.l      gcc lex.yy.c cal.tab.c      a < input.txt  main3:      bison -d prog1.y      flex prog1.l      gcc lex.yy.c prog1.tab.c      a < input.txt |

|  |
| --- |
| Terminal: |
|  |

## \*\*\* Parser (cal.y ->cal.tab.c, cal.tab.h ) , Lexical Analyzer (cal.l -> lex.yy.c) all commands Makefile\*\*\*

|  |
| --- |
| Makefile |
| main3:      bison -d prog1.y      flex prog1.l      gcc lex.yy.c prog1.tab.c      a < input.txt |

### **Generate Parser**:

* + **bison -d prog1.y**
    - This command runs **bison** on the **prog1.y** file, generating **prog1.tab.c** and **prog1.tab.h**.

### **Generate Lexical Analyzer**:

* + **flex prog1.l**
    - This command runs **flex** on the **prog1.l** file, generating **lex.yy.c**.

### **Compile Lexer and Parser**:

* + **gcc lex.yy.c prog1.tab.c**
    - This compiles both the lexical analyzer (**lex.yy.c**) and the parser (**prog1.tab.c**) using **gcc**, producing an executable (default name **a.out** or **a**).

### **Run the Executable**:

* + **a < input.txt**
    - This runs the compiled program with **input.txt** as its input.

### \*\*\* Parse & Lexically Analyze “While loop” in short: \*\*\*

#### Parser code: prog1.y

|  |
| --- |
| %{  #include<stdio.h>  *void* yyerror(*char* \**s*);  *int* yylex();  %}  %token INT\_TYPE ID NUM ASSIGN SEMI WHILE LP LT RP LB INCREMENT ADDOP RB  %start S  %%  S: declr while\_loop;  declr: INT\_TYPE ID ASSIGN NUM SEMI  while\_loop: WHILE LP ID LT NUM RP LB ID INCREMENT SEMI RB  %%  *int* main()  {      yyparse();      printf("Parsing Finished");      return 0;  }  *void* yyerror(*char* \**s*)  {      fprintf(stderr, "error:%s\n");      // for parser we must specify: what will be shown when error occur  } |

|  |
| --- |
| Terminal >      bison -d prog1.y |

#### Lexical Analyzer Code: prog1.l

|  |
| --- |
| %option noyywrap  %{  #include"prog1.tab.h"  %}  letter [a-zA-Z]  digit [0-9]  num {digit}{digit}\*  id ({letter}|"\_")({letter}|"\_"|{digit})\*  delim [ \n]  %%  {delim} {}  "int" {printf("%s -> INT\_TYPE\n", yytext); return(INT\_TYPE);}  "while" {printf("%s -> WHILE\n", yytext); return(WHILE); }  {id} {printf("%s -> ID\n", yytext); yylval=atoi(yytext); return(ID);}  {num} {printf("%s -> NUM\n", yytext); return(NUM);}  "=" {printf("%s -> ASSIGN\n", yytext); return(ASSIGN); }  ";" {printf("%s -> SEMI\n", yytext); return(SEMI); }  "(" {printf("%s -> LP\n", yytext); return(LP); }  "<" {printf("%s -> LT\n", yytext); return(LT); }  ")" {printf("%s -> RP\n", yytext); return(RP); }  "{" {printf("%s -> LB\n", yytext); return(LB); }  "++" {printf("%s -> INCREMENT\n", yytext); return(INCREMENT); }  "+" {printf("%s -> ADDOP\n", yytext); return(ADDOP); }  "}" {printf("%s -> RB\n", yytext); return(RB); }  %%  // int main()  // {  //     // while and int --> match with id --> so write their rule before id  //     // {delim} {}  --> space or \n (newline)  no operation {}  //     yylex();  //     return 0;  // } |

|  |
| --- |
| Terminal>      flex prog1.l      gcc lex.yy.c prog1.tab.c      a < input.txt |