Lab sheet 1

1. Understanding C Language Processing System – Behind the Scenes
   a. Create a file add.c with the following code

```
//Program to add two numbers
#include<stdio.h>
#define add(a,b) (a+b)//using macro
int main() {
      int a=5 , b  = 4;
      printf("Sum = %d", add(a,b));
      return 0;
}
```

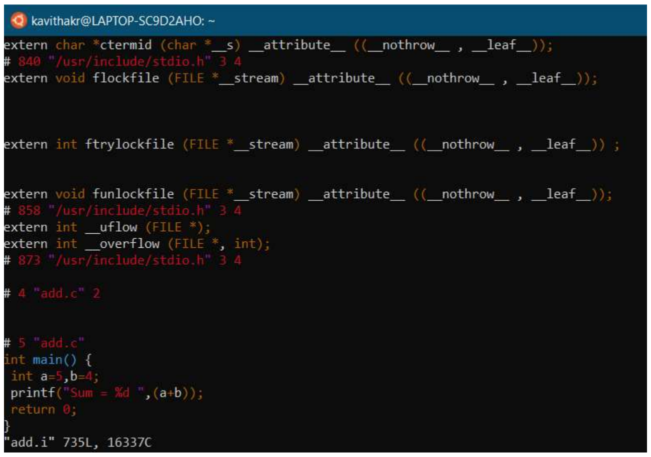
   b. Compile using below code. We get the all intermediate files in the current directory along with the executable.

      $ gcc –Wall –save-temps add.c –o add

   c. **Pre-processing**

      This is the first phase through which source code is passed. This phase include: Removal of Comments, Expansion of Macros, Expansion of the included files. Conditional compilation. The pre-processed output is stored in the **add.i**.

      $vi **add.i**



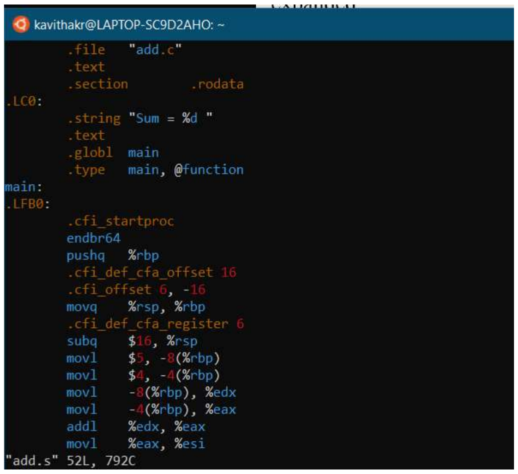
```
kavithakr@LAPTOP-SC9D2AHO: ~
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 840 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));


extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 858 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 873 "/usr/include/stdio.h" 3 4

# 4 "add.c" 2


# 5 "add.c"
int main() {
 int a=5,b=4;
 printf("Sum = %d ",(a+b));
 return 0;
}
"add.i" 735L, 16337C
```

Analysis:
- printf contains now a + b rather than add(a, b) that's because macros have expanded.
- Comments are stripped off.
- #include<stdio.h> ismissing instead we see lots of code. So header files has been expanded and included in our source file.

   d. **Compiling**

      The next step is to compile filename.i and produce an intermediate compiled output file **filename.s**.

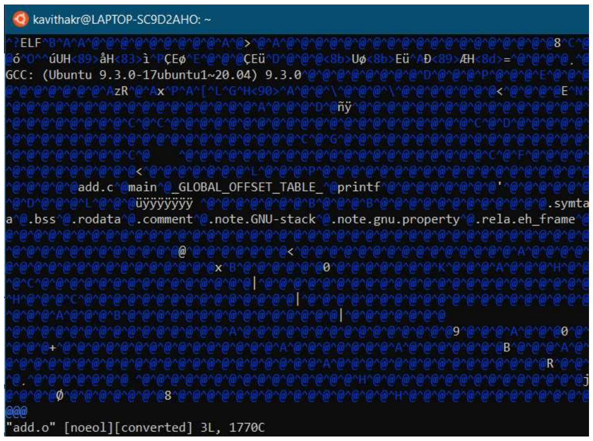
      $vi add.s



```
kavithakr@LAPTOP-SC9D2AHO: ~
        .file   "add.c"
        .text
        .section        .rodata
.LC0:
        .string "Sum = %d "
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $5, -8(%rbp)
        movl    $4, -4(%rbp)
        movl    -8(%rbp), %edx
        movl    -4(%rbp), %eax
        addl    %edx, %eax
        movl    %eax, %esi
"add.s" 52L, 792C
```

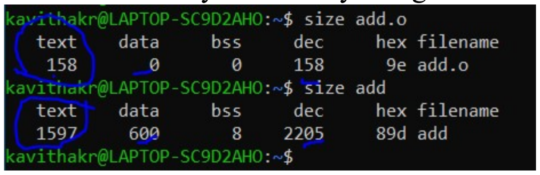
The snapshot shows that it is in assembly language, which assembler can understand

e. **Assembly**
The filename.s is taken as input and turned into **filename.o** by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language,
$vi add.o



```
kavithakr@LAPTOP-SC9D2AHO: ~
^?ELF^B^A^A^@^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@8^C^@
@ó^O^^úUH<89>åH<83>ì^PÇEø^E^@^@^@ÇEü^D^@^@^@<8b>Uø<8b>Eü^AÐ<89>ÆH<8d>=^@^@^@^@.^@
GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0^@^@^@^@^@^@^@^D^@^@^@^P^@^@^@^E^@^@^@
@^@^@^@^@^@^@^AzR^@^Ax^P^A^[^L^G^H<90>^A^@^@^\^@^@^@^\^@^@^@^@^@^@^@<^@^@^@^@E^N^
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^D^@ñÿ^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
...
^@^@^@^@^@add.c^@main^@_GLOBAL_OFFSET_TABLE_^@printf^@^@^@^@^@^@^@'^@^@^@^@^@^@
^@^D^@^@^@^L^@^@^@üÿÿÿÿÿÿÿ^@^@^@^@^@^@^@^B^@^@^@^B^@^@^@^@^@^@^@^@^@^@^@.symta
a^@.bss^@.rodata^@.comment^@.note.GNU-stack^@.note.gnu.property^@.rela.eh_frame^@
...
@@@
"add.o" [noeol][converted] 3L, 1770C
```

f. **Linking**
This is the final phase in which all the linking of function calls with their definitions are done. It adds some extra code to our program which is required when the program starts and ends. This task can be easily verified by using **$size filename.o** and **$size filename**.



```
kavithakr@LAPTOP-SC9D2AHO:~$ size add.o
   text    data     bss     dec     hex filename
    158       0       0     158      9e add.o
kavithakr@LAPTOP-SC9D2AHO:~$ size add
   text    data     bss     dec     hex filename
   1597     600       8    2205     89d add
kavithakr@LAPTOP-SC9D2AHO:~$
```

2. Write a lexical analyser in Java. The analyser should read the input string (program or program fragments) from a file and determine the lexemes and their corresponding token classes/types using string processing functions. For example, if the input is:

```
if (txn >= 20) rxn = txn * 100;
else rxn = txn - 10;
```

The output should be saved to a file as below:

<IF,if><LPAREN,(><ID,txn><GEQ,>=><NUM,20><RPAREN,)><ID,rxn><ASSIGN,=><ID,txn><TIMES,*><NUM,100><SEMICOLON,;><ELSE,else><ID,rxn><ASSIGN,=><ID,txn>< MINUS,-><NUM,10><SEMICOLON,;>

3. Implement a DFA in java
   a. to recognize the token Identifier(ID)
   b. To recognize the numbers(NUM)

Generate error message if the lexeme is not a valid identifier and number.

Sample Output
Enter a lexeme : print1
Token: <ID, print1>

Enter a lexeme : 12
Token :<NUM,12>

Enter a lexeme: 12kd
Token : Not a valid lexeme