# 21AIE313 Introduction to Modern Compiler Design

## S6 BTech AI

## Lab Sheet 1

**Name:** J Viswaksena **Roll.No:** AM.EN.U4AIE21035

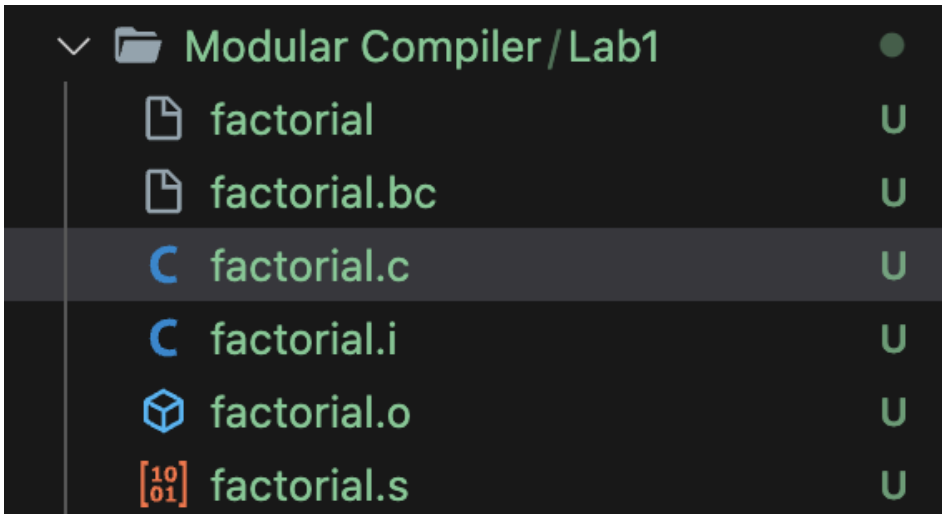
1. Getting started with C programming language processing system - Behind the Scenes

(a) Create a file named factorial.c with the following code.

```c
#include <stdio.h>

#define ENABLE_PRINT 1
#define N 5

int main()
{
    int i, fact = 1;
    for (i = 1; i <= N; i++)
    {
        fact *= i;
    }

#if ENABLE_PRINT
    printf("Factorial of %d is %d\n", N, fact);
#endif

    return 0;
}
```

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

```
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % gcc -Wall -save-temps factorial.c -o factorial
```



**(c) Preprocessing:**This is the first phase through which source code is passed. This phase include:
(i) Removal of Comments
(ii) Expansion of Macros
(iii) Expansion of the included files
(iv) Elimination of white spaces
(v) Conditional compilation
The pre-processed output is stored in the factorial.i(contain pure High level language). Here comment lines are eliminated, #include statement is missing(corresponding files are included into the program), macros are expanded, white spaces are eliminated.

```
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % vi factorial.i
```

```
# 2 "factorial.c" 2



int main()
{
    int i, fact = 1;
    for (i = 1; i <= 5; i++)
    {
        fact *= i;
    }


    printf("Factorial of %d is %d\n", 5, fact);


    return 0;
}
```

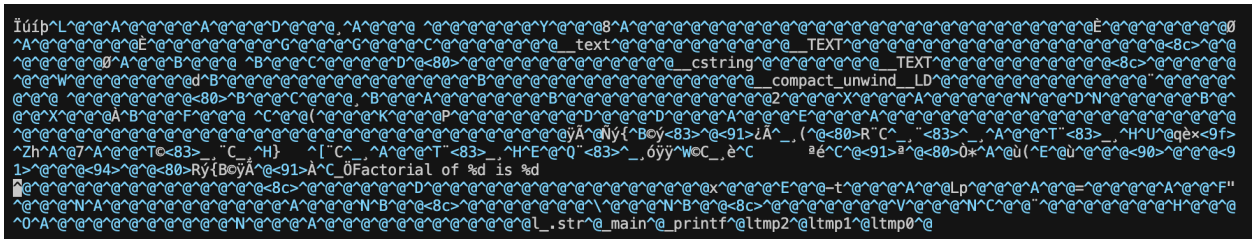
**d. Compiling:** The next step is to compile factorial.i and produce an intermediate compiled output file factorial.s.

```
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % vi factorial.s
```

```
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 14, 0     sdk_version 14, 4
        .globl  _main                           ; -- Begin function main
        .p2align        2
_main:                                  ; @main
        .cfi_startproc
; %bb.0:
        sub     sp, sp, #48
        .cfi_def_cfa_offset 48
        stp     x29, x30, [sp, #32]             ; 16-byte Folded Spill
        add     x29, sp, #32
        .cfi_def_cfa w29, 16
        .cfi_offset w30, -8
        .cfi_offset w29, -16
        stur    wzr, [x29, #-4]
        mov     w8, #1
        stur    w8, [x29, #-12]
        stur    w8, [x29, #-8]
        b       LBB0_1
LBB0_1:                                 ; =>This Inner Loop Header: Depth=1
        ldur    w8, [x29, #-8]
        subs    w8, w8, #5
        cset    w8, gt
        tbnz    w8, #0, LBB0_4
        b       LBB0_2
LBB0_2:                                 ;   in Loop: Header=BB0_1 Depth=1
        ldur    w9, [x29, #-8]
        ldur    w8, [x29, #-12]
        mul     w8, w8, w9
        stur    w8, [x29, #-12]
        b       LBB0_3
LBB0_3:                                 ;   in Loop: Header=BB0_1 Depth=1
        ldur    w8, [x29, #-8]
        add     w8, w8, #1
        b       LBB0_1
LBB0_4:
        ldur    w9, [x29, #-12]
                                        ; implicit-def: $x8
        mov     x8, x9
        mov     x9, sp
        mov     x10, #5
        str     x10, [x9]
        str     x8, [x9, #8]
        adrp    x0, l_.str@PAGE
        add     x0, x0, l_.str@PAGEOFF
        bl      _printf
        mov     w0, #0
        ldp     x29, x30, [sp, #32]             ; 16-byte Folded Reload
        add     sp, sp, #48
        ret
        .cfi_endproc
                                        ; -- End function
        .section        __TEXT,__cstring,cstring_literals
l_.str:                                 ; @.str
        .asciz  "Factorial of %d is %d\n"

.subsections_via_symbols
        .section        __TEXT,__text,regular,pure_instructions
```

**e. Assembling:** The factorial.s is taken as input and turned into factorial.o by assembler. This file contain machine level instructions. At this phase, the assembly code is converted into machine code as shown below.

```
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % vi factorial.o
```



**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.

```
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % size factorial.o
  __TEXT   __DATA   __OBJC  others  dec     hex
  163      0        0       32      195     c3
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % size factorial
  __TEXT   __DATA   __OBJC  others  dec     hex
  16384    0        0       4295000064      4295016448      10000c000
```

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.

Input.txt

```
1    if (i <= 20)
2        i= i * 20;
3    else i = i+10;
```

Java Code:

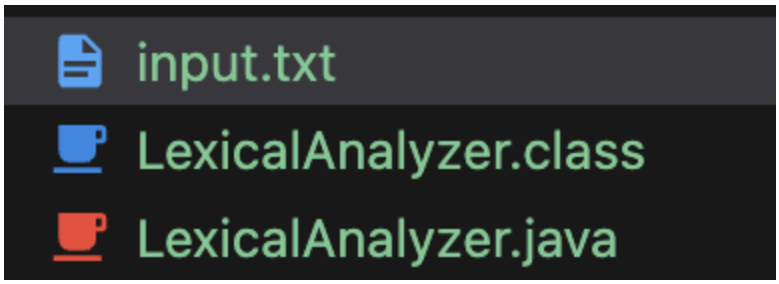
```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class LexicalAnalyzer {

   // Define token types
```

```
    private static final Map<String, String> tokenTypes = new HashMap<>();
    static {
        tokenTypes.put("if", "IF");
        tokenTypes.put("else", "ELSE");
        tokenTypes.put("(", "LPAREN");
        tokenTypes.put(")", "RPAREN");
        tokenTypes.put("<=", "LEQ");
        tokenTypes.put("=", "ASSIGN");
        tokenTypes.put("*", "MUL");
        tokenTypes.put("+", "ADD");
        tokenTypes.put(";", "SEMICOLON");
        // Add more token types as needed
    }

    public static void main(String[] args) {
        String inputFile = "./input.txt";
        String outputFile = "output.txt";
        analyzeLexemes(inputFile, outputFile);
    }

    public static void analyzeLexemes(String inputFile, String outputFile) {
        try (BufferedReader reader = new BufferedReader(new FileReader(inputFile));
             BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
            String line;
            while ((line = reader.readLine()) != null) {
                line = line.trim();
                if (!line.isEmpty()) {
                    String[] tokens =
line.split("\\s+|(?<=<=)|(?=<=)|(?<=\\+)|(?=\\+)|(?<=\\*)|(?=\\*)|(?<=\\()|(?=\\))|(?<
=;)|(?=;)|(?<==)");
                    for (String token : tokens) {
                        if (tokenTypes.containsKey(token)) {
                            writer.write("<" + tokenTypes.get(token) + "," + token +
">");
                        } else if (token.matches("[a-zA-Z][a-zA-Z0-9]*")) {
                            writer.write("<ID," + token + ">");
                        } else if (token.matches("\\d+")) {
                            writer.write("<NUM," + token + ">");
                        }
                    }
                }
            }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

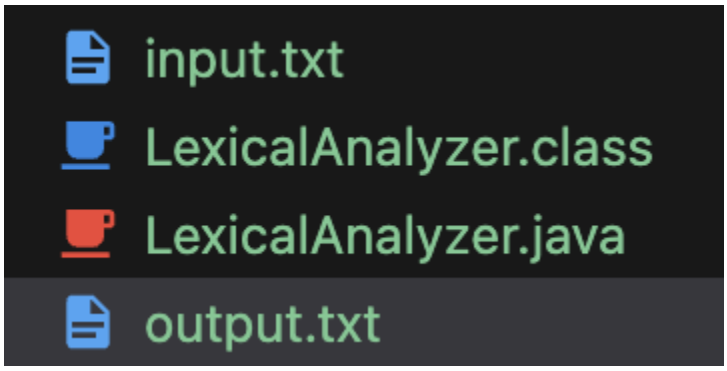
Running Javacompiler:

```
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % javac LexicalAnalyzer.java
```




Run Java:

```
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % java LexicalAnalyzer
```




```
1   <IF,if><LPAREN,(><ID,i><LEQ,<=><NUM,20><RPAREN,)><ID,i><MUL,*><NUM,20><SEMICOLON,;><ELSE,else><ID,i><ASSIGN,=><ID,i><ADD,+><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.

Java Code:

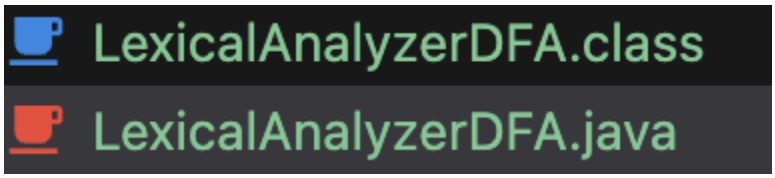
```
import java.util.Scanner;

public class LexicalAnalyzerDFA {

    // DFA states
    private static final int STATE_START = 0;
```

```
    private static final int STATE_ID = 1;
    private static final int STATE_NUM = 2;
    private static final int STATE_ERROR = -1;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print("Enter a lexeme: ");
            String lexeme = scanner.nextLine().trim();
            if (lexeme.isEmpty()) {
                break;
            }
            String token = getToken(lexeme);
            System.out.println("Token: " + token);
        }
        scanner.close();
    }

    public static String getToken(String lexeme) {
        int currentState = STATE_START;
        for (char c : lexeme.toCharArray()) {
            switch (currentState) {
                case STATE_START:
                    if (Character.isLetter(c) || c == '_') {
                        currentState = STATE_ID;
                    } else if (Character.isDigit(c)) {
                        currentState = STATE_NUM;
                    } else {
                        return "Not a valid lexeme";
                    }
                    break;
                case STATE_ID:
                    if (Character.isLetterOrDigit(c) || c == '_') {
                        currentState = STATE_ID;
                    } else {
                        return "Not a valid lexeme";
                    }
                    break;
                case STATE_NUM:
                    if (Character.isDigit(c)) {
                        currentState = STATE_NUM;
                    } else {
```

```
                        return "Not a valid lexeme";
                    }
                    break;
                default:
                    return "Not a valid lexeme";
            }
        }

        // Final state check
        switch (currentState) {
            case STATE_ID:
                return "<ID, " + lexeme + ">";
            case STATE_NUM:
                return "<NUM, " + lexeme + ">";
            default:
                return "Not a valid lexeme";
        }
    }
}
```

Running Javacompiler:

```
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % javac LexicalAnalyzerDFA.java
```




Run Java:

```
(base) viswaksenajayam@Viswaksenas-MacBook-Air Lab1 % java LexicalAnalyzerDFA
Enter a lexeme: _ab1
Token: <ID, _ab1>
Enter a lexeme: 15
Token: <NUM, 15>
Enter a lexeme: 3rd
Token: Not a valid lexeme
Enter a lexeme: my_variable
Token: <ID, my_variable>
Enter a lexeme: 123_abc
Token: Not a valid lexeme
Enter a lexeme: test123
Token: <ID, test123>
Enter a lexeme: a
Token: <ID, a>
Enter a lexeme: _
Token: <ID, _>
Enter a lexeme: a123_b456
Token: <ID, a123_b456>
Enter a lexeme: ^Z
```